6

Objektorientierung 2

Dieses Kapitel behandelt die folgenden Themen:

- Wie funktioniert eine String-Klasse?
- Unterschied zwischen objekt- und klassenbezogenen Daten und Funktionen
- Klassen-Templates
- Template-Metaprogrammierung
- Templates mit variabler Anzahl von Parametern

6.1 Eine String-Klasse

Die Verarbeitung der Zeichenketten oder C-Strings in C++, wie sie in Abschnitt 5.3 beschrieben sind, ist ziemlich mühselig und fehleranfällig, weil stets der Speicherplatz genau beachtet werden muss. Die Benutzung der in `<cstring>` deklarierten Standardfunktionen wie `strcpy(ziel, quelle)` aus der C-Bibliothek setzt voraus, dass für `ziel` genügend Speicher vorhanden ist. Auch die Standardeingabe ist gefährlich (Seite 194). Deshalb bietet die C++-Standardbibliothek eine String-Klasse an, die die Verarbeitung von Strings stark vereinfacht und sicherer macht, indem die Speicherverwaltung und die Prüfungen auf das terminierende Nullbyte '\0' in der Klasse gekapselt werden. Die Benutzung der Standardklasse `string` ist von Seite 86 bekannt.

In diesem Abschnitt geht es darum zu zeigen, wie eine String-Klasse aufgebaut sein kann. Es wird die Klasse `MeinString` vorgestellt, die nur einen kleinen Teil der im Standard

vorgesehenen Funktionen liefert und vergleichsweise einfach aufgebaut ist. Die Klasse `MeinString` zeigt insbesondere die Notwendigkeit von Kopierkonstruktor, Destruktor und Zuweisungsoperator. Die Namen der Methoden entsprechen den Namen der Klasse `string` des C++-Standards, die Sie später ohnehin benutzen werden. Um eine Verwechslung zu vermeiden, wird hier für die Klasse selbst ein anderer Name gewählt. Um kompatibel zum bekannten Stringverhalten zu bleiben, wird in der Klasse vorausgesetzt, dass jede Zeichenkette mit einem Nullbyte abgeschlossen wird. Das Überladen von Operatoren wird erst in Kapitel 9 behandlet. Aus diesem Grund wird der Zuweisungsoperator »verboten«, indem er als `private` deklariert wird.

Listing 6.1: Klasse MeinString, erste Version

```
// /cppbuch/k6/meinstring/meinstring.h
// einfache String-Klasse. Erste Version
#ifndef MEINSTRING_H
#define MEINSTRING_H
#include<cstddef>                  // size_t
#include<iostream>

class MeinString {
  public:
   MeinString(const char* str = "");   // allg. Konstruktor
   MeinString(const MeinString&);      // Kopierkonstruktor
   ~MeinString();                      // Destruktor
   MeinString& assign(MeinString);     // Zuweisung
   MeinString& assign(const char *);   // Zuweisung eines char*
   // Zur Begründung des Rückgabetyps MeinString& statt void siehe Punkt 6
   // der Faustregeln zur Methodenkonstruktion auf Seite 559.
   const char& at(size_t position) const; // Zeichen holen
   char& at(size_t position);          // Zeichen holen
            // Wegen des Rückgabetyps Referenz kann das Zeichen geändert werden.
   size_t length() const { return len;} // Anzahl der Zeichen
   const char* c_str() const { return start;} // C-String zurückgeben
  private:
   size_t len;                            // Länge
   char *start;                           // Zeiger auf den Anfang
   void operator=(const MeinString&); // noch nicht behandelten Operator verbieten
};

void anzeigen(std::ostream&, const MeinString&); // siehe Text
#endif
```

Die Klasse `MeinString` hat nur zwei private Daten: `start` ist ein Zeiger auf den Beginn der Zeichenkette. Die Zeichenkette kann verschiedene Längen haben, daher wird dem Zeiger mit `new` der jeweils benötigte Platz zugewiesen. Daraus folgt, dass der Destruktor die Aufgabe hat, diesen Speicherplatz wieder freizugeben. `start` ist mit der Methode `c_str()` öffentlich lesbar. Das ist notwendig, um ein `MeinString`-Objekt als C-String an eine Funktion übergeben zu können. Ein Beispiel ist die `fstream`-Funktion `open()`, die einen Dateinamen des Typs `const char*` verlangt. `const` im Rückgabetyp verhindert, dass der Aufrufer der Funktion verändernden Zugriff erhält.

`len` enthält die aktuelle Länge des Strings. Die Abfrage mit `length()` ist schneller als der bekannte Aufruf von `strlen()` aus *string.h*, nicht durch die Realisierung als `inline`-Funktion, sondern weil der String zur Längenermittlung nicht jedesmal erneut durchlaufen wird. Abbildung 6.1 zeigt ein `MeinString`-Objekt namens `einMstring` als UML-Diagramm. Das Minuszeichen zeigt an, dass die Attribute privat sind. Weil es zunächst nur um die internen Daten geht, wurden die Methoden in der Abbildung weggelassen.

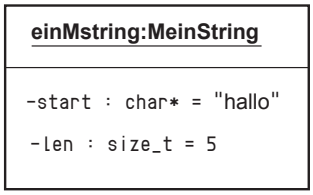

Abbildung 6.1: Ein Objekt der Klasse `MeinString` namens `einMstring`

Implementierung der Klasse MeinString

Eine mögliche Implementation zeigt die Datei *cppbuch/k6/meinstring/meinstring.cpp*. Innerhalb der Klasse `MeinString` werden die vorgefertigten Funktionen `strcpy()` und `strlen()` des Headers `<cstring>` genutzt.

Der allgemeine Konstruktor erzeugt aus einem klassischen C-String ein Objekt der Klasse `MeinString`, damit Anweisungen wie

```
MeinString meinStringObjekt("beliebige Zeichenkette");
```

möglich sind. Dazu wird zunächst die Länge des C-Strings ermittelt und ausreichend Platz bereitgestellt, auch für das Nullbyte. Anschließend werden alle Zeichen einschließlich '\0' kopiert:

```
MeinString::MeinString(const char *s)    // allg. Konstruktor
: len(strlen(s)), start(new char[len+1]) {
   strcpy(start, s);
}
```

Im zweiten Teil der Initialisierungsliste wird auf das Attribut `len` Bezug genommen, das im ersten Teil initialisiert wird. Entsprechend Punkt 1 auf Seite 156 spielt aber nur die Reihenfolge der Attribute im `private`-Teil der Klasse eine Rolle, nicht die Reihenfolge in der Initialisierungsliste! Wenn die Reihenfolge umgekehrt wäre, nämlich

```
// falsch, falls sich die Initialisierung von start auf len bezieht
char *start;                    // Zeiger auf den Anfang
size_t len;                     // Länge
```

würde der Konstruktor fehlerhaft arbeiten: Zuerst würde `start` initialisiert, wobei eine undefinierte Menge an Speicher zugewiesen würde (`len` ist noch unbekannt!). Danach erst würde `len` initialisiert, falls nicht das Programm mit einer Fehlermeldung schon abgebrochen wurde.

Durch den Vorgabewert "" in der Deklaration auf Seite 234 wird kein Standardkonstruktor benötigt. Wenn ein leerer String etwa mit `MeinString leererString;` erzeugt wird, ist die Wirkung genau dieselbe, die der folgende Konstruktors erzielen würde:

```
MeinString::MeinString()      // (hier fiktiver) Standardkonstruktor
: len(0), start(new char[1]) { // Platz für '\0'
    *start = '\0';             // leerer String
}
```

Der Kopierkonstruktor kann die Länge des Objekts, mit dem initialisiert wird, direkt übernehmen:

```
MeinString::MeinString(const MeinString& m) // Kopierkonstruktor
: len(m.len), start(new char[len+1]) {
    strcpy(start, m.start);
}
```

Hier ist deutlich zu sehen, dass ein eigener Kopierkonstruktor für die Klasse notwendig ist. Der bei Abwesenheit dieses Konstruktors durch das System erzeugte Kopierkonstruktor würde nur die Länge und den Zeiger kopieren, nicht aber ein echtes Duplikat erzeugen! In diesen und ähnlichen Fällen muss stets ein besonderer Kopierkonstruktor gebildet werden, zum Beispiel, wenn ein Objekt wie hier Zeiger enthält, weil vom Kopierkonstruktor des Systems zwar die Zeiger kopiert werden (»flache« Kopie (englisch *shallow copy*)), nicht aber die Datenbereiche (wie Arrays, Strings oder andere Objekte), auf die die Zeiger verweisen (»tiefe« Kopie (englisch *deep copy*)). Der Unterschied wird in Abbildung 6.2 sichtbar. Der obige Kopierkonstruktor erzeugt also eine »tiefe Kopie«. Vergleichen Sie dazu den Abschnitt »Kopieren von Strings« auf Seite 197. Dieselbe Problematik tritt natürlich auch bei Zuweisungen auf (siehe `assign()`-Methode unten).

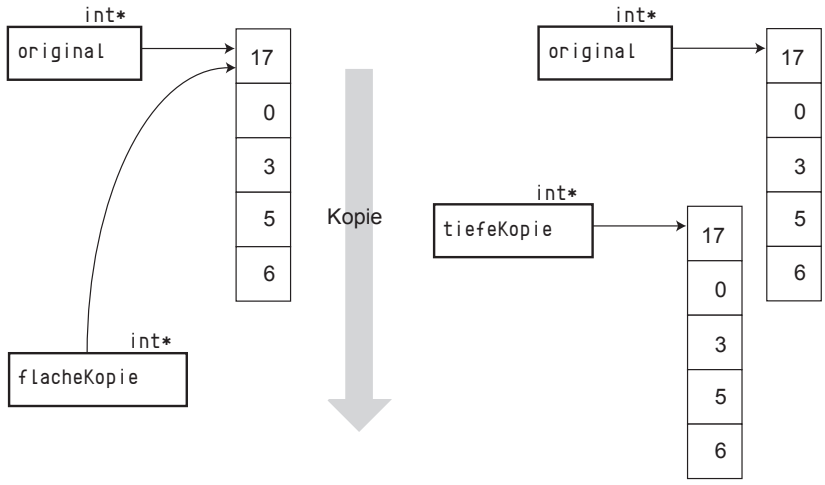

Abbildung 6.2: »Flache« und »tiefe« Kopie eines Objekts

Es gibt verschiedene Möglichkeiten für die Bedeutung einer Kopie: Es ist ein Unterschied, ob nach einer Zuweisung `a = b;` das Objekt `a` einen Zeiger enthält, der auf *den-*

selben Speicherbereich wie der entsprechende Zeiger des Objekts `b` zeigt (Referenzsemantik, Abbildung 6.2 links), oder ob der Zeiger von `a` auf ein neu erzeugtes *Duplikat* des Speicherbereichs verweist (Wertsemantik, siehe Abbildung 6.2 rechts). Der systemerzeugte Destruktor würde nur `start` und `len` vom Stack entfernen, was keinesfalls ausreichend ist. Der folgende Destruktor gibt den durch `new[]` beschafften Platz frei:

```
MeinString::~MeinString() {              // Destruktor
    delete [] start;
}
```

Die überladenen `assign()`-Methoden erlauben Zuweisungen von `MeinString`-Objekten oder C-Strings, zum Beispiel

```
MeinString nochEinString("hallo");  // allg. Konstruktor
einString.assign(nochEinString);    // Zuweisung
einString.assign("neuer Text");     // Zuweisung
einString.assign(einString);        // Zuweisung auf sich selbst?
einString = nochEinString;// (noch) nicht erlaubt, siehe Text:
```

Die Zuweisung mit dem Zuweisungsoperator `=` ist hier nicht gestattet, weil der systemgenerierte Zuweisungsoperator kein echtes Duplikat erzeugt. Wie Sie eigene Zuweisungsoperatoren schreiben können, wird unten in Abschnitt 9.2.2 erläutert. Die Methoden zur Zuweisung müssen in ausreichender Menge neuen Speicherplatz beschaffen, anschließend die Daten kopieren und dann den vorher benutzten Speicherplatz freigeben.

Bei der Zuweisung könnte man daran denken, dass eine Zuweisung auf sich selbst zwar nicht sinnvoll, aber syntaktisch möglich ist. Eine Kopie durchzuführen wäre vertane Zeit. Dieser Fall ist aber so selten, dass auf eine Optimierung hier verzichtet wird. Die Funktion `swap()` vertauscht zwei String-Objekte. Sie wird für die Zuweisung benötigt und benutzt die Bibliotheksfunktion `std::swap()`.

```
void MeinString::swap(MeinString& mString) {
   std::swap(len, mString.len);          // Bibliotheksfunktion
   std::swap(start, mString.start);
}
```

Um zu verhindern, dass ein `MeinString`-Objekt in einen nichtkonsistenten Zustand gerät, falls etwas mit der Speicherbeschaffung schief gehen sollte, wird der alte Speicherplatz erst nach vollendeter Kopie freigegeben. Das geht am einfachsten, indem eine Kopie erzeugt wird, denn nur dabei wird Speicher beschafft. Dann wird das aktuelle Objekt mit der Kopie vertauscht. Schließlich wird der alte Speicherplatz durch den Destruktor der Kopie freigegeben, entsprechend dem Verfahren des Abschnitts 5.7.3.

```
MeinString& MeinString::assign(MeinString m) { // Zuweisung eines MeinString
   swap(m);                                    // Platz mit Kopie m tauschen
   return *this;
}   // Der Destruktor der Kopie m wird hier aktiv.
```

Die Zuweisung von C-Strings kann von der `assign(MeinString)`-Funktion profitieren. Die Methode wird daher noch kürzer:

```
MeinString& MeinString::assign(const char *s) { // Zuweisung eines char*
   return assign(MeinString(s));
}
```

Es wird ein temporäres `MeinString`-Objekt erzeugt, das der ersten `assign()`-Funktion übergeben wird. Um auf einzelne Zeichen des Strings lesend zuzugreifen, gibt es die Methode `at()`, der die Position eines Zeichens innerhalb der Zeichenkette übergeben wird. `at()` ist für konstante `MeinString`-Objekte überladen. Der String wird durch das Lesen nicht verändert (`const`):

```
char& MeinString::at(size_t position) { // Zeichen per Referenz holen
    assert(position < len); // Nullbyte lesen ist nicht erlaubt
    return start[position];
}

const char& MeinString::at(size_t position) const { // Zeichen holen
    assert(position < len); // Nullbyte lesen ist nicht erlaubt
    return start[position];
}
```

Die Rückgabe per nicht-konstanter Referenz erlaubt die Änderung eines Zeichens im String. Erinnern wir uns: Erstens ist eine Referenz nichts anderes als ein anderer Name für etwas, in diesem Fall ein Zeichen innerhalb des Strings, und zweitens kann man sich das Ergebnis eines Funktionsaufrufs anstelle des Aufrufs eingesetzt denken.

```
einString.at(0) = 'X';
```

heißt also, dass die gesamte linke Seite der Zuweisung nichts anderes darstellt als das Zeichen mit der Nummer 0 im String! Die Wirkung dieser zunächst ungewohnten Schreibweise ist dieselbe, als wenn wir `einString.start[0] = 'X';` geschrieben hätten, wobei wir hier ignorieren, dass `start` privat ist. Entsprechend der C-Konvention beginnt die Zählung bei 0. Mit `at()` kann bei der Abfrage in Bedingungen *nicht* mehr auf das (interne) abschließende Nullbyte eines Strings zugegriffen werden. Ersatzweise bietet sich die Abfrage auf die Länge des Strings an:

```
// at()-Anwendung zur Anzeige eines MeinString-Objekts
int i = -1;
while(++i < einString.length()) {
    cout << einString.at(i);
}
```

Eine direkte Ausgabe eines `MeinString`-Objekts, etwa der Art `cout << einString;`, ist nicht möglich, weil der Ausgabeoperator `<<` für `MeinString`-Objekte noch nicht definiert ist. Das kann geändert werden, wie die Übung 9.6 auf Seite 331 zeigt. Hier kann die Ausgabe auf zwei Arten realisiert werden:

```
cout << einString.c_str();
anzeigen(cout, einString);
```

Letzteres ist der Aufruf der oben deklarierte Funktion `anzeigen()`, die wie folgt definiert ist:

```
void anzeigen(std::ostream& os, const MeinString& m) {
   os << m.c_str();
}
```

6.1.1 Optimierung der Klasse MeinString

Bei verschiedenen Operationen oben wird neuer Speicher beschafft und der alte freigegeben. Dies ist nicht notwendig, wenn der vorhandene Speicher genug Platz hat. Dann kann direkt in den alten Speicher kopiert werden. Weil in diesem Fall die Speicherbeschaffung mit `new` entfällt, ist keine Exception zu erwarten, sodass dieser Weg auch exception-sicher ist. Um dies zu implementieren, wird am besten ein Attribut `cap` (für Capacity) eingeführt, das den gesamten Speicherplatz (ohne das Nullbyte) für das `MeinString`-Objekt angibt. Die Methode `capacity()` gibt den insgesamt den für Zeichen zur Verfügung stehenden Platz zurück. Das Attribut `len`, das die Anzahl der tatsächlich gespeicherten Zeichen angibt, ist immer kleiner oder gleich `cap`.

Eine Methode `reserve(size_t groesse)` kann den entsprechenden Speicherplatz reservieren (bzw. nichts tun, wenn der aktuelle Platz schon gleich oder größer als `groesse` ist). Da `reserve()` dafür sorgen muss, dass der Inhalt des Objekts erhalten bleibt, ist ggf. ein Umkopieren notwendig. Falls Speicher ohne Umkopieren reserviert werden soll, sinnvoll zum Beispiel bei der Methode `assign()`, kann dies von einer privaten Hilfsfunktion `reserve_only()` erledigt werden. Falls keine Operationen mehr mit einem `MeinString`-Objekt zu erwarten sind, reduziert die Methode `void shrink_to_fit()` den Speicherplatz auf das Notwendige. In der Header-Datei sind die folgenden Änderungen vorzunehmen:

```
// Auszug aus cppbuch/k6/meinstringOpt/meinstring.h
public:
   size_t capacity() const { return cap;} // Kapazität zurückgeben
   void reserve(size_t bytes);          // Platz reservieren mit Erhalt des Inhalts
   void shrink_to_fit();                // Platz minimieren
private:
   size_t cap;                          // Kapazität (direkt nach len eintragen)
   void reserve_only(size_t bytes);     // nur Platz reservieren
```

In der Implementierungsdatei sind nicht nur Methoden betroffen, sondern auch Konstruktoren. Nur die Änderungen sind im Folgenden angegeben:

```
MeinString::MeinString(const char *s)          // allg. Konstruktor
   : len(strlen(s)), cap(len), start(new char[cap+1]) {
   strcpy(start, s);
}

MeinString::MeinString(const MeinString& m)   // Kopierkonstruktor
   : len(m.len), cap(len), start(new char[cap+1]) {
   strcpy(start, m.start);
}

void MeinString::reserve(size_t groesse) {
   if(groesse > cap) {              // nichts tun, wenn der Platz reicht
      char *temp = new char[groesse+1]; // neuen Platz beschaffen, Inhalt erhalten
      strcpy(temp, start);          // umkopieren
      delete [] start;              // alten Platz freigeben
      start = temp;                 // Verwaltungsinformation aktualisieren
      cap = groesse;                // Verwaltungsinformation aktualisieren
   }
}
```

```
void MeinString::reserve_only(size_t groesse) {
   if(groesse > cap) {              // nichts tun, wenn der Platz reicht
      char *temp = new char[groesse+1]; // nur neuen Platz beschaffen
      delete [] start;              // alten Platz freigeben
      start = temp;                 // Verwaltungsinformation aktualisieren
      cap = groesse;                // Verwaltungsinformation aktualisieren
   }
}

void MeinString::shrink_to_fit() {
   if(cap > len) {          // nichts tun, wenn kein Speicher eingespart wird
      char *temp = new char[len+1]; // neuen Platz beschaffen
      strcpy(temp, start);          // umkopieren
      delete [] start;              // alten Platz freigeben
      start = temp;                 // Verwaltungsinformation aktualisieren
      cap = len;                    // Verwaltungsinformation aktualisieren
   }
}

MeinString& MeinString::assign(const MeinString& m) { // Zuweisung eines MeinString
   reserve_only(m.len);
   strcpy(start, m.start);
   len = m.len;
   return *this;
}

MeinString& MeinString::assign(const char *s) { // Zuweisung eines char*
   size_t temp = strlen(s);
   reserve_only(temp);
   strcpy(start, s);
   len = temp;
   return *this;
}
```

Die `swap()`-Methode aus dem vorherigen Abschnitt funktioniert hier nicht, weil dort bei der Kopie stets Speicherplatz angefordert wird. Hier wird Speicherplatz nur dann angefordert, wenn der alte Platz nicht reicht.

Die `assign()`-Methoden benutzen `reserve_only()`. Bitte beachten Sie, dass das Attribut `len` für die Anzahl der Zeichen erst *nach* dem Aufruf von `reserve_only()` aktualisiert wird. Der Grund: Falls etwas bei der Speicherbeschaffung schief gehen sollte, bliebe das `MeinString`-Objekt in einem unveränderten Zustand. Der Aufruf der Methoden ist daher *exception safe*.

Mehr zum Thema Exception-Safety lesen Sie in Abschnitt 20.3.

Übung

6.1 Ergänzen Sie die Klasse `MeinString` um eine Methode `insert(size_t pos, const MeinString& m)`, die den Inhalt von `m` vor `pos` einfügt, also am Anfang, wenn `pos` = 0 ist.

6.1.2 friend-Funktionen

Die oben gezeigte Schleife zur Anzeige eines `MeinString`-Objekts könnte in eine Funktion `anzeigen()` gepackt werden, für die hier drei verschiedene Möglichkeiten beschrieben werden sollen:

1. Elementfunktion `void anzeigen(std::ostream &os) const;`
   Die Methode müsste in der Klasse deklariert werden. Der Parameter `os` vom Typ `ostream&` erlaubt die Ausgabe nicht nur auf der Standardausgabe, sondern auch auf dem Fehlerkanal oder in eine Datei:

   ```
   einString.anzeigen(std::cout);
   einString.anzeigen(std::cerr);
   ```

   Innerhalb der Methode kann direkt auf `start` zugegriffen werden, weil `start` vom Grunddatentyp `char*` ist und der Ausgabeoperator `<<` in der Klasse `ostream` für alle Grunddatentypen definiert ist:

   ```
   void MeinString::anzeigen(std::ostream& os) const { // Version 1
      os << start;
   }
   ```

2. Globale Funktion `void anzeigen(std::ostream&, const MeinString&);`
   Diese Funktion braucht dank `c_str()` nicht direkt auf die privaten Daten zuzugreifen. Das String-Objekt wird jetzt als Parameter übergeben.

   ```
   // Aufruf:
   anzeigen(std::cout, einString);
   // Implememention (Version 2):
   void anzeigen(std::ostream& os, const MeinString& m) {
          os << m.c_str();
   }
   ```

3. `friend`-Funktion
   Falls keine der beiden Möglichkeiten benutzt werden soll (wofür es hier keine guten Gründe gibt), kann ein Mittelding aus den beiden vorangegangenen Möglichkeiten gebildet werden, nämlich eine `friend`-Funktion. Eine als `friend` deklarierte Funktion ist *keine* Methode der Klasse, hat aber das Recht, auf deren private Daten zuzugreifen. Dies muss der Klasse natürlich bekannt sein. Das Schlüsselwort `friend` in der Deklaration sorgt dafür. Im Programmcode wird auf private Daten zugegriffen:

   ```
   // Version 3
   // Deklaration innerhalb der Klasse (meinstring.h)
   friend void anzeigen(std::ostream& os, const MeinString& m);
   // Definition (meinstring.cpp)
   void anzeigen(std::ostream& os, const MeinString& m) {
      os << m.start;
   }
   ```

Der Aufruf der Funktion wird im dritten Fall genauso wie im zweiten Fall geschrieben. Weil die Datenkapselung durch `friend`-Deklarationen durchlöchert wird, sollte man sparsam damit umgehen. Nicht nur fremden Funktionen, sondern auch anderen Klassen kann der Zugriff auf private Daten gestattet werden, wenn dies wegen des engen Zusammenwirkens der Klassen notwendig ist.

6.2 Klassenspezifische Daten und Funktionen

Klassenspezifische Daten sind Daten, die *nur einmal* für *alle* Objekte einer Klasse existieren. Sie sind also nicht an einzelne Objekte, sondern an alle Objekte einer Klasse gleichzeitig gebunden. Dies können Verwaltungsdaten wie die Anzahl der Objekte sein oder auch Bezugsdaten, die für alle Objekte gelten, wie ein gemeinsamer Koordinatenursprung für grafische Objekte. Diese von mehreren Objekten einer Klasse *gemeinsam* benutzbaren Daten müssen nicht global sein. Sie können genausogut gekapselt werden wie andere Daten eines Objekts. Sie sind dann *innerhalb einer Klasse* und *nur für Objekte dieser Klasse* gleichermaßen zugreifbar. Diese Daten sind `static` – eine weitere Bedeutung dieses Schlüsselworts (die bisher bekannten sind »Variable für Funktionen mit Gedächtnis« und »nur in dieser Datei gültige Deklaration«). Bei der Definition und Initialisierung wird für eine beliebige Anzahl von Objekten einer Klasse nur *ein* Speicherplatz pro `static`-Element angelegt, auf den von *allen* Objekten dieser Klasse zugegriffen werden kann. Klassenspezifische Funktionen führen Aufgaben aus, die *an eine Klasse, nicht aber an ein Objekt gebunden sind*. Sie können zum Beispiel mit den `static`-Daten arbeiten. Auch Konstruktoren sind klassenspezifische Funktionen, weil das zu konstruierende Objekt beim Aufruf noch nicht existiert. Das Beispiel demonstriert sowohl klassenspezifische Daten als auch klassenspezifische Funktionen.

Listing 6.2: Klasse `NummeriertesObjekt`, erste Version

```
// cppbuch/k6/numobj/numobj.h
#ifndef NUMOBJ_H
#define NUMOBJ_H

class NummeriertesObjekt { // noch nicht vollständig! (siehe Text)
    public:
      NummeriertesObjekt();
      NummeriertesObjekt(const NummeriertesObjekt&);
      ~NummeriertesObjekt();
      unsigned long seriennummer() const {
         return serienNr;
      }
      static int anzahl() {
        return anz;
      }
      static bool testmodus;
    private:
      static int anz;    // int statt unsigned (s. Text)
      static unsigned long maxNummer;
      const unsigned long serienNr;
};
#endif       // Ende von numobj.h
```

Die Klasse `NummeriertesObjekt` hat die Aufgabe, jedem Objekt eine unverwechselbare Seriennummer mitzugeben und über die aktuelle Anzahl aktiver Objekte Buch zu führen.

Die `static`-Funktion `anzahl()` soll die momentane Anzahl der Objekte dieser Klasse zurückgeben und ist daher objektunabhängig. Die Funktion `seriennummer()` bezieht sich im Gegensatz dazu nur auf ein einzelnes Objekt. Die öffentliche Variable `testmodus` dient dazu, während der Laufzeit eines Programms den Testmodus für bestimmte Programmabschnitte aus- oder einzuschalten, um auf der Standardausgabe Entstehen und Vergehen aller Objekte zu dokumentieren.

Implementation

In der hier diskutierten Implementierung (siehe auch *cppbuch/k6/numobj/numobj.cpp*) werden die Namespace-Bezeichner bei der Standardausgabe und `endl` weggelassen, weil sie vorab bekannt gemacht werden:

```
using std::cout; // erfordert #include<iostream>
using std::endl;
```

Zur Initialisierung der `static`-Attribute genügt die Angabe von Typ, Klasse und Variablenname. Man sieht auch daran, dass die Initialisierung nicht an ein einzelnes Objekt gebunden ist. Die Initialisierung ist gleichzeitig die Definition der Variablen, die nur genau einmal im Programm vorhanden sein darf (one definition rule, siehe Seite 127).

```
// Initialisierung und Definition der klassenspezifischen Variablen:
int          NummeriertesObjekt::anz    = 0;
unsigned long NummeriertesObjekt::maxNummer = 0L;
bool         NummeriertesObjekt::testmodus = false;
```

Sie kann nicht in einen Konstruktor verlegt werden, weil sie sonst bei jeder Erzeugung eines Objekts durchgeführt würde. Der Standardkonstruktor initialisiert die objektspezifische Konstante `serienNr` in der Initialisierungsliste und aktualisiert die Anzahl aller Objekte:

```
// Standardkonstruktor
NummeriertesObjekt::NummeriertesObjekt()
: serienNr(++maxNummer) {
   ++anz;
   if(testmodus) {
      if(serienNr == 1) {
         cout << "Start der Objekterzeugung!\n";
      }
      cout << " Objekt Nr. " << serienNr << " erzeugt" << endl;
   }
}
```

Der Kopierkonstruktor hat hier eine besondere Bedeutung. Bisher diente er dazu, ein Duplikat eines Objekts bei der Initialisierung zu erzeugen. Das darf hier nicht sein! Das neu erzeugte Objekt soll nicht die Seriennummer eines anderen erhalten, sondern eine neue bekommen, weil sie sonst nicht eindeutig wäre. Der Kopierkonstruktor muss also dasselbe Verhalten wie der Standardkonstruktor aufweisen, mit Ausnahme des Testmodus, damit der unterschiedliche Aufruf protokolliert wird.

```
// Kopierkonstruktor
NummeriertesObjekt::NummeriertesObjekt( const NummeriertesObjekt &X)
```

```
: serienNr(++maxNummer) {
   ++anz;
   if(testmodus) {
      cout << " Objekt Nr. " << serienNr
           << " mit Nr. "  << X.seriennummer() // bzw. X.serienNr
           << "  initialisiert " << endl;
   }
}
```

Die Klassendeklaration auf Seite 242 ist noch nicht vollständig: Was geschieht bei der Zuweisung eines Objekts? Der systemerzeugte Zuweisungsoperator würde bei einer Zuweisung wie zum Beispiel

```
NummeriertesObjekt NumObjekt1;
NummeriertesObjekt NumObjekt2;
// ... irgendwelcher Programmcode
NumObjekt2 = NumObjekt1;            // ?
```

eine Zuweisung der Elemente von `NumObjekt1` an `NumObjekt2` bewirken. Das einzige Element, das dafür in Frage kommt, ist die private Konstante `serienNr`. Einer Konstanten kann aber nichts zugewiesen werden, sodass ein funktionierender systemerzeugter Zuweisungsoperator nicht erzeugt werden kann. Wenn die Zuweisung überhaupt erlaubt sein soll, darf sie einfach nichts bewirken! Dieses Problem können Sie leicht nach Studium des Kapitels 9 lösen, wo in einer Übungsaufgabe (Seite 330) auf dieses Problem eingegangen wird.

Der Destruktor vermerkt, dass es nun ein Objekt weniger gibt. Er wird am Ende eines Blocks und bei der Löschung dynamischer Objekte durch `delete` aufgerufen. Ein versehentliches zusätzliches `delete` kann vom Destruktor festgestellt werden, wenn nämlich `anz` negativ wird. Aus diesem Grund ist `anz` vom Typ `int` und nicht `unsigned int`. Die Zusicherung am Ende des Destruktors garantiert, dass das Löschen eines »hängenden Zeigers« nur im Testmodus gemeldet und toleriert wird.

```
// Destruktor
NummeriertesObjekt::~NummeriertesObjekt() {
  --anz;
  if (testmodus) {
     cout << " Objekt Nr. "
          << serienNr << " gelöscht" << endl;
     if (anz == 0) {
        cout << "letztes Objekt gelöscht!" << endl;
     }
     if (anz < 0) {
        cout << " FEHLER! zu oft delete aufgerufen!" << endl;
     }
  }
  else {
     assert(anz >= 0);
  }
}
```

Konstruktor und Destruktor dokumentieren Werden und Vergehen der Objekte, sofern der Testmodus eingeschaltet ist. Klassenspezifische Funktionen können auch objektgebunden

aufgerufen werden, wie `main()` unten zeigt. Der klassenbezogene Aufruf einer Funktion oder die klassenbezogene Benennung eines Attributs ist dem objektgebundenen Aufruf vorzuziehen, weil der objektgebundene Aufruf die `static`-Eigenschaft verschleiert:

```
// schlechter Stil:
cout << dasNumObjekt.anzahl();      // objektgebundener Aufruf
cout << zeigerAufNumObjekt->anzahl(); // objektgebundener Aufruf
// richtig:
cout << NummeriertesObjekt::anzahl(); // klassenbezogener Aufruf
```

Wenn die letzte Zeile des Destruktors aufgerufen wird, gibt es mindestens eine `delete`-Anweisung zu viel. Zuwenige `delete`-Anweisungen können dagegen nicht zuverlässig ermittelt werden: Die Überprüfung würde durch einen Überschuss von mit `new` erzeugten und nicht gelöschten Objekten ausgetrickst werden.

6.2.1 Klassenspezifische Konstante

Klassenspezifische Variable müssen außerhalb der Klassendefinition definiert und initialisiert werden. Dies gilt nicht für klassenspezifische Konstanten, für die der Compiler keinen Platz anlegen muss, weil er direkt ihren Wert einsetzen kann. In C++ ist diese Ausnahme jedoch auf integrale und Aufzählungstypen beschränkt:

```
class KlasseMitKonstanten {
   enum RGB {rot = 0x0001, gelb = 0x0002, blau = 0x0004};
   static const unsigned int MAX_ZAHL = 1000;
   // Verwendung zum Beispiel:
   static int cArray[MAX_ZAHL];
   // ..
};
```

Listing 6.3: Klassenmethoden und Daten

```
// cppbuch/k6/numobj/nummain.cpp
// Demonstration von nummerierten Objekten
#include "numobj.h"
#include<iostream>
using namespace std;

int main() {
    // Testmodus für alle Objekte der Klasse einschalten
    NummeriertesObjekt::testmodus = true;
    NummeriertesObjekt dasNumObjekt_X;   // ... wird erzeugt
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
         << dasNumObjekt_X.seriennummer() << endl;

    // Anfang eines neuen Blocks
    {   NummeriertesObjekt dasNumObjekt_Y; // ... wird erzeugt
        cout << NummeriertesObjekt::anzahl()
             << " Objekte aktiv" << endl;
        NummeriertesObjekt *p = new NummeriertesObjekt; // dynamisch erzeugt
        cout << NummeriertesObjekt::anzahl() << " Objekte aktiv" << endl;
        delete p;                // *p wird gelöscht
        cout << NummeriertesObjekt::anzahl() << " Objekte aktiv" << endl;
```

```
        delete p;  // Fehler: ein delete zu viel!
    }              // Blockende: dasNumObjekt_Y wird gelöscht

    cout << " Kopierkonstruktor: " << endl;
    NummerviertesObjekt dasNumObjekt_X1 = dasNumObjekt_X;
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
         << dasNumObjekt_X.seriennummer() << endl;
    cout << "Die Seriennummer von dasNumObjekt_X1 ist: "
         << dasNumObjekt_X1.seriennummer() << endl;
    // Zuweisung wird wegen Konstanz der serienNr vom Compiler verboten
     dasNumObjekt_X1 = dasNumObjekt_X;  // Fehler
}  // dasNumObjekt_X und dasNumObjekt_X1 werden gelöscht
```

Diese Konstanten werden *innerhalb* der Klassendefinition initialisiert. Dabei wird `enum` stets ohne das Schlüsselwort `static` deklariert. In allen anderen Fällen muss es bei klassenspezifischen Objekten und Funktionen angegeben werden.

6.3 Klassen-Templates

Genau wie für Funktionen Templates definiert werden können (siehe Seite 134), sind Templates für Klassen möglich. Sie werden auch »parametrisierte Datentypen« genannt. Das Prinzip soll hier kurz dargestellt werden, indem wir einen Datentyp für einen einfachen Stapel (englisch *stack*), genannt `SimpleStack`, entwerfen.

6.3.1 Ein Stack-Template

Ein Stack hat die Eigenschaft, dass auf ihm Elemente abgelegt und wieder entnommen werden können, wobei die Reihenfolge der Entnahme entgegengesetzt der Ablage ist – wie bei einem Stapel von Tellern, auf den nur von oben zugegriffen wird. Die ab Seite 776 beschriebene Stack-Klasse der C++-Standardbibliothek basiert auf Templates. Um zu zeigen, wie es geht, soll hier ein Stack als *Template* für verschiedene Datentypen konstruiert werden. Die englischen Namen der Methoden sind auch in der deutschen Informatikwelt weit verbreitet und werden deshalb beibehalten. Zunächst setzen wir voraus, dass der Stack maximal 20 Elemente aufnehmen kann. Der dafür benötigte Behälter ist ein C-Array. Die auf Funktions-Templates bezogenen Erläuterungen bezüglich Dateien mit Templates auf Seite 138 gelten ebenso für Klassen-Templates. Prototyp und Definitionen sind in *einer* Datei *simstack1.t* zusammengefasst.

Der Datentyp `T` in der folgenden Template-Klasse steht für einen beliebigen Datentyp als *Platzhalter*. Bei der Definition der Methoden außerhalb der Klasse muss der Datentyp in spitzen Klammern zusätzlich zum Namen der Klasse angegeben werden (`<T>`). Innerhalb der Klassendefinition wird der Typ `T` bei den Prototypen der Methoden vorausgesetzt, wenn er nicht angegeben ist. Bei der Benutzung in einem Programm wird ein konkreter Datentyp angegeben, der nicht nur eine Klasse, sondern auch ein Grunddatentyp sein kann. Der Compiler erzeugt damit Objekte dieses Datentyps nach dem Vorbild des Tem-

plates. Die Anwendung des Klassen-Templates zeigt das Beispiel auf Seite 247, in dem zwei Stacks unterschiedlichen Datentyps mit dem Template `SimpleStack` erzeugt werden.

Listing 6.4: Template-Klasse für einen einfachen Stack

```
// cppbuch/k6/stack/simstack1.t      ein einfaches Stack-Template
#ifndef SIMSTACK1_T
#define SIMSTACK1_T
#include<cassert>

template<typename T>
class SimpleStack {
  public:
    static const unsigned int MAX_SIZE = 20; // siehe Text
    SimpleStack() : anzahl(0){}
    bool empty() const { return anzahl == 0;}
    bool full() const { return anzahl == MAX_SIZE;}
    unsigned int size() const { return anzahl;}
    void clear() { anzahl = 0;}     // Stack leeren
    const T& top() const;           // letztes Element sehen
    void pop();                     // Element entfernen
      // Vorbedingung für top und pop: Stack ist nicht leer
    void push(const T& x);          // x auf den Stack legen
      // Vorbedingung für push: Stack ist nicht voll
  private:
    unsigned int anzahl;
    T array[MAX_SIZE];          // Behälter für Elemente
};

// noch fehlende Methoden-Implementierungen
template<typename T>
const T& SimpleStack<T>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template<typename T>
void SimpleStack<T>::pop() {
    assert(!empty());
    --anzahl;
}

template<typename T>
void SimpleStack<T>::push(const T& x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif     // SIMSTACK1_T
```

Listing 6.5: Anwendung des Stack-Templates

```
// cppbuch/k6/stack/simmain1.cpp
// Anwendungsbeispiele für Stack-Template
#include<iostream>
```

```
#include"simstack1.t"
using namespace std;

int main() {
    SimpleStack<int> einIntStack; // ein Stack für int-Zahlen
    int i = 100;
    while(!einIntStack.full()) {
          einIntStack.push(i++); // Stack füllen
    }
    cout << "Anzahl : " << einIntStack.size() << endl;
    // Stack-Methoden aufrufen
    cout << "oberstes Element: " << einIntStack.top() << endl;
    cout << "alle Elemente entnehmen und anzeigen: " << endl;
    while(!einIntStack.empty()) {
        i = einIntStack.top();
        einIntStack.pop();
        cout << i << '\t';
    }
    cout << endl;
    SimpleStack<double> einDoubleStack; // ein Stack für double-Zahlen
    // Stack mit (beliebigen) Werten füllen
    double d = 1.00234;
    while(!einDoubleStack.full()) {
         d = 1.1 * d;
         einDoubleStack.push(d);
         cout << einDoubleStack.top() << '\t';
    }
    // einDoubleStack.push(1099.986); // Fehler, da Stack voll
    cout << "\n4 Elemente des Double–Stacks entnehmen:" << endl;
    for(i = 0; i < 4; ++i) {
        cout << einDoubleStack.top() << '\t';
        einDoubleStack.pop();
    }
    cout << endl;
    cout << "Restliche Anzahl : " << einDoubleStack.size() << endl;
    cout << "clear Stack" << endl;
    einDoubleStack.clear();
    cout << "Anzahl : " << einDoubleStack.size() << endl;
   // einDoubleStack.pop(); // Fehler, da Stack leer
}
```

Die Erzeugung der `SimpleStack`-Objekte für verschiedene Datentypen geschieht erst beim Lesen der Definition durch den Compiler, der dann in Kenntnis der Template-Beschreibung in *simstack1.t* einen Stack für `int`- und einen für `double`-Zahlen erzeugt. Die Erzeugung eines Objekts für einen konkreten Datentyp anstelle des Platzhalters `T` im Template wird *Instanziierung eines Templates* genannt. Ein Stack kann mit Hilfe des Templates für ganz verschiedene Datentypen deklariert werden, zum Beispiel können wir Stacks für rationale Zahlen oder andere beliebige Objekte bauen, zum Beispiel Datumobjekte, falls wir den Typ Datum vorher definiert haben:

```
SimpleStack<Rational> einStackFuerRationaleZahlen;
SimpleStack<Datum> einStackFuerDaten;
```

6.3.2 Stack mit statisch festgelegter Größe

Einen kleinen Schönheitsfehler hat der `SimpleStack`: Seine Größe ist auf `MAX_SIZE` fest eingestellt. Es gäbe natürlich die Lösung, die Größe dem Konstruktor zu übergeben, der den benötigten Platz dynamisch mit `new` beschafft, und den Stack dynamisch je nach Bedarf zu erweitern.

Der Template-Mechanismus bietet jedoch auch eine statische Lösung: Innerhalb der spitzen Klammern `< >` können *mehrere* Datentypen (Klassen und Grunddatentypen) und Werte integraler Typen angegeben werden, die in ihrer Gesamtheit einen neuen Datentyp definieren. Die Größe eines Stacks gehört dann zum Datentyp, wird also zur Compilierzeit festgelegt. Die geringfügigen Änderungen in *simstack.t* sind

- Streichen der Konstante `MAX_SIZE`,
- Ersatz von `<class T>` durch `<class T, unsigned int MAX_SIZE>` und Anpassung der darauf aufbauenden Definitionen (siehe unten).

Die Deklarationen in einem Anwendungsprogramm sind ebenfalls zu modifizieren, die Benutzung bleibt sonst gleich:

Listing 6.6: Template-Klasse für einen Stack, 2. Version

```
// cppbuch/k6/stack/simstack2.t    einfaches Stack-Template, 2. Version
#ifndef SIMSTACK2_T
#define SIMSTACK2_T
#include<cassert>
                  // Parameter MAX_SIZE zur Festlegung der Stackgröße
template<typename T, unsigned int MAX_SIZE>
class SimpleStack {
   // ... genau wie oben, aber ohne MAX_SIZE
};

// noch fehlende Implementierungen
template<typename T, unsigned int m> // Parameter m wird nicht benutzt
const T& SimpleStack<T, m>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template<typename T, unsigned int m> // Parameter m wird nicht benutzt
void SimpleStack<T, m>::pop() {
    assert(!empty());
    --anzahl;
}

template<typename T, unsigned int m> // Parameter m wird nicht benutzt
void SimpleStack<T, m>::push(const T& x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif     // SIMSTACK2_T
```

Eine mögliche Anwendung finden Sie auf der nächsten Seite.

```
// ein int-Stack mit max. 100 Elementen:
SimpleStack<int, 100> einIntStack;
// Stack füllen
int i;
while(!einIntStack.full()) {
    cin >> i;
    einIntStack.push(i);
}
```

```
// ein char-Stack mit max. 9 Elementen
SimpleStack<char, 9> einCharStack;
// ...
```

Weil die Größe `MAX_SIZE` nicht dem Objekt, sondern dem Template übergeben wurde, ist die Größe eines `SimpleStack` schon zur Übersetzungszeit bekannt, sodass `simpleStack`-Objekten statisch Speicherplatz zugeteilt wird, also ohne Rückgriff auf den dynamischen Speicher.

Übungen

6.2 Schreiben Sie eine Klasse `Format` zum Formatieren von Zahlen. Benutzungsbeispiel:

```
// Konstruktion des Format-Objekts
Format f(12, 3);      // Ausgabe 12 Zeichen breit, 3 Nachkommastellen
cout << f.toString(789.906625) << endl; // Benutzung
cout << f.toString(-123456789.906625) << endl;
```

Das Ergebnis soll ␣␣␣␣␣789,907 im ersten Fall und -123456789,907 im zweiten Fall sein, wobei ␣ hier für ein Leerzeichen steht. Im zweiten Fall reichen 12 Plätze zur Darstellung aller Ziffern nicht aus. Die Weite wird daher automatisch erweitert, um Informationsverlust zu vermeiden.

6.3 Das folgende Programmfragment soll einen Ausschnitt einer Party simulieren. Das Array `alle` fasst alle Teilnehmer zusammen; mit seiner Hilfe werden ihre Namen und die ihrer Bekannten ausgegeben.

```
int main() {
  Teilnehmer otto("Otto");
  Teilnehmer andrea("Andrea");
  Teilnehmer jens("Jens");
  Teilnehmer silvana("Silvana");
  Teilnehmer miriam("Miriam");
  Teilnehmer paul("Paul");
  Teilnehmer* const alle[] = {&otto, &andrea, &jens,
        &silvana, &miriam, &paul, 0}; // 0 = Endekennung
  andrea.lerntKennen(jens);
  silvana.lerntKennen(otto);
  paul.lerntKennen(otto);
  paul.lerntKennen(silvana);
  miriam.lerntKennen(andrea);
  jens.lerntKennen(miriam);
  jens.lerntKennen(silvana);
  if(jens.kennt(andrea)) {
    cout << "Jens kennt Andrea" << endl;
```

```
  }
  int i = 0;
  // Ausgabe aller Teilnehmer mit Angabe, wer wen kennt:
  while(alle[i]) {
    cout << alle[i]->gibNamen() << " kennt: ";
    alle[i]->druckeBekannte();
    ++i;
  }
}
```

Schreiben Sie eine zu diesem Programm passende Klasse `Teilnehmer`. Dabei soll berücksichtigt werden, dass »kennenlernen« als »sich gegenseitig kennenlernen« gemeint ist. Wenn Paul also Silvana kennen lernt, lernt sie ihn umgekehrt auch kennen. Ein Methodenaufruf, der meint, jemand lernt sich selbst kennen (etwa `jens.lerntKennen(jens)`), soll ignoriert werden. Was muss für den Gültigkeitsbereich der Objekte gelten, wenn Sie die Bekannten eines Teilnehmers in einem Attribut des Typs `vector<Teilnehmer*>` speichern? Gibt es eine bessere Möglichkeit?

6.4 Template-Metaprogrammierung[1]

Zur Vertiefung des Verständnisses von Templates wird gezeigt, wie der *Compiler* zum Rechnen gebracht werden kann. Diese Art der Programmierung von Templates heißt *Template-Metaprogrammierung*. Das folgende Programm berechnet eine Zweierpotenz, im Beispiel $2^{11} = 2048$. Der Compiler versucht bei der Übersetzung, den Wert des Attributs `Zweihoch<11>::Wert` zu ermitteln. Die klassenspezifische Aufzählungskonstante `Wert` hat für jeden Typ der Klasse `Zweihoch`, der von `n` abhängt, einen anderen Wert. Bei der Ermittlung stellt der Compiler fest, dass `Zweihoch<11>::wert` dasselbe wie `2·Zweihoch<10>::wert` ist. `Zweihoch<10>::wert` wiederum ist dasselbe wie `2·Zweihoch<9>::wert` usw.

Listing 6.7: Rekursive Templates

```
// cppbuch/k6/rekursiveTemplates/zweihoch.cpp
#include<iostream>

template<int n>
struct Zweihoch {
  enum { wert = 2*Zweihoch<n-1>::wert};
};
template<> struct Zweihoch<0> {
   enum { wert = 1};
};

int main() {
    std::cout << Zweihoch<11>::wert << std::endl;
}
```

[1] Der Rest des Kapitels kann beim ersten Lesen übersprungen werden.

Die Rekursion bricht bei der Berechnung von `Zweihoch<0>::wert` ab, weil das Template für diesen Fall spezialisiert und der Wert mit 1 besetzt ist. Der Compiler erzeugt insgesamt 12 Datentypen (0 bis 11), die er zur Auswertung heranzieht. Weil er konstante Ausdrücke zur Compilationszeit kennt und berechnen kann, wird an die Stelle von `Zweihoch<11>::wert` direkt das Ergebnis 2048 eingetragen, sodass zur Laufzeit des Programms keinerlei Rechnungen mehr nötig sind!

Diese Methode zur Berechnung von Zweierpotenzen schlägt damit jede andere, was die Rechenzeit des Programms angeht. Dieses Verfahren wird mit gutem Erfolg erweitert auf andere Probleme wie zum Beispiel die Berechnung der schnellen Fouriertransformation und die Optimierung von Vektoroperationen ([Ve95]), stellt aber hohe Anforderungen an die verwendeten Compiler. Insbesondere ist die Tiefe der möglichen Template-Instanziierungen begrenzt.

Als Ergänzung werden im folgenden Beispiel Primzahlen vom *Compiler* berechnet, aber erst zur Laufzeit ausgegeben. Dabei gibt es keinerlei Schleifen oder Funktionsaufrufe, nur die statische, allerdings rekursive Konstruktion von Objekten. Die Spezialisierungen sorgen für den Abbruch der Rekursion. Anstelle der 17 kann eine andere Zahl stehen. Die mögliche Höchstzahl ist abhängig vom verwendeten Compiler. Dieses Programm wurde nach einer Idee von Erwin Unruh geschrieben, der 1994 ein Programm konstruierte, das bei Übersetzung Primzahlen in den Fehlermeldungen des Compilers erzeugte [Unr]. Für sich genommen, scheinen beide Beispiele eher Kuriositäten zu sein. Als Übung zum Verständnis der angegebenen weiterführenden Literatur [CE] und [Ve95] bzw. der darauf aufbauenden numerischen Bibliotheken sowie des folgenden Abschnitts sind sie aber gut geeignet.

Listing 6.8: Primzahlen mit Templates berechnen

```
// cppbuch/k6/rekursiveTemplates/primzahl.cpp
#include<iostream>
using namespace std;

template<int p, int i>
struct istPrimzahl {
   // p ist nur dann prim, wenn p nicht durch i teilbar ist und auch nicht durch
   // alle anderen Teiler zwischen 2 und i. Wenn i==2 ist, wird
   // istPrimzahl<0, 1>::prim gefragt, d.h. Abruch der Rekursion (s.u.).
   enum {prim = (p%i) && istPrimzahl<(i>2? p:0), i-1>::prim};
};

template<int i>
struct druckePrimzahlenBis {
   // Der folgende Konstruktoraufruf sorgt dafür, dass die
   // kleineren Primzahlen rekursiv ausgegeben werden.
   druckePrimzahlenBis<i-1> a;
   enum { prime = istPrimzahl<i, i-1>::prim};
   druckePrimzahlenBis() {
     if(prime) {
        cout << i << endl;
     }
   }
};
```

```
// Rekursionsabbruch durch Spezialisierungen
template<> struct istPrimzahl<0,1> { enum {prim = 1};};
template<> struct druckePrimzahlenBis<2> { //
   druckePrimzahlenBis() { cout << 2 << endl; }
};

int main() {
   druckePrimzahlenBis<17> a;
}
```

Die in diesem Abschnitt diskutierte Metaprogrammierung bekommt steigendes Gewicht. Es geht darum, schon zur Compilationszeit abhängig vom Typ oder bestimmten Eigenschaften Entscheidungen zu treffen, etwa welcher Algorithmus gewählt werden soll. Diese Typinformationen sind in sogenannten Traits-Klassen festgelegt. Traits heißt Eigenschaft oder Merkmal. Der C++-Standard unterstützt Metaprogrammierung durch Bereitstellung von Templates für Typinformationen. Die vordefinierten Typ-Klassen der Standardbibliothek finden Sie im Header `<type_traits>`, weitere Informationen in [ISOC++, Abschnitt 20.7]. Der Abschnitt 29.1.1 dieses Buchs zeigt die Anwendung von Traits.

Das obige Beispiel zeigt, dass schon zur Compilationszeit gerechnet werden kann. Schleifen werden dabei stets mit einer Rekursion realisiert – wie auch im nächsten Abschnitt zu sehen. Menschen mit Lisp- oder Prolog-Erfahrung wird das bekannt vorkommen.

6.5 Variadic Templates: Templates mit variabler Parameterzahl

Die Anzahl der Parameter von Funktionen und Operatoren wird Stelligkeit oder Arität genannt. Zum Beispiel ist die Addition zweistellig, weil sie zwei Argumente benötigt. Bis zu diesem Abschnitt werden Templates mit einer festen Anzahl von Parametern (Typen) definiert. Bei der Benutzung müssen die Typen entsprechend der Anzahl angegeben werden. Die Stelligkeit ist festgelegt. So hat das Stack-Template von Seite 249 die Stelligkeit zwei:

```
template<typename T, unsigned int MAX_SIZE> // zwei Parameter
class SimpleStack {
   // ... Rest weggelassen
};
```

Die C++-Standardbibliothek kann in vielen Teilen einfacher geschrieben werden, seitdem es Templates mit variabler Stelligkeit (englisch *variadic templates*) gibt. Ein Beispiel dafür ist die Bibliotheksklasse `tuple` (für Tupel) von Seite 752. Ein anderes Beispiel ist eine Funktion zum Ausdrucken aller Parameter, bei der man vorher noch nicht weiß, mit wie vielen Argumenten sie aufgerufen werden wird. Für jede beliebige Anzahl von Parametern jeweils eine Funktion zu schreiben, ist nicht praktikabel. Aus diesem Grund gibt es in der Programmiersprache C die Funktion `int printf(const char* format, ...)`,

der eine beliebig lange Parameterliste übergeben werden kann. `format` ist der C-String, der die Formatierung steuert. Die drei Punkte heißen Ellipse, was Auslassung bedeutet. Sie stehen für eine Folge von Parametern. So gibt der Aufruf `printf("Wert = %.4f\n", 1.2345678);` die Zeile »Wert = 1.2346« aus. Dabei meint `%.4f\n`, dass eine Float-Zahl mit 4 Dezimalstellen nach dem Komma ausgegeben und dann eine neue Zeile begonnen wird. `printf()` gibt die Anzahl der ausgegebenen Zeichen zurück bzw. -1 bei einem Fehler. `printf()` ist jedoch nicht typsicher, das heißt, falsche Typen in der Parameterliste können nicht schon vom Compiler entdeckt werden.

Douglas Gregor hatte unter der Überschrift »Variadic Templates« einen Vorschlag, wie Ellipsen typsicher gestaltet werden können, entwickelt. Der Vorschlag wurde in den C++-Standard aufgenommen. Einen Übersichtsartikel, aus dem die folgenden, leicht abgewandelten Beispiele stammen, finden Sie unter [GrJ]. Wie Templates mit variabler Stelligkeit funktionieren, sei am Beispiel einer Funktion `anzeigen()` demonstriert, der beliebig viele Parameter zur Ausgabe mit `cout <<` übergeben werden können. Dabei prüft der Compiler, ob der Typ eines Parameters überhaupt zum Ausgabeoperator `<<` passt. Zur Vereinfachung wird auf die Steuerung des Ausgabeformats verzichtet.

Listing 6.9: Funktion mit variabler Parameteranzahl

```
// cppbuch/k6/variadicTemplate/anzeigen.cpp, nach [GrJ]
#include<iostream>

void anzeigen() {
   std::cout << std::endl;
 }

template<typename T, typename... Rest>
void anzeigen(const T& obj, const Rest&... rest) {
   std::cout << obj << " ";
   anzeigen(rest...);
}

int main() {
   anzeigen(1);
   anzeigen(2, "Hallo");
   anzeigen("Text", 7.978353, 3);
}
```

Wie Sie sehen, wird `anzeigen()` mit einem bis drei Parametern aufgerufen. Das entscheidende Element ist die Template-Definition:

- Die Ellipse ... nach `typename` bedeutet, dass an dieser Stelle null oder mehr Template-Parameter in `Rest` zusammengefasst werden. Der Parameter `Rest` wird »Template Parameter Pack« genannt. Wenn `anzeigen()` mit mehreren Parametern aufgerufen wird, wird der erste `T` zugeordnet, alle anderen dem Parameter `Rest` – daher der Name.
- In der Parameterliste der Funktion wird `Rest` fast wie ein normaler Typ verwendet. Der syntaktische Unterschied besteht nur darin, dass eine Ellipse ... folgt, um die Eigenschaft »Template Parameter Pack« zu markieren.

- Dasselbe gilt für den Funktionsaufruf `anzeigen(rest...)`: In diesem Aufruf ist `rest...`, gekennzeichnet durch eine Ellipse, ein »Template Parameter Pack«. Dieser Aufruf hat einen Parameter weniger als die aufrufende Funktion!

Wie verarbeitet der Compiler die Anweisung `anzeigen("Text", 7.978353, 3);`? Die Abfolge in einzelnen Schritten ist:

- Zuerst wird eine Ausgabe für das erste Objekt, den C-String *"Text"*, erzeugt. Der anschließende Aufruf `anzeigen(rest...);` ist nichts anderes als `anzeigen(7.978353, 3);` – es wird nur der Rest übergeben.
- Der Aufruf `anzeigen(7.978353, 3);` wird genauso behandelt; es wird 7.978353 ausgegeben und dann `anzeigen(3);` aufgerufen.
- `anzeigen(3);` resultiert in der Ausgabe von 3 und dem Aufruf `anzeigen();`.
- `anzeigen();` ruft die Funktion ohne Parameter oben in der Datei. Diese Funktion beendet nur die laufende Zeile.

Der Compiler erzeugt aus dem Aufruf `anzeigen("Text", 7.978353, 3);` also die folgenden Funktionen aus dem Template:

```
void anzeigen(const char*, double, int);
void anzeigen(double, int);
void anzeigen(int);
```

Die Funktion ohne Argumente (`anzeigen()`) ist vorhanden und wird daher nicht erzeugt. Sie beendet die Rekursion.

Anzahl der Parameter ermitteln

Templates mit variabler Stelligkeit sind auch für Klassen möglich, wie hier am Beispiel einer Struktur zum Zählen der Parameteranzahl gezeigt wird. Die Auswertung geschieht zur Compilationszeit:

Listing 6.10: Anzahl der Parameter ermitteln

```
// cppbuch/k6/variadicTemplate/paramzaehlen.cpp, nach [GrJ]
#include<iostream>

// Template-Deklaration
template<typename... Args> struct Anzahl;

// partielle Spezialisierung (Rekursion)
template<typename Head, typename... Tail>
struct Anzahl<Head, Tail...> {
   static const int wert = 1 + Anzahl<Tail...>::wert;
};

// partielle Spezialisierung (Rekursionsabbruch)
template<>
struct Anzahl<> {
  static const int wert = 0;
};

int main() {
   std::cout << "Parameteranzahl von Anzahl<char*, int, double>: "
```

```
            << Anzahl<char*, int, double>::wert << std::endl;
}
```

Ganz oben steht die Template-Deklaration. Eine Implementierung fehlt, weil sie nicht gebraucht wird. Anschließend sehen Sie eine partielle Spezialisierung, in der die Liste der Typen in das erste Element (`Head`) und den Rest (`Tail`) zerlegt wird. Bei der Auswertung dieser Spezialisierung stellt der Compiler fest, dass er zur Berechnung `Anzahl<Tail...>::wert` benötigt. Dazu muss er den Typ `Anzahl<Tail...>` instanziieren, dessen Typliste nunmehr um ein Element verkürzt ist. Auf `Anzahl<Tail...>` wird dasselbe Verfahren angewendet, sodass auch `Tail` in einen Kopf und einen Rest verwandelt wird usw. Dieser Prozess wird abgearbeitet, bis die Typliste leer ist. Dann kommt die zweite Spezialisierung zur Geltung, und die Rekursion bricht ab. Der Compiler erzeugt also bei der Auswertung von `Anzahl<char*, int, double>::wert` sukzessive die folgenden Typen aus dem Template:

```
Anzahl<char*, int, double>
Anzahl<int, double>
Anzahl<double>
```

`Anzahl<>` wird, weil vorhanden, nicht erzeugt. Für jeden der neu erzeugten Typen wird die Konstante `wert` bereits zur Compilationszeit berechnet, sodass sich 3 als Ausgabe des Programms ergibt.