4 Programmstrukturierung

***Inhalt*:** *Große Programme müssen in übersichtliche Teile zerlegt werden. Sie werden dazu verschiedene Mechanismen kennen lernen:*

- *Delegieren von Teilaufgaben in Form von Funktionen beschreiben*
- *Strukturieren und Zusammenfassen von zusammengehörigen Teilaufgaben in Modulen*

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.

4.1 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.

4.1.1 Aufbau und Prototypen

Auf Seite 78 wird die Fakultät einer Zahl berechnet. Dies soll die Grundlage für eine einfache Funktion bilden, die diese Aufgabe ausführt. Eine Funktion `Fakultaet()` könnte wie folgt in ein Programm integriert werden:

```
// cppbuch/k4/fakulta2.cpp
#include<iostream>
using namespace std;

// Funktionsprototyp (Deklaration)
unsigned long Fakultaet(int);

int main() {
   int n;
   do {
      cout << "Fakultät berechnen. Zahl >= 0? :";
      cin >> n;
   } while(n < 0);

   cout << "Das Ergebnis ist "
        << Fakultaet(n) << endl;              // Aufruf

   // alternativ mit Zwischenablage des Ergebnisses:
   unsigned long Erg = Fakultaet(n);
   cout << "Das Ergebnis ist " << Erg << endl;
}

// Funktionsimplementation (Definition)
unsigned long Fakultaet(int zahl) {
   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 4.3.4 genauer eingegangen.

Der Aufruf der Funktion geschieht einfach durch Namensnennung. Von der Funktion auszuwertende Daten werden in runden Klammern `( )` übergeben. Wenn eine Funktion einen Rückgabetyp ungleich `void` hat, muss im Funktionskörper `{...}` irgendwo ein Ergebnis dieses Typs mit der Anweisung `return` zurückgegeben werden. Andere Möglichkeiten der Ergebnisrückgabe werden in Abschnitt 4.2 vorgestellt. Die Wirkung eines Funktionsaufrufs ist, dass das zurückgegebene Ergebnis an die Stelle des Aufrufs tritt!

Syntax eines Funktions*prototypen* (vergleiche mit obigem Beispiel):

Rückgabetyp Funktionsname `(` *Parameterliste* `);`

Der Rückgabetyp kann ein nahezu beliebiger Datentyp sein. Ausnahmen sind die Rückgabe einer Funktion[1] 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 n     );
      ⋮              ⋮        ⋮       ⋮        ⋮
  Rückgabetyp    Funktionsname  (  Parameterliste  );
```

Die *Parameterliste* besteht in diesem Fall nur aus einem einzigen Parameter.

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);`

Die Parameternamen 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. ***Tipp***

[1] Auch eine Funktion ist von einem bestimmten Typ.

Syntax der Funktions*definition*:

Rückgabetyp Funktionsname (*Formalparameterliste*) *Block*

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:

```
unsigned long   Fakultaet   (       int x       )   {...}
```

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

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

Syntax des Funktions*aufrufs*:

Funktionsname (*Aktualparameterliste*)

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* mit dem richtigen Datentyp in der Formalparameterliste vorhanden ist. Der Aufruf der Funktion bewirkt, dass `n` an die Stelle des Platzhalters `x` gesetzt wird, sodass der Programmcode im Block für `n` durchgeführt wird. Am Schluss wird die berechnete Fakultät mit dem richtigen Ergebnisdatentyp zurückgegeben. Zurückgegeben wird nur der Wert von (`zahl`), nicht `zahl` selbst. Die Variable `zahl` ist *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.

4.1.2 Gültigkeitsbereiche und Sichtbarkeit in Funktionen

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Variable (siehe Seite 57). 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* und 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.

```
// cppbuch/k4/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 compilierfähig,
    // 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 eines Blocks.

Die Ausnahme bilden Variable, die innerhalb eines Blocks oder einer Funktion als `static` definiert werden. `static`-Variable erhalten einen festen Speicherplatz und werden nur ein einziges Mal *beim ersten Aufruf der Funktion* initialisiert. Falls kein Initialisierungswert vorgegeben ist, werden sie automatisch auf 0 gesetzt. Sie wirken wie ein Gedächtnis für eine Funktion, weil sie zwischen Funktionsaufrufen ihren Wert nicht verlieren. Eine Funktion, die sich merken kann, wie oft sie aufgerufen wurde, und dies anzeigt, sieht so aus:

```
// cppbuch/k4/static.cpp
#include<iostream>
using namespace std;

void func( ) {                // zählt die Anzahl der Aufrufe
    static int anz = 0;       // siehe Text
    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. `static`-Variablen sind globalen Variablen vorzuziehen, weil unabsichtliche Änderungen vermieden werden und mit dieser Variablen verbundene Fehler leichter lokalisiert werden können. Außerdem erforderte 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 4.3.3 eingegangen.

4.2 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,
- Parameter, die der Funktion bekannt gemacht werden und
- 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*.

4.2.1 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.

```
// cppbuch/k4/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. Der intern ablaufende Kopiervorgang bei der Parameterübergabe großer Objekte kann recht aufwendig werden, sodass in solchen Fällen die Übergabe per Referenz zu bevorzugen ist (siehe folgender Abschnitt 4.2.2).

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. Die letzte Ziffer einer Zahl erhält man durch modulo 10 (Restbildung), und sie kann durch ganzzahlige Division mit 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 oder weniger, falls die Zahl mit Nullen endet. Eine Rekursion *muss* auf eine Abbruchbedingung zulaufen, damit keine unendlich tiefe Verschachtelung entsteht mit der Folge eines Stacküberlaufs.

Tipp

```
// cppbuch/k4/qsum.cpp
#include<iostream>
using namespace std;

int qsum(long z) {                          // Übergabe per Wert
    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);
}
```

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;
}
```

4.2.2 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 57). 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 Beispiel:

Tipp

```
// cppbuch/k4/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);            // Syntax wie bei Übergabe per Wert
    cout << i << " = neuer Wert von i nach addiere_7\n";
}

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

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[2] 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 111.

4.2.3 Gefahren bei der Rückgabe von Referenzen

Bei der Rückgabe von Referenzen muss darauf geachtet werden, dass das zugehörige Objekt tatsächlich noch existiert (vgl. Kapitel 4.1.2). 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);
    //...
}
```

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&`):

[2] Hinweis für Leser mit Pascal-Kenntnissen: In Pascal dient dazu das Schlüsselwort VAR in der Parameterliste, allerdings gibt es dort keine `const`-Referenzen.

```
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.

4.2.4 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 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 wie folgt 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:

```
// cppbuch/k4/preis.cpp
#include<iostream>
#include<string>
using namespace std;

// Funktionsprototyp 2. Parameter mit Vorgabewert:
void PreisAnzeige(double Preis,
                  const string& Waehrung="Deutsche Mark");
```

```
// Hauptprogramm
int main() {
      // zwei Aufrufe mit unterschiedlicher Parameterzahl :
      PreisAnzeige(12.35);    // Default-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 Deutsche Mark
99.99 US-Dollar

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

4.2.5 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. Diese Wandlungsfähigkeit wird teilweise in der Literatur als *Polymorphismus* bezeichnet. Die Definition von Polymorphismus ist nicht einheitlich; die in diesem Buch benutzte ist auf Seite 276 und im Glossar auf Seite 671 zu finden. 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:

```
// cppbuch/k4/ueberlad.cpp
#include<iostream>
using namespace std;

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

// zweite Funktion gleichen Namens, aber unterschiedlicher Signatur
int max(int x, int y) {
```

```
    return x > y ? x : y;
}
int main() {
    double a=100.2, b= 333.777;
    int c=1700, d = 1000;
    cout << max(a,b) << endl;  // Aufruf von max(double, double)
    cout << max(c,d) << endl;  // Aufruf von max(int, int)
}
```

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

```
float e = 2.7182, pi = 3.14159;
cout << max(e,pi);
```

führt zum Aufruf von `max(double, double)`, und `max(31,'A')` zum Aufruf von `max(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 `max(3.1, 7)` ist nicht eindeutig interpretierbar. Das erste Argument spricht für `max(double, double)`, das zweite für `max(int, int)`. Der Compiler kann sich nicht entscheiden und erzeugt eine Fehlermeldung. Es bleibt einem natürlich unbenommen, selbst eine Typumwandlung vorzunehmen. Die Aufrufe

```
cout << max(3.1, static_cast<float>(7));
cout << max(3.1, static_cast<double>(7));
int x = 66;
char y = static_cast<char>(x);
cout << max(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 `char` in `int` 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 max(int, int)
cout << max(-1000, static_cast<char>(600));  // ergibt 88
cout << max(-1000, static_cast<char>(128));  // ergibt -128
cout << max(-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 5.3.4 und 9.5 werden wir eine Möglichkeit zur benutzerspezifischen Typumwandlung für beliebige Datentypen kennen lernen.

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. Machen wir einen 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.

4.2.6 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 erst in Kapitel 6 besprochen werden. Die Auswertung der Argumente wird bis dahin zurückgestellt (ab Seite 221).

Es bleibt dem Hersteller eines Compilers überlassen, ob er weitere Versionen mit zum Beispiel 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.

4.2.7 Spezifikation von Funktionen

Eine Funktion erledigt eine Teilaufgabe und ändert dabei den *Zustand* eines Programms. Es ist unbedingt sinnvoll, die Zustandsänderung durch die Bedingungen, die *vor* und *nach* dem Aufruf gelten, im Funktionskopf als Kommentar zu spezifizieren. Dazu gehören Annahmen über die Importschnittstelle (Eingabedaten, zum Beispiel Wertebereich), die Fehlerbedingungen, die Exportschnittstelle (Ausgabedaten) u.a.m.

Für Vor- und Nachbedingung werden auch die englischen Begriffe *precondition* und *postcondition* mit den Abkürzungen *pre* und *post* benutzt. Vor- und Nachbedingungen einzelner Programmteile können zum Nachweis der Korrektheit eines Programms benutzt werden. Die Beschreibung der Nachbedingung gibt an, *was* die Funktion leistet; dies ist die Spezifikation der Funktion.

Das *wie* sollte nicht beschrieben werden, um die Möglichkeit einer späteren Änderung der Implementierung nicht einzuschränken, zum Beispiel einen langsamen durch einen schnelleren Algorithmus zu ersetzen. 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 hervorragende Vertiefung der Thematik dieses Abschnitts ist in [Mey98] zu finden. Beispiel:

```
int newton(
// Berechnung der Nullstelle einer Funktion nach dem Newton-Verfahren

// Import
  double (*fp) (double),      // fp ist Zeiger auf eine Funktion,
                              // deren Nullstelle gesucht werden soll
                              // (siehe Text)
  double untergrenze,         // Intervallgrenzen
  double obergrenze,
  double epsilon,             // Genauigkeit

// Export
  double &nullstelle)

/*Returncodes :
      0 : Nullstelle gefunden
      1 : Nullstelle nicht gefunden
      2 : Verfahren konvergiert nicht
      3 : Vorbedingungen nicht erfüllt

  Vorbedingungen (preconditions):
      1. untergrenze < obergrenze
      2. epsilon > Darstellungsgenauigkeit
      3. stetig differenzierbare Funktion f
      4. im Intervall befindet sich eine Nullstelle

  Nachbedingungen (postconditions):
    bei Returncode 0:
      1. untergrenze <= nullstelle <= obergrenze
      2. abs(f(nullstelle)) <= epsilon

    sonst:
      3. siehe Returncodebeschreibung
*/

{
      // Programmcode
}
```

Durch die richtig gesetzten Kommentarzeichen ist das Beispiel bereits compilierbar, sodass der Compiler die Schnittstelle prüfen kann. Die Spezifikation sollte mit in eine Header-Datei übernommen werden, wobei dann `{ // Programmcode }` durch ein Semikolon zu ersetzen ist. Eine Header-Datei soll unter anderem die Prototypen von Funktionen enthalten (siehe Abschnitt 4.3). Sie wundern sich, was `(*fp)(double)` bedeutet? Warten Sie noch ein wenig, Zeiger auf Funktionen werden in Abschnitt 6.7 besprochen.

4.2.8 Beispiel Taschenrechnersimulation

Um ein etwas umfangreicheres Beispiel mit Funktionen zu geben, wird ein Taschenrechner simuliert, eine beliebte Aufgabe (siehe auch [Mar86], nach dem dieses Beispiel entworfen wurde, oder etwas komfortabler und aufwendiger [Str97, Kapitel 3.1]). Die hier verwendete und nur kurz beschriebene Methode des *rekursiven Abstiegs* ermöglicht es, auf elegante und einfache Art beliebig verschachtelte Ausdrücke auszuwerten. In [Aho95] können fortgeschrittene Interessierte ausführliche Erläuterungen der Methode finden.

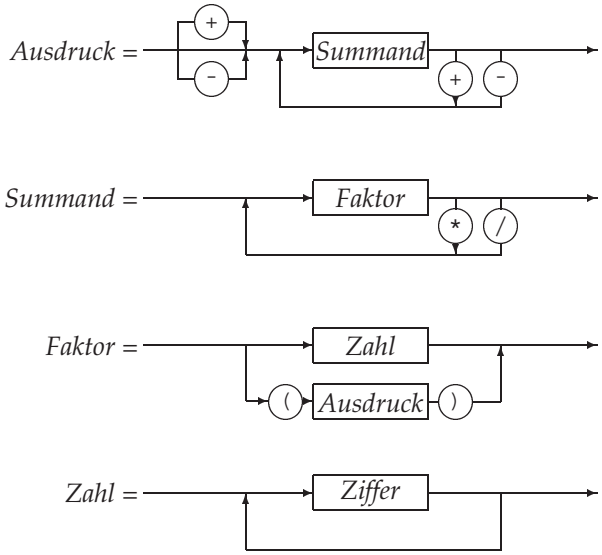

Abbildung 4.1: Syntaxdiagramm für *Ausdruck*

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. Die Syntax, das heißt die grammatikalische Struktur eines Ausdrucks, kann als so genanntes *Syntaxdiagramm* dargestellt werden, wie in Abbildung 4.1 zu sehen ist.

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 4.2 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.

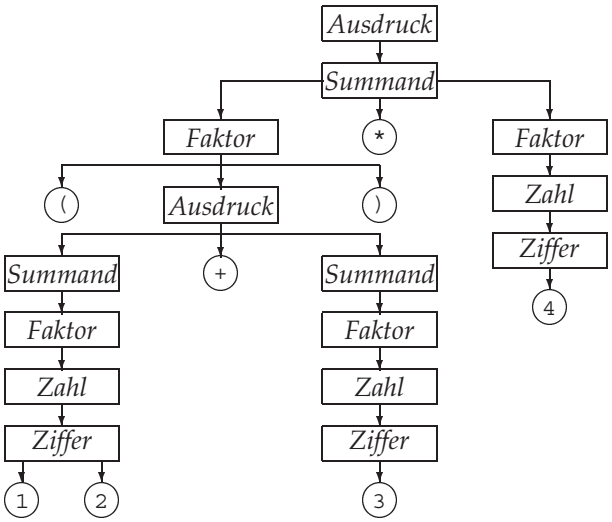

Abbildung 4.2: Ableitungsbaum für (12+3)*4

Abbildung 4.2 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 4.1 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.
- Vorrangregeln sollen beachtet werden.
- Keine aufwendige Syntaxprüfung

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

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

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

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

Damit kann das Hauptprogramm hingeschrieben werden:

```
int main() {
    char ch;
    while(true) {                    // Abbruch mit break
        cout << "\n>>";
        cin.get(ch);
        if(ch != 'e')
            cout << ausdruck(ch);
        else break;
    }
}
```

`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 (siehe Seite 96). Mit jedem weiteren Aufruf von `cin.get()` wird ein weiteres Zeichen geholt. `cin >> ch` wird nicht gewählt, weil RETURN 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. Zum Beispiel wird eine Schleife im Diagramm in eine `while()`-Anweisung transformiert. Die Entsprechung zwischen dem Syntaxdiagramm auf Seite 119 und Programmcode ist offensichtlich. Die Variable `c` wird als Referenz übergeben, damit bei Ende der Funktion der neue Wert dem aufrufenden Programm 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"; //*** s.u.
        else cin.get(c);               // ) überspringen
    }
    else f = zahl(c);
    return f;
}

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');
        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.

Übungsaufgaben

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

4.2 Schreiben Sie ein Programm, das eine exakte Kopie seines eigenen Quellcodes auf dem Bildschirm ausgibt, *ohne* auf eine Datei zuzugreifen. (Etwas schwieriger, Hinweis siehe [Hof85].)

4.3 Vervollständigen Sie das Beispiel in Abschnitt 4.2.8 und bringen Sie es zum Laufen.

4.3 Grundsätze der modularen Gestaltung

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: