Klassen

- Klassen
- Instanzen
- Objekte

- `class`
- `struct`
- `private`, `public`
- `string`
- `#include <cmath>`
- `#include <cstdio>`
- `#include <string>`

Klassen & Objekte

- Klassen sind (benutzerdefinierte) Datentypen
  - erweitern `struct` aus C
  - bestehen aus Daten und Methoden
  - Methoden = Fktn. auf den Daten der Klasse
- Deklaration etc. wie bei Struktur-Datentypen
  - Zugriff auf Members über Punktoperator
  - sofern dieser Zugriff erlaubt ist!
    - Zugriffskontrolle = Datenkapselung
- formale Syntax: `class ClassName{ ... };`
- Objekte = Instanzen einer Klasse
  - entspricht Variablen dieses neuen Datentyps
  - wobei Methoden nur 1x im Speicher liegen
- später: Kann Methoden überladen
  - d.h. Funktionalität einer Methode abhängig von Art des Inputs
- später: Kann Operatoren überladen
  - z.B. $x + y$ für Vektoren
- später: Kann Klassen von Klassen ableiten
  - sog. Vererbung
  - z.B. $\mathbb{C} \supset \mathbb{R} \supset \mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N}$
  - dann: $\mathbb{R}$ erbt Methoden von $\mathbb{C}$ etc.

Zugriffskontrolle

- Klassen (und Objekte) dienen der Abstraktion
  - genaue Implementierung nicht wichtig
- Benutzer soll so wenig wissen wie möglich
  - *black-box* Programmierung
  - nur Ein- und Ausgabe müssen bekannt sein
- Richtiger Zugriff muss sichergestellt werden
- Schlüsselwörter `private`, `public` und `protected`
- `private` (Standard)
  - Zugriff nur von Methoden der gleichen Klasse
- `public`
  - erlaubt Zugriff von überall
- `protected`
  - teilweiser Zugriff von außen ($\rightsquigarrow$ Vererbung)

Beispiel 1/2

```
 1  class Dreieck {
 2  private:
 3    double x[2];
 4    double y[2];
 5    double z[2];
 6
 7  public:
 8    void setX(double, double);
 9    void setY(double, double);
10    void setZ(double, double);
11    double flaeche();
12  };
```

- Dreieck in $\mathbb{R}^2$ mit Eckpunkten `x,y,z`
- Benutzer kann Daten `x,y,z` nicht lesen + schreiben
  - `get/set` Funktionen in `public`-Bereich einbauen
- Benutzer kann Methode `flaeche` aufrufen
- Benutzer muss nicht wissen, wie Daten intern verwaltet werden
  - kann interne Datenstruktur später verändern
  - z.B. Dreieck kann auch durch einen Punkt und zwei Vektoren abgespeichert werden
- Zeile 2: `private:` kann weggelassen werden
  - alle Members/Methoden standardmäßig `private`
- Zeile 7: ab `public:` ist Zugriff frei
  - d.h. Zeile 8 und folgende

Beispiel 2/2

```
 1  class Dreieck {
 2  private:
 3    double x[2];
 4    double y[2];
 5    double z[2];
 6
 7  public:
 8    void setX(double, double);
 9    void setY(double, double);
10    void setZ(double, double);
11    double getFlaeche();
12  };
13
14  int main() {
15    Dreieck tri;
16
17    tri.x[0] = 1.0; // Syntax-Fehler!
18
19    return 0;
20  }
```

- Zeile 8–11: Deklaration von `public`-Methoden
- Zeile 15: Objekt `tri` vom Typ Dreieck deklarieren
- Zeile 17: Zugriff auf `private`-Member
- Beim Kompilieren tritt Fehler auf

```
dreieck2.cpp:17: error: 'x' is a private
member of 'Dreieck'
dreieck2.cpp:3: note: declared private here
```

- daher: get/set-Funktionen, falls nötig

Methoden implementieren 1/2

```
 1  #include <cmath>
 2
 3  class Dreieck {
 4  private:
 5    double x[2];
 6    double y[2];
 7    double z[2];
 8  public:
 9    void setX(double, double);
10    void setY(double, double);
11    void setZ(double, double);
12    double getFlaeche();
13  };
14
15  double Dreieck::getFlaeche() {
16    return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
17                     - (z[0]-x[0])*(y[1]-x[1]) );
18  }
```

- Implementierung wie bei anderen Funktionen
  - direkter Zugriff auf Members der Klasse
- Signatur: `type ClassName::fctName(input)`
  - `type` ist Rückgabewert (`void`, `double` etc.)
  - `input` = Übergabeparameter wie in C
- Wichtig: `ClassName::` vor `fctName`
  - d.h. Methode `fctName` gehört zu `ClassName`
- Darf innerhalb von `ClassName::fctName` auf alle Members der Klasse direkt zugreifen (Zeile 16–17)
  - auch auf `private`-Members
- Zeile 1: Einbinden der `math.h` aus C

Methoden implementieren 2/2

```
 1  #include <cmath>
 2
 3  class Dreieck {
 4  private:
 5    double x[2];
 6    double y[2];
 7    double z[2];
 8
 9  public:
10    void setX(double, double);
11    void setY(double, double);
12    void setZ(double, double);
13    double getFlaeche();
14  };
15
16  void Dreieck::setX(double x0, double x1) {
17    x[0] = x0; x[1] = x1;
18  }
19
20  void Dreieck::setY(double y0, double y1) {
21    y[0] = y0; y[1] = y1;
22  }
23
24  void Dreieck::setZ(double z0, double z1) {
25    z[0] = z0; z[1] = z1;
26  }
27
28  double Dreieck::getFlaeche() {
29    return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
30                     - (z[0]-x[0])*(y[1]-x[1]) );
31  }
```


Methoden aufrufen

```
 1  #include <iostream>
 2  #include "dreieck4.cpp" // Code von letzter Folie
 3
 4  using std::cout;
 5  using std::endl;
 6
 7  // void Dreieck::setX(double x0, double x1)
 8  // void Dreieck::setY(double y0, double y1)
 9  // void Dreieck::setZ(double z0, double z1)
10
11  // double Dreieck::getFlaeche() {
12  //     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
13  //                      - (z[0]-x[0])*(y[1]-x[1]) );
14  // }
15
16  int main() {
17    Dreieck tri;
18    tri.setX(0.0,0.0);
19    tri.setY(1.0,0.0);
20    tri.setZ(0.0,1.0);
21    cout << "Flaeche= " << tri.getFlaeche() << endl;
22    return 0;
23  }
```

- `getFlaeche` agiert auf den Members von `tri`
  - d.h. `x[0]` in Implementierung entspricht `tri.x[0]`
- **Output:** `Flaeche= 0.5`

Methoden direkt implementieren

```
1   #include <cmath>
2
3   class Dreieck {
4   private:
5     double x[2];
6     double y[2];
7     double z[2];
8
9   public:
10    void setX(double x0, double x1) {
11      x[0] = x0;
12      x[1] = x1;
13    };
14    void setY(double y0, double y1) {
15      y[0] = y0;
16      y[1] = y1;
17    };
18    void setZ(double z0, double z1) {
19      z[0] = z0;
20      z[1] = z1;
21    };
22    double getFlaeche() {
23      return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
24                     - (z[0]-x[0])*(y[1]-x[1]) );
25    }
26  };
```

- kann Methoden auch in Klasse implementieren
- ist aber unübersichtlicher ⇒ besser nicht!

Klasse `string`

```
1   #include <iostream>
2   #include <string>
3   #include <cstdio>
4   using std::cout;
5   using std::string;
6
7   int main() {
8     string str1 = "Hallo";
9     string str2 = "Welt";
10    string str3 = str1 + " " + str2;
11
12    cout << str3 << "! ";
13    str3.replace(6,4, "Peter");
14    cout << str3 << "! ";
15
16    printf("%s?\n",str3.c_str());
17
18    return 0;
19  }
```

- **Output:** `Hallo Welt! Hallo Peter! Hallo Peter?`
- Zeile 3: Einbinden der `stdio.h` aus C
- Wichtig: `string` ≠ `char*`, sondern mächtiger!
- liefert eine Reihe nützlicher Methoden
  - '+' zum Zusammenfügen
  - `replace` zum Ersetzen von Teilstrings
  - `length` zum Auslesen der Länge u.v.m.
  - `c_str` liefert Pointer auf `char*`
- http://www.cplusplus.com/reference/string/string/

Strukturen

```
1   struct myStruct {
2     double x[2];
3     double y[2];
4     double z[2];
5   };
6
7   class myClass {
8     double x[2];
9     double y[2];
10    double z[2];
11  };
12
13  class myStructClass {
14  public:
15    double x[2];
16    double y[2];
17    double z[2];
18  };
19
20  int main() {
21    myStruct var1;
22    myClass var2;
23    myStructClass var3;
24
25    var1.x[0] = 0;
26    var2.x[0] = 0; // Syntax-Fehler
27    var3.x[0] = 0;
28
29    return 0;
30  }
```

- Strukturen = Klassen, wobei alle Members `public`
  - d.h. `myStruct` = `myStructClass`
- besser direkt `class` verwenden

Naive Fehlerkontrolle

- Wozu Zugriffskontrolle?
- Vermeidung von Laufzeitfehlern!
- bewusster Fehlerabbruch

- `assert`
- `#include <cassert>`

Wozu Zugriffskontrolle? 1/2

- Fakt ist: alle Programmierer machen Fehler
  - Code läuft beim ersten Mal nie richtig
- Großteil der Entwicklungszeit geht in Fehlersuche
- "Profis" unterscheiden sich von "Anfängern" im Wesentlichen durch effizientere Fehlersuche
- Syntax-Fehler sind leicht einzugrenzen
  - es steht Zeilennummer dabei (Compiler!)
- Laufzeitfehler sind viel schwieriger zu finden
  - Programm läuft, tut aber nicht das Richtige
  - manchmal fällt der Fehler ewig nicht auf
    $\Rightarrow$ sehr schlecht
- Möglichst viele Fehler bewusst abfangen!
  - Funktions-Input auf Konsistenz prüfen!
    - Fehler-Abbruch, falls inkonsistent!
  - Zugriff kontrollieren mittels `get` und `set`
    - reine Daten sollten immer `private` sein
    - Benutzer kann/darf Daten nicht verpfuschen!

Wozu Zugriffskontrolle? 2/2

```
 1  class Bruch {
 2  public:
 3    int zaehler;
 4    unsigned int nenner;
 5  };
 6
 7  int main() {
 8    Bruch meinBruch;
 9    meinBruch.zaehler = -1000;
10    meinBruch.nenner = 0;
11
12    return 0;
13  }
```

- Wie sinnvolle Werte sicherstellen? (Zeile 10)
  - mögliche Fehlerquellen direkt ausschließen
  - Aufgabe des Programmierers
    - Programm bestimmt, was Nutzer darf
  - Laufzeitfehler möglichst früh unterbrechen
- Lösung: `get` und `set` Funktionen für Memberdaten `zaehler, nenner` einbauen
- Lösung: Verwendung von C-Bibliothek `assert.h`
  - Einbinden `#include <cassert>`
  - `assert(condition);` liefert Fehlerabbruch, falls `condition` falsch
  - mit Ausgabe der Zeilennummer im Source-Code

C-Bibliothek assert.h

```
 1  #include <iostream>
 2  #include <cassert>
 3  using std::cout;
 4
 5  class Bruch {
 6  private:
 7    int zaehler;
 8    unsigned int nenner;
 9  public:
10    int getZaehler() { return zaehler; };
11    unsigned int getNenner() { return nenner; };
12    void setZaehler(int z) { zaehler = z; };
13    void setNenner(unsigned int n) {
14      assert(n>0);
15      nenner = n;
16    }
17    void print() {
18      cout << zaehler << "/" << nenner << "\n";
19    }
20  };
21
22  int main() {
23    Bruch x;
24    x.setZaehler(1);
25    x.setNenner(3);
26    x.print();
27    x.setNenner(0);
28    x.print();
29    return 0;
30  }
```

- Output:

```
1/3
Assertion failed: (n>0), function setNenner,
file assert.cpp, line 14.
```


File-Konventionen

- Aufteilen von Source-Code auf mehrere Files
- Precompiler, Compiler, Linker
- Objekt-Code
- `name.hpp`
- `name.cpp`

- `g++ -c`
- `make`

Aufteilen von Source-Code

- längere Source-Codes auf mehrere Files aufteilen
- Vorteil:
  - übersichtlicher
  - Bildung von Bibliotheken
    - Wiederverwendung von alten Codes
    - vermeidet Fehler
- `g++ name1.cpp name2.cpp ...`
  - erstellt *ein* Exe aus mehreren Source-Codes
  - Reihenfolge der Codes nicht wichtig
  - analog zu `g++ all.cpp`
    - wenn `all.cpp` ganzen Source-Code enthält
  - insb. Funktionsnamen müssen eindeutig sein
  - `int main()` darf nur 1x vorkommen
- analog für C und `gcc`

Precompiler, Compiler & Linker

- Beim Kompilieren von Source-Code werden mehrere Stufen durchlaufen:

(1) Preprocessor-Befehle ausführen, z.B. `#include`

(2) Compiler erstellt Objekt-Code

(3) Objekt-Code aus Bibliotheken wird hinzugefügt

(4) Linker ersetzt symbolische Namen im Objekt-Code durch Adressen und erzeugt Executable

- Bibliotheken = vorkompilierter Objekt-Code
  - plus zugehöriges Header-File
- Standard-Linker in Unix ist `ld`
- Nur Schritt (3) fertig, d.h. Objekt-Code erzeugen
  - `g++ -c name.cpp` erzeugt Objekt-Code `name.o`
- Objekt-Code händisch hinzufügen + kompilieren
  - `g++ name.cpp bib1.o bib2.o ...`
  - `g++ name.o bib1.o bib2.o ...`
  - Reihenfolge + Anzahl der Objekt-Codes ist egal
- Ziel: selbst Bibliotheken erstellen
  - spart ggf. Zeit beim Kompilieren
  - vermeidet Fehler

File-Konventionen

- Jedes C++ Programm besteht aus mehreren Files
  - C++ File für das Hauptprogramm `main.cpp`
  - Konvention: pro verwendeter Klasse zusätzlich
    - Header-File `myClass.hpp`
    - Source-File `myClass.cpp`
- Header-File `myClass.hpp` besteht aus
  - `#include` aller benötigten Bibliotheken
  - Definition der Klasse
  - nur Signaturen der Methoden (ohne Rumpf)
  - Kommentare zu den Methoden
    - Was tut eine Methode?
    - Was ist Input? Was ist Output?
- Source-File `myClass.cpp` enthält Source-Code der Methoden
- Warum Code auf mehrere Files aufteilen?
  - Übersichtlichkeit & Verständlichkeit des Codes
  - Anlegen von Bibliotheken
- Header-File beginnt mit

  ```
  #ifndef MYCLASS
  #define MYCLASS
  ```

- Header-File endet mit

  ```
  #endif
  ```

- Dieses Vorgehen erlaubt mehrfache Einbindung!

dreieck.hpp

```
 1  #ifndef DREIECK
 2  #define DREIECK
 3
 4  #include <cmath>
 5
 6  // The class Dreieck stores a triangle in R2
 7
 8  class Dreieck {
 9  private:
10    // the coordinates of the nodes
11    double x[2];
12    double y[2];
13    double z[2];
14
15  public:
16    // define or change the nodes of a triangle,
17    // e.g., triangle.setX(x1,x2) writes the
18    // coordinates of the node x of the triangle.
19    void setX(double, double);
20    void setY(double, double);
21    void setZ(double, double);
22
23    // return the area of the triangle
24    double getFlaeche();
25  };
26
27  #endif
```

dreieck.cpp

```
1   #include "dreieck.hpp"
2
3   void Dreieck::setX(double x0, double x1) {
4       x[0] = x0; x[1] = x1;
5   }
6
7   void Dreieck::setY(double y0, double y1) {
8       y[0] = y0; y[1] = y1;
9   }
10
11  void Dreieck::setZ(double z0, double z1) {
12      z[0] = z0; z[1] = z1;
13  }
14
15  double Dreieck::getFlaeche() {
16      return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
17                     - (z[0]-x[0])*(y[1]-x[1]) );
18  }
```

- Erzeuge Objekt-Code aus Source (Option `-c`)
  - `g++ -c dreieck.cpp` liefert `dreieck.o`
- Kompilieren `g++ dreieck.cpp` liefert Fehler
  - Linker `ld` scheitert, da kein `main` vorhanden

```
Undefined symbols for architecture x86_64:
"_main", referenced from:
   implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
```


dreieck_main.cpp

```
1   #include <iostream>
2   #include "dreieck.hpp"
3
4   using std::cout;
5   using std::endl;
6
7   int main() {
8     Dreieck tri;
9     tri.setX(0.0,0.0);
10    tri.setY(1.0,0.0);
11    tri.setZ(0.0,1.0);
12    cout << "Flaeche= " << tri.getFlaeche() << endl;
13    return 0;
14  }
```

- Kompilieren mit `g++ dreieck_main.cpp dreieck.o`
  - erzeugt Objekt-Code aus `dreieck_main.cpp`
  - bindet zusätzlichen Objekt-Code `dreieck.o` ein
  - linkt den Code inkl. Standardbibliotheken

Statische Bibliotheken und make

```
1   exe : dreieck_main.o dreieck.o
2           g++ -o exe dreieck_main.o dreieck.o
3
4   dreieck_main.o : dreieck_main.cpp dreieck.hpp
5           g++ -c dreieck_main.cpp
6
7   dreieck.o : dreieck.cpp dreieck.hpp
8           g++ -c dreieck.cpp
```

- UNIX-Befehl `make` erlaubt Abhängigkeiten von Code automatisch zu handeln
  - Automatisierung spart Zeit für Kompilieren
  - Nur wenn Source-Code geändert wurde, wird neuer Objekt-Code erzeugt und abhängiger Code wird neu kompiliert
- Aufruf `make` befolgt Steuerdatei `Makefile`
- Aufruf `make -f filename` befolgt `filename`
- Datei zeigt Abhängigkeiten und Befehlen, z.B.
  - Zeile 1 = Abhängigkeit (nicht eingerückt!)
    - Datei `exe` hängt ab von ...
  - Zeile 2 = Befehl (eine Tabulator-Einrückung!)
    - Falls `exe` älter ist als Abhängigkeiten, wird Befehl ausgeführt (und nur dann!)
- mehr zu `make` in Schmaranz C-Buch, Kapitel 15

Konstruktor & Destruktor

- Konstruktor
- Destruktor
- Überladen (Einführung)
- optionaler Input & Default-Parameter
- Schachtelung von Klassen

- `this`
- `ClassName(...)`
- `~ClassName()`
- Operator :
- `for(int j=0; j<dim; ++j) { ... }`

Konstruktor & Destruktor

- Konstruktor = Aufruf automatisch bei Deklaration
  - kann Initialisierung übernehmen
  - kann verschiedene Aufrufe haben, z.B.
    - Anlegen eines Vektors der Länge Null
    - Anlegen eines Vektors $x \in \mathbb{R}^N$ und Initialisieren mit Null
    - Anlegen eines Vektors $x \in \mathbb{R}^N$ und Initialisieren mit gegebenem Wert
  - formal: `className(input)`
    - kein Output, eventuell Input
    - versch. Konstruktoren haben versch. Input
    - Standardkonstruktor: `className()`
- Destruktor = Aufruf automat. bei Lifetime-Ende
  - Freigabe von dynamischem Speicher
  - es gibt nur Standarddestruktor: `~className()`
    - kein Input, kein Output
- Überladen = ein Methoden-Name hat mehrere verschiedene Signaturen und Funktionalitäten
  - wichtig: Signaturen eindeutig verschieden!
  - später mehr!

Konstruktor: Ein Beispiel

```
1  #include <iostream>
2  #include <string>
3  using std::cout;
4  using std::string;
5
6  class Student {
7  private:
8    string lastname;
9    int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   };
14   Student(string name, int id) {
15     lastname = name;
16     student_id = id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   };
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- Konstruktor hat keinen Rückgabewert (Z. 11, 14)
  - Name `className(input)`
  - Standardkonstr. `Student()` ohne Input (Z. 11)
- Output

```
Student generiert
Student (Praetorius, 12345678) angemeldet
```


Namenskonflikt & Pointer `this`

```
1  #include <iostream>
2  #include <string>
3  using std::cout;
4  using std::string;
5
6  class Student {
7  private:
8    string lastname;
9    int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   };
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   };
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- `this` gibt Pointer auf das aktuelle Objekt
  - `this->` gibt Zugriff auf Member des akt. Objekts
- Namenskonflikt in Konstruktor (Zeile 14)
  - Input-Variable heißen wie Members der Klasse
  - Zeile 14–16: Lösen des Konflikts mittels `this->`

Destruktor: Ein Beispiel

```
1  #include <iostream>
2  #include <string>
3  using std::cout;
4  using std::string;
5
6  class Student {
7  private:
8    string lastname;
9    int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   };
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   };
20   ~Student() {
21     cout << "Student (" << lastname << ", ";
22     cout << student_id << ") abgemeldet\n";
23   }
24 };
25
26 int main() {
27   Student var("Praetorius",12345678);
28   return 0;
29 }
```

- Zeile 20–23: Destruktor (ohne Input + Output)
- Output

```
Student (Praetorius, 12345678) angemeldet
Student (Praetorius, 12345678) abgemeldet
```

Methoden: Kurzschreibweise

```
 1  #include <iostream>
 2  #include <string>
 3  using std::cout;
 4  using std::string;
 5
 6  class Student {
 7  private:
 8    string lastname;
 9    int student_id;
10  public:
11    Student() : lastname("nobody"), student_id(0) {
12      cout << "Student generiert\n";
13    };
14    Student(string name, int id) :
15      lastname(name), student_id(id) {
16        cout << "Student (" << lastname << ", ";
17        cout << student_id << ") angemeldet\n";
18    };
19    ~Student() {
20      cout << "Student (" << lastname << ", ";
21      cout << student_id << ") abgemeldet\n";
22    }
23  };
24
25  int main() {
26    Student test;
27    return 0;
28  }
```

- Zeile 11, 14–15: Kurzschreibweise für Zuweisung
  - ruft entsprechende Konstruktoren auf
  - eher schlecht lesbar
- Output

```
Student generiert
Student (nobody, 0) abgemeldet
```


Noch ein Beispiel

```
 1  #include <iostream>
 2  #include <string>
 3  using std::cout;
 4  using std::string;
 5
 6  class Test {
 7  private:
 8    string name;
 9  public:
10    void print() {
11      cout << "Name " << name << "\n";
12    };
13    Test() : name("Standard") { print(); };
14    Test(string n) : name(n)  { print(); };
15    ~Test() {
16      cout << "Loesche " << name << "\n";
17    };
18  };
19
20  int main() {
21    Test t1("Objekt1");
22    {
23      Test t2;
24      Test t3("Objekt3");
25    }
26    cout << "Blockende" << "\n";
27    return 0;
28  }
```

- Ausgabe:

```
Name Objekt1
Name Standard
Name Objekt3
Loesche Objekt3
Loesche Standard
Blockende
Loesche Objekt1
```


Schachtelung von Klassen

```
 1  #include <iostream>
 2  using std::cout;
 3  using std::endl;
 4
 5  class Class1 {
 6  public:
 7    Class1() { cout << "Konstr Class1" << endl; };
 8    ~Class1() { cout << "Destr Class1" << endl; };
 9  };
10
11  class Class2 {
12    Class1 obj1;
13  public:
14     Class2() { cout << "Konstr Class2" << endl; };
15     ~Class2() { cout << "Destr Class2" << endl; };
16  };
17
18  int main() {
19    Class2 obj2;
20    return 0;
21  }
```

- Klassen können geschachtelt werden
  - Standardkonstr./-destr. automatisch aufgerufen
  - Konstruktoren der Member zuerst
  - Destruktoren der Member zuletzt
- Ausgabe:

```
Konstr Class1
Konstr Class2
Destr Class2
Destr Class1
```


vector.hpp

```
 1  #ifndef VECTOR
 2  #define VECTOR
 3
 4  #include <cmath>
 5  #include <cstdlib>
 6  #include <cassert>
 7
 8  // The class Vector stores vectors in Rd
 9
10  class Vector {
11  private:
12    // dimension of the vector
13    int dim;
14    // dynamic coefficient vector
15    double* coeff;
16
17  public:
18    // constructors and destructor
19    Vector();
20    Vector(int dim, double init);
21    ~Vector();
22
23    // return vector dimension
24    int size();
25
26    // read and write vector coefficients
27    void set(int k, double value);
28    double get(int k);
29
30    // compute Euclidean norm
31    double norm();
32  };
33
34  #endif
```

vector.cpp 1/2

```
1   #include "vector.hpp"
2   #include <iostream>
3   using std::cout;
4
5   Vector::Vector() {
6     dim = 0;
7     coeff = NULL;
8   }
9   Vector::Vector(int dim, double init = 0) {
10      int j = 0;
11      this->dim = dim;
12      coeff = (double*) malloc(dim*sizeof(double));
13      for (j=0; j<dim; ++j) {
14          coeff[j] = init;
15      }
16  }
17  Vector::~Vector() {
18    if (dim > 0) {
19      free(coeff);
20    }
21    // just for demonstration purposes
22    cout << "free vector, length " << dim << "\n";
23  }
```

- erstellt drei Konstruktoren (Zeile 5, Zeile 9)
  - Standardkonstruktor (Zeile 5)
  - Deklaration `Vector var(dim,init);`
  - Deklaration `Vector var(dim);` mit `init` = Null
  - optionaler Input durch Default-Parameter (Z. 9)
    - nur in `vector.cpp`, nicht in `vector.hpp` angeben
- **ohne Destruktor:** nur Speicher von Pointer frei
- **Achtung:** `g++` erfordert expliziten Type Cast bei `malloc` (Zeile 12)

vector.cpp 2/2

```
24  int Vector::size() {
25    return dim;
26  }
27
28  void Vector::set(int k, double value) {
29    assert(k>=0 && k<dim);
30    coeff[k] = value;
31  }
32
33  double Vector::get(int k) {
34    assert(k>=0 && k<dim);
35    return coeff[k];
36  }
37
38  double Vector::norm() {
39    double norm = 0;
40    int j = 0;
41    for (j=0; j<dim; ++j) {
42      norm = norm + coeff[j]*coeff[j];
43    }
44    return sqrt(norm);
45  }
```

- kontrollierter Zugriff auf Koeffizienten (Z. 29, 34)
- in C++ darf man Variablen überall deklarieren
  - im ursprünglichen C nur am Blockanfang
  - ist kein guter Stil, da unübersichtlich
    - C-Stil möglichst beibehalten! Code wartbarer!
- **vernünftig:** `for (int j=0; j<dim; ++j) { ... }`
  - für lokale Zählvariablen (in Zeile 41–42)

main.cpp

```
1   #include "vector.hpp"
2   #include <iostream>
3
4   using std::cout;
5
6   int main() {
7     Vector vector1;
8     Vector vector2(20);
9     Vector vector3(10,4);
10    cout << "Norm = " << vector1.norm() << "\n";
11    cout << "Norm = " << vector2.norm() << "\n";
12    cout << "Norm = " << vector3.norm() << "\n";
13
14    return 0;
15  }
```

- Kompilieren mit

```
g++ -c vector.cpp
g++ main.cpp vector.o
```

- Output:

```
Norm = 0
Norm = 0
Norm = 12.6491
free vector, length 10
free vector, length 20
free vector, length 0
```