Programmieren C

Andreas Krennmair
a.krennmair@aon.at

14. September 2001

Inhaltsverzeichnis

1 Einleitung **4**
- 1.1 Geschichte . . . 4
- 1.2 Grundinhalte . . . 4
- 1.3 Vorteile . . . 4
- 1.4 Nachteile . . . 4
- 1.5 Aufbau eines C-Programms . . . 5
  - 1.5.1 Precompileranweisungen . . . 5
  - 1.5.2 Deklarationen . . . 5
  - 1.5.3 Definitionen . . . 5

2 Bestandteile eines C-Programmes **6**
- 2.1 Bezeichner . . . 6
- 2.2 Datentypen . . . 6
- 2.3 Konstante . . . 7

3 Operatoren **7**
- 3.1 Arithemetische Operatoren . . . 7
- 3.2 Vergleichsoperatoren . . . 7
- 3.3 Logische Operatoren . . . 8
- 3.4 Bitoperatoren . . . 8
- 3.5 Adreßoperatoren . . . 8
- 3.6 sizeof-Operator . . . 8
- 3.7 Sequenzoperator . . . 8
- 3.8 Punkt- und Pfeiloperator . . . 8
- 3.9 Klammeroperatoren . . . 9
- 3.10 Alternativoperator . . . 9
- 3.11 Typecast-Operator . . . 9
- 3.12 Weitere Operatoren . . . 10
- 3.13 Prioritätsreihenfolge der Operatoren . . . 11

4 Ausdrücke und Anweisungen **12**
- 4.1 Selektionsanweisungen . . . 12
  - 4.1.1 If-Anweisung . . . 12
  - 4.1.2 Switch-Anweisung . . . 12
- 4.2 Schleifen . . . 13
  - 4.2.1 while-Schleife . . . 13
  - 4.2.2 do-while-Schleife . . . 13
  - 4.2.3 for-Schleife . . . 13
  - 4.2.4 Kontrolltransferanweisungen . . . 14

5 Arrays **14**
- 5.1 Eindimensionale Arrays . . . 14
- 5.2 Initialisierung . . . 14
- 5.3 Mehrdimensionale Arrays . . . 15
- 5.4 Zeichenketten . . . 15

6 Funktionen **16**
- 6.1 C-Speicherklassen . . . 16

7 Strukturen **17**
7.1 Definition und Verwendung 17
7.2 Zulässige Operationen auf Strukturen 18
7.3 Initialisierung 19
7.4 Kompliziertere Strukturdefinitionen 19
7.4.1 Arrays von Strukturen 19
7.4.2 Strukturen in Strukturen 19
7.5 Selbstdefinierte Datentypen 19

8 Zeiger **20**
8.1 Dynamische Speicherverwaltung 20
8.2 Zeigerbegriff 20
8.3 Zeiger und Arrays 21
8.3.1 Zeigerarithmetik 22
8.4 Zeiger auf Funktionen 22

9 Bilschirm I/O **23**
9.1 Zeichenorientierte Ein-/Ausgabe 23
9.1.1 putchar 23
9.1.2 getchar 24
9.2 Formatierte Ein- und Ausgabe 24
9.2.1 printf 24
9.2.2 scanf 24

10 Datei-I/O **25**
10.1 Standard-Datei-I/O 25
10.1.1 Öffnen einer Datei 25
10.1.2 putc 26
10.1.3 getc 26
10.1.4 Schließen einer Datei 27
10.1.5 fprintf und fscanf 27
10.1.6 Die Standarddateien 27
10.2 Zusätzliche Funktionen zum Datei-I/O 27
10.2.1 fflush 27
10.2.2 rewind 27
10.2.3 fseek 27
10.2.4 ftell 28
10.3 Typisierte Dateien 28
10.3.1 Schreiben 28
10.3.2 Lesen 29

A Sortieralgorithmen **29**
A.1 Konventionen 29
A.2 Elementare Sortierverfahren 30
A.2.1 Selection Sort 30
A.2.2 Insertion Sort 31
A.2.3 Bubble Sort 32
A.3 Kompliziertere Sortierverfahren 32
A.3.1 Shell Sort 32
A.3.2 Quick Sort 33

1 Einleitung

1.1 Geschichte

- BCPL ⇒ B (Ken Thompson) ⇒ C (Dennis Ritchie): zu Beginn als systemnahe Sprache (zur Programmierung des UNIX- Betriebssystems), später auch Anwendungsprogrammierung.
- Normung durch ANSI (ANSI-C)

1.2 Grundinhalte

- Typenkonzept
- Blockstruktur
- rekursive Funktionen
- Systemaufrufe
- maschinenunabhängiges Programmieren
- hardwarenahe Operationen
- separates Kompilieren
- Standard-Include-Dateien

nicht unmittelbar enthalten:

- Array-Operatoren
- Ein-/Ausgabeoperationen (stdio.h)
- mathematische Funktionen (math.h)
- Heap-Verwaltung (malloc.h)

1.3 Vorteile

- Unterstützung der strukturierten Programmierung
- Unterstützung der modularen Programmierung
- Unterstützung der portablen Programmierung
- hohe Ausführungsgeschwindigkeit
- große Akzeptanz in der Softwareindustrie
- Weiterentwicklung ⇒ C++

1.4 Nachteile

- Disziplin beim Programmieren notwendig (=,==)
- keine vollkommene Schnittstellenprüfung

1.5 Aufbau eines C-Programms

3 Grundkomponenten:

- Precompileranweisungen
- Deklarationen von Variablen und Funktionen
- Definition von Variablen und Funktionen

1.5.1 Precompileranweisungen

Werden vor der eigentlichen Compilierung durch einen Preprozessor bearbeitet.

```
#include <stdio.h>   /* über INCLUDE-Pfad */
#include "mydat.h" /* aktuelles Verzeichnis */
#define MAX 100
```

1.5.2 Deklarationen

Dienen zur Beschreibung von Variablen und Funktionen, die an einer anderen Stelle im Programm definiert sind.

1.5.3 Definitionen

Beschreiben Variablen, Funktionen und Datentypen und beinhalten auch ihre Realisierung (Variable: Speicherplatz anlegen, Funktion: Source Code).

Komplettes C-Programm:

- mindestens 1 Funktion (main-Funktion)
- eventuell weitere Funktionen

Source-File muss alle Informationen enthalten, damit es kompiliert werden kann.

```
#include <stdio.h>
#define Z1 9
#define Z2 5

int max(int, int );

int main(void)
{
  int c;
  c = max(Z1, Z2);
  printf("Das Maximum der beiden Zahlen ist: %d",c);
  return 0;
}

int max(int a, int b)
{
  if (a>b)
    return a;
```

```
  else
    return b;
}
```

2 Bestandteile eines C-Programmes

2.1 Bezeichner

Identifizieren Variablen, Funktionen, Datentypen und Labels.

- muss mit Buchstaben oder Underscore (_) beginnen.
- kann mit Buchstaben, Ziffern, Underscore fortgesetzt werden.
- Länge ist compilerabhängig und betriebssystemabhängig.
- C ist *case-sensitive*.

2.2 Datentypen

Es gibt in C 5 Basisdatentypen:

- void
- char
- int
- float
- double

Diese können zum Teil durch Typmodifikatoren bezüglich ihres Wertebereiches und ihres Speicherbedarfes verändert werden:

- signed
- unsigned
- short
- long

char	8 bit	
short int	16 bit	-32768 - 32767
int	16 bit	
	32 bit	-2147483648 - 2147483647
long int	32 bit	
unsigned long	32 bit	0 - 4294967295
float	32 bit	$\pm\ 10^{\pm\ 38}$ mit 7 Nachkommastellen
double	64 bit	$\pm\ 10^{\pm\ 308}$ mit 16 Nachkommastellen

Verknüpfung von Datentypen:

- Ergebnisdatentyp ist immer der größere Datentyp.
- Zuweisung: bei verträglichen Datentypen erfolgt eine automatische Typkonvertierung, wobei es zu einem Informationsverlust kommen kann.

2.3 Konstante

Erhalten immer den nächstgrößeren passenden Datentyp.

Character-Konstante

- müssen unter einfachen Hochkommas stehen.
- Darstellung besonderer ASCII-Codes durch Backslash-Codes.
- Beispiele: `'a','\x','\n','\0'`

Stringkonstante

- Stehen immer unter doppelten Hochkomma.

```
"Das ist ein String"
"Müller\'s Büro"
"Ausgabe: \n"
```

3 Operatoren

3.1 Arithemetische Operatoren

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`%`	Modulo-Division
`++`	Inkrement
`--`	Dekrement

Beispiele:

```
int a=5, b=2, c;
float f;

c = a / b;  // c = 2
f = a / b; // f = 2.0
f = (float) a / b; // f = 2.5
c = a++; // c = 5, a = 6
c = ++a; // a = 7, c = 7
```

3.2 Vergleichsoperatoren

`<,<=,>,>=,==,!=`

Rückgabewerte Bei Gleichheit ungleich 0, sonst 0.

3.3 Logische Operatoren

Logisches UND	&&
Logisches ODER	||
Logisches NICHT	!

3.4 Bitoperatoren

Bitweises UND	&
Bitweises ODER	|
Bitweises exklusives ODER	^
Bitweise Negation	~
Verschiebung nach Links	<<
Verschiebung nach Rechts	>>

Beispiel:

```
char x = 7;     // 00000111b = 7
x = x << 1; // 00001110b = 14
x = x << 3; // 01110000b = 112
x = x << 2; // 11000000b = -64
x = x >> 2; // 11110000b = -16
```

Bei einer Linksverschiebung werden rechts Nullen eingefügt, bei einer Rechtsverschiebung werden links bei einem vorzeichenbehafteten Datentyp Einsen, bei einem vorzeichenlosen Datentypen Nullen eingefügt.

3.5 Adreßoperatoren

&	liefert Adresse eines Objektes
*	liefert den Inhalt einer bestimmten Adresse

Beispiele:

```
int x, y, *z; // PASCAL: x,y : integer; z : ^integer;
z = &x; // PASCAL: z:=@x;
y = *z; // PASCAL: y:=z^;
```

3.6 sizeof-Operator

Bestimmt die Größe eines Datenobjekts od. Datentyps.

3.7 Sequenzoperator

Beistrich (,): Folge von Anweisungen. Sinnvoll z.B. bei Initialisierung der for-Schleife.

3.8 Punkt- und Pfeiloperator

Punktoperator (.): Dient zum Ansprechen der Elemente innerhalb einer Struktur.

Beispiel:

```
struct datum {
  int tag;
  int monat;
  int jahr;
} gebtag;

gebtag.tag = 29;
/* ... */
```

Pfeiloperator (->): Hat den selben Zweck wie der Punkt-Operator, wird aber dann verwendet, wenn man von der Struktur die Adresse kennt.

Beispiel:

```
struct datum zeugnisdat, *zdat;
/* ... */
zeugnisdat.tag = 7;
zdat = &zeugnisdat;
zdat->tag = 8;
```

3.9 Klammeroperatoren

Runde Klammern ((,)):

- Priorität in Ausdrücken ändern
- Parameterliste eingrenzen (für Funktionen)

Eckige Klammern: ([,]):

- Definition der Arraygröße
- Indizierung

3.10 Alternativoperator

`ausdruck1 ? ausdruck2 : ausdruck3 ;` entspricht

```
  if (ausdruck1)
    ausdruck2;
  else
    ausdruck3;
```

3.11 Typecast-Operator

`(type)ausdruck`

Beispiel:

```
int i = 3;
float f;

f = (float) i;
```

3.12 Weitere Operatoren

`+=,-=,*=,/=,&=,...`

Beispiel:

```
a += 3; // a = a + 3;
x /= y; // x = x / y;
```

3.13 Prioritätsreihenfolge der Operatoren

Operator	Name	Assoziativität
A++	Postinkrement	von links nach rechts
A–	Postdekrement	
A(B)	Funktionsaufruf	
A[B]	Feldindex	
A.B	Elementzugriff	
A->B	Elementkennzeichnung	
++A	Präinkrement	von rechts nach links
--A	Prädekrement	
-A	Unäres Minus	
+A	Unäres Plus	
!A	Logische Negation	
~	Einerkomplement	
∗A	Dereferenzierung	
&A	Adresse	
sizeof(A)	sizeof	
(type)A	Typumwandlung	
A∗B	Multiplikation	von links nach rechts
A/B	Division	
A%B	Restwert	
A+B	Addition	
A-B	Subtraktion	
A«B	Linksschieben	
A»B	Rechtsschieben	
A<B	Kleiner als	
A<=B	Kleiner gleich	
A>B	Größer	
A>=B	Größer gleich	
A==B	Gleichheit	
A!=B	Ungleichheit	
A&B	Bitweises UND	
A^B	Bitweises EXCLUSIV-ODER	
A|B	Bitweises ODER	
A&&B	Logisches UND	
A||B	Logisches ODER	
A?B:C	Bedingung	
A=B	Zuweisung	
A+=B	Additionszuweisung	
A-=B	Subtraktionszuweisung	
A∗=B	Multiplikationszuweisung	
A/=B	Divisionszuweisung	
A%=B	Restwertzuweisung	
A&=B	Bitweises-UND-Zuweisung	
A|=B	Bitweises-ODER-Zuweisung	
A^=B	Bitweises-EXCLUSIV-ODER-Zuweisung	
A«=B	Linksschiebe-Zuweisung	
A»=B	Rechtsschiebe-Zuweisung	
,	Komma	

4 Ausdrücke und Anweisungen

Unterschied von Ausdruck und Anweisung:
ausdruck; ≡ Anweisung

4.1 Selektionsanweisungen

4.1.1 If-Anweisung

Syntax:

```
  if (ausdruck)
    anweisung1;
 [else
    anweisung2;]
```

Achtung auf "Dangling else"!

4.1.2 Switch-Anweisung

Syntax:

```
switch (ausdruck)
{
  case konst_ausdruck1:
         anweisung1;
  case konst_ausdruck2:
         anweisung2;
  /* ... */
 [default:
         anweisungn;]
}
```

Hinweise:

- Der Einsprung erfolgt an der Stelle, an der der getestete Ausdruck mit dem konstanten Ausdruck übereinstimmt.
- Es werden dann *alle* Anweisungen bis zum Ende des Switch- Statements ausgeführt, außer es ist eine break-Anweisung vorhanden.
- Mehrere Werte durch Beistriche getrennt bzw. Bereichs- angaben sind in C nicht erlaubt.

```
switch (monat)
{
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12: tage = 31; break;
```

```
      case 4:
      case 6:
      case 9:
      case 11: tage = 30; break;
      case 2: if (schaltjahr(jahr))
                tage=29;
              else
                tage=28;
    }
```

4.2 Schleifen

4.2.1 while-Schleife

Syntax:

```
while (ausdruck)
  anweisung;
```

4.2.2 do-while-Schleife

Syntax:

```
do
  anweisung;
while (ausdruck);
```

4.2.3 for-Schleife

Syntax:

```
for (ausdruck1;ausdruck2;ausdruck3)
  ausdruck4;
```

Entspricht:

```
ausdruck1;
while (ausdruck2)
{
  ausdruck4;
  ausdruck3;
}
```

Beispiel:

```
int i;
for (i=0;i<10;i++)
  printf("%d\n",i);
scanf("%d",&i);
for (;i<10;i++)
  printf("%d\n",i);
for (;;);  // Endlosschleife
```

4.2.4 Kontrolltransferanweisungen

- `break`: Schleifenverarbeitung wird beendet.
- `continue`: Beendet den momentanen Schleifendurchlauf.

5 Arrays

5.1 Eindimensionale Arrays

Syntax: `datentyp array_name[anzahl_elemente];`

Hinweise:

- Elemente mit Index 0, ..., anzahl_elemente-1.
- anzahl_elemente muss zur Übersetzungszeit konstant sein.
- array_name ist ein Zeiger auf das Element mit dem Index 0. Diesem Arraynamen kann aber nichts zugewiesen werden, da er eine konstante Adresse darstellt.
- Vom Compiler und auch vom Laufzeitsystem werden keine Prüfungen der Feldgrenzen durchgeführt.

Beispiele:

```
int a[100]; // a[0], ..., a[99]
int *pa;

pa = a; // pa = &a[0];
```

5.2 Initialisierung

- globale Arrays: mit 0 automatisch
- lokale Arrays: nicht automatisch

Beispiel:

```
int a[3] = {1,2,3};

for (i=0;i<3;i++)
  a[i]=i+1;
```

Werden weniger Werte angegeben, als das Array Elemente hat, so werden die restlichen Elemente bei globalen Arrays mit 0 gefüllt, bei lokalen Arrays bleiben sie undefiniert.

Implizite Längenbestimmung: `int a[] = {1,2,3};`
Verwendung v.a. für Stringkonstante.

5.3 Mehrdimensionale Arrays

Definition: `datentyp array_name[dim1][dim2]...[dimn];`

Beispiel:

```
int matrix[5][3];
```

matrix $\equiv$ & matrix[0][0]

Ein zweidimensionales Array entspricht einem eindimensionalen Array, dessen Elemente eindimensionale Arrays sind.

Initialisierung:

```
int matrix[5][3] = { {1, 2, 3},
                     {4, 5, 6},
                     {1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9} };
                                          //  global  |  lokal
int matrix2[5][5] = { {1},                //  10000      1????
                      {0, 1},             //  01000      01???
      {0, 0, 1},          //  00100      001??
                      {0, 0, 0, 1},       //  00010      0001?
                      {0, 0, 0, 0, 1} };  //  00001      00001

int matrix3[5][5] = { {1, 2, 3, 4, 5},
                      { }, { }, { },
                      {6, 7, 8, 9, 10} };
```

5.4 Zeichenketten

- in C als Character-Array
- Abschluß duch '\ 0'

Beispiel:

```
char mystring[10] = "Hallo"; // Speicherinhalt: 'H' 'a' 'l' 'l' 'o' 0 ? ? ?
/* ... */
mystring = "Ungültig";       // Falsch!
strcpy(mystring,"Gültig");   // Richtig!

/* Einlesen: */
scanf("%s",mystring);          // Abbruch bei Leerzeichen
gets(mystring);                // weiter bei Leerzeichen

printf("%s",mystring);         // formatierte Ausgabe
puts(mystring);                // unformatierte Ausgabe
```

6 Funktionen

Funktionsdeklaration:

```
ftype fname(param1, param2, ... paramn);
            ^ptype [pname]
```

Funktionsdefinition:

```
ftype fname(param1, param2, ... paramn)
{           ^ptype pname (formaler Parameter)
 /* ... */
}
```

Hinweise:

- Bevor eine Funktion aufgerufen wird, muss sie entweder definiert oder deklariert wrden. Ansonsten wird automatisch der Funktionstyp int angenommen, was zur Fehlermeldung "function redefinition" führen kann.
- Wird für eine Funktion kein Funktionstyp vereinbart, so wird ebenfalls int als Typ angenommen.
- Häufig verwendete Funktionsdeklarationen werden in einem Headerfile zusammengefasst.

Funktionsaufruf:

```
fname(wert1, wert2, ... , wertn);
var = fname(wert1, wert2, ... , wertn);
```

- Die Übergabe der Parameter erfolgt mit "call by value". Eine Referenzübergabe kann in C nur mit der Übergabe von Adressen realisiert werden.
- Arrays werden beim Funktionsaufruf mit ihrem Namen übergeben, d.h. die Adresse des 1. Arrayelements wird übergeben. Ein Array wird "by reference" übergeben.

6.1 C-Speicherklassen

- auto: wird bei einer Variablendefinition *innerhalb* einer Funktion keine Speicherklasse angegeben, so gehört sie zur Speicherklasse auto. auto-Variable werden bei Betreten des Blocks, in dem sie definiert sind, am Stack angelegt, und ihr Speicherplatz wird bei Verlassen des Blocks wieder freigegeben.
- register: Der einzige Unterschied zur auto-Variable ist der, daß der Compiler versucht, die Variable in einem Register zu halten (nicht garantiert).
- extern: Unterschied zwischen

  - extern-Definition: sind außerhalb von Funktionen definierte Variable. Lebensdauer: gesamte Programmausführung. Gültigkeitsbereich: ab der Definition bis zum Ende des Source-Files (kann durch extern-Deklaration erweitert werden). Andere Bezeichnung: globale Variable.
  - extern-Deklaration: damit kann die Variable auch in anderen Source-Dateien verfügbar gemacht werden. Dem Compiler wird mitgeteilt, daß die Variable in einer anderen Source-Datei definiert ist.

  Externe Variable sollten nur sparsam verwendet werden. Stattdessen sollten die Daten als Parameter übergeben werden, wodurch die Allgemeinheit von Funktionen gewahrt bleibt.

- static:
  - local static: Gültigkeitsbereich: wie auto. Lebensdauer: gesamte Programmlaufzeit (die Variable einer Funktion wird nicht zerstört. Sie behält ihren Wert von Aufruf zu Aufruf bei.
  - global static: Gültigkeitsbereich: Sourcefile. In anderen Sourcedateien können sie nicht verfügbar gemacht werden. Lebensdauer: gesamte Programmlaufzeit. Damit kann auch der Gültigkeitsbereich von Funktionen, der normalerweise global ist, auf eine Source-Datei beschränkt werden.
- const: Die Variable darf nicht verändert werden.
- volatile: Dem Compiler wird mitgeteilt, daß die Variable 'heimlich' geändert weden kann. Daher wird bei jedem Zugriff die zugehörige Speicherstelle ausgelesen.

7 Strukturen

Beispiel: Die Daten eines Angestellten (Anrede, Name, PLZ, Ort, Straße, Geburtsdatum, ...) sollen verwaltet werden.

Nachteile bei Verwendung von Einzelkomponenten:

- Aufruf von Bearbeitungsfunktionen (Anlegen, Ändern, Lesen, Löschen, ...): jede Komponente muss einzeln übergeben werden.
- Soll ein weiterer Datensatz definiert werden, so muss für jeder Komponente die Variablendefinition wiederholt werden.
- Die logische Einheit der Komponenten ist im Programm nicht erkennbar.

7.1 Definition und Verwendung

Syntax:

```
struct [typname] {
  Komponente_1;
 [Komponente_2;
    /* ... */
  Komponente_n;]
} [Varname1, Varname2, ...]; /* Zugriff: varname.komponentenname */
```

7.2 Zulässige Operationen auf Strukturen

- Übergabe an eine Funktion (call by value) – formeller und aktueller Parameter müssen gleiche Zusammensetzung haben. Derselbe Typname sollte verwendet werden.

  Beispiel:

```
struct ang {
  char anrede[5];
  char name[20];
  int  plz;
};

int main(void)
{
  struct ang ang1, ang2;
  printf("Anrede: ");
  scanf("%s",&ang1.anrede);
  /* ... */
  write_ang(ang1);
  read_ang(ang2);
  /* ... */
  return 0;
}

void write_ang(struct ang angest)
{
  printf("Anrede: %s\n",angest.anrede);
  /* ... */
}

void read_ang(struct ang *pang)
{
  strcpy(pang->anrede,"Herr");
}
```

- Rückgabe einer Struktur aus einer Funktion

```
struct typname func_name (...)
{
/* ... */
}
```

- Zuweisung einer Strukturvariablen an eine weitere Strukturvariable.

  Voraussetzung: die beiden Strukturen müssen den gleichen Aufbau haben. Ob es tatsächlich funktioniert, hängt vom jeweiligen Compiler ab.

7.3 Initialisierung

Die Werte werden in geschweiften Klammern angegeben.

```
struct ang ang1 = {"Herr", "Huber", 4020, ...};
```

Es ist erlaubt, weniger Elemente zu initialisieren.

7.4 Kompliziertere Strukturdefinitionen

7.4.1 Arrays von Strukturen

Beispiel:

```
struct ang {
  char anrede[5];
  char name[20];
  int plz;
} firma[100];       // Array mit 100 Elementen vom Typ struct ang
```

7.4.2 Strukturen in Strukturen

Beispiel:

```
  struct datum {
    int tag;
    int monat;
    int jahr;
  };
  struct uhrzeit {
    int stunde;
    int minute;
    int sekunde;
  };

  struct termin {
    struct datum d;
    struct uhrzeit u;
  } t;
```

Zugriff:

```
t  /* Zugriff auf Element vom Typ struct termin */
t.d  /* Zugriff auf Element vom Typ struct datum  */
t.d.tag         /* Zugriff auf Element vom Typ int           */
```

7.5 Selbstdefinierte Datentypen

Syntax: `typedef alt_typ neu_typ;`

Beispiel:

```
typedef float real;
/* ... */
real x;

typedef struct datum {
            int tag;
int monat;
int jahr;
} datum;
```

8 Zeiger

8.1 Dynamische Speicherverwaltung

Statische Speicherverwaltung

- globale Variable, static Variable: Speicherplatzallokierung am Beginnen des Programms vom Compiler.
- lokale Variable: Speicherplatzallokierung beim Betreten des Blocks am Stack.
- Nachteile: am Beispiel Adressverwaltung mit Array: Zur Entwicklungszeit muss bekannt sein, wieviele Adressen ungefähr gespeichert werden sollen.

Dynamische Speicherverwaltung

- Während der Laufzeit kann Speicher am Heap angelegt werden. Die Speicherverwaltung (Anlegen, Freigeben und Adressverwaltung der Speicherbereiche) ist dem Programmierer übertragen.

8.2 Zeigerbegriff

Definition: `typname * zname;`

- Zeigervariable sind typisiert, d.h. sie dürfen nur auf Variablen dieses Typs zeigen (ausgenommen void).
- Es ist nicht möglich, den Typ des Zeigers während des Programmablaufs zu verändern.
- Durch die Definition der Zeigervariablen wird soviel Speicher reserviert, wie zur Darstellung des Zeigers nötig ist, aber *kein* Speicherplatz für ein Objekt vom jeweiligen Typ.
- Nach der Definition ist auch der Wert des Zeigers wie für jede andere Variable undefiniert. Er zeigt also auf irgendeine Speicherstelle im Adressraum.

Wertzuweisung: einfachste Möglichkeit: mit Adressoperator.

```
int i;
int * ptr;

i = 5;
ptr = &i;
```

Dereferenzierung: durch den unären Operator $*$ erhält man das durch den Zeiger referenzierte Objekt. Der Typ des Ausdrucks entspricht dem Grundtyp des Zeigers.

```
int i;
int * ptr;

i = 5;
ptr = &i;
*ptr = 6;
```

Dynamische Speicherzuweisung: `void * malloc(size)`

- Deklaration in stdlib.h oder malloc.h
- reserviert auf dem Heap size Bytes und liefert einen Zeiger auf das erste Byte zurück.
- Falls nicht genügend Speicherplatz vorhanden ist, wird NULL zurückgeliefert.
- Angabe der Größe in Bytes mit Konstante oder sizeof()-Operator

```
int * ptr;
ptr = malloc(sizeof(int));
if (ptr != NULL)
  *ptr = 15;
else
  /* Fehlermeldung */
```

Rückgabe von Speicher: `void free(void * p);`

8.3 Zeiger und Arrays

Ein Zeiger eines bestimmten Typs kann nicht nur auf eine einzelne Variable, sondern auf eine Folge von Werten dieses Typs (Array) zeigen $\Rightarrow$ Name des Arrays gleichbedeutend mit dem Zeiger auf das erste Arrayelement.

Unterschiede:

```
double x1[100];
double * x2;
```

- Zur Übersetzungszeit wird für x1 Speicher für 100 dobule-Variablen reserviert, während für x2 nur der Speicherplatz für eine Zeigervariable reserviert wird.
- Da x2 eine Variable ist, können ihr Werte zugewiesen werden. Eine Zuweisung an x1 ist nicht möglich, da x1 eine Konstante ist.

8.3.1 Zeigerarithmetik

Addition von Zeiger und int: x-Typ-Zeiger + int ⇒ x-Typ-Zeiger
Das Ergebnis zeigt auf das Element, das sich soviele Positionen nach dem Ausgangszeiger befindet wie durch den Integerwert angegeben wurde.

Subtraktion von Zeiger und int: x-Typ-Zeiger - int ⇒ x-Typ-Zeiger
Als Ergebnis wird ein Zeiger auf das Element zurückgegeben, das sich entsprechend viele Positionen vor dem Ausgangszeiger befindet.
Vorsicht auf Arraygrenzen!!!

Subtraktion von Zeiger und Zeiger: x-Typ-Zeiger - x-Typ-Zeiger ⇒ int
Der Ergebniswert ist der Abstand zwischen diesen beiden Zeigern angegeben in Elementen

Inkrement, Dekrement

```
x-Typ-Zeiger++
++x-Typ-Zeiger
x-Typ-Zeiger--
--x-Typ-Zeiger
```

Beispiel:

```
void strcpy(char *ziel, char *quelle)
{
  while (*ziel++ = *quelle++);
}
```

8.4 Zeiger auf Funktionen

Durch einen Funktionszeiger wird die Adresse des 1. Maschinenbefehls der Funktion gespeichert.

Definition:

- Auch Funktionszeiger sind typisiert, das bedeutet, daß mindestens der Typ des Rückgabewertes der Funktion, auf die der Zeiger verweist, festgelegt werden muss.

Beispiel:

```
double (*f)();
double (*f)(int, char);
void (*plot)(double(*f)());
```

Zuweisung:

```
#include <stdio.h>
#include <math.h>
int main(void)
{
  double (*f)();
  f = sin; // sin-Funktion in math.h, liefert double zurück
  return 0;
}
```

Aufruf:

```
/* siehe oben */
d = (*f)(3.1415);    /* d = sin(3.1415); */
```

Übergabe als Parameter:

```
#include <stdio.h>
#include <math.h>

void plot(double (*fp)())
{
  double x;
  printf("x          f(x)\n");
  for (x=0.0;x<=10.0;x=x+1.0)
    printf("%2.0f %10.6f\n",x,(*fp)(x));
}

int main(void)
{
  plot(sin);
  plot(cos);
  plot(sqrt);
  return 0;
}
```

9 Bilschirm I/O

9.1 Zeichenorientierte Ein-/Ausgabe

Sind nützlich, um komplexere Ein-/Ausgabe-Funktionen zu entwickeln.

9.1.1 putchar

`int putchar(int c); /* stdio.h */`

Die Ausgabe der Zeichen erfolgt verzögert. Die Zeichen werden solange in einem Puffer zwischengespeichert, bis entweder

- eine Zeilenschaltung erfolgt oder

- der Puffer voll ist oder
- das Programm beendet wird oder
- fflush aufgerufen wird.

9.1.2 getchar

```
int getchar(); /* stdio.h */
```

Zum Anzeigen eines Eingabeendes wird die Konstante EOF (-1) zurückgeliefert. Da diese Konstante mit keinem Zeichen kollidieren darf, wird ein Integer-Wert zurückgeliefert. Um die Eingabe über die Tastatur zu beenden, ist unter MS-DOS Ctrl+Z einzugeben, in Unix normalerweise Ctrl+D.

Auch die Eingabe über `getchar()` ist gepuffert. Der Puffer wird erst dann geleert, wenn die Enter-Taste eingegeben wurde.

9.2 Formatierte Ein- und Ausgabe

Daten sollen in Übereinstimmung zu ihren Typen ausgegeben werden.

9.2.1 printf

Einfache Formatanweisungen:
`%d, %c, %s, %x, %X, %o, %f, %g, %e, %E`

Erweiterte Formatanweisungen:
Syntax: `%[flags][width][.preci][l]type`

- Flags:
  - `-`: Ausgabe wird linksbündig formatiert
  - `+`: Positiven Zahlen wird ein +-Zeichen vorangestellt.
  - `#`: Oktalen wird 0, hexadezimalen Zahlen 0x vorangestellt.
  - `0`: Es wird linksbündig mit Nullen aufgefüllt.
- Width: die minimale Breite des Ausgabefeldes wird angegeben. Mit $*$ variierbar.
- .preci: Anzahl der auszugebenden Dezimalstellen. Mit $*$ wieder variabel gestaltbar.
- l: Die Long-Version des entsprechenden Grundtyps wird genommen.

9.2.2 scanf

Bestandteile des Formatstrings:

- Formatanweisungen: `%...`
- Whitespaces (z.B. Leerzeichen, Tabulatoren). Tauchen solche Zeichen zwischen 2 Formatanweisungen auf, so werden alle Whitespaces des Eingabestrings überlesen.

- Andere Zeichen: Beim Lesen wird ein solches Zeichen erwartet, es wird aber nicht gespeichert.
- Rückgabewert: Anzahl der eingelesenen Variablen.

`scanf()` liest beim Aufruf Zeichen für Zeichen von der Standardeingabe und vergleicht das Ergebnis ebenfalls Zeichen für Zeichen mit dem Formatstring. Passen die Zeichen, so wird weitergelesen, andernfalls abgebrochen.

Einfache Formatanweisungen: siehe `printf()`.

Erweiterte Formatanweisungen: `%[*][width][l]type`

- $*$: die nachfolgenden Zeichen werden gelesen, der Wert aber nicht gespeichert.
- l: der übergebene Zeiger ist Zeiger auf Long-Variable.
- width: gibt an, wieviele Zeichen maximal eingelesen werden.

10 Datei-I/O

10.1 Standard-Datei-I/O

Der Zugriff auf Dateien erfolgt in 3 Phasen:

- Öffnen/Anlegen der Datei
- Zugriff auf die Datei
- Schließen der Datei

10.1.1 Öffnen einer Datei

```
FILE * fopen(char * fname, char * mode);
```

- FILE ist eine Struktur, die Informationen über die Datei enthält. Der Rückgabewert ist für die nachfolgenden Operationen notwendig.
- fname enthält den Namen der Datei, eventuell auch mit Pfadangabe.
- mode gibt die Art an, in der die Datei geöffnet wird.
  - "r": wird zum Lesen geöffnet. Rückgabewert NULL: Datei existiert nicht.
  - "w": Anlegen einer Datei.
  - "a": Öffnet die Datei zum Schreiben am Ende der Datei bzw. wird die Datei automatisch angelegt, wenn sie nicht existiert.
  - "r+": Öffnen der Datei zum Lesen und Schreiben. Rückgabewert NULL wenn Datei nicht existiert.
  - "w+": Anlegen einer neuen Datei zum Verändern.
  - "a+": Öffnen der Datei zum Lesen oder zum Beschreiben am Ende.
  - "t": Textmodus (Default).
  - "b": Binärmodus.

Text- und Binärdateien:

- dienen dazu, Inhalte in lesbarer Form zu verarbeiten. Es ist also möglich, Textdateien mit einem Editor zu lesen und zu bearbeiten.
- Bei Binärdateien werden keine Konvertierungen an eingelesenen oder ausgegebenen Zeichen vorgenommen.

10.1.2 putc

```
int putc(char c, FILE * f);
```

- Ein einzelnes Zeichen wird in die Datei f geschrieben.
- Nach jedem Aufruf wird die Dateiposition um eine Stelle weiter bewegt. Dadurch können durch mehrfache Aufrufe fortlaufend Zeichen in die Datei geschrieben werden.
- Die Ausgabe wird gepuffert, die Zeichen werden also erst dann in die Datei geschrieben, wenn der Puffer voll ist.

10.1.3 getc

```
int getc(FILE * f);
```

- Ein einzelnes Zeichen wird aus der Datei f gelesen.
- Da nach jedem Aufruf die Dateiposition eine Stelle weiterbewegt wird, kann durch mehrfache Aufrufe ein Lesen von mehreren Zeichen aus der Datei durchgeführt werden.
- Rückgabewert: der ASCII-Wert des gelesenen Zeichens oder EOF beim Dateieende.

Beispiel: Datei kopieren

```
#include <stdio.h>

int main(void)
{
  FILE * quelle, * ziel;
  char name[100];
  char c;
  printf("Name der Quelldatei: ");
  scanf("%99s",name);
  if ((quelle=fopen(name,"rb"))==NULL)
    fprintf(stderr,"Datei %s kann nicht geöffnet werden\n",name);
  else
  {
    printf("Name der Zieldatei: ");
    scanf("%99s",name);
    if ((ziel=fopen(name,"wb"))==NULL)
      fprintf(stderr,"Datei %s kann nicht geöffnet werden\n",name);
```

```
    else
    {
      while ((c=getc(quelle))!=EOF)
        putc(c,ziel);
    }
  }
  return 0;
}
```

10.1.4 Schließen einer Datei

```
int fclose(FILE * f);
```

Rückgabewert:

- bei Erfolg 0
- bei Fehler EOF

10.1.5 fprintf und fscanf

Sie unterscheiden sich von den bisher bekannten Funktionen `printf` und `scanf` durch den zusätzlichen Parameter an erster Stelle der Parameterliste.

```
int fprintf(FILE * fp, char * format, ...);
int fscanf(FILE * fp, char * format, ...);
```

10.1.6 Die Standarddateien

	stdin	Standardeingabe
3 vordefinierte Dateibezeichner:	stdout	Standardausgabe
	stderr	Standardfehler

Dadurch könnten die Datei-I/O-Funktionen auch für die Ausgabe am Bildschirm bzw. für das Einlesen von der Tastatur verwendet werden.

10.2 Zusätzliche Funktionen zum Datei-I/O

10.2.1 fflush

```
int fflush(FILE * );
```

Der zur Datei f hinzugehörige Puffer wird geleert. Alle Zeichen auf dem Puffer werden in die Datei geschrieben.

10.2.2 rewind

```
void rewind(FILE * );
```

Der Dateizeiger wird an den Anfang der Datei gesetzt.

10.2.3 fseek

```
int fseek(FILE * fp, long offset, int origin);
```

Konstanten für origin:

- `SEEK_SET (0)`
- `SEEK_CUR (1)`
- `SEEK_END (2)`

Die Positionsangabe ergibt sich aus den Parametern offset und origin. Offset bezeichnet die Entfernung von dem Bezugspunkt origin. Offset kann auch negative Werte annehmen.

10.2.4 ftell

```
long ftell(FILE * );
```

Damit kann die aktuelle Position des Dateizeigers ermittelt werden. Der Rückgabewert ist der Offset, gezählt vom Dateibeginn (bei einem Fehler -1). Auch ftell arbeitet im Textmodus nicht korrekt. Immerhin kann der Rückgabewert verwendet werden, um eine Dateiposition festzuhalten und später wieder mit fseek anzuspringen

10.3 Typisierte Dateien

Im Allgemeinen wird über diese Dateien eine Folge identisch strukturierter Datensätze verwaltet. Diese Dateien werden im Binärmodus geöffnet und können sowohl sequentiell als auch direkt gelesen od. geschrieben werden.

Die folgenden Beispiele verwenden folgende Struktur als Datentyp:

```
typedef struct {
    int angnr;
    char angname[20];
    float anggehalt;
} angtyp;
```

10.3.1 Schreiben

```
int fwrite(void * buf, int size, int cnt, FILE * fp);
```

Es werden cnt Elemente, von denen jedes die Größe size hat, in die Datei fp geschrieben. Der Puffer buf muss mindestens $size * cnt$ Bytes an gültigen Daten enthalten. Es ist damit möglich, eine einzelne Struktur oder auch ein komplettes Array hinauszuschreiben.

Beispiel:

```
angtyp ang;
FILE * fp;
if ((fp=fopen(``ang.dat'',''wb+''))==NULL)
  printf(``Datei nicht geöffnet!\n'');
else {
  printf(``AngNr:'');
  scanf(``%d'',&(ang.angnr));
```

```
  while (ang.angnr!=0){
    printf(``AngName:'');
    scanf(``%19s'',ang.angname);
    printf(``AngGehalt:'');
    scanf(``%f'',&(ang.anggehalt));
    fwrite(&ang,sizeof(angtyp),1,fp);
    printf(``AngNr:'');
    scanf(``%d'',&(ang.angnr));
  }
  fclose(fp);
}
```

fwrite liefert die Anzahl der tatsächlich geschriebenen Sätze zurück.

10.3.2 Lesen

```
int fread(void * buf, int size, int cnt, FILE * fp);
```

fread liest cnt Elemente aus der Datei fp und schreibt sie in den Puffer buf. Da jedes Element die Größe size hat, muss der Puffer mindestens size*cnt Byte groß sein.

fread liefert die Anzahl der gelesenen Elemente zurück.

Beispiel:

```
angtyp ang;
FILE * fp;
if ((fp=fopen(``ang.dat'',''rb+''))==NULL)
  printf(``Datei nicht geöffnet!\n'');
else {
  while (fread(&ang,sizeof(angtyp),1,fp)==1)
    printf(``%3d %20s %8.2f\n'',ang.angnr,ang.angname,ang.anggehalt);
  fclose(fp);
}
```

A Sortieralgorithmen

A.1 Konventionen

In den folgenden Beispielen und Beschreibungen wird folgendes Unterprogramm benoetigt:

```
void swap(itemType a[], int i, int j)
{
  itemType t = a[i];
  a[i] = a[j];
  a[j] = t;
}
```

Implementiert wird jeweils eine Funktion `sort()`, die folgendermassen deklariert ist:

```
void sort(itemType a[i], int N);
```

A.2 Elementare Sortierverfahren

Diese sind großteils besser, falls die Anzahl der zu sortierenden Elemente kleiner als 500 ist.

- Begriffe:
  - interne Sortiermethode: alle Elemente im Hauptspeicher.
  - externe Sortiermethode: Sortieren auf Platte/Band.
- Sortiertypen:
  - Sortieren am Ort: kein zusätzlicher Speicherplatz wird benötigt (mit Ausnahme Hilfsvariable, Stack, ...)
  - Sortieren mit verketteter Liste: N Listenzeiger: sinnvoll bei großen Datensätzen.
  - Sortieren mit Kopie des Feldes.
- Stabilität: relative Reihenfolge gleicher Schlüssel bleibt erhalten. Bsp.: Studenten sind alphabetisch geordnet. Sortierung nach Noten $\Rightarrow$ Studenten mit gleichen Noten bleiben geordnet.

A.2.1 Selection Sort

Verfahrensweise:

- Finde kleinstes Element und tausche es gegen das an 1. Stelle aus.
- Finde kleinstes Element und tausche es gegen das an 2. Stelle aus.
- ...

Implementierung:

```
void selection(itemType a[], int N)
{
  int i,j, min;
  for (i = 1 ; i < N ; i++ )
  {
    min = i;
    for (j = i + 1 ; j <= N ; j++ )
      if (a[j] < a[min])
        min = j;
    swap(a, min, j);
  }
}
```

Eigenschaften:

- links vom Index i stehende Elemente sind bereits sortiert.
- gut geeignet zum Sortieren von großen Datensätzen mit kleinen Schlüsseln.
- jedes Element wird höchstens einmal bewegt.

A.2.2 Insertion Sort

Verfahrensweise: Es wird ein Element nach dem anderen betrachtet und an seinen richtigen Platz innerhalb der vorhergehenden Elemente eingefügt (größere Elemente werden um 1 Position nach rechts verschoben).

Implementierung:

```
void insertion(itemType a[], int N)
{
  int i, j, v;
  for (i = 1 ; i < N ; i++)
  {
    v = a[i];
    j = i;
    while (a[j-1] > v)
    {
      a[j] = a[j-1];
      j--;
    }
    a[j] = v;
  }
}
```

Problem: Ist das aktuelle Element (v) das kleinste, so geht die while-Schleife über das linke Ende hinaus.

Abhilfe:

- Markenschlüssel in a[0]: mindestens so klein wie das kleinste Element im Feld.
- while-Bedingung:

  ```
  ( (j > 0) && (a[j-1] > v) )
  ```

korrekte Implementierung

```
void insertion(itemType a[], int N)
{
  int i, j;
  itemType v;
  for (i = 1 ; i < N ; i++ )
  {
    v = a[i];
    j = i;
    while ( ( j > 0 ) && (a[j-1] > v) )
    {
      a[j] = a[j-1];
      j--;
    }
```

```
    a[j] = v;
  }
}
```

Eigenschaften:

- gut geeignet für fast sortierte Datenbestände

A.2.3 Bubble Sort

Verfahrensweise: Durchlaufe die Elemente N-mal und vertausche, wenn notwendig, benachbarte Elemente.

Implementierung:

```
void bubble(itemType a, int N)
{
  int i, j;
  for (i = N ; i > 0 ; i-- )
  {
    for (j = 1 ; j < i ; j++)
    {
      if (a[j-1] > a[j])
        swap(a,j-1,j);
    }
  }
}
```

Eigenschaften:

- nach dem ersten Durchlauf steht das größte Element an letzter Stelle, nach zweitem Durchlauf das zweitgrößte an vorletzter Stelle, usw.
- Bubble Sort ist eine Abart des Selection Sort mit viel mehr Aufwand.

A.3 Kompliziertere Sortierverfahren

A.3.1 Shell Sort

Erweiterung/Modifikation des Insertion Sort.

Problem bei Insertion Sort: da nur benachbarte Elemente verglichen werden, müssten z.B. N Verschiebungen durchgeführt werden, wenn das kleinste Element am Ende des Feldes steht.

Lösung: Erhöhung der Geschwindigkeit durch Vertauschen von Elementen, die weit voneinander entfernt sind.

Implementierung:

```
void shellsort(int a, int N)
{
  int i, j, h, v;
  for (h = 1; h <= N/9 ; h = 3 * h +1 ); /* Beispiel fuer Distanzbestimmung */
  for (; h > 0 ; h = h / 3 )
    for (i = h + 1; i <= N ; i++)
    {
      v = a[i];
      j = i;
      while (j > h && a[j-h] > v)
      {
        a[j] = a[j-h];
        j = j - h;
      }
      a[j] = v;
    }
}
```

Beispiel zur Berechnung einer Distanzfolge:

- N = 10:
- h = 1; h<=1,111...
- h = 4; h<=1,111...
- N = 100:
- h = 1; h<=100/9
- h = 4; h<=100/9
- h = 13; h<=100/9
- h = 4; h<=100/9
- h = 1; h<=100/9
- h = 0

Schlechte Distanzenfolge: 64, 32, 16, 8, 4, 2, 1
Elemente an ungeraden Positionen werden erst beim letzten Durchlauf mit Elemente an geraden Positionen verglichen.

A.3.2 Quick Sort

- Der am haeufigsten verwendete Algorithmus
- Sortieren am Ort
- N Elemente $\Rightarrow$ durchschnittl. $N * ld(N)$ Operationen
- extrem kurze innere Schleife

- Rekursive Implementierung
- Grundlegender Algorithmus: beruht auf dem Zerlegen einer Datei in 2 Teile und anschliessendem Sortieren der beiden Teile.

```
void quicksort(int a[], int l, int r)
{
  int i;
  if (r > l)
  {
    i = partition(l,r);
    quicksort(a,l,i-1);
    quicksort(a,l+1,r);
  }
}
```

- Entscheidendes Element ist die Funktion partition, die das Feld so umordnen muss, dass folgende Bedingungen erfuellt sind:
  - das Element `a[i]` befindet sich an einem endgueltigen Platz.
  - Alle Elemente `a[l]` bis `a[i-1]` sind kleiner oder gleich `a[i]`.
  - `a[i+1]` bis `a[r]` sind groesser oder gleich `a[i]`.
- Strategie fuer partition:
  - `a[r]` wird als Element gewaehlt, das in seine endgueltige Position gebracht werden soll.
  - Durchsuche das Feld von links beginnend, bis ein Element groesser als `a[r]` gefunden wird.
  - Durchsuche das Feld von rechts beginnend, bis ein Element kleiner als `a[r]` gefunden wird.
  - Tausche die beiden Elemente aus.
  - Wiederhole Punkt 2 bis 4 solange, bis sich der linke und recht Index treffen.
  - Vertausche `a[r]` mit dem Element, auf das der linke Index verweist.
  - `a[r]` steht an der endgültigen Position. Linker Index kann als Returnwert zurueckgeliefert werden.

Implementierung:

```
void quicksort(int a[], int l, int r)
{
  int v, i, j, t;
  if (r > l)
  {
    v = a[r];
    i = l - 1;
    j = r;
    for (;;)
```

```
    {
      while (a[++i] < v);
      while (a[--j] > v);
      if (i==j)
break;
      t = a[i];
      a[i] = a[j];
      a[j] = t;
    }
    t = a[i];
    a[i] = a[r];
    a[r] = t;
    quicksort(a,l,i-1);
    quicksort(a,i+1,r);
  }
}
```