Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 4 (8 Seiten)

Strukturen

Es können Datentypen strukturiert zusammengefaßt werden zu komplexeren Daten-"strukturen". Zum Beispiel kann etwa in einem Grafikpaket es sinnvoll sein, Punkte auf dem Bildschirm– durch ihre Pixel-Koordinanten beschrieben– zu einem neuen Datentyp zusammenzufassen. Dies gelingt durch die folgende Deklaration.

```
struct punkt {
        int x; int y;
};
```

Damit wird ein neuer Datentyp geschaffen namens `struct punkt`. Es sind dann Deklarationen möglich wie

```
struct punkt a, b, z;
```

m usw., syntaktisch analog zu

```
int a, b, z;
```

Die Deklaration

```
struct punkt p;
```

definiert eine Variable `p` vom Typ `struct punkt`. Durch

```
struct punkt p = { 123, 456 };
```

wird mit der Deklaration eine Initialisierung vorgenommen, auf die Komponenten (= "members") der Struktur wird mittels des `'.'`-Operators durch die syntaktische Konstruktion

```
structure-name.member
```

zugegriffen.

Aufgabe 5.1: Welchen Wert liefert in diesem Fall `sizeof(struct punkt)` *?*

Im obigen Fall bezeichnet `p.x` die `x`-Komponente von `p`. Zum Beispiel kann man mit

```
printf("%d %d", p.x, p.y);
```

die Koordinaten von `p` ausdrucken, mit

```
double dist, sqrt(double);
...
dist = sqrt((double)p.x * p.x + (double)p.y * p.y);
```

kann man nach einer bekannten Formel den Abstand zum Punkt `(0,0)` berechnen.
Strukturen können geschachtelt werden. Durch

```
struct recht {
        struct punkt lu;
        struct punkt ro;
};
```

läßt sich ein Datentyp `recht` deklarieren, der ein (horizontales) Rechteck durch Fixieren des linken unteren und rechten oberen Eckpunktes beschreibt. Nach der Deklaration

```
struct recht screen;
```

bezeichnet z. B. `screen.lu.x` die `x`-Koordinate der linken unteren Ecke usw.

Strukturen können als Variable an Funktionen übergeben und von ihnen als Wert zurückgegeben werden. Zum Beispiel ist folgendes eine Funktion, die aus zwei `int` ein `p` macht:

```
struct punkt mkpunkt(int x, int y) {
        struct punkt temp;
        temp.x = x;
        temp.y = y;
        return temp;
}
```

Zwei Punkte können (als Vektoren) addiert werden etwa durch:

```
struct punkt add(struct punkt p1, struct punkt p2) {
        p1.x += p2.x;
        p1.y += p2.y;
        return p1;
}
```

Durch `struct punkt *p;` wird ein Zeiger `p` auf Objekte vom Typ `struct punkt` erklärt. `(*p).x` bezeichnet dann die `x`-Koordinate eines `struct punkt`-Objektes, auf das `p` zeigt. Äquivalent hierzu ist die Notation `p->x`.

Beispiel: Kalenderdatum

Kalenderdaten sind ein Beispiel für eine mehrteilige Struktur, bestehend z. B. aus Tag, Monat, Jahr. Sie können als Einheit aufgefasst und so bearbeitet werden, z. B., um die Anzahl der Tage zwischen zwei Daten zu ermitteln, etwa für Zinsberechnungen etc.
In unserem Beispiel haben wir auch noch zu jedem Datum die Nummer des Tages im Jahr als Komponente hinzugenommen.

Aufgabe 5.2: Erweitern Sie die Struktur, so dass auch eine Komponente "Kalenderwoche" enhalten ist. (Hinweis: Als erste Kalenderwoche eines Jahres gilt die erste Woche, die einen Donnerstag enthält. Nach Konvention beginnt eine Woche jeweils am Montag.)

Aufgabe 5.3: Erweitern Sie dies Programm so. dass die Funktion `add` *auch negative* `int` *verarbeitet.*

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

char *tage[] = { "Sonntag", "Montag", "Dienstag",
                 "Mittwoch", "Donnerstag", "Freitag", "Sonnabend" };

struct datum {                    /* t m j = tag monat jahr              */
  int  t;  int  m; int j; int n;  /* n = Nummer des Tages im Jahr        */
};
typedef struct datum datum;

/* Hier ist die zweite Komponente ein int-Array der Laenge 2: */
typedef struct monat { char *name;  int n[2]; } monat;

/* Array aus struct monat : Die Zahlen geben die Anzahl Tage an,
   die nach Ende das Monats verflossen sind - in der zweiten Spalte fuer
   Schaltjahre */
monat mon[] = {
  {"null",    {  0,  0}},
  {"Januar",  { 31, 31}},   {"Februar",   {59, 60}},  {"Maerz",    { 90, 91}},
  {"April",   {120,121}},   {"Mai",      {151,152}},  {"Juni",     {181,182}},
  {"Juli",    {212,213}},   {"August",   {243,244}},  {"September",{273,274}},
  {"Oktober", {304,305}},   {"November", {334,335}},  {"Dezember", {365,366}}
};

int schalt(int j) { /* gregorianisch */
  return  ( j % 4 == 0 &&  j % 100 != 0 || j % 400 == 0 );
}
int jahr_laenge(int j)  { return 365 + schalt(j);   }

int mon_laenge(datum d) { /* braucht "zulaessiges" d.m */
  return mon[d.m].n[ schalt(d.j) ] - mon[d.m-1].n[ schalt(d.j) ];
}

void printdatum(datum d) { /* braucht "zulaessiges" d.m */
  printf("%d. %s, %d\n", d.t, mon[d.m].name, d.j);
}

/* berechnet die Nummer des Tages im Jahr, braucht "zulaessiges" Datum*/
void normalize(datum *p) {
  p->n = p->t + mon[ p->m - 1 ].n[ schalt(p->j) ];
}

/* Baut ein Datum und kontrolliert Zulaessigkeit: */
datum makedatum(int tag, int monat, int jahr) {
  datum d = { tag, monat, jahr, 0 };
  if ( 1 <= d.m && d.m <= 12 && 1 <= d.t && d.t <= mon_laenge(d) ) {
    normalize(&d);
    return d;
  }
  printf("Das Datum gibt es nicht : %d.%d.%d\n", tag,monat,jahr);
  exit(1);
}
```

```
/* Liefert zum Tag n >= 1 und Jahr j
   das Datum des n-ten Tages ab und inklusive 1.1.j */
datum dat(int j, int n) {
  int i;
  for (i=1; i <= 12; i++)
    if (n <= mon[i].n[ schalt(j) ] )
      return makedatum( n - mon[i-1].n[ schalt(j) ], i, j);
  return dat( j+1, n - jahr_laenge(j) );
}

/* Berechne u - v */
int diff(datum u, datum v) {
  int n, i;
  if (u.j < v.j || (u.j == v.j && u.n < v.n ) ) return -diff(v,u);
  n = u.n;
  for(i = u.j - 1; i >= v.j; i--)
    n += jahr_laenge(i);
  n -=  v.n;
  return n;
}

/* Liefert das Datum des folgenden Tages */
datum inc_tag(datum v) {
  return dat(v.j, v.n+1 );
}

datum add(datum u, int t) {
  if (t >= 0) {
    return dat(u.j, u.n + t);
  }
  printf ("Nicht implementiert (t negativ) : %d\n", t);
  exit(1);
}

char *wochentag(datum d) {
  datum e; int t;
  e = makedatum(31,12,2000);      /* ist ein Sonntag */
  t = diff(d,e) % 7;
  if ( t < 0 )  t += 7;
  return tage[t];
}

main() {
  int t, m, j;
  datum d, dd;
  while(1) {
    printf("Bitte Tag Monat Jahr (als Zahlen) eingeben : ");
    scanf("%d %d %d", &t, &m, &j);
    d = makedatum(t,m,j);
    printf("Das Datum ist : "); printdatum(d);
    printf("Es ist der %d-te Tag im Jahr %d.\n", d.n, d.j);
    printf("Es ist ein %s.\n", wochentag(d) );
```

```
        printf("Der naechste Tag ist : ");  printdatum( inc_tag(d) );
        printf ("wieviele Tage dazu ? "); scanf("%d", &t);
        dd = add(d,t);
        printdatum( dd );
        printf("Unterschied nach diff : %d\n", diff(d,dd));
      }
    }
```

Strukturen können "selbstreferentiell" sein, d. h. auf Zeigeobjekte des eigenene Typs verweisen.

In Verzeichnis der Beispielprogramme findet sich die Datei `baum.c`, die die Struktur eines Binärbaums implementiert.

```
          struct node {
                   char *wort;
                   int  zahl;
                   struct node *links;
                   struct node *rechts;
            };
    typedef struct node knoten;
```

Jedes Objekt `struct node` enthält zwei Zeiger `links, rechts` auf ein `struct node`.

Diese Struktur ist geeignet, ungeordnete Daten zu sortieren.

Mit dem zugehorigen Programm kann man die Häufigkeit des Auftretens von Wörtern abzählen.

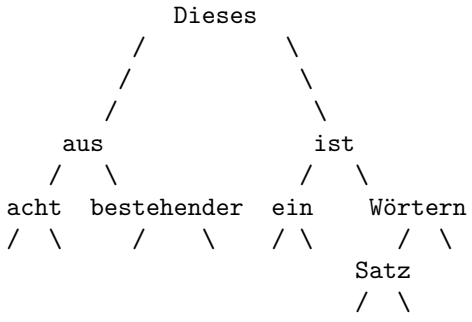
Aufgabe 5.4: Studieren Sie die Wirkungsweise des Programms `baum.c`.

Hinweise: Das Programm liest Wörter aus der Standard-Eingabe und fügt sie über die Funktion `suche()` in einen "Binärbaum" ein. Dabei wird der Binärbaum selbst konstruiert: Er besteht aus "Knoten" `struct node` mit vier Komponenten: einem String (`char *wort`), einem Häufigkeitszähler (`int zahl`) und zwei Zeigern auf Knoten (`struct node *links, *rechts`), die ihrerseits tatsächlich auf weitere Knoten zeigen oder den Wert `NULL` haben koennen. Anfänglich hat der Eingangsknoten (die "Wurzel" des Baumes) den Wert `NULL`.

Für ein gelesenes Wort wird der Baum auf folgende Weise durchsucht: Beginnend bei der Wurzel, wird entweder ein Knoten mit dem Wort als String eingerichtet, wobei der Zähler auf 1 und die "Teilbäume" `links, rechts` auf `NULL` initialisiert werden, oder das Wort wird mit dem Wort am betrachteten Knoten verglichen. Bei Gleichheit wird ledigliche die Grösse `zahl` inkrementiert. Geht das neue Wort dem String am Knoten lexikografisch voran bzw. folgt es ihm lexikografisch nach, wird für das Wort in gleicher Weise mit dem linken bzw. rechten Teilbaum verfahren, d.h. es wird dort jeweils entweder ein neuer Knoten eingerichtet oder der Vergleich mit dem dort vorgefundenen String vorgenommen.

Z. B. ergibt der folgende Satz den darunter stehenden Baum (alle Zähler = 1):

"Dieses ist ein aus acht Wörtern bestehender Satz"



```
/* titel: baum.c; Binaerbaum als Beispiel zu selbstreferentiellen Strukturen  */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <locale.h>
#define MAX_WORT      40
char *progname;
struct  node {
            char *wort;
            int  zahl;
            struct node *links;
            struct node *rechts;
};
typedef struct node knoten;
/*  main liest aus der Standard-Eingabe die Woerter und gibt sie
lexikographisch sortiert mit Haeufigkeitsangabe auf die
Standard-Ausgabe aus. */
int dcount = 0;
main() {
    knoten *unterbaum, *suche();
    int wcount=0;
    char   wort [ MAX_WORT ] ;
    int    c;
    setlocale(LC_ALL, "deutsch");
    unterbaum = NULL;
    while ( ( c = lieswort( wort, MAX_WORT) ) != EOF )
      if (c == 'a') {
        unterbaum = suche( unterbaum, wort );
        wcount++;
      }
    printf("%d Woerter gelesen, davon paarweise verschieden: %d\n",wcount, dcount);
    inorder( unterbaum );
}
/*  lieswort liest aus einer Datei ein Wort der Laenge < lim in den
Speicher, auf den w zeigt, und gibt 'a' zurueck, falls es wirklich
Alphazeichen gelesen hat, jedoch EOF im Fall zu grosser
Wortlaenge. Falls Nicht-Alpha-Zeichen gelesen werden, werden diese
einzeln zurueckgegeben. */
int lieswort(char *w, int lim) {
    int c;
    while ( !isalpha( c = getchar() ) )
      if ( c == EOF ) return c;
    *w++ = c; lim--;   /* Alpha-Zeichen gesehen, suche nach weiteren:  */
    while ( isalpha( c = getchar() )  && lim > 0 ) {
      *w++ = c; lim--;
    }
    if   (lim == 0) {
      printf("Error: Wort zu lang "); return EOF;
    }
    else *w = 0;
    return 'a';
}
```

```
/*   suche  durchsucht den Baum auf das Vorkommen des Wortes w,
     traegt es gegebenenfalls lexikographisch richtig ein bzw. erhoeht
     dessen Zahl und gibt den Pointer auf seinen Knoten zurueck.
     Die c-Funktion strcmp (resp. strcoll) vergleicht zwei
     Zeichenketten lexikographisch
     mit Rueckgabe einer negativen Zahl, 0 oder einer positiven Zahl. */
knoten *suche(knoten *p, char *w) {
     knoten *p_knoten();
     char *merke_wort();
     int  cond;
     if   (p == NULL) {
          p = p_knoten();
          p->wort = merke_wort(w);
          p->zahl = 1;
          p->links = p->rechts = NULL;
          dcount++;
     }
     else if   ((cond = strcoll(w, p->wort)) == 0)
               p->zahl++;
          else if   (cond < 0)
                    p->links = suche(p->links, w);
               else p->rechts = suche(p->rechts, w);
     return p ;
}
/* p_knoten gibt einen Zeiger auf freien Speicherplatz fuer eine
Knoten-Variable zurueck. */
knoten *p_knoten(void) {
     return (knoten *)malloc( sizeof( knoten ) ) ;
}
/*   inorder  gibt den Unterbaum, auf den p zeigt, in Inorder aus.
     (rekursive Version!)                  */
inorder(knoten *p) {
     if (p != NULL){
       inorder(p->links);
       printf("%4d %s\n", p->zahl, p->wort);
       inorder(p->rechts);
     }
}
/*  merke_wort speichert das Wort, auf dessen ersten Charakter s
zeigt, und gibt einen Zeiger auf dessen neue Lokation an.  die
c-Funktionen strlen, strcpy geben die Laenge einer Zeichenkette
zurueck beziehungsweise kopieren diese. */
char *merke_wort(char *s) {
    char *p;
    if ( ( p = (char *)malloc( strlen(s)+1 ) ) != NULL )
       strcpy(p, s);
    return p ;
}
```

Parsen komplizierter Deklarationen:

Im Verzeichnis `dcl` findet man kleine Programme aus K&R, die das Parsen von Deklarationen erlauben. Die Programme können im Verzeichnis `dcl` durch den Befehl `make` kompiliert werden, die ausführbaren Versionen heissen dann `dcl` und `undcl`.
Das Programm `dcl` wandelt Deklarationen in eine Textbeschreibung um:
Typische Ein- und Ausgaben von `dcl` sind:

```
./dcl
 int *funct[]()
 funct:  array[] of function returning pointer to int
 double (*FUNCT[])()
 FUNCT:  array[] of pointer to function returning double
 char *argv[]
 argv:  array[] of pointer to char
 char **argv
 argv:  pointer to pointer to char
```

`undcl` wandelt umgangssprachliche Beschreibungen in Deklarationen um.
Typische Ein- und Ausgaben von `undcl` sind folgende (die in Klammern gesetzten Zeilen gehören nicht zur Ein- oder Ausgabe und sollen hier nur der Erläuterung dienen):

```
./undcl
     (wurzel ist Funktion auf double)
 wurzel () double
 double wurzel()
     (argv ist Array von Pointern auf char)
 argv [] * char
 char (*argv[])
     (argv ist Pointer auf Pointer auf char)
 argv * * char
 char (*(*argv))
     (FCT ist Array von Funktionen auf Pointer auf long)
 FCT [] () * long
 long (*FCT[]())
    (funct ist Array von Pointern von Funktionen auf double)
 funct [] * () double
 double (*funct[])()
```

Aufgabe 5.5: Übersetzen und testen Sie die Programme `dcl` *und* `undcl` *anhand der in diesem Übeungsblatt vorkommenden Deklarationen.*