

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 9 (4 Seiten)

Selbstreferentielle Strukturen und Klassen

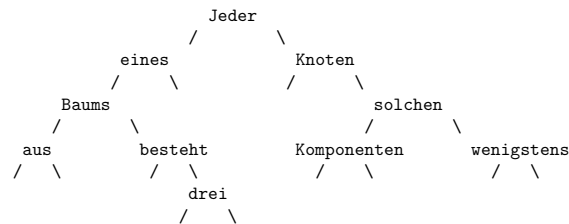
Zum Ordnen einer Folge von “zufällig” gereihten Daten eignet sich die Struktur des Binärbaums:

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.

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.

Als Erweiterung kann man den Fall betrachten, dass ein “neues” Wort mit einem bereits eingetragenen Wort identisch ist; man kann für diesen Fall einen Zähler bei den Knoten vorsehen, der ggf. erhöht wird und registriert, wie oft jedes Wort “gesehen” wurde.

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 dies leistet. Um deutsche Wörter inclusive Umlauten korrekt sortieren zu können, verwenden wir 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.

```

// btree1.cc Eine Binaerbaum-Klasse
#include <stdio.h>
#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; }
    tree(node *nn) { n = nn; }
    tree& insert(const T &e) {
        if (n == NULL) { n = new node; n->e = e; }
        else if (e==n->e) { n->c++; }
        else if (e< n->e) { tree t = tree(n->l); t.insert(e); n->l = t.n; }
        else { tree t = tree(n->r); t.insert(e); n->r = t.n; }
        return *this;
    }
    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;
    }
};

void main() {
    char s[300]; // Wort wie gelesen, inclusive Satzzeichen etc.
    char t[300]; // Wort nur mit Buchstaben
    char *p, *q;
    tree<string> tr; // Deklaration eines tree - Knotens
    setlocale(LC_ALL, "deutsch");
    while ( cin >> s ) {
        p = s; q = t;
        while (*p) { // Satzzeichen usw. ausfiltern.
            if ( isalpha(*p) )
                *q++ = *p;
            p++;
        }
        *q = '\0';
        // if (t[0]) // Nur wenn das Wort nicht "leer" ist
            tr.insert(t);
    }
    inorder(cout, tr); // Ausgabe der sortierten Woerter
}

```

Zur Entspannung ein Spielzeug: Ein Raumschiff soll auf dem Mond landen. Es hat eine bestimmte Treibstoffmenge, die es zum Bremsen verwenden soll. Möglichst soll die Landegeschwindigkeit gleich 0 m/sec sein. Jede Sekunde können Sie als Kommandant eine bestimmte Menge Treibstoff verbrauchen, um das Schiff abzubremsen, aber die Gravitationsbeschleunigung wirkt dem natürlich entgegen.

Die Zustände des Raumschiffes werden durch die Daten **v** (Geschwindigkeit im m/sec), **h** (Höhe über dem Mondboden in m), **f** (Treibstoff-Vorrat), **b** (Beschleunigung in m/sec²) gekennzeichnet, die natürlich im Sinne der objektorientierten Programmierung durch eine Klasse repräsentiert werden. Einige Member-Funktionen übernehmen die Zustandsanzeige und die Steuerung des Raumschiffes nach Ihren Anweisungen, z. B. sorgt die Member-Funktion **decel** für die Verzögerung (deceleration) bei gegebener Bremsreibstoffmenge.

Die wirklichen Abhängigkeiten von **h** und **v** von der Zeit *t* lauten $h(t) = h_0 + tv_0 + t^2b/2$; $v(t) = v_0 + tb$. **decel** rechnet also sozusagen im Sekunden-Takt. Sieger ist, wer bei minimalem Treibstoff-Verbrauch mit Geschwindigkeit 0 die Höhe 0 erreicht.

Die Anzeige liefert auch die jeweils aktuelle Bremsverzögerung in Vielfachen von *g*, der Erdbeschleunigung. Harte Bremsstöße, die etwa mit 20 *g* arbeiten, führen vermutlich zur Zerstörung des Raumschiffes und der Besatzung und sind daher tunlichst zu vermeiden.

Aufgabe 9.2: Setzen Sie im Programm den Wert B auf 0 und simulieren Sie damit ein Andock-Manöver eines Raumschiffes (Atlantis) an ein anderes (Mir).

```
// mondlandung.cc
#include <stdio.h>
#include <iostream.h>
#define B 1.62      /* m/qsec Fallbeschleunigung Mond */
#define G 9.81      /* m/qsec Fallbeschleunigung Erde */
#define GM 2*G
class state {
    double v, h, f, b; // Geschwindigkeit, Hoehe, Treibstoff-Vorrat, Beschleunigung
public:
    double height() { return h; }
    double fuel() { return f; }
    state (double vv, double hh, double ff) {
        v = vv; h = hh; f = (ff >= 0) ? ff : -ff; b = 0.0;
    }
    state decel(double ff) {
        ff = (ff >= 0) ? ff : -ff ;
        if ( ff >= f ) ff = f;
        f = f - ff; b = ff - B; h = h + v + b/2; v = v + b;
    }
}
void print() {
    if ( h > 0.0 )
        printf("%8.2lf %8.2lf %8.2lf %8.2lfg  :: ", v, h, f, b/G);
    else {
        if ( v <= -2.0 ) printf("Crash:\n");
        else printf("Landung:\n");
        printf("%8.2lf %8.2lf %8.2lf %8.2lfg  :: \n", v, h, f, b/G);
        printf("Aufprall entspricht einem Sturz aus %2.2lf m \
Hoehe auf die Erde.\n", v*v/(2*G) );
    }
}
};
main() {
    state s = state(-106.0, 533.0, 140.0); double ff;
    printf(" v h f b :: Treibstoff ?\n");
    while ( s.height() > 0.0 ) {
        if ( s.fuel() > 0.0 ) {
```

```
s.print(); cin >> ff;
    }
    else ff = 0.0;
    s.decel(ff);
}
s.print();
}
```

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.

```
// 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);
}
```

Aufgabe 9.3: 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 9.4: 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 aneinandergelängt.

Aufgabe 9.5: 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.)