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 Zeiger

```
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;`
  - 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 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.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 dim, double init);
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 = 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      // just for demonstration purposes
17      cout << "new vector, length " << dim << "\n";
18  }
19  Vector::~Vector() {
20    if (dim > 0) {
21      free(coeff);
22    }
23    // just for demonstration purposes
24    cout << "free vector, length " << dim << "\n";
25  }
```

- 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    int j = 0;
41    for (j=0; j<dim; ++j) {
42      norm = norm + coeff[j]*coeff[j];
43    }
44    return sqrt(norm);
45  }
```

- geändert: Zeile 24, 33, 38

Read-Only Refs als Input 5/5

```
 1  #include "vector.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

Überladen von Funktionen 1/2

```
1  void f(char*);
2  int f(char*);
3  double f(char*, int nP=1);
4  double f(char*, int);
5  int f(char*, char*, int nP=1);
```

- Mehrere Funktionen gleichen Namens möglich
  - wie bei Konstruktoren
  - unterscheiden sich durch ihre Signaturen (Input)
- Diesen Vorgang nennt man Überladen
- Durch Aufruf wird die Richtige ausgewählt
  - Compiler erkennt dies über Input-Parameter
- Input muss Variante eindeutig festlegen
- Rückgabewerte können unterschiedlich sein
  - Also: unterschiedliche Rückgabeparameter und gleiche Eingabeparameter 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 von Operatoren

- Default-Parameter & Forward Declaration
- `operator`

Klasse für Komplexe Zahlen

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

- Default-Parameter dürfen nur in der ersten Deklaration genannt werden
  - Zeile 9: Forward Declaration des Konstruktors
  - Zeile 15–18: 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

Unäre Operatoren

```
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   }
```

- Vorzeichenwechsel `~` (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   using std::cout;
3
4   class Complex {
5   private:
6       double re;
7       double im;
8   public:
9       Complex(double=0, double=0);
10      Complex(const Complex& rhs);
11      ~Complex();
12      Complex& operator=(const Complex& rhs);
13
14      double real() const;
15      double imag() const;
16      void print() const;
17
18      const Complex operator~() const;
19      const Complex operator-() const;
20  };
```

- Zeile 9: Forward Declaration mit Default-Input

complex_part.cpp

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


Ein erstes Beispiel

```
 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
```