TECHNISCHE UNIVERSITÄT DRESDEN

PROCESS CONTROL SYSTEMS
PROCESS SYSTEMS ENGINEERING

DRESDEN concept

Vorlesung Mikrorechentechnik 2

Einführung C++

SS 2020

Übersicht

Unser erstes C++ Programm : Rechnen mit komplexen Zahlen

Kompetenzen

- Wie definiere ich in C++ eine neue Klasse?
  - Schlüsselwort class,
  - Was steht in Header- und Quelldatei einer Klasse ?
- Wie kann ich den neuen Datentyp nahtlos in meinem Programm nutzen?
  - Anlegen und überladen von Methoden und Operatoren
- Wie implementiere ich Methoden einer Klasse?
  - Namensraumkonzept
  - Elementfunktionen
  - Nichtelementfunktionen

Aufgabe: Berechne Frequenzgang eines Filters

Frequenzgang PT1: $F(j\,\omega) = Kp / (1 + j\,\omega\,T)$

- j und F(j w) sind komplexe Zahlen, C++ kennt keine komplexen Zahlen

Offene Frage: Welche Attribute und Schnittstellen benötigt eine Klasse

- $a,b \in \Re$ und $i^2=-1$
- $c = a + b\,i;$

Arithmetik

- $c_1+c_2 = (a_1+a_2) + (b_1+b_2)i$
- $c_1-c_2 = (a_1-a_2) + (b_1-b_2)i$
- $c_1*c_2 = (a_1a_2-b_1b_2) + (a_1b_2+b_1a_2)i$
- $c_1/c_2 = (a_1a_2+b_1b_2)/(a_2^2+ b_2^2)+((a_2b_1-a_1b_2) /(a_2^2+ b_2^2))i$

Behaviour Driven Development

Bevor wir anfangen zu Programmieren:

- **Schnittstellen** definieren
- **Verhalten** spezifizieren

Wie wollen wir neue Objekte anlegen?

- mit Real- und Imaginärteil
- wenn nicht angeben, dann auf 0 initalisiert

Welche Interna wollen wir miteilen?

- Lesender Zugriff auf Realteil, Imaginärteil

Welches Verhalten erwarten wir?

- Arithmetische Grundoperationen ?
  - Complex ? Complex
  - double ? Complex; Complex ? double

Weitere Schnittstellen

- Ausgabe auf Console: `re, im`

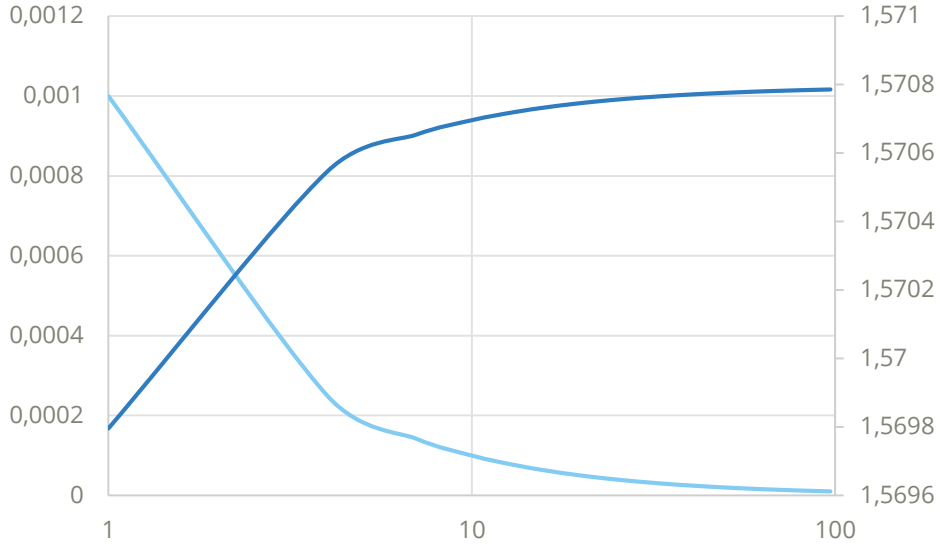
Schnittstellendefinition: Was erwarten wir ?

```
// given
      Complex c(2,1), j(0,1), r, d(1);
// when                 // then
                              r == (0, 0)
                              d == (1, 0)
      r = j * j               r == e(-1, 0)
      r = c + c               r == e( 4, 2)
      r = c - j               r == e( 2, 2)
      r = c * j               r == e(-1, 2)
      r = c * c               r == e( 3, 4)
      r = c * 2               r == e( 4, 2)
      r = 2 * c               r == e( 4, 2)
      r = c / j               r == e( 1,-2)
      r = c / c               r == e( 1, 0)
```

```
#include "Complex.h"
#include <iostream>
#include <string>
void tap(int n, const std::string& msg, Complex r, bool result) {
	std::cout << (result?"ok ":"not ok ") << n << " - " << msg << ", got " << r << std::endl;
}
int main() {  int n(0);
	// given
	Complex c(2,1), j(0,1), r;
	// when                                          // then
	r = j* j;  tap(n++, "j * j == [-1  0]", r, r.Re() == -1 && r.Im() == 0);
	r = c + c; tap(n++, "c + c == [ 4, 2]", r, r.Re() ==  4 && r.Im() == 2);
	r = c + j; tap(n++, "c - j == [ 2, 2]", r, r.Re() ==  2 && r.Im() == 2);
	r = c * j; tap(n++, "c * j == [-1, 2]", r, r.Re() == -1 && r.Im() == 2);
	r = c * c; tap(n++, "c * c == [ 3, 4]", r, r.Re() ==  3 && r.Im() == 4);
	// …
}
```

Aufgabe: Frequenzgangs eines Filter berechnen

```
#include <iostream>
#include "Complex.h"
Complex PT1(double w, double Kp, double T1) {
	static Complex j(0,1);
	return Kp / ( 1 + j * w * T1 );
}
int main() {
	double Kp = 1, T1 = 1000,  w;
	for (w=1; w<100; w+=3) {
		std::cout << w << ',' << PT1(w,Kp,T1) << std::endl;
	}
	return 0;
}
```



Complex ist kein integrierter Datentyp von C++ (wie double, int, ...)

Wir ergänzen heute C++ so, dass es mit komplexen Zahlen rechnen kann !

FrequencyResponse.cpp – bekanntes von C

```
#include <iostream>
#include "Complex.h"
Complex PT1(double w, double Kp, double T1) {
      static Complex j(0,1);
      return Kp / ( 1 + j * w * T1 );
}
int main() {
      double Kp = 1, T1 = 1000,  w;
      for (w=1; w<100; w+=3) {
            std::cout << w << ',' << PT1(w,Kp,T1) << std::endl;
      }
      return 0;
}
```

Deklaration vordefinierter Systemfunktionen

Funktion mit Kopf und Körper mit Variablen und Code

Einstiegspunkt **main**

Zwei erste C++ Konzepte

```
#include <iostream>
#include "Complex.h"
Complex PT1(double w, double Kp, double T1) {
      static Complex j(0,1);
      return Kp / ( 1 + j * w * T1 );
}
int main() {
      double Kp = 1, T1 = 1000,  w;
      for (w=1; w<100; w+=3) {
            std::cout << w << ',' << PT1(w,Kp,T1) << std::endl;
      }
      return 0;
}
```

C++ Konzept:

Namespace::

C++ Konzept:
Strom und
Ausgabeoperator <<

1. C++ Konzept Namensraum

Problem C: Ein Name einer Methode kann nur einmal verwendet werden → Namenskonflikte

Lösungsansatz C++: Strukturierung durch hierarchische Namensräume: „Urbas aus Dresden“

- Spezifikation eines Symbols aus einem Namensraums durch Angabe des vollständigen Namen mit Namensraumoperator :: namespace::methode

Komfortfunktion Vorauswahl von Namensräumen

```
using namespace std;
```

Standard Bibliotheksfunktionen

- Namensraum std

2. C++ Konzept Strom

Ein Strom verarbeitet Sequenzen von Zeichen

- Ein/Ausgabe auf Konsole: `std::cout`, `std::cin`, `std::cerr`
- Bidirektional (Datei, Zeichenkette)

Ausgabeoperator **<<**

- Delegiert Aufbereitung der Information an Objekt
- Verkettung:

```
std::cout << a << b << c << std::endl;
```

Spezielle Symbole

- **endl**: Zeilenvorschub

Komplexe Zahlen

Definition

- $c = a + b\,i;\ \ a,b \in \Re$ und $i^2=-1$

Grundlegende Rechenoperationen

- $c_1+c_2 = (a_1+a_2) + (b_1+b_2)\,i$
- $c_1-c_2 = (a_1-a_2) + (b_1-b_2)\,i$
- $c_1*c_2 = (a_1a_2-b_1b_2) + (a_1b_2+b_1a_2)\,i$
- $c_1/c_2 = (a_1a_2+b_1b_2)/(a_2^2+ b_2^2) + ((a_2b_1-a_1b_2) /(a_2^2+ b_2^2))\,i$

```
class Complex {
  private:
  double real, imag;
  public:
  Complex(double r = 0.0, double i = 0.0);
  double Re();
  double Im();
  Complex& operator+=(const Complex& rhs);
}
Complex& Complex::operator+=(const Complex& rhs) {
  real+=rhs.real; imag+=rhs.imag;
  return *this;
}
```

Abbildung in eine C++ Klasse

Abstrakter Datentyp mit Attributen a,b $\in \Re$

Operatoren +, -, *, /

Auswahl Real und Imaginärteil

Ausgabeoperator << auf Strom

Initialisieren, Kopieren, ...

Header: Complex.h

- Deklaration der Klasse
- Methoden und Attribute
- Implementierung von Inline-Methoden

Implementierung: Complex.cpp

- Implementierung der Klassenspezifischen Algorithmen

Complex.h

C++ Konzept:

Klasse

```
class Complex {
  private:
  double real, imag;
  public:
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) { };
  double Re();
  double Im();
  Complex& operator+=(const Complex& rhs);
}
```

Complex.h

```
class Complex {
  private:
  double real, imag;
  public:
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) { };
  double Re();
  double Im();
  Complex& operator+=(const Complex& rhs);
}
```

C++ Konzept:

Klasse

C++ Konzept:

Zugriffsmoderator

Complex.h

```
class Complex {
  private:
  double real, imag;
  public:
  Complex(double r = 0.0, double i = 0.0);
  double Re();
  double Im();
  Complex& operator+=(const Complex& rhs);
}
```

C++ Konzept: Klasse

C++ Konzept: Zugriffsmoderator

C++ Konzept: Konstruktor

Complex.cpp - Gerüst

```
#include "Complex.h"

Complex::Complex(double r, double i) : real(r), imag(i)  {
std::cerr << "DBG: Complex(" << r << "," << i << „)" << std::endl;
}

double Complex::Re() { return real; }
double Complex::Im() { return imag; }

Complex& Complex::operator+=(const Complex& rhs) {
real+=rhs.real; imag+=rhs.imag;
return *this;
}
```

Einbinden der Definition

Methode mit Name der Klasse ist Konstruktor

Namensraum der Klasse

OOD: Attribute, Zugriffsmethoden, Konstruktoren, Operatoren

Private Attribute

- `double real;`
- `double imag;`

Öffentliche lesende Zugriffsmethoden

- `double Re();`
- `double Im();`

Öffentlicher Konstruktor mit Defaultwerten

- `Complex(double r = 0.0, double i = 0.0);`

Öffentliche Methoden, die das Objekt manipulieren

- `Complex& Complex::operator+=(const Complex& rhs);`

```
class Complex {
private:
  double real, imag;
public:
// Constructor
  Complex(double r = 0.0, double i = 0.0);
// getter for attributes
  double Re() const;
  double Im() const;
// Operations
  Complex& operator+=(const Complex& rhs);
  Complex& operator-=(const Complex& rhs);
  Complex& operator*=(const Complex& rhs);
  Complex& operator/=(const Complex& rhs);
};
```

Ausgabe von Klassen auf Strom

Wir wollen schreiben können:

```
Complex c(2,3);
std::cout << c << std::endl;
```

Wir müssen eine Methode `operator<<` für den Datentyp `Complex` definieren

```
inline std::ostream& operator<<(std::ostream& lhs,
const Complex& rhs) {
      return lhs << rhs.Re() << "," << rhs.Im();
}
```

C++ Konzept: operator

Häufig Ausdrucksmuster der Art:

- ***Ergebnis = LinkerOperand Operator RechterOperand;***
- Arithmetik, Logik, …

Lösungsansatz in C oder Java:

- ***Ergebnis = Methode(Operand1, Operand2)***
- Methodennamen beginnen mit [_A-Za-z]

C++: Überschreiben von Operatoren

- ***Ergebnis = operatorXX(Operand1, Operand2)***
- XX ist das symbolische Kürzel des Operators

complex.h - Entwurf operator<<

Der Ausgabeoperator genügt dem Muster

- LinkerOp Operator RechterOp
- *ostrom* << *objekt*

→ `operator<<(ostream &lhs, Complex &rhs)`

Was soll rückgegeben werden? rhs? lhs?

Lösung aus Forderung nach Verkettung

- ***strom << a << b << c << endl;***
- C++ löst von links nach rechts auf
- Schritt 1: `(((strom << a) << b) << c) << endl;`
- Schritt 2: `(( ergebnis? << b) << c) << endl;`
- Schritt 3: `( ergebnis? << c) << endl;`

! Konvention: **`operator<<`** gibt Strom zurück

- `ostream& operator<<(ostream &lhs, Complex &rhs)`

Überladen von Operatoren in C++

Fast alle Operatoren sind überladbar:

```
+       -       *       /       %       ^       &       |
~       !       ,       =       <       >       <=      >=
++      --      <<      >>      ==      !=      &&      ||
+=      -=      /=      %=      ^=      & =     |=      *=
<<=     >>=     [ ]     ( )     ->      ->*     new     delete
```

Nicht überladen werden können

:: .* . ?:

Es gibt keine neuen Operatoren

Regeln zum Überladen von Operatoren (1/2)

Ein überladener Operator muss mindestens einen Operanden vom Typ **Klasse** besitzen

Priorität, Assoziativität und Anzahl der Operanden eines Operators können nicht verändert werden

- Beispiel: `A == B + C;`
- Resultiert, unabhängig vom Typ von A,B,C und den ggf. überladenen Operatoren immer in: `operator==(A, operator+(B, C));`

Regeln zum überladen von Operatoren (2/2)

Kurzschlusseigenschaft von logischem und (&&), logischem oder (||) bleiben nicht erhalten

- Es werden immer beide Operanden ausgewertet
- Reihenfolge der Auswertung ist nicht festgelegt

Reihenfolge der Auswertung für Kommaoperator ist nicht definiert

Operator als Elementfunktion einer Klasse

Überladene Operatoren dürfen sowohl als herkömmliche Nichtelementfunktion als auch als Elementfunktion definiert werden.

Beispiel: Binärer Operator += : `c += j`

Definition als Nichtelementfunktion (nicht empfohlen):

```
Complex& operator+=(const Complex& lhs, const Complex& rhs)
```

Definition als Elementfunktion:

```
Complex& Complex::operator+=(const Complex& rhs)
```

Elementfunktion vs Nichtelementfunktion

Definition als Nichtelementfunktion:

```
Complex& operator+=(const Complex& lhs, Const complex& rhs)
```

Definition als Elementfunktion:

```
Complex& Complex::operator+=(const Complex& rhs)
```

Achtung: Bei einer Definition als Elementfunktion ist der erste Operand das Objekt, repräsentiert durch den impliziten Zeiger **this.**

Der implizite Zeiger this

Jede Elementfunktion hat einen impliziten Parameter: **this**
this ist ein Zeiger auf ein Objekt des Klassentyps.
this ist an das Objekt gebunden, für das die Elementfunktion aufgerufen wurde.
this wird immer dann benötigt, wenn wir uns auf das **Objekt als Ganzes** beziehen wollen.

Beispiel complex

Per Definition muss += eine Referenz (= ein Stellvertreter, gekennzeichnet durch das &) auf das Objekt zurückgeben

```
// complex += complex
Complex& Complex::operator+=(const Complex& rhs) {
      real+=rhs.real; imag+=rhs.imag;
      return *this;
}
```

Heuristiken für Operatorüberladung (1/4)

Der Compiler überlädt automatisch:

- Zuweisungsoperator (=): Elementweise Zuweisung
- Addressierungs- (&) und Kommaoperator (,)

Wir überladen üblicherweise

Arithmetische Operatoren (+, -, *, /, ...)

- gemäß Häufigkeit und eindeutiger Interpretierbarkeit

Zuweisungsoperatoren (+=, -=, ...)

- Günstig, wenn entsprechender arithmetischer Operator definiert ist

Heuristiken zur Operatorüberladung (2/4)

Gleichheits- und relationale Operatoren

- Wenn Schlüssel in assoziativem Container, dann mindestens <
- Wenn Eintrag in sequentiellem Container, dann < und == für Algorithmen wie find (==) und sort (<)
- Wenn == dann auch !=, wg. Erwartung der Anwender der Klassen
- Wenn < dann ist auch >,>= und <= ratsam

Ein/Ausgabe

- IO-Strom-Operatoren <<, >>

Elementfkt vs. Nichtelementfkt.

Elementfunktionen müssen sein:

- Zuweisung (=), Indizierung ([]), Aufruf (()) und Elementzugriffspfeil (->)

Elementfunktion sollten sein:

- Operatoren die den Zustand ihres Objekts verändern oder damit eng verknüpft sind
- Zuweisungsoperatoren (+= -=), Inkrementierung (++), Dekrementierung (--), Dereferenzierung (*)

Nichtelementfunktionen sollten sein:

- Symetrische Operatoren wie Arithmetik (+ - * /) Gleichheit (==), relationale (< > <= >=) und bitweise (& | ^) Operatoren

Rechenoperationen

Ziel: Frequenzgang eines Filters berechnen

```
double R, C, w;
complex c, j;
c = ( R / R + 1 / ( j * w * C ) );
```

Definition

- $a,b,d \in \Re$ und $i^2=-1$
- $c = a + b\,i;$

Arithmetik

- $c_1+c_2 = (a_1+a_2) + (b_1+b_2)i$, $c_1-c_2 = (a_1-a_2) + (b_1-b_2)i$
- $c_1*c_2 = (a_1a_2-b_1b_2) + (a_1b_2+b_1a_2)i$
- $c_1/c_2 = (a_1a_2+b_1b_2)/(a_2^2+ b_2^2)+((a_1b_2-b_1a_2) /(a_2^2+ b_2^2))i$

complex + complex

Elementfunktion operator+=

```
// complex += complex
Complex& Complex::operator+=(const Complex& rhs) {
          real+=rhs.real; imag+=rhs.imag;
          return *this; }
```

Nichtelementfunktion operator+

```
// complex = complex + complex
inline Complex operator+(const Complex& lhs, const Complex& rhs) {
     Complex cret(lhs);
     return cret += rhs;
}
```

Zusammenfassung

Deklaration der Klasse Foo

- Headerdatei Foo.h
- Schlüsselwort class

Implementierung der Klasse

- Quellcode-Datei Foo.cpp
- Namensraum beachten Foo::

Handregeln Elementfunktion vs. Nichtelementfunktion

- Elementfunktionen müssen sein:

Zuweisung (=), Indizierung ([]), Aufruf (()) und Elementzugriffspfeil (->)

- Elementfunktion sollten sein:

Operatoren die den Zustand ihres Objekts verändern oder damit eng verknüpft sind, also Zuweisungsoperatoren (+= -=), Inkrementierung (++), Dekrementierung (--), Dereferenzierung (*)

- Nichtelementfunktionen sollten sein:

Symetrische Operatoren wie Arithmetik (+ - * /) Gleichheit (==), relationale (< > <= >=) und bitweise (& | ^) Operatoren

Ausblick

Objektorientierte Analyse und Design (OOD/OOA) **Verteilte Ampelsteuerung**

- Wiederholung: Merkmale der Objektorientierung
- Neu: Systembeispiel verteilte Ampelsteuerung

Lernziele

- Sie können die wesentlichen **Prinzipien der Objektorientierung benennen** und **beschreiben**
  - Objekt-Klasse, Kapselung, Objektidentitiät, Kohärenz, Vererbung, Zusicherung-Verantwortlichkeit
- Sie können die wesentlichen Schritte der vorgestellten Methodik zur objektorientierten Analyse und Design mit Unified Modeling Language **wiedergeben** und für einfache Sachverhalte **anwenden**

Ausblick