C++ Wiederholung

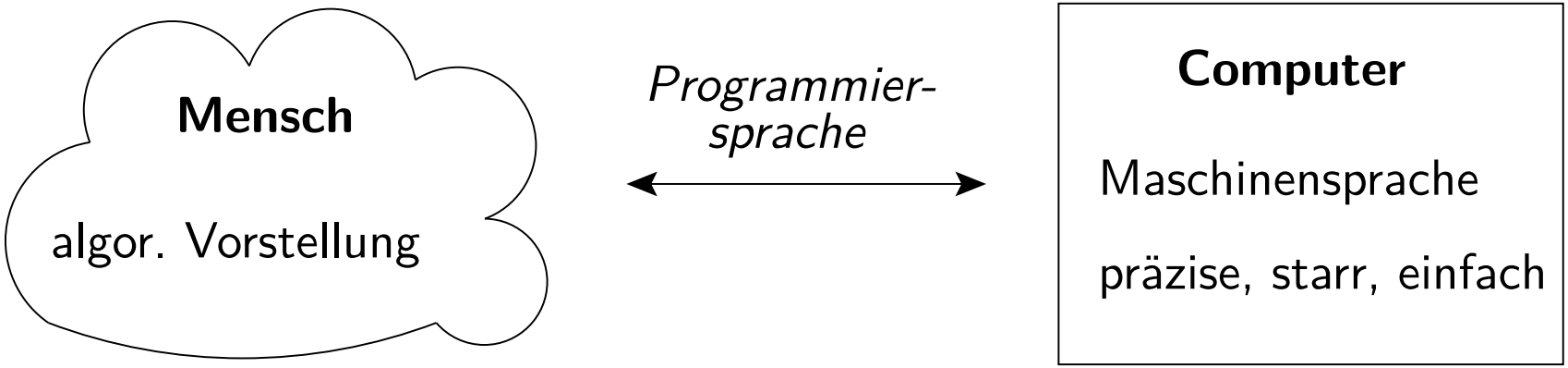
Sven Groß

11. Jan 2016

Übersicht über behandelte Themen

1. Datentypen und Variablen
   - Elementare Datentypen
   - Referenzen
   - Zeiger
2. Operatoren
3. Kontrollstrukturen
   - Verzweigungen
   - Schleifen
4. Ein-/Ausgabe
   - Streams
   - Datei-IO per File Streams
5. Funktionen
   - Definition und Aufruf
   - Wert-/Referenzparameter
   - Default-Parameter
   - Rekursion
   - Überladung
6. Zusammengesetzte Datentypen
   - Felder
   - Dynamische Speicherverwaltung
   - STL-Container
   - Strukturen (Structs)
7. Objektorientierung und Klassen
   - Attribute und Methoden
   - Überladen von Operatoren
   - Konstruktoren und Destruktor
   - Zugriffskontrolle

Wozu Programmiersprachen?



- Ziel: Algorithmus (z.B. Gauß-Algor.) auf Computer umsetzen
- *Programmiersprache* (FORTRAN, C, C++, Java...) als verständliche, unterstützende, abstrahierende Darstellungsform
- Übersetzung von Programmiersprache in Maschinensprache durch entsprechenden *Compiler*

Elemente einer Programmiersprache

notwendig:

- Daten (Variablen)
- Ein-/Ausgabe
- Operatoren (Arithmetik, Vergleich, ...)
- Kontrollstrukturen (Verzweigungen, Schleifen, ...)

hilfreich:

- Kommentare
- Präprozessor (`#include <iostream>`)
- Strukturierung durch Funktionen, Klassen, ...

Datentypen und Variablen

Elementare Datentypen in C++

ganze Zahlen	`int`	`int j = -3;`
reelle Zahlen	`double`	`double z = 3.1415;`
Zeichenketten	`std::string`	`std::string s = "Hello World!";`
	`char[]`	`char s[] = "Hello World!";`
logische Werte	`bool`	`bool bestanden = true;`

...

- weitere Datentypen in der Kurzanleitung
- Jede Variable muß vor Benutzung mit *Typ Name;* **deklariert** werden:
  `int k;`
  - $\rightsquigarrow$ Problem: `k` kann beliebigen Wert enthalten (Datenmüll)
- *besser:* **Initialisierung** mit *Typ Name=Wert;* bzw. *Typ Name(Wert);*
  `int k= 7;`

Beispiele zur Variablendeklaration

```
int zahl= 1, Zahl= 100;    // unterschiedl. Variablen!

double x, y= -1.3E-2,      // x nicht init., y = -0.013,
       z= 2e3;             // z = 2000

std::string Name1= "Donald␣Duck",  // Zeichenkette
            Name2= "Micky␣Mouse";

bool hat_bestanden= true,          // Wahrheitswerte
     istSchlau= false;
```

Merke:

- aussagekräftige Variablennamen verwenden!
- Variablen bei Deklaration auch initialisieren!
- Groß-/Kleinschreibung beachten!

Gültigkeitsbereich (*scope*) von Variablen

- Jede Variable ist nur bis zum Ende des Blockes `{...}` gültig, in dem sie erzeugt wurde (***scope***).

```
{
  int k= 7;  // ab hier ist k definiert
  ...
  {
    ...
    double x= 1.23;  // ab hier ist x definiert
    ...
    std::cout << k;
  }                  // scope von x endet hier
  ...
  std::cout << x;    // FEHLER: x nicht definiert
  ...
}              // scope von k endet hier
```

- Verlässt eine Variable ihren Scope, wird sie automatisch zerstört.

Impliziter und expliziter Cast

- automatische, **implizite Typumwandlung** (**Cast**), leider auch bei Informationsverlust,
  z.B. `int` $\rightleftarrows$ `double`, `float` $\rightleftarrows$ `double`, `int` $\rightleftarrows$ `bool`

$\rightsquigarrow$ oft nützlich, aber manchmal auch gefährlich.

- Typumwandlung kann auch **explizit** angefordert werden:

```
int a=2, b=5;

double c= a / b,                    // 0, int-Division!

       d= double(a) / double(b), // 0.4, C-style cast

       e= static_cast<double>( b ) / a;
            // 2.5, C++-style cast + impliziter Cast
```

- Typumwandlung von/nach `string` funktioniert (leider) anders, dazu später mehr.

const - Deklaration

- Das Schlüsselwort `const` vor dem Datentyp erzeugt **konstante** Variablen.

```
const double Pi = 3.1415;
```

- Konstante Variablen dürfen gelesen,
  aber **nicht** verändert werden.
- taucht oft im Zusammenhang mit Funktionsparametern auf:

```
void write_perReference( const RiesigerDatenTyp& v);
```

  - Referenzparameter vermeidet Kopieren des Arguments (sinnvoll bei großen Datentypen),
  - `const`-Deklaration schützt vor unabsichtlichem Verändern.

Referenzen

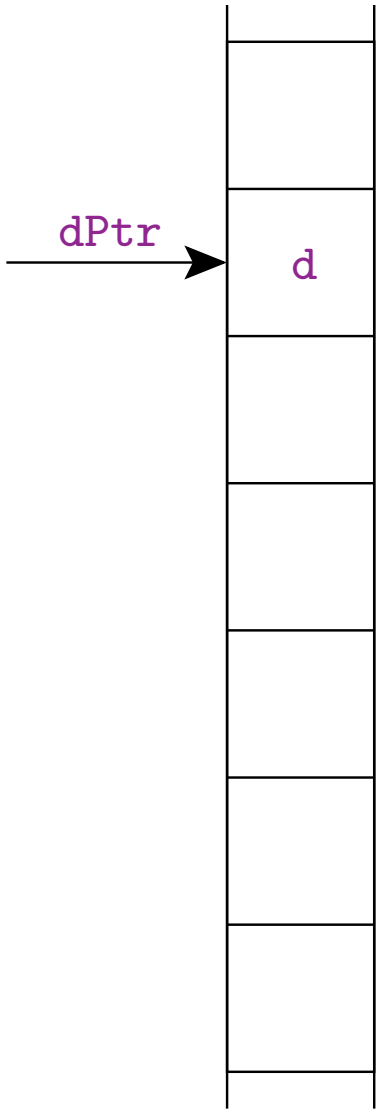
- Referenz dient als **Alias** für bestehende Variable, die dann unter neuem Namen angesprochen werden kann
- Referenzen müssen initialisiert werden

```
int a= 3;
int &r= a;
// a hat Wert 3, r verweist auf 3

a+= 4;
// a hat Wert 7, r verweist auf 7

r-= 2;
// a hat Wert 5, r verweist auf 5
```

Zeiger (*Pointer*)

- Variable, die eine **Speicheradresse** enthält

```
int    i= 5;
double d= 47.11;

int    *iPtr; // Zeiger auf int
double *dPtr; // Zeiger auf double
```



- **Adressoperator** `&` liefert Zeiger auf Variable
  (nicht zu verwecheln mit Referenz-Deklaration)

```
iPtr= &i;
dPtr= &d;
```

- **Dereferenzierungsoperator** `*` liefert Wert an Speicheradresse
  (nicht zu verwecheln mit Zeiger-Deklaration)

```
cout << (*dPtr);   // gibt d aus
(*iPtr)= 42;       // i ist jetzt 42
```

Zeiger (2)

- synonym: `int *iPtr;` oder `int* iPtr;` oder `int * iPtr;`

```
int* iPtr, j, *kPtr;   // zwei int-Zeiger und ein int (j)

double * d_zeiger,     // ein double-Zeiger
       *e_zeiger;      // noch ein double-Zeiger
```

- Speicheradressen werden meist im Hexadezimalsystem (0,...,9,A,...,F) angegeben, `cout << iPtr;` liefert z.B. Ausgabe `0xBF` $= 11 \cdot 16 + 15 = 191$
- **Nullzeiger** zeigt ins „Nirgendwo“, signalisiert leeren Zeiger:

```
iPtr= 0;
```

  Initialisierung mit `0` oder `NULL` (in C++11: `nullptr`)

- `const`-Qualifizierung für Zeiger und/oder Datentyp möglich:

```
const int * p1;   // Zeiger auf konstanten int
int * const p2;   // konstanter Zeiger auf int
```

  $\rightsquigarrow$ Eselsbrücke: von rechts nach links lesen!

Operatoren

Operatoren in C++

Arithmetik	`+ - * / %`	`3 + 5 * 2`
Zuweisung	`=`	`j = 3`
Vergleich	`== != > >= < <=`	`j == 5`
Logik	`&& || !`	`(x >= 0.0) && (x <= 1.0)`
Ein-/Ausgabe	`<< >>`	`cout << "Hello World!";`

...

- weitere Operatoren in der Kurzanleitung
- Priorität der Operatoren, z.B. `*` vor `+`. Im Zweifel Klammern setzen!
- implizite Typumwandlung (*Cast*): `1/4.0` liefert `0.25`
  (Umwandlung `int` → `double`)
- Vorsicht bei Integer-Division: `1/4` liefert `0` !

Beispiele zu Operatoren

```
double x= 3, y= 10/x;      // y = 10.0/3.0 = 3.333...

int a= 3, b= 10/a,         // b = 3      (ganzzahlige Div.)
      rest= 10%a;          // rest = 1   (Modulo-Operator)

string s1= "Blumento", s2= "pferde",
      s3= s1 + s2;         // "Blumentopferde"

bool istNeg= b < 0,        // false
     istPos= !istNeg && (b != 0); // Negierung und ungleich

a= b= 77;          // Zuweisungsoperator gibt etwas zurueck

bool isTen= (a == 10),     // false
     wrong= (a =  10);     // 10 -> true  (nur 0 -> false)
```

Merke:

- Vergleich `==` und Zuweisung `=` nicht verwechseln!
- Vorsicht bei Division von `int`s.

Weitere Operatoren

Operatoren in C++

Zuweisung	`=`	`j = j + 3`
	`+= -= *= /=`	`j += 3`
In-/Dekrement	`++ --`	`j++ ++j`
Auswahl	`?:`	`cond ? val1 : val2` (3 Argumente)

- **Zuweisung**: `j*= 5;` als kürzere Schreibweise für `j= j*5;`
- **Inkrement**: `++j;` als kürzere Schreibweise für `j= j+1;` bzw. `j+= 1;`
- **Dekrement**: `--j;` als kürzere Schreibweise für `j= j-1;` bzw. `j-= 1;`
- **Auswahl**: `betrag= (j >= 0) ? j : -j;` als kürzere Schreibweise für

```
if (j >= 0)
    betrag= j;
else
    betrag= -j;
```

Weitere Operatoren – Beispiele

- **Zuweisung**:

```
double x= 1;
x+= 5;    // wie x= x+5, also x ist 6
x*= 5;    // wie x= x*5, also x ist 30
x/= 4;    // wie x= x/4, also x ist 7.5
```

- **Inkrement**: unterschiedliche Rückgabe der Prä-/Postfix-Variante

```
int a= 0, b= 0;
cout << "Rueckgabe: Postfix " << a++
     <<             ", Praefix " << ++b << endl;
cout << "Nachher: a = " << a
     <<         ", b = " << b << endl;
```

liefert Ausgabe

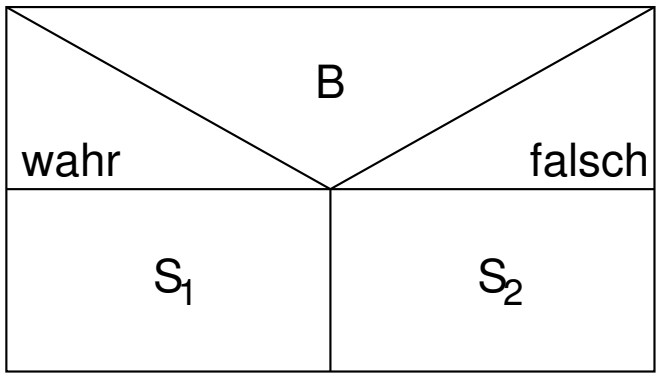
```
Rueckgabe: Postfix 0, Praefix 1
Nachher: a = 1, b = 1
```

Kontrollstrukturen: bedingte Verzweigung

- `if-else`-Verzweigung:

```
if (i < 0)
{ // Bedingung erfuellt
    cout << "i ist negativ"
         << endl;
}
else
{ // Bedingung nicht erfuellt
    cout << "i ist nicht negativ"
         << endl;
}
```

Nassi-Shneiderman-Diagramm:



- kein `;` nach `if` oder `else` !
- `else`-Teil kann auch entfallen

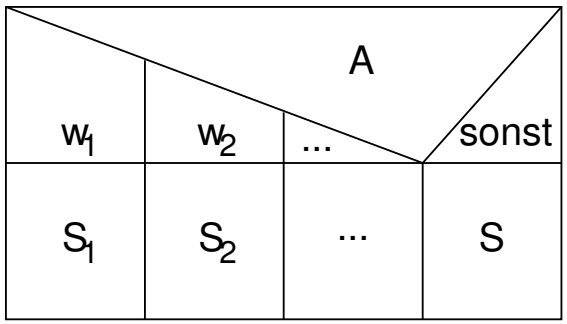
Bedingte Verzweigung (2)

- Fallunterscheidung mit `if-else`-Verzweigung:

```
if (i==1)
    cout << "Eins\n";
else if (i==2)
    cout << "Zwei\n";
else if (i==3)
    cout << "Drei\n";
else
    cout << "???\n";
```

- Alternativ mit `switch-case`-Verzweigung (`break` nicht vergessen!):

```
switch (i)
{
  case 1:  cout << "Eins\n"; break;
  case 2:  cout << "Zwei\n"; break;
  case 3:  cout << "Drei\n"; break;
  default: cout << "???!\n";
}
```



Schleifen – while

Schleifen werden gebraucht, um Anweisungsblöcke mehrfach zu wiederholen.

- Solange Bedingung B erfüllt ist, wiederhole Block S.

```
while (B)
{
  // Anweisungen aus S
}
```

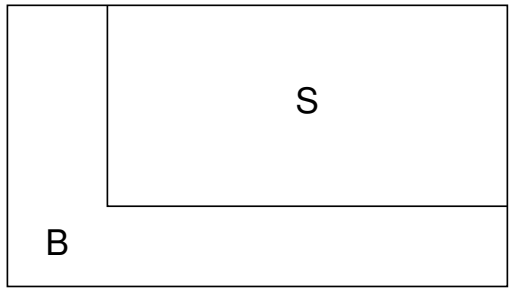
- abweisende Schleife: erst B testen, dann (evtl.) S ausführen.
- kein ; nach while (B) !
  Sonst wird nur ; (leere Anweisung) wiederholt!

```
int i=1;

while ( i<10 )
{
    cout << i << "␣";
    ++i;  // kurz fuer i= i+1;
}
```

Schleifen – do...while

- Variante: **nicht abweisende Schleife**

```
do
{
  // Anweisungen aus S
} while (B);
```



- Erst S ausführen, dann B testen.
- Schleife wird *mindestens einmal* durchlaufen.
- kein ; nach do !

```
int i=1;

do
{
    cout << i << "␣";
    i= i*3;
} while ( i<100 );
```

Schleifen – for

- Oft braucht man Schleifen, die genau `n`-mal durchlaufen werden.

```
int i= 1;
while (i <= n)
{
  // Anweisungen...

  i++;  // kurz fuer i= i+1;
}
```

- `i` heißt Zählvariable.
- Kurzform (sehr praktisch): **`for`-Schleife**

`for` (*Initialisierung; Bedingung; Zähleränderung*)

```
for (int i= 1; i <= n; i++)
{
  // Anweisungen...
}
```

for-Schleife – Beispiel

```
// wir berechnen die Summe der Zahlen 1, 2, ..., 100

int sum= 0;

for (int i=1; i<=100; ++i)
{
    sum= sum + i;
    cout << sum << " ";
}

// cout << i;     // Fehler: i nicht definiert!

cout << "\nSumme = " << sum << endl;
```

- Ausgabe:

```
1 3 6 10 15 21 28 ...
Summe = 5050
```

- Scope der Zählvariable ist der Schleifenblock.

Schleifenabbruch mit `break` und `continue`

- `break` bricht die Schleife ab,
  Code nach der Schleife wird als Nächstes bearbeitet.
- `continue` bricht nur den aktuellen Schleifendurchlauf ab,
  es wird mit nächstem Schleifendurchlauf fortgefahren.

Beispiel: Es wird wiederholt der Kehrwert von eingegebenen Zahlen berechnet.
Null als Eingabe wird ignoriert.
Durch Eingabe von 99 wird die Berechnung abgebrochen.

```
1   while (true) {  // Endlosschleife
2       int x;
3       cout << "Wert: "; cin >> x;
4
5       if (x==0)   // ignorieren: 1/x nicht def.
6           continue;
7       cout << "Kehrwert: " << ( 1.0/x ) << endl;
8       if (x==99)  // Abbruch
9           break;
10  }
11  cout << "Tschoe wa!" << endl;  // Aachener Abschiedsgruss
```

Schleifen: weitere Beispiele

- Diese Schleife zählt in Fünferschritten:

```
for (int i= 1; i <= 100; i+= 5)
{
    cout << i << endl;
}
```

- Diese Schleife zählt rückwärts:

```
for (int i= 10; i >= 1; --i)
{
    cout << i << endl;
}
```

- Quiz: Was macht diese Schleife?

```
for (int i= 1; i <= 10; ++i)
{
    cout << i << endl;
    i*= 2;
}
```

Ein- und Ausgabe

- benötigt Header `iostream` für *input/output stream*: `#include<iostream>`
- `using std::cout;` etc., um `cout` statt `std::cout` schreiben zu dürfen
- **Ausgabe** mittels `cout` (oder `cerr`)

```
int alter= 7;
cout << "Ich bin " << alter << " Jahre alt." << endl;
cerr << "Bald werde ich " << alter+1 << ".\n";
```

- besondere Steuerzeichen: z.B. `\n` neue Zeile, `\t` Tabulator
- **Eingabe** mittels `cin`

```
cout << "Bitte Gewicht in kg eingeben: ";
double gewicht;
cin >> gewicht;
```

- Eselsbrücke: `>>`, `<<` zeigen in Richtung des Informationsflusses

```
cout << "Hallo Leute!\n";   // auf den Bildschirm
int k;
cin >> k;                   // von der Tastatur
```

Anmerkungen zu cin

- Kennen wir schon bei `cout`:

```
int k= 42;
cout << "Die␣Antwort␣ist";
cout << k;
cout << endl;
```

ist gleichbedeutend mit

```
int k= 42;
cout << "Die␣Antwort␣ist" << k << endl;
```

- Ähnlich bei `cin`:

```
int a, b;
cin >> a;
cin >> b;
```

ist gleichbedeutend mit

```
int a, b;
cin >> a >> b;
```

Anmerkungen zu cin (2)

- `cin` liest aus dem **Eingabestrom** (i.d.R. Tastatureingaben)
- `cin` liest per Eingabeoperator `>>` höchstens bis zum nächsten **Whitespace** (Leerzeichen, Tabulator, Zeilenumbruch)
- je nach einzulesendem Datentyp werden Eingaben unterschiedlich interpretiert

```
int k;
string s;
// Eingabe:  5  11erRaus
cin >> k >> s;  // liest k = 5,   s = "11erRaus"

// Eingabe:  5  11erRaus
cin >> s >> k;  // liest s = "5",   k = 11
cin >> s;       // liest s = "erRaus"
```

- **Quiz:** Was passiert bei Eingabe von `1,82` ?

```
double laenge;
cin >> laenge;
```

File Streams: Ein-/ Ausgabe über Dateien

- zuerst header-Datei für *file streams* einbinden:
  `#include <fstream>`
- Öffnen des *input* bzw. *output file streams* zur Ein- bzw. Ausgabe:

```
std::ifstream ifs;
ifs.open("mydata.txt");

std::ofstream ofs("results.dat");
```

- gewohnte Verwendung von Ein- und Ausgabeoperator
  (wie mit `cin` und `cout`)

```
ifs >> a;
ofs << "Das␣Ergebnis␣lautet␣" << f(x) << ".\n";
```

- bei Dateiausgabe: `endl` vermeiden, durch `\n` ersetzen (Pufferung)
- Schließen des *file streams* nach Verwendung:
  `ifs.close(); ofs.close()`

Basisklassen `istream`, `ostream`

- `cout` und `ofs` haben unterschiedliche Datentypen, aber gemeinsamen Basis-Datentyp: `std::ostream`
- `cin` und `ifs` haben unterschiedliche Datentypen, aber gemeinsamen Basis-Datentyp: `std::istream`
- Basis-Datentypen werden auch als **Basisklassen** bezeichnet, von denen speziellere Datentypen **abgeleitet** werden, z.B. `std::ofstream` ist abgeleitet von `std::ostream`

Beispiel:

```
void Ausgabe( double x, std::ostream& os)
{
    os << x << '\n';
}
```

kann benutzt werden als

- `Ausgabe( 3.14, cout);` → Bildschirmausgabe
- `Ausgabe( 3.14, ofs);` → Dateiausgabe

Einlesen per Eingabeoperator >>

- Eingabeoperator `>>` erwartet, dass Daten durch Whitespace getrennt sind (Leerzeichen, Tabulator, Zeilenumbruch)
- Abfrage des Stream-Status:
  - `ifs.good()` : Stream bereit zum Lesen
  - `ifs.eof( )` : Ende der Datei (*end of file*) erreicht
  - `ifs.fail()` : letzte Leseoperation nicht erfolgreich
  - `ifs.clear()`: setzt Status zurück, ermöglicht weiteres Lesen

⚠ Leseoperation `>>` wird nur bei `good`-Status durchgeführt

Beispiel:

```
ifstream ifs( "input.txt");
double d;  int i;  string s;

ifs >> d >> i;    // good
ifs >> i;         // fail
ifs.clear();      // good
ifs >> s;         // good
ifs >> s;         // fail eof
```

Datei `input.txt`:

```
17.5
4711 Hallo
```

Zeilenweises Lesen

- Whitespace-Trennung wie hier manchmal unerwünscht:

```
string vorname, nachname;
cout << "Bitte␣Vor-␣und␣Nachname␣eingeben:␣";
cin >> vorname >> nachname;
```

- Um eine ganze Zeile bis zum nächsten *Enter* (’\n’) zu lesen:

```
string name;
cout << "Bitte␣Vor-␣und␣Nachname␣eingeben:␣";
getline( cin, name);
```

- funktioniert genauso für *input file streams*:

```
ifstream ifs( "fragen.txt"); // eine Zeile pro Frage
vector<string> question;

while ( !ifs.eof() ) {
    string zeile;
    getline( ifs, zeile);
    question.push_back( zeile); // hinten anhaengen
}
```

1. Datentypen und Variablen
   - Elementare Datentypen
   - Referenzen
   - Zeiger
2. Operatoren
3. Kontrollstrukturen
   - Verzweigungen
   - Schleifen
4. Ein-/Ausgabe
   - Streams
   - Datei-IO per File Streams
5. Funktionen
   - Definition und Aufruf
   - Wert-/Referenzparameter
   - Default-Parameter
   - Rekursion
   - Überladung
6. Zusammengesetzte Datentypen
   - Felder
   - Dynamische Speicherverwaltung
   - STL-Container
   - Strukturen (Structs)
7. Objektorientierung und Klassen
   - Attribute und Methoden
   - Überladen von Operatoren
   - Konstruktoren und Destruktor
   - Zugriffskontrolle

Funktionen

- Beispiele von Funktionen:

```
double sqrt( double x)
```

```
double pow( double basis, double exp)
```

Rückgabetyp Name (formale Parameterliste)

- Aufruf:

```
sqrt( 4.0);
```
→ liefert `2.0`

```
pow( x, 2.0);
```
→ liefert x^2

Name (Argumentliste)

- Funktionen sind Sinneinheiten für Teilprobleme (Übersichtlichkeit, Struktur, Wiederverwendbarkeit)

Funktionsdefinition

Beispiel: *Mittelwert zweier reeller Zahlen*

Eingabe/Parameter:	zwei reelle Zahlen `x, y`
Ausgabe:	eine reelle Zahl
Rechnung:	`z = (x + y)/2`, dann `z` zurückgeben

```
double MW( double x, double y)     // Funktionskopf
{ // Funktionsrumpf
    double z = (x + y)/2;
    return z;
}
```

- `x`, `y` und `z` sind lokale Variablen des Rumpfblockes

  → werden beim Verlassen des Rumpfes zerstört

Funktionsaufruf

```
1   double MW( double x, double y)     // Funktionskopf
2   { // Funktionsrumpf
3       return (x + y)/2;
4   }
5
6   int main()      // ist auch eine Funktion...
7   {
8       double a= 4.0, b= 8.0, result;
9       result= MW( a, b);
10      return 0;
11  }
```

Was geschieht beim Aufruf `result = MW( a, b);` ?

- Parameter `x, y` werden angelegt
  als Kopien der Argumente `a, b`. (*call by value*)
- Rumpfblock wird ausgeführt.
- Bei Antreffen von `return` wird der dortige Ausdruck als Ergebnis zurückgegeben und der Rumpf verlassen (`x, y` werden zerstört).
- An `result` wird der zurückgegebene Wert `6.0` zugewiesen.

weitere Funktionen

- `int main()` ist das Hauptprogramm, gibt evtl. Fehlercode zurück
- Es gibt auch Funktionen ohne Argumente und/oder ohne Rückgabewert:

```
void SchreibeHallo()
{
    cout << "Hallo!" << endl;
}
```

- Eine Funktion mit Rückgabewert `bool` heißt auch **Prädikat**.

```
bool IstGerade( int n)
{
    if ( n%2 == 0)
        return true;
    else
        return false;
}
```

Funktionen – Beispiele

```
1  double quad( double x)
2  { // berechnet Quadrat von x
3    return x*x;
4  }
5
6  double hypothenuse( double a, double b)
7  { // berechnet Hypothenuse zu Katheten a, b (Pythagoras)
8    return std::sqrt( quad(a) + quad(b) );
9  }
10
11 int main()
12 {
13   double a, b;
14   cout << "Laenge␣der␣beiden␣Katheten:␣";  cin >> a >> b;
15
16   double hypoth= hypothenuse( a, b);
17   cout << "Laenge␣der␣Hypothenuse␣=␣" << hypoth << endl;
18   return 0;
19 }
```

Funktionen – Hinweise

- Variablen `a, b` in `main` haben nichts mit Var. `a, b` in `hypothenuse` zu tun: Scope ist lokal in der jeweiligen Funktion.
- Grundsätzlicher Rat: Variablen nur in Funktionen deklarieren (**lokal**), *nie* ausserhalb (**global**) bis auf wenige Ausnahmen!
- Aufruf einer Funktion nur *nach* deren Deklaration/Definition möglich:

```
double quad( double x);  // Deklaration der Funktion quad

// ab hier darf die Funktion quad benutzt werden

double hypothenuse( double a, double b)
{ // berechnet Hypothenuse zu Katheten a, b (Pythagoras)
  return std::sqrt( quad(a) + quad(b) );
}

double quad( double x)   // Definition der Funktion quad
{ // berechnet Quadrat von x
  return x*x;
}
```

Wertparameter (call by value)

Diese Funktion tut nicht, was sie soll:

- Funktionsdefinition

```
void tausche( double x, double y)
{
    double tmp= x;
    x= y;
    y= tmp;
}
```

- Aufruf

```
double a=5, b=7;
tausche( a, b);
cout <<   "a=" << a << ", b=" << b << endl;
// liefert:  a=5, b=7
```

- **Grund:** Es werden nur die Kopien `x, y` getauscht!
  `a, b` werden nicht verändert.

Referenzparameter (call by reference)

Abhilfe: Referenzparameter

- Funktionsdefinition mit Referenzparametern (beachte die `&` !)

```
void tausche( double& x, double& y)
{
    double tmp= x;
    x= y;
    y= tmp;
}
```

- Was passiert beim Aufruf `tausche( a, b);` ?
  - Parameter `x, y` werden angelegt
    als Alias der Argumente `a, b`. (*call by reference*)
  - Beim Tausch von `x, y` werden auch `a, b` verändert.

⇝ liefert nun: a=7, b=5.

Default-Parameter

Beispiel: Funktion

```
void makeCoffee( int cups, bool withMilk= true,
                 bool withSugar= true);
```

kann so aufgerufen werden:

```
makeCoffee( 1, false, false); // 1x schwarz
makeCoffee( 2, false);        // 2x mit Zucker
makeCoffee( 3);               // 3x mit Zucker+Milch
```

- **Default-Parameter**: Funktionsparameter mit Default-Werten
- können beim Aufruf weggelassen werden, dann mit Default-Werten belegt
- nur am Ende der Parameterliste:
  nach Default-Parameter keine normalen Parameter mehr erlaubt
- auch bei Methoden und Konstruktoren möglich

Rekursive Funktionen

- **Rekursive Funktionen** rufen sich selber auf
- sicherstellen, dass Rekursion abbricht!
- einfache Rekursion kann auch durch Iteration (Schleife) ersetzt werden

```
int fakultaet_rekursiv( int n)
{
    if (n>1)
        return n*fakultaet_rekursiv(n-1);
    else
        return 1;     // Abbruch der Rekursion
}

int fakultaet_iterativ( int n)
{
    int fak= 1;
    for (int i=1; i<=n; ++i)
        fak*= i;

    return fak;
}
```

Rekursive Funktionen (2)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_rekursiv( int n)
{
    if (n >= 2)
      return fib_rekursiv( n-2) + fib_rekursiv( n-1);
    else
      return 1;     // Abbruch der Rekursion
}
```

- Rekursion hier sehr ineffizient!
  - `fib_rekursiv( n )` erzeugt $mathcal{O}(2^{n-1})$ Aufrufe von sich selbst.
  - Bei Berechnung von a_n wird z.B. a_2 sehr oft neu berechnet.

$\rightsquigarrow$ **besser:** Rekursion durch Iteration ersetzen

Rekursive Funktionen (3)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_iterativ( int n)
{
    if (n < 2)
        return 1;

    int ak, ak1= 1, ak2= 1;  // a_k, a_{k-1}, a_{k-2}

    for (int k= 2; k <= n; ++k)
    {
        ak= ak2 + ak1;       // nach Definition
        ak2= ak1;  ak1= ak; // Variablen shiften
    }
    return ak;
}
```

Iterativer Ansatz ist deutlich effizienter:

- Aufwand zur Berechnung von a_n ist linear in n (nicht exponentiell)
- a_2 wird genau einmal berechnet (für `k=2`)

Signatur einer Funktion

- **Signatur** einer Funktion = Name + Parameterliste
- Funktionen dürfen selben Namen besitzen,
  aber Signatur muss eindeutig sein.
- Drei verschiedene Funktionen mit verschiedener Signatur:

```
void PrintSum( int x, int y);
void PrintSum( double x, double y);
void PrintSum( double x, double y, double z);
```

- *Nicht* erlaubt, da beide Funktionen selbe Signatur besitzen:

```
int    Summe( double x, double y);
double Summe( double a, double b);
```

- Eindeutige Signatur wichtig, damit der Compiler beim Funktionsaufruf die richtige Funktion auswählen kann, z.B. bei `PrintSum( 1.0, 3.0);`

Funktionsüberladung

```
1   void PrintSum( int x, int y)
2   {
3       cout << "i-sum = " << x + y << endl;
4   }
5
6   void PrintSum( double x, double y)
7   {
8       cout << "d-sum = " << x + y << endl;
9   }
10
11  int main()
12  {
13      PrintSum( 1, 3);      // Ausgabe "i-sum = 4"
14      PrintSum( 1.3, 2.7); // Ausgabe "d-sum = 4"
15      return 0;
16  }
```

- **Überladen**: Funktionen mit gleichem Namen und gleicher Parameterzahl, aber verschiedenen Typen $\rightsquigarrow$ unterscheidbar durch Signatur
- Eindeutige Signatur wichtig, damit richtige Funktion ausgewählt wird

Funktionsüberladung (2)

```
1   void PrintSum( double x, double y)
2   {
3       cout << "d-sum = " << x + y << endl;
4   }
5
6   int main()
7   {
8       PrintSum( 1, 3);      // Ausgabe "d-sum = 4"
9       PrintSum( 1.3, 2.7); // Ausgabe "d-sum = 4"
10      return 0;
11  }
```

- `PrintSum( 1, 3);` bewirkt keinen Compilerfehler, obwohl keine `int`-Funktion deklariert ist.
- Grund: automatische, implizite Typumwandlung (**Cast**) für `int` → `double`

⇝ oft nützlich, aber manchmal auch gefährlich.

Funktionsüberladung (3)

```
1   void PrintSum( int x, int y)
2   {
3       cout << "i-sum = " << x + y << endl;
4   }
5
6   int main()
7   {
8       PrintSum( 1, 3);      // Ausgabe "i-sum = 4"
9       PrintSum( 1.3, 2.7); // Ausgabe "i-sum = 3"
10      return 0;
11  }
```

- `PrintSum( 1.3, 2.7);` bewirkt keinen Compilerfehler, obwohl keine `double`-Funktion deklariert ist.
- Grund: automatische, implizite Typumwandlung (**Cast**) für `double` → `int`

⇝ Informationsverlust!

Funktionsüberladung – warnendes Beispiel

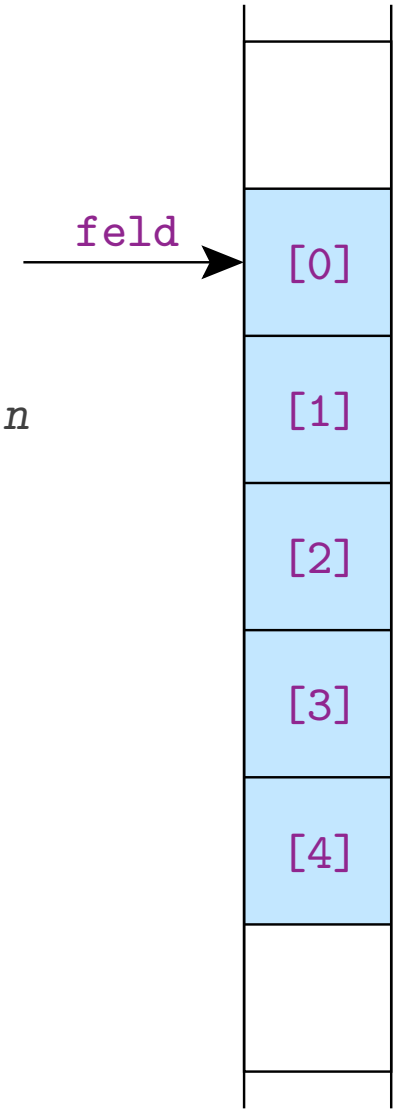
```
1  #include <iostream>
2  using namespace std;  // gefaehrlich !
3
4  void max( double a, double b)
5  {
6      double maxi= a>b ? a : b;
7      cout << "Das␣Maximum␣ist␣" << maxi << endl;
8  }
9
10 int main()
11 {
12     max( 3, 4);      // ruft std::max fuer int auf
13     return 0;
14 }
```

- am besten passende Funktion wird aufgerufen,
  hier `int std::max( int, int)`
- **Besser:** `using std::cout; using std::endl;` statt `using namespace std;`,
  um Vorteil des Namensraums (Kapselung, Eindeutigkeit) zu erhalten

1. Datentypen und Variablen
   - Elementare Datentypen
   - Referenzen
   - Zeiger
2. Operatoren
3. Kontrollstrukturen
   - Verzweigungen
   - Schleifen
4. Ein-/Ausgabe
   - Streams
   - Datei-IO per File Streams
5. Funktionen
   - Definition und Aufruf
   - Wert-/Referenzparameter
   - Default-Parameter
   - Rekursion
   - Überladung
6. Zusammengesetzte Datentypen
   - Felder
   - Dynamische Speicherverwaltung
   - STL-Container
   - Strukturen (Structs)
7. Objektorientierung und Klassen
   - Attribute und Methoden
   - Überladen von Operatoren
   - Konstruktoren und Destruktor
   - Zugriffskontrolle

Felder (*Arrays*)

- Wichtiger Verwendungszweck für Zeiger: **Felder**
- ⚠ Nummerierung der Einträge beginnt bei 0, endet bei n-1
- Zugriff auf Einträge mit `[ ]`-Operator
- keine Bereichsprüfung (Speicherverletzung !!)




```
double feld[5];   // legt ein Feld von 5 doubles an
                  // (nicht initialisiert),
                  // feld ist ein double-Zeiger

for (int i=0; i<5; ++i)
  feld[i]= 0.7;   // setzt Eintraege 0...4

cout << (*feld); // gibt ersten Eintrag aus
                 // (feld[0] und *feld synonym)

cout << feld[5]; // ungueltiger Zugriff!
                 // Kein Compilerfehler, evtl.
                 // Laufzeitfehler (segmentation fault)
```

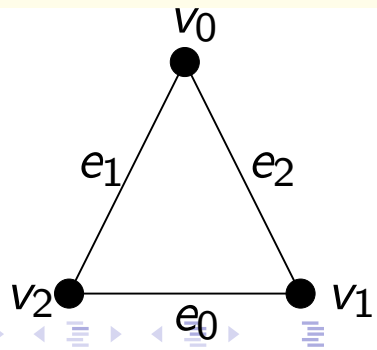
Initialisierung von Feldern

Felder können bei Erzeugung auch **initialisiert** werden,
in diesem Fall muss die Länge nicht spezifiziert werden.

```
double werte[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};
    // legt ein Feld von 5 doubles an
    // und initialisiert es mit geg. Werten
double zahl[5]=  { 1.1, 2.2}; // (Rest mit Null initial.)

int lottozahlen[]= { 1, 11, 23, 29, 36, 42};
    // legt ein Feld von 6 ints an (ohne Laengenangabe)
    // und initialisiert es mit geg. Werten

int vertex_of_edge[3][2]= { {1, 2}, {0, 2}, {0, 1} };
    // legt ein zweidimensionales int-Feld an
    // mit 3x2 gegebenen Werten. Typ: int**
```

Quiz:

- Welchen Typ hat der Ausdruck `vertex_of_edge[1]`?
- Wie kann ich den 1. Eckpunkt der 3. Kante abfragen?



Kopieren von Feldern

```
double werte[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};

double *nochEinFeld= werte;  // kopiert nur den Zeiger !!!
nochEinFeld[2]= 99;          // veraendert werte[2] !!!

double auchFalsch[5]; // reserviert neuen Speicher (5 doubles)
// auchFalsch= werte; // Compilerfehler

double richtig[5];  // neuen Speicher reservieren

for (int i=0; i<5; ++i)
    richtig[i]= werte[i]; // Werte kopieren
```

⚠ Vorsicht beim Kopieren von Feldern!

- Zuweisung = verändert nur den Zeiger, nicht aber den Feldinhalt
- Kopie benötigt eigenen Speicher
- Eintrag für Eintrag kopieren
- Alternative für Felder: **Vektoren** bequem mit = kopieren (dazu später mehr)

Spezielle char-Felder: C-Strings

Relikt aus C-Zeiten: **C-Strings**

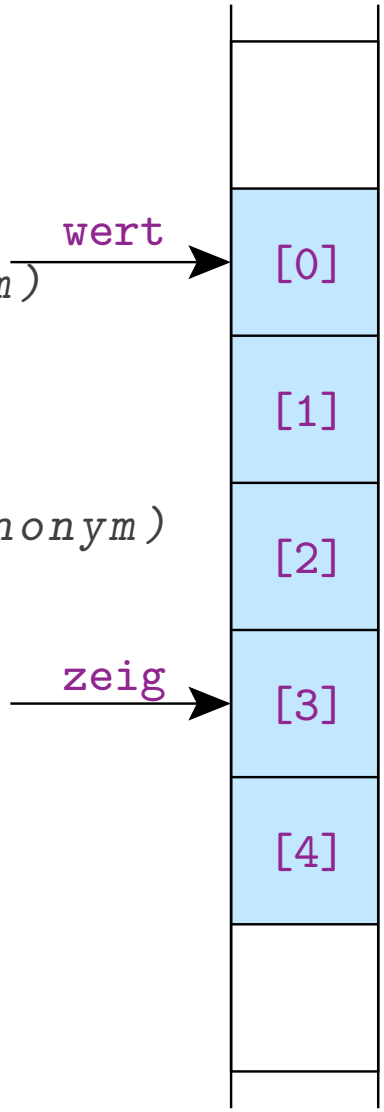
- char-Felder
- hören mit Terminationszeichen '\0' auf
- Initialisierung mit "..."

```
char cstring[]= "Hallo";
    // Zeichenkette mit 5 Zeichen, aber Feld mit 6 chars
    // { 'H', 'a', 'l', 'l', 'o', '\0' }

char message[]= "Hier␣spricht␣Edgar␣Wallace\n";
    // noch ein C-String

string botschaft= message;
    // C++-String, der mit C-String initialisiert wird

const char *text= botschaft.c_str();
    // ...und wieder als C-String
```

Zeigerarithmetik

- Zeiger + `int` liefert Zeiger:

```
double wert[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};

cout << (*wert); // gibt ersten Eintrag aus
                 // (wert[0] und *wert synonym)

double *zeig= wert + 3;
cout << (*zeig); // gibt vierten Eintrag aus
                 // (wert[3] und *(wert+3) synonym)

double *ptr= &(wert[2]);
ptr++;
bool same= (zeig == ptr);  // true
```



- Zeiger – Zeiger liefert `int` (Abstand):

```
int differenz= zeig - ptr,
    index= zeig - wert;
```

Quiz: Werte von `differenz` und `index`?

Casts bei Zeigern

```
void Ausgabe( const double* feld, int n)
{
    ...
}

int main()
{
    double data[]= { 47, 11, 0, 8, 15};

    Ausgabe( data, 5);
    return 0;
}
```

- Impliziter Cast `double*` → `const double*` bei Aufruf von `Ausgabe`
- `const` ermöglicht nur Lesezugriff auf Einträge von `feld`, verhindert, dass Einträge von `data` versehentlich in der Funktion `Ausgabe` verändert werden
- **Best Practice:** `const` benutzen, wenn möglich ⇝ mehr Datensicherheit

Beispiel: so geht es nicht...

Beispiel: Funktion, die ein Feld kopieren soll

```
1  double* copy( double* original, int n)
2  {
3      double kopie[n];  // statisches Feld
4      for (int i=0; i<n; ++i)
5          kopie[i]= original[i];
6      return kopie;
7  }
8
9  int main()
10 {
11     double zahlen[3]= { 1.2, 3.4, 5.6};
12     double *zahlenkopie;
13     zahlenkopie= copy( zahlen, 3);
14     ...
15     return 0;
16 }
```

⚠ Das funktioniert so nicht! `zahlenkopie` zeigt auf ungültigen Speicherbereich, da lokales Feld `kopie` bereits zerstört wurde.

Dynamische Speicherverwaltung

- **dynamische Felder**,
  - falls Speicher vor Verlassen des Scopes freigegeben werden soll
  - falls Speicher nach Verlassen des Scopes freigegeben werden soll
  - (in C: falls Länge erst zur Laufzeit festgelegt werden soll)
- Speicherplatz belegen (*allocate*) mit `new`
- Speicherplatz freigeben (*deallocate*) mit `delete`,
  passiert **nicht automatisch**!

```
int *feld= new int[5]; // legt dyn. int-Feld der Laenge 5 an
int *iPtr= new int;    // legt dynamisch neuen int an
...
delete[] feld;         // Speicher wieder freigeben
delete   iPtr;
```

Strikte Regel: Auf jedes `new` (bzw. `new[]`) muss später ein zugehöriges `delete` (bzw. `delete[]`) folgen!

Dynamische Speicherverwaltung (2)

- **normale Variablen:** (automatische Speicherverwaltung)
  - werden erzeugt bei Variablendeklaration
  - werden automatisch zerstört bei Verlassen des Scope
- **dynamische Variablen:** (dynamische Speicherverwaltung)
  - werden erzeugt mit `new` (liefert Zeiger auf neuen Speicher zurück)
  - werden zerstört mit `delete` (Freigabe des Speichers)

  $\rightsquigarrow$ volle Kontrolle = volle Verantwortung!

Dynamische Speicherverwaltung (3)

```
1  {
2      double x= 0.25, *dPtr= 0; // neue double-Var. x,
3                                // neuer double-Zeiger dPtr
4
5      int *iPtr= 0, a= 33;      // neue int-Var. a,
6                                // neuer int-Zeiger iPtr
7      {
8          double y= 0.125;   // neue double-Var. y
9          dPtr= new double;  // neue double-Var. (ohne Namen)
10         *dPtr= 0.33;
11         iPtr= new int[4];  // neues int-Feld (ohne Namen)
12
13     } // Ende des Scope: y wird automatisch zerstoert
14     for (int i=0; i<4; ++i)
15         iPtr[i]= i*i;
16     ...
17     delete[] iPtr; // int-Feld wird zerstoert
18     delete dPtr;   // double-Var. wird zerstoert
19
20 } // Ende des Scope: Var. x, a werden automatisch zerstoert,
21   //          Zeiger dPtr, iPtr werden automatisch zerstoert.
```

Beispiel: so ist es richtig

Beispiel: Funktion, die ein Feld kopieren soll

```
1  double* copy( double* original, int n)
2  {
3      double *kopie= new int[n];  // dynamisches Feld
4      for (int i=0; i<n; ++i)
5          kopie[i]= original[i];
6      return kopie;
7  }
8
9  int main()
10 {
11     double zahlen[3]= { 1.2, 3.4, 5.6};
12     double *zahlenkopie;
13     zahlenkopie= copy( zahlen, 3);
14     ...
15     delete[] zahlenkopie;  // Speicher wieder frei geben
16     return 0;
17 }
```

`zahlenkopie` zeigt auf gültigen Speicherbereich, da das mit `new int[5]` erzeugte dynamische Feld erst am Ende von `main` mit `delete[]` zerstört wird.

Sequentielle Container: Vektoren

- C++-Alternative zu C-Feldern (*Arrays*)
- ein Datentyp aus der STL (*Standard Template Library*)
- Datentyp der Einträge bei Deklaration angeben, z.B. `vector<double>`
- Angabe der Größe bei Deklaration oder später mit `resize`,
  Abfrage der Größe mit `size`
- zusammenhängender Speicherbereich garantiert,
  aber möglicherweise keine feste Adresse
- Zugriff auf Einträge mit `[ ]`-Operator oder `at`
  Nummerierung 0, . . . , n-1, für `[ ]` keine Bereichsprüfung!

```
#include <vector>
using std::vector;

vector<double> v(2), w;
w.resize(5);

for (int i=0; i < w.size(); ++i)
    w[i]= 0.07;   // alternativ:  w.at(i)= 0.07;
```

Vektoren (2)

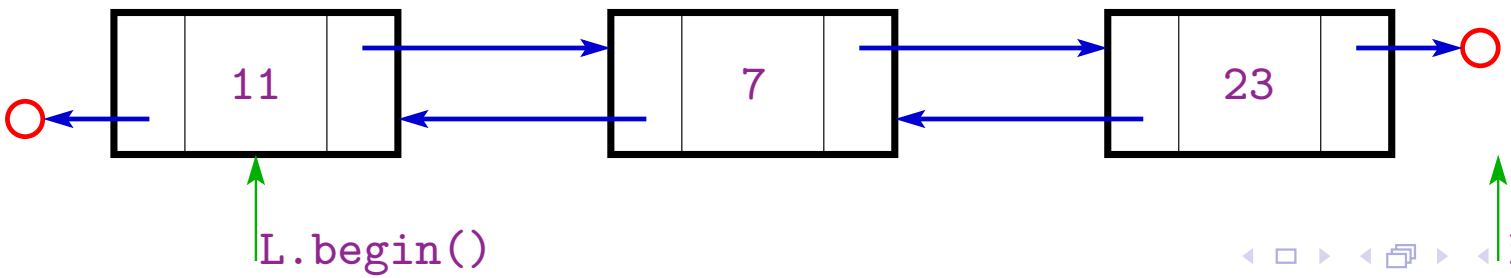
```
w.push_back( 4.1); // haengt neuen Eintrag hinten an

v.pop_back();      // letzten Eintrag loeschen

double* ptr= &(v[0]);

v= w;              // Zuweisung, passt Groesse von v an

// ptr zeigt evtl auf ungueltigen Speicherbereich!!
```

- Zuweisung `=` funktioniert wie erwünscht (im Gegensatz zu Feldern)
- kein Vektor im mathematischen Sinne:
  arithmetische Operatoren (`+, -, *`) nicht implementiert
- Ändern der Länge führt evtl. zu Reallozierung an anderer Speicheradresse
  $\rightsquigarrow$ Vorsicht mit Zeigern!
- STL definiert weitere **Container** wie `list, map`, dazu heute mehr
- Container verwenden **Iteratoren** statt Zeiger, auch dazu heute mehr

Sequentielle Container: Listen

- **Seq. Container** für linear angeordnete Daten (1 Vorgänger/Nachfolger)
- STL bietet z.B. `vector` (kennen wir schon), `list` (heute), `stack`, ...
- `list`: doppelt verkettete Liste
  - `#include<list>` einbinden
  - jedes Element enthält Zeiger auf Vorgänger/Nachfolger
  - Zugriff auf Elemente mittels **Iterator**

```
list<int> L;
L.push_back( 7);     L.push_back( 23);
L.push_front( 11);

for (list<int>::iterator it= L.begin(); it != L.end(); ++it)
    cout << *it << ",";
```



Liste (2)

Vergleich zu Vektor:

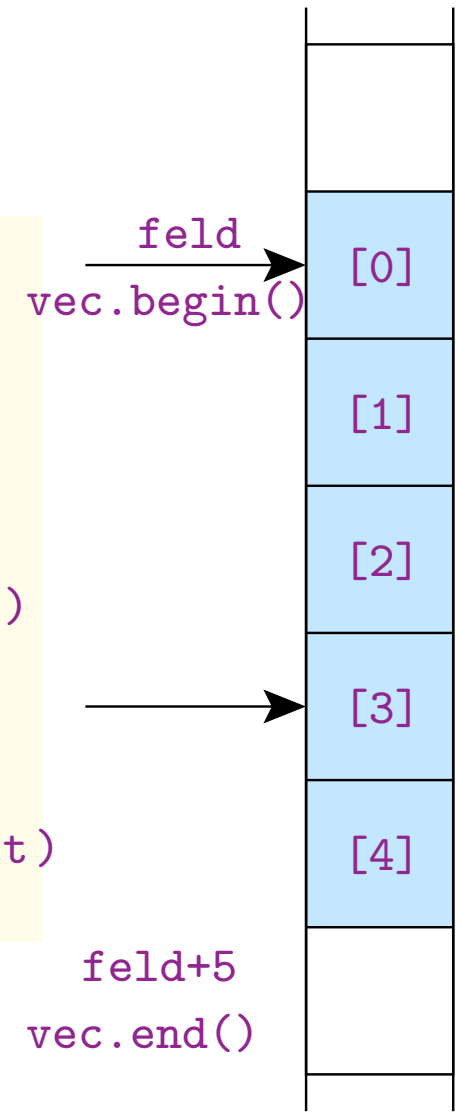
- — kein direkter Zugriff, z.B. auf 3. Element, möglich (kein *random access*)
- — mehr Speicherbedarf als Vektor
- \+ Einfügen/Löschen von Elementen aus der Mitte
- \+ Zusätzliche Funktionalität, z.B. `sort`, `merge`, `reverse`

```
    // Eintraege von L:  11 7 23
list<int>::iterator pos= L.begin();
++pos;                   // pos zeigt nun auf 7
L.insert( pos, 33);      // 11 33 7 23
                         // pos zeigt weiter auf 7
L.erase( L.begin() ); //     33 7 23

L.sort();                // 7 23 33
```

Iteratoren

- werden in der STL benutzt, um Container zu durchlaufen
- verhalten sich wie Zeiger

```
double feld[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};

vector<double> vec(5);
for (int i=0; i<5; ++i)
    vec[i]= 1.1*(i+1);

for (double* zeig= feld; zeig != feld+5; ++zeig)
    cout << (*zeig) << endl;

for (vector<double>::iterator it= vec.begin();
                              it != vec.end(); ++it)
    cout << (*it) << endl;
```



- Datentyp: `ContainerT::iterator`
- Operatoren: `++`, `--`, `==`, `!=`, `*` (Dereferenzierung)

Assoziative Container: Map

- **Assoziative Container:** allgemeiner Indextyp (nicht notw. ganzzahlig), Index heisst auch **Schlüssel** (*key*)
- Beispiel: Telefonbuch

```
map< string, int> TelBuch;

TelBuch["Harry"]=   666;  TelBuch["Ron"]= 123;
TelBuch["Hermine"]= 999;
```

- hier Index- bzw. Schlüsseltyp `string`, Werttyp `int`
- **eindeutige** Zuordnung: Schlüssel $\mapsto$ Wert
- Container speichert Paare vom Typ `pair< string, int>`, Zugriff auf Attribute `first` (Schlüssel), `second` (Wert)

```
for( map<string,int>::iterator it= TelBuch.begin();
                               it!=TelBuch.end();   ++it)
  cout << it->first << "s␣Nummer␣ist␣"
       << it->second << endl;
```

Map (2)

```
Harrys Nummer ist 666
Hermines Nummer ist 999
Rons Nummer ist 123
```

- Einträge werden nach Schlüsseln **sortiert** gespeichert,
  interne Datenstruktur meist binärer Suchbaum: Suche in $\mathcal{O}(\log n)$
- Randnotiz: C++11 bietet auch `unordered_map` ohne Sortierung,
  interne Datenstruktur Hash-Map, Suche in $\mathcal{O}(1)$
- Maps bieten **Random-Access-Zugriff** über Schlüssel:

```
cout << "Harrys Nummer: " << TelBuch["Harry"] << endl;

// oder alternativ so:
map<string,int>::iterator p= TelBuch.find("Harry");
if (p != TelBuch.end() ) // Eintrag gefunden
  cout << "Harrys Nummer: " << p->second << endl;
```

Set

- **Set:** Menge von **eindeutigen** Elementen, entspricht Map mit Schlüssel = Wert
- sortierte Speicherung

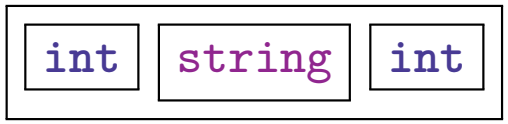
```
set<int> prim;

for (int i=7; i>=2; --i)
    prim.insert( i);
// jetzt:  prim = { 2 3 4 5 6 7 }

prim.erase(4);
set<int>::iterator p= prim.find(6);
prim.erase( p);

for (set<int>::iterator it= prim.begin();
                        it!=prim.end();   ++it)
    cout << *it << "\t";
```

Ausgabe:

```
2   3   5   7
```

Strukturen (*Structs*)

- **Strukturen**: Zusammenfassung unterschiedlicher Datentypen zu einem *neuen benutzerdefinierten Datentyp*
- z.B. neuer Datentyp `Datum`:




```
struct Datum
{
    int    Tag;
    string Monat;
    int    Jahr;
}; // Semikolon nicht vergessen!
```

```
Datum MamasGeb, MeinGeb;

MamasGeb.Tag=   12;
MamasGeb.Monat= "Februar";
MamasGeb.Jahr=  1954;

MeinGeb= MamasGeb;   // Zuweisung
MeinGeb.Jahr+= 30;   // ich bin genau 30 Jahre juenger
```

Klassen: Attribute und Methoden

- **Objektorientierung**: Objekte beinhalten nicht nur **Daten**, sondern auch **Funktionalität** des neu definierten Datentyps
- Konzeptionelle Erweiterung von Structs: **Klassen**
- Elemente einer **Klasse**:
  - **Attribute** oder **Datenelemente**: Daten (vgl. Structs)
  - **Methoden** oder **Elementfunktionen**: Funktionen, die das Verhalten der Klasse beschreiben und auf Datenelementen operieren (auch in Structs)

```
class Student
{
  public:
    string Name;                    // Datenelemente
    int    MatNr;
    double Note;

    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
}; // Semikolon nicht vergessen !!!
```

Klassen und Objekte

```
Student s; // erzeugt Variable s vom Typ Student

s.Name= "Hans␣Schlauberger";
s.Note= 1.3;
s.MatNr= 234567;

if ( s.hat_bestanden() )
{
  cout << "Herzlichen␣Glueckwunsch!" << endl;
}
```

- Datentyp `Student` heisst auch **Klasse**
- Variable `s` heisst auch **Objekt**
- bereits bekannte Klassen:
  `string, vector<...>, ifstream, ...`

Methodendefinition

- entweder *inline* innerhalb der Klassendefinition (`student.h`), nur für kurze Methoden sinnvoll

```
class Student
{
  ...
    bool hat_bestanden() const
    {
      return Note <= 4.0;
    }
  ...
};
```

- oder Deklaration in Klasse, Definition außerhalb (`student.cpp`)

```
  bool Student::hat_bestanden() const
  {
    return Note <= 4.0;
  }
```

const-Qualifizierung

```
class Student
{
  ...
    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
};
```

- `const` hinter Methode sichert zu, dass diese das Objekt nicht verändert
- `berechne_Note` kann nicht `const` sein, da Attribut `Note` verändert wird

```
const Student bob;          // bob ist konstant

if (bob.hat_bestanden())    // ok
  cout << "Glueckwunsch!"

bob.berechne_Note( 34);     // Compiler-Fehler, fuer bob
                            // duerfen nur const-Methoden
                            // aufgerufen werden
```

Objektzeiger

- Wdh.: Zugriff auf Attribute und Methoden über Objektzeiger:

```
Student s;
Student *s_zeiger= &s;

(*s_zeiger).Note= 3.3;
if ( (*s_zeiger).hat_bestanden() ) ...
```

einfachere Schreibweise mit `->` ist besser lesbar:

```
s_zeiger->Note= 3.3;
if ( s_zeiger->hat_bestanden() ) ...
```

- in jeder Methode wird implizit ein `this`-**Zeiger** übergeben, der auf das aufrufende Objekt zeigt

```
void Student::setzeName( string Name)
{
    this->Name= Name;
}
```

this-Zeiger – Beispiel

Beispiel: Zuweisungsoperator `operator=` als Methode

- Definition:

```
Student& Student::operator= ( const Student& s)
{
    Name= s.Name;
    this->MatNr= s.MatNr;
    Note= s.Note;
    return *this;
}
```

- Aufruf:

```
Student bobby;

bobby= bob;   // Zuweisung
```

  - Zuweisung `bobby= bob` bewirkt Aufruf `bobby.operator=( bob )`
  - `this` zeigt auf `bobby`

Überladen von Operatoren (*operator overloading*)

- praktisch wäre, für Objekte vom Typ `Datum` folgendes schreiben zu können:

```
if (MamasGeb < MeinGeb) // Datumsvergleich
    cout << "Mama ist aelter als ich.";
```

- möglich, wenn man folgende Funktion `operator<` definiert:

```
bool operator< (const Datum& a, const Datum& b)
{
    ...
}
```

Das nennt man **Operatorüberladung**.

- `MamasGeb < MeinGeb` ist dann äquivalent zum Funktionsaufruf `operator< ( MamasGeb, MeinGeb )`
- für viele Operatoren möglich, z.B. `+ - * / == >> <<` usw., je nach konkretem Datentyp sinnvoll

Überladen von Operatoren – Beispiel

Beispiel: Ein-/Ausgabeoperator für `Datum`:

- Wir wollen Code schreiben wie

```
ifstream ifs( "Datum.txt" );    // 3. Dezember 2014
Datum heute;
ifs >> heute;
cout << "Heute␣ist␣der␣" << heute << endl;
```

- Definition des **Ausgabeoperators** `<<` :

```
ostream& operator<< ( ostream& out, const Datum& d)
{
    out << d.Tag << ".␣" << d.Monat << "␣" << d.Jahr;
    return out;
}
```

- **Quiz:** Wie könnte der **Eingabeoperator** `>>` definiert werden?

```
istream& operator>> ( istream& in, Datum& d)
{
    // ???
}
```

Konstruktoren

- wird beim Erzeugen eines Objektes automatisch aufgerufen
- Zweck: initialisiert Attribute des neuen Objektes
- Klasse kann mehrere Konstruktoren haben:
  - Default-Konstruktor
    `Student() { ... }`
  - Kopierkonstruktor
    `Student( const Student& s) { ... }`
  - darüber hinaus weitere, allgemeine Konstruktoren möglich, z.B.
    `Student (string name, int matnr, double note= 5.0) { ... }`

```
Student a;                    // per Default-Konstruktor
Student b( "Hans Schlau", 234567, 1.3);
Student c( b);                // per Kopierkonstruktor
```

- Kein Wertparameter beim Kopierkonstruktor!
  `Student( Student s); //nicht erlaubt`
  Call by value würde eine Kopie `s` erzeugen, für die der Kopierkonstruktor benötigt wird.

Konstruktor – Beispiel

```
class Student
{ ...
  public:
    Student();
    Student( string name, int matnr, double note= 5.0);
    Student( const Student&);
};
```

```
Student::Student()
{ // Default-Konstruktor
    Name="Max␣Mustermann";  MatNr= 0;  Note= 5.0;
}

Student::Student( const Student& s)
{ // Kopier-Konstruktor
    Name= s.Name;  MatNr= s.MatNr;  Note= s.Note;
}

Student::Student( string name, int matnr, double note= 5.0)
{ // weiterer Konstruktor
    Name= name;  MatNr= matnr;  Note= note;
}
```

Objekte erzeugen durch verschiedene Konstruktoren

```
Student alice,  // Default-Konstruktor
        bob( "Bob␣Meier", 123456, 2.3),
        chris( "Chris␣Schmitz", 333333);

Student lisa( alice); // Kopierkonstruktor
Student bobby= bob;   // Kopierkonstruktor
```

- `Student bobby( bob);` bzw.
  `Student bobby= bob;` synonyme Syntax
  → Kopierkonstruktor
- `Student alice;` → Default-Konstruktor

⚠ `Student alice();` keine gültige Syntax für Default-Konstruktor-Aufruf
(verwechselbar mit Funktionsdeklaration!)

Destruktor

- wird automatisch beim Vernichten eines Objektes aufgerufen
- Zweck: z.B. Speicher eines dynamischen Feldes wieder freigeben
- jede Klasse hat genau *einen* Destruktor
  `~Student () { ... }`
- wird kein Destruktor deklariert, so gibt es immer einen *impliziten* Destruktor, der nichts tut (wie `~Student() {}`)
- Ebenso gibt es in jeder Klasse auch einen *impliziten*
  - Default-Konstruktor (falls keine Konstruktoren deklariert werden)
  - Kopierkonstruktor
  - Zuweisungsoperator `operator=`

```
class Array
{
    double* feld;
  public:
    Array( int laenge) { feld= new double[laenge]; }
    ~Array()           { delete[] feld; }
}; // Problem: impliziter Kop.konstr. und operator= ungeeignet
```

Zugriffskontrolle

- `private`-Bereich: Zugriff nur durch Methoden derselben Klasse
- `public`-Bereich: Zugriff uneingeschränkt über Objekt möglich

```
class Student
{
  private:
    string Name;                    // Attribute
    int    MatNr;
    double Note;

  public:
    // Konstruktor
    Student( string Name, double Note, int MatNr);
    // Destruktor
    ~Student();

    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
};
```

Zugriffskontrolle: Klassen vs. Strukturen

- `public`-/`private`-Bereiche auch in Strukturen möglich
- einziger Unterschied: Default-Verhalten, wenn nichts angegeben
  - Strukturen: standardmäßig `public`
  - Klassen: standardmäßig `private`

⇝ für objektorientierte Programmierung werden i.d.R. Klassen verwendet

```
struct Datum
{
    int Tag, Monat, Jahr;    // alles public
  private:
    double geheim;
};

class Date
{
    int Day, Month, Year;    // alles private
  public:
    int  GetMonth() const;
    void SetMonth( int month);
};
```

Vorteile der Zugriffskontrolle

- `private`-Bereich: Zugriff nur durch Methoden derselben Klasse
  - in der Regel für Attribute einer Klasse

  ⇝ Attribute sollen vor willkürlichem Zugriff geschützt werden
  - Zugriff auf private Attribute nur über public-Methoden der Klasse möglich
- `public`-Bereich: Zugriff uneingeschränkt über Objekt möglich
  - in der Regel für Methoden einer Klasse

  ⇝ ermöglicht Arbeiten mit einer Klasse über wohldefinierte **Schnittstelle**

⇝ Vorteile objektorientierter Programmierung:

- **Datenkapselung**: für den Nutzer einer Klasse tritt deren interne Arbeitsweise in den Hintergrund
- **einfache Wartung**: private Interna der Klasse können ausgetauscht werden ohne Auswirkung auf den Nutzer
- **Wiederverwendbarkeit** durch wohldefinierte öffentliche Schnittstelle