Einführung in die Programmiersprachen C und C++

Markus Kirschmer, Fakultät für Mathematik

Übungsblatt 4

Datenstrukturen

Datentypen können strukturiert zusammengefaßt werden zu komplexeren Datenstrukturen. 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 p;
```

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

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

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

```
structure-name.member
```

zugegriffen.

Aufgabe 4.1. *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` ausgeben. Mit

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

kann man nach einer bekannten Formel den Abstand von `p` zum Ursprung berechnen.

Strukturen können geschachtelt werden. Durch die Koordinaten von `p` ausgeben. Mit

```
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 Variablen an Funktionen übergeben und von ihnen als Wert zurückgegeben werden. Zum Beispiel ist folgendes eine Funktion, die aus zwei `int` ein `struct punkt` 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 des Punktes, auf den `p` zeigt. Äquivalent hierzu ist die Notation `p->x`.

Beispiel 1: Komplexe Zahlen

Das Programmbeispiel `complex.c` zeigt eine Anwendung von structs für die programmiertechnische Behandlung komplexer Zahlen $z = x + iy$, wobei x, y reelle Zahlen sind und i eine Wurzel aus -1.

Die Komponenten von z sind der Real- und Imaginäteil x und y. Näheres im Programm, zu übersetzen mit dem Befehl `gcc complex.c -lm`. Auf ähnliche Weise lassen sich Vektoren und Matrizen behandeln.

Hinweis: Moderne Compiler, welche den Standard C99 unterstützen, kennen schon komplexe Zahlen, siehe `man complex`. Hier ein Beispielprogramm:

```
/* comple.c                              */
/* check that exp(i * pi) == -1          */
/* compile with: gcc complex.c -lm       */
#include <math.h>          /* for atan */
#include <stdio.h>
#include <complex.h>

int main(void) {
  double pi = 4 * atan(1.0);
  double complex z = cexp(I * pi);
  printf("exp(i*pi) = %f + %fi\n", creal(z), cimag(z));
}
```

Beispiel 2: 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 4.2. *Erweitern Sie das Progamm um eine Funktion* `int Kalenderwoche(datum d)` *das die Kalenderwoche des Datums* `d` *zurückliefert.*

Hinweis: Kalenderwochen beginnen an einem Montag und die erste Kalenderwoche enthält immer den 4.1.

Aufgabe 4.3. *Erweitern Sie das Programm so, daß die Funktion* `add` *auch negative* `int` *verarbeitet.*

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

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

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[] = {
  {"",        {  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);
}
```



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

int wochentag(datum d) {
  /* Gauss Formel für den 1.1. im Jahr d.j */
  int w = (1 + 5*((d.j-1)%4) + 4*((d.j-1)%100) + 6*((d.j-1)%400)) % 7;
  return (w + d.n - 1) % 7;
}

int main() {
  int t, m, j;
  datum d, dd;

  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", tage[ 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., über Zeiger auf Objekte 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.

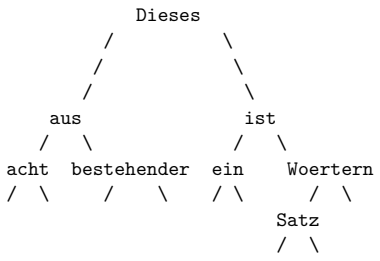
Mit dem zugehörigen Programm kann man die Häufigkeit des Auftretens von Wörtern zählen.

Aufgabe 4.4. *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. Dieser 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 auf weitere Knoten zeigen oder den Wert `NULL` haben können. Anfänglich hat der Eingangsknoten (die sogenannte „Wurzel“ des Baumes) selbst 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 lediglich der Zähler `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“



Die *Höhe* (engl. height) eines leeren Binärbaums ist 0. Hat ein Binärbaum `b` die beiden Teilbäume `l` und `r`, so definiert man seine Höhe als

```
height(b) = max( height(l), height(r)) + 1
```

Der obige Baum hat z.B. die Höhe 4.

[2]

```
/* titel: baum.c; Binaerbaum als Beispiel zu selbstreferentiellen Strukturen  */
#include <stdio.h>
#include <ctype.h>
#include <wctype.h>
#include <string.h>
#include <stdlib.h>
#include <locale.h>
#define MAX_WORT      40
struct  node {
            char *wort;
            int  zahl;
            struct node *links;
            struct node *rechts;
};
typedef struct node knoten;

/* dcount zählt die paarweise verschiedenen Worte */
int dcount = 0;

/* lieswort liest aus einer Datei ein Wort der Laenge < lim in den Speicher,
   auf den s zeigt, und gibt 1 zurueck, falls es wirklich ein Wort gelesen hat,
   jedoch 0 im Fall zu grosser Wortlaenge oder am Dateiende. */

int lieswort_utf8(char *s, int lim) {
  /* static: Variablen bleiben nach Beendigung der Funktion bestehen */
  static int n=0;
  static wchar_t wline[2000], *wlp=wline;
  wchar_t w[lim], *wp = w;
  int max=lim-1;
  /* auf schon angefangener Zeile Wortanfang suchen */
  if (n>0) {
    while (*wlp && !iswalpha(*wlp)) wlp++;  /* naechsten Buchstaben suchen */
    if (!*wlp) n=0;                         /* Zeile zu Ende */
  }
  /* falls Zeile zu Ende: neue Zeile lesen und Wortanfang suchen */
  while (n==0) {
    char line[2000];
    if (!fgets(line,2000,stdin)) return 0;  /* Zeile lesen */
    n = mbstowcs(wline,line,2000);          /* Zeile konvertieren */
    wlp = wline;
    while (*wlp && !iswalpha(*wlp)) wlp++;  /* finde Wortanfang */
    if (!*wlp) n=0;                         /* Zeile zu Ende */
  }
  /* nun haben wir einen Wortanfang */
  while (iswalpha(*wlp) && max>0) {         /* Wort kopieren */
    *wp++=*wlp++; max--;
  }
  *wp = L'\0';                              /* Wortende markieren */
  if (max==0 || wcstombs(s,w,lim)==lim) {   /* nach UTF-8 konvertieren */
    printf("Error: Wort zu lang\n");
    return 0;
  }
  return 1;
```

```
}

/* 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) {
  int cond;
  if (p == NULL) {
    /* mit malloc Speicher für neuen Knoten reservieren. */
    p = (knoten *)malloc( sizeof( knoten ) ) ;
    p->wort = strdup(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 ;
}

/* inorder gibt den Unterbaum, auf den p zeigt, sortiert aus
   (rekursive Version!) */
void inorder(knoten *p) {
  if (p != NULL){
    inorder(p->links);
    printf("%4d %s\n", p->zahl, p->wort);
    inorder(p->rechts);
  }
}

/* main liest aus der Standard-Eingabe die Woerter und gibt sie lexikographisch
   sortiert mit Haeufigkeitsangabe auf die Standard-Ausgabe aus. */
int main() {
  knoten *baum=NULL;
  int  wcount=0;
  char wort[ MAX_WORT ];

  setlocale(LC_ALL, "de_DE.UTF-8");
  while ( lieswort_utf8( wort, MAX_WORT) ) {
    baum = suche( baum, wort );
    wcount++;
  }
  printf("%d Woerter gelesen, davon paarweise verschieden: %d\n", wcount, dcount);
  inorder( baum );
}
```

Aufgabe 4.5. *Ändern sie das Programm* `baum.c` *so ab, daß die Höhe des Binärbaums mit ausgegeben wird.*