§5 Allgemeine Aspekte von Funktionen

Leitidee: *In diesem Abschnitt sollen nur diejenigen Eigenschaften von Funktionen behandelt werden, die auch in anderen modernen imperativen Programmiersprachen anzutreffen sind.*

- Überblick
- Parameter - Übersicht
- Wertparameterübergabe (Bsp.)
- Referenzparameterübergabe (Bsp.)
- Funktionen als Parameter: Sehnentrapezregel
- Modellhafte Implementierung des Funktionsaufrufs
- Speicherlayout bei der Funktionsausführung
- Rekursive Funktionen
- Auswertung arithmetischer Ausdrücke

Funktionen - Überblick

- Funktionswert festlegen: `return` *Ausdruck*
  (Umwandlung entspr. Ergebnistyp wie bei Zuweisung)
- Funktionen ohne Ergebnis (Ergebnistyp `void`): `return`
- Unzulässig: C-Vektoren und Funktionen als Fkt.werte
  Zulässig: Entsprechende Zeiger *(später!)*
- Typumwandlung der Argumente entspr. Parametertyp

Caveat

- Reihenfolge der Argumentauswertung nicht festgelegt, implementierungsabhängig $\rightarrow$ *Inf.bl.10, S.1/2*
- `f();` Funktionsaufruf bei leerer Argumentliste
  `f;` kein Funktionsaufruf (Umwandlung in Funktionszeiger $\rightarrow$ *Prog.II*)

Parameter

- Wertparameter (`T x`) [`T` Datentyp, `x` Parametername]
  - Beim Funktionsaufruf Kopie der Argumentwerte auf gleichnamige lokale Variablen
  - Vereinbarung dieser Variablennamen in Parameterliste, *nicht* im Funktionsblock
  - Immer Wertparameterübergabe, ausgenommen C-Vektoren und Funktionen, Referenzparameter und konstante Referenzparameter
- Referenzparameter (`T& x`)
  - Beim Funktionsaufruf Benutzung des Speicherplatzes der eingesetzten Variable (keine Kopie des Argumentwerts!)
  - Änderung der eingesetzten Variable zulässig (Hauptzweck!)
  - Konstanten und Ausdrücke (außer in Sonderfällen) als Argumente *unzulässig*
- Konstante Referenzparameter (`const T& x`)
  - Beim Funktionsaufruf Benutzung des Speicherplatzes der eingesetzten Variable, keine Kopie (Hauptzweck!)
  - Änderung der eingesetzten Variable *unzulässig*
  - Konstanten und Ausdrücke zulässig (Compiler legt Hilfsvariable an)

Wertparameterübergabe (call by value)

- Eingesetzte Argumente werden bei Wertparameterübergabe kopiert.
- Eine Änderung dieser Werte in der Funktion bewirkt *keine* Änderung evtl. eingesetzter Variablen in der *aufrufenden* Funktion.

```
void vertausche(int iv, int jv)
// iv=im; jv=jm; (Zuweisung)
{ int hv;
  hv=iv; iv=jv; jv=hv;
  return;
}

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

Referenzparameterübergabe (call by reference)

- Direkter Lese- und Schreibzugriff auf eingesetzte Variable *statt* Kopie des Variablenwerts

```
void vertausche(int& iv, int& jv)
// iv ≡ im; jv ≡ jm; (gleiche Speicherplätze)
{ int hv;
  hv=iv;   // d.h. hv=im
  iv=jv;   // d.h. im=jm
  jv=hv;   // d.h. jm=hv
  return;
}

int main()
{ int im=2; jm=4;
  vertausche(im,jm); // im=4, jm=2
  return 0;
}
```

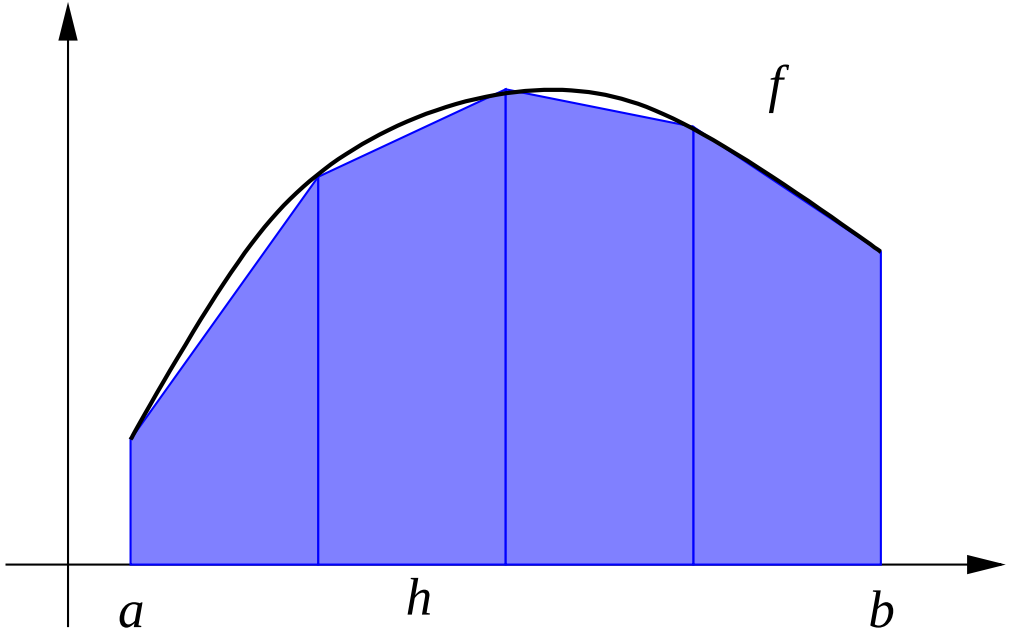
Anmerkung: Funktion `swap` in `<algorithm>` vorhanden!

Funktionen als Parameter: Sehnentrapezregel

→ *Infbl.10, S.4*

Näherung für $\int_a^b f(x)dx$: $\quad h\sum_{i=0}^{n} w_i f(x_i)$

mit $h = \frac{b-a}{n}$, $x_i = a + ih$ $(i{=}0,\ldots,n)$, $w_i = \begin{cases} \frac{1}{2} & i{=}0,n \\ 1 & i{=}1,\ldots,n{-}1 \end{cases}$



- Param.liste von sehnentrapez: Param. *x* von *g* überflüssig
  d.h. `double g(double)` statt `double g(double x)`

Funktionsaufruf - modellhafte Implementierung

Aufruf:

- Speicherung auf Stack (in neuem Stacksegment):
  - übergebene Argumente
  - lokale Variablen der Funktion
  - Rücksprungadresse
- Sprung zum Codeanfang der aufgerufenen Funktion (Startadresse!)

Beendigung:

- Übergabe des Funktionswerts
- Rücksprung
- Freigabe des Stacksegments

Beispiel für Funktionsaufrufe (→ nächste Seite):

`g(...)`	
`f(...)`	ruft g auf
`main(...)`	ruft f auf

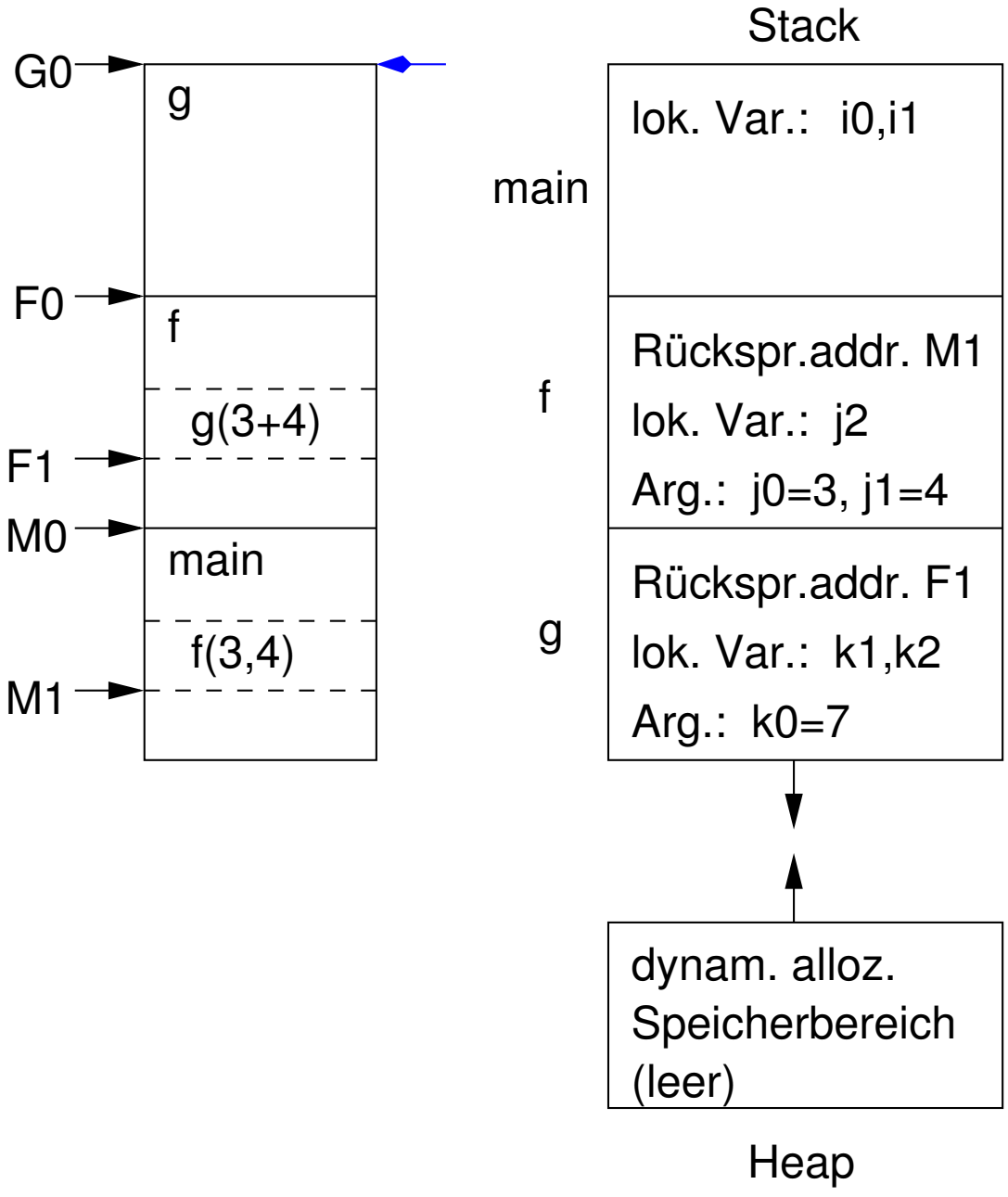
Beispiel für Funktionsaufrufe

```
void g(int k0)
{
  int k1,k2;
  return;
}

void f(int j0,int j1)
{
  int j2;
  g(j0+j1);
  return;
}

int main()
{
  int i0=3,i1=4;
  f(i0,i1);
  return 0;
}
```

Speicherlayout - Modell

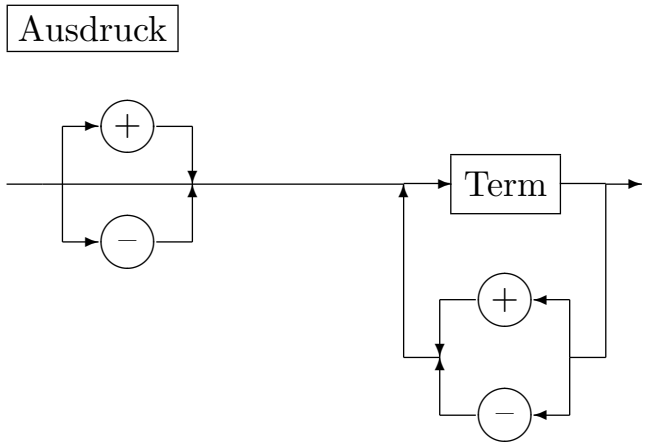


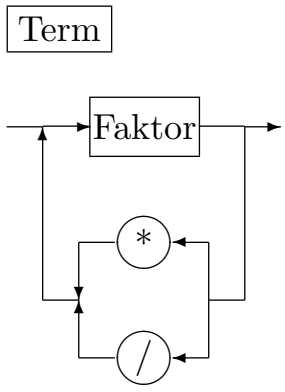
Rekursive Funktionen

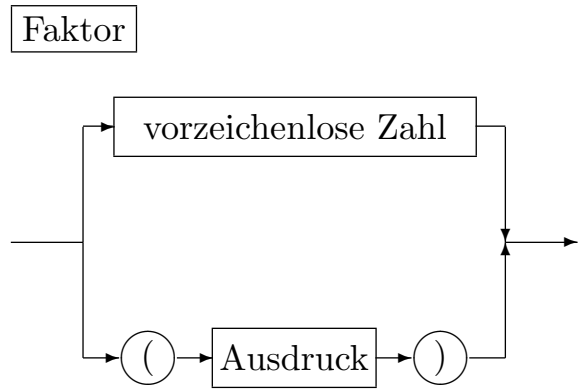
- Rekursive Funktionen rufen sich selbst direkt oder indirekt auf.
- Der Stack ermöglicht die Implementierung rekursiver Funktionen.
- Für wechselseitig rekursive Funktionen sind „Vorwärts-deklarationen“ erforderlich, damit Funktionen aufgerufen werden können, die erst später (d.h. weiter unten im Programm) definiert werden.
- Diese Funktionsdeklaration entsteht aus der Funktions-definition durch Ersetzen des Funktionsblocks durch einen Strichpunkt. (Weglassen der Parameternamen möglich.)
- In der Regel empfiehlt es sich wegen des erhöhten Speicher- und Rechenaufwands rekursive Funktionen zu vermeiden und durch iterativ arbeitende Funktionen zu ersetzen.

Auswertung arithmetischer Ausdrücke I

Syntaxdiagramme







Auswertung arithmetischer Ausdrücke II

Die Syntaxdiagramme beschreiben folgende Situation:

$$\begin{aligned} \text{Ausdruck} &= [\pm]\,\text{Term}\,[\pm\,\text{Term}\pm\ \ldots\ \pm\,\text{Term}] \\ \text{Term} &= \text{Faktor}\,[{}^{*}\!/\ \text{Faktor}\ {}^{*}\!/\ \ldots\ {}^{*}\!/\ \text{Faktor}] \\ \text{Faktor} &= \text{vorzeichenloseZahl} \mid (\text{Ausdruck}) \end{aligned}$$

Bem.: `+-4` ist hier im Unterschied zu C++ nicht erlaubt.

Bsp.: `-(7-2*(3+5))` ist syntaktisch zulässig:

`7` vorzeichenlose Zahl $\Rightarrow$ `7` Faktor $\Rightarrow$ `7` Term

`3` vorzeichenlose Zahl $\Rightarrow$ `3` Faktor $\Rightarrow$ `3` Term
`5` vorzeichenlose Zahl $\Rightarrow$ `5` Faktor $\Rightarrow$ `5` Term
$\Rightarrow$ `3+5` Ausdruck $\Rightarrow$ `(3+5)` Faktor
`2` vorzeichenlose Zahl $\Rightarrow$ `2` Faktor
$\Rightarrow$ `2*(3+5)` Term

$\Rightarrow$ `7-2*(3+5)` Ausdruck $\Rightarrow$ `(7-2*(3+5))` Faktor
$\Rightarrow$ `(7-2*(3+5))` Term $\Rightarrow$ `-(7-2*(3+5))` Ausdruck

Auswertung arithmetischer Ausdrücke III

- Der Weg durch die Syntaxdiagramme bei der Verarbeitung der Eingabe ist jeweils durch Lesen des nächsten Zeichens eindeutig bestimmt.
- Bei jedem Durchlauf durch die 3 Syntaxdiagramme wird mindestens ein Zeichen der Eingabe gelesen.

Idee:

- Jede der Funktionen `WertAusdruck`, `WertTerm` und `WertFaktor` liest ab der aktuellen Eingabeposition (entsprechend ihrem Syntaxdiagramm) und wertet die gelesenen Daten aus. Das nächste Zeichen ist dann das erste der noch nicht ausgewerteten Eingabe.
- Verzweigungen in den Syntaxdiagrammen werden mit `cin.peek` („Vorausschauen“) gefunden.
- Bei Fehlern wird die Funktion `fehler` aufgerufen, die das Programm mit einer Fehlermeldung beendet.

Auswertung arithmetischer Ausdrücke IV

Anmerkung: Das „Vorausschauen“ mit `peek()` ist möglich, weil die Benutzereingabe vom Terminaltreiber zeilenweise gelesen wird (Zeilenpufferung) und das Programm aus diesem Puffer liest. Das letzte Zeichen im Puffer ist `\n`.

Verbesserung: Zur Lokalisierung von Syntaxfehlern ist es sinnvoll, die Eingabezeile auf einen String zu schreiben und diesen über einen Inputstringstream zu lesen. Mittels `tellg` kann dann die Fehlerposition bestimmt werden. ($\rightarrow$ *Übungen*)