Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 9 (4 Seiten)

Selbstreferentielle Strukturen und Klassen

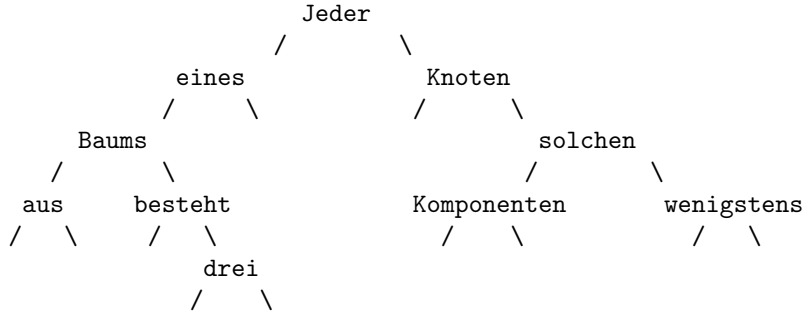
Zum Ordnen einer Folge von "zufällig" gereihten Daten eignet sich die Struktur des Binärbaums, wie in Blatt 4 durch ein C-Programm demonstriert. Hier eine C++-Version.

Zur Erinnerung (vgl. Blatt 4):

Jeder Knoten eines solchen Binärbaums besteht aus wenigstens drei Komponenten: den Daten, die zu diesem Knoten gehören sollen, und zwei Zeigern auf jeweils den "linken" und "rechten Teilbaum" zu diesem Knoten, in denen die Daten vorhanden sein sollen, die den Daten des aktuellen Knotens in der Ordnung (z. B. alphabetisch) vorangehen bz. nachfolgen. (Im vorliegenden Fall gibt es eine weitere Komponente vom Typ `int`, die zählt, wie häufig ein Wort gesehen wurde.)
Nehmen wir an, wir wollen die Wörter eines Textes auf diese Weise organisieren, z. B. die Wörter des Satzes:

"Jeder Knoten eines solchen Baums besteht aus wenigstens drei Komponenten"

Der Reihenfolge nach verarbeitet, erhalten wir folgende Struktur:



Das Vorgehen ist wie folgt: Zu einem gelesenen Wort wird ein Knoten konstruiert mit dem Wort als Dateneintrag und einem "linken" und "rechten" Zeiger auf einen Knoten gleichen Typs, die zunächst auf "nichts" (= null) zeigen.
Das nächste gelesene Wort wird mit dem ersten verglichen, geht es diesem in der (alphabetischen) Ordnung voran, wird es als "linker Unter-Knoten" eingetragen, folgt es in der Ordnung nach, wird es als "rechter Unter-Knoten" verwendet.
Jedes weitere Wort wird ähnlich behandelt: Es wird mit dem Anfangs-("Wurzel")-Knoten verglichen, geht es in der Ordnung diesem voran bzw. folgt es ihm nach, wird in den linken bzw. rechten Teilbaum gegangen und wieder mit dem dort stehenden Wort verglichen, und so wird fortgefahren, bis man zu einem Knoten kommt, an dem noch kein Wort steht. Dort wird es als Knoten eingetragen.
Mit einer Zählerkomponente bei jedem Knoten kann man die Häufigkeit der Wörter zählen.
Die so entstehende Struktur ist "selbstreferentiell", weil jeder Knoten auf zwei Objekte *gleichen* Typs zeigt.
Eine geordnete Ausgabe erhält man durch die folgende rekursive Vorschrift, bezogen auf die einzelnen Knoten, mit Start an der "Wurzel" des Baums:

1. Ausgabe des linken Teilbaums, falls vorhanden.
2. Ausgabe des Wortes am aktuellen Knoten, falls vorhanden.
3. Ausgabe des rechten Teilbaums, falls vorhanden.

Eine C-Implementation findet man in K&R, 2. Auflage, Chapter 6, Sect. 6.5.
Wir geben hier eine C++-Klasse an, die das gleich leistet wie das C-Programm von Blatt 4. Um deutsche Wörter inclusive Umlauten korrekt sortieren zu können, verwenden wir hier wieder die `locale`-Funktionen.

Aufgabe 9.1: Erweitern Sie die Funktionalität, so dass zu jedem Knoten=Wort auch noch die Zeilenzahlen seines Auftretens festgehalten werden.

Bemerkung: Die Datei `btree.c` im Verzeichnis zu diesem Übungsblatt enthält auch Funktionen, die das korrekte Löschen eines Knotens aus einem geordneten Binärbaum erlauben.

```
// btree.cc : Eine Binaerbaum-Klasse

#include <ctype.h>
#include <iostream.h>
#include <locale.h>
#include "string-class.h"

template<class T>
class tree {
  struct node{
    T e; int c; node *l, *r;   // Element, Zaehler, linker/rechter Teilbaum
    node() { c = 1; l = NULL; r = NULL; }
  } *n;
public:
  tree() { n = NULL; }        // Konstruktor : fuer Deklarationen
  tree(node *nn) { n = nn;}   // Konstruktor : fuer Initialisierungen

  node* insert(const T &e) {  // Member-Fumktion :
    if (n == NULL) {
      n = new node;           // liefert und initialisiert struct node n
      n->e = e;               // Datenelement
    }
    else if (e == n->e) { n->c++; } // e vorhanden, erhoehe Zaehler
    else if (e <  n->e)             // ordne lexikographisch ein :
      n->l = tree(n->l).insert(e);
    else
      n->r = tree(n->r).insert(e);
    return this->n;
  }
  friend ostream& inorder(ostream& os, tree t) {
    if (t.n) {
      inorder(os, t.n->l);
      os << t.n->e << '(' << t.n->c << ")\n" ;
      inorder(os, t.n->r);
    }
    return os;
  }
};

bool getword(istream& is, string& s) {
  if ( ! (is >> s) ) return false ;
  char *p, *q;
  p = q = &s[0];
  while (*p) {
    if ( isalpha(*p) ) // Nur alpha-Zeichen werden beruecksichtigt
      *q++ = *p;       // Kopieren von Zeichen
    p++;
  }
  *q = '\0';
  return true;
}
```

```
int main() {
  string s;
  tree<string> tr;              // Deklariere Baum aus Strings
  setlocale(LC_ALL, "deutsch"); // cf. "man setlocale", "man locale"
  while ( getword(cin, s) ) {
    tr.insert(s);
  }
  inorder(cout, tr);   // Ausgabe der sortierten Woerter
}
```

Die Klasse `tree` ist hier als Template mit "generischem" Datentyp `T` definiert und hat nur eine Komponente, einen Zeiger auf eine Struktur `node`. Diese Struktur nun trägt hier die oben genannten Komponenten und besitzt selbst eine Konstruktorfunktion.
Es gibt zwei Konstruktoren für `tree`, eine Member-Funktion `insert()` für das Einsetzen eines neuen Wortes, und eine Friend-Funktion `inorder()` zur (geordneten) Ausgabe.
In der Funktion `insert()` wird per `new node` Speicherplatz fuer ein Objekt vom Typ `node` allokiert. `new` ist C++-Schlüsselwort und liefert, aufgerufen für einen Datentyp, einen Zeiger auf diesen Typ, der auf freien Speicherplatz für "'eine Instanz" dieses Typs zeigt. Hat der Typ einen Konstruktor ohne Argument, wird dieser automatisch dabei aufgerufen. `new` erfüllt also eine ähnliche Funktion in C++ wie `malloc()` für C, aber initialisiert den Speicher darüberhinaus nach Maßgabe eines Konstruktors. (Mit `delete`, angewandt auf durch `new` gewonnenen Speicher, kann man diesen wieder freigeben.)
Im weiteren Code bedeutet die Anweisung

```
n->l = tree(n->l).insert(e);
```

folgendes: Für den linken Teilbaum `tree(n->l)` wird die Member-Funktion `insert()` mit dem Argument `e` aufgerufen, Resultat ist ein neuer `node` (mit dem alten linken Teilbaum, um einen `node` mit `e` erweitert), dieser wird `n`-Komponente des neuen linken Teilbaums. (Genaugenommen wird hier nur dann eine echte Zuweisung vorgenommen, wenn der linke Teilbaum vorher leer war, also `n->l == NULL`.)
Schließlich wird der `node this->n` zurückgegeben, `this` (als Schlüsselwort in C++) bedeutet in einer Member-Funktion einen Zeiger auf das implizite Argument, also auf das Objekt selbst, zu dem die Member-Funktion gehört. `*this` ist also das Objekt – in diesem Fall der linke Teilbaum – selbst, mit der `n`-Komponente `this->n`.

Kommandozeilen-Argumente werden in C++ wie in C behandelt. Der Zugriff auf Dateien ist einfach: Durch den Befehl `ifstream from(argv[1]);` wird ein Objekt `from` vom Typ `ifstream` (Eingabestrom) definiert, über den die Zeichen der `argv[1]` der Reihe nach mit den üblichen Mitteln wie `cin, cin.get(),...` gelesen werden können. Dabei ist `from` ein frei gewählter Name. Die Anweisung `ifstream from(argv[1]);` ist analog zu einer Deklaration mit Initialisierung wie etwa `int anzahl = 3;` zu sehen: `ifstream` ist ein Datentyp, `from` das Objekt mit frei gewähltem Namen (wie `anzahl`, das initialisiert wird auf die Datei namens `argv[1]`. Ganz analog arbeitet der Befehl `ofstream to(argv[2]);`, hier ist `to` das Objekt vom Typ `ofstream`, eine Datei, in die geschrieben wird.

Aufgabe 9.2: Testen Sie das kompilierte Programm `fileio`*, indem Sie Befehle wie*

```
fileio quelle ziel; echo $status
```

eingeben, wobei `"quelle"` *und* `"ziel"` *auch weggelassen werden bzw. durch Namen nicht existenter Dateien ersetzt werden. Die* `$status`*-Variable zeigt dann den durch* `exit` *zurückgegebenen* `int`*-Wert.*

Aufgabe 9.3: Modifizieren Sie das Programm `fileio.cc` *so, dass ein Aufruf des ausführbaren Programms* `fileio` *folgendes tut:*
`./fileio` *(ohne Parameter): Programm liest von der Tastatur (*`cin`*) und schreibt die gelesenen Daten auf den Bildschirm (nach* `cout`*).*
`./fileio quelle` *(ein Parameter): Liest aus Datei* `quelle`*, schreibt den Inhalt nach* `cout`*,*
`./fileio quelle ziel` *(zwei Parameter): Liest aus Datei* `quelle`*, schreibt den Inhalt in die Datei* `ziel`*,*
`./fileio quelle_1 ... quelle_n ziel` *(mehr als zwei Parameter): Liest der Reihe nach aus den Dateien* `quelle_1, ..., quelle_n` *und schreibt alles in die Datei* `ziel`*, die Dateien* `quelle_1, ... quelle_n` *werden also aneinandergehängt.*

Aufgabe 9.4: Benutzen Sie die Programme `fileio.cc` *und* `quicksort()` *, um ein Programm zu schreiben, das die Wörter aus einer Textdatei liest und alphabetisch geordnet mit Häufigkeitsangabe ausgibt. (Hinweis: Benutzen Sie z. B. das Programm* `cc.cc` *von Blatt 1, um Wörter zu erkennen.)*

```
// fileio.cc   Lesen und Schreiben von Dateien

#include <iostream.h>
#include <fstream.h>

void error(int i, char* s, char* t = "") {
  cerr << s << ' ' << t << '\n'; exit(i);
}


int main(int argc, char* argv[] ) {

  if (argc != 3) {
    cerr << "Falscher Aufruf, korrekter Aufruf:\n";
    error (1, argv[0], "Quelldatei Zieldatei");
  }

  ifstream from(argv[1]); // Deklaration eines ifstream namens from
                          // der aus der Datei namens argv[1] liest.
  if (!from) error(2, argv[1], "kann nicht zum Lesen geoeffnet werden.\n");

  ofstream to(argv[2]);
  if (!to) error(3, argv[2], "kann nicht zum Schreiben geoeffnet werden.\n");

  char c;

  while (from.get(c))
    to.put(c);

  from.close(); to.close(); exit(0);

  return 0;
}
```