10. Klassen

Prof. Dr. Markus Gross

Informatik I für D-ITET (WS 03/04)

- Objektorientierte Programmierung
- Das Konzept der Klassen
- Members
- Objekte
- Konstruktoren und Destruktoren
- `this`-Pointer
- Public und Private Sections

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Objektorientierung

- *Objektorientierung* (*OO*) ist ein konzeptioneller Ansatz zum Klassendesign, welcher programmiersprachenunabhängig ist
- Merkmale von OO sind:
  - Abstraktion (abstraction)
  - Verkapselung (encapsulation)
  - Polymorphismus (polymorphism)
  - Vererbung (inheritance)
  - Wiederverwendbarkeit (reusability)
- OO-Programmierung ist *datenzentriert*
- Repräsentation der Daten steht im Vordergrund
  - Methoden zur Datenbearbeitung
- In C++ dient die Klasse zur Implementierung der OO (`class`)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

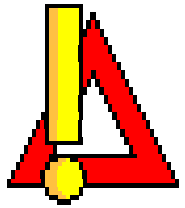
Objektorientierung

- Merkmale eines klassischen C-Structs:
  - Verkapselt Mitgliedsvariablen
  - Zugriff mittels . -Operator
  - Legt fest, wieviel Speicher benötigt wird
  - Bestimmt, welche Operationen auf den Daten möglich sind
  - Operationen (Methoden) werden in Form von Funktionen definiert
  - Funktionen haben globalen Gültigkeitsbereich
- Idee: Verstecke Daten komplett vor dem Benutzer
- Zugriff über bestimmte *Zugriffsfunktionen*
- Diese Funktionen stellen das *Interface* zur Klasse
- Funktionen bekommen somit *lokalen* Charakter

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementation einer Klasse

- Erfolgt durch *Klassendeklaration*
  - Beschreibt Mitgliedsdaten (data members)
  - Und Mitgliedsfunktionen (member functions/methods)
  - Diese stellen das *public interface* der Klasse dar
- Sowie durch *Definition* der Methoden
- Konvention: Klassennamen schreiben wir mit führendem Grossbuchstaben
- “Eselsbrücke” : Die Klasse ist zunächst ein Struct mit lokalen Funktionen und verbesserten Zugriffsrechten !



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel: `Stock`-Klasse

- Benötigte *Methoden*
  - Neue Aktie ins Portfolio einfügen
  - Aktien kaufen und verkaufen
  - Aktienwert anpassen
  - Anzeige und Darstellung wichtiger Information
- Benötigte *Daten*
  - Firmenname
  - Anzahl der Aktien
  - Aktienwert
  - Gesamtwert des Portfolios

Beispiel_1: Stock-Klasse

```
class Stock
{

private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }


public:
    void acquire(const char * co, int n, double pr);
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};
```

Private Mitgliedsvariablen

Oeffentliche Mitgliedsfunktionen

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

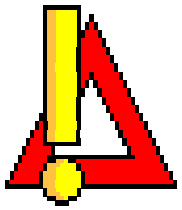
Objekte

- **`Stock`** ist somit als neuer Typenname für eine Klasse deklariert
- Erlaubt die Definition von *Objekten* dieses Klassentyps
  - **`Stock sally;`**
  - **`Stock solly;`**
- Mit der Definition werden sowohl Mitgliedsvariablen angelegt, als auch Zugriffsfunktionen festgelegt
- Die *Zugriffskonstrolle* erfolgt durch die Schlüsselwörter **`private`** und **`public`**
- Mitglieder der **`private section`**
  - Nur für Klassenmitglieder sichtbar
  - Kein Zugriff von Aussen möglich
  - Sind verkapselt (data hiding)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Objekte

- Mitglieder der **`public section`**
  - Stellen das Klasseninterface in Form von Zugriffsfunktionen dar
  - Diese können auf private Daten der Klasse zugreifen
  - Verstecken die Details der Implementation
  - Sind von Aussen zugreifbar
- Klasse kann damit als “Black Box” verwendet werden, wobei nur das Interface bekannt gegeben wird
  - Klassenbibliotheken
  - Wiederverwendbarkeit von Code
- Goldene Regel: Jegliche Daten sollten möglichst in die private Section der Klasse
- **`private`** ist default Zugriffstyp



Mitgliedsfunktionen

- Definition erfolgt in Analogie zur Definition regulärer Funktionen
  - Verwendung des *Scope* Operators (`::`)
  - Haben Zugriff auf die private Section der Klasse
  - Somit kann gleicher Name für verschiedene Klassen verwendet werden
- Beispiel:
  - `void Stock::update(double price)`
- `update()` wird hier als Mitglied der Klasse `Stock` definiert
- `update()` hat also *class scope*
- `Stock::update()` ist der qualifizierte Name (*qualified name*) der Methode

Beispiel_2: Implementation

```
void Stock::acquire(const char * co, int n, double pr)
{
    strncpy(company, co, 29); // truncate co to fit if needed
    company[29] = '\0';
    shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(int num, double price)
{
    shares += num;
    share_val = price;
    set_tot();
}

void Stock::sell(int num, double price)
{
    if (num > shares)
    {
        cerr << "You can't sell more than you have!\n";
        exit(1);
    }
    shares -= num;
    share_val = price;
    set_tot();
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}
```

Verwendung privater Mitglieder der Klasse

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

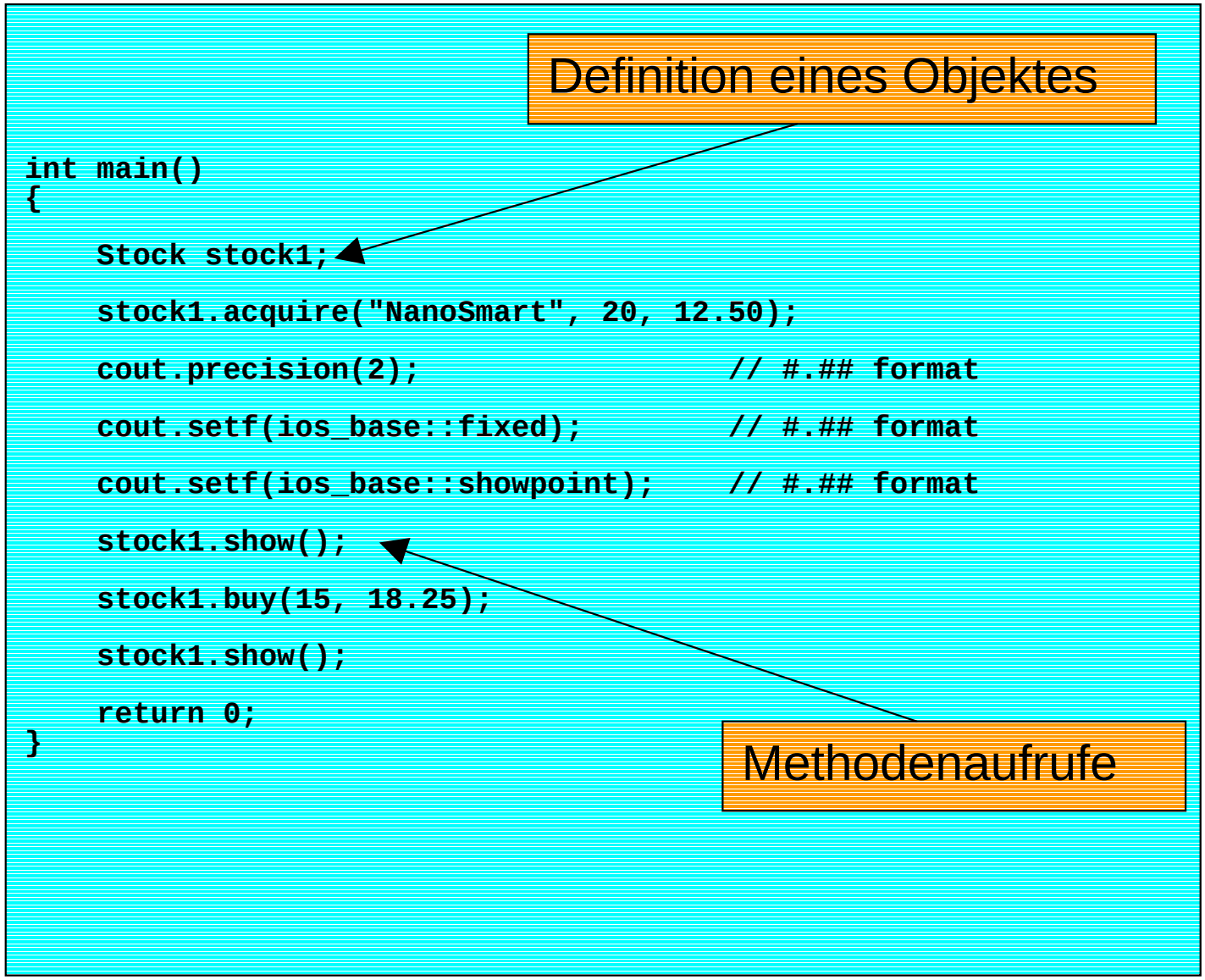
Scope und Zugriff

- Deklaration der Klassen wird in einem getrennten File vorgenommen
- Headerfile
  `stock.h`
- Definition der Methoden erfolgt im Quellfile
  `stock.cpp`
- Zugriff auf Methoden eines Objektes erfolgt durch bekannten membership Operator

```
Stock kate;
kate.show(); // Aufruf einer Mitgliedsfunktion
```

- Mitgliedsvariablen müssen für jedes Objekt getrennt verwaltet werden
- Mitgliedsfunktionen werden vom Compiler nur *einmal* angelegt !

Beispiel_3: Verwendung

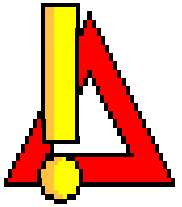


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Klassendeklaration

- Eine Klasse besitzt also sowohl Mitgliedsdaten, als auch Mitgliedsfunktionen
- Allgemeine Deklarationssyntax sieht wie folgt aus

```
class ClassName
{
private:
   data member variables
public:
   member function prototypes // Interface
}
```



- Zur Verwendung von Mitgliedsfunktionen muss zuerst ein entsprechendes Objekt angelegt werden
- Aufruf erfolgt mit dem Membership Operator

```
objectName.foo();
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Konstruktoren

- Daten der private Section der Klasse sind von aussen nicht sichtbar
- Daher können sie nicht direkt initialisiert werden
  - Vergleiche Struct und dessen Initialisierung
- Man benötigt eine spezielle Funktion zur Initialisierung
  - Sowohl als Funktionsargument, als auch als Rückgabetyp
- Dies kann durch einen *Konstruktor* erreicht werden
- Wird bei Objektdefinition automatisch aufgerufen
- Der Name eines Konstruktors entspricht dem Klassennamen
- Man benötigt Prototyp und Funktionsdefinition
- Argumente dürfen keine Namen von Mitgliedsvarablen tragen

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Konstruktoren

- Beispiel:

```
Stock(const char *co, int n = 0, double pr = 0.0);
Stock::Stock(const char *co, int n, double pr)
{......}
```

- Konstruktoren können explizit bei der Objektdefinition aufgerufen werden

```
Stock food = Stock("World", 250, 1.0);
```

- Konstruktoren können implizit aufgerufen werden

```
Stock food("World", 250, 1.0);
```

- In Verbindung mit Pointern

```
Stock *pstock = new Stock("World", 250, 1.0);
```

- Default Konstruktor wird von C++ angelegt, wenn kein Konstruktor implementiert ist
- Eigener Default Konstruktoren möglich `Stock();`

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Destruktoren

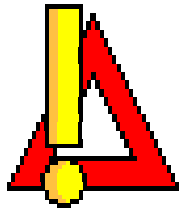
- Wenn Gültigkeitsbereich des definierten Objektes ausläuft, wird ein *Destruktor* aufgerufen
- Destruktoren sorgen für ordnungsgemässe Freigabe von verwendetem Speicher (Kontext **new**)
- Destruktoren haben keine Argumente
- Tragen Klassennamen mit vorgestellter Tilde

```
~Stock(); // Destruktor
```

- Compiler generiert ebenfalls einen Default Konstruktor
- Destruktoren werden *automatisch* aufgerufen

```
Stock::~Stock(){... // implementation }
```

- Beim fortschrittlichen Klassendesign muss genaustens auf Konstruktoren und Destruktoren geachtet werden



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Header Files

- Klassendeklarationen werden in eigene Headerfiles geschrieben
- Präprozessor-Direktiven verhindern die Mehrfacheinbindung eines Headerfiles

```
#ifndef _STOCK1_H
#define _STOCK1_H
// place include file contents here
#endif
```

- Beim ersten Durchlauf wird **`_STOCK1_H`** generiert
- Bei weiteren Aufrufen wird Inhalt ignoriert und somit Mehrfachdeklaration vermieden
- Beispiel unserer **`Stock`**-Klasse

Beispiel_4: Headerfile

```
// stock1.h

#ifndef _STOCK1_H_
#define _STOCK1_H_

class Stock
{

private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }

public:
    Stock();          // default constructor
    Stock(const char * co, int n = 0, double pr = 0.0);
    ~Stock();         // noisey destructor
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};

#endif
```

Direktive

Konstruktoren

Destruktor

Beispiel_5:Verwendung von Stock

```
// usestok1.cpp -- use the Stock class

#include <iostream>
using namespace std;
#include "stock1.h"

int main()
{
// using constructors to create new objects
    Stock stock1("NanoSmart", 12, 20.0);              // syntax 1
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // syntax 2

    cout.precision(2);                              // #.## format
    cout.setf(ios::fixed, ios::floatfield);     // #.## format
    cout.setf(ios::showpoint);                      // #.## format

    stock1.show();
    stock2.show();
    stock2 = stock1;                          // object assignment

// using a constructor to reset an object
    stock1 = Stock("Nifty Foods", 10, 50.0);    // temp object

    cout << "After stock reshuffle:\n";
    stock1.show();
    stock2.show();
    return 0;
}
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bemerkungen

- Objekte können einander zugewiesen werden

```
stock1 = stock2;
```

- Destruktoren werden am Ende von main aufgerufen
  - Last in – First out
- Mit Konstruktoren können Objekte reinitialisiert werden
- Dabei legt der Compiler ein unsichtbares, temporäres Objekt zu Kopierzwecken an
- Dieses wird vom Destruktor wieder gelöscht
  - Implementationsabhängig
- Konstante Objekte benötigen Funktionen, welche das entsprechende Objekt nicht verändern

```
const Stock land = Stock(.....);
```

- Neue Art von konstanter Funktion (nachgestellt)

```
void stock::show() const
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Der `this`-Pointer

- Manche Mitgliedsfunktionen müssen das Objekt, welches sie aufruft, erkennen
- Dies erfolgt über den sogenannten **`this`**-Pointer
- Beispiel: Funktion, welche zwei **`Stock`**-Objekte vergleicht und das grössere Objekt zurückgibt
- Eine solche Funktion benötigt ein Objekt als Argument
  - Objekt wird von der Funktion nicht verändert
  - Wir verwenden einen Call by Reference
  - Funktion gibt eine Referenz auf das grössere Objekt zurück

```
const Stock & topval(const Stock & s) const;
// Prototyp, Funktion verändert Objekt nicht!
top = stock1.topval(stock2); //oder auch
top = stock2.topval(stock1);
```

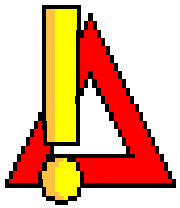
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Der `this`-Pointer

- Implementation

```
const Stock & Stock::topval(const Stock & s) const
 {
   if (s.total_val > total_val)
          return s;
   else
          return *this; // Pointer auf Objekt
}
```

- Der **`this`**-Pointer ist also ein Pointer auf das aufrufende Objekt
- **`total_val`** entspricht also **`this->total_val`**
- **`this`** kann in diesem Fall nicht geändert werden



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Arrays von Objekten

- Arrays von Objekten können beliebig angelegt werden
- Beispiel:

```
Stock mystuff[4];  // Definition
mystuff[2].show(); // Aufruf
```

- Initialisierung erfolgt mittels Konstruktoren
- In diesem Fall muss ein Default-Konstruktor vorhanden sein
- Member Variablen haben *class scope*