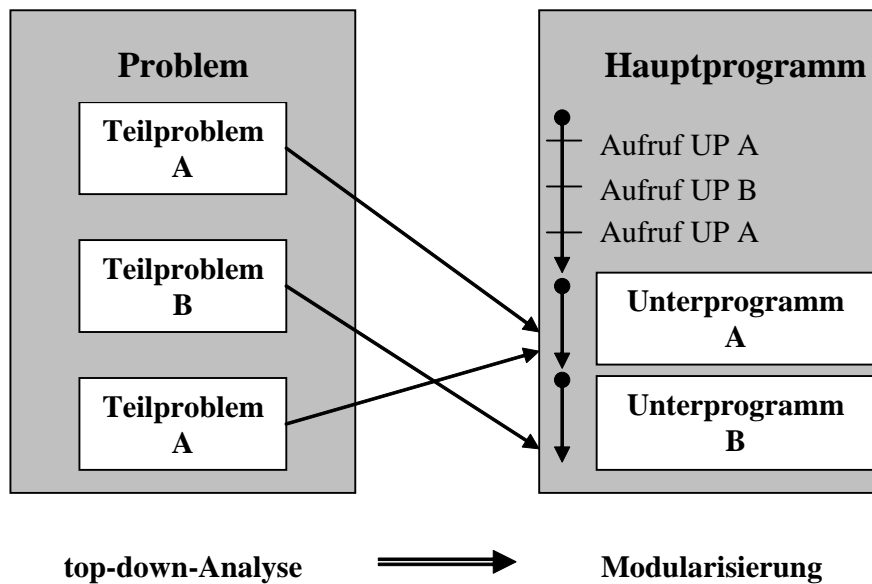


Inhalt

6	Funktionen.....	6-2
6.1	Unterprogrammtechnik	6-2
6.2	C-Funktionen.....	6-3
6.2.1	Reihenentwicklungen	6-3
6.2.2	Funktionsvereinbarung.....	6-4
6.2.3	Funktionsrückprung.....	6-6
6.2.4	Funktionsaufruf	6-7
6.3	Die Hauptfunktion int main().....	6-8
6.4	Das vollständige Programm <code>exptab.c</code>	6-8
6.5	Bibliotheken	6-11

6 Funktionen

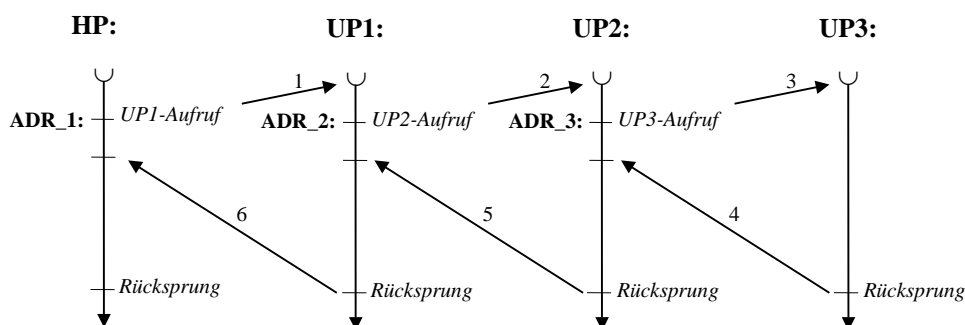
6.1 Unterprogrammtechnik



Mittels **top-down-Analyse** zerlegt man ein komplexes Problem solange in kleinere Teilprobleme bis diese überschaubar werden. Diese Teilprobleme kann man zunächst als **Unterprogramme** realisieren und auch einzeln testen, ehe man sie zu einem Ganzen zusammenfasst, um schließlich das komplexe Problem zu lösen.

- => **Unterprogramme dienen der Strukturierung eines Programms,**
1. **der Ausgliederung wiederkehrender Berechnungen und**
 2. **der Zerlegung komplexer Probleme in kleinere.**

Verarbeitung von Unterprogrammen im Compiler



Kellerung

Tiefe														
1		leer	→ ₁	ADR ₁ +1	→ ₂	ADR ₂ +1	→ ₃	ADR ₃ +1	→ ₄	ADR ₂ +1	→ ₅	ADR ₁ +1	→ ₆	leer
2				ADR ₁ +1		ADR ₂ +1		ADR ₁ +1						
3						ADR ₁ +1								

- Unterprogramme:** Unterprogrammaufruf
 Unterprogrammvereinbarung (Deklaration und Definition)
 Unterprogrammrückprung

6.2 C-Funktionen

In C werden Unterprogramme grundsätzlich als **Funktionen** realisiert, diese liefern **höchstens** einen Funktionswert. Da das Hauptprogramm *int main()* auch eine Funktion darstellt, ist ein C-Programm im Wesentlichen eine **Folge von C-Funktionen**.

modul.c

Direktiven für den Präprozessor

globale Deklarationen

Typ funktion_1(...)

Typ funktion_2(...)

...

Typ funktion_n(...)

int main(...)

{

lokale Deklarationen

Anweisungsfolge

}

Typ funktion_1(...)

{

lokale Deklarationen

Anweisungsfolge

}

Typ funktion_2(...)

{

lokale Deklarationen

Anweisungsfolge

}

...

Typ funktion_n(...)

{

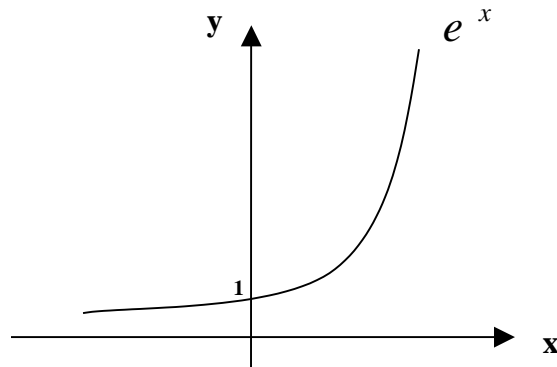
lokale Deklarationen

Anweisungsfolge

}

6.2.1 Reihenentwicklungen

Wir wollen die Unterprogrammtechnik anhand einer Reihenentwicklung für die Exponentialfunktion $f(x) = e^x$ mit $x \in \mathbb{Q}$ einführen und gleichzeitig Fortpflanzung von Fehlern durch die Zahlendarstellung im Rechner untersuchen.



Reihenentwicklung: $f(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$

Algorithmus für die Reihenentwicklung

Berechne den neuen Summanden aus dem alten und addiere ihn zur bis dahin berechneten Summe. Wiederhole das solange, bis der neue Summand so klein ist, dass er keinen Beitrag zur Summe bildet, d.h. die Summe sich nicht mehr verändert (Rechnergenauigkeit).

- (1) `summeNeu = 1, summand = 1, i = 1` *Startwerte*
- (2) **berechne**
 - (a) `summand = summand * x / i` *Berechnen des neuen Summanden*
 - (b) `summeAlt = summeNeu` *Berechnen der neuen Summe*
 - (c) `summeNeu = summeNeu + summand`
 - (d) `i = i + 1`
- solange** `summeNeu != summeAlt` *Abbruch*

Programm für die Reihenentwicklung

```
(1) summeNeu = 1; summand = 1; i = 1;
(2) do
    {
        summand = xx / i++; /* (a), (d) */
        summeAlt = summeNeu; /* (b) */
        summeNeu += summand; /* (c) */
    } while( summeNeu != summeAlt);
```

6.2.2 Funktionsvereinbarung

Zunächst wird eine Funktion mit dem Namen `myexp` definiert, welche einen Parameter `double xx` besitzt und einen Funktionswert vom Typ `double` zurückliefert.

Jede Funktion, außer der Hauptfunktion `int main()`, wird üblicherweise in zwei Schritten vereinbart. Diese Vorgehensweise entspricht der **top-down-Analyse**: Im ersten Schritt wird dem Compiler mitgeteilt, welche Funktionen benötigt werden, im zweiten Schritt wird festgelegt, welche Operationen die Funktionen ausführen.

1. Schritt: Deklaration einer Funktion (Festlegen des Prototyps)

Dem Compiler muss vor dem ersten Aufruf einer Funktion der Funktionsname (symbolische Adresse der Funktionsdefinition), der Typ des Funktionswerts, die Parametertypen und ihre Anzahl bekannt sein.

Syntax:

Funktionstyp Funktionsname ([Parametertyp [Parametername] , ...]) ;

Funktionstyp:

- Gibt den Typ des Funktionswertes an, kein Funktionswert: **void**.

Parametertyp:

- Treten Parameter in der Funktion auf, so müssen ihre Typen aufgelistet werden.
- C-Funktionen benötigen stets eine Parameterliste, auch wenn sie leer ist: *f()*.

Parametername:

- *C-Name*, Parameternamen können bei der Deklaration angegeben werden.

Die Deklaration einer Funktion erfolgt in der Regel im **globalen Deklarationsteil** des Programms oder in einer **Headerdatei**. Eine Headerdatei besitzt als Suffix **.h** und fasst mehrere Deklarationen zusammen.

Deklaration der Funktion *myexp*

```
double myexp( double);          /* globale Deklaration */
```

Deklaration der Sandardfunktionen und der Funktionen aus der Mathematischen Bibliothek

```
# include <stdio.h>              /* Standardfunktionen */
# include <math.h>              /* mathematische Funktionen */
```

2. Schritt: Definition einer Funktion

Die Folge der Anweisungen, die beim Funktionsaufruf ausgeführt werden soll, wird **nach** der Definition der Funktion ***int main()*** oder auf einem anderen Modul zusammengefasst.

Syntax:

*Funktionstyp Funktionsname ([Parametertyp Parametername , ...])
zusammengesetzte Anweisung*

Parametername:

- *C-Name*, Parameternamen müssen bei der Definition angegeben werden:
Die formalen Parameter sind für die Definition der Funktion als Platzhalter der Argumentwerte notwendig und müssen deshalb dem Compiler mitgeteilt werden.

Semantik:

1. Die Parameterübermittlung erfolgt durch eine Kopie der Werte der Argumente - **call by value**. Falls notwendig werden die Argumente in den Parametertyp konvertierten.
2. Die aufgerufene Funktion wird mit diesen Werten ausgeführt.

Definition der Funktion *myexp*

```

/*
 * Exponentialfunktion (Reihenentwicklung)
 * e hoch x = 1 + x + x hoch 2 / 2! + x hoch 3 / 3! + ...
 */

double myexp( double xx)
{
    double summeAlt, summeNeu = 1, summand = 1;
    int i = 1;                /* Zaehler fuer Durchlaeufe */

    do                        /* Reihenentwicklung */
    {                          /*Berechnung des neuen Summanden */
        summand *= xx / i++;
                                /*Berechnung der neuen Summe */
        summeAlt = summeNeu;
        summeNeu += summand;
    } while(( summeNeu != summeAlt) /*&& (i < 100)*/);

    return summeAlt;
}

```

6.2.3 Funktionsrücksprung

Eine Funktion wird entweder nach der Abarbeitung der letzten Anweisung mit einem zufälligen Funktionswert oder durch die **return** - Anweisung verlassen.

Syntax: **return [Ausdruck] ;**

Semantik:

- Der Wert des Ausdrucks wird in den Funktionstyp konvertiert und als Funktionswert zurückgegeben, d.h. im Programm an die Stelle des Funktionsaufrufs gesetzt.
- Das Programm wird an der Aufrufstelle fortgesetzt.
- Der Rückgabewert wird i. R. von der aufrufenden Funktion ausgewertet.

Die Funktion **double myexp(double);** liefert einen Wert für die Exponentialfunktion zurück. Dieser wurde in der Variablen **summeAlt** berechnet.

```

double myexp( double xx)
{
    . . .
    return summeAlt;                /* double */
}

```

6.2.4 Funktionsaufruf

Syntax: *Funktionsname* ([*Argument* , ...])

Der Funktionsaufruf ist ein Ausdruck, kann also in komplexen Ausdrücken als Operand verwendet werden.

Funktionsname:

- *C-Name*, symbolischer Bezeichner für die Speicheradresse der Funktionsdefinition.

Argument:

- C benötigt beim Funktionsaufruf eine *Argumentenliste*, auch wenn sie leer ist: *f()*.
- Gibt man keine Argumentenliste an, so erfolgt kein Funktionsaufruf, sondern es wird die Adresse der Funktionsdefinition ermittelt: *f*.
- Anzahl und Typen der Argumente sind durch die Funktionsvereinbarung festgelegt.
- Für *Argument* kann ein beliebiger mit dem vorgegebenen Typ verträglicher (in ihn konvertierbarer) Ausdruck eingesetzt werden
- Geschachtelte Funktionsaufrufe sind erlaubt.

Semantik:

1. Die Werte der Argumente werden berechnet, in den vorgegebenen Parametertyp konvertiert und eine Kopie des Wertes der Funktion übergeben (**call by value**).
2. Die aufgerufene Funktion wird mit diesen Werten ausgeführt und danach wird die Aufrufstelle durch den Funktionswert, falls vorhanden, ersetzt und die Ausführung des Programms fortgesetzt.

In einem Testprogramm *exptab.c* soll mit der Funktion **myexp** und der Funktion **exp** aus der mathematischen Bibliothek experimentiert werden. Wir finden dort sowohl Funktionsaufrufe vorgefertigter Funktionen als auch den unserer Funktion.

exptab.c

```

/*
 * Exponentialfunktion e hoch x
 * Aufruf: gcc -Wall -ansi -O exptab.c -o exptab -lm
 */

# include <stdio.h>
# include <math.h>          /* mathematische Funktionen */

double myexp( double);      /* Exponentialfunktion */

/* Rahmenprogramm zur Exponentialfunktion */
int main()
{
    double x;                /* Argument */

    /* Vorbereitung Ueberschrift */
    printf( "\nExponentialfunktion - e hoch x  -\n");

    /* Eingabe */

```

```

    printf( "\nx = ");
    scanf( "%lf", &x);

    /* Verarbeitung und Ausgabe*/
    /* Vergleich mit Standardfunktion aus <math.h> */
    printf("e ^ %g = %g, %g\n", x, myexp( x), exp( x));

    return 0;
}

```

6.3 Die Hauptfunktion *int main()*

- Es muss genau eine Funktion mit dem Namen *main* geben, diese ist der Einstiegspunkt in das Programm und darf selbst **nicht** aufgerufen werden.
- Für die Funktion *int main()* fällt die Funktionsdeklaration mit der Funktionsdefinition zusammen.
- Sie ist vom Typ *int* und muss einen Rücksprung *return intAusdruck ;* aufweisen.
- Der Rückgabewert kann vom übergeordneten Programm ausgewertet werden. Vereinbarungsgemäß wird der Rückgabewert **0** als korrekter Programmlauf und jeder anderer Wert als Fehlercode betrachtet.

```

/*
 * Hauptprogramm
 */

int main()
{
    . . .
    return 0;
}
/* int */

```

6.4 Das vollständige Programm *exptab.c*

Das Programm wird erweitert:

Durch eine Schleife wird die Möglichkeit der Berechnung mehrerer Werte gesteuert.

Hier nun das vollständige Programm:

exptab.c

```

/*
 * Exponentialfunktion e hoch x
 * Aufruf gcc -Wall -O -ansi exptab.c -o exptab -lm
 * tabelliert Zwischenergebnisse
 */

# include <stdio.h>
# include <math.h>
/* mathematische Funktionen */

```



```

double myexp( double);           /* Exponentialfunktion */

/*
 * Rahmenprogramm zur Exponentialfunktion
 */

int main()
{
    double x;                       /* Argument */
    char c[ 2];                     /* Abbruch j/n */

    /* Vorbereitung Ueberschrift */
    printf( "\nExponentialfunktion - e hoch x  -\n");

    do
    {
        /* Eingabe */
        printf( "\nx = ");
        scanf( "%lf", &x);

        /* Verarbeitung und Ausgabe*/
        /* Vergleich mit Standardfunktion aus <math.h> */
        printf("e ^ %g = %g, %g\n", x, myexp( x), exp( x));

        /* Nachbereitung */
        printf( "Noch einmal j/n? ");
        scanf( "%1s", c);
        } while( c[ 0] == 'j');

    return 0;
}

/*
 * Exponentialfunktion (Reihenentwicklung)
 * e hoch x = 1 + x + x hoch 2 / 2! + x hoch 3 / 3! ...
 */

double myexp( double xx)
{
    double summeAlt, summeNeu = 1, summand = 1;
    int i = 1, j = 0;

    /* Testausdruck-Ueberschrift */
    printf( "\n%3s%25s%25s%25s\n",
           "i", "summand", "summeAlt", "summeNeu");

    do /* Reihenentwicklung */
    { summand *= xx / i++;

        summeAlt = summeNeu;

```

```
    summeNeu += summand;
                                                    /* Testausdruck */
    printf( "%3d%25.10g%25.10g%25.10g\n",
           i, summand, summeAlt, summeNeu);

    j++;
                                                    /* Blaettern */
    if( j == 20){ j = 0; fflush( stdin); getchar();}

} while(( summeNeu != summeAlt) /*&& (i < 100)*/);

return summeAlt;
}
```

Test - Ergebnisse

```
Exponentialfunktion - e hoch x -
x = 1
e ^1 = 2.718282, 2.718282
x = -3
e ^ -3 = 0.0497871, 0.0497871
x = -10
e ^-10 = 0.000045, 0.000045
x = -13.5
e ^ -13.5 = 1.37097e-06, 1.37096e-06
x = -19.5
e ^ -19.5 = -2.45344e-10, 3.39827e-09
```

Durch höhere Genauigkeit lassen sich die **Fortpflanzungsfehler** nicht vermeiden.

Einschränkung des Fehlers

Diese Fehler treten für $x < 0$ auf. Man berechnet deshalb $e^{|x|}$. Für $x < 0$ ist dann $e^x = \frac{1}{e^{|x|}}$.

6.5 Bibliotheken

Header - Deklarationen von Bibliotheksfunktionen

Zahlreiche Funktionen stehen in Bibliotheken zur Verfügung. Ihre Deklarationen sind in Headerdateien zusammengefasst und stehen damit für den globalen Deklarationsteil zur Verfügung, **# include < * .h >**. Die Headerdateien haben einen Namen mit dem Suffix **.h**, sind ASCII-Dateien und damit lesbar und befinden sich standardmäßig im Verzeichnis **/usr/include**.

Die Deklarationen aus einer Headerdatei werden im Präprozessorlauf anstelle

include < * .h > (festgelegter Suchpfad für Headerdateien **/usr/include**)

include " * .h " (Suchpfad: aktuelles Verzeichnis)

als globale Deklarationen in das Programm gesetzt.

Übersicht über die wichtigsten Header der Standardbibliothek

<assert.h>	Testhilfen	
<ctype.h>	Zeichenverarbeitung	isdigit,...
<errno.h>	Fehlernummern	
<float.h>	Interne Datenformate (Gleitkommatypen)	FLT_MAX,...
<limits.h>	Interne Datenformate (Ganzzahlige Typen)	INT_MAX,...
<local.h>	Länderspezifische Darstellung von Zahlen	
<math.h>	Mathematische Funktionen	sin, cos,...
<setjmp.h>	Sprünge zwischen Funktionen	
<signal.h>	Behandlung von Signalen	
<stdarg.h>	Behandlung von Funktionen mit variabler Parameterzahl	
<stddef.h>	Elementare Typen	size_t, NULL,...
<stdio.h>	Ein-/ Ausgabe	printf, scanf,...
<stdlib.h>	Diverse Hilfsroutinen	malloc,...
<string.h>	Stringverarbeitung	strcpy, strcat,...
<time.h>	Termine und Zeiten	time

Bibliotheken - Definitionen von Bibliotheksfunktionen

Die Definition der Funktionen stehen bereits übersetzt in Bibliotheken bereit. Dabei kann es zu einer Bibliothek mehrere Header geben. Die Bibliotheken selbst stehen in festgelegten Verzeichnissen **/lib** und **/usr/lib**. Ihr Name hat den Aufbau **liblibrary.a** für statische und **liblibrary.so*** für dynamische Bibliotheken.

Alle Standard-C-Funktionen sind in der Bibliothek **libc.a** bzw. **libc.so** enthalten. Zu ihr gehören zum Beispiel die Header **stdio.h**, **stdlib.h**, **string.h** und weitere.

Möchte man die Funktionen einer Bibliothek auflisten, so wechsle man in das Verzeichnis der Bibliothek und führe den folgenden Befehl aus:

```
$ cd /lib
$ nm Bibliotheksname
```

Alle mathematischen Bibliotheksfunktionen erhält man mittels:

```
$ nm libm.so | pg
```

Erläuterungen zum Gebrauch von C-Funktionen findet man hingegen im **Manual**:

```
$ man C-Funktion
```

Binder

Der Binder hat die Aufgabe, die benötigten Funktionen aus den Bibliotheken zu dem Objektcode des Programms hinzuzuladen und daraus ein ausführbares Programm zu erzeugen (s. Kapitel 1). Für die Funktionen aus der Bibliothek **libc.a** macht er das automatisch.

Für andere Funktionen muss man das dem Binder durch die Option **-llibery** mitteilen. Zum Beispiel sind Programme mit Funktionen der mathematischen Bibliothek mit dem Binderhinweis **-lm** zu übersetzen.

Beispiel aus *exptab.c*

```
/*
 * Exponentialfunktion e hoch x
 * Aufruf: gcc -Wall -ansi -O exptab.c -o exptab -lm
 */

# include <stdio.h>
# include <math.h>          /* mathematische Funktionen */

. . .

int main()
{
. . .

/* Vergleich mit Standardfunktion aus <math.h> */
    printf("e ^ %g = %g, %g\n", x, myexp( x), exp( x));

. . .

    return 0;
}
```

Es ist möglich, sich eigene Bibliotheken zu erzeugen.

