

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 7 (4 Seiten)

Klassen in C++ :

In C gibt es die Möglichkeit, komplexe Typen durch Strukturen auszudrücken:

```
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 `(*u).x`, `(*u).y` (erst dereferenzieren, dann auf die Komponenten (per `.-Operator`) zugreifen. Synonym hierfür ist die kürzere Schreibweise `u->x`, `u->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.

Ein Beispiel:

```
// 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" :
    punkt();                     // Konstruktor-Funktion zur Definition
    punkt(int, int);             // Konstruktor-Funktion zur Initialisierung
    double laenge();             // Eine Member-Funktion
    friend double arg(punkt);     // Eine Friend-Funktion
    friend ostream& operator<<(ostream&, punkt); // Ausgabeoperator
    friend int operator<(punkt, punkt);         // Vergleichsoperator
};
punkt::punkt() {}
punkt::punkt(int xx, int yy) {
    x = xx; y = yy;
}
double punkt::laenge() {
    return sqrt(x*x+y*y);
}
double arg(punkt z) {
    return atan(z.y/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); }
// Ende der Klassendefinition
```

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 Konstruktor impliziert die Allokation dieses impliziten Arguments. Der Aufruf des ersten Konstruktors bewirkt also die Definition eines Objektes vom 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 `laenge()` durch `z.laenge()`. Diese hat im vorliegenden Fall kein (explizites) formales Argument, kann im allgemeinen aber solche haben. Der Wert des Ausdrucks `z.laenge()` ist die Länge $\text{sqrt}(x*x+y*y) = \sqrt{x^2 + y^2}$ von `z`. Der volle Prototyp lautet `double punkt::laenge()`, 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.laenge()`; , `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` mittels 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

- die Funktion `laenge` als Friend-Funktion implementiert ist,*
 - 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.

```

#include "quicksort.h"          // enthaelt quicksort- und swap-Template
int a[] = { 1, 2, 3, 4, 5};
int s = sizeof(a)/sizeof(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-Konstrukturen
    cout << "u = " << u << ", v = " << v << '\n';
    cout << "l(u) = " << u.laenge() << ", l(v) = " << v.laenge() << '\n';
    cout << "arg(u) = " << arg(u) << ", arg(v) = " << arg(v) << '\n';
    if (u < v)
        cout << " u < v \n";    // Test fuer operator< : lexikographische 0.
    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-1);
    for (int i = 0; i < s*s; i++ ) {
        cout << w[i] << ' ';
        if ( i%s == s-1 )        // Ausgabe des geordneten Arrays
            cout << '\n';
    }
}
// Ausgabe: u = (0,32)
// u = (1,2), v = (3,4)
// l(u) = 2.23607, l(v) = 5
// arg(u) = 1.10715, arg(v) = 0.785398
// u < v
// (1,1) (2,1) (3,1) (4,1) (5,1)
// (1,2) (2,2) (3,2) (4,2) (5,2)
// (1,3) (2,3) (3,3) (4,3) (5,3)
// (1,4) (2,4) (3,4) (4,4) (5,4)
// (1,5) (2,5) (3,5) (4,5) (5,5)
// quicksort arbeitet:
// (1,1) (1,2) (1,3) (1,4) (1,5)
// (2,1) (2,2) (2,3) (2,4) (2,5)
// (3,1) (3,2) (3,3) (3,4) (3,5)
// (4,1) (4,2) (4,3) (4,4) (4,5)
// (5,1) (5,2) (5,3) (5,4) (5,5)

```

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)).*

Das folgende Beispiel definiert eine Klasse `date`, die Kalenderdaten enthält. Fast alle Funktionen sind diesmal inline geschrieben.

```
// date.cc
#include <iostream.h>
#include <iomanip.h>
#include <time.h>
#include <sys/time.h>

class date {
    int day; int month; int year;
public:
    date(){ today();}           // Konstruktor, traegt aktuelles Datum ein.
    date(int d, int m, int y){ day = d; month = m; year = y; }
    date(int d, int m) { today(); day = d; month = m; }
    date(int d) { today(); day = d; }
    friend ostream& operator<<(ostream& os, date d){
        return os<<setw(2)<<d.day<<'.'<<setw(2)<<d.month<<'.'<<d.year<<'\n';
    }
    void today();           // Member-Funktion aktuelles Datum
    // void next(); // Uebungsaufgabe: ersetzt den Wert durch das folgende Datum
};

// Initialisierung auf aktuelles Datum fuer Unix:
void date::today() {           // siehe "man gettimeofday" und "man localtime"
    struct timeval tv; struct tm *t;
    gettimeofday(&tv, 0); time_t tt = tv.tv_sec; t = localtime(&tt);
    day = t->tm_mday; month = t->tm_mon + 1; year = t->tm_year+1900;
}

main() {
    date datum;                // Ausgabe:
    cout<<datum;               // 26. 2.2001
    datum = date(9,5,2001); cout<<datum; // 9. 5.2001
    datum = date(24,12);      cout<<datum; // 24.12.2001
    datum = date(17);         cout<<datum; // 17. 2.2001
    datum.today();            cout<<datum; // 26. 2.2001
}
```

Aufgabe 7.5: Schreiben Sie die Funktion `next()`, die zu einem Datum `date d` durch den Aufruf `d.next()` das nachfolgende Datum nach `d` einträgt. Implementieren Sie den gleichen Effekt durch das Überladen des Operators `++`.

Aufgabe 7.6: Implementieren Sie `date operator+(date,int)`, so dass für `date d`, `int n` der Wert von `d+n` das Datum `n` Tage vor oder nach `d` (je nach Vorzeichen) ist.

Aufgabe 7.7: Implementieren Sie `int operator-(date,date)`, so dass für `date d`, `date dd` der Wert von `d-dd` die Anzahl der Tage zwischen `dd` und `d` ist.

Hinweis: 4.6, 4.7 kann man so bearbeiten, dass man zu jedem Datum die Anzahl der Tage, die seit einem bestimmten Datum (etwa: 1.1.1970) verflossen sind, berechnet, und z. B. diese Zahl als weitere Komponente in die Klasse `date` einbaut. Schaltjahre beachten! Für den Zeitraum seit dem 1.1.1970 kann man sich die Lösung durch das Studium der Manual-Einträge zu den C-Funktionen `gettimeofday`, `localtime` besonders einfach machen.