

# Programmstrukturierung

Große Anwendungen müssen in kleinere Teile zerlegt werden. Hier werden Mechanismen behandelt, wie dies erfolgen kann.

Diskutiert werden dazu die nicht objektorientierten Sprachmittel von C++:

- Funktionen,
- Compilerdirektive,
- Funktionstemplates und
- Standardfunktionen.

## Inhalt

|  |    |
|--|----|
| 1. Funktionen.....                             | 4  |
| 1.1. Grundlagen.....                           | 5  |
| 1.2. Gültigkeitsbereich und Sichtbarkeit ..... | 11 |
| 1.3. Parameterübergabemechanismen .....        | 14 |
| 1.3.1. Allgemeine Verfahren.....               | 14 |
| 1.3.2. Wertübergabe .....                      | 16 |

|         |   |    |
|---------|---|----|
| 1.3.3.  | Referenzübergabe .....                            | 19 |
| 1.3.4.  | Arrays als Parameter .....                        | 21 |
| 1.3.5.  | Read-only Parameter .....                         | 23 |
| 1.3.6.  | Variable Anzahl Parameter, Default Parameter..... | 25 |
| 1.4.    | <code>static</code> Variablen in Funktionen ..... | 27 |
| 1.5.    | Seiteneffekte .....                               | 29 |
| 1.6.    | Inline Funktionen.....                            | 29 |
| 1.7.    | Rekursive Funktionen .....                        | 31 |
| 1.8.    | Überladen von Funktionen .....                    | 46 |
| 1.9.    | Funktion <code>main</code> .....                  | 49 |
| 1.10.   | Ausnahmebehandlung über Funktionsgrenzen .....    | 51 |
| 1.11.   | Die Funktion als Vertrag .....                    | 53 |
| 1.12.   | Dokumentation und Testen .....                    | 57 |
| 1.12.1. | Dokumentationsregeln .....                        | 57 |
| 1.12.2. | Testen .....                                      | 58 |
| 2.      | Modulare Gestaltung von Programmen .....          | 59 |

|   |    |
|---|----|
| 2.1. Getrenntes Übersetzen von Programmen .....           | 60 |
| 2.2. Dateiübergreifende Gültigkeit und Sichtbarkeit ..... | 66 |
| 2.3. Compilerdirektive und Makros .....                   | 70 |
| 2.3.1. #include .....                                     | 71 |
| 2.3.2. #ifdef, #ifndef, #define.....                      | 71 |
| 2.3.3. Makros.....  | 74 |
| 3. Funktionstemplates.....                                | 79 |
| 4. Standardfunktionen/Bibliotheken .....                  | 82 |

# 1. Funktionen

Funktionen sind ein C++ Sprachmittel, mit dem man große Programme in übersichtliche Teile zerlegen kann.

Funktionen sind Mittel zum Zweck der **Modularisierung** von Anwendungen:

- eine Anwendung wird durch Aufteilung in überschaubare Einheiten (Module) zerlegt,
- ein einzelnes Modul ist austauschbar,
- man erhält einfache, klare Schnittstellen und
- kann eine Aufteilung von Aufgaben an mehrere Programmierer vornehmen.

Funktionen dienen der **Abstraktion**:

- eine Lösung wird schrittweise verfeinert, wobei
- Details der Implementierung verborgen werden.

Da eine Funktion eine klare Schnittstelle und Arbeitsweise hat, kann dieselbe Funktion **mehrfach verwendet** werden

- im selben Programm
- oder in anderen Programmen, indem die Funktion in eine Bibliothek gestellt wird.

## 1.1. Grundlagen

Ein einfaches Programm, an dem die Verwendung von Funktionen demonstriert wird, berechnet die Fakultät von eingegebenen Zahlen:

```
$ cat fakultaet.cpp
#include <iostream.h>
long fakultaet(int);           // Funktionsprototyp (Deklaration)

main() {
    int x;
    long res;
    cout << "Eingabe Integer x: ".
    cin >> x;
    res = fakultaet(x);        // Aufruf der Funktion
    cout << "fak(" << x << ") = " << res << endl;
}

long fakultaet(int n) {      // Funktionsimplementierung (Definition)
    long fak = 1;
    for (int i=2; i<= n; i++)
        fak = fak*i;
    return fak;
}
$
```

aktueller Parameter

formaler Parameter

Die Funktionsdeklaration teilt dem Compiler mit, um welche Art von Funktion es sich handelt.

Die Funktionsdefinition spezifiziert, was genau die Funktion macht.

Der Funktionsaufruf bewirkt, dass die Aktionen der Funktion mit den Werten ausgeführt werden, die der aktuelle Parameter definiert. Dabei wird nach der Ausführung der Funktionsaktionen, der Name assoziiert mit dem berechneten Wert.

Die Syntax eines **Funktionsprototyps** hat die Form:

Rückgabetyt Funktionsname ( Parameterliste )

`long fakultaet (int);`

Beispiele:

```
int func(); // leere Parameterliste
int func (int, char); // Liste mit Parametertypen
int func (int x, char y); // Liste mit Parametertypen und Parameternamen
void proc (int x, char y); // „ohne“ Rückgabetyt = Prozedur
```

Die Syntax einer **Funktionsdefinition** hat die Form:

Rückgabetyyp Funktionsname ( formale Parameterliste ) { Block }

`long fakultaet(int) { ... return res; };`

Beispiele:

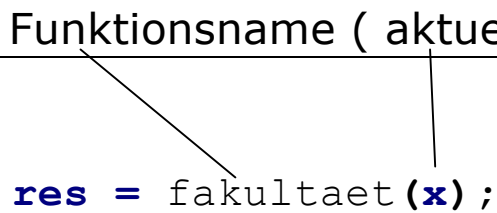
```
long power (long a, int b) {
    long result = 1;

    while (b > 0) {
        result *= a;
        b--;
    }
    return result;
}
```

```
double pi ()
{
    return 3.1415927;
}
```

Die Syntax eines **Funktionsaufrufs** hat die Form:

Funktionsname ( aktuelle Parameterliste )



```
res = fakultaet(x);
```

Zu beachten ist, dass die formalen und aktuellen Parameter in Anzahl und korrespondierenden Typen übereinstimmen müssen.

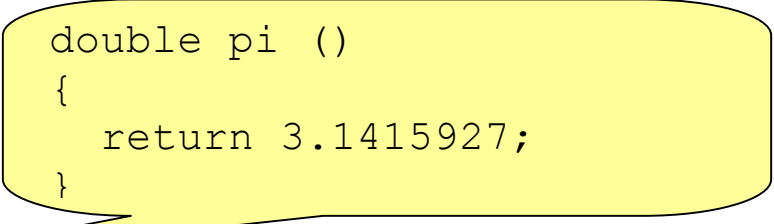
Durch den Aufruf wird der Code der Funktion (ihr Block) ausgeführt, dabei werden formale Parameter durch aktuelle Parameter ersetzt und das Ergebnis nach Beendigung der Funktion an den Aufrufer übergeben.

## Beispiel:

```
double pi ()
{
    return 3.1415927;
}

main() {
    float x;
    x = 2.3 * pi();
    cout << x;
}
```

```
main() {
    float x;
    x = 2.3 * ?
    cout << x;
}
```



```
double pi ()
{
    return 3.1415927;
}
```

## Anstelle der Struktur

1. Funktionsprototyp
2. main mit Funktionsaufruf
3. Funktionsdefinition

kann auch folgendes stehen:

1. Funktionsdefinition
2. main mit Funktionsaufruf

## Beispiel:

```
#include <iostream.h>
long power (long a, long b) {
    long result = 1;

    while (b > 0) {
        result *= a;
        b--;
    }
    return result;
}

main() {
    int x, y;
    cout << "Eingabe Integer x y: ";
    cin >> x >> y;
    cout << x << " hoch " << y << " = " << power(x,y) << endl;
}
```

## 1.2. Gültigkeitsbereich und Sichtbarkeit

In C++ gibt es Gültigkeits- und Sichtbarkeitsregeln für Namen:

- Namen sind nur **nach** der **Deklaration** und nur **innerhalb** des **Blocks gültig**, in dem sie deklariert wurden. Sie sind also block lokal.
- Namen sind auch **gültig** für Blöcke, die innerhalb des Blocks angelegt sind (**innerer** Block).
- Die **Sichtbarkeit** (engl. *visibility*) (z.B. von Variablen) wird eingeschränkt durch die Deklaration von Variablen gleichen Namens: im **inneren** Block ist die Variable des äußeren Blocks **nicht sichtbar**, sie ist verdeckt.
- Die Variablen, die innerhalb eines Funktionsblocks deklariert sind, sind **funktionslokale** Objekte und daher beim Aufrufer nicht bekannt. Z.B. kann in der o.a. `main` **nicht** auf die lokale Variable `result` der Funktion `power` zugegriffen werden.
- Formale Parameter von Funktionen verhalten sich bzgl. der Gültig- und Sichtbarkeit wie lokale Variable.

Beispiel:

```

$ cat gueltigSichtbar01.cpp
#include <iostream.h>

int a=0;           // global gültig
int b=2;

void main() { // Blocktiefe 1
    cout << "a: " << a << endl;

    int a = 1; // neues a, verdeckt globales a
    cout << "a: " << a << endl;

    { // Blocktiefe 2
        cout << "a: " << a << endl; // a aus Block 1
        int a = 2 // a aus Block 1 gültig, aber nicht mehr sichtbar
        cout << "a: " << a << endl;
    }

    cout << "a: " << a << endl;
}
$

```

```

$ gueltigSichtbar01

a: 0

a: 1

a: 1
a: 2

a: 1

$

```

In C++ existiert ein **Scope**-Operator (::), mit dem der Zugriff auf die Objekte, die in der ganzen Datei gültig sind (also nicht jeweils eine Stufe höher), möglich ist.

```

$ cat gueltigSichtbar02.cpp
#include <iostream.h>

int a=0;           // global gültig
int b=2;

void main() { // Blocktiefe 1
    cout << "a: " << a << endl;

    int a = 1; // neues a, verdeckt globales a
    cout << "a: " << a << endl;
    cout << "::a: " << ::a << endl;

    { // Blocktiefe 2
        cout << "a: " << a << endl; // a aus Block 1
        int a = 2; // a aus Block 1 gültig, aber nicht mehr sichtbar
        cout << "a: " << a << endl;
        cout << "::a: " << ::a << endl;
    }

    cout << "a: " << a << endl;
}
$

```

```

$ gueltigSichtbar02

a: 0

a: 1
::a: 0

a: 1

a: 2
::a: 0

a: 1

$

```

Im Rechner wird ein Datenbereich für block**lokale Objekte** bei Betreten des Gültigkeitsbereichs auf einem speziellen Speicherbereich (**Stack**) angelegt und bei Verlassen des Gültigkeitsbereichs automatisch entfernt.

Eine Ausnahme bilden als `static` markierte Variable: `static` Variable erhalten einen festen Speicherplatz und werden nur einmal beim ersten Aufruf der Funktion initialisiert. *Wir werden das später noch genauer sehen.*

### **1.3. Parameterübergabemechanismen**

Die Parameter einer Funktion bilden die Schnittstelle zum Datentransfer zwischen Aufrufer und aufgerufener Funktion (neben dem Rückgabewert der Funktion).

#### **1.3.1. Allgemeine Verfahren**

In Programmiersprachen werden u.a. folgende Arten der Parameterübergabe unterschieden:

##### **Wertübergabe (engl.: call by value)**

Mechanismus:

- Der Wert des aktuellen Parameters wird beim Aufruf des Unterprogramms ermittelt und in den formalen Parameter kopiert.

Folge:

- Als aktuelle Parameter sind auch Ausdrücke möglich.
- Änderungen am formalen Parameter im Unterprogramm beeinflussen den Wert des aktuellen Parameters nicht.

- Soll mit call by value eine Änderung durch das aufgerufene Unterprogramm erzielt werden, so müssen Zeiger übergeben werden und das Unterprogramm muss dereferenzieren. Dabei bleibt aber der Parameterwert (Zeiger) unverändert. **Das werden wir später sehen!**

Programmiersprachen:

- Ada, ALGOL 60, **C**, **C++**, Pascal

## **Adressübergabe (Variablenübergabe, engl.: call by reference)**

Mechanismus:

- Die Speicheradresse des aktuellen Parameters wird ermittelt und in den formalen Parameter kopiert.

Folge:

- Ausdrücke haben keine Adressen, damit können sie nicht als Parameter übergeben werden.
- Änderungen des formalen Parameters sind beim Aufrufer sichtbar.

Programmiersprachen:

Ada, **C++**, Pascal

## **Namensübergabe (engl.: call by name)**

Mechanismus:

- Der Wert des aktuellen Parameters wird bei jeder Verwendung im Unterprogramm neu berechnet (im Gegensatz zur einmaligen Berechnung beim Eintritt in das Unterprogramm bei Adressübergabe).

Folge:

- Ausdrücke sind als aktuelle Parameter möglich.
- Falls der aktuelle Parameter ein Ausdruck ist oder einen enthält (z.B. als Index), so entsteht ein hoher Rechenaufwand zur Laufzeit.
- Änderungen des formalen Parameters sind beim Aufrufer sichtbar.
- Die Wirkung ist i.A. schwer nachvollziehbar.

Programmiersprachen:

- ALGOL 60

### **1.3.2. Wertübergabe**

Der **Wert** des aktuellen Parameters wird beim Aufruf der Funktion ermittelt und in den formalen Parameter **kopiert**.

Somit sind als aktuelle Parameter auch Ausdrücke möglich:

```
$ cat add01.ccp
#include <iostream.h>
float addiere(float x, float y) {
    return x + y;
}

void main() {
    float x = 2;
    float y;
    y = addiere(x, x*3);
    cout << y << endl;
}
```

Ausdruck

```
$ add01
8
$
```

Der Aufruf der Funktion `incl1` hat **keinen** Effekt beim Aufrufer hinterlassen.

```
$ cat incl1.c
#include <iostream.h>
void incl1(int p) {
    p++;
    cout <<"  in incl1: p = " << p << endl;
}

main() {
    int x;
    x=1;
    cout << " vor incl1: x = " << x << endl;
    incl1(x);
    cout << "nach incl1: x = " << x << endl;
}
$
```

```
$ incl1

vor incl1: x = 1
in incl1: p = 2
nach incl1: x = 1

$
```

gleich

Dies ist aber kein Nachteil, denn damit kann man die formalen Parameter ändern und als lokale Variable verwenden.

Beispiel (Quersumme):

```

$ cat quersumme.cpp
#include <iostream.h>
long qsum(long z) {
    long sum = 0;
    while (z>0) {
        sum += z % 10;
        z = z / 10;
    }
    return sum;
}

main() {
    long x;
    cout << "Integer: ";
    cin >> x;
    cout << "Quersumme von " << x << " ist " << qsum(x) << endl;
}
$

```

z wird verändert

```

$ quersumme
Integer: 1001
Quersumme von 1001
ist 2
$

```

### 1.3.3. Referenzübergabe

Die **Speicheradresse** des aktuellen Parameters wird ermittelt und in den formalen Parameter **kopiert**.

Verwendet werden muss hier ein formaler Parameter vom Typ „Referenz“, ausgedrückt durch das Zeichen „&“. Damit arbeiten Aufrufer und aufgerufene Funktion mit derselben Speicherstelle, also mit demselben Objekt.

Der Aufruf der Funktion inc2 **hat einen** Effekt beim Aufrufer hinterlassen.

```
$ cat inc2.c
#include <iostream.h>
void incl(int &p) {
    p++;
    cout <<" in incl: p = " << p << endl;
}

main() {
    int x;
    x=1;
    cout << " vor incl: x = " << x << endl;
    incl(x);
    cout << "nach incl: x = " << x << endl;
}
$
```

```
$ inc2
```

```
vor incl: x = 1
in incl: p = 2
nach incl: x = 2
```

erhöht

Hier kann als aktueller Parameter **kein Ausdruck** und **keine Konstante** angegeben werden!

### 1.3.4. Arrays als Parameter

Aus dem Erbe von C gibt es in C++ einen Spezialfall: Arrays als Parameter.

- es wird nicht das Array, sondern seine Anfangsadresse übergeben ohne dass man explizit call by reference (durch &) angibt;
- die Funktion arbeitet nicht mit einer lokalen Kopie, sondern mit dem Original;
- die Funktion kennt die Größe des Arrays **nicht**.

D.h. Arrays sind **immer** Referenzparameter.

Array als Ergebnis einer Funktion ist **illegal**.

Beispiel (Kumuliere Arraywerte)

```
$ cat kumuliere.cpp
#include <iostream.h>
void kumuliere (int v[], int laenge) {
    int i;
    // sizeof(v) liefert 4 = Grösse einer Adresse !!!
    cout << "kum sizeof " << sizeof(v) << endl;
    for (i=1; i<laenge; i++)
        v[i] += v[i-1];
    // return v wäre illegal
}

int main () {
    int x[] = {1,2,3,4,5,6};
    cout << "main sizeof " << sizeof(x) << endl;
    kumuliere (x, 3);
    for (int i=0; i<5; i++) cout << x[i];
    cout << endl;
}
$
```

```
$ kumuliere
main sizeof 24
kum sizeof 4
13645
$
```

### 1.3.5. Read-only Parameter

Wenn man Parameter nicht verändern will oder wenn man Referenzparameter nur aus Effizienzgründen verwendet (also nicht zur Rückgabe von Werten), dann sollte man solche Parameter als `const` deklarieren:

- dies ist die Zusicherung an den Compiler, dass der Funktionskörper diese Parameter nicht ändert;
- der Compiler kontrolliert das und entdeckt versehentliche Änderungen.

Beispiel (Summe Elemente in int Array)

```
$ cat arraySumme.cpp
#include <iostream.h>
int Summe (const int v [], int laenge) {
    int sum = 0;
    for (int i=0; i<laenge; i++)
        sum += v[i];
    return sum;
}
int main () {
    int x[] = {1,2,3,4,5,6};
    cout << Summe (x, 3) << endl;
}
$
```

## Hörsaalübung

Eine Funktion für die Addition von zwei Zahlen soll entwickelt werden, inklusive eines Testprogramms.

1. Version mit Ergebniswert der Funktion und Wertübergabe
2. Version mit Rückgabewert als formaler Parameter und Ergebnistyp der Funktion void

Eine Funktion für die Summation der Werte eines int-Arrays (als Parameter) soll entwickelt werden, inklusive eines Testprogramms.

1. Version mit Ergebniswert der Funktion
2. Version mit Rückgabewert als formaler Parameter und Ergebnistyp der Funktion void

### 1.3.6. Variable Anzahl Parameter, Default Parameter

Funktionen können mit **variabler** Anzahl von Parametern aufgerufen werden. In der Prototypendeklaration müssen für die nicht angegebenen Parameter Vorgabewerte (engl. *defaults*) angegeben werden.

Dadurch können Programme in ihrer Funktionalität erweitert werden, ohne dass die „alten“ Funktionsaufrufe geändert werden müssen.

Beispiel (Ausgabe Aktienkurs)

```
$ cat aktienKurs01.cpp
#include <iostream.h>
#include <iomanip.h>
void printRate(float) ;

main() {
    float r=19.126;
    printRate(r);
}

void printRate(float rate) {
    cout << setw(10) << setprecision(6)
         << rate << endl;
}
$
```

```
$ aktienKurs01
    19.126
$
```

Das Programm soll so geändert werden, dass die Währung mit ausgegeben wird, der Default soll „USD“ sein.

```
$ cat aktienKurs02.cpp
include <iostream.h>
#include <iomanip.h>
#include <string>

void printRate(float rate, string currency="USD") ;

main() {
    float r=19.126;

    printRate(r);
    printRate(r, "EUR");
}

void printRate(float rate, string currency) {
    cout << setw(10) << setprecision(6)
        << rate << " " << currency << endl;
}
$
```

```
$ aktienKurs01
    19.126 USD
    19.126 EUR
$
```

## 1.4.static Variablen in Funktionen

In C++ gibt es eine Speicherklasse `static`, mit der der Speicherplatz funktionslokale Variable nach Terminierung erhalten bleibt. Damit hat man die Möglichkeit, dass Funktionen ein „Gedächtnis“ erhalten. Beim ersten Aufruf der Funktion wird der Speicherplatz initialisiert, bei jedem weiteren Aufruf wird mit dem vorherigen Wert weiter gerechnet.

Beispiel (Zufallszahlengeneratoren):

Algorithmus:

Zufallszahlen können nach der *linearen Kongruenzmethode* wie folgt ermittelt werden:

$$R_{n+1} = (a * R_n + c) \% m \text{ mit } R, a, c \geq 0, m \geq R, a, c$$

Die Kunst besteht darin,  $a, c$  und  $m$  so zu wählen, dass der Generator eine Gleichverteilung erzeugt. Als gute ausreichend gut haben sich große Zahlen für  $a$  und Primzahlen für  $c$  und  $m$  erwiesen.

Eine Funktion `rand` verwendet eine `static Variable`, die jeweils das zuletzt berechnete  $R_n$  speichert.

Programm:

```
$ cat random01.cpp
#include <iostream.h>

float random(long seed) {
    static long r = seed * 1000; // erster Wert
    const int a = 125;
    const int c = 3;
    const int m = 1717;
    r = (r*a+c)%m;
    return (float) r/m; // Zahl [0,1]
}

main() {
    for (int i = 1; i<=20; i++)
        cout << random(101) << endl;
}
$
```


```
$ random01
0.942924
0.86721
0.403029
0.380314
0.54106
0.634246
0.282469
0.310425
0.804892
0.613279
0.661619
0.704135
0.0186372
0.331392
0.425743
0.219569
0.447874
0.986022
0.254514
0.815958
$
```

## 1.5. Seiteneffekte

Seiteneffekte entstehen, wenn Funktionen globale Variable verändern. Dies sollte stets vermieden werden. Eine Funktion sollte ausschließlich Werte ändern, die sie über ihre Parameter erhält, d.h. die Wirkung einer Funktion ist an der Aufrufstelle ersichtlich.

Negativ Beispiel:

```
$ cat gerade.cpp
#include <iostream.h>
int x = 0; // globale Variable
void nexteGeradeZahl () {
    x += 2;
}
main () {
    cout << x << endl;
    nexteGeradeZahl ();
    cout << x << endl;
    x++; // ab jetzt bewirkt nexteGeradeZahl eine ungerade Zahl
    nexteGeradeZahl ();
    cout << x << endl;
}
$
```



Funktion verändert globale Variable

**Wie ist das Programm zu ändern, damit die Funktion `nexteGeradeZahl` immer eine gerade Zahl liefert (static verwenden?) ?**

## 1.6.Inline Funktionen

Bei **trivialen** Funktionen kann der Aufwand für die Parameterübergabe größer sein als die "eigentliche Funktion"; trotzdem ist die Verwendung einer Funktion sinnvoll wegen Abstraktion, Zugriffsschutz, etc. (gute Praxis bei Klassen: Zugriff auf Attribute nur über Funktionen -> später)

Ein vorangestellte `inline` bewirkt die unmittelbare Substitution des Körpers der Funktion an der Aufrufstelle (nach Ermessen des Compilers) zur Übersetzungszeit.

Beispiel:

```
$ cat inline01.cpp
#include <iostream.h>
inline int quadrat(int x) {
    return x*x;
}

main() {
    cout << quadrat(4) << endl;
}
$
```

## 1.7. Rekursive Funktionen

In einer Funktion kann eine andere Funktion aufgerufen werden. C++ erlaubt es auch (wie die meisten anderen Programmiersprachen), dass die gleiche Funktion im eigenen Funktionskörper aufgerufen wird. Dies bezeichnet man als **direkte Rekursion**.

Das folgende Programm erzeugt eine rekursive Struktur als Ausgabe:

```

$ cat rekursion1.cpp
#include <iostream.h>
void druck(int x, int l) {
    for (int i = 0; i <= l; i++)
        cout<< " ";
    cout << x << endl;
    if (x != 0)
        druck(x-1, l+1);
    for (int i = 0; i <= l; i++)
        cout<< " ";
    cout << x << endl;
}
main() {
    int x;
    cout << "Eingabe Integer: ";
    cin >> x;
    druck(x,0);
}
$

```

```

$ rekursion1.exe
Eingabe Integer: 4
4
 3
  2
   1
    0
    0
   1
  2
 3
4
$

```

**Wie ist das Programm zu ändern, so dass nur einer der Zweige dargestellt wird?**

**Indirekte Rekursion** liegt vor, wenn eine Funktion indirekt über eine Funktion aufgerufen wird, die sie selbst aufgerufen hat.

```
void proc1 (int i1);
void proc2 (int i2);
{ ...
  proc1 (...);
  ...
}
void proc1 (int i1);
{ ...
  proc2 (...);
  ...
}
```

Rekursion wird häufig verwendet, wenn

- eine Algorithmus rekursiv definiert ist oder
- wenn eine rekursive Datenstruktur zu implementieren ist (-> später).

## Beispiel (Berechnung Fakultät)

Algorithmus:

$$\text{fac}(0) = \text{fac}(1) = 1$$

$$\text{fac}(n) = n * \text{fac}(n - 1) \text{ für } n \in \mathbb{N}$$

```
$ cat fak.cpp
include <iostream.h>
int fak(int n) {
    if (n == 0 || n == 1)
        return(1);
    else
        return(n*fak(n-1));
}

main() {
    int x;
    cout << "Eingabe Integer x: ";
    cin >> x;
    cout << "fak(" << x << ") = " << fak(x) << endl;
}
$
```

## Hörsaalübung

1. Die **Fibonacci Zahlen** sind wie folgt definiert:

Die n-te Fibonacci Zahl ist die Summe der n-1-ten und der n-2-ten Fibonacci Zahlen, wobei die erste und zweite Fibonacci Zahl eins ist.

1,1,2,3,5,8,13,21,...

$$fibu(0) = fibu(1) = 1$$

D.h. 
$$fibu(n) = fibu(n-1) + fibu(n-2) \text{ für } n \geq 2$$

Schreiben Sie ein Programm, das die ersten n Fibonacci Zahlen ausgibt; n ist eine Benutzereingabe.

2. Entwickeln Sie eine rekursive Funktion

```
long suche(string s, char c)
```

die im String s nach dem Zeichen c sucht und den Wert des Index als Resultat liefert, für den gilt:  $s[j] = c$  oder  $j = -1$  falls  $c \notin s$

Ein **Beispiel** für eine **indirekte Rekursion** wird mit der Simulation eines Taschenrechners gezeigt.

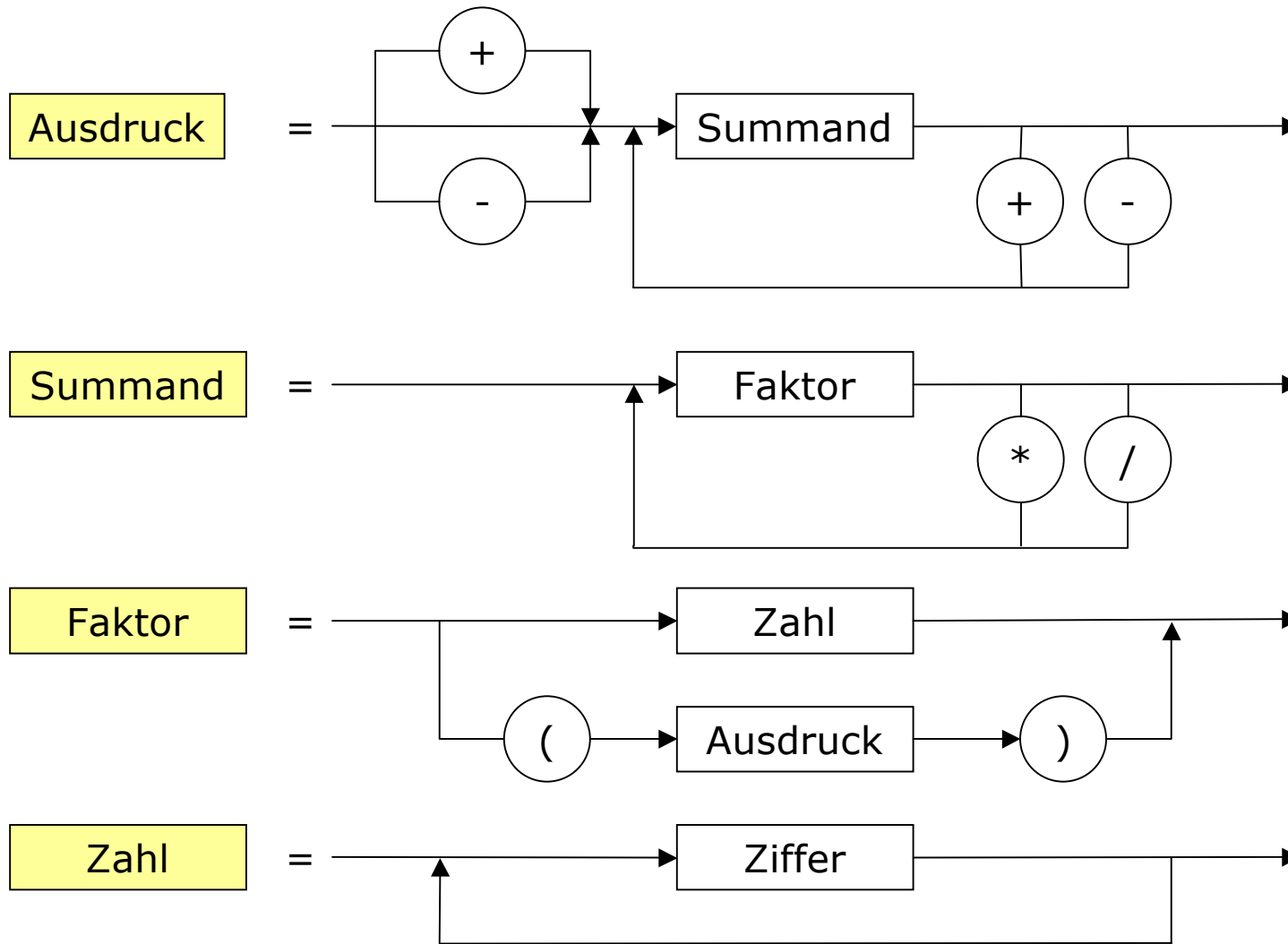
Aufgabe:

Ein Taschenrechner soll programmiert werden. Der Rechner soll dabei die vier Grundrechenarten mit Berücksichtigung von Prioritäten beherrschen.

Ein Benutzer soll einen Ausdruck in gewohnter Form mit Klammerung eingeben können, etwa der Form:  $(13+7) * 5 - (2 * 3 + 7) / (-8)$

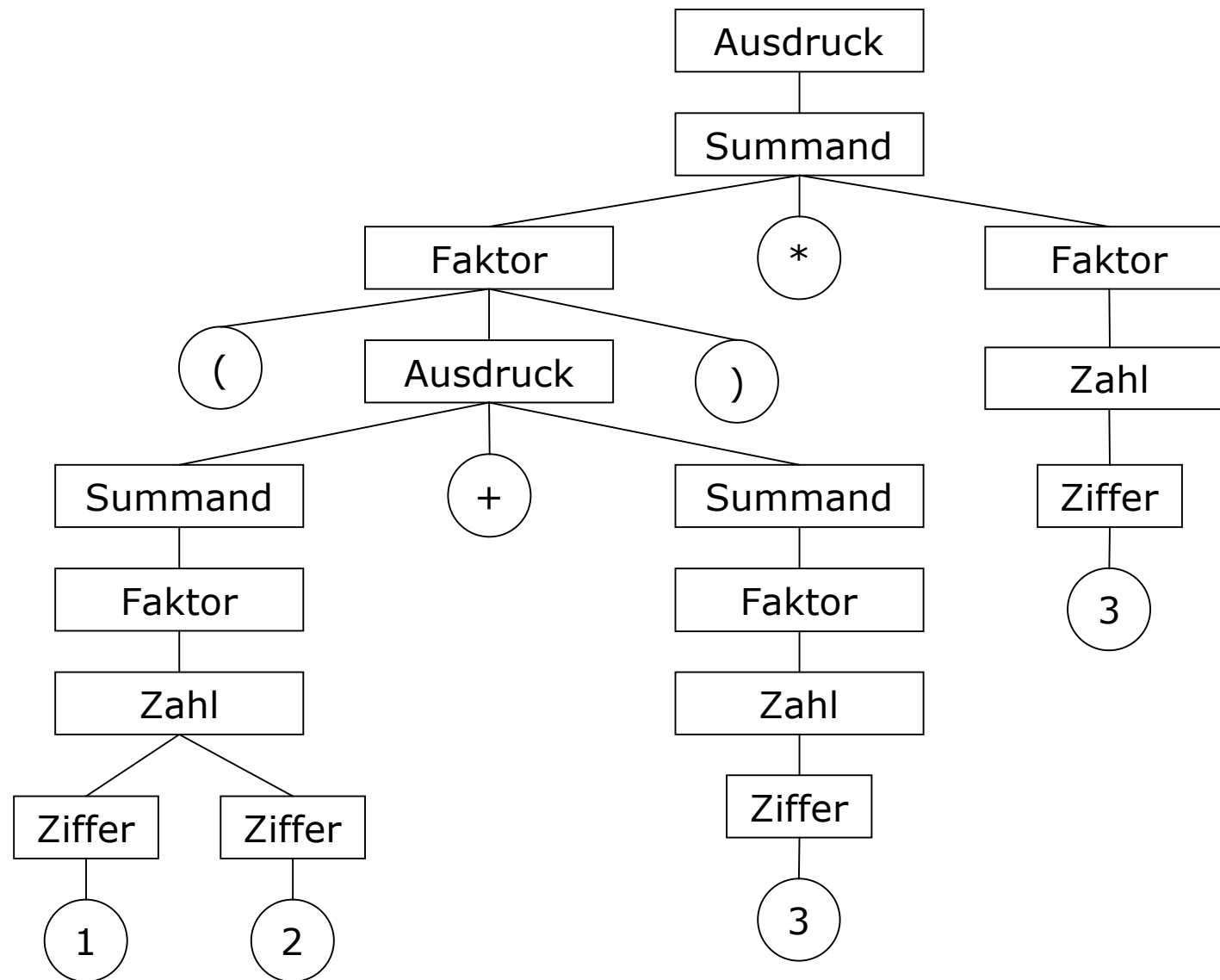
Methode:

Rekursiver Abstieg durch Verwendung der folgenden Syntax-Regel für mathematische Ausdrücke (Syntaxdiagramm für Ausdrücke)



Zur Vereinfachung erlauben wir keine Leerzeichen oder Tabulatoren im Ausdruck. Am Syntaxdiagramm wird deutlich, dass es sich um indirekte Rekursion handelt. Rekursiver Abstieg bedeutet, dass der Ausdruck bis zum Erreichen des Rekursionsendes aufgelöst werden soll.

Mit diesen Regeln kann die Struktur eines Ausdrucks durch einen Ableitungsbaum veranschaulicht werden, z.B. für  $(12+3) * 4$ :



## Lösung:

Das Programm wird so aufgebaut, dass für jede **Syntaxregel** eine **Funktion** geschrieben wird, die als Ergebnis den Wert des Teilausdrucks zurück liefert und dabei jeweils die Eingabezeichen konsumiert („weg frisst“), die zu ihrer Regel gehören.

Der Benutzer bekommt eine Prompt, der Anzeigt, dass Eingaben möglich sind. Der eingegebene Ausdruck wird mit RETURN abgeschlossen. Dann wird das Ergebnis ausgegeben.

Das Programm soll bei Eingabe von dem Zeichen „e“ und RETURN beendet werden.

## Beispieldialog:

```
$ rechner
> (12+3)*4
60
> 12+(-1
Rechte Klammer fehlt!
11
> e
$
```

## Hauptprogramm:

```
// ***** Hauptprogramm *****  
int main() {  
    char ch;  
    while(1) {  
        cout << "\n> "; // Prompt des Rechners  
        cin.get(ch);  
        if (ch != 'e')  
            cout << ausdruck(ch);  
        else break;  
    }  
} // main
```

`cin.get(ch)` ist eine vordefinierte Funktion, die das nächste Zeichen aus dem Tastaturpuffer einliest.

Die Funktion `ausdruck` wird mit dem gelesenen Zeichen aufgerufen.

# Ausdruck:

```

long ausdruck(char& c) {
    long a;
    if (c == '-') {
        cin.get(c);
        a = -summand(c);
    }
    else {
        if (c == '+')
            cin.get(c);
        a = summand(c);
    }
    while(c == '+' || c == '-')
        if (c == '+') {
            cin.get(c);
            a += summand(c);
        }
        else {
            cin.get(c);
            a -= summand(c);
        }
    }
    return a;
} // ausdruck

```

// Übergabe per Referenz!  
 // Hilfsvariable für Ausdruck  
 // - im Eingabestrom überspringen  
 // Rest an summand() übergeben

// + überspringen

// + überspringen

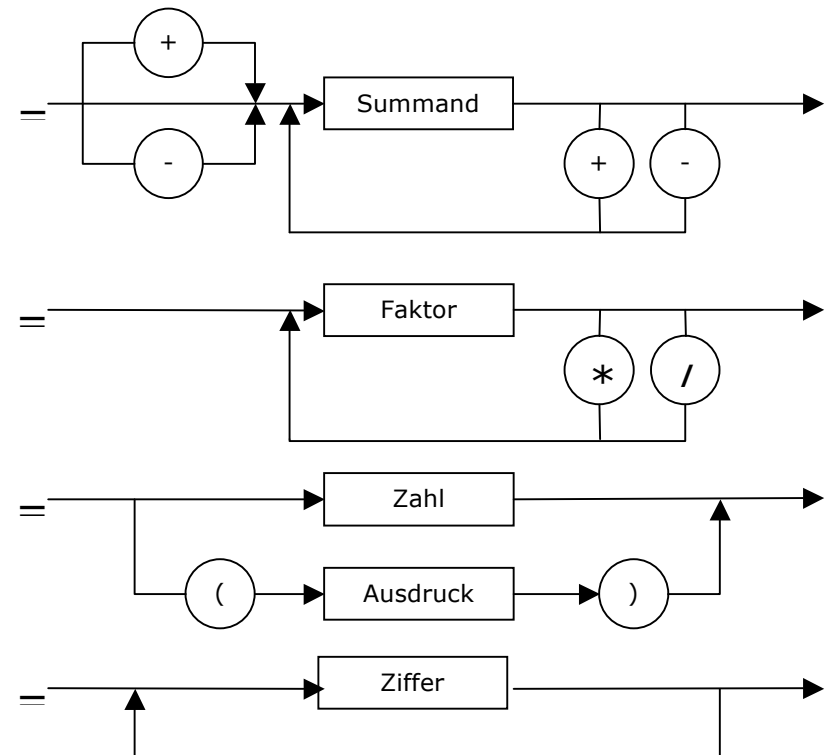
// - überspringen

Ausdruck

Summand

Faktor

Zahl



Da hier `summand` aufgerufen wird, und immer ein Zeichen der Eingabe konsumiert wird, muss die Rekursion enden und `a` ist das Ergebnis.

Hier erkennt man folgende generelle Vorgehensweise des Programms:

- eine Funktion für eine Regel analysiert ein Zeichen und delegiert den Rest an die Folgeregel gemäß der syntaktischen Struktur, wenn das Zeichen zu der Regel passt. D.h. hier im Ausdruck wird „+“ und „-“ konsumiert und der Rest an `summand` delegiert, dessen Ergebnis dann verwendet wird, um das Ergebnis des `ausdrucks` zurück geben zu können.
- Passt das Zeichen nicht zur Regel, wird die jeweilige Funktion verlassen – dann muss sich der Aufrufer darum kümmern.

## Summand:

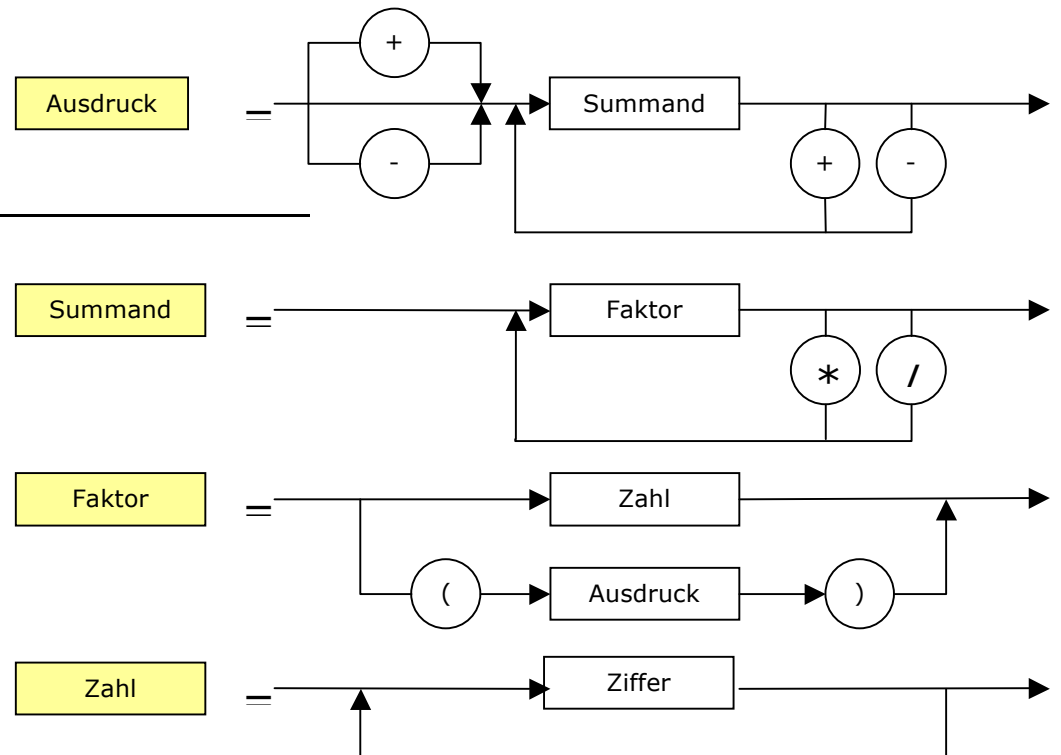
```

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

```

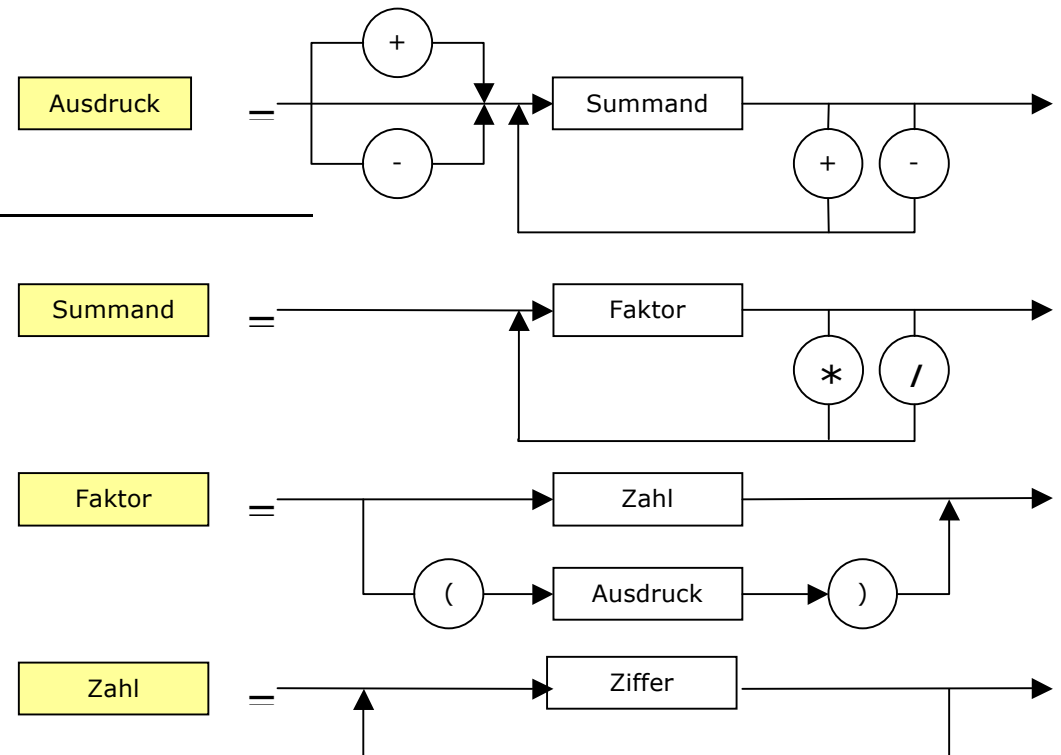
// \* überspringen

// / überspringen



## Faktor:

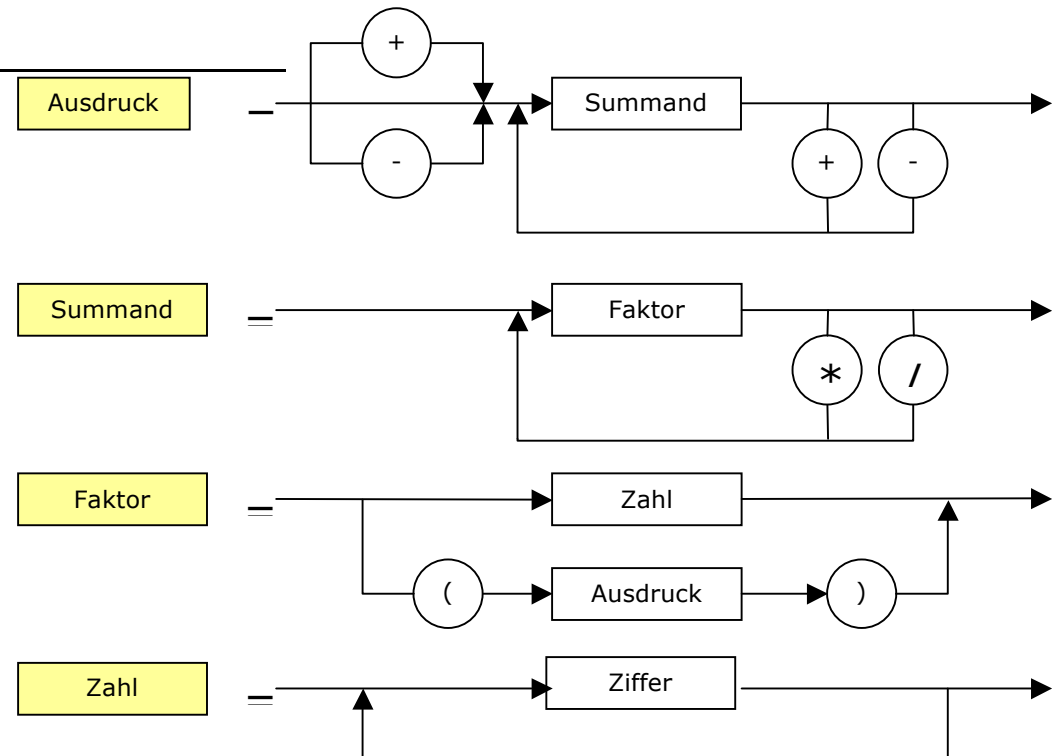
```
long faktor(char& c) {  
    long f;  
  
    if (c == '(') {  
        cin.get(c);           // ( überspringen  
        f = ausdruck(c);  
        if (c != ')')  
            cout << "Rechte Klammer fehlt!\n";  
        else cin.get(c);     // ) überspringen  
    }  
    else f = zahl(c);  
 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



## Zahl:

```
long zahl(char& c) {  
    long z = 0;  
    while (isdigit(c)) {  
        z = 10*z + long(c-'0');  
        cin.get(c);  
    }  
    return z;  
}  
// zahl
```

liefert aus Zeichen korrespondierende Ziffer



## 1.8. Überladen von Funktionen

In C++ können Funktionen überladen (engl. *overloading*) werden. Dies bedeutet, dass derselbe Funktionsname verwendet werden kann, um gleichartige Operationen mit Daten unterschiedlichen Typs ausführen zu können.

Dadurch werden mehrere Funktionen, die alle denselben Namen haben erzeugt. Die Entscheidung, welche der Funktionen ausgeführt wird, wird durch den **Aufrufkontext** bestimmt: die Funktionen müssen alle unterschiedliche **Signaturen** (=Reihenfolge und Typ der Parameter) haben. Die Signatur wird vom Compiler verwendet, um die Funktion zu bestimmen.

So werden im folgenden Beispiel drei Funktionen mit dem selben Namen realisiert, die das maximale Element eines Arrays ermitteln, aber sich in der Signatur unterscheiden.

Beispiel (Maximales Feld in Array):

```

$ cat maxElement.cpp
#include <iostream.h>
double maxElement(double daten[], int anzahl) {           // Fkt. 1
    double max=daten[0];
    for (int i=1; i<anzahl; i++)
        if (daten[i] > max)
            max=daten[i];
    return max;
}

int maxElement(int daten[], int anzahl) {               // Fkt. 2
    int max=daten[0];
    for (int i=1; i<anzahl; i++)
        if (daten[i] > max)
            max=daten[i];
    return max;
}

int maxElement(int daten[], int anzahl, int start) {    // Fkt. 3
    int max=daten[start];
    for (int i=start+1; i<anzahl; i++)
        if (daten[i] > max)
            max=daten[i];
    return max;
}

```

```
main() {  
    double doubleFeld[]={1.2, 3.4, 4.5, -6.7};  
    cout << maxElement(doubleFeld, 4) << endl;           // Fkt. 1  
  
    int intFeld[]={1, 3, 4, -6};  
    cout << maxElement(intFeld, 4) << endl;             // Fkt. 2  
  
    cout << maxElement(intFeld, 3, 1) << endl;         // Fkt. 3  
}  
$
```

```
$ maxElement  
4.5  
4  
4  
$
```

## 1.9. Funktion `main`

Ein C++ Programm (nicht objektorientierte Sicht) besteht aus einer Menge von Funktionen, von denen genau eine den Namen `main` hat.

Somit darf `main` nicht überladen werden. Weiterhin darf `main` nicht von einer anderen Funktion aus aufgerufen werden.

Der Ergebnistyp von `main` ist nicht fest vorgegeben; normalerweise ist es `int`. Damit kann die Laufzeitumgebung eines Programms den `return`-Wert von `main` auswerten. In Unix gibt es die Konvention, dass ein Programm im Fehlerfall einen von 0 verschiedenen Wert zurück liefert, ansonsten 0.

Die einfache Form der Funktion `main` ist:

```
$ cat main.cpp
#include <iostream.h>
int main() {
    int x;
        cout << "Positive Zahl: ";
    cin >> x;
    if (x>0)
        cout << sqrt(x) << endl;
    else
        return 1;

    return 0;
}
$
```

```
$ main
Positive Zahl: 2
1.41421
$ echo $?
0
$ main
Positive Zahl: -2
$ echo $?
1
$
```

Die Version von `main` mit Parametern, bei der auf die komplette Aufrufumgebung zugreifbar ist, werden wir später kennen lernen, nach dem Zeiger bekannt sind.

## 1.10. Ausnahmebehandlung über Funktionsgrenzen

Eine `throw`-Anweisung erzeugt bekanntlich ein Ausnahmeobjekt, das eine Fehlermeldung und/oder Daten zum Wiederaufsetzen enthält.

Das Ausnahmeobjekt wird in der Aufrufhierarchie zur Fehlerbehandlung weitergereicht:

- jede Funktion, die kein passendes `catch` enthält, wird ordentlich beendet,
- dann wird in der Aufrufumgebung weiter nach einem `catch` gesucht.

Damit ist die Fehlerbehandlung über mehrere Aufrufebenen hinweg möglich:

- eine Bibliotheksfunktion erkennt Fehler, kann sie aber nicht behandeln
- die Aufrufumgebung kann Fehler behandeln, aber nicht selbst erkennen.

Beispiel (Funktion wurzel „`caught`“ nicht alle Fehler)

```

$ cat ausnahme.cpp
#include <iostream.h>
float wurzel(int x, int maxWert) {
    try {
        if (x<0)
            throw "negative Diskriminante";
        if (x > maxWert)
            throw x-maxWert;
    } catch (const char * text) {
        cout << "Fehler: " << text << endl;
        return -1;
    } // kein catch für int !!
    return sqrt(x);
}

main() {
    int zahl; float ergebnis;
    cout << "Zahl eingeben: ";
    try { cin >> zahl;
        ergebnis = wurzel(zahl, 100);
        cout << ergebnis << endl;
    } catch (int wert) {
        cout << "Fehler: Wert um " << wert << " zu groß" << endl;
    }
}
$

```

```

$ ausnahme
Zahl eingeben: 2
1.41421
$ ausnahme
Zahl eingeben: -2
Fehler: negative Diskrimi-
nante
-1
$ ausnahme
Zahl eingeben: 105
Fehler: Wert um 5 zu groß
$

```

## 1.11. Die Funktion als Vertrag

Eine Funktion führt eine Teilaufgabe aus und ändert dabei den Zustand eines Programms.

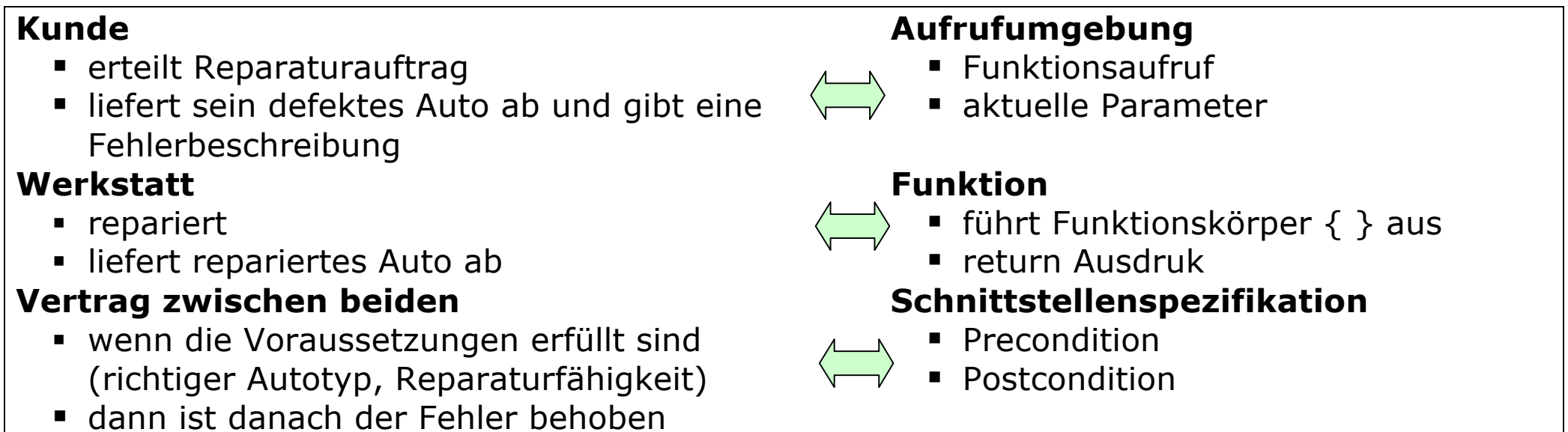
In guten Programmen sind die Bedingungen, die vor (Precondition) und nach (Postcondition) dem Funktionsaufruf erfüllt sind, im Programmkopf dokumentiert.

Sind die Bedingungen nicht erfüllt, sollte ein Programm geordnet abbrechen.

Diese Sichtweise kann als **Vertrag** aufgefasst werden.

### **Schnittstellenspezifikation = Vertrag zwischen Aufrufer und Funktion**

Beispiel Autoreparatur:



Pre- und Postcondition beschreiben "**von außen sichtbare**" **Bedingungen**:

- PRE: betrifft **aktuelle Parameter** (und **globale** Daten)
- POST: betrifft **aktuelle Parameter**, (**globale** Daten) und **Ergebnis**

PRE und POST sind Teil der **Funktionsbeschreibung**:

- können umgangssprachlich formuliert werden
- besser ist aber: formale Spezifikation, logischer Ausdruck

Die **Korrektheit** einer Funktion ist dann:

- **wenn PRE eingehalten** ist, **dann gilt** nach Ausführung auch **POST**

Noch besser als Korrektheit ist **Robustheit**:

- **Verletzung** der **Precondition** führt nicht zum Absturz, sondern
- zu **definiertem Verhalten** für jeden möglichen Parametersatz

## Beispiel (Fragment)

```
#include <assert.h>
```

```
double DritteWurzel (double x)
```

```
// liefert die Kubikwurzel von x
```

```
// PRE: x >= 0
```

```
// POST: Ergebnis = dasjenige y mit  
//          abs(y3-x) <= 10-4 und y >= 0
```

```
{
```

```
assert (x>=0.0);
```

```
double y;
```

```
// hier folgt der Algorithmus zur ...
```

```
// ... Berechnung der Kubikwurzel
```

```
assert ((fabs(y*y*y-x)<=1e-4) && (y>=0.0));
```

```
return y;
```

```
}
```

der Name "Wurzel" wäre eine  
**ungenau**e Beschreibung

mathematisch oder  
umgangssprachlich

PRE: programmiersprachlich

PRE: programmiersprachlich

Mit Zusicherungen (engl. *assertion*) werden im Entwicklungsstadium Programmierfehler abgefangen.

Die Verletzung einer assertion bewirkt das kontrollierte Beenden des Programms. Die Einhaltung von PRE und POST wird mit `assert` überprüft.

Es ist eine **reine Testhilfe**, damit sollen Programmierfehler abgefangen werden, es hilft nicht gegen fehlerhafte Benutzereingaben.

### Hörsaalübung:

Nehmen Sie die u.A. Funktion `suche` und überlegen Sie sich PRE- und POST-Condition.

Fügen Sie diese mittels `assert` in die Funktion ein und Testen Sie mittels eines Hauptprogramms.

```
long suche(char a[], char c, long pos, long maxPos) {  
    if (pos >=maxPos)  
        return -1;  
    else if (a[pos] == c)  
        return pos;  
    else  
        return suche(a, c, pos+1, maxPos);  
}
```

## 1.12. Dokumentation und Testen

### 1.12.1. Dokumentationsregeln

#### Funktionskopf

- Funktionsnamen spiegeln die eigentliche Aufgabe wider
  - wo der Name nicht ausreicht, wird eine umgangssprachliche Kurzbeschreibung als Kommentar im Funktionskopf eingefügt
- PRE und POST spezifizieren Details und Einschränkungen

#### Funktionskörper

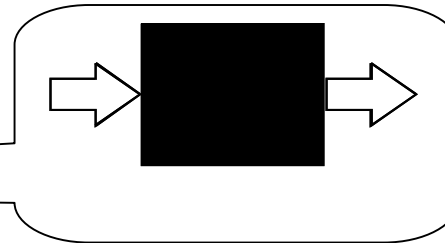
- assert / throw an "strategisch wichtigen" Stellen plazieren
- sonstige Kommentare erläutern Vorgehensweise und Algorithmen, soweit nicht offensichtlich
  - nicht Informationen des Programmtextes duplizieren

## 1.12.2. Testen

Testen ist praktisch immer **unvollständig**, deshalb muss man sich die Testfälle sorgfältig überlegen.

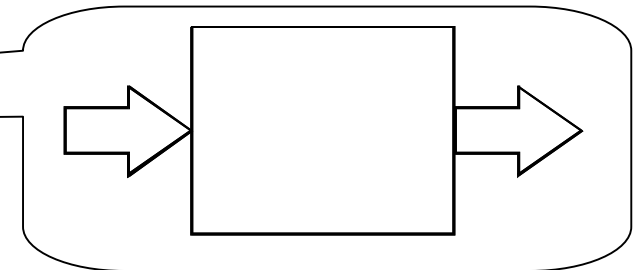
Testen bezüglich des zulässigen Wertebereichs:

- „Black Box Test“
- erstes und letztes Element und eins in der Mitte ⇒ Korrektheit
- unzulässiges Element oberhalb und unterhalb ⇒ Robustheit



Testen bezüglich der inneren Struktur

- „White Box Test“
- jeden Programmpfad mindestens einmal durchlaufen
- Schleifen: 0, 1, n-maliger Durchlauf
- switch: undefinierte case Marke



## 2. Modulare Gestaltung von Programmen

Eine **Anwendung** besteht i.A. **aus mehreren Dateien**, die jeweils einzeln übersetzt und getestet werden. Am Ende der Programmentwicklung werden dann die einzelnen Teile erst zu einer Anwendung zusammen geführt.

Jedes **Programmierteam** ist dann für die Entwicklung und den Test eines **Moduls** verantwortlich. Ein Projektleiter bzw. ein Teamkoordinator sorgt für das Zusammenfügen der einzelnen Modulen und organisiert einen Integrationstest.

Damit jede Datei einzeln übersetzt werden kann, müssen Konstanten, Funktionsprototypen (Klasseninterfaces) etc. bekannt sein. Dies wird möglich, indem eine Datei andere Dateien inkludiert (`#include`).

Folgender **Aufbau** hat sich bewährt:

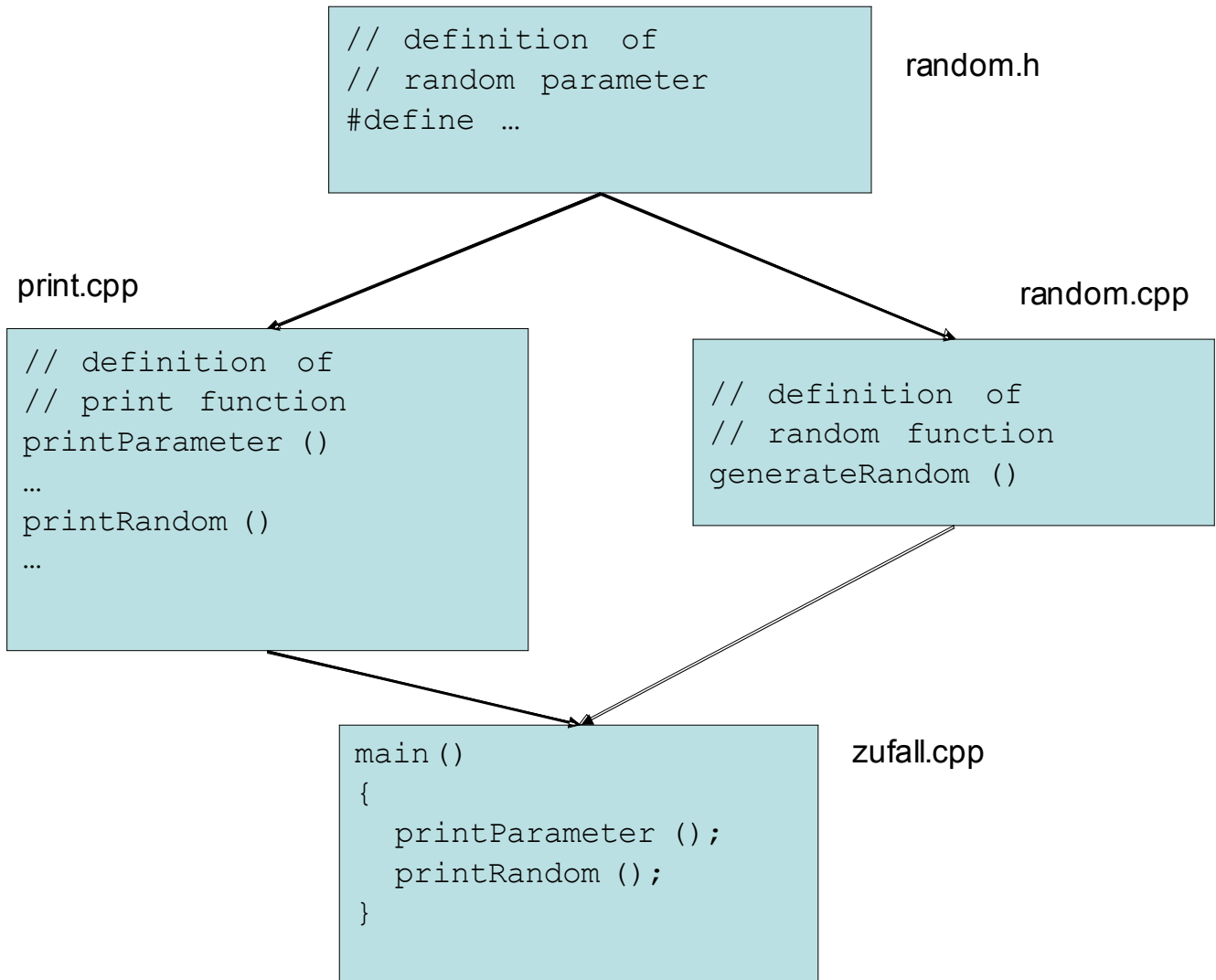
- In Dateien mit Endung **.h (Header Dateien)** sollten Konstanten, Schnittstellenbeschreibungen, Deklarationen, Funktionsprototypen und (wenn unbedingt notwendig) globale Daten abgelegt werden. (Später werden wir sehen, auch Klassendeklarationen)
- **Implementierungsdateien** (da steht der eigentliche Code drin) haben die Endung **.cpp**.
- Eine Datei mit der Funktion `main` erhält den Anwendungsnamen mit Endung **.cpp**.

## 2.1. Getrenntes Übersetzen von Programmen

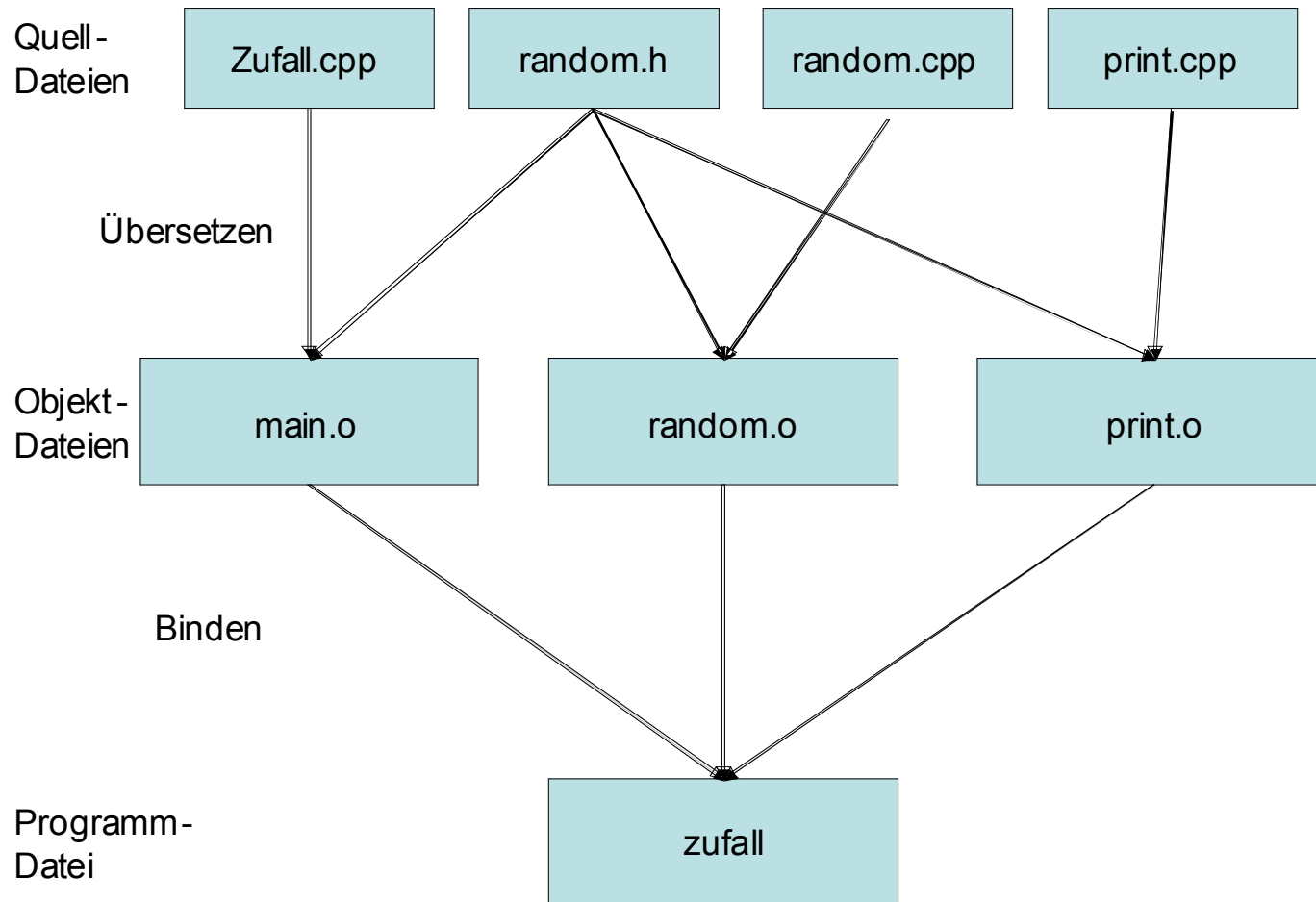
Den Aufbau und die getrennt Übersetzung verdeutlichen wir am Beispiel einer „Anwendung“, die Zufallszahlen erzeugt.

Ein Programm „zufall“ soll eine Folge von 10 Zufallszahlen auf den Bildschirm schreiben. Verwendet werden soll der Algorithmus von Knuth.

Das Programm soll folgende modulare Struktur aufweisen:



Neben den „logischen“ Abhängigkeiten, gibt es bei C-Programmen die Abhängigkeiten:



## Programm-Dateien:

```
$ cat random.h
const long R=1000001;
const long a=100001;
const long m=1717;
const long c=3;
$
```

```
$ cat random.cpp
#include "random.h"
float generateRandom(void) {
    static long int r=R;
    r = (a*r)%m;
    return r/(float)m;
}
$
```

```
$ cat print.cpp
#include "random.h"
#include <iostream.h>

void printParameter(void) {
    cout << "a=" << a << endl;
    cout << "c=" << c << endl;
    cout << "m=" << m << endl;
}

void printRandom(float i) {
    cout << i << endl;
}
$
```

```
$ cat zufall.cpp
extern void printParameter(void);
extern void printRandom(float);
extern float generateRandom(void);

main()
{
    printParameter();
    for (int i=1; i<=10; i++)
        printRandom(generateRandom());
}
$
```

Ein Unix makefile zeigt die Abhängigkeiten und die einzelnen Kommandos zum Übersetzen. (In Windows funktioniert das so ähnlich). Durch das Kommando „make“ wird das Makefile gelesen, die darin befindlichen definierten Abhängigkeiten erkannt und entsprechend die Kommandos ausgeführt (-> vgl. Veranstaltung „Unix für Entwickler“).

```
$ cat Makefile
zufall: zufall.o print.o random.o
    g++ zufall.o print.o random.o -o zufall
zufall.o: zufall.cpp
    g++ -c zufall.cpp -o zufall.o
print.o: print.cpp
    g++ -c print.cpp
random.o: random.cpp
    g++ -c random.cpp
clean:
    rm *.o
$
```

## 2.2. Dateiübergreifende Gültigkeit und Sichtbarkeit

Variablen in C++ gehören zu einer Speicherklasse. Eine Speicherklasse bestimmt die Gültigkeit und die Sichtbarkeit und ev. den Speicherort. Die Speicherklasse wird bei der Variablendeklaration angegeben.

**Globale** Variablen werden außerhalb aller Funktionen angegeben. Sie sind in allen Teilen des Programms gültig, auch in anderen Dateien. In einer anderen Datei muss die Variable aber als extern deklariert werden. Dadurch wird kein neues Objekt angelegt, es wird nur bekannt gemacht (für den Compiler)

Beispiel

```

$ cat datei1.cpp
#include <iostream.h>
int global;           // Deklaration und Definition
void fkt();
main() {
    global = 18;
    cout << global << endl;
    fkt();
}
$ cat datei2.cpp
#include <iostream.h>
extern int global;   // Deklaration, aber keine Definition
void fkt() {
    global = 42;
    cout << global << endl;
}
$

```

Beide Dateien können getrennt übersetzt werden und danach zusammen gebunden werden:

```
$ g++ -c datei1.cpp
```

```
$ ls -l
```

```
datei1.o
```

Übersetzen

```
...
```

```
$ g++ -c datei2.cpp
```

```
ls -l
```

```
datei2.o
```

Binden

```
...
```

```
$ g++ datei1.o datei2.o -o bsp1
```

```
$ bsp1
```

```
18
```

```
42
```

```
$
```

Soll die Gültigkeit auf eine Datei beschränkt werden, wird bei der Definition das Schlüsselwort `static` vorangestellt.

```
$ cat datei3.cpp
#include <iostream.h>
static int global;           // nicht mehr global!
void fkt();
main() {
    global = 18;
    cout << global << endl;
    fkt();
}
$
```

Jetzt kann `datei2.cpp` und `datei3.cpp` zwar noch einzeln übersetzt werden. Beim Binden wird aber ein Fehler angezeigt, da die Variable `global` nicht in `datei2.cpp` bekannt ist.

```
$ g++ datei3.o datei2.o -o bsp2
datei2.o: In function `ostream::_ls(ostream &(*) (ostream &))':
/usr/include/g++-3/iostream.h:106: undefined reference to `global'
collect2: ld returned 1 exit status
$
```

Auf Dateiebene außerhalb aller Funktionen definierte **Variable** sind global und in andere Dateien benutzbar, wenn sie dort als extern deklariert sind.

**Konstanten** sind nur in der Definition sichtbar. Sollen sie anderen Dateien zugänglich gemacht werden, **müssen** sie **explizite** als **extern** definiert sein.

```
// Datei4.cpp
extern const float pi = 3.14159;    // Definition
...

// Datei5.cpp
extern const float pi;              // Deklaration (Initialisierung in Datei4.cpp)
...
```

Alle Variablen die innerhalb eines Blockes definiert werden, heißen **auto** Variablen, da sie beim Betreten des Blocks angelegt (nicht initialisiert) und beim Verlassen des Blocks automatisch zerstört werden. Sie können, müssen aber nicht mit dem Schlüsselwort **auto** gekennzeichnet werden.

### **2.3. Compilerdirektive und Makros**

Am normalen Übersetzungsprozess sind folgende Programme beteiligt, die in der angegebenen Reihenfolge ausgeführt werden:

1. Präprozessor
2. Compiler
3. Binder

Mit Compilerdirektiven kann der Präprozessor gesteuert werden. Sie werden durch das Zeichen **#** am Zeilenanfang eingeleitet.

### 2.3.1. #include

Mit `include` können Dateien in eine Datei hineinkopiert werden. Dabei kann der Pfadname absolut und relativ angegeben werden:

```
#include "Datei18.h"           // Datei im aktuellen Verzeichnis
#include "../Datei18.h"       // Datei im Väterverzeichnis

#include /home/as/Vorlesungen/ProgI/Programmstruktur/bsp18.h // absoluter Pfad
```

Sollen Dateien aus den **include-Verzeichnissen** der Programmierumgebung sind in **spitze** Klammern einzuschließen:

```
#include <iostream.h>
```

### 2.3.2. #ifdef, #ifndef, #define

in allen Dateien, die zu einem Programm gehören darf es beliebig viele Deklarationen, aber nur genau eine Definition geben.

Es kann zu Übersetzungsproblemen kommen, wenn eine Header-Dateien mehrfach inkludiert wird und die Header-Datei eine Definition enthält.

## Beispiel:

```
$ cat a.h
#include <iostream.h>
int global=1;
$
```

↓

```
$ cat c.cpp
#include "a.h"
void fkt() {
    global = 42;
    cout << global << endl;
}
$
```

→

```
$ cat b.cpp
#include "a.h"
#include "c.cpp"
main() {
    global = 18;
    cout << global << endl;
    fkt();
}
$
```

Der Versuch, das Programm zu übersetzen scheitert:

```
$ g++ -c c.cpp
$ g++ -c b.cpp
In file included from c.cpp:1,
    from b.cpp:2:
a.h:2: redefinition of `int global'
a.h:2: `int global' previously defined here
$
```

Abhilfe schafft die Möglichkeit des bedingten Definierens mit `#if defined (#ifdef)` und `#if !defined (#ifndef)`.

Die Header-Datei wird erweitert um eine bedingte Definition

```
$ cat a1.h  
#include <iostream.h>
```

```
#ifndef globaleDefinitionen  
#define globaleDefinitionen
```

```
int global=1; // Deklaration und Definition
```

```
#endif // globaleDefinitionen
```

```
$
```

falls der Name `globaleDefinition` nicht definiert ist, dann definiere ihn und akzeptiere alles bis `#endif`

```
$ cat c1.cpp  
#include "a1.h"  
void fkt() {  
    global = 42;  
    cout << global << endl;  
}  
$
```

```
$ cat b1.cpp  
#include "a1.h"  
#include "c1.cpp"  
main() {  
    global = 18;  
    cout << global << endl;  
    fkt();  
}  
$
```


Damit kann das Programm übersetzt und gebunden werden:

```
$ g++ -c c1.cpp
$ g++ -c b1.cpp
$ g++ -o c1.o b1.o -o bsp2
$ bsp2
18
42
$
```

### 2.3.3. Makros

Durch `#define` können neben symbolische Konstanten allgemeine Textersetzungen spezifiziert werden, die vom Präprozessor ausgeführt werden.

Beispiel

|  |   |
|--|---|
| <pre>\$ cat pi.cpp #include &lt;iostream.h&gt; #define PI 3.14 main() {     cout &lt;&lt; PI &lt;&lt; endl; } \$</pre> |  <pre>#include &lt;iostream.h&gt; main() {     cout &lt;&lt; 3.14 &lt;&lt; endl; }</pre> |
|--|---|

Ein weiteres (nicht zu empfehlendes) Beispiel für Pascal-Programmierer:

|   |  |
|---|--|
| <pre>#define then #define begin { #define end ;}  if (i&gt;0)   then begin     a = 1;     b = 2   end</pre> | <pre>if (i&gt;0)   {     a = 1;     b = 2   ;}</pre> |
|---|--|

Solche Makros können parametrisiert werden:

|  |  |
|--|--|
| <pre>#define max(A,B) ((A)&gt;(B)?(A):(B))  x = max(y, 2+x);</pre> | <pre>X = ((y)&gt;(2+x)?(y):(2+x));</pre> |
|--|--|

Dadurch lässt sich eine „Funktion“ realisieren, die für unterschiedliche Datentypen verwendbar ist.

### **Achtung:**

Die ist eine Fehlerquelle!

```
$ cat quad.cpp
#include <iostream.h>

#define QUAD(X) ((X)*(X))

main() {
    int z = 2;
    cout << QUAD(++z) << endl;
}
$
```

es wird **nicht** z erhöht und dann quadriert, sondern ++z wird **zwei mal** ausgeführt!

```
$ quad
16
$
```

Ein eher sinnvolles Beispiel der Verwendung von Makros ist es Testsequenzen ein- und auszublenen.

Zunächst der Schritt der Programmentwicklung, also mit Testausgaben (Beispiel Fakultät):

```
$ cat fak.cpp
#include <iostream.h>
#define TEST_EIN
#ifdef TEST_EIN
    #define PRINT(WERT) cout << (WERT) << endl;
#else
    #define PRINT(WERT) /* nichts */
#endif

int fak(int n) {
    if (n == 0 || n == 1) {
        PRINT(1);
        return(1);
    }
    else {
        int erg = n*fak(n-1);
        PRINT(erg);
        return(erg);
    }
}
```

```
main() {
    int x;
    cout << "Eingabe Integer x: ";
    cin >> x;
    cout << "fak(" << x << ") = " << fak(x) << endl;
}
$
```

```
$ fak
Eingabe Integer x: 3
1
2
6
fak(3) = 6
$
```

Am Ende der Programmentwicklung und des erfolgreichen Testens wird einfach die `#define`-Zeile auskommentiert; dadurch wird der leere String beim Ersetzen verwendet:

```
// #define TEST_EIN
```

Somit sind Testausgaben (nach Neuübersetzung) unterdrückt.

```
$ fak
Eingabe Integer x: 3
fak(3) = 6
$
```

### 3. Funktionstemplates

Bei der Definition von Funktionen ist der Ergebnistyp und der Typ der formalen Parameter anzugeben.

Soll eine Funktion für unterschiedliche Typen realisiert werden, können **Templates** verwendet werden. Dabei wird anstelle der Typangabe ein Platzhalter verwendet.

Allgemeine Form:

```
template < typename TypBezeichner > Funktionsdefinition
```

Anstelle von `typename` kann auch `class` stehen (-> später)

Beispiel (Bubblesort für int und double-Arrays)

```

$ cat bubbleSort.cpp
#include <iostream.h>
#include <string>

template <typename T>
void bubbleSort(T daten[], int anzahl) {

    for (int lauf=1; lauf <anzahl; lauf++) { // Läufe
        for (int element=anzahl-1; element>=lauf; element--) { // ein Lauf
            if (daten[element-1] > daten[element]) { // vertauschen
                T tmp = daten[element-1];
                daten[element-1] = daten[element];
                daten[element] = tmp;
            }
        }
    }
}

```

```

main() {
    char charDaten[] = "deacb";
    // Sortieren der Daten
    bubbleSort(charDaten, 5);
    // Ausgeben
    cout << charDaten << endl;

    double doubleDaten[] = {3.1, 8.9, -2.0, 0.5};
    // Sortieren der Daten
    bubbleSort(doubleDaten, 4);
    // Ausgeben
    for (int i=0; i<4; i++)        cout << doubleDaten[i] << " ";
    cout << endl;
}
$

```

```

$ bubbleSort
abcde
-2 0.5 3.1 8.9
$

```

## 4. Standardfunktionen/Bibliotheken

C++ zeichnet sich durch eine Vielzahl von verfügbaren Bibliotheksfunktionen aus (-> Klassen). Dies Funktionen gehören nicht zur Sprache, werden aber in allen C++ Umgebungen bereitgestellt.

Wir werden im Verlauf der Veranstaltung noch einige davon kennen lernen.

## Hörsaalübung

Entwickeln Sie eine Funktion als Template

```
long suche(T a[], T x)
```

die im Array a nach der Komponente x sucht und den Wert des Index als Resultat liefert, für den gilt:  $a[j] = x$ . Wenn x nicht in a enthalten ist, soll das Ergebnis -1 sein.

Entwickeln Sie ein dazu passendes Testprogramm für je ein Integer-Array und ein Array von Zeichen.