Funktionen

Es sollen hier nur Eigenschaften von Funktionen behandelt werden, die typischerweise auch in anderen imperativen Programmiersprachen anzutreffen sind. Die Syntax der Funktionsvereinbarung und die Umwandlung von Funktionen in Funktionszeiger wird in Programmieren II behandelt.

Funktionswert

Funktionen mit dem Ergebnistyp `void` liefern *keinen* Funktionswert und werden durch eine leere `return-Anweisung` oder durch Erreichen des Blockendes beendet.

Alle anderen Funktionen wird der Funktionswert durch das Argument der `return`-Anweisung zugewiesen. Dieses Argument wird in den Ergebnistyp der Funktion umgewandelt, sofern mittels impliziter Typumwandlung möglich.

C-Vektoren und Funktionen sind als Funktionswerte unzulässig.

Funktionsaufruf

Dieser erfolgt durch Verwendung in Ausdrücken mit eingesetzten Argumenten, z.B. `x = f(2); g(x,y); h(); .`

Argumente werden entsprechend den Typangaben in der Parameterliste umgewandelt.

Die Reihenfolge der Argumentauswertung ist *nicht* festgelegt, jedoch sind alle Seiteneffekte eingetreten, bevor die Funktionsausführung begonnen wird.

Vorsicht:
```
h(); // Funktionsaufruf bei leerer Argumentliste
h;   // Funktionsadresse als Wert des Ausdrucks (spaeter)
```

Bsp.: Reihenfolge der Argumentauswertung

Programm:

```
#include <iostream>

using namespace std;

void f(int i1, int i2)
{
  cout << "Fertig." << endl;
}

int g(int i)
{
  cout << "Argument " << i << " ausgewertet." << endl;
  return i;
}

int main()
{
  f(g(1),g(2));
  return 0;
}
```

Ausgabe:

g++-3.4 (Solaris 10 sparc)	*g++-9.4 (Ubuntu Linux 20.04, amd64)*
`Argument 1 ausgewertet.`	`Argument 2 ausgewertet.`
`Argument 2 ausgewertet.`	`Argument 1 ausgewertet.`
`Fertig.`	`Fertig.`

Bei überladenen Operatoren (z.B. Shiftoperatoren für Ein/Ausgabe) muss mit ähnlichen Effekten gerechnet werden, weil die Überladung mittels Operatorfunktionen realisiert wird.

Bsp.: Reihenfolge der Operandenauswertung

Programm:

```
#include <iostream>

using namespace std;

int f(int i)
{
  cout << i << '*' << endl;
  return i;
}

int main()
{
  cout << f(1) << " " << f(2) << endl;
  return 0;
}
```

Ausgabe:

g++-9.4 Ubuntu Linux 20.04, amd64	*g++-4.9 Debian Linux, amd64*
`1*`	`2*`
`1 2*`	`1*`
`2`	`1 2`

Parameterliste

Leere Parameterlisten sind zulässig, sie können auch mit `void` vereinbart werden.

Der Gültigkeitsbereich der Parameternamen ist der Funktionsblock.

Wertparameter

Parameter mit Ausnahme von Funktionen, C-Vektoren und Referenzparameter sind *Wertparameter*, d.h. ihre Werte werden beim Funktionsaufruf auf gleichnamige (implizit definierte) lokale Variablen in der Funktion kopiert. (Vereinbart man eine Variable innerhalb des Funktionsblocks, die denselben Namen wie ein Parametername trägt, so überdeckt der Variablenname den Parameternamen.)

Für einen Wertparameter kann nicht nur eine Variable, sondern auch ein Ausdruck (insbesondere eine Konstante) eingesetzt werden, wenn dieser implizit in den Datentyp des Wertparameters umgewandelt werden kann. Änderungen an einem Wertparameter haben keinen Einfluss auf den Wert einer eingesetzten Variablen.

Bsp.: Diese Vertauschungsfunktion funktioniert nicht

```
void vertausche(int i, int j)
{ int h;
  h=i; i=j; j=h;
  return;
}

int main()
{ int i=2,j=4;
  vertausche(i,j); // i,j unveraendert
  return 0;
}
```

Referenzparameter

Durch Anhängen des Zeichen `&` an den nicht konstanten Datentyp eines Parameters wird dieser zu einem Referenz- oder Variablenparameter. Referenzparameter werden beim Funktionsaufruf *nicht* kopiert, statt dessen wird direkt der Speicherplatz der eingesetzten Variablen verwendet. Konstanten dürfen für einen Referenzparameter *nicht* eingesetzt werden.

Bsp.: Diese Vertauschungsfunktion funktioniert

```
void vertausche(int& i, int& j)
{ int h;
  h=i; i=j; j=h;
  return;
}
```

Konstante Referenzparameter

Wenn in der Parameterliste `T x` einen Wertparameter vom Datentyp `T` und `T& x` einen Referenzparameter für Variablen vom Datentyp `T` vereinbaren würde, dann vereinbart `const T& x` einen konstanten Referenzparameter. Auch hier wird direkt der Speicherplatz einer eingesetzten Variablen benutzt, ihr Wert darf aber *nicht* durch die Funktion verändert werden. Im Unterschied zu Referenzparametern ist auch das Einsetzen von Konstanten oder Ausdrücken (wie bei Wertparametern) zulässig, ggf. wird vom Compiler eine Hilfsvariable angelegt.

Konstante Referenzparameter kommen zum Einsatz, wenn eingesetzte Argumente auf Grund ihrer Größe nicht kopiert und auch nicht verändert werden sollen.

Bsp.: Euklidische Norm

```
double euklidische_norm(const vector<double>& a)
{
  double s=0;
  for (vector<double>::size_type i=0; i<a.size(); ++i)
    s += a[i]*a[i];
  return sqrt(s);
}
```

Funktionen als Parameter

Das Weglassen des Funktionsblocks in einer Funktionsdefinition führt zu einem Parameter, für den eine gleichartige Funktion eingesetzt werden kann. (Dabei ist zusätzlich möglich, in der Parameterliste dieses Funktionsparameters nur die Datentypen aufzuführen.)

Im Funktionsaufruf einer Funktion, die eine Funktion als Parameter enthält, erscheint nur der Funktionsname ohne Parameterliste.

Bsp.: Sehnentrapezregel

```
double sehnentrapez(double a, double b, int n, double g(double))
{
  double s, h=(b-a)/n;
  s = (g(a)+g(b))/2.0;
  for (int i=1; i<n; ++i) s+=g(a+i*h);
  return s*h;
}

double f(double x)
{
  return exp(-x*x);
}

int main()
{
  double a,b; int n;
  cout << "a b n: "; cin >> a >> b >> n;
  cout << "sehnentrapez(a,b,n,f)   = " << sehnentrapez(a,b,n,f)   << endl;
  cout << "sehnentrapez(a,b,n,sin) = " << sehnentrapez(a,b,n,sin) << endl;
  return 0;
}
```

Rekursive Funktionen

Selbstrekursive Funktionen

Eine Funktion kann sich in ihrem Rumpf (Funktionsblock) auch selbst ein- oder mehrmals wieder aufrufen. Das ist syntaktisch zulässig, weil ihr Name ab dem Vereinbarungspunkt (Ende der Parameterliste) bis zum Ende der Übersetzungseinheit sichtbar ist (Gültigkeitsbereich → Inf.bl.2, S.3). Jeder Aufruf der Funktion (sofern noch nicht beendet) hat seinen eigenen Satz lokaler Variablen und Wertparameter. Rekursive Funktionen sollten bei zeit- oder speicherkritischen Programmen möglichst durch Funktionen ersetzt werden, die stattdessen iterativ arbeiten. Das ist auch zu empfehlen, weil die Stackgröße und die Verschachtelungstiefe bei Funktionsaufrufen beschränkt sein können.

Bsp.: Fibonacci-Zahlen als rekursive Funktion (ineffizient)

```
long fib(int n)
{
  if (n==1 || n==2)
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
```

Wechselseitig rekursive Funktionen

Eine Rekursion kann auch indirekt entstehen, wenn sich mehrere Funktionen wechselseitig aufrufen. In diesem Fall ist es syntaktisch notwendig, zunächst einzelne Namen der beteiligten Funktionen einzuführen ohne diese zu definieren. Das wird durch *Funktionsdeklarationen* geleistet. Diese entstehen durch Ersetzen des Funktionsblocks durch einen Strichpunkt. (Namen in der Parameterliste der Funktion können weggelassen werden.)

Bzgl. der Verwendung gelten dieselben Empfehlungen wie für selbstrekursive Funktionen.