

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 8 (8 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 unzweckmäßig 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; w = u - v; usw.` immer zwei gleichberechtigte Operanden `u,v` auftreten, die in der Regel auch nicht modifiziert werden.

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.

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

Wir betrachten noch die logischen Operatoren und den 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; }
};

```

Für den Logarithmus und zum Potenzieren komplexer Zahlen benötigt man die Eulersche Formel : $e^{ix} = \cos(x) + i \sin(x)$, die sich aus den Potenzreihen für Exponential-Funktion, Sinus- und Cosinus-Funktion ergibt:

$$\begin{aligned}
 e^{ix} &= \exp(ix) = 1 + \frac{(ix)^1}{1!} + \frac{(ix)^2}{2!} + \frac{(ix)^3}{3!} + \frac{(ix)^4}{4!} + \frac{(ix)^5}{5!} + \dots \\
 \cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \\
 i \sin(x) &= \frac{ix^1}{1!} - \frac{ix^3}{3!} + \frac{ix^5}{5!} - \dots
 \end{aligned}$$

Für die **Berechnung des komplexen Logarithmus** verwenden wir die Beziehung $\log(z z') = \log(z) + \log(z')$ (die mit der Potenzreihenentwicklung für die Logarithmusfunktion gezeigt wird), angewandt auf die Zerlegung $z = |z| \cdot \frac{z}{|z|} = |z| \cdot e^{i \arg(z)}$ (die letzte Beziehung folgt aus der Eulerschen Formel und aus $\arg(\cos(x) + i \sin(x)) = \arctan(\sin(x)/\cos(x)) = x$). Logarithmieren ergibt damit $\log(z) = \log|z| + i \arg(z)$. Diese Formel begründet

```

friend complex log(complex z) {
    return complex(log(abs(z)),arg(z));
}

```

Zum Berechnen **komplexer Potenzen** schreiben wir für zwei komplexe Zahlen u, v :

$$u^v = e^{v \log u} = e^{\operatorname{Re}(v \log u) + i \operatorname{Im}(v \log u)} = \underbrace{e^{\operatorname{Re}(v \log u)}}_{\text{reelles Potenzieren}} \cdot \underbrace{e^{i \operatorname{Im}(v \log u)}}_{\text{Eulersche Formel}}.$$

Als Programm:

```

friend complex operator^(complex u, complex v) {
    complex z = v*log(u);
    REAL a = exp(z.r);
    return complex(a*cos(z.i), a*sin(z.i));
}

```

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.

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?

```
// complex.cc

#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#define PI 3.14159265358979323846
#define E 2.7182818284590452354      /* e */

template <class REAL, class FLOAT>
class complex {
    REAL r; REAL i;
public:
    // complex() {};
    complex(REAL re=0, REAL im = 0) { r=re; i=im;}
    // Addition:
    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);    }
    // Subtraktion:
    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);    }
    // Multiplikation:
    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; }
    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);    }
    // Division:
    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); }
friend complex inv(complex z) {
    REAL a = z.r*z.r + z.i*z.i;
    return complex(z.r/a,-z.i/a);
}

// Absolutbetrag, Argument:
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;
    // x.r > 0 :
    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; }

// Wurzel:
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)); }

// Potenzieren: Achtung: Nach der Prioritätenliste fuer
// Operatoren hat der Operator ^ nicht die Prioritaet, die man fuer das
// Potenzieren erwartet, man muss also klammern: (u^v) * w

friend complex log(complex z) {
    return complex(log(abs(z)),arg(z));
}

friend complex operator^(complex u, complex v) {
    complex z = v*log(u);
    REAL a = exp(z.r);
    return complex(a*cos(z.i), a*sin(z.i));
}

friend complex operator^(complex z, REAL d) {
    return z^complex(d);
}
friend complex operator^(REAL u, complex z) {
    return complex(u)^z;
}

/* alternativ:
friend complex operator^(REAL u, complex z) {
    REAL r = pow(u,z.r); REAL l = log(u);
    z.i = l*fmod (z.i, 2*PI);
}

```

```

    return complex( r*cos(z.i), r*sin(z.i));
}

REAL warg(complex z) {
    return arg(z)/PI*180;
}

// Boolesche Operatoren:
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); }
// Ausgabeoperator:
friend ostream& operator<<(ostream& os, const complex z) {
    REAL rr, ii;
    rr = z.r; ii = z.i;
    if ( abs(rr) < 1.0e-15) rr=0;
    if ( abs(ii) < 1.0e-15) ii=0;
    os<<setw(6)<<setprecision(2)<< rr;
    os << ( ii>=0 ? " + " : " - ");
    os<<setw(6)<<setprecision(2)<< ( ii>=0 ? ii : -ii ) << 'i';
    return os; }
};

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

main() {
    complex_ld u,v;
    v = u = sqrt(sqrt(complex_ld(0,1))); // 4-te Wurzel aus i
    cout << u << '\n';
    cout << "\n";

    for (int i=0; i<16; i++) {
        cout << u << " hoch "<<setw(2)<< i+1 << " : "<< v << "    warg = ";
        cout << setw(6)<<setprecision(4) << warg(v) << (char)186 << '\n';
        v *= u;
    }
    cout << "\n";
    v = u = 1/u;
    for (int i=0; i<16; i++) {
        cout << u << " hoch "<<setw(2)<< i+1 << " : "<< v << "    warg = ";
        cout <<setw(6)<<setprecision(4) << warg(v) << (char)186 << '\n';
    }
}

```

```

    v *= u;
}
cout << "\n";
cout << " e ^ (2 pi i) = " << ( complex_ld(E,0)^complex_ld(0, 2*PI) ) << '\n';
cout << " e ^ (pi i) = " << ( E^complex_ld(0, PI) ) << '\n';
cout << " e ^ (pi/2 i) = " << ( E^complex_ld(0, PI)/complex_ld(2) ) << '\n';
cout << " e ^ (pi/4 i) = " << ( E^complex_ld(0, PI)/complex_ld(4) ) << '\n';
cout << " e ^ (pi/8 i) = " << ( E^complex_ld(0, PI)/complex_ld(8) ) << '\n';
cout << " i ^ i = " << ( complex_ld(0,1)^complex_ld(0,1) ) << '\n';
cout << " e ^ (-pi/2) = " << ( complex_ld(E,0)^complex_ld(-PI/2,0) ) << '\n';
cout << "-1 ^ 0.5 = " << ( complex_ld(-1,0)^complex_ld(.5,0) ) << '\n';
cout << "-2i ^ 0.5 = " << ( complex_ld(0,-2)^complex_ld(.5,0) ) << '\n';
cout << " 2i ^ 0.5 = " << ( complex_ld(0,2)^complex_ld(.5,0) ) << '\n';

}

```

Hier ein Template, das die Bitkodierung der elementaren Datentypen für Linux-basierende Intel-Rechner ausgibt.

Die Funktion `bits()` kann jeden Typ `T` als zweites Argument `x` aufnehmen, dessen im Programm tatsächliche Grösse wird zur Compilezeit durch den `sizeof(x)`-Aufruf festgestellt.

```

// bits.cc
// Darstellung der Bit-Codierung der verschiedenen elementaren Datentypen
// stark abhaengig von der Rechnerarchitektur!!

#include <iostream.h>

template <class T>
ostream& bits(ostream &os, const T& x) {
    char *p = (char *) &x;          // Wandle Adresse von x in char-Adresse
    int k = sizeof(x);
    k = (k==12)?k-2:k;    // Fuer long double werden nur 10 Byte verwendet !
    // Die letzte Zeile sollte bei allgemeinerer Verwendung auskommentiert werden
    for (int i = k-1; i >= 0; i--) {    // Durchlaufen der Bytes von x
        for (int c = 128; c > 0; c >>= 1) // ein Byte bitweise ausgeben
            os << ( p[i] & c ? 1 : 0 );
        os << " ";                    // Trennung der Bytes
    }
    return os << "\n";
}

int main() {
    char c; int i; long long ll;
    float f; double d; long double ld;
    // Ausgabe der Groessen
    cout << "sizeof(char)      = " << sizeof(char)      << "\n";
    cout << "sizeof(int)       = " << sizeof(int)       << "\n";
    cout << "sizeof(long long) = " << sizeof(long long)  << "\n";
    cout << "sizeof(float)     = " << sizeof(float)     << "\n";
    cout << "sizeof(double)    = " << sizeof(double)    << "\n";
    cout << "sizeof(long double) = " << sizeof(long double) << "\n";
    while (1) {
        cout << " : "; cin >> ld;
    }
}

```

Die “Floating Point Unit” hat eine 80-Bit-Architektur, daher die hohe Präzision bei `long double`. Ganze Zahlen werden binär dargestellt:
Für jedes $n \in \mathbb{N}$ gibt es $\epsilon_i \in \{0, 1\}, i = 0, \dots, k-1$, so dass

Reelle nicht negative Zahlen $r \in \mathbb{R}_{\geq 0}$ können geschrieben werden als

[illegible]