C++

- Was ist C++
- Wie erstellt man ein C++ Programm?
- `Hello World!` mit C++

- `main`
- `cout`, `cin`, `endl`
- `using std::`
- Scope-Operator `::`
- Operatoren <<, >>
- `#include <iostream>`

Was ist C++

- Weiterentwicklung von C
  - Entwicklung ab 1979 bei AT&T
  - Entwickler: Bjarne Stroustrup
- C++ ist abwärtskompatibel zu C
  - keine Syntaxkorrektur
  - aber: stärkere Zugriffskontrolle bei "Strukturen"
    - Datenkapselung
- Compiler:
  - frei verfügbar in Unix/Mac: `g++`
  - Microsoft Visual C++ Compiler
  - Borland C++ Compiler

Objektorientierte Programmiersprache

- C++ ist objektorientiertes C
- Objekt = Zusammenfassung von Daten + Fktn.
  - Funktionalität hängt von Daten ab
  - vgl. Multiplikation für Skalar, Vektor, Matrix
- Befehlsreferenzen
  - `http://en.cppreference.com/w/cpp`
  - `http://www.cplusplus.com`

Wie erstellt man ein C++ Prg?

- Starte Editor Emacs aus einer Shell mit `emacs &`
  - Die wichtigsten Tastenkombinationen:
    - `C-x C-f` = Datei öffnen
    - `C-x C-s` = Datei speichern
    - `C-x C-c` = Emacs beenden
- Öffne eine (ggf. neue) Datei `name.cpp`
  - Endung `.cpp` ist Kennung für C++ Programm
- Die ersten beiden Punkte kann man auch simultan erledigen mittels `emacs name.cpp &`
- Schreibe *Source-Code* (= C++ Programm)
- Abspeichern mittels `C-x C-s` nicht vergessen
- Compilieren z.B. mit `g++ name.cpp`
- Falls Code fehlerfrei, erhält man *Executable* `a.out`
  - unter Windows: `a.exe`
- Diese wird durch `a.out` bzw. `./a.out` gestartet
- Compilieren mit `g++ name.cpp -o output` erzeugt Executable `output` statt `a.out`

Hello World

```
1  #include <iostream>
2
3  int main() {
4    std::cout << "Hello World!\n";
5    return 0;
6  }
```

- C++ Bibliothek für Ein- und Ausgabe ist iostream
- `main` hat zwingend Rückgabewert `int`
  - `int main()`
  - `int main(int argc, char* argv[])`
    - insbesondere `return 0;` am Programmende
- Scope-Operator `::` gibt *Name Space* an
  - alle Fktn. der Standardbibliotheken haben `std`
- `std::cout` ist die Standard-Ausgabe (= Shell)
  - Operator `<<` übergibt rechtes Argument an `cout`

```
1  #include <iostream>
2  using std::cout;
3
4  int main() {
5    cout << "Hello World!\n";
6    return 0;
7  }
```

- `using std::cout;`
  - `cout` gehört zum *Name Space* `std`
  - darf im Folgenden abkürzen `cout` statt `std::out`

Shell-Input für Main

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   int main(int argc, char* argv[]) {
6     int j = 0;
7     cout << "This is " << argv[0] << endl;
8     cout << "got " << argc-1 << " inputs:" << endl;
9     for (j=1; j<argc; ++j) {
10      cout << j << ": " << argv[j] << endl;
11    }
12    return 0;
13  }
```

- << arbeitet mit verschiedenen Typen
- kann mehrfache Ausgabe machen <<
- endl ersetzt "\n"
- Shell übergibt Input als C-Strings an Programm
  - Parameter jeweils durch Leerzeichen getrennt
  - argc = Anzahl der Parameter
  - argv = Vektor der Input-Strings
  - argv[0] = Programmname
  - d.h. argc−1 echte Input-Parameter
- Output für Shell-Eingabe ./a.out Hello World!

```
This is ./a.out
got 2 inputs:
1: Hello
2: World!
```


Eingabe / Ausgabe

```
1   #include <iostream>
2   using std::cin;
3   using std::cout;
4   using std::endl;
5
6   int main() {
7     int x = 0;
8     double y = 0;
9     double z = 0;
10
11    cout << "Geben Sie einen Integer ein: ";
12    cin >> x;
13    cout << "Geben Sie zwei Double ein: ";
14    cin >> y >> z;
15
16    cout << x << " * " << y << " / " << z;
17    cout << " = " << x*y/z << endl;
18
19    return 0;
20  }
```

- std::cin ist die Standard-Eingabe (= Tastatur)
  - Operator >> schreibt Input in Variable rechts
- Beispielhafte Eingabe / Ausgabe:

```
Geben Sie einen Integer ein: 2
Geben Sie zwei Double ein: 3.6 1.3
2 * 3.6 / 1.3 = 5.53846
```

- cin / out gleichwertig mit printf / scanf in C
  - aber leichter zu bedienen
  - keine Platzhalter + Pointer

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
  - sogenannte *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 leichter verändern, falls das nötig wird
  - 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 $\Rightarrow$ 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` $\neq$ `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?
    - insb. Default-Parameter + optionaler Input
- 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 Befehle, 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_first.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 = 0);
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_first.cpp 1/2

```
 1  #include "vector_first.hpp"
 2  #include <iostream>
 3  using std::cout;
 4
 5  Vector::Vector() {
 6    dim = 0;
 7    coeff = (double*) NULL;
 8  }
 9  Vector::Vector(int dim, double init) {
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 = 0`
  - optionaler Input durch Default-Parameter (Z. 9)
    - wird in `vector.hpp` angegeben (letzte Folie!)
- **ohne Destruktor:** nur Speicher von Pointer frei
- **Achtung:** `g++` erfordert expliziten Type Cast bei `malloc` (Zeile 12)

vector_first.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_first.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_first.cpp
g++ main.cpp vector_first.o
```

- Output:

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


Referenzen

- Definition
- Unterschied zwischen Referenz und Pointer
- direktes Call by Reference
- Referenzen als Funktions-Output

- `type&`

Was ist eine Referenz?

```
 1  #include <iostream>
 2  using std::cout;
 3  using std::endl;
 4
 5  int main() {
 6    int var = 5;
 7    int &ref = var;
 8
 9    cout << "var = " << var << endl;
10    cout << "ref = " << ref << endl;
11    ref = 7;
12    cout << "var = " << var << endl;
13    cout << "ref = " << ref << endl;
14
15    return 0;
16  }
```

- Referenzen sind Aliasnamen für Objekte/Variablen
- `type& ref = var`
  - erzeugt eine Referenz `ref` zu `var`
  - `var` muss vom Datentyp `type` sein
  - Referenz muss bei Definition initialisiert werden!
- nicht verwechselbar mit Address-Of-Operator
  - `type&` ist Referenz
  - `&var` liefert Speicheradresse von `var`
- Output:

```
var = 5
ref = 5
var = 7
ref = 7
```


Address-Of-Operator

```
 1  #include <iostream>
 2  using std::cout;
 3  using std::endl;
 4
 5  int main() {
 6    int var = 5;
 7    int& ref = var;
 8
 9    cout << "var = " << var << endl;
10    cout << "ref = " << ref << endl;
11    cout << "Adresse von var = " << &var << endl;
12    cout << "Adresse von ref = " << &ref << endl;
13
14    return 0;
15  }
```

- muss: Deklaration + Init. bei Referenzen (Zeile 6)
  - sind nur Alias-Name für denselben Speicher
  - d.h. `ref` und `var` haben dieselbe Adresse
- Output:

```
var = 5
ref = 5
Adresse von var = 0x7fff532e8b48
Adresse von ref = 0x7fff532e8b48
```

Funktionsargumente als Pointer

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   void swap(int* px, int* py) {
6     int tmp = *px;
7     *px = *py;
8     *py = tmp;
9   }
10
11  int main() {
12    int x = 5;
13    int y = 10;
14    cout << "x = " << x << ", y = " << y << endl;
15    swap(&x, &y);
16    cout << "x = " << x << ", y = " << y << endl;
17    return 0;
18  }
```

- Output:

  ```
  x = 5, y = 10
  x = 10, y = 5
  ```

- bereits bekannt aus C:
  - übergebe Adressen `&x`, `&y` mit Call-by-Value
  - lokale Variablen `px`, `py` vom Typ `int*`
  - Zugriff auf Speicherbereich von `x` durch Dereferenzieren `*px`
  - analog für `*py`
- Zeile 6–8: Vertauschen der Inhalte von `*px` und `*py`

Funktionsargumente als Referenz

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   void swap(int& rx, int& ry) {
6     int tmp;
7     tmp = rx;
8     rx = ry;
9     ry = tmp;
10  }
11
12  int main() {
13    int x = 5;
14    int y = 10;
15    cout << "x = " << x << ", y = " << y << endl;
16    swap(x, y);
17    cout << "x = " << x << ", y = " << y << endl;
18    return 0;
19  }
```

- Output:

  ```
  x = 5, y = 10
  x = 10, y = 5
  ```

- echtes Call-by-Reference in C++
  - Funktion kriegt als Input Referenzen
  - Syntax: `type fctName( ..., type& ref, ... )`
    - dieser Input wird als Referenz übergeben
- `rx` ist lokaler Name (Zeile 5–10) für den Speicherbereich von `x` (Zeile 13–18)
- analog für `ry` und `y`

Referenzen vs. Pointer

- Referenzen sind Aliasnamen für Variablen
  - müssen bei Deklaration initialisiert werden
  - kann Referenzen nicht nachträglich zuordnen!
- keine vollständige Alternative zu Pointern
  - keine Mehrfachzuweisung
  - kein dynamischer Speicher möglich
  - keine Felder von Referenzen möglich
  - Referenzen dürfen nicht `NULL` sein
- **Achtung:** Syntax verschleiert Programmablauf
  - bei Funktionsaufruf nicht klar, ob Call by Value oder Call by Reference
  - anfällig für Laufzeitfehler, wenn Funktion Daten ändert, aber Hauptprogramm das nicht weiß
  - passiert bei Pointer nicht
- **Wann Call by Reference sinnvoll?**
  - falls Input-Daten umfangreich
    - denn Call by Value kopiert Daten
  - dann Funktionsaufruf billiger

Refs als Funktions-Output 1/3

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   int& fct() {
6     int x = 4711;
7     return x;
8   }
9
10  int main() {
11    int var = fct();
12    cout << "var = " << var << endl;
13
14    return 0;
15  }
```

- Referenzen können Output von Funktionen sein
  - sinnvoll bei Objekten (später!)
- wie bei Pointern auf Lifetime achten!
  - Referenz wird zurückgegeben (Zeile 7)
  - aber Speicher wird freigegeben, da Blockende!
- Compiler erzeugt Warnung

  ```
  reference_output.cpp:7: warning: reference to
  stack memory associated with local variable
  'x' returned
  ```

Refs als Funktions-Output 2/3

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   class Demo {
6   private:
7     int val;
8   public:
9     Demo(int input) {
10      val = input;
11    }
12    int getContent() {
13      return val;
14    }
15  };
16
17  int main() {
18    Demo var(10);
19    int x = var.getContent();
20    x = 1;
21    cout << "x = " << x << ", ";
22    cout << "val = " << var.getContent() << endl;
23    return 0;
24  }
```

- Output:

  ```
  x = 1, content = 10
  ```

- Auf Folie nichts Neues!
  - nur Motivation der folgenden Folie

Refs als Funktions-Output 3/3

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   class Demo {
6   private:
7     int val;
8   public:
9     Demo(int input) {
10      val = input;
11    }
12    int& getContent() {
13      return val;
14    }
15  };
16
17  int main() {
18    Demo var(10);
19    int& x = var.getContent();
20    x = 1;
21    cout << "x = " << x << ", ";
22    cout << "val = " << var.getContent() << endl;
23    return 0;
24  }
```

- Output:

  ```
  x = 1, content = 1
  ```

- Achtung: `private` Member wurde geändert
  - Das will man eigentlich nicht!
  - Das kann Laufzeitfehler produzieren!
- Beachte: Code von `getContent` gleich
  - nur andere Signatur

Schlüsselwort `const`

- Konstanten definieren
- read-only Referenzen

- `const`
- `const int*, int const*, int* const`
- `const int&`

elementare Konstanten

- möglich über `#define CONST wert`
  - einfache Textersetzung `CONST` durch `wert`
  - fehleranfällig & kryptische Fehlermeldung
    - falls `wert` Syntax-Fehler erzeugt
  - Konvention: Konstantennamen groß schreiben
- besser als konstante Variable
  - z.B. `const int var = wert;`
  - z.B. `int const var = wert;`
    - beide Varianten haben dieselbe Bedeutung!
  - wird als Variable angelegt, aber Compiler verhindert Schreiben
  - zwingend Initialisierung bei Deklaration
- **Achtung** bei Pointern
  - `const int* ptr` ist Pointer auf `const int`
  - `int const* ptr` ist Pointer auf `const int`
    - beide Varianten haben dieselbe Bedeutung!
  - `int* const ptr` ist konstanter Pointer auf `int`

Beispiel 1/2

```
1   int main() {
2     const double var = 5;
3     var = 7;
4     return 0;
5   }
```

- Syntax-Fehler beim Kompilieren:

```
const.cpp:3: error: read-only variable is not
assignable
```

```
1   int main() {
2     const double var = 5;
3     double tmp  = 0;
4     const double* ptr = &var;
5     ptr = &tmp;
6     *ptr = 7;
7     return 0;
8   }
```

- Syntax-Fehler beim Kompilieren:

```
const_pointer.cpp:6: error: read-only
variable is not assignable
```


Beispiel 2/2

```
1   int main() {
2     const double var = 5;
3     double tmp  = 0;
4     double* const ptr = &var;
5     ptr = &tmp;
6     *ptr = 7;
7     return 0;
8   }
```

- Syntax-Fehler beim Kompilieren:

```
const_pointer2.cpp:4:14: error: cannot
initialize a variable of type 'double *const'
with an rvalue of type 'const double *'
```

  - Der Pointer `ptr` hat falschen Typ (Zeile 4)

```
1   int main() {
2     const double var = 5;
3     double tmp  = 0;
4     const double* const ptr = &var;
5     ptr = &tmp;
6     *ptr = 7;
7     return 0;
8   }
```

- zwei Syntax-Fehler beim Kompilieren:

```
const_pointer3.cpp:5: error: read-only
variable is not assignable
const_pointer3.cpp:6: error: read-only
variable is not assignable
```

  - Zuweisung auf Pointer `ptr` (Zeile 5)
  - Dereferenzieren und Schreiben (Zeile 6)

Read-Only Referenzen

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   int main() {
6     double var = 5;
7     double& ref = var;
8     const double& cref = var;
9     cout << "var = " << var << ", ";
10    cout << "ref = " << ref << ", ";
11    cout << "cref = " << cref << endl;
12    ref = 7;
13    cout << "var = " << var << ", ";
14    cout << "ref = " << ref << ", ";
15    cout << "cref = " << cref << endl;
16    // cref = 9;
17    return 0;
18  }
```

- `const type& cref`
  - deklariert konstante Referenz auf `type`
    - alternative Syntax: `type const& cref`
  - d.h. `cref` entspricht Variable vom Typ `const type`
  - Zugriff auf Referenz nur **lesend** möglich
- Output:

```
var = 5, ref = 5, cref = 5
var = 7, ref = 7, cref = 7
```

- Zeile `cref = 9;` würde Syntaxfehler liefern

```
error: read-only variable is not assignable
```


Read-Only Refs als Output 1/2

```
1   #include <iostream>
2   using std::cout;
3   using std::endl;
4
5   class Demo {
6   private:
7     int val;
8   public:
9     Demo(int input) {
10      val = input;
11    }
12    int& getContent() {
13      return val;
14    }
15  };
16
17  int main() {
18    Demo var(10);
19    int& x = var.getContent();
20    x = 1;
21    cout << "x = " << x << ", ";
22    cout << "val = " << var.getContent() << endl;
23    return 0;
24  }
```

- Output:

```
x = 1, content = 1
```

- Achtung: `private` Member wurde geändert

Read-Only Refs als Output 2/2

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  class Demo {
6  private:
7    int val;
8  public:
9    Demo(int input) { val = input; }
10   const int& getContent() { return val; }
11 };
12
13 int main() {
14   Demo var(10);
15   const int& x = var.getContent();
16   // x = 1;
17   cout << "x = " << x << ", ";
18   cout << "val = " << var.getContent() << endl;
19   return 0;
20 }
```

- Output:

  ```
  x = 10, content = 10
  ```

- Zuweisung `x = 1;` würde Syntax-Fehler liefern

  ```
  error: read-only variable is not assignable
  ```

- Deklaration `int& x = var.getContent();` würde Syntax-Fehler liefern

  ```
  error: binding of reference to type 'int' to
  a value of type 'const int' drops qualifiers
  ```

- sinnvoll, falls Read-Only Rückgabe sehr groß ist
  - z.B. Vektor, langer String etc.

Type Casting

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  double square(double& x) {
6      return x*x;
7  }
8
9  int main() {
10   const double var = 5;
11   cout << "var = " << var << ", ";
12   cout << "var*var = " << square(var) << endl;
13   return 0;
14 }
```

- `const type` ist stärker als `type`
  - kein Type Casting von `const type` auf `type`
- Syntax-Fehler beim Kompilieren:

  ```
  const_typecasting.cpp:12 error: no matching
  function for call to 'square'
  const_typecasting.cpp:5: note: candidate
  function not viable: 1st argument
  ('const double') would lose const qualifier
  ```

- Type Casting von `type` auf `const type` ist aber OK!
- bad-hack Lösung: Signatur ändern auf
  - `double square(const double& x)`

Read-Only Refs als Input 1/5

```
1  #include "vector_first.hpp"
2  #include <iostream>
3  #include <cassert>
4
5  using std::cout;
6
7  double product(const Vector& x, const Vector& y){
8    double sum = 0;
9    assert( x.size() == y.size() );
10   for (int j=0; j<x.size(); ++j) {
11     sum = sum + x.get(j)*y.get(j);
12   }
13   return sum;
14 }
15
16 int main() {
17   Vector x(100,1);
18   Vector y(100,2);
19   cout << "norm(x) = " << x.norm() << "\n";
20   cout << "norm(y) = " << y.norm() << "\n";
21   cout << "x.y = " << product(x,y) << "\n";
22   return 0;
23 }
```

- Vorteil: schlanker Daten-Input ohne Kopieren!
  - und: Daten können nicht verändert werden!
- Problem: Syntax-Fehler beim Kompilieren, z.B.

  ```
  const_vector.cpp:9: error: member function
  'size' not viable: 'this' argument has type
  'const Vector', but function is not marked
  const
  ```

  - d.h. Problem mit Methode `size`

Read-Only Refs als Input 2/5

```
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, double = 0);
21   ~Vector();
22
23   // return vector dimension
24   int size() const;
25
26   // read and write vector coefficients
27   void set(int k, double value);
28   double get(int k) const;
29
30   // compute Euclidean norm
31   double norm() const;
32 };
33
34 #endif
```

- Read-Only Methoden werden mit `const` markiert
  - `className::fct(... input ...) const`
- neue Syntax: Zeile 24, 28, 31

Read-Only Refs als Input 3/5

```
1   #include "vector_new.hpp"
2   #include <iostream>
3   using std::cout;
4
5   Vector::Vector() {
6     dim = 0;
7     coeff = (double*) NULL;
8   }
9   Vector::Vector(int dim, double init) {
10      this->dim = dim;
11      coeff = (double*) malloc(dim*sizeof(double));
12      for (int j=0; j<dim; ++j) {
13          coeff[j] = init;
14      }
15      // just for demonstration purposes
16      cout << "new vector, length " << dim << "\n";
17  }
18  Vector::~Vector() {
19    if (dim > 0) {
20      free(coeff);
21    }
22    // just for demonstration purposes
23    cout << "free vector, length " << dim << "\n";
24  }
```

- keine Änderungen!

Read-Only Refs als Input 4/5

```
24  int Vector::size() const {
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) const {
34    assert(k>=0 && k<dim);
35    return coeff[k];
36  }
37
38  double Vector::norm() const {
39    double norm = 0;
40    for (int j=0; j<dim; ++j) {
41      norm = norm + coeff[j]*coeff[j];
42    }
43    return sqrt(norm);
44  }
```

- geändert: Zeile 24, 33, 38

Read-Only Refs als Input 5/5

```
1   #include "vector_new.hpp"
2   #include <iostream>
3   #include <cassert>
4
5   using std::cout;
6
7   double product(const Vector& x, const Vector& y){
8     double sum = 0;
9     assert( x.size() == y.size() );
10    for (int j=0; j<x.size(); ++j) {
11      sum = sum + x.get(j)*y.get(j);
12    }
13    return sum;
14  }
15
16  int main() {
17    Vector x(100,1);
18    Vector y(100,2);
19    cout << "norm(x) = " << x.norm() << "\n";
20    cout << "norm(y) = " << y.norm() << "\n";
21    cout << "x.y = " << product(x,y) << "\n";
22    return 0;
23  }
```

- Vorteil: schlanker Daten-Input ohne Kopieren!
  - und: Daten können nicht verändert werden!
- Output:

```
new vector, length 100
new vector, length 100
norm(x) = 10
norm(y) = 20
x.y = 200
free vector, length 100
free vector, length 100
```


Zusammenfassung Syntax

- bei normalen Datentypen (nicht Pointer, Referenz)
  - `const int var`
  - `int const var`
    - dieselbe Bedeutung = Integer-Konstante
- bei Referenzen
  - `const int& ref` = Referenz auf `const int`
  - `int const& ref` = Referenz auf `const int`
- Achtung bei Pointern
  - `const int* ptr` = Pointer auf `const int`
  - `int const* ptr` = Pointer auf `const int`
  - `int* const ptr` = konstanter Pointer auf `int`
- bei Methoden, die nur Lese-Zugriff brauchen
  - `className::fct(... input ...) const`
  - kann Methode sonst nicht mit `const`-Refs nutzen
- sinnvoll, falls Rückgabe eine Referenz ist
  - `const int& fct(... input ...)`
  - lohnt sich nur bei großer Rückgabe, die nur gelesen wird

Überladen von Funktionen

- Default-Parameter & Optionaler Input
- Überladen & `const` bei Variablen
- Überladen & `const` bei Referenzen
- Überladen & `const` bei Methoden

Default-Parameter 1/2

```
1  void f(int x, int y, int z = 0);
2  void g(int x, int y = 0, int z = 0);
3  void h(int x = 0, int y = 0, int z = 0);
```

- kann Default-Werte für Input von Fktn. festlegen
  - durch `= wert`
  - der Input-Parameter ist dann optional
  - bekommt Default-Wert, falls nicht übergegeben
- Beispiel: Zeile 1 erlaubt Aufrufe
  - `f(x,y,z)`
  - `f(x,y)` und `z` bekommt implizit den Wert `z = 0`

```
1  void f(int x = 0, int y = 0, int z); // Fehler
2  void g(int x, int y = 0, int z);     // Fehler
3  void h(int x = 0, int y, int z = 0); // Fehler
```

- darf nur für hintere Parameter verwendet werden
  - d.h. nach optionalem Parameter darf kein obligatorischer Parameter mehr folgen

Default-Parameter 2/2

```
 1  #include <iostream>
 2  using std::cout;
 3
 4  void f(int x, int y = 0);
 5
 6  void f(int x, int y = 0) {
 7      cout << "x=" << x << ", y=" << y << "\n";
 8  }
 9
10  int main() {
11      f(1);
12      f(1,2);
13      return 0;
14  }
```

- Default-Parameter darf nur einmal gegeben werden
- Kompilieren liefert Syntax-Fehler:

  `default_wrong.cpp:6: error: redefinition of default argument`
- d.h. Default-Parameter entweder in Zeile 4 oder 6
- Output nach Korrektur:

  `x=1, y=0`

  `x=1, y=2`
- **Konvention:** bei der Forward Declaration
  - d.h. Default-Parameter werden in `hpp` festgelegt
- brauche bei Forward Decl. keine Variablennamen
  - `void f(int, int = 0);` in Zeile 4 ist OK

Überladen von Funktionen 1/2

```
1  void f(char*);
2  int f(char *);                // Syntax-Fehler
3  double f(char*, int = 0);     // Syntax-Fehler
4  double f(char*, double);
5  int f(char*, char*, int = 1);
```

- Mehrere Funktionen gleichen Namens möglich
  - wie bei Konstruktiven
  - unterscheiden sich durch ihre Signaturen
- Input muss Variante eindeutig festlegen
- bei Aufruf wird die richtige Variante ausgewählt
  - Compiler erkennt dies über Input-Parameter
  - Achtung mit implizitem Type Case
- Diesen Vorgang nennt man Überladen
- Reihenfolge bei der Deklaration ist unwichtig
  - d.h. kann Zeilen 1–5 beliebig permutieren
- Rückgabewerte können unterschiedlich sein
  - Also: unterschiedliche Output-Parameter und gleiche Input-Parameter geht nicht
    - Zeile 2: Syntax-Fehler, da Input gleich zu 1
    - Zeile 3: Syntax-Fehler, da optionaler Input
    - Zeile 4 + 5: OK

Überladen von Funktionen 2/2

```
 1  #include <iostream>
 2  using std::cout;
 3  using std::endl;
 4
 5  class Car {
 6  public:
 7    void drive();
 8    void drive(int km);
 9    void drive(int km, int h);
10  };
11
12  void Car::drive() {
13    cout << "10 km gefahren" << endl;
14  }
15
16  void Car::drive(int km) {
17    cout << km << " km gefahren" << endl;
18  }
19
20  void Car::drive(int km, int h) {
21    cout << km << " km gefahren in " << h
22      << " Stunde(n)" << endl;
23  }
24
25  int main() {
26    Car TestCar;
27    TestCar.drive();
28    TestCar.drive(35);
29    TestCar.drive(50,1);
30    return 0;
31  }
```

- Ausgabe:
```
10 km gefahren
35 km gefahren
50 km gefahren in 1 Stunde(n)
```


Überladen und const 1/2

```
 1  #include <iostream>
 2  using std::cout;
 3
 4  void f(int x) { cout << "int\n"; };
 5  void f(const int x) { cout << "const int\n"; };
 6
 7  int main() {
 8      int x = 0;
 9      const int c = 0;
10      f(x);
11      f(c);
12      return 0;
13  }
```

- `const` wird bei Input-Variablen nicht berücksichtigt
  - Syntax-Fehler beim Kompilieren:
```
overload_const.cpp:2: error: redefinition
of 'f'
```

```
 1  #include <iostream>
 2  using std::cout;
 3
 4  void f(int& x) { cout << "int\n"; };
 5  void f(const int& x) { cout << "const int\n"; };
 6
 7  int main() {
 8      int x = 0;
 9      const int c = 0;
10      f(x);
11      f(c);
12      return 0;
13  }
```

- `const` wichtig bei Referenzen als Input
  - Kompilieren OK und Output:
```
int
const int
```


Überladen und const 2/2

```
 1  #include <iostream>
 2  using std::cout;
 3
 4  class Demo {
 5  private:
 6      int content;
 7  public:
 8      Demo() { content = 0; }
 9      void f() { cout << "normales Objekt\n"; };
10      void f() const { cout << "const Objekt\n"; };
11  };
12
13  int main() {
14      Demo x;
15      const Demo y;
16      x.f();
17      y.f();
18      return 0;
19  }
```

- kann Methode durch `const`-Methode überladen
  - `const`-Meth. wird für `const`-Objekte verwendet
  - sonst wird "normale" Methode verwendet
- Output:
```
normales Objekt
const Objekt
```


Überladen von Operatoren

- Kopierkonstruktor
- Type Casting
- Zuweisungsoperator
- Unäre und binäre Operatoren

- `operator`

Klasse für Komplexe Zahlen

```
1  #include <iostream>
2  #include <cmath>
3  using std::cout;
4
5  class Complex {
6  private:
7    double re;
8    double im;
9  public:
10   Complex(double=0, double=0);
11   double real() const;
12   double imag() const;
13   double abs() const;
14   void print() const;
15 };
16
17 Complex::Complex(double re, double im) {
18   this->re = re;
19   this->im = im;
20 }
21 double Complex::real() const {
22   return re;
23 }
24 double Complex::imag() const {
25   return im;
26 }
27 double Complex::abs() const {
28     return sqrt(re*re + im*im);
29 }
30 void Complex::print() const {
31   cout << re << " + " << im << " * i";
32 }
```

- Default-Parameter sollten in der ersten Deklaration genannt werden
  - Zeile 10: Forward Declaration des Konstruktors
  - Zeile 17–20: Code des Konstruktors

Kopierkonstruktor

```
1  Complex::Complex(const Complex& rhs) {
2    re = rhs.re;
3    im = rhs.im;
4  }
```

- `className::className(const className& rhs)`
- Spezieller Konstruktor für den Aufruf
  - `Complex lhs = rhs;`
  - oder auch `Complex lhs(rhs);`
- erzeugt neues Objekt `lhs`, kopiert Daten von `rhs`
  - also Input als konstante Referenz (read-only)
- wird automatisch erstellt (Shallow Copy), falls nicht explizit programmiert
  - hier formal unnötig, da nur statische Daten

Zuweisungsoperator

```
1  Complex& Complex::operator=(const Complex& rhs) {
2    re = rhs.re;
3    im = rhs.im;
4    return *this;
5  }
```

- `className& className::operator=(const className& rhs)`
- Falls `Complex lhs, rhs;` bereits deklariert
  - Zuweisung `lhs = rhs;`
  - keine Deklaration, also Referenz zurückgeben
  - Input als konstante Referenz (read-only)
  - Output als Referenz für Zuweisungsketten
    - z.B. `a = b = c = d;`
- Funktionalität:
  - Daten von `lhs` durch `rhs` überschreiben
  - ggf. dynamische Daten von `lhs` vorher freigeben
- `this` is Pointer auf das Objekt selbst
  - d.h. `*this` ist das Objekt selbst
- wird automatisch erstellt (Shallow Copy), falls nicht explizit programmiert
  - hier formal unnötig, da nur statische Daten

Type Casting

```
1  Complex::Complex(double re = 0, double im = 0) {
2    this->re = re;
3    this->im = im;
4  }
```

- Konstruktor gibt Type Cast `double` auf `Complex`
  - d.h. $x \in \mathbb{R}$ entspricht $x + 0i \in \mathbb{C}$

```
1  Complex::operator double() const {
2    return re;
3  }
```

- Type Cast `Complex` auf `double`, z.B. durch Realteil
  - formal: `ClassName::operator type() const`
    - implizite Rückgabe
- Beachte ggf. bekannte Type Casts
  - implizit von `int` auf `double`
  - oder implizit von `double` auf `int`

Unäre Operatoren

- unäre Operatoren = Op. mit einem Argument

```
1  const Complex Complex::operator-() const {
2    return Complex(-re,-im);
3  }
```

- Vorzeichenwechsel - (Minus)
  - `const Complex Complex::operator-() const`
    - Output ist vom Typ `const Complex`
    - Methode agiert nur auf aktuellen Members
    - Methode ist read-only auf aktuellen Daten
  - wird Member-Methode der Klasse
- Aufruf später durch `-x`

```
1  const Complex Complex::operator~() const {
2    return Complex(re,-im);
3  }
```

- Konjugation ~ (Tilde)
  - `const Complex Complex::operator~() const`
    - Output ist vom Typ `const Complex`
    - Methode agiert nur auf aktuellen Members
    - Methode ist read-only auf aktuellen Daten
  - wird Member-Methode der Klasse
- Aufruf später durch `~x`

complex_part.hpp

```
1  #include <iostream>
2  #include <cmath>
3  using std::cout;
4
5  class Complex {
6  private:
7    double re;
8    double im;
9  public:
10   Complex(double=0, double=0);
11   Complex(const Complex& rhs);
12   ~Complex();
13   Complex& operator=(const Complex& rhs);
14
15   double real() const;
16   double imag() const;
17   double abs() const;
18   void print() const;
19
20   operator double() const;
21
22   const Complex operator~() const;
23   const Complex operator-() const;
24 };
```

- Zeile 10: Forward Declaration mit Default-Input
- Zeile 10 + 20: Type Casts `Complex` vs. `double`

complex_part.cpp

```
1  #include "complex_part.hpp"
2  using std::cout;
3  Complex::Complex(double re, double im) {
4    this->re = re;
5    this->im = im;
6    cout << "Konstruktor\n";
7  }
8  Complex::Complex(const Complex& rhs) {
9    re = rhs.re;
10   im = rhs.im;
11   cout << "Kopierkonstruktor\n";
12 }
13 Complex::~Complex() {
14   cout << "Destruktor\n";
15 }
16 Complex& Complex::operator=(const Complex& rhs) {
17   re = rhs.re;
18   im = rhs.im;
19   return *this;
20 }
21 double Complex::real() const {
22   return re;
23 }
24 double Complex::imag() const {
25   return im;
26 }
27 double Complex::abs() const {
28     return sqrt(re*re + im*im);
29 }
30 void Complex::print() const {
31   cout << re << " + " << im << "*i";
32 }
33 Complex::operator double() const {
34   cout << "Complex -> double\n";
35   return re;
36 }
37 const Complex Complex::operator-() const {
38   return Complex(-re,-im);
39 }
40 const Complex Complex::operator~() const {
41   return Complex(re,-im);
42 }
```


Beispiel: Konstruktor

```
1  #include <iostream>
2  #include "complex_part.hpp"
3  using std::cout;
4
5  int main() {
6    Complex w(1);
7    Complex x;
8    Complex y(1,1);
9    Complex z = y;
10   x = ~y;
11   w.print(); cout << "\n";
12   x.print(); cout << "\n";
13   y.print(); cout << "\n";
14   z.print(); cout << "\n";
15   return 0;
16 }
```

- Output:

```
Konstruktor
Konstruktor
Konstruktor
Kopierkonstruktor
Konstruktor
Destruktor
1 + 0*i
1 + -1*i
1 + 1*i
1 + 1*i
Destruktor
Destruktor
Destruktor
Destruktor
```

Beispiel: Type Cast

```
1  #include <iostream>
2  #include "complex_part.hpp"
3  using std::cout;
4
5  int main() {
6    Complex z((int) 2.3, (int) 1);
7    double x = z;
8    z.print(); cout << "\n";
9    cout << x << "\n";
10   return 0;
11 }
```

- Konstruktor fordert `double` als Input (Zeile 6)
  - erst expliziter Type Cast `2.3` auf `int`
  - dann impliziter Type Cast auf `double`
- Output:

```
Konstruktor
Complex -> double
2 + 1*i
2
Destruktor
```


Binäre Operatoren

```
1  const Complex operator+(const Complex& x, const Complex& y){
2    double xr = x.real();
3    double xi = x.imag();
4    double yr = y.real();
5    double yi = y.imag();
6    return Complex(xr + yr, xi + yi);
7  }
8  const Complex operator-(const Complex& x, const Complex& y){
9    double xr = x.real();
10   double xi = x.imag();
11   double yr = y.real();
12   double yi = y.imag();
13   return Complex(xr - yr, xi - yi);
14 }
15 const Complex operator*(const Complex& x, const Complex& y){
16   double xr = x.real();
17   double xi = x.imag();
18   double yr = y.real();
19   double yi = y.imag();
20   return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
21 }
22 const Complex operator/(const Complex& x, const double y){
23   return Complex(x.real()/y, x.imag()/y);
24 }
25 const Complex operator/(const Complex& x, const Complex& y){
26   double norm = y.abs();
27   return x*~y / (norm*norm);
28 }
```

- binäre Operatoren = Op. mit zwei Argumenten
  - z.B. `+`, `-`, `*`, `/`
- außerhalb der Klassendefinition als Funktion
  - formal: `const type operator+(const type& rhs1, const type& rhs2)`
  - **Achtung:** kein `type::` da kein Teil der Klasse!
- Zeile 22 + 25: beachte $x/y = (x\overline{y})/(y\overline{y}) = x\overline{y}/|y|^2$

complex.hpp

```
1  #include <iostream>
2  #include <cmath>
3  using std::cout;
4
5  class Complex {
6  private:
7    double re;
8    double im;
9  public:
10   Complex(double=0, double=0);
11   Complex(const Complex&);
12   ~Complex();
13   Complex& operator=(const Complex&);
14
15   double real() const;
16   double imag() const;
17   double abs() const;
18   void print() const;
19
20   operator double() const;
21
22   const Complex operator~() const;
23   const Complex operator-() const;
24 };
25
26 const Complex operator+(const Complex&, const Complex&);
27 const Complex operator-(const Complex&, const Complex&);
28 const Complex operator*(const Complex&, const Complex&);
29 const Complex operator/(const Complex&, const double);
30 const Complex operator/(const Complex&, const Complex&);
```

- "vollständige Bibliothek" ohne unnötige `cout` im folgende `cpp` Source-Code

complex.cpp 1/2

```
1  #include "complex.hpp"
2  using std::cout;
3
4  Complex::Complex(double re, double im) {
5    this->re = re;
6    this->im = im;
7  }
8
9  Complex::Complex(const Complex& rhs) {
10   re = rhs.re;
11   im = rhs.im;
12 }
13
14 Complex::~Complex() {
15 }
16
17 Complex& Complex::operator=(const Complex& rhs) {
18   re = rhs.re;
19   im = rhs.im;
20   return *this;
21 }
22
23 double Complex::real() const {
24   return re;
25 }
26
27 double Complex::imag() const {
28   return im;
29 }
30
31 double Complex::abs() const {
32   return sqrt(re*re + im*im);
33 }
34
35 void Complex::print() const {
36   cout << re << " + " << im << "*i";
37 }
38
39 Complex::operator double() const {
40   return re;
41 }
```

complex.cpp 2/2

```
42  const Complex Complex::operator-() const {
43    return Complex(-re,-im);
44  }
45
46  const Complex Complex::operator~() const {
47    return Complex(re,-im);
48  }
49
50  const Complex operator+(const Complex& x, const Complex& y){
51    double xr = x.real();
52    double xi = x.imag();
53    double yr = y.real();
54    double yi = y.imag();
55    return Complex(xr + yr, xi + yi);
56  }
57
58  const Complex operator-(const Complex& x, const Complex& y){
59    double xr = x.real();
60    double xi = x.imag();
61    double yr = y.real();
62    double yi = y.imag();
63    return Complex(xr - yr, xi - yi);
64  }
65
66  const Complex operator*(const Complex& x, const Complex& y){
67    double xr = x.real();
68    double xi = x.imag();
69    double yr = y.real();
70    double yi = y.imag();
71    return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
72  }
73
74  const Complex operator/(const Complex& x, const double y){
75    return Complex(x.real()/y, x.imag()/y);
76  }
77
78  const Complex operator/(const Complex& x, const Complex& y){
79    double norm = y.abs();
80    return x*~y / (norm*norm);
81  }
```


complex_main.cpp

```
1   #include "complex.hpp"
2   using std::cout;
3
4   int main() {
5     Complex w;
6     Complex x(1,0);
7     Complex y(0,1);
8     Complex z(3,4);
9     w = x + y;
10    w.print(); cout << "\n";
11
12    w = x*y;
13    w.print(); cout << "\n";
14
15    w = x/y;
16    w.print(); cout << "\n";
17
18    w = z/(x + y);
19    w.print(); cout << "\n";
20    return 0;
21  }
```

- Output:

```
1 + 1*i
0 + 1*i
0 + -1*i
3.5 + 0.5*i
```


Zusammenfassung Syntax

- Konstruktor (= Type Cast auf `Class`)
  `Class::Class( ... input ... )`
- Destruktor
  `~Class()`
- Type Cast von `Class` auf `type`
  `Class::operator type() const`
  - explizit durch Voranstellen `(type)`
  - implizit bei Zuweisung auf Var. vom Typ `type`
- Kopierkonstruktor (Deklaration mit Initialisierung)
  `Class::Class(const Class&)`
  - Aufruf durch `Class var(rhs);`
  - oder `Class var = rhs;`
- Zuweisungsoperator
  `Class& Class::operator=(const Class&)`
- unäre Operatoren, z.B. Tilde `~` und Vorzeichen `-`
  `const Class Class::operator-() const`
- binäre Operatoren, z.B. `+`, `-`, `*`, `/`
  `const Class operator+(const Class&, const Class&)`
  - außerhalb der Klasse als Funktion

Welche Operatoren überladen?

```
+    -    *    /    &    ^
|    ~    !    =    <    >    +=
-=   *=   /=   %=   ^=   &=   |=
<<   >>   >>=  <<=  ==   !=   <=
>=   &&   ||   ++   --   ->*  ,
->   []   ()   new  new[] delete delete[]
```

- als unäre Operatoren, vorangestellt `++var`
  `const Class Class::operator++()`
- als unäre Operatoren, nachgestellt `var++`
  `const Class Class::operator++(int)`
- als binäre Operatoren
  `const Class operator+(const Class&, const Class&)`
- kann Operatoren auch überladen
  - z.B. Division `Complex/double` vs. `Complex/Complex`
  - z.B. unär und binär (neg. Vorzeichen vs. Minus)
  - unterschiedliche Signatur beachten!
- Man kann keine neuen Operatoren definieren!
- Man kann `.`, `:`, `::`, `sizeof`, `.*` nicht überladen!

- `https://www.c-plusplus.net/forum/232010-full`

Dynamische Speicherverwaltung

- dynamische Speicherverwaltung in C++
- Dreierregel
- `new`, `new ... []`
- `delete`, `delete[]`

new vs. malloc

- `malloc` reserviert nur Speicher
  - **Nachteil:** Konstr. werden nicht aufgerufen
    - d.h. Initialisierung händisch
- ein dynamisches Objekt

```
type* var = malloc(sizeof(type));
*var = ...;
```

- dynamischer Vektor von Objekten der Länge `N`

```
type* vec = malloc(N*sizeof(type));
vec[j] = ...;
```

- `new` reserviert Speicher + ruft Konstruktoren auf
- ein dynamisches Objekt (mit Standardkonstruktor)

```
type* var = new type;
```

- ein dynamisches Objekt (mit Konstruktor)

```
type* var = new type(... input ... );
```

- dyn. Vektor der Länge `N` (mit Standardkonstruktor)

```
type* vec = new type[N];
```

  - Standardkonstruktor für jeden Koeffizienten
- **Konvention:** Immer `new` verwenden!
- Aber: Es gibt keine C++ Variante von `realloc`

delete vs. free

- `free` gibt Speicher von `malloc` frei

```
type* vec = malloc(N*sizeof(type));
free(vec);
```

  - unabhängig von Objekt / Vektor von Objekten
  - nur auf Output von `malloc` anwenden!
- `delete` ruft Destruktor auf und gibt Speicher von `new` frei

```
type* var = new type(... input ... );
delete var;
```

  - für ein dynamische erzeugtes Objekt
  - nur auf Output von `new` anwenden!
- `delete[]` ruft Destruktor für jeden Koeffizienten auf und gibt Speicher von `new ...[N]` frei

```
type* vec = new type[N];
delete[] vec;
```

  - für einen dynamischen Vektor von Objekten
  - nur auf Output von `new ...[N]` anwenden!
- **Konvention:** Falls Pointer auf keinen dynamischen Speicher zeigt, wird er händisch auf `NULL` gesetzt
  - d.h. nach `free`, `delete`, `delete[]` folgt

```
vec = (type*) NULL;
```


vector.hpp

```
1  #ifndef VECTOR
2  #define VECTOR
3  #include <cmath>
4  #include <cstdlib>
5  #include <cassert>
6
7  // The class Vector stores vectors in Rd
8  class Vector {
9  private:
10   int dim;
11   double* coeff;
12
13 public:
14   // constructors, destructor, assignment
15   Vector();
16   Vector(int dim, double init=0);
17   Vector(const Vector&);
18   ~Vector();
19   Vector& operator=(const Vector&);
20   // return length of vector
21   int size() const;
22   // read and write entries
23   const double& operator[](int k) const;
24   double& operator[](int k);
25   // compute Euclidean norm
26   double norm() const;
27 };
28
29 // addition of vectors
30 const Vector operator+(const Vector&, const Vector&);
31 // scalar multiplication
32 const Vector operator*(const double, const Vector&);
33 const Vector operator*(const Vector&, const double);
34 // scalar product
35 const double operator*(const Vector&, const Vector&);
36
37 #endif
```

- Überladen von `[ ]`
  - falls konstantes Objekt, Methode aus Zeile 23
  - falls "normales Objekt", Methode aus Zeile 24

Dreierregel

- auch: Regel der großen Drei
- Wenn Destruktor oder Kopierkonstruktor oder Zuweisungsoperator implementiert ist, so müssen alle drei implementiert werden!
- notwendig, wenn Klasse dynamische Felder enthält
  - anderenfalls macht Compiler automatisch Shallow Copy (OK bei elementaren Typen!)
  - denn Shallow Copy führt sonst auf Laufzeitfehler bei dynamischen Feldern

vector.cpp 1/3

```
1  #include "vector.hpp"
2  #include <iostream>
3  using std::cout;
4
5  Vector::Vector() {
6    dim = 0;
7    coeff = (double*) NULL;
8    cout << "constructor, empty\n";   //*** demonstration only
9  }
10
11 Vector::Vector(int dim, double init) {
12   this->dim = dim;
13   coeff = new double[dim];
14   for (int j=0; j<dim; ++j) {
15     coeff[j] = init;
16   }
17   cout << "constructor, length " << dim << "\n";       //***
18 }
19
20 Vector::Vector(const Vector& rhs) {
21   dim = rhs.size();
22   coeff = new double[dim];
23   for (int j=0; j<dim; ++j) {
24     coeff[j] = rhs[j];
25   }
26   cout << "copy constructor, length " << dim << "\n";  //***
27 }
28
29 Vector::~Vector() {
30   if (dim > 0) {
31     delete[] coeff;
32   }
33   cout << "free vector, length " << dim << "\n";       //***
34 }
```


vector.cpp 2/3

```
35 Vector& Vector::operator=(const Vector& rhs) {
36     dim = rhs.size();
37     delete[] coeff;
38     coeff = new double[dim];
39     for (int j=0; j<dim; ++j) {
40         coeff[j] = rhs[j];
41     }
42     cout << "shallow copy, length " << dim << "\n";    //***
43     return *this;
44 }
45
46 int Vector::size() const {
47   return dim;
48 }
49
50 const double& Vector::operator[](int k) const {
51   assert(k>=0 && k<=dim);
52   return coeff[k];
53 }
54
55 double& Vector::operator[](int k) {
56   assert(k>=0 && k<=dim);
57   return coeff[k];
58 }
59
60 double Vector::norm() const {
61   double norm = 0;
62   for (int j=0; j<dim; ++j) {
63     norm = norm + coeff[j]*coeff[j];
64   }
65   return sqrt(norm);
66 }
```

- Zugriff über [ ] sowohl für Read-Only Vektoren als auch für Vektoren mit Schreibzugriff erlaubt
  - Zeile 50: Read-Only Vektoren
  - Zeile 55: Vektoren mit Schreibzugriff

vector.cpp 3/3

```
67 const Vector operator+(const Vector& rhs1,
68                        const Vector& rhs2) {
69     assert(rhs1.size() == rhs2.size());
70     Vector result(rhs1);
71     for (int j=0; j<result.size(); ++j) {
72         result[j] = result[j] + rhs2[j];
73     }
74     return result;
75 }
76
77 const Vector operator*(const double scalar,
78                        const Vector& input) {
79     Vector result(input);
80     for (int j=0; j<result.size(); ++j) {
81         result[j] = result[j] * scalar;
82     }
83     return result;
84 }
85
86 const Vector operator*(const Vector& input,
87                        const double scalar) {
88     return scalar*input;
89 }
90
91 const double operator*(const Vector& rhs1,
92                        const Vector& rhs2) {
93     double scalarproduct = 0;
94     assert(rhs1.size() == rhs2.size());
95     for (int j=0; j<rhs1.size(); ++j) {
96         scalarproduct = scalarproduct + rhs1[j]*rhs2[j];
97     }
98     return scalarproduct;
99 }
```

- Zeile 86: Falls man `vector * double` nicht implementiert, kriegt man kryptischen Laufzeitfehler:
  - impliziter Type Cast `double` auf `int`
  - Aufruf Konstruktor mit einem `int`-Argument
  - vermutlich `assert`-Abbruch in Zeile 94

Beispiel

```
 1  #include "vector.hpp"
 2  #include <iostream>
 3  using std::cout;
 4
 5  int main() {
 6    Vector vector1;
 7    Vector vector2(100,4);
 8    Vector vector3 = 4*vector2;
 9    cout << "*** Addition\n";
10    vector1 = vector2 + vector2;
11    cout << "Norm1 = " << vector1.norm() << "\n";
12    cout << "Norm2 = " << vector2.norm() << "\n";
13    cout << "Norm3 = " << vector3.norm() << "\n";
14    cout << "Skalarprodukt = " << vector2*vector3 << "\n";
15    cout << "Norm " << (4*vector3).norm() << "\n";
16    return 0;
17  }
```

- Output:

```
constructor, empty
constructor, length 100
copy constructor, length 100
*** Addition
copy constructor, length 100
shallow copy, length 100
free vector, length 100
Norm1 = 80
Norm2 = 40
Norm3 = 160
Skalarprodukt = 6400
Norm copy constructor, length 100
640
free vector, length 100
free vector, length 100
free vector, length 100
free vector, length 100
```