3

Programmstrukturierung

Dieses Kapitel behandelt die folgenden Themen:

- Funktionen und ihr Aufbau
- Rückgabe von Funktionsergebnissen
- Die verschiedenen Arten der Parameterübergabe
- Makros
- Modulare Gestaltung und Strukturierung von Programmen
- Funktionen mit parametrisierten Datentypen
- Namespaces

Große Programme müssen in übersichtliche Teile zerlegt werden. Sie werden dazu verschiedene Mechanismen kennenlernen. Sie erfahren, wie eine Funktion aufgebaut ist, wie man einer Funktion die von ihr benötigten Daten mitteilen kann und auf welche Weise die Ergebnisse von Funktionen zurückgegeben werden können. Die Simulation eines Taschenrechners zeigt beispielhaft den wechselseitigen Einsatz von Funktionen. Anschließend werden Grundsätze der modularen Gestaltung behandelt, ohne deren Einhaltung große Programme oder Programmsysteme kaum mehr handhabbar sind. Der Einsatz von Funktionen mit parametrisierten Datentypen ermöglicht einen breiteren Einsatz von Funktionen ohne fehlerträchtige Vervielfachung des Programmcodes.

3.1 Funktionen

Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Die Teilaufgabe kann einfach, aber auch sehr komplex sein. Die notwendigen Daten werden der Funktion mitgegeben, und sie gibt das Ergebnis der erledigten Aufgabe an den Aufrufer (Auftraggeber) zurück. Eine einfache mathematische Funktion ist zum Beispiel $y = sin(x)$, wobei x der Funktion als notwendiges Datum übergeben und y das Ergebnis zugewiesen wird. In C++ können die verschiedensten Funktionen programmiert oder benutzt werden, nicht nur mathematische.

Eine Funktion muss nur einmal definiert werden. Anschließend kann sie beliebig oft nur durch Nennung ihres Namens aufgerufen werden, um die ihr zugewiesene Teilaufgabe abzuarbeiten. Dieses Prinzip setzt sich in dem Sinne fort, dass Teilaufgaben selbst wieder in weitere Teilaufgaben unterteilbar sein können, die durch Funktionen zu bearbeiten sind. Wie in einer großen Firma die Aufgaben nur durch Arbeitsteilung, Delegation und einer sich daraus ergebenden hierarchischen Struktur zu bewältigen sind, wird in der Informatik die Komplexität einer Aufgabe durch Zerlegung in Teilaufgaben auf mehreren Ebenen reduziert – nach dem Prinzip »Teile und herrsche«. Bereits vorhandene Standardlösungen von Teilaufgaben können aus Funktionsbibliotheken abgerufen werden – ebenso wie neu entwickelte Funktionen in Bibliotheken aufgenommen werden können.

3.1.1 Aufbau und Prototypen

Auf Seite 74 wird die Fakultät einer Zahl berechnet. Dies soll die Grundlage für eine einfache Funktion `fakultaet()` bilden, die diese Aufgabe ausführt. Die Fakultät $n!$ ist definiert als das Produkt $n(n-1)(n-2)...3 \cdot 2 \cdot 1$. Dabei ist $0! = 1$ festgelegt. Das Beispiel zeigt die Integration der Funktion in ein `main`-Programm:

Listing 3.1: Beispielprogramm mit einer Funktion

```
// cppbuch/k3/fakulta2.cpp
#include<iostream>
using namespace std;
unsigned long fakultaet(int);        // Funktionsprototyp (Deklaration)

int main() {
   int n;
   do {
      cout << "Fakultät berechnen. Zahl >= 0? :";
      cin >> n;
   } while(n < 0);
   cout << "Das Ergebnis ist " << fakultaet(n) << endl;    // Aufruf
}

unsigned long fakultaet(int zahl) { // Funktionsimplementation (Definition)
    unsigned long fak = 1;
    for(int i = 2; i <= zahl; ++i)
       fak *= i;
   return fak;
}
```

Eine *Deklaration* sagt dem Compiler, dass eine Funktion oder eine Variable mit diesem Aussehen irgendwo definiert ist. Damit kennt er den Namen bereits, wenn er auf einen Aufruf der Funktion stößt, und ist in der Lage, eine Syntaxprüfung vorzunehmen. Eine *Definition* veranlasst den Compiler, entsprechenden Code zu erzeugen und den notwendigen Speicherplatz anzulegen. Eine Funktionsdeklaration, die nicht gleichzeitig eine Definition ist, wird *Funktionsprototyp* genannt. Eine Vereinbarung einer Variablen mit `int i;` ist sowohl eine Deklaration als auch eine Definition. Auf die Begriffe Deklaration und Definition wird in Abschnitt 3.3.4 genauer eingegangen. Der Aufruf der Funktion geschieht einfach durch Namensnennung. Von der Funktion auszuwertende Daten werden in runden Klammern `( )` übergeben. Der Funktionstyp `void` bedeutet, dass nichts zurückgegeben wird. Wenn eine Funktion einen Rückgabetyp ungleich `void` hat, muss im Funktionskörper `{...}` irgendwo ein Ergebnis dieses Typs mit der Anweisung `return` zurückgegeben werden. Wenn eine Funktion etwas tut, ohne dass ein Funktionsergebnis zurückgegeben wird, wirkt sie nur durch sogenannte *Seiteneffekte*. Andere Möglichkeiten der Ergebnisrückgabe werden in Abschnitt 3.2 vorgestellt. Die Wirkung eines Funktionsaufrufs ist, dass das zurückgegebene Ergebnis an die Stelle des Aufrufs tritt! Abbildung 3.1 zeigt die Syntax eines *Funktionsprototyps* (siehe obiges Beispiel).

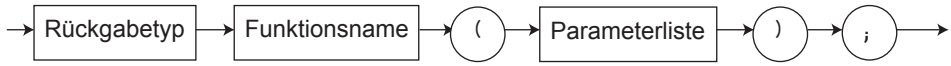

Abbildung 3.1: Syntaxdiagramm eines Funktionsprototyps

Der Rückgabetyp, auch Typ der Funktion genannt, kann ein nahezu beliebiger Datentyp sein. Ausnahmen sind die Rückgabe einer Funktion sowie die Rückgabe des bisher noch nicht besprochenen C-Arrays. Betrachten Sie die Zuordnung der einzelnen Teile der obigen Deklaration von `fakultaet()`:

```
unsigned long     fakultaet     (       int       );
      ⋮               ⋮         ⋮        ⋮        ⋮
 Rückgabetyp    Funktionsname    (   Parameterliste   );
```

Die *Parameterliste* besteht in diesem Fall nur aus einem einzigen Parametertyp. Je nach Aufgabenstellung bestehen für den Aufbau einer *Parameterliste* folgende Möglichkeiten:

	Beispiel:
leere Liste:	`int func();`
gleichwertig ist:	`int func(void);`
Liste mit Parametertypen:	`int func(int, char);`
Liste mit Parametertypen und -namen:	`int func(int x, char y);`

Parameternamen wie `x` und `y` dienen der Erläuterung. Sie dürfen entfallen, was aber nur dann tolerierbar ist, wenn der Sinn unmissverständlich ist. In allen anderen Fällen ist es vorteilhafter, die Namen hinzuschreiben, damit später die Benutzung der Funktion sofort klar wird, ohne die Dokumentation bemühen zu müssen. Abbildung 3.2 zeigt die Syntax einer *Funktionsdefinition* Der eigentliche Programmcode ist im Block der Funktionsdefinition enthalten. Betrachten wir auch jetzt die Zuordnung der einzelnen Teile der obigen Definition von `fakultaet()`, wobei der Programmcode durch »...« angedeutet ist:

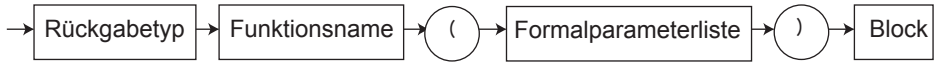

Abbildung 3.2: Syntaxdiagramm einer Funktionsdefinition

```
unsigned long     fakultaet     (        int zahl        )   {...}
```

Rückgabetyp *Funktionsname* (*Formalparameterliste*) *Block*

Die *Formalparameterliste* enthält im Unterschied zur reinen Deklaration zwingend einen Parameternamen (hier `zahl`), der damit innerhalb des Blocks bekannt ist. Der Name ist frei wählbar und völlig unabhängig vom Aufruf, weil er nur als Platzhalter dient. Abbildung 3.3 zeigt die Syntax eines *Funktionsaufrufs*.

Abbildung 3.3: Syntaxdiagramm eines Funktionsaufrufs

Die *Aktualparameterliste* enthält Ausdrücke und/oder Namen der Objekte oder Variablen, die an die Funktion übergeben werden sollen. Sie kann leer sein. In unserem Beispiel besteht die Aktualparameterliste nur aus `n`. Dass der Datentyp von `n` mit dem Datentyp in der *Deklaration* übereinstimmt, wird vom Compiler geprüft. Der Linker stellt fest, ob eine entsprechende *Definition* der Funktion mit dem richtigen Datentyp in der Formalparameterliste vorhanden ist. Der Aufruf der Funktion bewirkt, dass der Wert von `n` an die Stelle des Platzhalters `zahl` gesetzt und dann der Programmcode im Block durchgeführt wird. Am Schluss wird die berechnete Fakultät mit dem richtigen Ergebnisdatentyp zurückgegeben. Zurückgegeben wird nur der Wert von `fak`, nicht `fak` selbst. Die Variablen `fak` und `zahl` sind *lokal*, d.h. im Hauptprogramm nicht bekannt und nicht zugreifbar. Ergebnisrückgabe heißt einfach, dass an die Stelle des Aufrufs von `fakultaet()` im Hauptprogramm das Ergebnis eingesetzt wird.

Das Prinzip der Ersetzung der Formalparameter durch die Aktualparameter ist eine wichtige Voraussetzung, um eine Funktion universell verwenden zu können. Es ist ganz gleichgültig, ob die Funktion in einem Programm mit `fakultaet(zahl)` oder in einem anderen Programm mit `fakultaet(xyz)` aufgerufen wird, wenn nur der Datentyp des Parameters mit dem vorgegebenen (in diesem Fall `int`) übereinstimmt.

3.1.2 Gültigkeitsbereiche und Sichtbarkeit in Funktionen

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Variable (siehe Seite 58). Die gleichen Regeln gelten auch für Funktionen. Der Funktionskörper ist ein Block, also ein durch geschweifte Klammern { } begrenztes Programmstück. Danach sind alle Variablen einer Funktion nicht im Hauptprogramm gültig und auch nicht sichtbar. Eine Sonderstellung haben die in der Parameterliste aufgeführten Variablen: Sie werden innerhalb der Funktion wie lokale Variable betrachtet, und von außen gesehen stellen sie die *Datenschnittstelle* zur Funktion dar. Die Datenschnittstelle ist ein Übergabepunkt für Daten. *Eingabeparameter* dienen zur Übermittlung von Daten an die Funktion, und über *Ausgabeparameter* (Abschnitt 3.2.2) sowie den `return`-Mechanismus gibt eine Funktion Daten

an den Aufrufer zurück. Die Variable `zahl` aus `fakultaet()` ist also von `main()` aus nicht zugreifbar, wie umgekehrt alle in `main()` deklarierten Variablen in `fakultaet()` nicht benutzt werden können. Diese Variablen sind *lokal*. Ein Beispiel soll das verdeutlichen, wobei hier die *Deklaration* von `f1()` gleichzeitig eine *Definition* ist, weil sie nicht nur den Namen vor dem Aufruf von `f1()` einführt, sondern auch den Funktionskörper enthält. Dieses Vorgehen ist nur für sehr kleine Programme wie hier zu empfehlen.

Listing 3.2: Sichtbarkeitsbereich

```
// cppbuch/k3/scope.cpp
#include<iostream>
using namespace std;

int a = 1;                    // überall bekannt, also global

void f1( ) {
     int c = 3;               // nur in f1() bekannt, also lokal
     cout << "f1: c= " << c << endl;
     cout << "f1: globales a= " << a << endl;
}

int main() {
    cout << "main: globales a= " << a << endl;
    // cout << "f1: c= "<< c; ist nicht compilierbar, weil c in main() unbekannt ist.
    f1( );                    // Aufruf von f1()
}
```

Das Programm erzeugt folgende Ausgabe:

main: globales a= 1
f1: c= 3
f1: globales a= 1

Beim Betreten eines Blocks wird für die innerhalb des Blocks deklarierten Variablen Speicherplatz beschafft; die Variablen werden gegebenenfalls initialisiert. Der Speicherplatz wird bei Verlassen des Blocks wieder freigegeben. Dies gilt auch für Variablen in Funktionen, wobei der Aufruf einer Funktion dem Betreten des Blocks entspricht. Die Rückkehr zum Aufrufer der Funktion wirkt wie das Verlassen des Blocks.

3.1.3 Lokale static-Variable: Funktion mit Gedächtnis

Die Ausnahme bilden Variablen, die innerhalb eines Blocks oder einer Funktion als `static` definiert werden. Wenn es Konstante sind, die schon zur Compilationszeit bekannt sind, geschieht die Initialisierung *vor* dem Aufruf jedweder Funktion. In allen anderen Fällen wird die Variable *beim ersten Aufruf der Funktion* initialisiert. Im Beispiel unten wird `anz` schon vor dem Aufruf von `func()` mit 0 initialisiert (zur Compilationszeit bekannte Konstante). Würde `anz` den Wert von einer anderen Funktion `g()` erhalten, zum Beispiel `static int anz = g();`, dann würde `anz` erst beim ersten Aufruf von `func()` initialisiert. Falls kein Initialisierungswert vorgegeben ist, werden `static`-Zahlen auf 0 gesetzt. `static`-Variable wirken wie ein Gedächtnis für eine Funktion, weil sie zwischen Funktionsaufrufen ihren Wert nicht verlieren. Eine Funktion, die anzeigt, wie oft sie aufgerufen wurde, sieht so aus:

Listing 3.3: Funktion mit Gedächtnis

```
// cppbuch/k3/static.cpp
#include<iostream>
using namespace std;

void func( ) {            // zählt die Anzahl der Aufrufe
    static int anz = 0;   // behält den letzten Wert
    cout << "Anzahl = " << ++anz << endl;
}

int main() {
   for(int i = 0; i < 3; ++i)
      func( );
}
```

Die Ausgabe des Programms ist

Anzahl = 1
Anzahl = 2
Anzahl = 3

Ohne das Schlüsselwort `static` würde drei Mal *1* ausgegeben werden, weil die Zählung stets bei 0 begänne. Lokale `static`-Variablen sind globalen Variablen vorzuziehen, weil unabsichtliche Änderungen in anderen Funktionen vermieden werden und mit dieser Variablen verbundene Fehler leichter lokalisiert werden können. Außerdem erfordert eine globale Variable eine Absprache unter allen Benutzern der Funktion über den Namen. Gerade das soll aber vermieden werden, um eine Funktion universell einsetzbar zu machen. Auf die dateiübergreifende Gültigkeit von Variablen und Funktionen wird in Abschnitt 3.3.3 eingegangen.

3.2 Schnittstellen zum Datentransfer

Der Datentransfer in Funktionen hinein und aus Funktionen heraus kann unterschiedlich gestaltet werden. Er wird durch die Beschreibung der Schnittstelle festgelegt. Unter Schnittstelle ist eine formale Vereinbarung zwischen Aufrufer und Funktion über die Art und Weise des Datentransports zu verstehen und darüber, was die Funktion leistet. In diesem Zusammenhang sei nur der Datenfluss betrachtet. Die Schnittstelle wird durch den Funktionsprototyp eindeutig beschrieben und enthält

- den Rückgabetyp der Funktion,
- den Funktionsnamen,
- Parametertypen, die der Funktion bekannt gemacht werden, und somit
- die Art der Parameterübergabe.

Der Compiler prüft, ob die Definition der Schnittstelle bei einem Funktionsaufruf eingehalten wird. Zusätzlich zur Rückgabe eines Funktionswerts gibt es die Möglichkeit, die

an die Funktion über die Parameterliste gegebenen Daten zu modifizieren. Danach unterscheiden wir *zwei Arten des Datentransports*: die Übergabe *per Wert* und *per Referenz*.

3.2.1 Übergabe per Wert

Der Wert wird *kopiert* und der Funktion übergeben. Innerhalb der Funktion wird mit der *Kopie* weitergearbeitet, das Original beim Aufrufer *bleibt unverändert erhalten*. Im Beispiel wird beim Aufruf der Funktion `addiere_5()` der aktuelle Wert von `i` in die funktionslokale Variable `x` kopiert, die in der Funktion verändert wird. Der Rückgabewert wird der Variablen `erg` zugewiesen, `i` hat nach dem Aufruf denselben Wert wie zuvor. Abbildung 3.4 verdeutlicht den Ablauf.

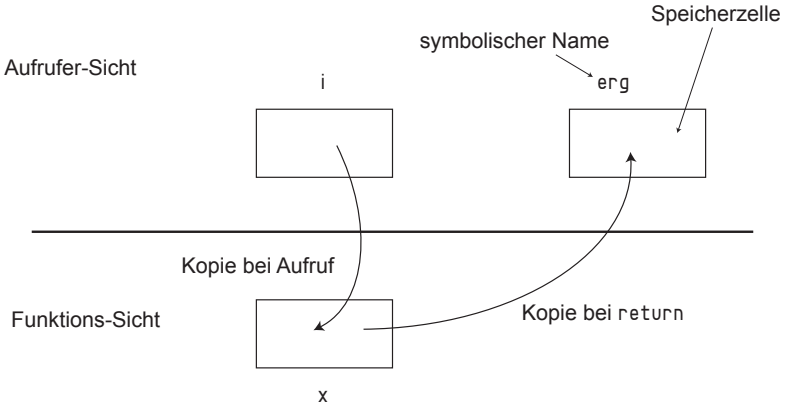

Abbildung 3.4: Parameterübergabe per Wert (Bezug: Programmbeispiel)

Listing 3.4: Übergabe per Wert

```
// cppbuch/k3/per_wert.cpp
#include<iostream>
using namespace std;
int addiere_5(int);     // Deklaration (Funktionsprototyp)

int main() {
    int erg, i = 0;
    cout << i << " = Wert von i\n";
    erg = addiere_5(i);
    cout << erg << " = Ergebnis von addiere_5\n";
    cout << i << " = i unverändert!\n";
}
int addiere_5(int x) { // Definition
     x += 5;
     return x;
}
```

Die Übergabe per Wert soll generell bevorzugt werden, wenn ein Objekt nicht geändert werden soll und es nicht viel Speicherplatz einnimmt. Letzteres ist für Grunddatentypen der Fall.

Übungen

3.1 Schreiben Sie eine Funktion `int dauerInSekunden(int stunden, int minuten, int sekunden)`, die die Gesamtzahl der Sekunden zurückgibt, berechnet aus den Parametern.

3.2 Schreiben Sie eine Funktion `double power(double x, int y)`, die x^y berechnen soll. Wenn Sie nicht mehr genau wissen sollten, was x^y bedeutet – hier ein paar Beispiele: $x^3 = x \cdot x \cdot x$, $x^{-2} = 1/(x \cdot x)$, $x^0 = 1$.

Rekursion

Innerhalb von Funktionen können andere Funktionen aufgerufen werden, die wiederum andere Funktionen aufrufen. Die Verschachtelung kann beliebig tief sein. Der Aufruf einer Funktion durch sich selbst wird *Rekursion* genannt. Das Programm zur Berechnung der Quersumme einer Zahl zeigt die Rekursion:

Listing 3.5: Beispielprogramm 1 mit Rekursion

```
// cppbuch/k3/qsum.cpp
#include<iostream>
using namespace std;

int qsum(long z){                          // Parameter per Wert übergeben
   if(z != 0 ) {
      int letzteZiffer = z % 10;
      return letzteZiffer + qsum(z/10); // Rekursion
   }
   else {    // Abbruchbedingung z == 0
      return 0;
   }
}

int main() {
   cout << "Zahl: ";
   long zahl;
   cin >> zahl;
   cout << "Quersumme = " << qsum(zahl);
}
```

Die letzte Ziffer einer Zahl erhält man durch modulo 10 (Restbildung), und sie kann durch ganzzahlige Division durch 10 von der Zahl abgetrennt werden. Anstatt die Summation in einer Schleife vorzunehmen, lässt sich das Prinzip des Programms in zwei Sätzen zusammenfassen:

1. Die Quersumme der Zahl 0 ist 0.
2. Die Quersumme einer Zahl ist gleich der letzten Ziffer plus der Quersumme der Zahl, die um diese Ziffer gekürzt wurde.

Die Quersumme von 348156 ist also (6 + die Quersumme von 34815). Auf jede Quersumme wird Satz 2 angewendet, bis Satz 1 gilt. Durch das sukzessive Abtrennen wird die Zahl irgendwann 0, sodass Satz 1 erfüllt ist und die Rekursion anhält. In diesem Fall ist die Verschachtelungstiefe gleich der Anzahl der Ziffern. Eine Rekursion *muss* auf eine

Abbruchbedingung zulaufen, damit keine unendlich tiefe Verschachtelung entsteht mit der Folge eines Stacküberlaufs. Zum Vergleich sei hier eine iterative Variante gezeigt:

```
int qsum(long z) {
   int sum = 0;
   while(z > 0) {
      sum += z % 10;
      z = z / 10;
   }
   return sum;
}
```

Eines der bekanntestesten Beispiele zur Rekursion sind die »Türme von Hanoi«. Dieses Beispiel hat eine leicht zu entwickelnde rekursive Lösung. Eine nicht-rekursive Lösung ist komplizierter und schwieriger zu finden. Die Geschichte: Buddhistische Mönche des Brahma-Tempels haben die Aufgabe, 64 Scheiben aus Gold, die ein Loch in der Mitte haben, von Stab A nach Stab B zu bringen. Stab C kann als Zwischenablage dienen.

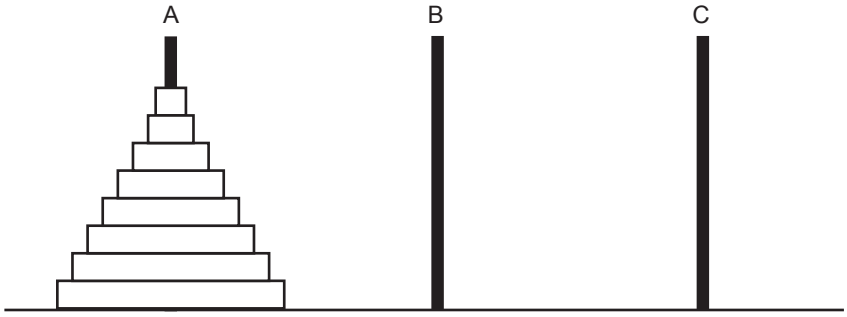

Abbildung 3.5: Türme von Hanoi

Die Mönche müssen zwei Regeln beachten:

1. Es darf nur eine Scheibe zurzeit bewegt werden.
2. Nie darf eine größere auf einer kleineren Scheibe zu liegen kommen.

Die Legende sagt, dass das Ende der Welt kommt, wenn die Mönche ihre Aufgabe beendet haben. Wie sieht ein Algorithmus aus, der den Mönchen sagt, welche Scheibe von welchem Stapel zu welchem Stapel bewegt werden soll, um die Aufgabe zu erfüllen? Ein einfacher Vorschlag:

1. Bringe 63 Scheiben von Stapel A nach Stapel C.
2. Bringe die unterste Scheibe von A nach Stapel B.
3. Bringe alle 63 Scheiben von Stapel C nach Stapel B – fertig!

Die Lösung ist sehr einfach, jedoch sagt sie nichts darüber, wie *63* Scheiben zu bewegen sind. Aber die Komplexität des Problems ist reduziert: Wenn wir wüssten, wie 63 Scheiben zu bewegen sind, wissen wir, wie alle 64 zu bewegen sind. Ein einfacher Vorschlag, 63 Scheiben von A nach C zu bringen:

1. Bringe 62 Scheiben von Stapel A nach Stapel B.
2. Bringe die unterste Scheibe von A nach Stapel C.

3. Bringe 62 Scheiben von Stapel B nach Stapel C.

Wir sehen ein allgemeines Muster, und auch, dass die Rollen von A, B, C gewechselt haben. Eine allgemeinere Formulierung für n Scheiben wäre:

1. Bringe n – 1 Scheiben vom Quell-Stapel zum Arbeits-Stapel.
2. Bringe die unterste Scheibe vom Quell-Stapel zum Ziel-Stapel.
3. Bringe n – 1 Scheiben vom Arbeits-Stapel zum Ziel-Stapel.

Dies ruft nach einer rekursiven Formulierung! Die Abbruchbedingung ist klar: Falls 0 Scheiben zu bewegen sind, tun wir nichts.

Listing 3.6: Beispielprogramm 2 mit Rekursion

```
// cppbuch/k3/hanoi.cpp
#include<iostream>
using namespace std;

void bewegen(int n, int quelle, int ziel, int zwischen) {
   if (n > 0) {                          // Abbruchbedingung: n == 0
      bewegen(n - 1, quelle, zwischen, ziel);   // rekursiver Aufruf
      cout << "Bringe eine Scheibe von " << quelle
           << " nach " << ziel << endl;
      bewegen(n - 1, zwischen, ziel, quelle);   // rekursiver Aufruf
   }
}

int main() {
   cout << "Türme von Hanoi! Anzahl der Scheiben: ";
   int scheiben;
   // besser nicht 64 eingeben, sondern eine kleinere Zahl,
   // zum Beispiel 4 (Begründung siehe unten).
   cin >> scheiben;
   bewegen(scheiben, 1, 2 ,3);
}
```

Analyse des Algorithmus

Wie viele Bewegungen braucht es? Jeder Aufruf von `bewegen()` erzeugt zwei neue Aufrufe. Auf jedem Level n gibt es zwei Aufrufe des Levels (n-1). Die Anweisung zwischen den Aufrufen, also die tatsächliche Bewegung, wird nur ausgeführt, wenn n$\geq$ 1 ist. Also ist die Gesamtzahl der Bewegungen $N = 1 + 2 + 4 + 8 + 16 + ... + 2^{n-1} = 2^n - 1$.

Wenn wir $n = 64$ und eine Sekunde pro Bewegung annehmen, erhalten wir eine *sehr* lange Zeitdauer, nämlich N = 18.446.744.073.709.551.615 Sekunden, also ungefähr $5.85 \cdot 10^{11}$ Jahre oder etwa das 50-fache des Alters unseres Universums. Selbst wenn die Legende stimmen sollte, dass nach der Erledigung der Aufgabe die Welt untergeht, bräuchten wir uns keine Sorgen zu machen!

Übungen

3.3 Schreiben Sie die Funktion zur Berechnung der Fakultät von Seite 102 als rekursive Funktion. Dabei gilt: 0! = 1, 1! = 1, $n! = n \cdot (n-1)!$

3.4 *Für Menschen mit Informatik-Vorkenntnissen:* Ein rekursiver Aufruf *am Ende* einer Funktion, die keinen Wert liefert (sogenannte Restrekursion), kann stets durch Einführung einer Schleife in die Funktion beseitigt werden. Wie müsste die Funktion `bewegen()` im Beispiel oben umgebaut werden, damit nur der erste rekursive Aufruf übrig bleibt? Hinweis: Eine `while(n > 0)`-Schleife umschließt den ersten rekursiven Aufruf. Die Änderung von n und die Änderung der Reihenfolge der Parameter a, b, c ersetzen den zweiten Aufruf.

3.2.2 Übergabe per Referenz

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe durch eine *Referenz* des Objekts geschehen. Die Syntax des Aufrufs ist die gleiche wie bei der Übergabe per Wert; anstatt mit einer Kopie wird jedoch *direkt mit dem Original* gearbeitet, wenn auch unter anderem Namen (vergleiche Seite 55). Der Name ist lokal bezüglich der Funktion, und er bezieht sich auf das übergebene Objekt. Es wird also keine Kopie angelegt. Daher ergibt sich bei großen Objekten ein Laufzeitvorteil. Innerhalb der Funktion vorgenommene Änderungen wirken sich direkt auf das Original aus.

Es wurde darauf hingewiesen, dass die Übergabe von *nicht zu verändernden* Objekten generell per Wert erfolgen soll mit der Ausnahme großer Objekte aus Effizienz- und Speicherplatzgründen. Wenn zwar der Laufzeitvorteil, aber keine Änderung des Originals erwünscht ist, kommt die Übergabe eines Objekts als *Referenz auf* `const` in Frage. Die Angabe in der Parameterliste könnte zum Beispiel `const TYP& unveraenderliches_grosses_Objekt` lauten. Innerhalb der Funktion darf auf das übergebene Objekt natürlich nur lesend zugegriffen werden; dies wird vom Compiler geprüft. Das Prinzip der Übergabe per Referenz zeigt folgendes Beispielprogramm.

Abbildung 3.6 zeigt, dass dasselbe Objekt unter verschiedenen Namen vom aufrufenden Programm und von der Funktion zugreifbar ist.

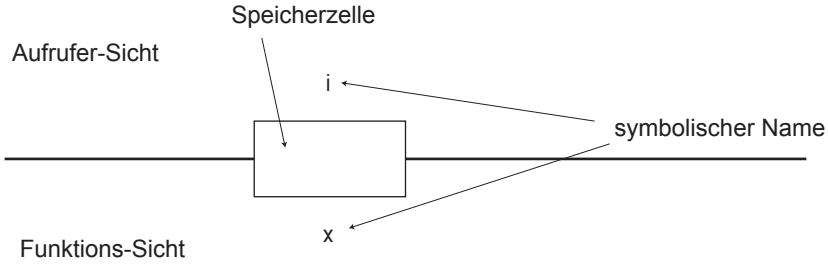

Abbildung 3.6: Parameterübergabe per Referenz (Bezug: Programmbeispiel)

Die Stellung des `&`-Zeichens in der Parameterliste ist beliebig. `(int& x)` ist genau so richtig wie `(int &x)` oder `(int & x)`. Bei der Diskussion über Laufzeitvorteile durch Referenzparameter darf nicht vergessen werden, dass es häufig Fälle gibt, in denen bewusst die Kopie eines Parameters *ohne* Auswirkung auf das Original geändert werden soll, sodass nur eine Übergabe per Wert in Frage kommt. Ein Beispiel ist der Parameter `z` der Funktion `qsum()` von Seite 109.

Listing 3.7: Übergabe per Referenz

```
// cppbuch/k3/per_ref.cpp
#include<iostream>
using namespace std;

void addiere_7(int&);  // int& = Referenz auf int

int main() {
    int i = 0;
    cout << i << " = alter Wert von i\n";
    addiere_7(i);
    cout << i << " = neuer Wert von i nach addiere_7\n";
}

void addiere_7(int& x) {
    x += 7;        // Original des Aufrufers wird geändert!
}
```

3.2.3 Gefahren bei der Rückgabe von Referenzen

Bei der Rückgabe von Referenzen muss darauf geachtet werden, dass das zugehörige Objekt tatsächlich noch existiert (vgl. Kapitel 3.1.2). Das folgende Beispiel zeigt, wie man es *nicht* machen soll:

Negativ-Beispiel!

```
int& maxwert(int a, int b) { // Referenz ?
    // a und b sind lokale Kopien der übergebenen Daten!
    if(a > b) return a;              // Fehler!
    else      return b;              // Fehler!
}

int main() {
    int x = 17, y = 4;
    int& z = maxwert(x, y);
    cout << z << endl;       // z ist undefiniert
    int x1 = maxwert(y, x); // Anweisung enthält kein z!
    cout << z << endl;       // vermutlich anderer Wert!
}
```

Fehler! Begründung: Es wird eine Referenz auf eine *lokale* Variable zurückgegeben, die nicht mehr definiert ist und deren Speicherplatz früher oder später überschrieben wird. Korrekt wäre es, nicht die Referenz, sondern eine Kopie des Objekts zurückzugeben (Rückgabetyp `int` statt `int&`):

```
int maxwert(int a, int b) { ... }
```

Eine weitere Möglichkeit `int& maxwert(int& a, int& b) {...}` ist nicht empfehlenswert. Sie funktioniert zwar im obigen Programmbeispiel, erlaubt aber keine konstanten Argumente wie zum Beispiel in einem Aufruf `z = maxwert(23,y)`. Eine Konstante hat keine Adresse, weil der Compiler den Wert direkt in das Compilationsergebnis eintragen kann, ohne sich auf eine Speicherstelle zu beziehen.

3.2.4 Vorgegebene Parameterwerte und variable Parameterzahl

Funktionen können mit variabler Parameteranzahl aufgerufen werden. In der Deklaration des Prototypen werden für die nicht angegebenen Parameter *vorgegebene Werte* (englisch *default values*) spezifiziert. Der Vorteil liegt *nicht* in der ersparten Schreibarbeit, weil die Standardparameter nicht angegeben werden müssen! Eine Funktion kann um verschiedene Eigenschaften *erweitert* werden, die durch weitere Parameter nutzbar gemacht werden. Die Programme, die die alte Version der Funktion benutzen, sollen aber weiterhin wartbar und übersetzbar sein, ohne dass jeder Funktionsaufruf geändert werden muss. Nehmen wir an, dass ein Programm eine Funktion `adressenSortieren()` zum Beispiel aus einer firmenspezifischen Bibliothek benutzt. Die Funktion sortiert eine Adressendatei alphabetisch nach Nachnamen. Der Aufruf sei

```
// Aufruf im Programm1
adressenSortieren(adressdatei);
```

Die Sortierung nach Postleitzahlen und Telefonnummern wurde später benötigt und nachträglich eingebaut. Der Aufruf in einer neuen Anwendung könnte lauten:

```
// anderes, NEUES Programm2
enum Sortierkriterium {Nachname, PLZ, Telefon};
adressenSortieren(adressdatei, PLZ);
```

Das alte Programm1 soll ohne Änderung übersetzbar sein. Durch den Funktionsaufruf mit unterschiedlicher Parameterzahl ist dies möglich. Der Vorgabewert wäre hier `Nachname`. Die Parameter mit Vorgabewerten erscheinen in der Deklaration *nach* den anderen Parametern. Programmbeispiel:

Listing 3.8: Vorgegebene Parameter

```
// cppbuch/k3/preis.cpp
#include<iostream>
#include<string>
using namespace std;

// Funktionsprototyp 2. Parameter mit Vorgabewert:
void preisAnzeige(double preis,
                  const string& waehrung = "Euro");

// Hauptprogramm
int main() {
     // zwei Aufrufe mit unterschiedlicher Parameterzahl :
     preisAnzeige(12.35);  // vorgegebener Parameter wird eingesetzt
     preisAnzeige(99.99, "US-Dollar");
}

// Funktionsimplementation
void preisAnzeige(double preis, const string& waehrung) {
      cout << preis << ' ' << waehrung << endl;
}
```

Ausgabe des Programms: `12.35 Euro` und `99.99 US-Dollar`

Falls der Preis in € angezeigt werden soll, braucht keine Währung genannt zu werden. Dies ist der Normalfall. Andernfalls ist die Währungsbezeichnung als Zeichenkette im zweiten Argument zu übergeben.

3.2.5 Überladen von Funktionen

Funktionen können überladen werden. Deswegen darf für gleichartige Operationen mit Daten verschiedenen Typs *derselbe Funktionsname* verwendet werden, obwohl es sich nicht um dieselben Funktionen handelt. Ein Programm wird dadurch besser lesbar. Die Entscheidung, welche Funktion von mehreren Funktionen gleichen Namens ausgewählt wird, hängt vom Kontext, also der Umgebungsinformation ab: Der Compiler trifft die richtige Zuordnung anhand der *Signatur* der Funktion, die er mit dem Aufruf vergleicht. Die Signatur besteht aus der Kombination des Funktionsnamens mit Reihenfolge und Typen der Parameter. Beispiel:

Listing 3.9: Überladen von Funktionen

```
// cppbuch/k3/ueberlad.cpp
#include<iostream>
using namespace std;

double maximum(double x, double y) {
   return x > y ? x : y; // Bedingungsoperator siehe Seite 67
}

// zweite Funktion gleichen Namens, aber unterschiedlicher Signatur
int maximum(int x, int y) {
   return x > y ? x : y;
}

int main() {
   double a = 100.2;
   double b = 333.777;
   int c = 1700;
   int d = 1000;
   cout << maximum(a,b) << endl; // Aufruf von maximum(double, double)
   cout << maximum(c,d) << endl; // Aufruf von maximum(int, int)
}
```

Der Compiler versucht, nach bestimmten Regeln immer die beste Übereinstimmung mit den Parametertypen zu finden:

```
const float E = 2.7182, PI = 3.14159;
cout << maximum(E, PI);
```

führt zum Aufruf von `maximum(double, double)`, und `maximum(31,'A')` zum Aufruf von `maximum(int, int)`, weil `float`-Werte in `double`-Wert konvertiert und der Datentyp `char` auf `int` abgebildet wird. Dies gelingt nur bei einfachen und zueinander passenden Datentypen und eindeutigen Zuordnungen. Der Aufruf `maximum(3.1, 7)` ist nicht eindeutig interpretierbar. Das erste Argument spricht für `maximum(double, double)`, das zweite für `maximum(int, int)`. Der Compiler kann sich nicht entscheiden und erzeugt eine Fehler-

meldung. Es bleibt einem natürlich unbenommen, selbst eine Typumwandlung vorzunehmen. Die Aufrufe

```
cout << maximum(3.1, static_cast<float>(7));
cout << maximum(3.1, static_cast<double>(7));
int x = 66;
char y = static_cast<char>(x);
cout << maximum(static_cast<int>(0.1), static_cast<int>(y));
```

sind daher zulässig und unproblematisch, abgesehen vom Informationsverlust durch die Typumwandlung in der letzten Zeile. Die Umwandlung nach `int` schneidet die Nachkommaziffern ab. Der Typ `char` kann vorzeichenbehaftet (`signed`) sein. In diesem Fall ergibt die interne Umwandlung von `int` in `char` nur dann ein positives Ergebnis, wenn nach dem Abschneiden der höherwertigen Bits das Bit Nr. 7 nicht gesetzt ist, wobei die Zählung mit dem niedrigstwertigen Bit beginnt, das die Nr. 0 trägt:

```
// Voraussetzung: char ist signed char. Aufgerufen wird maximum(int, int).
cout << maximum(-1000, static_cast<char>(600)); // ergibt 88
cout << maximum(-1000, static_cast<char>(128)); // ergibt -128
cout << maximum(-1000, static_cast<char>(129)); // ergibt -127 usw.
```

Das Abschneiden der höherwertigen Bits wird deutlich, wenn man zum Beispiel 600 als $2^9 + 88$ schreibt. In den Abschnitten 4.3.4 und 9.4 werden wir eine Möglichkeit zur benutzerspezifischen Typumwandlung für beliebige Datentypen kennenlernen.

Gemäß der Regel, dass ein C++-Name, gleichgültig ob Funktions- oder Variablenname, alle gleichen Namen eines äußeren Gültigkeitsbereichs überdeckt, funktioniert das oben beschriebene Überladen nur innerhalb *desselben* Gültigkeitsbereichs. Ein Test:

```
#include<iostream>
using namespace std;

void f(char c) {
    cout << "f(char)   c=" << c << endl;
}

void f(double x) {
    cout << "f(double) x=" << x << endl;
}

int main() {   // neuer Gültigkeitsbereich (Block) beginnt
   void f(double);                    // ***
   f('a');
}
```

Die Deklaration innerhalb eines anderen Gültigkeitsbereichs führt dazu, dass `f` mit dem `char`-Parameter nicht mehr sichtbar ist. Es wird `f(double)` ausgeführt, wobei das Zeichen 'a' in eine `double`-Zahl umgewandelt wird. Machen Sie die Gegenprobe, indem Sie die `***`-Zeile löschen! Der Compiler findet sich dann wieder zurecht, und `f(char)` wird ausgeführt.

3.2.6 Funktion main()

`main()` ist eine spezielle Funktion. Jedes C++-Programm startet definitionsgemäß mit `main()`, sodass `main()` in jedem C++-Programm genau einmal vorhanden sein muss. Die

Funktion ist nicht vom Compiler vordefiniert, ihr Rückgabetyp soll `int` sein und ist ansonsten aber implementationsabhängig. `main()` kann nicht überladen oder von einer anderen Funktion aufgerufen werden. Die zwei folgenden Varianten sind mindestens gefordert und werden daher von jedem Compilerhersteller zur Verfügung gestellt:

```
// erste Variante
int main() {
    ...
    return 0;    // Exit-Code
}
```

```
// zweite Variante
int main( int argc, char* argv[]) { // siehe Text
    ...
    return 0;    // Exit-Code
}
```

Die zweite Variante verwendet *Zeiger* (`char*`) und C-Arrays, die in Kapitel 5 besprochen werden. Die Auswertung der Argumente wird bis dahin zurückgestellt (ab Seite 207).

Es bleibt dem Hersteller eines Compilers überlassen, ob er weitere Versionen mit erweiterten Argumentlisten anbietet. Die mit `return` zurückgegebene Zahl wird an die aufrufende Umgebung des Programms übergeben. Damit kann bei einer Abfolge von Programmen ein Programm den Rückgabewert des Vorgängers abfragen, zum Beispiel zur gezielten Reaktion auf Fehler. Wenn irgendwo im Programm die im Header `<cstdlib>` deklarierte Funktion `void exit(int)` aufgerufen wird, ist die Wirkung dieselbe, wobei jedoch der aktuelle Block verlassen wird, ohne automatische Objekte (Stackvariable) freizugeben. Der Argumentwert von `exit()` ist dann der Rückgabewert des Programms. `return` darf in `main()` weggelassen werden; dann wird automatisch 0 zurückgegeben.

3.2.7 Beispiel Taschenrechnersimulation

Um ein etwas umfangreicheres Beispiel mit Funktionen zu geben, wird ein Taschenrechner simuliert, eine beliebte Aufgabe (siehe auch [Mar], nach dem dieses Beispiel entworfen wurde, oder etwas komfortabler und aufwendiger [Str, Kapitel 6.1]). Die hier verwendete und nur kurz beschriebene Methode des *rekursiven Abstiegs* ermöglicht es, auf elegante und einfache Art beliebig verschachtelte Ausdrücke auszuwerten. In [ALSU] können fortgeschrittene Interessierte ausführliche Erläuterungen der Methode finden.

Syntax eines mathematischen Ausdrucks

Zunächst sei die Syntax eines mathematischen Ausdrucks wie zum Beispiel (13 + 7) ∗ 5 – (2 ∗ 3 + 7)/(–8) beschrieben, wobei der Schrägstrich das Zeichen für die ganzzahlige Division sein soll. Ein *Ausdruck* wird als *Summand* oder *Summe von Summanden* aufgefasst, die sich ihrerseits aus *Faktoren* zusammensetzen. Durch die zuerst auszuführende Berechnung der Faktoren ist die Prioritätsreihenfolge »Punktrechnung vor Strichrechnung« gewährleistet. Ein *Faktor* kann eine *Zahl* oder ein *Ausdruck in Klammern* sein. Die Verschachtelung mit Klammern sei beliebig möglich. Eine *Zahl* besteht aus einer oder mehreren Ziffern. Eine Ziffer ist eines der Zeichen 0 bis 9. Zur Vereinfachung sei ein mathematischer Ausdruck auf ganze Zahlen und die vier Grundrechenarten beschränkt.

Leerzeichen sind im Ausdruck nicht erlaubt. Abbildung 3.7 zeigt die Syntax eines Ausdrucks.

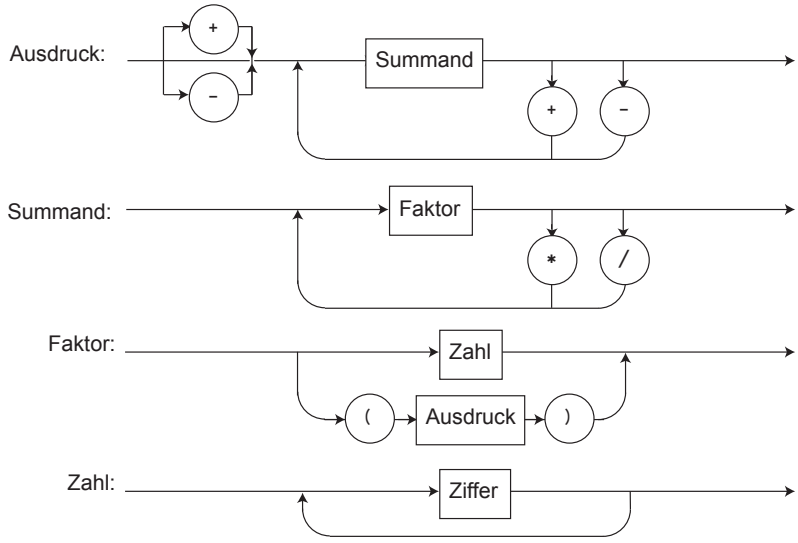

Abbildung 3.7: Syntaxdiagramm für einen mathematischen Ausdruck

Aus dem Syntaxdiagramm wird die indirekte Rekursion deutlich: Ausdruck ruft Summand, Summand ruft Faktor, Faktor ruft Ausdruck etc. Da jeder arithmetische Ausdruck endlich ist, endet die Rekursion irgendwann. Die Auflösung eines Ausdrucks bis zum Rekursionsende nennt man *rekursiver Abstieg*. Abbildung 3.8 zeigt den Ableitungsbaum des Ausdrucks (12 + 3) ∗ 4, in dem die äußeren Elemente (die »Blätter« des »Baums«) die Zahl- oder Operatorzeichen sind. Die inneren Elemente, durch Kästen dargestellt, sind noch aufzulösen.

Abbildung 3.8 ist wie folgt zu interpretieren: Der *Ausdruck* ist ein *Summand*, nämlich (12 + 3) ∗ 4, bestehend aus dem *Faktor* (12 + 3), dem Multiplikationszeichen ∗ und dem *Faktor* 4. Die Faktoren werden dem Syntaxdiagramm entsprechend weiter ausgewertet. Der erste Faktor zum Beispiel ist ein durch runde Klammern () begrenzter *Ausdruck* usw.

Wir gehen so vor, dass wir das obige Syntaxdiagramm 3.7 direkt in ein Programm transformieren. Rekursive Syntaxstrukturen werden dabei auf rekursive Strukturen im Programm abgebildet. Ziel:

- Berechnung beliebig verschachtelter arithmetischer Ausdrücke, wobei hier zur Vereinfachung nur ganze Zahlen zugelassen sein sollen.
- Leerzeichen sind nicht erlaubt; keine aufwendige Syntaxprüfung
- Vorrangregeln sollen beachtet werden.

Wie kann man nun ein Programm schreiben, das die gewünschte Berechnung liefert? Zunächst ein paar Vorgaben:

a) Das Programm soll ein Promptzeichen ›› ausgeben und dann die Eingabe des Ausdrucks erwarten.

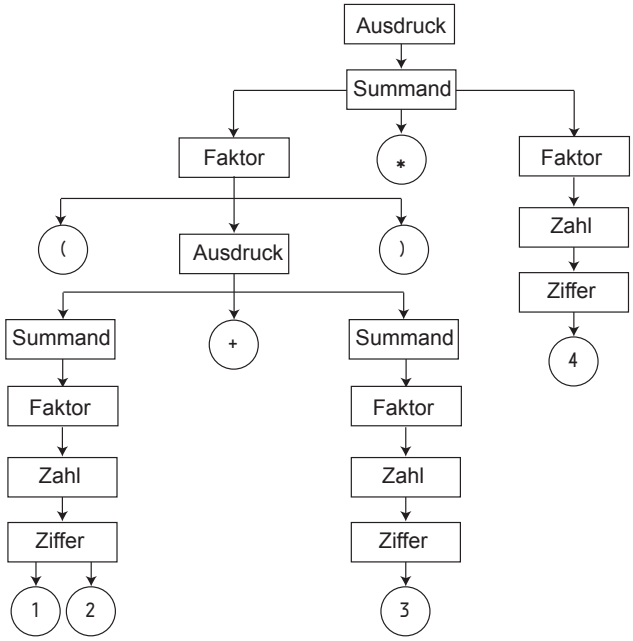

Abbildung 3.8: Ableitungsbaum für (12+3)*4

b) Der Ausdruck wird mit ENTER abgeschlossen. Anschließend wird das Ergebnis ausgegeben.

c) a) und b) sollen wiederholt werden, bis 'e' als Endekennung eingegeben wird.

Damit kann das Hauptprogramm geschrieben werden:

```
int main() {
   char ch;
   do {
      cout << "\n>>";
      cin.get(ch);
      if (ch != 'e') {
         cout << ausdruck(ch);
      }
   } while(ch != 'e');
}
```

`cin.get(ch)` ist eine vordefinierte Prozedur, die das nächste Zeichen aus dem Tastaturpuffer, in den das Betriebssystem die eingegebenen Zeichen der Reihe nach abgelegt hat, einliest, wie auf Seite 94 beschrieben. Mit jedem weiteren Aufruf von `cin.get()` wird ein weiteres Zeichen geholt. `cin >> ch` wird nicht gewählt, weil ENTER dann ignoriert wird. Nachdem der Rahmen abgesteckt ist, geht es nun an den Kern des Problems: `ausdruck()` ist offensichtlich eine Funktion, die die eingegebene Zeichenkette auswertet und einen `int`-Wert, nämlich das Ergebnis, zurückgibt. Wir haben es uns einfach gemacht und die ganze Arbeit an die Funktion delegiert. Wie kann die Funktion `ausdruck()` aussehen?

Dazu ein paar Vorüberlegungen: Laut Syntaxdiagramm ist *Ausdruck* entweder

a) –*Summand*

b) +*Summand* oder einfach nur

c) *Summand*

sowie mögliche zusätzliche, durch + oder – getrennte weitere Summanden. Man kann also die Zeichen + oder – gegebenenfalls überlesen und dann `ausdruck()` den Wert einer Funktion `summand()` zuweisen, die den Rest der Zeichenkette auswertet und ein `int`-Ergebnis zurückgibt. Das ermöglicht es `ausdruck()`, seinerseits einen Teil der Arbeit an `summand()` zu delegieren. Einer schiebt es auf den anderen, wie im richtigen Leben! Daraus ergibt sich die Vorgehensweise:

- Aus dem Syntaxdiagramm leitet sich die folgende syntaktische Konstruktion ab, wobei das aktuelle Zeichen überlesen wird, wenn es *nicht* zu dieser Konstruktion gehört. *Andernfalls* ist das Zeichen das erste zu analysierende Zeichen der syntaktischen Folgekonstruktion und wird der zugehörigen Funktion übergeben.
- Die Folgekonstruktion wird als Funktion aufgerufen und verhält sich wie der Aufrufer. Wenn die Funktion auf ein Zeichen stößt, das *nicht* zu der zugehörigen syntaktischen Konstruktion passt, wird es an den Aufrufer *zurückgegeben*.

Beispiel :

Ausdruck: Aus dem Syntaxdiagramm ergibt sich *Summand* als folgende syntaktische Konstruktion. '–' oder '+' müssen gegebenenfalls übersprungen werden, weil sie kein Element von *Summand* sind.

Summand wird anschließend genauso behandelt wie Ausdruck usw. Die Rekursion muss wegen der endlichen Länge eines Ausdrucks irgendwann ein Ende haben.

Nach diesen Vorbemerkungen bilden wir das Syntaxdiagramm direkt auf ein C++-Programm ab, wobei dem syntaktischen Term *Ausdruck* eine Funktion mit dem Namen `ausdruck()` zugeordnet wird. Eine Schleife wird im Diagramm in eine `while()`-Anweisung transformiert. Die Entsprechung zwischen dem Syntaxdiagramm auf Seite 117 und dem Programmcode ist offensichtlich. Die Variable `c` wird als Referenz übergeben, damit bei Ende der Funktion der neue Wert der aufrufenden Funktion zur weiteren Analyse zur Verfügung steht.

```
long ausdruck(char& c) {      // Übergabe per Referenz!
   long a;                    // Hilfsvariable für Ausdruck
   if(c == '–') {
      cin.get(c);             // - im Eingabestrom überspringen
      a = –summand(c);        // Rest an summand() übergeben
   }
   else {
      if(c == '+')
         cin.get(c);          // + überspringen
      a = summand(c);
   }
   while(c == '+' || c == '–')
      if(c == '+') {
         cin.get(c);          // + überspringen
         a += summand(c);
      }
```

```
        else {
           cin.get(c);          // - überspringen
           a -= summand(c);
        }
    return a;
}
```

Summand wird auf die gleiche Art wie `ausdruck()` gebildet:

```
long summand(char& c) {
    long s = faktor(c);
    while(c == '*' || c == '/')
        if(c == '*') {
            cin.get(c);         // * überspringen
            s *= faktor(c);
        }
        else {
            cin.get(c);         // / überspringen
            s /= faktor(c);
        }
    return s;
}
```

Auch Faktor wird auf ähnliche Art konstruiert:

```
long faktor(char& c) {
    long f;
    if(c == '(') {
        cin.get(c);                    // ( überspringen
        f = ausdruck(c);
        if(c != ')' )
           cout << "Rechte Klammer fehlt!\n";  // *** siehe Text unten
        else cin.get(c);               // ) überspringen
    }
    else
       f = zahl(c);
    return f;
 }
```

Nun bleibt nur noch die Funktion zur Analyse einer Ziffernfolge:

```
long zahl(char& c) {
    long z = 0;
    // isdigit() ist eine Funktion (genauer: ein Makro), das zu true ausgewertet wird,
    // falls c ein Zifferzeichen ist. Die Verwendung setzt #include<cctype> voraus.
    while(isdigit(c)) { // d.h. c >= '0' && c <= '9'
        // Zur Subtraktion von '0' siehe Seite 54.
        z = 10*z + long(c-'0'); // implizite Typumwandlung
        cin.get(c);
    }
    return z;
}
```

Letztlich ist die Umsetzung einer Syntax in ein Programm reine Fleißarbeit, wenn man weiß, wie es geht. Deswegen gibt es dafür Werkzeuge wie die Programme *lex* und *yacc* oder *bison*. Nun haben wir alle Bausteine zusammen, die zur Auswertung eines beliebig verschachtelten arithmetischen Ausdrucks nötig sind. Es bleibt dem Leser überlassen, das Programm zu vervollständigen, einschließlich Trennung von Prototypen und Definitionen, und es zum Laufen zu bringen. Erweiterungen können leicht eingebaut werden, um Leerzeichen an syntaktisch sinnvollen Stellen zu erlauben oder Hinweise auf Syntaxfehler auszugeben, wie in der mit *** markierten Zeile gezeigt wird. Falls doch noch Verständnisschwierigkeiten auftreten sollten, spielt man am besten selbst »Computer«, indem man einen Ausdruck Schritt für Schritt am Schreibtisch dem Programm folgend abarbeitet.

3.2.8 Spezifikation von Funktionen

Eine Funktion erledigt eine Teilaufgabe. Es ist sinnvoll, diese Teilaufgabe im Funktionskopf als Kommentar zu spezifizieren. Dazu gehören Annahmen über die Importschnittstelle (Eingabedaten, zum Beispiel Wertebereich), die Fehlerbedingungen, die Exportschnittstelle (Ausgabedaten). Die Bedingung, die ein Eingabeparameter erfüllen muss, damit die Funktion richtig arbeitet, nennt man Vorbedingung. Der Zustand eines Programms nach Abarbeitung der Funktion wird Nachbedingung genannt. Die Spezifikation ist für den Benutzer einer Funktion von Interesse.

Wie die Aufgabe gelöst wird, sollte im Funktionskopf nicht beschrieben werden, um die Möglichkeit einer späteren Änderung der Implementierung nicht einzuschränken, zum Beispiel einen langsamen durch einen schnelleren Algorithmus zu ersetzen. Das schließt nicht aus, dass innerhalb der Funktion manche Stellen kommentierend erklärt werden. Die Interna einer Funktion sind nur für Entwickler von Interesse, nicht aber für den Benutzer.

Eine Spezifikation kann als *Vertrag* zwischen Aufrufer und Funktion aufgefasst werden. Die Funktion gewährleistet die Nachbedingung, wenn der Aufrufer die Vorbedingung einhält. Die Analogie zu einem Vertrag zwischen Kunde und Softwarehaus liegt auf der Hand. Eine Vertiefung des Themas »Design by Contract« ist in [Mey] zu finden.

Die Spezifikation sollte mit in eine Header-Datei übernommen werden. Eine Header-Datei soll unter anderem die Prototypen von Funktionen enthalten (siehe folgender Abschnitt 3.3). Mehr zur Dokumentation von Programmen erfahren Sie in Abschnitt 19.1.

Übungen

3.5 Schreiben Sie eine Funktion `void str_umkehr(string& s)`, die die Reihenfolge der Zeichen im String `s` umkehrt.

3.6 Vervollständigen Sie das Beispiel in Abschnitt 3.2.7 und bringen Sie es zum Laufen.

3.7 Schreiben Sie eine Funktion `istAlphanumerisch(const string& text)`, die `true` zurückgibt, wenn `text` nur Buchstaben und Ziffern enthält, andernfalls `false`.

3.3 Modulare Programmgestaltung

C++ bietet eine große Flexibilität in der Organisierung eines Softwaresystems. Die Erfahrung lehrt, dass die Aufteilung eines großen Programms in einzelne, getrennt übersetzbare Dateien, die zusammengehörige Programmteile enthalten, sinnvoll ist. Folgender Aufbau empfiehlt sich:

- Die Standard-Header haben die uns schon bekannte Form `<headername>`. Darüber hinaus kann es eigene (oder andere) Header-Dateien geben, die typischerweise die Endung **.h* [oder auch **.hpp, *.hxx*, je nach Computer- oder Entwicklungssystem] im Dateinamen haben. Sie enthalten Konstanten, Schnittstellenbeschreibungen wie Klassendeklarationen, Deklarationen globaler Daten und Funktionsprototypen.
- Implementationsdateien enthalten die Implementation der Klassen und den Programmcode der Funktionen (Endung im Dateinamen: **.cpp* [auch **.cxx, *.cc, *.c*]).
- Main-Datei. Sie enthält das Hauptprogramm `main( )`.

Wirkung von #include

Damit eine Datei einzeln für sich übersetzbar ist, müssen Konstanten, Klasseninterfaces und Funktionsprototypen bekannt sein. Das wird erreicht durch das Einschließen der Header-Dateien mit der Präprozessordirektive `#include` "*filename.h*". Präprozessordirektiven werden von einem dem eigentlichen Compiler vorgeschalteten Präprozessor verarbeitet, der auch die Kommentare ausblendet.

Anstelle von *filename.h* ist natürlich der richtige Name einzutragen. Die Datei *filename.h* wird im aktuellen Verzeichnis gesucht und an dieser Stelle eingelesen. Die eingelesene Datei kann selbst auch `#include`-Direktiven enthalten, die genauso verarbeitet werden. Weiteres zu diesen Direktiven, insbesondere auch zur Form `#include<Header>` (keine Anführungszeichen als Begrenzer), ist auf Seite 128 zu finden.

Zwei Strukturen, die in den nächsten Abschnitten behandelt werden, sind möglich:

- die Steuerung der Übersetzung nur durch `#include`-Anweisungen;
- das Einbinden von bereits vorübersetzten Programmteilen; besonders sinnvoll bei großen Programmen, von denen einige Teile schon stabil laufen.

3.3.1 Steuerung der Übersetzung nur mit #include

Nehmen wir an, dass das `main`-Programm (Datei *meinprog.cpp*) die Funktionen `func_a1()` und `func_a2()` aus der Datei *a.cpp* und eine Funktion `func_b()` aus der Datei *b.cpp* benutzt. Mit `#include` werden diese Dateien in *meinprog.cpp* eingeschlossen. Nur bei sehr *kleinen* Programmen ist dieses Verfahren ausreichend. Im Normalfall gelten jedoch die Empfehlungen des folgenden Abschnitts. `#include "a.cpp"` wirkt, als ob an der Stelle der `#include`-Anweisung die Datei *a.cpp* selbst hingeschrieben worden wäre:

```
// nicht empfehlenswert! (»quick and dirty«)
#include "a.cpp"
#include "b.cpp"
int main() {
   func_a1( ); // Funktionsaufrufe
```

```
    func_a2( );
    func_b( );
}
```

3.3.2 Einbinden vorübersetzter Programmteile

Bei größeren und sehr großen Programmen ist es sinnvoll, Schnittstellen (Funktionsprototypen und Klassen) und Implementationen (Programmcode) zu trennen. Daher nehmen wir ferner an, dass die Schnittstellen in den Header-Dateien *a.h* und *b.h* abgelegt sind.

Um die automatische Prüfung der Schnittstellen durch den Compiler zu ermöglichen, werden die Header-Dateien mit `#include` in allen Dateien eingeschlossen, die diese Schnittstellen verwenden. Mit den Header-Dateien kann jede Datei einzeln übersetzt werden. Wenn es Änderungen gibt, müssen nur noch die davon betroffenen Dateien neu compiliert werden. Die Dateien könnten folgenden Inhalt haben:

```
// a.h
void func_a1();
void func_a2();
```

```
// a.cpp
#include "a.h"
void func_a1() {
     // Programmcode zu func_a1
}

void func_a2() {
    // Programmcode zu func_a2
}
```

```
// b.h
void func_b();
```

```
// b.cpp
#include "b.h"
void func_b() {
    // Programmcode zu func_b
}
```

In diesem sehr einfachen Beispiel ist es nicht zwingend, *a.h* in *a.cpp* und *b.h* in *b.cpp* einzubinden, weil die **.cpp*-Dateien keine Informationen verwenden, die nur in den **.h*-Dateien vorkommen. Das ist jedoch nicht die Regel, wie wir später sehen werden.

```
// meinprog.cpp
#include "a.h"
#include "b.h"

int main() {
    func_a1( );
    func_a2( );
    func_b( );
}
```

Die erste Zeile gibt jeweils den Namen der Datei im Kommentar an. Wir nehmen an, dass *a.cpp* und *b.cpp* bereits übersetzt sind, die Dateien *a.o* und *b.o* also existieren. In *meinprog.cpp* sei eine Änderung notwendig gewesen. Den Übersetzungsablauf zeigt Abbildung 3.9. Der *lib*-Anteil unten links in der Abbildung enthält die benötigten Systemfunktionen.

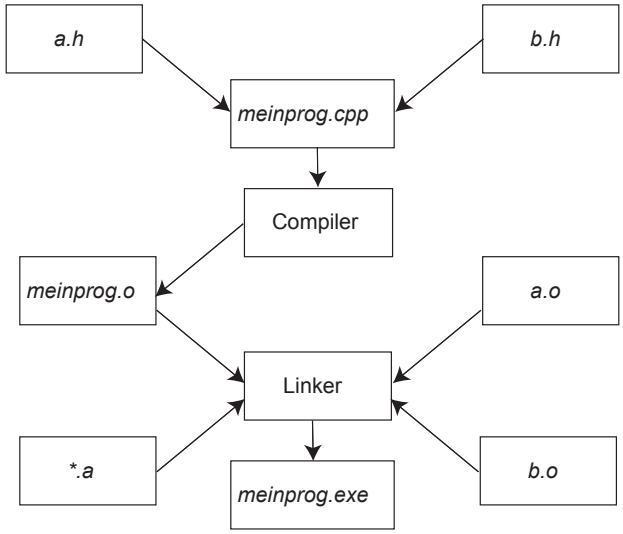

Abbildung 3.9: Compilations- und Link-Ablauf

Die Steuerung der Übersetzung und des Bindens ist je nach System unterschiedlich. Üblich sind Make-Dateien, auch Makefiles genannt, in denen die Reihenfolge und die Abhängigkeiten der Dateien beschrieben sind, sodass bei Änderungen nur die davon betroffenen neu übersetzt werden müssen. Make-Dateien werden in Kapitel 17 beschrieben. Eine andere Methode mit gleicher Wirkung sind sogenannte »Projekte«, in denen die zu übersetzenden und zu bindenden Dateien angegeben werden. Wenn eine ganze Reihe gut getesteter Programmbausteine zu einem Thema vorliegen, können die zugehörigen **.o*- (oder **.obj*)-Dateien in einer Bibliotheks- oder **.a*- (oder **.lib*)-Datei zusammengefasst werden. Die Konzepte

- Trennung von Schnittstellen und Implementation und
- Gruppierung zusammengehöriger Funktionen und Klassen zu Bibliotheksmodulen

sind Standard in allen größeren Programmierprojekten.

3.3.3 Dateiübergreifende Gültigkeit und Sichtbarkeit

Die Speicherklasse einer Variablen wird unter anderem durch die Worte `static`, `extern`, und `mutable` bestimmt. Mit `static` und `extern` werden Sichtbarkeit und Lebensdauer von Variablen eingestellt. `mutable` kann erst in Abschnitt 4.5 erläutert werden.

Alle nicht globalen und nicht-`static`-Variablen sind sogenannte *automatische* Variablen. Automatische Variablen werden bei Betreten eines Blocks mit undefiniertem Inhalt an-

gelegt, sofern sie nicht explizit initialisiert werden. Sie haben dann also *nicht* den Wert 0. Bei Verlassen des Blocks werden sie wieder zerstört (siehe Abschnitt 1.7).

extern bei Variablen

Variablen, die außerhalb von `main()` und jeglicher anderer Funktion definiert sind, heißen *global*. Sie sind in *allen* Teilen eines Programms gültig, auch in anderen Dateien. Eine globale Variable muss nur in einer anderen Datei als `extern` deklariert werden, um dort benutzbar zu sein.

```
// datei1.cpp
int global;              // Deklaration und Definition
int main() {
   global = 17;
}
```

```
// datei2.cpp
extern int global;       // Deklaration, aber keine Definition
void func1( ) {
    global = 123;
}
```

datei2.cpp ist für sich allein übersetzbar. Das Schlüsselwort `extern` sagt dem Compiler, dass eine Variable irgendwo anders definiert ist. Erst beim Binden, auch Linken genannt, wird die Referenz aufgelöst.

Tipp

Globale Variablen und Objekte sollen vermieden werden, weil sie für alle zugreifbar sind. Ursachen von mit ihnen verbundenen Fehlern sind daher schwer lokalisierbar.

static

Um den Gültigkeitsbereich von Variablen und Funktionen auf *eine Datei zu beschränken*, wird manchmal das Schlüsselwort `static` eingesetzt. Diese Bedeutung von `static` ist nicht mit der aus Abschnitt 3.1.2 zu verwechseln. In C++ kommt es mehrfach vor, dass Schlüsselwörter oder Operatoren mehrere Bedeutungen haben, die sich im konkreten Fall aus dem Kontext ergeben. Wegen der mehrfachen Bedeutung von `static` kann es aber zu Missverständnissen kommen, sodass die Verwendung für dateiglobale Variablen nicht empfohlen wird.

`static`-Variablen werden stets mit 0 initialisiert, wobei 0 gegebenenfalls in den passenden Datentyp umgewandelt wird. `static` dient zur Vermeidung von Namenskonflikten zwischen verschiedenen Dateien. Der gewünschte Zweck, die Gültigkeit einer Variable auf eine Datei zu beschränken, lässt sich ohne die Verwendung von `static` mit einem anonymen Namespace erreichen:

```
namespace {      // anonymer Namespace
  int global;
}
```

```
int main() {
   global = 17;
}
```

Anonyme Namespaces in verschiedenen Übersetzungseinheiten sind verschieden und nicht von außen zugreifbar. *Innerhalb* einer Übersetzungseinheit ist ein anonymer Namespace jedoch bekannt. Die Wirkung ist, als ob

```
namespace XXXX {
  int global;
}

using namespace XXXX;

int main() {
   global = 17;
}
```

geschrieben worden wäre, wobei XXXX irgendein beliebiger Name ist, der sonst nirgendwo in demselben Sichtbarkeitsbereich vorkommt. Mit dieser Änderung in *datei1.cpp* würden beide Dateien anstandslos übersetzt, aber es würde einen Linker-Fehler bei *Datei2.o* geben, weil jetzt die Gültigkeit von `global` nur auf *datei1.cpp* beschränkt ist. Alles, was der Compiler in einem Durchgang liest, ist eine *Übersetzungseinheit*. Man sagt, dass die nur innerhalb einer Übersetzungseinheit gültigen Variablen und Funktionen *intern gebunden* werden (internes Linken (englisch *internal linkage*)), während globale Variablen und Funktionen *extern gebunden* werden (externes Linken (englisch *external linkage*)).

extern bei Konstanten

Auf Dateiebene (außerhalb von `main()`) definierte *Variablen* sind *global* und in anderen Dateien benutzbar, wenn sie dort als `extern` deklariert sind. Bei *Konstanten* (`const`) ist es jedoch anders: Konstanten sind nur in der Definitionsdatei sichtbar! Sollen Konstante anderen Dateien zugänglich gemacht werden, müssen sie als `extern` deklariert und initialisiert werden:

```
// datei1.cpp
extern const float PI = 3.14159; // Deklaration und Definition

// datei2.cpp
// Deklaration ohne Definition
extern const float PI; // ohne Initialisierung
```

Ohne `extern` in *datei1.cpp* wäre der Geltungsbereich von `PI` auf *datei1.cpp* beschränkt.

3.3.4 Übersetzungseinheit, Deklaration, Definition

Der Text, den der Compiler in einem Durchgang verdauen muss, heißt *Übersetzungseinheit*. In diesem Sinn geht es bei der Steuerung der Übersetzung mit `#include` in Abschnitt 3.3.1 nur um eine einzige Übersetzungseinheit. Große Programme werden jedoch in viele Übersetzungseinheiten gegliedert, um sie handhabbar zu machen. Insbesondere müssen bereits übersetzte und funktionstüchtige Teile nicht immer wieder neu übersetzt werden

(Abschnitt 3.3.2). Zum Verständnis ist es wichtig, klar zwischen den Begriffen *Deklaration* und *Definition* zu unterscheiden:

- Eine *Deklaration* führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung.
- Eine Deklaration ist auch eine *Definition*, wenn *mehr als nur der Name eingeführt wird*, zum Beispiel wenn Speicherplatz für Daten oder Code angelegt oder die innere Struktur eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt.

Der Verdeutlichung dienen einige Beispiele. Folgende Deklarationen sind gleichzeitig Definitionen:

```
int a;                          // Speicherplatz für a wird angelegt
extern const float PI = 3.14159; // Speicherplatz für PI wird angelegt
int f(int x) { return x*x;}     // enthält Programmcode
struct meinStrukt {             // definiert meinStrukt, d.h.
   int c;                       // beschreibt die innere Struktur
   int d;                       // beschreibt die innere Struktur
};
meinStrukt X;                   // Speicherplatz für X wird angelegt
enum meinEnum { li, re };       // definiert meinEnum
meinEnum Y;                     // Speicherplatz für Y wird angelegt
```

Die folgenden Zeilen sind Deklarationen, aber *keine* Definitionen:

```
extern int a;
extern const float PI;
int f(int);
struct meinStrukt;
enum meinEnum;
```

One Definition Rule

Die unter dem englischen Namen *one definition rule* bekannte Regel ist bei der Strukturierung von Programmen zu beachten: Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat *genau eine* Definition. Dabei spielt es keine Rolle, ob das Programm aus vielen oder wenigen Übersetzungseinheiten besteht, ob die Definition selbst geschrieben wurde oder von einer Programmbibliothek (englisch *library*) zur Verfügung gestellt wird. Aus der *one definition rule* ergibt sich, was in den verschiedenen Dateitypen enthalten sein sollte (mit Beispielen):

Header-Dateien (**.h*)

- Funktionsprototypen (Schnittstellen)

```
void meineFunktion(int einParameter);
```

- reine Deklaration (nicht Definition) globaler Variablen

```
extern int global;
```

- Deklaration globaler Konstanten (nicht Definition, das heißt ohne Initialisierung)

```
extern const int GLOBALE_KONSTANTE;
```

- Definition von Konstanten, die nur in der Übersetzungseinheit sichtbar sind

```
const int MAXI = 10;
```

- Definition von Datentypen wie `enum` oder `struct` (weil die **.cpp*-Dateien die Größe von Objekten dieser Datentypen kennen müssen)

```
struct Punkt {
    int x;
    int y;
};
enum Wochenende {Samstag, Sonntag};
```

Implementationsdateien (**.cpp*)

- Funktionsdefinitionen (Implementation)

```
void meineFunktion(int Parameter) {
   // ... Programmcode
}
```

- Definition globaler Objekte (nur *einmal* im ganzen Programm)

```
int global;
Punkt einPunkt;
Wochenende einWochenende;
```

- Definition und Initialisierung globaler Konstanten (nur *einmal* im Programm)

```
extern const int GLOBALE_KONSTANTE = 1;
```

Variablen, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind global. Wenn dieselbe Header-Datei von mehreren Implementationsdateien eingebunden wird, werden diese Variablen *mehrfach* angelegt – im Widerspruch zur »one definition rule«. Der Linker kann diese mehrfach angelegten Variablen gleichen Namens stillschweigend zusammenlegen oder er gibt eine Warnung oder Fehlermeldung aus, dass die Variable doppelt oder mehrfach definiert ist. Globale Variablen sollten also immer extern deklariert werden, und die Definition sollte nur in einer Übersetzungseinheit vorkommen.

Konstanten, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind nicht global und beziehen sich nur auf die Übersetzungseinheit. Wenn dieselbe Header-Datei von mehreren Implementationsdateien eingebunden wird, werden diese Konstanten entsprechend mehrfach angelegt. Falls der Compiler die Konstanten in besonderen Speicherplätzen ablegt (was durchaus nicht sein muss), bedeutet das Mehrfachanlegen zugleich Speicherplatzverschwendung.

3.3.5 Compilerdirektiven und Makros

Compilerdirektiven sind Anweisungen an den dem Compiler vorgeschalteten Präprozessor, die den Übersetzungsprozeß steuern, wie zum Beispiel `#include`. Compilerdirektiven beginnen stets mit `#` am Zeilenanfang.

#include

Bereits bekannt ist die `#include`-Anweisung (siehe Seiten 32 und 122). Die Dateispezifikation kann außer dem Dateinamen den vollständigen Pfad enthalten, wobei Verzeichnisnamen durch einen Schrägstrich / zu trennen sind. In der MS-Windows-Welt ist auch der

\ (Backslash) möglich, der Schrägstrich ist aber aus Portabilitätsgründen zu bevorzugen. Beispiele:

```
// relativer Pfad
#include"dateiname.h"
#include"../include/dateiname.h"  // .. kennzeichnet das übergeordnete Verzeichnis
```

```
// absoluter Pfad
#include"/home/users/breymann/cppbuch/include/dateiname.h"  // Unix
#include"C:/cppbuch/include/dateiname.h"                    // Windows
```

Wird die Datei im aktuellen Verzeichnis nicht gefunden, wird in den voreingestellten *include*-Verzeichnissen gesucht. Falls auch diese Suche fehlschlägt, wird versucht, die Direktive in der Standard-Header-Form zu interpretieren.

Die Standard-Header-Form ist `#include<header>`. Der Platzhalter `header` muss nicht unbedingt eine Datei sein [ISOC++]. Die bisher gängigen Implementierungen fassen `header` jedoch als Datei auf, und es wird in den voreingestellten *include*-Verzeichnissen gesucht, die Suche im aktuellen Verzeichnis entfällt. Die voreingestellten *include*-Verzeichnisse sind die zum System gehörenden *include*-Verzeichnisse, in denen zum Beispiel mit `#include<iostream>` alles Nötige zur Ein- und Ausgabe gefunden wird. Sie können aber auch eigene *include*-Verzeichnisse als voreingestellte definieren. Wenn Sie zum Beispiel das Programm *merge1.cpp* von Seite 673 mit

```
g++ merge1.cpp
```

im aktuellen Verzeichnis compilieren wollen, erhalten Sie eine Fehlermeldung, weil ein im Programm geforderter Header nicht gefunden wird. Bei Voreinstellung des *include*-Verzeichnisses mit der I-Option des Compilers verschwindet der Fehler:

```
g++ -I../../include merge1.cpp
```

#define, #ifdef, #ifndef

Es kann zu Problemen beim Übersetzen führen, wenn Header-Dateien *mehrfach* eingebunden sind, sodass sich mehrfache Definitionen ergäben. Wenn im Beispiel auf Seite 123 sowohl *a.h* als auch *b.h* eine Datei *c.h* benötigten, müssten beide Dateien die Anweisung `#include "c.h"` enthalten. Durch die `#include`-Anweisungen in *meinprog.cpp* würde also *c.h zwei*mal eingelesen. Abhilfe schaffen die Anweisungen `#if defined` (Abkürzung `#ifdef`), `#if !defined` (Abkürzung `#ifndef`), und `#define`.

```
// c.h
#ifndef C_H
#define C_H

void func_c1();
void func_c2();
enum Farbtyp {rot, gruen, blau, gelb};
#endif // C_H
```

Bedeutung:

Falls der (beliebige) Name `C_H` nicht definiert ist,
dann definiere `C_H` und akzeptiere alles bis `#endif`.

Die Wirkung des *ersten* Lesens von *c.h* als indirekte Folge von `#include "a.h"` in *meinprog.cpp* ist:

- `#ifndef C_H` liefert TRUE, weil `C_H` noch nicht definiert ist.
- `#define C_H` definiert `C_H`.
- Alles bis `#endif` wird gelesen.

Die Wirkung des *zweiten* Durchlaufs von *c.h* als indirekte Folge von `#include "b.h"` in *meinprog.cpp* ist:

- `#ifndef C_H` liefert FALSE (d.h. 0), weil `C_H` bereits definiert ist.
- Alles bis `#endif` wird ignoriert.

`#if`-Blöcke erstrecken sich nicht über Dateigrenzen. Nach `#endif` in derselben Zeile stehender Text zur Dokumentation ist nur erlaubt, wenn er als Kommentar markiert ist (siehe oben: `// C_H`). Mit `#undef` kann eine Definition rückgängig gemacht werden.

Makros mit #define

Es gibt eine weitere Bedeutung von `#define`, nämlich das Ersetzen von Makros durch Zeichenketten, wobei Parameter erlaubt sind. Mehrere Parameter werden durch Kommas getrennt. Die Makrodefinitionen

```
// nicht empfehlenswert! (Begründung folgt)
#define PI 3.14
#define schreibe cout
#define QUAD(x) ((x)*(x))
```

erlauben in einem Programm den Text

```
schreibe << PI << endl;
y = QUAD(z);
```

und würden interpretiert werden als:

```
cout << 3.14 << endl;
y = ((z)*(z));
```

Wenn ein Makro durch einen sehr langen Text ersetzt werden soll, der über mehrere Zeilen geht, ist jede Zeile mit Ausnahme der letzten mit einem \ (Backslash) abzuschließen. Es ist möglich, mit einem Makro ganze Unterprogramme für verschiedene Datentypen zu schreiben, wobei der Datentyp der Parameter ist, der dem Makro übergeben wird. Eine bessere Möglichkeit dafür sind jedoch Funktionsschablonen oder -templates, die in Abschnitt 3.4 besprochen werden.

Die Textersetzung mit `#define` sollte im Allgemeinen *nicht* verwendet werden, wenn es Alternativen gibt. Wie gefährlich Makros sein können, lässt sich schon an dem einfachen `QUAD`-Makro zeigen. Der Aufruf

```
int z = 3;
int y = QUAD(++z);
```

soll `y` das Quadrat von `z` zuweisen, nachdem `z` um 1 erhöht wurde – oder? In Wirklichkeit wird `z` *zweimal* erhöht:

```
int y = ((++z)*(++z)); // expandierter Makroaufruf
```

und das Ergebnis ist falsch. Makronamen sind zudem einem symbolischen Debugger nicht zugänglich, wie die Pseudo-Konstante `PI` im obigen Beispiel. Ferner kann auf `PI` kein Zeiger (siehe Kapitel 5) gerichtet werden. Ein weiterer Nachteil von Makros besteht in der Umgehung der Typkontrolle:

```
// Makrodefinition
#define MULT(a,b) ((a)*(b))

// Makroaufruf
int a, b = 2, c = 3;
a = MULT(b,c); // ok
a = MULT(b,"Fehler!");
```

Der Compiler bekommt das Makro durch den vorgeschalteten Präprozessor gar nicht erst zu sehen, sondern bekommt nur das Ergebnis der Makroexpansion (= Textersetzung) vorgesetzt. Deshalb sind Compilerfehlermeldungen bei Fehlern innerhalb großer Makros manchmal nicht ohne Weiteres nachvollziehbar. Eine übliche Anwendung des Makros `#define` zur Textersetzung mit Parametern ist die gezielte Ein- und Ausblendung von Testsequenzen in einem Programm. Beispiel:

```
#define TEST_EIN
#ifdef TEST_EIN
  #define TESTANWEISUNG(irgendwas) irgendwas
#else
  #define TESTANWEISUNG(irgendwas) /* nichts */
#endif
// ... irgendwelcher Programmcode

// nur im Test soll bei Fehlern eine Meldung ausgegeben werden:
TESTANWEISUNG(if(x < 0) cout << "sqrt(negative Zahl)!" << endl;)
y = sqrt(x);
// ... mehr Programmcode
```

Der Parameter ist `irgendwas`. Falls `TEST_EIN` gesetzt ist, wird beim Compilieren durch den Präprozessor überall im Programm `TESTANWEISUNG(irgendwas)` durch `irgendwas` ersetzt. Wenn nach erfolgreichem Testen des Programms alle Testanweisungen verschwinden sollen, genügt es, die Zeile `#define TEST_EIN` zu löschen oder mit `//` in einen Kommentar zu verwandeln, mit der Wirkung, dass der Präprozessor jede `TESTANWEISUNG()` durch einen Kommentar `/*nichts*/` ersetzt, der schlicht ignoriert wird. `#define`-Makros können mehrere durch Kommas getrennte Parameter enthalten. In `irgendwas` sollte kein Komma enthalten sein, weil der Präprozessor sich sonst über die falsche Parameteranzahl beschwert. Zusammengefasst hat dieses Vorgehen zwei Vorteile:

- Nach Testabschluss wird das lauffähige Programm schneller und benötigt weniger Speicher durch die fehlenden Testanweisungen.
- Die Testanweisungen können im Programm zum späteren Gebrauch stehen bleiben. Sie müssen nicht einzeln auskommentiert oder gelöscht werden.

Die Technik, durch Makros gesteuert verschiedene Dinge ein- oder auszuschließen, wird sehr gut in den Header-Dateien des *include*-Verzeichnisses des Compilers sichtbar. Schauen Sie mal nach! Diese Art der Makrobenutzung ist weit verbreitet und hat ihre Vorteile.

Es gibt jedoch eine Lösung, die nur mit den Sprachelementen von C++ auskommt (also ohne Makros, die ja vom Präprozessor verarbeitet werden):

```
const bool TEST_EIN = true;
// ... irgendwelcher Programmcode
// nur im Test soll bei Fehlern eine Meldung ausgegeben werden:
if(TEST_EIN) if(x < 0) cout << "sqrt(negative Zahl)!" << endl;
y = sqrt(x);
```

Diese Lösung hat die gleichen oben genannten Vorteile. Die einzige Voraussetzung ist, dass der Compiler »toten« Programmcode von vornherein ignoriert, falls nämlich nach Abschluss der Testphase die erste Zeile in `const bool TEST_EIN = false;` geändert und dadurch die `if`-Anweisung überflüssig wird. Dies ist für einen modernen Compiler kein Problem.

Umwandlung von Parametern in Zeichenketten

Speziell für Testausgaben ist das Makro PRINT nützlich, das das Argument mit vorangestelltem `#` in eine Zeichenkette wandelt. Ohne einen Namen oder einen Ausdruck doppelt schreiben zu müssen, hat man Text und Ergebnis auf dem Bildschirm:

```
#define PRINT(X) cout << (#X) << "= " << (X) << endl;
```

Damit kann kurz zum Beispiel

```
PRINT(int(xptr)-int(xptr2));
```

geschrieben werden anstatt

```
cout << "int(xptr)-int(xptr2) = " << int(xptr)-int(xptr2) << endl;
```

mit dem Ergebnis *int(xptr)-int(xptr2) = 4* auf dem Bildschirm.

Empfehlung für den Aufbau von Header-Dateien

Eine Möglichkeit für den Aufbau von Header-Dateien ist das folgende Schema:

```
// Dateiname: fn.h
#ifndef fn_h
#define fn_h fn_h
// hier folgen die Deklarationen
#endif // fn_h
```

Um stets eindeutige Namen zu gewährleisten, empfiehlt sich die Ableitung aus dem Dateinamen, so wie `fn_h` aus *fn.h* entstanden ist. Warum tritt aber `fn_h` doppelt auf? Sehen wir uns dazu folgendes Beispiel an:

```
// Dateiname: fn.h
#ifndef fn_h
#define fn_h

// hier folgen die Deklarationen
enum fn_h { a,b,c};                    // Fehler!
#endif // fn_h
```

Es kann sein, dass zufällig derselbe Name im nachfolgenden Programmcode auftritt, weil der Dateiname meistens mit dem Dateiinhalt zu tun hat. `#define` dient zur Textersetzung oder definiert etwas als logisch wahr. Das zweite Auftreten ist für den Compiler nicht verständlich, weil anstatt `fn_h` eine 1 (= wahr) gesehen wird. Es empfiehlt sich also, entweder Variablennamen mit der Endung `_h` zu vermeiden oder den Text durch sich selbst zu ersetzen, damit er keine Änderung erfährt: `#define fn_h fn_h`.

Natürlich vermindert bereits die Endung `_h` die Gefahr einer zufälligen Namensgleichheit. Eine Alternative ist das Hervorheben durch Großschreibung, wie in der C-Welt üblich, meistens ohne Verdopplung – so wird es in diesem Buch gehandhabt:

```
#ifndef FN_H
#define FN_H
// ... usw.
```

Namespaces sollten *nicht* mit `using` in einer Header-Datei eingeführt werden, weil sie damit in allen Dateien bekannt werden, die diese Header-Datei verwenden, und damit Namenskonflikte produzieren können (Einzelheiten siehe Abschnitt 3.6 ab Seite 141). Besser ist die qualifizierte Ansprache entsprechend der zweiten der auf Seite 60 beschriebenen Möglichkeiten. Beispiel: In einer Header-Datei sollte `cout` weder mit `using std::cout;`, noch mit `using namespace std;` eingeführt, sondern qualifiziert als `std::cout` benannt werden.

Verifizieren logischer Annahmen zur Laufzeit mit assert

Ein weiteres sehr nützliches Makro ist `assert()` zur Überprüfung logischer Annahmen, die an der Stelle des Makros gültig sein sollen. Insbesondere lassen sich die in Abschnitt 3.2.8 beschriebenen Vor- und Nachbedingungen verifizieren. Das Wort `assert()` leitet sich vom englischen Wort *assertion* ab, das auf Deutsch »Zusicherung« heißt. Zusicherungen werden mit dem Header `<cassert>` eingebunden. Beispiel:

```
#include<cassert> // enthält Makrodefinition

const int GRENZE = 100;
int index;
// .... Berechnung von index
// Test auf Einhaltung der Grenzen:
assert(index >= 0 && index < GRENZE);
```

Falls die Annahme (`index >= 0 && index < GRENZE`) nicht stimmen sollte, wird das Programm mit einer Fehlermeldung abgebrochen, die die zu verifizierende logische Annahme, die Datei und die Nummer der Zeile enthält, in der der Fehler aufgetreten ist. Eine andere Möglichkeit wäre das »Werfen einer Ausnahme«, siehe Abschnitt 8.1. `assert()` ist wirkungslos, falls `NDEBUG` vor `#include<cassert>` definiert wurde, entweder durch die Compilerdirektive `#define NDEBUG` oder durch Setzen des Compilerschalters –D, mit dem Makrodefinitionen voreingestellt werden. Anwendungsbeispiel: `g++ -DNDEBUG meinProgramm.cpp`

Werfen Sie einen Blick in die Datei *assert.h*, um die Wirkungsweise des Makros zu studieren! Der oben im `PRINT()`-Makro verwendete Präprozessoroperator `#` verwandelt die logische Annahme in eine Zeichenkette. Die innerhalb von `assert()` verwendeten vordefinierten Makros `__FILE__` und `__LINE__` werden beim Compilieren durch einen String mit

dem Dateinamen beziehungsweise durch die Zeilennummer ersetzt. Die vordefinierten Makros können Sie innerhalb selbst geschriebener Makros verwenden, ebenso wie `__DATE__` und `__TIME__`, die Datum und Uhrzeit der Übersetzung in einen String verwandeln.

Tipp

Vermeiden Sie Seiteneffekte in `assert()` und anderen Makros!

Eine in der Zusicherung aufgerufene Funktion wird bei gesetztem `NDEBUG` nicht ausgeführt! Beispiel:

```
assert(datei_oeffnen(filename) == erfolgreich); // Fehler
assert(GRENZE <= max(x,y));                     // Fehler
```

Falls `NDEBUG` definiert ist, wird die Datei nicht geöffnet, und weder wird das Maximum von `x` und `y` berechnet noch `GRENZE` mit irgendeinem Wert verglichen.

Verifizieren logischer Annahmen zur Compilationszeit mit static_assert

Die Prüfung mit `assert` geschieht zur Laufzeit. Manchmal möchte man aber bereits zur Compilationszeit bekannte Annahmen prüfen. Zum Beispiel soll `long` statt `int` eingesetzt werden, um den Zahlenbereich zu erweitern. Es ist aber systemabhängig und nicht garantiert, dass die Anzahl der Bits für `long` größer als die für `int` ist. Die Prüfung wird mit `static_assert` durchgeführt:

```
static_assert(sizeof(long) > sizeof(int), "long hat nicht mehr Bits als int!");
```

Wenn die Behauptung `sizeof(long) > sizeof(int)` falsch ist, gibt schon der Compiler die Fehlermeldung »long hat nicht mehr Bits als int!« aus. In diesem Fall bringt der Ersatz von `int` durch `long` gar nichts. `static_assert` ist kein Makro, sondern ein neues Schlüsselwort.

Übungen

3.8 Warum sollte man das oben vorgestellte Makro `QUAD(x)` *nicht* viel einfacher so formulieren: `#define QUAD(x) x*x` ?

3.9 Strukturieren Sie die Lösung der Taschenrechneraufgabe von Seite 121 entsprechend den Empfehlungen des Abschnitts 3.3.

3.4 Funktions-Templates

Oft ist dieselbe Aufgabe für verschiedene Datentypen zu erledigen, zum Beispiel Sortieren eines `int`-Arrays, eines `double`-Arrays und eines String-Arrays. Das Kopieren einer Sortierfunktion für verschiedene Datentypen ist fehleranfällig, weil Änderungen in allen Versionen nachgezogen werden müssen. Mit *Templates* (deutsch *Schablonen*) können Funktionen mit parametrisierten Datentypen geschrieben werden. Mit »parametrisierten Datentypen« ist gemeint, dass eine Funktion für einen beliebigen, noch festzulegenden

Datentyp geschrieben wird. Für den noch unbestimmten Datentyp wird ein Platzhalter (Parameter) eingefügt, der später durch den tatsächlich benötigten Datentyp ersetzt wird. Die allgemeine Form für Funktions-Templates zeigt Abbildung 3.10.

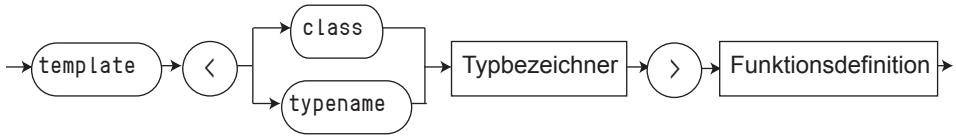

Abbildung 3.10: Syntax eines Funktions-Templates

Anstatt `class` kann auch `typename` geschrieben werden. Dabei ist *Typbezeichner* ein beliebiger Name, der in der *Funktionsdefinition* als Datentyp verwendet wird. In diesem Buch wird in der Regel `class` verwendet, wenn das Template eher für Klassen-Objekte verwendet werden soll. Wenn sowohl Klassen wie auch Grunddatentypen wie `int` und `double` gemeint sind, wird meistens `typename` gewählt.

Das folgende Programmbeispiel sortiert ein `int`-Feld und ein `double`-Feld, obwohl nur eine *einzige* Funktion `quicksort()` geschrieben wird, die hier den Datentypplatzhalter `T` benutzt. In der C++-Standardbibliothek gibt es eine Funktion `sort()` zum Sortieren (siehe Seite 667), die hier bewusst ignoriert wird, um den Template-Mechanismus am Beispiel zu demonstrieren. Die Wirkungsweise des Quicksort ist zum Beispiel in [CLR] beschrieben.

Damit ist `quicksort()` leicht anwendbar für Vektoren von Objekten einer beliebigen Klasse. Die Begriffe Objekt und Klasse sind Ihnen schon im ersten Kapitel begegnet, mehr erfahren Sie im nächsten Kapitel.

Listing 3.10: Funktions-Templates

```
// cppbuch/k3/qsort.cpp
#include<iostream>
#include<vector>
using namespace std;

template<typename T> // Template mit T als Parameter für den Datentyp (Platzhalter)
void tausche(T& a, T& b) { // a und b vertauschen
    const T TEMP = a;
    a = b;
    b = TEMP;
}

// Die folgende Funktion würde auch mit einer Parameterliste (T a, T b) arbeiten, d.h. einer
// Kopie per Wert. Da T aber für einen beliebigen Datentyp steht, wird eine Referenz bevorzugt,
// um Kopien von möglicherweise sehr großen Objekten zu vermeiden.
template<typename T>
bool kleiner(const T& a, const T& b) { // Vergleich
    return a < b;                      // zu < siehe auch Text am Abschnittsende
}

template<typename T>
void drucke(const vector<T>& V) {
   for (unsigned int i = 0; i < V.size(); ++i) {
```

```
        cout << V[i] <<" ";
   }
   cout << endl;
}

template<typename T>
void quicksort(vector<T>& a, int links, int rechts) {
    int li = links,
       re = rechts;
    // Die Verwendung von unsigned int für li und re wäre falsch, weil bei der
    // --Operation unten auch der Wert –1 auftreten kann. Eine Mischung der Typen
    // int und unsigned in Vergleichen oder bei impliziten Typkonversionen provoziert
    // ohnehin leicht Programmierfehler, die der Compiler nicht entdeckt (siehe Beispiel
    // if(u < i)... auf Seite 66).
    T el = a[(links+rechts)/2];
    do {
        while(kleiner(a[li],el)) ++li;
        while(kleiner(el,a[re])) --re;
        if (li < re) tausche(a[li],a[re]);
        if (li <= re) {++li; --re;}
    } while (li <= re);
    if (links < re) quicksort(a, links, re);
    if (li < rechts) quicksort(a, li, rechts);
}

int main () {
    vector<int> iV(10);
    iV[0]=100; iV[1]=22; iV[2]=-3; iV[3]=44; iV[4]=6;
    iV[5]= -9; iV[6]=-2; iV[7]= 1; iV[8]=8; iV[9]=9;
    // In den folgenden beiden Anweisungen werden vom Compiler, gesteuert durch den
    // Datentyp vector<int> des Parameters iV, aus den obigen Templates die Funktionen
    // quicksort( vector<int>&, int, int) und drucke(const vector<int>&) erzeugt,
    // ebenso wie die implizit aufgerufenen Funktionen tausche(int&, int&)
    // und kleiner(const int&, const int&).*
    quicksort(iV, 0, iV.size()-1);
    drucke(iV);
    vector<double> dV(8);
    dV[0]=1.09; dV[1]=2.2; dV[2]=79.6; dV[3]=-1.9; dV[4]=2.7;
    dV[5]=100.9; dV[6]=18.8; dV[7]=99.9;
    // Generierung der überladenen Funktionen quicksort(vector<double>&, int, int) und
    // drucke(const vector<double>&) (und der aufgerufenen Funktionen
    // tausche(double&, double&) und kleiner(const double&, const double&)):
    quicksort(dV, 0, dV.size()-1);
    drucke(dV);
}  // Ende von main()
```

Innerhalb von `main()` stellt der Compiler anhand des Funktionsaufrufs fest, für welchen Datentyp die Funktion benötigt wird, und bildet die Definition mithilfe des Templates. Für jeden verwendeten Datentyp wird vom Compiler aus dem Template eine Funktion erzeugt – ebenso wie Sie mit *einer* Form einen Schokoladen- und einen Nußkuchen backen können.

Mit `quicksort()` liegt ein universelles Sortierprogramm vor, das für verschiedene Datentypen geeignet ist. Auffällig ist, dass der Vergleich, welches Element kleiner ist, als Funktionsaufruf `kleiner()` innerhalb `quicksort()` formuliert wurde, anstatt `while(feld[i] < feld[j])` zu schreiben. Warum? Es ist nicht selbstverständlich, dass der Operator < für beliebige Klassen und Datentypen definiert ist. Durch Auslagern des Vergleichs braucht die Funktion `quicksort()` nicht verändert zu werden, wenn der Operator < anders definiert werden muss. Man kann mit einer ausgelagerten Vergleichsfunktion oder mit später zu besprechenden Zeigern auf Funktionen (Seite 223) oder mit Funktionsobjekten (Seite 344) leichter die Sortierung nach verschiedenen Kriterien realisieren.

3.4.1 Spezialisierung von Templates

Nehmen wir an, dass `double`-Zahlen wie bisher, `int`-Zahlen jedoch nach dem *Absolutbetrag* sortiert werden sollen. Der Vergleichsoperator < kann dann nicht mehr direkt auf die Zahlen angewendet werden. Zur Realisierung können wir aber ausnutzen, dass ein Template für festzulegende Datentypen *spezialisiert* werden kann. Um die Sortierung nach dem Absolutbetrag nur für `int`-Zahlen durchzuführen, muss das Template für die Funktion `kleiner()` spezialisiert werden. Spezialfälle von überladenen Funktionen werden *nach* den nicht-spezialisierten Templates eingefügt, so auch das spezialisierte Template `kleiner<int>()`:

```
// #include<cstdlib> für abs() nicht vergessen!
template<>
bool kleiner<int>(const int& a, const int& b) {
   // Das int in kleiner<int> darf weggelassen werden (Typdeduktion).
   return abs(a) < abs(b); // Vergleich nach dem Absolutbetrag!
}
```

Anstelle eines spezialisierten Templates kann auch eine gewöhnliche Funktion treten, die vom Compiler bevorzugt gewählt wird, wenn die Parametertypen passen, etwa

```
bool kleiner(int a, int b) { // gewöhnliche Funktion
   return abs(a) < abs(b); // Vergleich nach dem Absolutbetrag!
}
```

Der Unterschied besteht darin, dass in der Parameterliste einer gewöhnlichen Funktion weitgehende Typumwandlungen möglich sind, in einem spezialisierten Template jedoch nicht. Zum Beispiel könnte man `kleiner()` benutzen, um sich die jeweils kleinere Zahl anzeigen zu lassen:

```
// OHNE gewöhnliche Funktion, aber mit spezialisiertem Template
cout << (kleiner(3, 6) ? 3 : 6) << endl; // Ausgabe 3
cout << (kleiner(3.4, 6) ? 3 : 6) << endl;// Fehlermeldung:
      // no matching function for call to kleiner(double, int)

// MIT gewöhnlicher Funktion, aber OHNE Templates
cout << (kleiner(3, 6) ? 3 : 6) << endl; // Ausgabe 3
cout << (kleiner(3.1, 3.3) ? 3.1 : 3.3) << endl; // Ausgabe 3.3
  // falsch wegen Genauigkeitsverlust bei der Umwandlung
```

Weil schärfere Typprüfungen normalerweise erwünscht sind, sollten spezialisierte Templates statt gewöhnlicher Funktionen eingesetzt werden.

3.4.2 Einbinden von Templates

Im Abschnitt 3.3.2 auf Seite 123 wurde gezeigt, wie Funktionsimplementationen vorübersetzt und dann eingebunden werden können. Das gilt nicht für Funktions-Templates! Eine **.o*-Datei, die vom Compiler durch Übersetzen einer Datei nur mit einem Template erzeugt wird, ohne in der Datei einen konkreten Datentyp davon zu erzeugen, enthält keinen Programmcode und keine Daten.

Ein Template ist keine Funktionsdefinition im bisherigen Sinne, sondern eben eine Schablone, nach der der Compiler *erst bei Bedarf* eine Funktion zu einem konkreten Datentyp erzeugt. Dateien mit Templates sind deswegen mit `#include` einzulesen. Demzufolge könnten die Template-Definitionen ebensogut in der Header-Datei stehen. Manchmal wird eine besondere Extension für den Dateinamen verwendet, zum Beispiel *.t*:

```
// Datei schablone.t
#ifndef SCHABLONE_T
#define SCHABLONE_T
// hier folgen die Template-Deklarationen
//...

// ––– Implementierung ––––
// hier folgen die Template-Definitionen
//...
#endif
```

Diese Dateien werden wie **.h*-Dateien eingelesen, obwohl sie Template-Definitionen enthalten. Die Template-Definitionen können natürlich auch `inline` sein. Diese Lösung wird häufig in diesem Buch verwendet, weil sie einfacher ist (weniger Dateien) und weil die nicht eindeutige Zuordnung von Templates zu Header- oder Implementationsdateien vermieden wird. Es handelt sich hierbei nur um organisatorische Maßnahmen zur Einbindung von Templates. Ein weiteres Template-Compilationsmodell wird in Abschnitt 23.7 (Seite 619) diskutiert.

Übungen

3.10 Schreiben Sie eine Template-Funktion `getType(T t)` mit Template-Spezialisierungen, die den Typ des Parameters `t` als String zurückgibt. Eine möglich Anwendung könnte so aussehen (Ausgabe des Programms siehe Kommentare *//*):

```
#include<iostream>
#include"gettype.t"
using namespace std;
int main() {                    // Ausgabe
   int i;
   cout << getType(i) << endl;  // int
   unsigned int ui;
   cout << getType(ui) << endl; // unsigned int
   char c;
   cout << getType(c) << endl;  // char
   bool b;
   cout << getType(b) << endl;  // bool
   float f;           // Annahme: float ist nicht in getType() berücksichtigt:
   cout << getType(f) << endl;  // unbekannter Typ!
}
```

3.11 Schreiben Sie eine Template-Funktion `betrag(T t)`, die genau wie `abs()` den Betrag von `t` zurückgibt. Für manche Grunddatentypen wie `char` oder `bool` ist der Begriff »Betrag« nicht sinnvoll. Überlegen Sie, wie Sie durch eine spezialisierte Template-Funktion erreichen können, dass eine fälschliche Verwendung von `betrag()` mit einem `bool`-Argument zur Ausgabe einer Fehlermeldung und anschließendem Programmabbruch führt.

3.5 inline-Funktionen

Ein Funktionsaufruf kostet Zeit. Der Zustand des Aufrufers muss gesichert und Parameter müssen eventuell kopiert werden. Das Programm springt an eine andere Stelle und nach Ende der Funktion wieder zurück zur Anweisung nach dem Aufruf. Der relative Verwaltungsaufwand fällt umso stärker ins Gewicht, je weniger Zeit die Abarbeitung des Funktionskörpers selbst verbraucht. Der absolute Aufwand macht sich mit steigender Anzahl der Aufrufe bemerkbar, zum Beispiel in Schleifen. Um diesen Aufwand zu vermeiden, können Funktionen als `inline` deklariert werden. `inline` bewirkt, dass bei der Compilation der Aufruf durch den Funktionskörper ersetzt wird, also gar kein echter Funktionsaufruf erfolgt. Die Parameter werden entsprechend ersetzt, auch die Syntaxprüfung bleibt erhalten. Betrachten wir die einfache Funktion `quadrat()`, die das Quadrat einer Zahl zurückgibt:

```
inline int quadrat(int x) {
   return x*x;
}
```

Der Aufruf `z = quadrat(100);` wird wegen des Schlüsselworts `inline` vom Compiler durch `z = 100*100;` ersetzt. Gute Compiler würden darüber hinaus den konstanten Ausdruck berechnen und `z = 10000;` einsetzen. Der Verwaltungsaufwand für den Aufruf einer Funktion entfällt, das Programm wird schneller. Es ist nicht sinnvoll, die Ersetzung von vornherein *selbst* vorzunehmen, weil bei einer Änderung der Funktion alle betroffenen Stellen geändert werden müssten anstatt nur die Funktion selbst. `inline`-Deklarationen empfehlen sich ausschließlich für Funktionen mit einem Funktionskörper kurzer Ausführungszeit im Vergleich zum Verwaltungsaufwand für den Aufruf. `inline` ist nur eine *Empfehlung* an den Compiler, die Ersetzung vorzunehmen, er muss sich nicht daran halten. `inline`-Deklarationen sollten sich ausschließlich in Header-Dateien befinden. Der Grund wird klar, wenn wir das Gegenteil annehmen. Im Beispiel liegen drei Dateien vor:

```
// Deklarationsdatei X.h
int f(int);
int g(int);
```

```
// Datei X.cpp
#include"X.h"
inline int f(int a) { // Fehler: inline in Definitionsdatei!
   return a*a;
}
```

```
int g(int a) {
    a += 1;
    return f(a);      // inline ist hier bekannt
}
```

```
// Datei main.cpp
#include"X.h"         // enthält kein inline
int main() {
   int a = 1, b;
   b = f(a) + g(a);  // inline ist hier unbekannt!
}
```

Bei getrennter Compilation der Dateien *X.cpp* und *main.cpp* gibt es zwei Fälle:

1. Innerhalb der Funktion `g()` kann die `inline`-Ersetzung von `f()` vorgenommen werden.
2. In `main.cpp` weiß der Compiler nichts davon, dass `f() inline` sein soll, und nimmt einen Funktionsaufruf an.

Die Konsequenz ist, dass der Linker die Definition von `f(int)` nicht findet und sich mit einer Fehlermeldung[1] verabschiedet. Richtig wäre folgende Struktur:

```
// Datei X.h
inline int f(int a) { // inline in Deklarationsdatei!
    return a*a;
}

int g(int);
```

```
// Datei X.cpp
#include"X.h"
int g(int a) {
    a += 1;
    return f(a);   // inline ist hier bekannt
}
```

```
// Datei main.cpp
#include"X.h"
int main() {
   int a = 1, b;
   b = f(a) + g(a);  // inline von f(int) ist hier bekannt!
}
```

Inline-Funktionen werden in jeder Übersetzungseinheit expandiert und unterliegen damit dem internen Linken (vergleiche Seite 126).

[1] Der G++-Compiler ignoriert inline, wenn keine Optimierung eingeschaltet ist. Durch Einschalten der Optimierung wird der Fehler sichtbar, etwa `g++ -O2 X.cpp main.cpp`.

3.6 Namensräume

Ein Namensraum (englisch *namespace*) ist ein mit Namen gekennzeichneter Sichtbarkeitsbereich (scope). Ein Namespace erlaubt die Gruppierung zusammengehöriger Programmteile. Namespaces sind auch eingeführt worden, damit verschiedene Programmteile zusammenarbeiten können, die vorher (ohne Namespaces) aufgrund von Namenskonflikten im globalen Sichtbarkeitsbereich nicht zusammen verwendet werden konnten. Beispiel:

```
// abc.h (nützliche Funktionen der ABC-GmbH)
int print(const char*);
void func(double);
```

```
// xyz.h (nützliche Funktionen der XYZ Enterprises Ltd.)
int print(const char*);
void func();
```

```
// main.cpp
#include"abc.h"
#include"xyz.h"
int main() {
    print("hello world!");      // welches print()?
    func(1524.926);            // ok, überladen
    func();                    // ok, überladen
}
```

Es ist so nicht möglich, die Funktionsbibliotheken beider Firmen gleichzeitig zu benutzen. Die Lösung besteht in der Einführung von zusätzlichen, übergeordneten Sichtbarkeitsbereichen, den Namespaces. Die Deklaration ähnelt der von Klassen:

```
namespace abc {
    int print(const char*);
}  // ; ist nicht notwendig
```

Klassen und Funktionen werden durch *Using-Direktiven* nutzbar gemacht:

```
// abc.h (nützliche Funktionen der ABC-GmbH)
namespace abc {
   int print(const char*);
   void func(double);
}
```

```
// main.cpp
#include"abc.h"
#include"xyz.h"
int main() {                  // Using-Direktive:
   using namespace abc;       // alle Namen aus abc zugänglich machen
   print("hello world!");     // = abc::print()
}
```

Eine andere Möglichkeit ist der gezielte Zugriff auf Teile eines Namespace durch eine Using-Deklaration oder einen qualifizierten Namen, der die Funktion oder Klasse über den Bereichsoperator :: anspricht.

```
int main() {
    abc::print("hello world!");      // qualifizierter Name
    using abc::print;               // Using-Deklaration: lokales Synonym einführen
    print("hello world!");          // = abc::print()
}
```

Alle Klassen und Funktionen der C++-Standardbibliothek (Kapitel 26) sind im Namespace `std`. Aus diesem Grund wird in den Programmen dieses Buchs häufig `using namespace std;` benutzt. Alternativ ist der Zugriff über einen qualifizierten Namen möglich, zum Beispiel

```
std::cout << "keine using-Deklaration notwendig!";
```

Bei sehr langen Namen besteht die Möglichkeit der Abkürzung:

```
namespace SpecialSoftwareGmbH_KlassenBibliothek {
   // ....
}
// Abkürzung
namespace sskb = SpecialSoftwareGmbH_KlassenBibliothek;
using namespace sskb;  // Benutzung der Abkürzung
```

3.7 C++-Header

In C/C++ gibt es Bibliotheken mit verschiedenen Klassen und Funktionen. Die Funktionsbibliotheken entstammen teilweise der Sprache C. Beim Linken werden die benötigten Funktionen aus den Bibliotheksdateien dazugebunden. Die Header-Dateien sind im *include*-Verzeichnis zu finden. Auf die zu C++ gehörende Bibliothek wird in Kapitel 26 ab Seite 741 eingegangen. Ab Seite 873 werden die aus der Programmiersprache C kommenden Funktionen beschrieben.

Programme erhalten Zugriff zu Standardfunktionen und Klassen über das Einschließen der passenden Header mit `#include`. Diesen Headern können (müssen aber nicht) Dateien mit demselben oder ähnlichen Namen entsprechen. Die Namen der Header sind vom C++-Standard vorgeschrieben, die Implementierung durch die Compilerhersteller nicht. Alle Funktionsprototypen der C-Include-Dateien, deren Dateiname auf ».h« endet, gehören zur Sprache C und zum *globalen* Namensraum. Dieselben Funktionen werden auch unter C++ zur Verfügung gestellt, aber unter neuen Dateinamen, die sich durch ein vorangestelltes »c« und das Fehlen der Datei-Extension ».h« unterscheiden. Die Funktionen sind dann im Namespace `std` definiert. Die Möglichkeit, Funktionen mit der Datei-Extension ».h« einzubinden, bleibt unberührt. Eine C-Header-Datei *name.h* eines C++-Systems enthält den zugehörigen Header `<cname>`:

```
// name.h
#ifndef NAME_H
#define NAME_H
#include<cname>
using namespace std; // Namen global sichtbar machen
#endif
```

```
// cname
#ifndef CNAME
#define CNAME

namespace std {
    extern "C" void func(); // ein C-Prototyp, siehe Seite 144
    // ...
}
#endif
```

Die Beispiele zeigen die verschiedenen Möglichkeiten für `#include`:

```
// Beispiel 1: C-Funktionen, globaler Namensraum
#include<string.h>              // strlen()
#include<iostream>

int main() {
    char text[] = "Hello";
    std::cout << "Die Länge von " << text
              << " ist " << strlen(text) << std::endl;
}
```

```
// Beispiel 2: C++, dieselbe Funktion, Namespace std
#include<cstring>
#include<iostream>
using namespace std;

int main() {
    char text[] = "Hello";
    cout << "Die Länge von " << text << " ist " << strlen(text) << endl;
}
```

Ohne `using namespace std;` hätte in Beispiel 2 `std::strlen(text)`, `std::cout` und `std::endl` geschrieben werden müssen. Außer den C-Funktionen gibt es natürlich zusätzlich Standard-Header für die Klassen der C++-Standardbibliothek:

```
// Beispiel 3: C++, String-Klasse
#include<string>                    // ohne .h-Extension
#include<iostream>
using namespace std;

int main() {
    string text("Hello");
    cout << "Die Länge von " << text << " ist " << text.length() << endl;
}
```

```
// Beispiel 4: C++, String-Klasse
#include<string>
#include<iostream>
using std::cout;                    // begrenzte Auswahl
using std::endl;
using std::string;

int main() {
   string text("Hello");
   cout << "Die Länge von " << text << " ist " << text.length() << endl;
}
```

Um Namenskonflikte zu vermeiden, ist es grundsätzlich empfehlenswert, für ein Projekt (oder Teilprojekte) Namespaces zu definieren und zu benutzen. Besonders wichtig ist dies beim Schreiben von Bibliotheken. Bei der Benutzung sollte die selektive Auswahl wie etwa bei `std::cout` oder wie in Beispiel 4 bevorzugt werden. In Header-Dateien für Klassen sollte `using namespace std;` vermieden werden, damit Benutzer dieser Klassen nicht gezwungenermaßen `std` »erben«. In `main()`-Dateien ist die Verwendung hingegen unproblematisch.

3.7.1 Einbinden von C-Funktionen[2]

Das Einbinden von C-Funktionen wird mit `#include` bewerkstelligt. C-Prototypen werden in der Header-Datei mit

```
extern "C" {
   // .. hier folgen die C-Prototypen
}
```

eingebunden. Der Grund dafür liegt in der unterschiedlichen Behandlung von Funktionsnamen durch den C++-Compiler im Vergleich zu einem C-Compiler. Die korrekte Einbindung in C++-Header-Dateien wird in den C++-Header-Dateien über die Abfrage des Makros `__cplusplus` gesteuert. Damit werden `extern "C"{` samt schließender Klammer in eine Übersetzungseinheit integriert, wie unten zu sehen. Das Makro `__cplusplus` wird bei einer C++-Compilation automatisch gesetzt. Anstelle der Prototypen kann eine weitere `#include`-Anweisung stehen.

```
#ifdef __cplusplus
extern "C" {
#endif
   // .. hier folgen die C-Prototypen
#ifdef __cplusplus
}
#endif
```

Übungen

3.12 Gegeben sei die folgende Funktion[3] `fastbubblesort()`, die schneller als die Bubble-Sort-Variante von Seite 82 ist:

[2] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

[3] Leider habe ich die Vorlage zu dieser Aufgabe, die ich nur abgewandelt habe, nicht mehr gefunden. Der Autor des Originals möge mir verzeihen.

```
void fastbubblesort(vector<int>& feld) {
   int temp;
   do {
      temp = feld[0];
      for(size_t j = 1; j < feld.size(); j++) {
         if(feld[j] < feld[j-1]) { // vertauschen
            temp     = feld[j-1];
            feld[j-1] = feld[j];
            feld[j]  = temp;
         }
      }
   } while(temp != feld[0]); // keine Vertauschung mehr
}
```

Warum sollte diese Funktion schneller sein? Sie vergleicht wie üblich ein Vektor-Element mit dem vorhergehenden und vertauscht die Elemente, sofern das Element kleiner als der Vorgänger ist. Dieser Vorgang wird solange wiederholt, bis das Element `temp` unverändert bleibt, also nichts mehr zu sortieren ist. Weil gegebenenfalls schnell erkannt wird, dass nichts mehr zu sortieren ist, ist dieser Bubble-Sort bei teilweise vorsortierten Feldern im Mittel etwas schneller als eine Variante mit zwei geschachtelten Schleifen fixer Durchlaufanzahl. *Leider, leider, enthält die Funktion zwei schwere Fehler! Welche?*

3.13 Wer Mathematik nicht mag, überspringe bitte diese Aufgabe. Schreiben Sie eine Funktion `double polynom(const vector<double>& k, double x)`, die den Wert des Polynoms $f(x) = k_n x^n + k_{n-1} x^{n-1} + \cdots k_1 x + k_0$ zurückgibt. Vermeiden Sie zur effizienten Berechnung unnötige Mehrfachberechnungen der Potenzen von x. Der Vektor `k` soll nur die $n+1$ Koeffizienten enthalten, das heißt, `k[0]`=k_0, `k[1]`=k_1 usw.

3.14 Schreiben Sie ein Programm, das eine exakte Kopie seines eigenen Quellcodes auf dem Bildschirm ausgibt, *ohne* auf eine Datei zuzugreifen (schwierig und nur für Knobelfreunde, Hinweis siehe [Hof06]).

3.15 Was ist an der folgenden Funktion falsch? Zur Erinnerung: `UINT_MAX` ist die größtmögliche `unsigned int`-Zahl.

```
void pruefeZahl(unsigned int zahl) { // prüfen, ob zahl im Bereich liegt
   if(zahl < 0 || zahl > UINT_MAX) {
     cerr << "Zahl " << zahl
          << " liegt nicht im Bereich!" << endl;
   }
}
```