11. Klassen II

Prof. Dr. Markus Gross
Informatik I für D-ITET (WS 03/04)

- Fallbeispiel: Stack-Klasse
- Ueberladen von Operatoren
- `friend`-Funktionen
- Typenkonversion von Objekten

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstrakter Datentyp

- *Abstrakte Datentypen* sind Strukturen, welche Daten und Operationen unabhängig von der Programmiersprache oder Implementierung beschreiben
- Beispiele sind:
  - Tabellen
  - Listen
  - Fifos (First-In First-Out)
  - Stack
- Ein Stack speichert Daten linear, wobei jeweils das oberste Element zugreifbar ist
- Ein Stack ist eine spezielle Form einer Liste
- Mehr in Informatik II

Der Stack

- Operationen auf dem Stack
  - Create Stack
  - Add or remove items
  - Check if full or empty
- Operationen stellen das *Interface* zur Klasse dar und verbergen die Implementationsdetails
- `Item` sollte ein allgemeiner Datentyp sein
- Deklaration einer Stack-Klasse wie folgt:

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_1: Stack-Klasse

```
// stack.h -- class definition for the stack ADT

#ifndef _STACK_H_
#define _STACK_H_


typedef unsigned long Item;


class Stack
{

private:
    enum {MAX = 10};     // constant specific to class
    Item items[MAX];     // holds stack items
    int top;             // index for top stack item

public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);    // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);           // pop top into item
};

#endif
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_1: Stack-Implementierung

```
Stack::Stack()    // create an empty stack
{
    top = 0;
}
bool Stack::isempty() const
{
    return top == 0;
}
bool Stack::isfull() const
{
    return top == MAX;
}
bool Stack::push(const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
bool Stack::pop(Item & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

Beispiel_1: Stack-main

```
// Auszug

cout << "Please enter A to add a purchase order,\n"
        << "P to process a PO, or Q to quit.\n";
    while (cin >> c && toupper(c) != 'Q')
    {
        while (cin.get() != '\n')
            continue;
        if (!isalpha(c))
        {
            cout << '\a';
            continue;
        }
        switch(c)
        {
             case 'A':
             case 'a': cout << "Enter a PO number to add: ";
                       cin >> po;
                       if (st.isfull())
                           cout << "stack already full\n";
                       else
                           st.push(po);
                       break;
             case 'P':
             case 'p': if (st.isempty())
                           cout << "stack already empty\n";
                       else {
                           st.pop(po);
                           cout << "PO #" << po << " popped\n";
                       }
                       break;
        }
```

Ueberladen von Operatoren

- Funktionsheader in allgemeiner Form:
  `operatorop(argument_list)`
- Angenommen, ein `+` Operator ist überladen
- Bei Anwendung des Operators auf zwei Objekte
  ```
  district = sid + sara;
  ```
  Ruft der Compiler die entsprechende Operator-Funktion auf
  ```
  district = sid.operator+(sara);
  ```
- Damit entspricht der Operator einem *Funktionsaufruf*
- Das `sid` Objekt wird implizit verwendet (`this`-Pointer), das `sara` Objekt explizit als Argument
- Beispiel: `Time`-Klasse

Copyright: M. Gross, ETHZ, 2003

Beispiel_2: Time-Klasse

```
// mytime1.h -- Time class after operator overloading


#ifndef MYTIME1_H_
#define MYTIME1_H

_
#include <iostream>

using namespace std;


class Time
{

private:
    int hours;
    int minutes;

public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    void Show() const;
};

#endif
```

Ueberladen von Operatoren

- Das *Ueberladen* von Operatoren ist eine Variante des *Polymorphismus*
- Es erlaubt, die gleichen Operatoren für verschiedene Aufgaben zu verwenden
- In C++ sind bereits diverse Operatoren überladen
  - `*` steht sowohl für einen Pointer als auch für die Multiplikation
  - `&` steht sowohl für die Adresse als auch für die Referenz
- Das Ueberladen bedarf einer Neudefinition des entsprechenden Operators
- Dies erfolgt durch Implementation einer entsprechenden Operator-Funktion
- Nur für gültige C++Operatoren möglich!

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_2: Time-Methoden

```
// mytime1.cpp - Auszug

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show() const
{
    cout << hours << " hours, " << minutes << " minutes";
    cout << '\n';
}
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_2: Time-main

```
// usetime1.cpp -- use second draft of Time class
// compile usetime1.cpp and mytime1.cpp together

#include <iostream>
#include "mytime1.h"
using namespace std;


int main()
{
    Time A;
    Time B(5, 40);
    Time C(2, 55);

    cout << "A = ";
    A.Show();
    cout << "B = ";
    B.Show();
    cout << "C = ";
    C.Show();

    A = B.operator+(C);   // function notation
    cout << "A = B.operator+(C) = ";
    A.Show();
    B = A + C;            // operator notation
    cout << "A + C = ";
    B.Show();
    return 0;
}
```

Ueberladen von Operatoren

- Operatoren können also auf zwei Arten aufgerufen werden

```
A = B.operator+(C);
```

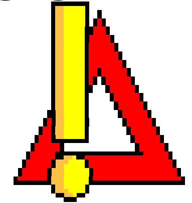
- Sowie

```
A = B + C;
```

- Das linke Objekt ist das aufrufende Objekt
- Einschränkungen:
  - Zumindest ein Operand muss ein benutzerdefinierter Typ sein
  - Die Syntax des Originaloperators muss eingehalten werden
  - Die Precedence der Operatoren muss eingehalten werden
  - Neue Operator-Symbole sind NICHT möglich

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Friend-Funktionen



- Auf die private Section einer Klasse kann nur mittels Methoden dieser Klasse zugegriffen werden
  - Encapsulation!
- Manchmal ist dies zu restriktiv
- Beispiel: Ueberladene Multiplikation

```
A = B*2.75; //entspricht
A = B.operator*(2.75);
```

- Jedoch

```
A = 2.75*B; // ? Asymmetrischer Aufruf
```

- Der Aufruf ist *nichtkommutativ*
- Zur Zahl existiert keine entsprechende Operator-Funktion
- Man benötigt eine Funktion, welche nicht Mitglied der Klasse ist und trotzdem auf die private Section zugreifen kann

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Friend-Funktionen

- Funktionsaufruf wäre also

```
A = operator*(2.75,B);
```

- Entsprechender Prototyp dieser Funktion

```
Time operator*(double m, const Time & t);
```

- Linker Operand entspricht erstem Argument
- Problem: Nicht-Mitgliedsfunktion hat keinen Zugriff auf private Daten der Klasse
- Einführung der `friend`-Funktion, welche Zugriff auf die private Section der Klasse hat
- *VORSICHT: Hier umgehen wir das Konzept der Verkapselung – Kritik an C++*

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Friend-Funktionen

- Prototyp der Friend-Funktion wird innerhalb der Klasse deklariert

```
friend Time operator*(double m, const Time & t);
```

- Implementation erfolgt konventionell ohne Scope-Operator `::`

```
Time operator*(double m, const Time & t)
{
Time result;
Result.hours = totalminutes / 60;
totalminutes = t.minutes;
......
}
```

- Gute Uebung: Vektorklasse in Kapitel 10

Typenkonversion von Klassen

- Zur Erinnerung:

  In C++ werden kompatible Typen automatisch konvertiert

  ```
  int side = 3.33; // wird nach 3 konvertiert
  ```

- Nichtkompatible Typen bedürfen expliziter Umwandlung

  ```
  int *p = (int *) 10; // Pointer auf 0xa
  ```

- Bei Objekten kann die Typenumwandlung entweder durch Konstruktoren, oder durch Konversionsfunktionen erfolgen
- Beispiel:

  `Stonewt`-Klasse, welche Gewicht in altem angelsächsischen Mass speichert

Beispiel_3: Stonewt-Klasse

```
// stonewt.h -- definition for Stonewt class

#ifndef STONEWT_H_
#define STONEWT_H_

class Stonewt
{

private:
    enum {Lbs_per_stn = 14};        // pounds per stone
    int stone;                      // whole stones
    double pds_left;                // fractional pounds
    double pounds;                  // entire weight in pounds

public:
    Stonewt(double lbs);            // constructor for double pounds
    Stonewt(int stn, double lbs);   // constructor for stone, lbs
    Stonewt();                      // default constructor
    ~Stonewt();
    void show_lbs() const;          // show weight in pounds format
    void show_stn() const;          // show weight in stone format
};

#endif
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_3: Stonewt-cpp

```
// stonewt1.cpp - Auszug


// construct Stonewt object from double value

Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;     // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}


// construct Stonewt object from stone, double values

Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}


Stonewt::Stonewt()          // default constructor, wt = 0
{
    stone = pounds = pds_left = 0;
}


Stonewt::~Stonewt()         // destructor
{
}
```

Konversion mit Konstruktoren

- `int` und `float` können mit Hilfe des Konstruktors in ein `Stonewt`-Objekt konvertiert werden

```
Stonewt(double lbs);
```

- Damit werden folgende Anweisungen möglich

```
Stonewt myCat;
myCat = 19.6;
```

- Temporäres Objekt angelegt und mit 19.6 initialisiert
- Nur Konstruktoren mit einem Argument können zur Typenkonversion verwendet werden
- Automatischer (impliziter) Aufruf des Konstruktors
- Kann durch `explicit` ausgeschaltet werden

Konversion mit Funktionen

- Umgekehrte Konversion von Stonewt in **`double`** mit Konstruktoren nicht machbar

```
Stonewt wolfe(256.7);
double host = wolfe; //möglich?
```

- Dazu bedarf es expliziter Konversionsfunktionen, welche vom Compiler verwendet werden
- Allgemeine Form

```
operator TypeName( );
```

- Beispiel:

```
operator double();
```

- Erweiterte Klassendefinition

Beispiel_4: Stonewt1-Klasse

```
// stonewt1.h -- revised definition for Stonewt class

#ifndef STONEWT1_H_

#define STONEWT1_H_
class Stonewt
{

private:
    enum {Lbs_per_stn = 14};      // pounds per stone
int stone;                        // whole stones
    double pds_left;              // fractional pounds
    double pounds;                // entire weight in pounds

public:
    Stonewt(double lbs);          // construct from double pounds
    Stonewt(int stn, double lbs); // construct from stone, lbs
    Stonewt();                    // default constructor
    ~Stonewt();
    void show_lbs() const;        // show weight in pounds format
    void show_stn() const;        // show weight in stone format
// conversion functions
    operator int() const;
    operator double() const;
};

#endif
```

Beispiel_4: Stonewt1-cpp

```
// stonewt1.cpp - Auszug

// conversion functions

Stonewt::operator int() const
{
    return int (pounds + 0.5);
}



Stonewt::operator double()const
{
    return pounds;
}
```