

# Einführung in die Programmiersprache C++

Thomas Wiemann  
Institut für Informatik  
AG Wissensbasierte Systeme

# Letzte Vorlesung

- ▶ Dynamischer Speicher in C++
- ▶ Benutzung von `new` und `delete`
- ▶ Nebeneffekte: Sicherer Copy-Konstruktor
- ▶ `valgrind`

Tafelproblem der Woche: Zirkuläre Abhängigkeiten  
zwischen Klassen

# inline (1)

- ▶ Funktions- bzw. Methodenaufrufe erzeugen einen Overhead
  1. Erzeugen eines neuen Stack-Frames
  2. Sprung zu Code der Funktion
  3. Übergabe der Argumente und Rückgabewerte
- ▶ Für kleine Funktionen kann der Code direkt angegeben werden:

```
int getX() { return x; }
```

```
...
```

```
cout << foo.getX();
```

- ▶ Compiler kann den Code inlinen, d.h. ersetzen:

```
cout << foo.x;
```

## inline (2)

- ▶ In C haben wir Makros mit `#define` definiert:

```
#define max(x, y) ((x) > (y) ? (x) : (y))
```

- ▶ C Makros sind simple Textersetzung

```
    k = max(i++, j--);  
    ↪ k = ((i++) > (j--) ? (i++) : (j--));
```

- ▶ `x` und `y` werden mehrfach ausgewertet
- ▶ Nicht typsicher
- ▶ Extra Klammern, um die Auswertreihenfolge sicher zu stellen
- ▶ Inline-Funktionen lösen diese Probleme
- ▶ Nur eine Auswertung der Argumente
- ▶ Typen werden vom Compiler überprüft

## inline (3)

- Funktionsimplementierungen in der Klassendefinition (im Header File) sind automatisch inline

```
class Point {  
    double x, y;  
    ...  
    double getX() const { return x; } // Inline!  
    double getY() const { return y; }  
    double distanceTo(const Point &p) const;  
};
```

- Man kann Methoden / Funktionen als inline deklarieren

```
inline double Point::distanceTo(const Point &p) const {  
    ...  
}
```

- Inline-Funktionen müssen im Header implementiert sein
- Am besten direkt nach der Klassendeklaration (in separater .icc-Datei)
- Normale Methoden in einer .cc-Datei sind nicht inline

## inline (4)

- ▶ inline ist ein Hinweis, der Compiler muss ihn nicht beachten
- ▶ Folgende Dinge können inline verhindern:
  1. Eine Rekursion in einer inline Funktion
  2. Das Erzeugen eines Pointers zu der inline Funktion
  3. ...
- ▶ Inline-Funktionen machen das Executable größer
- ▶ Man sollte nur kleine Funktionen inlinen
- ▶ Accessors sind gute Kandidaten

# Memberinitialisierung (1)

- ▶ Bisher haben wir Konstruktoren wie folgt geschrieben:

```
Point::Point(double x, double y)
{
    x_coord = x;    // Store x and y values.
    y_coord = y;
}
```

- ▶ Frage: Warum funktioniert das?
- ▶ Wie werden x\_coord und y\_coord erzeugt?
- ▶ Antwort: Alle Klassenvariablen werden erzeugt, bevor der Code des Konstruktors ausgeführt wird. Beispiel:

```
class GraphicsEngine {
    Matrix viewportTransform;
    Matrix modelViewTransform;
    ...
public:
    GraphicsEngine();
    ...
};
```

Die Konstruktoren für Matrix werden vor dem Konstruktor der Klasse GraphicsEngine aufgerufen

## Memberinitialisierung (2)

- ▶ Nach unserer Vorgehensweise würde der Konstruktor für GraphicsEngine so aussehen:

```
GraphicsEngine::GraphicsEngine()  
{  
    viewportTransform = Matrix(4, 4);  
    modelViewTransform = Matrix(4, 4);  
    ... // Rest of graphics-engine initialization  
}
```

- ▶ Aber die Matrizen wurden schon erzeugt
- ▶ Dieser Code ist ineffizient:
  1. 2 mal Default-Matrix-Konstruktor
  2. 2 mal Zwei-Argumente-Matrix-Konstruktor
  3. 2 mal Zuweisungs-Operator (Copy / Cleanup)
- ▶ Wir wollen eigentlich nur Nummer 2 machen
- ▶ So genannten Member Initializer Lists lösen das Problem



# Memberinitialisierung (3)

- ▶ In den Initialisierungslisten kann man festlegen, welche Konstruktoren für Member aufgerufen werden

```
GraphicsEngine::GraphicsEngine() :  
viewportTransform(4, 4), modelViewTransform(4, 4)  
{  
    ... // Rest of graphics-engine initialization  
}
```

- ▶ Nach der Konstruktor-Signatur und vor dem Code des Konstruktors
- ▶ Beachte:
  - Doppelpunkt vor der Initializer-List
  - Liste durch Kommata getrennt

# Memberinitialisierung (4)

- ▶ Am gebräuchlichsten sind MILs für Member-Klassen
- ▶ Dann großer Performanzgewinn
- ▶ Einige Member benötigen Initializer, z.B.
  - Member-Objekte ohne Default-Konstruktor
  - const-Member
  - Referenzen

# Gliederung

1.Einführung in C

**2.Einführung in C++**

...

2.3. Klassen und Objekte

2.4 Dynamische Speicherverwaltung in C++

**2.5 Operatoren**

2.6 I/O-Streams

2.7 Klassen und Vererbung I

3.C++ für Fortgeschrittene

4.Weitere Themen rund um C++

# Operatoren (1)

- ▶ In C++ können Operatoren neue Bedeutungen gegeben werden:
- ▶ Wir kennen:

```
complex c1(3, 5);           // Some complex number
complex c2(-2, 4);
complex c3 = c1;
c3.multiply(3);             // Do some math
c3.add(c2);
```

- ▶ Schöner:

```
complex c1(3, 5);           // Some complex numbers!
complex c2(-2, 4);
complex c3 = c1 * 3 + c2;    // Do some math!
```

- ▶ Dies wird „Überladen von Operatoren“ genannt
- ▶ „Syntax should follow semantics“
- ▶ Man schreibt:

```
c1 + c2
```

- ▶ Der Compiler sieht:

```
c1.operator+(c2);
```

## Operatoren (2)

- ▶ Die Klasse `complex` definiert eine Member-Funktion mit dem Name `operator+`
- ▶ Dies ist ein binärer Operator
  - Operator-Member des Objektes auf der linken Seite (*left hand side*) des Operators wird aufgerufen
  - Das Objekt auf der rechten Seite (*right hand side*) des Operators wird als Argument übergeben
- ▶ Weiteres Beispiel:

```
c1 = c2;           // Assignment operator
```

- ▶ Signatur

```
complex & complex::operator=(const complex &c)
```

# Operatoren (3)

`complex & complex::operator=(const complex &c)`

- ▶ Argument ist eine konstante Referenz
  - D.h. wird nicht verändert
- ▶ Rückgabewert ist eine nicht konstante Referenz zu der LHS der Zuweisung
- ▶ Erlaubt Operatorenketten:

`A = B = C = D = 0`

- ▶ Zuweisungen laufen in 3 Schritten ab:
  - Löschen des Inhalts des LHS-Objekts
    - Genau wie ein Destruktor
  - Kopieren des Inhalts des RHS-Objekts
    - Genau wie es der Copy-Konstruktor tut
  - Rückgabe der nicht konstanten Referenz zur LHS
    - Dies geschieht mit `return *this`

# Operatoren (4)

- ▶ `this` ist ein neues Schlüsselwort in C++
  - `this` ist ein Pointer auf die Adresse eines Objekts
  - In `c2.operator=(c1)` ist `this` ein Pointer zu `c2`
  - Der `=`-Operator gibt das zurück, worauf `this` zeigt
- ▶ Der Assignment-Operator verwendet also ähnlichen Code, wie der Konstruktor / Destructor
  - Ziel: Code wiederverwenden
  - D.h. gemeinsame Funktionen in „helper“-Funktionen, z.B. `cleanup()` oder `copy(complex &c)` auslagern
- ▶ Man muss immer überprüfen, ob eine Selbstzuweisung vorliegt:

```
c1 = c1;           // self assignment
```

Wegen der 3 Schritte!!!

# Operatoren (5)

- ▶ Die Überprüfung auf Selbstzuweisung ist einfach (dank `this`)

```
complex & complex::operator=(const complex &c)
{
    // Check for self-assignment.
    if (this == &c)
        return *this; // Skip assignment.

    ...                // Do normal assignment

    return *this;
}
```

- ▶ C++ unterstützt auch die Operatoren `+=` `-=` `/=` `*=` deren Verwendung analog ist
- ▶ Hier sollte es korrekte Selbstzuweisung geben
- ▶ Arithmetische Operatoren (`+` `-` `*` `/`) einfach, dank `+=` `-=`



# Operatoren (6)

- ▶ == und != Operatoren geben den Datentyp bool zurück

```
bool complex::operator==(const complex &c) const;  
bool complex::operator!=(const complex &c) const;
```

- ▶ RHS ist ein konstantes Objekt
- ▶ Memberfunktion ist ebenfalls konstant
- ▶ Reihenfolge zur Erstellung von der bereits vorgestellten Operatoren:
  1. Erstelle -=-Operator auf Basis von Konstruktor und Destructor (Code wiederverwenden!)
  2. Erstelle die Operatoren += -= etc.
  3. Benutze die in 2. erstellten Funktionen, um + - usw. zu programmieren
  4. Implementiere den == Operator
  5. Verwende 4. dazu != zu erstellen (`return !(*this == c)`)
- ▶ Es gibt noch weitere Operatoren!

# C++ Klassen und structs (1)

- ▶ In C++ sind structs wie Klassen
- ▶ Sie können Konstruktoren, Member-Funktionen usw. haben
- ▶ Der einzige Unterschied ist, dass per default alles `public` ist

```
struct s { ... };
```

```
class s { public: ... };    // same thing
```

- ▶ Konstruktoren für Klassen sind sinnvoll, z.B. um Initialwerte zu setzen
- ▶ structs werden in der Regel verwendet, wenn sie nicht den vollen Umfang einer Klasse haben:
- ▶ So genannte Helper-Klassen
- ▶ „a chunk of Data“
- ▶ Das verstecken von structs in Klassen ist sinnvoll
  - Teil der Abstrahierung / Kapselung
  - Gutes objektorientiertes Design

# C++ Klassen und structs (2)

- ▶ Man kann structs und Klassen in anderen Klassen deklarieren
- ▶ Beispiel:

```
class Scheduler {  
    private:  
        // A "scheduled task" struct  
        struct task {  
            int id;  
            string desc;  
            task *next;  
        };  
        task *schedTasks; // A list of tasks  
        ...  
};
```

- ▶ task kann in der Scheduler-Klasse benutzt werden.
- ▶ task kann nicht in anderen Klassen / außerhalb von Klassen benutzt werden.
- ▶ Wenn es public wäre, dann als Scheduler::task

# C++ Klassen und structs (3)

## ► Verkettete Liste als struct

```
struct node {  
    int index;  
    int value;  
    node *next;    // Pointer to next node in list  
};
```

## ► Besser:

```
struct node {  
    int index;    // Index of element in vector  
    int value;    // Value of element in vector  
    node *next;    // Pointer to next element, or 0  
    node(int idx, int val, node *np) :  
        index(idx), value(val), next(np) { }  
};
```

## ► Verkettung:

```
node *n1 = new node(3, 5, 0);    // Elem3 = value 5  
node *n2 = new node(5, -2, 0);    // Elem5 = value -2  
n1->next = n2;
```