

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 8 (4 Seiten)

Klassen-Templates in C++ , komplexe Zahlen:

Man kann die Struktur der komplexen Zahlen etwa wie folgt als Klasse implementieren:

```
class complex {
    double r; double i;
public:
    ...
}
```

Dies impliziert, dass man auf den Komponenten-Typ `double` festgelegt ist. Will man `float` oder `int` als Komponenten, muss man eine weitere Klasse festlegen. Einfacher wird dies durch ein *Klassen-Template*:

```
template <class REAL>
class complex {
    REAL r; REAL i;
public:
    ...
}
```

Dann hat man in `complex<double>`, `complex<float>`, `complex<int>` usw. verschiedene `complex`-Typen zur Verfügung, die als Real-Bestandteile jeden adäquaten Typ erlauben. Allerdings gibt es für manche Situationen Probleme. Wenn zum Beispiel die Betragsfunktion `REAL abs(complex z)` implementiert ist, wird im Fall `complex<int>` hier `REAL` durch `int` ersetzt, was ziemlich unzuweckmäßig ist.

Templates erlauben aber mehrere formale Typbezeichner als Argumente:

```
template <class REAL, class FLOAT>
class complex {
    REAL r; REAL i;
public:
    ...
    friend FLOAT abs(const complex x) {
        return sqrt(x.r*x.r+x.i*x.i);
    }
    ...
}
```

Die Typbezeichnung lautet dann z. B. `complex<int,double>` usw., und die Funktion `abs` liefert auch für den komplexen Typ mit `int`-Koeffizienten den Typ `double` zurück.

Die Datei `complex.cc` bietet eine Implementation, die hier im einzelnen kommentiert wird:

```
// complex.cc
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#define PI 3.14159265358979323846
template <class REAL, class FLOAT>
class complex {
    REAL r; REAL i;
public:
    complex() {} ;
    complex(REAL re, REAL im = 0) { r=re; i=im;}
```

Die Initialisierung `im = 0` bedeutet, dass für `complex<...> v; double u = 1;` eine Initialisierung mit nur einer Variablen `v = complex(u);` zulässig ist und die gleiche Bedeutung wie `v = complex(u,0);` hat. Die Komponente `im` wird also "per Default" auf 0 initialisiert.

Es folgen die arithmetischen Operationen inklusive Zuweisungs-Operatoren. Hier werden aus Effizienzgründen jeweils im Fall von komplexen Argumenten Referenzen auf die Klassen übergeben. D. h., für jeden Aufruf sind statt der Argumente (2 REAL-Objekte) nur jeweils eine Adresse an die Funktion zu übergeben. Dadurch, dass man die Argumente als `const` deklariert, wird verhindert, dass diese selbst versehentlich modifiziert werden, was bei beliebigen Referenzen natürlich denkbar wäre.

```
complex operator+=(const complex& y) {    // Member-Funktion
    r += y.r; i += y.i; }
complex operator+=(REAL y) { r += y; }
```

Es ist zweckmäßig, die Fälle, dass eines der Argumente vom Typ `REAL` ist, gesondert zu betrachten und die Operatoren oder Funktionen entsprechend zu überladen, wie im letzten Beispiel. Andernfalls würde etwa bei einer Zuweisung von Typ

```
double u=1, complex<double,double> v; v += u;
```

implizit immer noch die zweite Konstruktor-Funktion aufgerufen, die für die Umwandlung `double --> complex<...,>` zuständig ist.

Die Zuweisungs-Operatoren `+=`, `-=` usw. werden hier sämtlich als *Member-Funktionen* implementiert. Dies ist natürlich, da in `v += u` ja der linke Operand `v` modifiziert wird. Er erscheint also in der Implementation als das implizite Argument.

```
friend complex operator+(const complex& x, const complex& y) {
    return complex(x.r+y.r, x.i+y.i); }
friend complex operator+(const complex& x, REAL y) {
    return complex(x.r+y, x.i); }
friend complex operator+(REAL x, const complex& y) {
    return complex(x+y.r, y.i); }
```

Die binären Operatoren sind als *friend-Funktionen* implementiert, da bei ihren Anwendungen, etwa in `w = u + v;` immer zwei gleichberechtigte Operanden `u,v` auftreten, die in der Regel auch nicht modifiziert werden.

```
complex operator--(const complex& y) {    // Member-Funktion
    r -= y.r; i -= y.i; }
complex operator--(REAL y) { r -= y; }
friend complex operator-(const complex& x, const complex& y) {
    return complex(x.r-y.r, x.i-y.i); }
friend complex operator-(const complex& x, REAL y) {
    return complex(x.r-y, x.i); }
friend complex operator-(REAL x, const complex& y) {
    return complex(x-y.r, -y.i); }
friend complex operator-(const complex& x) {
    return complex(-x.r, -x.i); }
complex operator*=(const complex& y) {
    REAL rr = r*y.r-i*y.i; i = r*y.i+i*y.r; r = rr; }
complex operator*=(REAL y) { r *= y; i *= y; }
```

Bei den binären multiplikativen Funktionen sind Effizienzbetrachtungen noch wichtiger:

```
friend complex operator*(const complex& x, const complex& y) {
    return complex(x.r*y.r-x.i*y.i, x.r*y.i+x.i*y.r); }
friend complex operator*(const complex& x, REAL y) {
    return complex(x.r*y, x.i*y); }
friend complex operator*(REAL x, const complex& y) {
    return complex(x*y.r, x*y.i); }
```

In der ersten Funktion werden vier REAL-Multiplikationen ausgeführt, in der zweiten und dritten jeweils nur zwei. Das REAL-Argument kann man direkt übergeben, da durch Verwendung einer Referenz jedenfalls für elementare Typen wie double, float und int keine Zeitersparnis erzielt würde. Allerdings ergäbe sich ein Unterschied, wenn man etwa als REAL eine selbstentworfenen Klasse rationaler Zahlen (bestehend aus den beiden int-Komponenten "Zähler" und "Nenner") verwenden würde.

Die weiteren Funktionen werden ähnlich behandelt.

```
complex operator/=(const complex& y) {
    REAL a = y.r*y.r + y.i*y.i; REAL rr = (r*y.r+i*y.i)/a;
    i = (i*y.r-r*y.i)/a; r = rr; }
complex operator/=(REAL y) { r /= y; i /= y; }
friend complex operator/(const complex& x, const complex& y) {
    REAL a = y.r*y.r + y.i*y.i;
    return complex((x.r*y.r+x.i*y.i)/a, (x.i*y.r-x.r*y.i)/a); }
friend complex operator/(const complex& x, REAL y) {
    return complex(x.r/y, x.i/y); }
friend complex operator/(REAL x, const complex& y) {
    REAL a = y.r*y.r + y.i*y.i; x/=a;
    return complex(x*y.r, -x*y.i); }
```

Hier die FLOAT-wertigen Funktionen `abs` (Absolutbetrag), `arg` (Argument = Winkel zwischen komplexer Zahl als Vektor und der reellen Achse im Bogenmaß), `warg` (Argument im Winkelmaß) und `sqrt` (Quadratwurzel).

```
friend FLOAT abs(const complex& x) {
    return sqrt(x.r*x.r+x.i*x.i); }
friend FLOAT arg(const complex& x) {
    if (x.r == 0) return (x.i == 0) ? 0 : ( x.i > 0 ? 0.5*PI : 1.5*PI);
    if (x.r < 0) return atan(x.i/x.r) + PI;
    return (x.i >= 0) ? atan(x.i/x.r) : atan(x.i/x.r) + 2*PI ;
}
friend FLOAT warg(const complex& x) { return arg(x)/PI*180; }
friend complex sqrt(const complex& x) {
    FLOAT a = arg(x)/2, b = sqrt(abs(x));
    return complex( REAL(cos(a)*b), REAL(sin(a)*b)); }
```

Aufgabe 8.1: Überladen Sie die Funktionen `double pow(double,double)` aus der Standard-Bibliothek in mathematisch sinnvoller Weise zu `complex pow(complex,complex)`. Verfahren Sie ähnlich mit anderen Funktionen der Standardbibliothek wie `exp`, `sin`, `cos`, `log` usw. (cf. man `exp`.)

Es folgen die logischen Operatoren und der Ausgabeoperator:

```
friend int operator==(const complex& x, const complex& y) {
    return (x.r==y.r && x.i==y.i); }
friend int operator==(const complex& x, REAL y) {
    return (x.i == 0 && x.r==y); }
friend int operator==(REAL x, const complex& y) {
```

```

    return (y.i == 0 && x==y.r); }
friend int operator!=(const complex& x, const complex& y) {
    return (x.r!=y.r || x.i!=y.i); }
friend int operator!=(const complex& x, REAL y) {
    return (x.r!=y || x.i!=0); }
friend int operator!=(REAL x, const complex& y) {
    return (x!=y.r || y.i!=0); }
friend ostream& operator<<(ostream& os, const complex& z) {
    os<<setw(6)<<setprecision(3)<< z.r;
    os << ( z.i>=0 ? " + ":" - ");
    os<<setw(6)<<setprecision(3)<< ( z.i>=0 ? z.i : -z.i ) <<'i';
    return os; }
};

```

Aufgabe 8.2: Überladen Sie auch den Eingabeoperator << für die Klassen `complex<...>` und schreiben Sie ein Testprogramm, das es erlaubt, die von Ihnen implementierten Funktionen interaktiv zu testen.

Man sollte die am häufigsten verwendeten Typen per `typedef` abkürzen:

```

typedef complex<float,double>  complex_f;
typedef complex<double,double> complex_d;
typedef complex<int,double>    complex_i;

```

Bemerkung: Ein Cast zwischen zwei durch das gleiche Template definierten Typen ist nicht möglich.

```

main() {
    complex_d u,v;
    v = u = complex_d(1,1)/sqrt(2);
    cout << v << ' ' << u << '\n';
    v = u = sqrt(u);
    for (int i=0; i<8; i++) {
        cout <<u<<" hoch "<<i+1<<" : "<<v<< " warg = "<<setw(4)<<warg(v)<<'\\n';
        v *= u;
    }
    v = u = 1/u;
    for (int i=0; i<8; i++) {
        cout <<u<<" hoch "<<i+1<<" : "<<v<< " warg = "<<setw(4)<<warg(v)<<'\\n';
        v *= u;
    }
}
/* Ausgabe:
0.707 + 0.707i 0.707 + 0.707i
0.707 + 0.707i hoch 1 : 0.707 + 0.707i warg = 45
0.707 + 0.707i hoch 2 : 0 + 1i warg = 90
0.707 + 0.707i hoch 3 : -0.707 + 0.707i warg = 135
0.707 + 0.707i hoch 4 : -1 + 0i warg = 180
usw. */

```

Aufgabe 8.3: Definieren Sie eine Ordnung < auf den komplexen Zahlen und sortieren Sie ein Array von komplexen Zahlen mittels `quicksort` wie in Übungsblatt 7.

Aufgabe 8.4: Schreiben Sie eine Implementation einer Klasse `class rat { long z; long n; }` von rationalen Zahlen mit Zähler `z`, Nenner `n`. Verwenden Sie diese Klasse zum Studium einer Klasse `complex_r`; von rationalen komplexen Zahlen.

Hinweis: Denken Sie daran, die `REAL`-Argumente in der Klasse `complex` für diesen Fall zu Referenzen zu machen. Ist dies ohne Änderung des Klassen-templates `complex` möglich?