Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 7 (8 Seiten)

Klassen in C++ :

In C gibt es die Möglichkeit, zusammengesetzte Typen durch Strukturen auszudrücken. Die Anweisung

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

definiert einen neuen Datentyp namens `struct punkt`, der aus zwei Komponenten `x, y` besteht.
Der Datentyp `struct punkt` wird syntaktisch genauso behandelt wie jeder andere Typ. Man kann Variable diesen Typs definieren: Eine Definition `struct punkt u;` definiert dann ein Objekt (eine Variable) `u` vom Typ `struct punkt`, bestehend aus den beiden Komponenten `u.x, u.y`, die im vorliegenden Fall vom Typ `int` sind. Eine solche zusammengesetzte Struktur kann beliebig viele Komponenten verschiedener Typen haben, die Komponenten könnten ihrerseits bereits zusammengesetzte Strukturen haben.
Durch `struct punkt *w;` wird eine Zeigervariable `w` vom Typ `struct punkt` definiert; weist man ihr vermöge `w = &u;` die Adresse von `u` zu, so greift man auf die Komponenten zu durch `(*w).x, (*w).y` (erst dereferenzieren, dann auf die Komponenten (per `.`-Operator) zugreifen. Synonym hierfür ist die kürzere Schreibweise `w->x, w->y`.
Das Klassenkonzept in C++ basiert auf dem Strukturkonzept, erlaubt aber im allgemeinen nur ganz bestimmten sogenannten *Klassenfunktionen* den Zugriff auf die einzelnen Komponenten. Auf diese Weise kann man neue Typen definieren, deren Implementationsdetails (Komponenten usw.) nicht "öffentlich" zugänglich sind (ähnlich wie die Details der Fließkomma-Zahlen dem Programmierer nicht ohne weiteres zugänglich sind, und deren Einzelheiten oft auch gar nicht interessieren). Zusammen mit dem Konzept des Überladens von Funktionen und Operatoren bekommt der Programmierer auf diese Weise die Möglichkeit, neue Typen zu schaffen, die ähnlich unproblematisch zu benutzen sind wie die elementaren, "eingebauten" Typen `char, int, float, double` usw.

```
// punkt.cc, Ein erstes Beispiel fuer eine Klassendefinition
#include <iostream.h>
#include <math.h>

class punkt {
  int x; int y;                    // geschuetzte Komponenten
public:                            // Schnittstelle zur "Aussenwelt",
                                   // in diesem Fall gegeben durch Klassenfunktionen :
  punkt();                         // Constructor-Funktion zur Definition
  punkt(int, int);                 // Constructor-Funktion zur Initialisierung
  double dist();                   // Eine Member-Funktion
  friend double arg(punkt);        // Eine Friend-Funktion
  friend ostream& operator<<(ostream&, punkt);   // Ausgabeoperator
  friend int operator>(punkt, punkt);            // Vergleichsoperator
  friend int operator<(punkt, punkt);            // Vergleichsoperator
};

punkt::punkt() {};
punkt::punkt(int xx, int yy) { x = xx; y = yy; }

double punkt::dist() {      // Distanz zum Punkt (0,0)
   return sqrt(x*x+y*y);
}

double arg(punkt z) {       // Winkel der Richtung (0,0)-->(x,y) zur x-Achse
  return atan((double)z.y/(double)z.x);
```

```
}

ostream& operator<<(ostream& os, punkt z) {
   return os << '(' << z.x << ',' << z.y << ')';
}

int operator>(punkt u, punkt v) {          // lexikographische Ordnung
  if (u.x == v.x)
    return (u.y > v.y);
  return (u.x > v.x);
}

int operator<(punkt u, punkt v) {          // lexikographische Ordnung
  if (u.x == v.x)
    return (u.y < v.y);
  return (u.x < v.x);
}
//  Ende der Klassendefinition


#include "quicksort.h"            // enthaelt quicksort- und swap-Template

int a[] = { 1, 2, 3, 4, 5 };

int s = sizeof(a)/sizeof(int);    // Anzahl der Elemente im Array a

int main() {

  punkt u;                        // Hier arbeitet punkt()

  cout << "Ausgabe: u = " << u << '\n';  // Hier arbeitet operator<< = Ausgabe

  u = punkt(1,2);                 // Hier arbeitet punkt(int,int)

  punkt v = punkt(3,4);           // Hier arbeiten beide punkt-Constructoren

  cout << "u = " << u << ", v = " << v << '\n';
  cout << "l(u) = " << u.dist() << ", l(v) = " << v.dist() << '\n';
  cout << "arg(u) = "<< arg(u) << ", arg(v) = " << arg(v) << '\n';

  if (u < v)
    cout << " u < v \n";          // Test fuer operator< : lexikographische O.
  else
    cout << " ! u < v \n";

  punkt w[s*s];                   // Array wird definiert.

  for (int i=0; i < s; i++)
    for (int j=0; j < s; j++) {
      w[i+j*s] = punkt(a[i],a[j]); // Array wird initialisiert
    }

  for (int i = 0; i < s*s; i++ ) {
```

```
      cout << w[i] << ' ';
      if ( i%s == s-1 )
        cout << '\n';                // Ausgabe des Arrays
    }

    cout << "quicksort arbeitet:\n";

    quicksort(w, s*s);

    for (int i = 0; i < s*s; i++ ) { // Ausgabe des geordneten Arrays
      cout << w[i] << ' ';
      if ( i%s == s-1 )
        cout << '\n';
    }
  }
```

Erläuterung der Klassendefinition:

Die Klasse `punkt` besteht aus Objekten `z,...` vom Typ `punkt` mit je zwei `int`-Komponenten `z.x, z.y`. Auf diese Komponenten können *nur* die speziell im `public:`-Teil der Klassendefinition deklarierten (oder definierten) *Klassenfunktionen* zugreifen. Es gibt zwei Arten von Klassenfunktionen:

- *Member-Funktionen:* Member-Funktionen zeichnen sich dadurch aus, dass sie neben den formalen Argumenten noch ein weiteres *implizites* Argument vom Klassentyp haben (in unserem Fall vom Typ `punkt`). Dessen Komponenten werden in der Funktion mit den Komponentennamen der Klasse bezeichnet (in unserem Falle mit `x,y`). Konstruktoren sind immer Member-Funktionen. Der Funktionsaufruf der Konstruktoren impliziert die Allokation dieses impliziten Arguments. Der Aufruf des ersten Konstruktors bewirkt also die Definition eines Objektes von Typ `punkt`. Er erfolgt durch die Anweisung `punkt z;` – sieht also formal genau wie eine Definition einer Variablen `z` aus. Der Aufruf des zweiten Konstruktors erlaubt die Initialisierung einer Variablen `z`. Er erfolgt durch eine Anweisung wie `z = punkt(1,2);`, wodurch den Komponenten von `z` die Werte `z.x = 1, z.y = 2` zugewiesen werden. In der Funktionsdefinition sieht man die Komponenten `x,y` des impliziten Arguments, denen die Werte der formalen Argumente `xx, yy` zugewiesen werden.
  Member-Funktionen, die nicht Konstruktoren sind, können als "Komponenten" ihrer Klassenobjekte aufgefasst werden: Wird durch `punkt z;` das Objekt (die Variable) `z` vom Typ `punkt` erklärt, so erhält man die `x,y`-Komponenten per `z.x, z.y` und analog die zugehörige Member-Funktion `dist()` durch `z.dist()`. Diese hat im vorliegenden Fall kein (explizites) formales Argument, kann im allgemeinen aber solche haben. Der Wert des Ausdrucks `z.dist()` ist der Abstand `sqrt(x*x+y*y)` $= \sqrt{x^2+y^2}$ des Punktes `z` vom Punkt `(0,0)`. Der volle Prototyp lautet `double punkt::dist()`, vor den Namen ist in der Funktionsdefinition der Klassenname gefolgt von `::` zu setzen. Der Aufruf geschieht immer bezogen auf ein `punkt z;` mit der Syntax `z.dist();`, `z` ist dann das implizite Argument, und das Resultat ist der Wert der Funktion für dieses Element.
- *Friend-Funktionen:* Friend-Funktionen tragen das Schlüsselwort `friend` vor ihrer Deklaration innerhalb der Klassendefinition. Im vorliegenden Fall handelt es sich um die Funktion `double arg(punkt)` und um Operator-Funktionen zu den Operatoren `>>,<`. Für `punkt z;` liefert `arg(z);` das Bogenmaß des Winkelargumentes von `z`. Die Auswertung von `>>` für `cout, z` geschieht wie gewohnt durch `cout >> z;` ähnlich für `>`.
  Alle Klassenfunktionen greifen auf Klassenkomponenten von `punkt z` vermittels des Punkt-Operators zu: `z.x, z.y`.
  Alle Funktionsdefinitionen können auch bereits innerhalb der Klassendefinition gegeben werden. Dies erzwingt dann, dass der Code für diese Funktionen für jeden Aufruf "inline" substituiert wird (ähnlich wie bei Präprozessor-Substitutionen), so dass also kein echter Funktionsaufruf generiert wird, was schnelleren, aber auch längeren Maschinencode erzeugt.
  Die "Inline"-Compilation kann auch für Funktionsdefinitionen außerhalb der Klassendefinition erzwungen werden, indem vor die Definition das Schlüsselwort `inline` gesetzt wird.

Aufgabe 7.1: Ändern Sie die Klassendefinition und das Programm so, dass
i) die Funktion `dist` *als Friend-Funktion implementiert ist,*
ii) die Funktion `arg` *als Member-Funktion implementiert ist,*
bei sonst gleicher Arbeitsweise.

Aufgabe 7.2: Fügen Sie eine Operatordefinition für den Operator `>>` *im Zusammenhang mit* `istream& cin` *ein, die eine bequeme Eingabe von* `punkt`*-Objekten erlaubt.*

Aufgabe 7.3: Überladen Sie den Operator `+`*, so dass für* `punkt u, v;` *die Summe* `u+v` *komponentenweise gebildet wird.*

Aufgabe 7.4: Überladen Sie den Operator `*` *in zweifacher Weise so, dass für* `int a; punkt z;` *das Produkt* `a*z` *aus den Komponenten* `a*z.x, a*z.y` *besteht (Prototyp:* `punkt operator*(int,punkt)`*) , und dass für* `punkt u, v;` *das Produkt* `u*v` *gleich* `u.x*v.x+u.y*v.y` *wird (Prototyp:* `int operator*(punkt,punkt)`*).*

Datum: Übertragung des C-Beispiels von Übungsblatt 5:

(Versehentlich wurde auf dem in der Vorlesung verteilten Blatt das Programm `date0.c` *ausgedruckt.*
Das folgende Beispiel definiert eine Klasse `datum`, die Kalenderdaten enthält. (Siehe auch das entsprechende Beispiel aus dem C-Teil des Kurses.)
Fast alle Funktionen sind inline geschrieben.
Besonderheiten: Durch Überladen der Operatoren `+` wird ermöglicht, die Summe eines Datums und einer Zahl `t` als neues Datum zu erhalten, das `t` Tage nach dem Ausgangsdatum gilt.
Die Differenz zweier Daten ergibt sich durch Überladen des Operators `-` ähnlich als `int` und gibt die Anzahl der Tage zwischen diesen Daten an.

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

#include "string-class.h"

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

// Monat mit zwei Komponenten, zweite K. ein int-Array der Laenge 2:

struct monat { string name; int n[2]; };

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}}
};

class datum {
  int  t; int m; int j;  int n; // Tag, Monat, Jahr, Nummer des Tages im Jahr

public:
  // Zwei Konstruktor-Funktionen
  datum() {};
```

```
datum(int tag, int monat, int jahr) {
  t = tag; m = monat; j = jahr;

  if ( 1 <= m && m <= 12 && 1 <= t && t <= mon_laenge() ) {
    n = t + mon[ m - 1 ].n[ schalt(j) ];  //  normalisieren.
  }
  else {
    cout << "Das Datum gibt es nicht : " << t << "." << m << "." << j <<"\n";
    exit(1);
  }
}

// friend - Klassenfunktionen
friend int schalt(int j) {
  return ( j % 4 == 0 &&  j % 100 != 0 || j % 400 == 0 );
};

friend int jahr_laenge(int j) { return 365 + schalt(j);  }

int mon_laenge() {
  return mon[m].n[ schalt(j) ] - mon[m-1].n[ schalt(j) ];
}

friend ostream& operator<<(ostream& os, datum d) {
  os << d.t << ". " << mon[d.m].name << ", " << d.j << "\n";
  return os;
}

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

// Berechne u - v
friend int operator-(datum u, datum v) {
  int n, i;
  if (u.j < v.j || (u.j == v.j && u.n < v.n ) ) return -(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() const { // nachgestelltes const, bezeichnet die Zusicherung,
  // dass das implizite Argument nicht veraendert wird
  return dat(j, n+1 );
}
```

```
  friend datum operator+(datum u, int t) {
    if (t >= 0) {
      return dat(u.j, u.n + t);
    }
    cout << "Nicht implementiert (t negativ) : " << t << "\n" ;
    exit(1);
  }

  friend istream& operator>>(istream& is, datum& d) {
    cout << "Bitte Tag Monat Jahr eingeben : ";
    int t, m, j;
    is >> t >> m >> j;
    d = datum(t,m,j);
    return is;
  }

  int jahr() { return j;}
  int tag_im_jahr() { return n;}
};

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

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

  while(1) {
    cin >> d;
    cout << "Das Datum ist : " << d ;
    cout << "Es ist der " <<
      d.tag_im_jahr() << "-te Tag im Jahr " << d.jahr()  << ".\n";
    cout << "Es ist ein " << wochentag(d) << ".\n";
    cout << "Der naechste Tag ist : " << d.inc_tag();
    cout << "wieviele Tage dazu ? "; cin >> t;
    dd = d + t;
    cout << dd;
    cout << "Unterschied nach diff : " << dd - d << "\n";
  }
}
```

Aufgabe 7.5: Überladen Sie die Operatoren `++`, `--` *derart, dass sie den Übergang zum Datum des jeweils nächsten/vorigen Tages ermöglichen.*

Mondlandung: ein ganz einfaches Beispiel einer Klassendenfinition:

Die Klasse `state` definiert den Zustand einer Mondfähre, das mit Geschwindigkeit `v` (in m/sec) in der Höhe `h` über dem Mondboden (in m) bei Beschleunigung `b` (in m/sec^2) mit einem Treibstoff-Vorrat `f` auf den Mond fällt.
Durch Bremsstöße (durch Zünden der Rakete = Treibstoffverbrauch) soll die Geschwindigkeit so weit abgebremst werden, dass die Fähre schließlich sanft auf der Mondoberfläche landet. Den Bremsstößen entgegen wirkt die Fallbeschleunigung (Mondgravitation).
Es gibt es einen Konstruktor, der die Anfangswerte setzt.
Weiter gibt es die Member-Funktionen `double height(), fuel()`, die die aktuelle Höhe und den verbliebenen Treibstoff-Vorrat anzeigen, sowie eine Member-Funktion `state decel(double ff)`, die aus einem `state` den neuen `state` berechnet, nachdem ein Bremsstoß mit der Treibstoffmenge `ff` erfolgte.
Die wirklichen Abhängigkeiten von `h` und `v` von der Zeit$t$ lauten

$$h(t) = h_0 + tv_0 + t^2 b/2; \quad v(t) = v_0 + tb.$$

`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 $g = 9.81\text{m}/(\text{sec}^2)$, 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.

```
// mondlandung.cc

#include <stdio.h>
#include <iostream.h>

#define B 1.62      /*   m/qsec Fallbeschleunigung Mond   */
#define G 9.81      /*   m/qsec Fallbeschleunigung Erde   */

class state {
  double v, h, f, b;
         // Geschwindigkeit, Hoehe, Treibstoff-Vorrat, Beschleunigung
public:

  // Einzige Konstruktor-Funktion:
  state (double vv, double hh, double ff) {
    v = vv; h = hh; f = (ff >= 0) ? ff : -ff; b = 0.0;
  }

  // Einige Member-Funktionen:
  double height() { return h; }
  double fuel()   { return f; }

  state decel(double ff) {
    ff = (ff >= 0) ? ff : -ff ; // negativer Treibstoff-Verbrauch??
    if ( ff >= f ) ff = f;      // Vorrat verbraucht!
    f = f - ff;  b = ff - B;  h = h + v + b/2;  v = v + b;
    return *this;
  }
```

```
  void print() {
    if ( h > 0.0 )
      printf("%8.2f %8.2f %8.2f %8.2fg   ::  ", v, h, f, b/G);
    else {
      if ( v <= -2.0 )  printf("Crash:\n");
      else              printf("Landung:\n");
      printf("%8.2f %8.2f %8.2f %8.2fg   ::  \n", v, h, f, b/G);
      printf("Aufprall entspricht einem Sturz aus %2.2f m \
Hoehe auf die Erde.\n", v*v/(2*G) );
    }
  }
};

int main() {
  state s = state(-106.0, 533.0, 140.0);
  double ff;
  printf("    v        h        f          b     :: Treibstoff ?\n");
  while ( s.height() > 0.0 ) {
    ff = 0.0;
    if ( s.fuel() > 0.0 ) {
      s.print();
      cin >> ff;
    }
    s.decel(ff);
  }
  s.print();
}
```

Aufgabe 7.6: Modifizieren Sie das Programm so, dass
i) mit `B = 0` *ein "Andock-Manöver" an eine (unbewegliche) Raumstation simuliert wird.*
ii) durch Verwendung von zwei `state`*-"Instanzen" und alternative Eingaben ein Rendevous-Manöver zweier sich aufeinander zu bewegender Raumschiffe ermöglicht wird.*