

Exercise Week 12

GianAndrea Müller
`mailto:muellegi@student.ethz`

May 24, 2018

Time Schedule

- 30' Vectors class
- 10' Dynamisch allokiertes Speicherplatz
- 10' Dynamische Datentypen

Learning Objectives

- Verständnis: Klassen
- Kenntnis von Dynamischer Speicherverwaltung
- Kenntnis von Dynamischen Datenstrukturen

Klassen: Motivation

Was

- Klassen: Kombination von Variablen und Funktionen
- Instanzen
- Objekte

Objektorientiertes Programmieren

Die Grundidee besteht darin, die Architektur einer Software an den Grundstrukturen desjenigen Bereichs der Wirklichkeit auszurichten, der die gegebene Anwendung betrifft.

Wieso

- Verkapselung
- Wiederverwendbarkeit

Objektorientiertes Programmieren

Klassen: Grundlagen

Mitglieder

- **Mitgliedsfunktion:**

Funktion die in der Klasse angelegt wird.

- **Mitgliedsvariable:**

Variable die in der Klasse angelegt wird.

Mitglieder können nur über eine Instanz der Klasse aufgerufen werden.

```
1 //Definition der Klasse
2 class Vector {...};
3 Vector v1; //Deklaration einer Instanz
4 v1.memberVariable;
5 v1.memberFunction();
6 Vector* pv1 = &v1;
7 pv1->memberVariable;
```

Verkapselung

- **Private Mitglieder:**

Nach dem Schlüsselwort `private` kommen alle Variablen die versteckt sein sollen.

- **Öffentliche Mitglieder:**

Nach dem Schlüsselwort `public` kommen alle Variablen die öffentlich sein sollen.

Nur öffentliche Mitglieder können über einen der beiden Zugriffoperatoren erreicht werden.

Mitgliedsfunktionen haben Zugriff auf private Mitgliedsvariablen.

Klassen: Grundlagen

```
1 class Vector {  
2 private:  
3     double x;  
4     double y;  
5 };
```

[Codeboard](#)

[Zusätzliches Beispiel](#)

Klassen: Konstruktoren

Konstruktor

- Öffentliche Mitgliedsfunktion
- Wird automatisch beim Erstellen einer Instanz der Klasse aufgerufen!
- Wird eingesetzt um Mitgliedsvariablen zu initialisieren.

```
1 public:
2     Vector () {
3         x = 0;
4         y = 0;
5     }
6     Vector (double _x, double _y) : x(_x), y
        (_y) {}
```


Klassen: Konstruktoren

```
1  class Vector {
2      double x;
3      double y;
4  public:
5      Vector () : x(0),y(0){}
6      Vector (double _x, double _y)
7          : x(_x),y(_y) {}
8  };
9
10 Vector v1;
11 Vector v2();
12 Vector v3(1.0, 2.3);
```

Klassen: Zugriffsmethoden

Zugriff

- Der Zugriff ist durch Verkapselung eingeschränkt.
- Lösung: Sicherer Zugriff mit Zugriffsmethoden.

```
1 double get_x() const {return x;}
2 double get_y() const {return y;}
3
4 void set_x(const double _x) {x = _x;}
5 void set_y(const double _y) {y = _y;}
```

Klassen: Zugriff auf Mitglieder

this

- Der `this` Pointer speichert die Adresse seiner Instanz einer Klasse.
- Er ist in jeder Klasse vorhanden und kann in Mitgliedsfunktionen benutzt werden um auf die aktuelle Instanz zuzugreifen.

```
1 double get_x() const {return this->x;}  
2 double get_y() const {return (*this).y;}
```

Klassen: Arithmetische Operatoren

Argumentübergabe

- Die Operatoren, die als Mitgliedsfunktion überladen werden, erhalten die aufrufende Instanz als erstes Argument.
- Der Rückgabetyt hängt vom überladenen Operator ab. Für Zuweisungen wird eine Referenz zurückgegeben.

```
1 Vector& operator+= (const Vector& b){  
2     x += b.get_x();  
3     y += b.get_y();  
4     return *this;  
5 }  
6 //Im main:  
7 Vector v3(3,4), v4(1,2);  
8 v3 += v4;
```

Klassen: Arithmetische Operatoren

```
1  Vector& operator+= (const Vector& b){
2      x += b.get_x();
3      y += b.get_y();
4      return *this;
5  }
6
7  //Ausserhalb der Klasse
8  Vector operator+ (const Vector& a, const
9      Vector& b) {
10     Vector res = a;
11     res += b;
12     return res;
13 }
```

Dynamische allozierter Speicherplatz

```
1  int * dyn_int = new int (3);  
2  
3  int size = 5;  
4  int * dyn_array = new int [size];
```

Dynamische allokiertes Speicherplatz

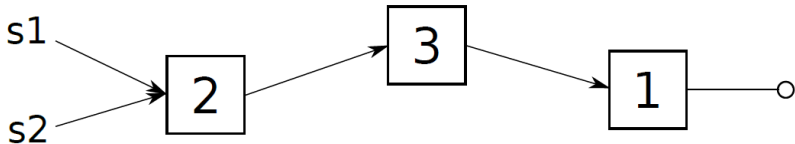
```
1  int * dyn_int = new int (3);  
2  
3  int size = 5;  
4  int * dyn_array = new int [size];  
5  
6  delete dyn_int;  
7  dyn_int = 0;  
8  delete[] dyn_array;  
9  dyn_array = NULL;
```

Dynamische Datentypen - Stack

```
1  class stack {
2  public:
3      void push (int value){...}
4      int pop (){}
5      ...
6      void print (){...}
7
8  private:
9      ln* top_node; //ln = list node
10 };
11
12 struct ln {
13     int key;
14     ln * next;
15 }
```


Dynamische Datentypen - Stack

```
1 stack s1;  
2 s1.push(1);  
3 s1.push(2);  
4 s1.push(3);  
5 s2(s1);
```



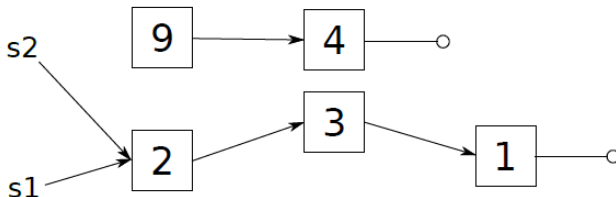
Dynamische Datentypen - Stack - Tiefe Kopie

```
1  stack::stack(const stack& s) : top_node(0)
   {
2      copy(s.top_node, top_node);
3  }
4
5  void stack::copy(const ln* from, ln*& to){
6      assert (to == 0);
7      if(from != 0){
8          to = new ln(from->key);
9          copy(from->next, to->next);
10     }
11 }
```

```
1  s2(s1);
```

Dynamische Datentypen - Stack - Zuweisung

```
1 stack s2;  
2 s2.push(4);  
3 s2.push(9);  
4 s2 = s1;
```



Dynamische Datentypen - Stack - Zuweisung

```
1 void stack::clear(ln* from){  
2     if (from != 0){  
3         clear(from->next);  
4         delete from;  
5     }  
6 }
```

Dynamische Datenstrukturen - Stack - Zuweisung

```
1 stack& stack::operator= (const stack& s) {  
2     if (top_node != s.top_node) { // test  
3         for self-assignment  
4             clear(top_node);  
5             top_node = 0; // fix dangling pointer  
6             copy(s.top_node, top_node);  
7     }  
8     return *this;  
9 }
```

```
1 s1 = s2;
```

Dynamische Datenstrukturen - Stack - Destruktor

```
1 void useStack(){
2     stack temp;
3     temp.push(2);
4     temp.pop();
5 } //end of scope, destruction
6
7 stack::~~stack() {
8     clear(top_node);
9 }
```

Klassen: Spezielle Mitgliedsfunktionen

Standardmitglieder

- Defaultkonstruktor
- Kopierkonstruktor
- Zuweisungsoperator
- Defaultdestruktor

Regel der drei: Wenn entweder Destruktor, Kopierkonstruktor und Zuweisungsoperator neu definiert werden sollten alle drei neu definiert werden.