Programmieren
– C-Arrays & Zeiger

Reiner Nitsch
✉ r.nitsch@fbi.h-da.de

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

Datenstruktur Array /187/

- Eine Datenstruktur, die mehrere Objekte zusammenfasst, nennt man Array (deutsch: Feld oder Reihe).
- Ein homogenes Array liegt vor, wenn alle Objekte des Array vom gleichen Typ sind.
- Beim statischen Array muss die maximale Objektanzahl zur Compile-Zeit festgelegt werden und kann zur Laufzeit nicht mehr verändert werden.
- Beim dynamischen Array kann die maximale Objektanzahl zur Laufzeit dem Bedarf angepasst werden. Beispiel: Vektoren (`class vector`) sind homogene dynamische Array.
- C-Arrays sind homogene statische Array, also viel primitiver als Vektoren.
- C-Arrays und Vektoren speichern ihre Datenelemente in einem zusammenhängenden Speicherbereich, d.h. physikalisch-sequentiell.

Definition von C-Arrays:

Syntaxdiagramm

Beispiel

```
int      intArray    [          5          ]   ;
```

oder besser

```
const int MAX_SIZE = 5;
int intArray[MAX_SIZE];
```

muß ein ganzzahliger Ausdruck mit konstantem Ergebnis sein

```
Index  0   1   2   3   4
```

intArray[MAX_SIZE];

physikalisch-sequentielle Speicherung

C-Array

- **Deklaration von C-Arrays in Klassen**

```
class Array{
public:
   enum { MAX_SIZE=5 }
private:
   int field[MAX_SIZE];
   …
};
```

Aufzählungstyp definiert symbolische Konstante 'MAX_SIZE' mit Sichtbarkeitskeitsbereich 'class Array' (Andere Möglichkeit gibt es nicht!)

Zugriff auf Konstante 'MAX_SIZE' von aussen

```
void main()
{
   Array a;
   if (Array::MAX_SIZE …)
   …
}
```

C-Array

- Zugriff auf Feld-Elemente mit Index-Operator [ ]

```
int intArray[MAX_SIZE];
intArray[0] = 4;
intArray[2] = intArray[0];
intArray[intArray[0]-1] = 15;
                =3
cin >> intArray[1]; //Eingabe 8
```

	Speicher	Adressen
[-1]	3	100
[0]	4	104
[1]	8	108
[2]	4	112
[3]	15	116
[4]	?	120
[5]	5	124

- Index muss vom Ganzzahltyp sein; gültiger Indexbereich `0 … MAX_SIZE - 1;` kann aber muss nicht vom Compiler überwacht werden!

- bietet keinerlei Schutzfunktionen (z.B. Kapselung, Indexüberwachung)

```
intArray[-1] = 3;
cin >> i; //Eingabe 4
intArray[i+1] = 5;
```

Indexfehler kann der Compiler nicht erkennen!

- Schutzfunktionen müssen die Zugriffsfunktionen übernehmen. Dazu wird die Konstante 'MAX_SIZE' benötigt:

```
if ( index>=0 && index<MAX_SIZE ) intArray[index] = 6;
```

C-Array

- Initialisierung von Arrays

a) außerhalb von Klassen

```
const int MAX_SIZE = 5;
int intArray1[MAX_SIZE] = { 2,4,7,3,1 };
int intArray2[] =         { 1,3,8,4,2 };
int intArray3[MAX_SIZE] = { 10,5 };
```

Initialisierungsliste

Es wird ein passendes Array erzeugt
Nachteil: Benannte Konstante MAX_SIZE fehlt!

Rest wird mit 0 initialisiert

b) innerhalb von Klassen

- Elementweise im Konstruktor

```
Array::Array() {   //Konstruktor
  for ( int index=0 ; index<MAX_SIZE ; index++ )
    field[index] = 0;
}
```

Membervariable müssen weiterhin elementweise initialisiert werden!

C-Array

- Zuweisung bei Arrays

```
intArray1 = intArray2;
```

Fehler: Zuweisungsoperator ist für Arrays nicht definiert! ☹

- **Abhilfe**: Elementweises kopieren

```
for( int index=0 ; index<MAX_SIZE ; ++index )
    intArray[index] = intArray2[index];
```

- C-Arrays und `sizeof`-Operator (Bestimmung der Array-Größe in Bytes)

```
const int MAX_SIZE=5;
int numberOfElements, sizeOfDoubleArray, sizeOfDouble;
double doubleArray[MAX_SIZE];
sizeOfDoubleArray = sizeof(doubleArray);
sizeOfDouble = sizeof(double);
numberOfElements = sizeOfDoubleArray/sizeOfDouble;
```

Ergebnisse
40
8
5

C-Array an Funktion übergeben

- Aufgabe: Ein Array mit Brüchen in aufsteigender Reihenfolge füllen.

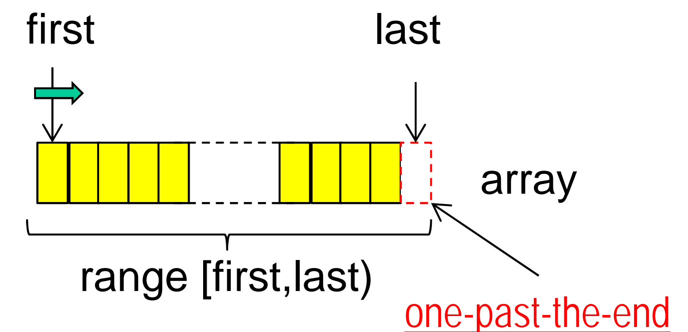
```
int main() {
  const int MAX = 1000;
  int a[MAX];
  generate(a,0,MAX);
  return 0;
}
```

Ein Array wird als Parameter an eine Funktion übergeben durch Angabe des Namens ohne [ ]

Sets the value of the elements in the range *[first,last)* to the value returned by successive calls to *next()*.

```
void generate(int array[],int first,int last)
{
  for( ;first!=last;++first )
    array[first] = next();
}
```

Semantik der Bereichsangabe
first: erstes bearbeitetes Element
last: erstes nicht mehr bearbeitetes Element
→ one-past-the-end Element



Beachte: `generate` ist nur übersetzbar, wenn die Funktion `next()` definiert ist! → `generate` ist von `next` abhängig.

Funktionen mit Gedächtnis /105/

```
int next() {
  int n=0;
  return n++;
}
```

Return-Werte sukzessiver Aufrufe? 0 0 0 0 ….
Grund: Die Funktion hat kein Gedächtnis!
(Speicherklasse von `n` ist "automatisch")

```
int next() {
  static int n=0;
  return n++;
}
```

Schlüsselwort `static` erweitert den Gültigkeitsbereich von "Block" auf "Programm". `static`-Variablen in Funktionen bekommen einen festen Speicherplatz und werden automatisch mit 0 initialisiert.

Return-Werte sukzessiver Aufrufe? 0 1 2 3 ….

Nachteil von `static`-Variablen in Funktionen:

- Ein erneutes Setzen eines anderen Startwertes (hier: `n=0`) ist nicht möglich!

Abhilfe: Gedächtnis eines Objekts statt dem einer Funktion verwenden → nächste Seite.

Einschränkungen bei C-Arrays

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

- "pass by value" ist mit Arrays nicht möglich
- Arrays können keine "return values" sein
- Arrays können keine L-values" sein

Mehrdimensionales Array

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
int matrix[5][10];

int m[2][3] = {{1,2,3},{4,5,6}};
```

2-dimensionales Feld
oder
1-dimensionales Feld mit
5 1-dimensionalen Feldern als Objekten

```
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
```

```
1 2 3
4 5 6
```

erstgenannte Dimension
entspricht äußerster Klammer

Stundenplan

	Mo	Di	Mi	Do	Fr	Sa	So
1							
2		NW					
3		PG1		PG1			
4				PG1L			
5				PG1L			

```
enum Day {MO,TU,WE,TH,FR,SA,SU}
enum Subject {PG1, PG1L, NW, …}
Subject timetable[ 5 ][ SU+1 ];
timetable[ 2 ][ TH ] = PG1;
timetable[ 1 ][ TU ] = NW;
//Stundenplan zeilenweise drucken
for( int block=0; block < 5; block++ )
{
  for( Day d=MO; d<=SU; d++)
    cout << timetable[block][d];
  cout << endl;
}
```

Mehrdimensionales Array

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

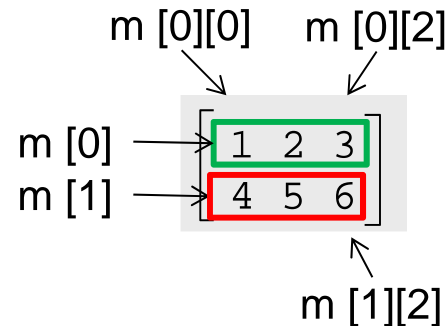
```
int matrix[2][3];
```

2-dimensionales Array ohne Initialisierung

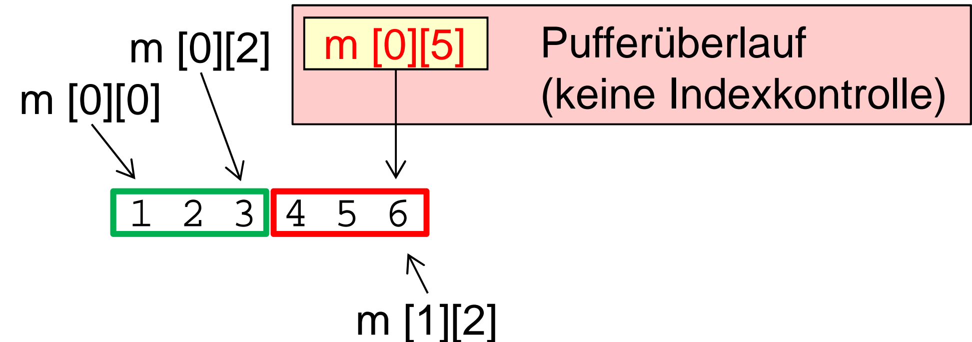
```
int matrix[2][3] = {{1,2,3},{4,5,6}};
```

2-dimensionales Array mit Initialisierung

erstgenannte Dimension entspricht äußerster Klammer



Logische Sicht



Speichersicht

Beispiel:

Stundenplan

	Mo	Di	Mi	Do	Fr	Sa	So
1							
2		NW					
3		PG1		PG1			
4				PG1L			
5				PG1L			

```
enum Day {MO,DI,MI,DO,FR,SA,SO}
enum Subject {PG1, PG1L, NW, …}
Subject timetable[ 5 ][ SO+1 ];
timetable[ 2 ][ DI ] = PG1;
timetable[ 1 ][ DI ] = NW;
//Stundenplan zeilenweise drucken
for( int block=0; block < 5; block++ ) {
  for( Day d=MO; d<=SO; d++)
    cout << timetable[block][d];
  cout << endl;
}
```

Zeigervariable (pointer)

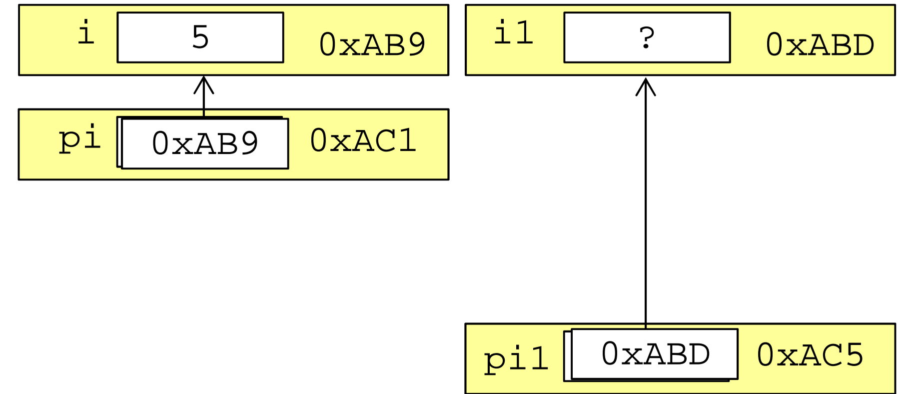
- werden zur indirekten Adressierung benutzt
- werden in C++ so definiert

```
int i   = 5, i1;

int* pi;
pi = &i;

int * pi1 = &i1;
```

Definition von Zeigervariablen (→ `int* pi;`)

Adressoperator (lies: "gib Adresse von") (→ `&i`)

Definition **und** Initialisierung (→ `int * pi1 = &i1;`)



- enthalten Adressen eines Objekts anstelle des Objekts selbst (Wertebereich und Darstellung maschinenabhängig, z.B. 32 Bit.)

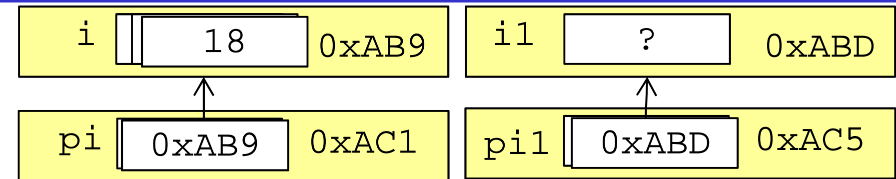
Speichersicht

Legende: Name | Wert | Adresse

- haben
  - → einen Namen — pi
  - → einen Speicherplatz — 4 Byte ab Adresse 0xAC1
  - → einen Typ — int*; lies: "Zeiger auf ein int-Objekt"

Zeigervariable (pointer)

- können den Wert des Objekts, auf das sie zeigen, lesen




```
i = *pi + 2;      // i=7
cout << *pi;      // Ausgabe: 7
```

Dereferenzierungsoperator bei R-Wert (lies: "Gib Wert von Adresse")

- können den Wert des Objekts, auf das sie zeigen, ändern

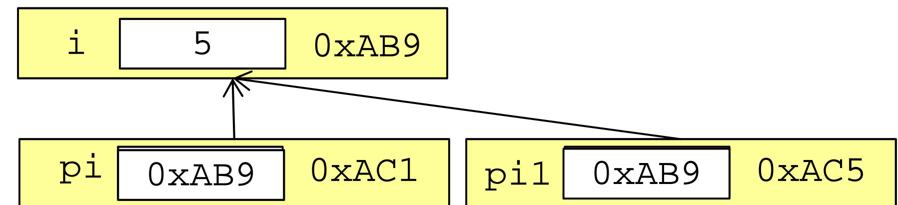
```
*pi = i + 2;      // -> i = 9
*pi += *pi;       // -> i =18
```

Dereferenzierungsoperator bei L-Wert (lies: "Setze Wert bei Adresse")

- können zur Laufzeit neue Adressen zugewiesen bekommen, um auf andere Objekte zu zeigen

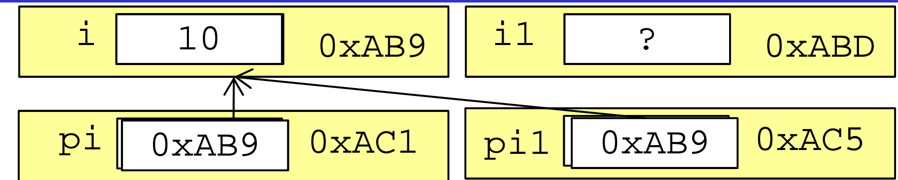
```
pi1 = pi;
```

Beide Zeiger verweisen auf denselben Wert



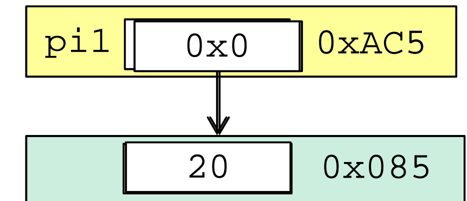
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

Zeigervariable (pointer)




```
i = 10;

// Vergleich der Objekt-Werte:
    cout<< (*pi == *pi1);        true
              10       10
// Vergleich der Zeiger-Werte:
    cout << (pi == pi1);         true
            0xAB9   0xAB9
// Vergleich der Zeiger-Adressen:
    cout << (&pi == &pi1);       false
            0xAC1   0xAC5
```

- können zur Laufzeit Adressen von Objekten zugewiesen bekommen, die erst zur Laufzeit erzeugt werden




```
pi1 = new int;
*pi1 = 20;
cout << *pi1;
delete pi1;
```

- `pi1 = new int;` — reserviert (allokiert) Speicher für ein int-Objekt (4 Byte) im dyn. Speicher (Heap)
- `cout << *pi1;` — Ausgabe: 20
- `delete pi1;` — gibt dyn. Speicher an System zurück; löscht nicht die Variable `pi1`

`pi1` ist jetzt ein "hängender Zeiger", weil er auf kein gültiges Objekt mehr zeigt

```
pi1 = NULL;
```

Jetzt kein "hängender Zeiger" mehr!

Zeigervariable (pointer)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

- bekommen zur Compile-Zeit nur Speicher für eine Objektadresse, aber nicht den Speicher für das Objekt selbst zugewiesen. Sie werden vom C++ Compiler nicht initialisiert.

```
double *pd;
*pd = 2.0;
```

falsch: nicht initialisiert; zeigt irgendwo hin

speichert 8 Byte irgendwo, evtl. mitten im Programmcode

→ Absturz zur Laufzeit vorprogrammiert und evtl. nicht reproduzierbar
→ Fehler schwer zu finden

Deshalb: Zeigervariable immer initialisieren!

```
double *pd = 0;
```

richtig

- `0´ zeigt an, dass sich ein Zeiger noch nicht oder nicht mehr auf ein definiertes Objekt zeigt.
  ⇒wenigstens reproduzierbare Abstürze
  ⇒ guter Programmierstil

Zeigervariable (pointer)

- Möchten Sie trotzdem auf Abstürze nicht verzichten; so einfach geht's!

```
double *pd = 0;
*pd = 3.5;          // Absturz bei schreibendem
double d = *pd;     // und lesendem Zugriff
```

- in beiden Fällen Zugriff über illegalen Zeiger
  - ⇒ "Schutzverletzung (Access Violation) unter Windows
  - ⇒ kein Betriebssystemfehler; das tut hier seine Arbeit

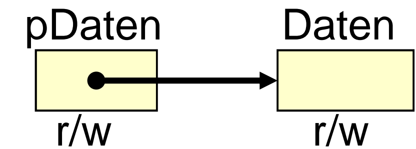
- unterliegen der Typkontrolle durch den Compiler

```
pi = &i;        // ok
pd = &i;        // Fehler
double*  int*
pd = 0xB001;    // Fehler
double*  long
pd = pi;        // Fehler
double*  int*
```

Anwendung von `const` auf Zeiger

1. r/w-Zeiger auf r/w-Daten

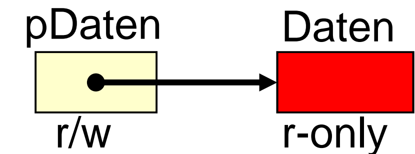
```
char Daten = 'x';
char *pDaten = &Daten;
```



2. r/w-Zeiger auf read-only-Daten

```
const char *pDaten = &Daten;
*pDaten = 'y';
```

Fehler: read-only-Daten

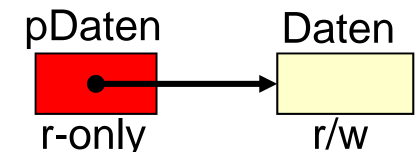


3. read-only-Zeiger auf r/w-Daten

```
char * const pDaten = &Daten;
char andereDaten = 'y';
pDaten = &andereDaten;
```

Initialisierung notwendig

Fehler



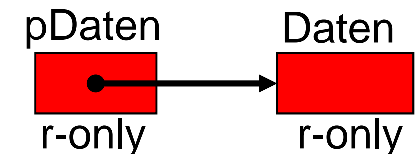
4. read-only-Zeiger auf read-only-Daten

```
const char * const pDaten = &Daten;
*pDaten = 'y';
pDaten = &andereDaten;
```

ok

Fehler: read-only Daten

Fehler: read-only Pointer

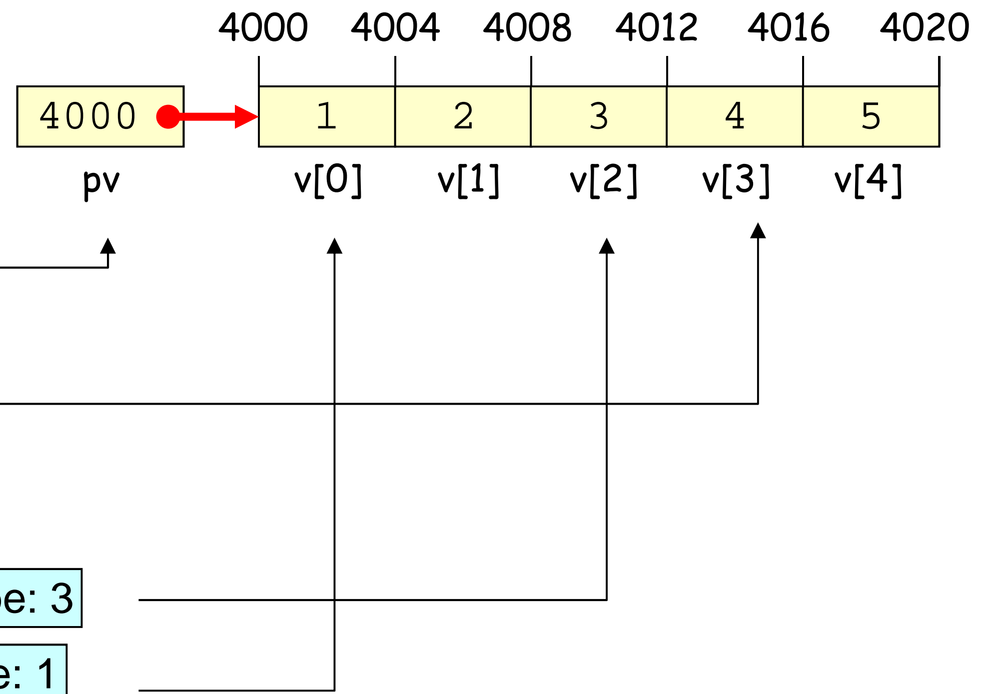


Ausdrücke mit Zeigern und Zeigerarithmetik

Für Zeigervariable sind folgende Operatoren definiert

```
++   +   +=   [ ]   <   <=   !=
--   -   -=   >    >=
```

Beispiele:




```
int v[5] = {1,2,3,4,5};
int *pv = v;
int *p;
p = pv + 3;
cout << *p << *(pv+3);      // Ausgabe: 4
// d.h. *(pv+3) und v[3] sind äquivalent
cout << *(--p);             // Ausgabe: 3
cout << *(p - 2);           // Ausgabe: 1
cout << p - pv;             // Ausgabe: 2 (2 Elemente zwischen p und pv)
```

- Zeiger können wie Arrays indiziert werden

```
cout << v[2];        // Ausgabe: 3
cout << pv[2];       // Ausgabe: 3
cout << *(pv+2);     // Ausgabe: 3
```

d.h. `pv[2]` und `*(pv+2)` sind äquivalent

Aufgabe 1

- Der Vektor v sei wie folgt definiert

```
int v[] = { 10, 20, 30, 40 }, i, *pv = 0;
```

- Was geben die folgenden Anweisungen auf dem Bildschirm aus?

```
for ( pv = v; pv <= v+3; pv++)
    cout << *pv << ' ';
```

10 20 30 40

```
for( pv = v, i = 1; i <= 3; i++)
    cout << pv[i] << ' ';
```

20 30 40

```
for( pv = v, i = 0; pv+i < &v[3]; i++)
    cout << *(pv+i) << ' ';
```

10 20 30

```
for( pv = v+3; pv > v; --pv)
    cout << v[pv-v] << ' ';
```

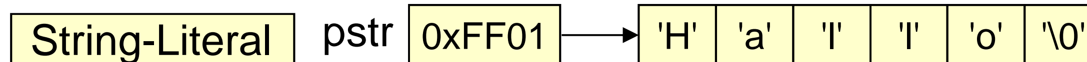
40 30 20

C-Zeichenketten /191/

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

- sind ein Erbe von C
- sind spezielle 1-dimensionale `char`-Arrays mit `'\0'` als Endemarke

String-Literal




```
const char* pstr = "Hallo";
```
pstr wird definiert und initialisiert

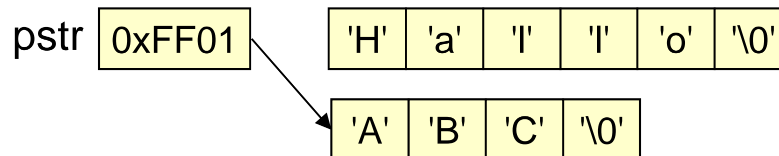
```
cout << pstr[0];
```
H — pstr zeigt auf das 1. Zeichen der Zeichenkette

```
pstr[0] = 'h';
```
Fehler: Die Zeichenkette ist in einem schreibgeschützen Speicherbereich abgelegt.

```
int i = *(pstr+5);
```
i=0; — Das Ende der Zeichenkette ist durch den ASCII-Wert 0 gekennzeichnet. Diesen hängt der Compiler automatisch an.

```
cout << pstr << endl;
```
Hallo — Der Operator << weiß, dass `pstr` nicht wie ein Zeiger zu behandeln ist, sondern eine '\0'-terminierte Zeichenkette referenziert.

```
pstr = "ABC";
```
Eine neue Zuweisung ist möglich, aber der Bezug zur vorherigen Zeichenkette geht dabei verloren (s. Abb.)



Verwaistes Objekt!

```
strlen(pstr);
```
5 — Header <cstring> definiert viele Funktionen zur Verarbeitung von C-Strings(z.B. `strlen` gibt Anzahl Zeichen ohne '\0' zurück!)

char-Arrays - häufige Fehler

```
char* ptext;

cin >> ptext;
```

Achtung: Ein nicht initialisierter `char`-Zeiger besitzt nur Speicherplatz für eine Adresse
`ptext` zeigt auf eine undefinierte Stelle im Speicher. Der Speicherinhalt wird ab dieser Stelle durch die Benutzereingabe überschrieben und automatisch '\0' angehängt (häufiger Anfängerfehler!)

```
char strasse[12],name[8];

cin >> name;     //Eingabe: "Schmachter"
cin >> strasse; // Eingabe: "Fetzenstrasse"
```

Richtig: Der Programmierer muss Speicher in Form eines `char`-Arrays bereitstellen.

Der Name eines C-Arrays ist immer der Bezeichner für die Adr. seines 1. Elements!

Funktionsweise des Einleseoperators >> bei `char`-Arrays

- Die Benutzereingabe wird gelesen
- Führende Trennzeichen werden ignoriert und entfernt
- Alle folgenden Nicht-Trennzeichen werden bis zum nächsten Trennzeichen in die Elemente des char-Arrays kopiert
- Das abschließende Trennzeichen wird nicht entfernt
- **An das letzte kopierte Element wird ein '\0'-Zeichen angehängt**

Achtung: Weder Compiler noch `cin` überwachen hier die Indexgrenzen!
Deshalb Gefahr von **Pufferüberlauf**

```
cout << name;
```

Ausgabe: `SchmachtFetzenstrasse` wegen Pufferüberlauf!

char-Arrays werden vom Ausgabeoperator << wie C-Strings behandelt (Ausgabe bis zur Endemarke '\0')

```
const int ZMAX=100;
char text[ZMAX];
```

Richtig: Der Programmierer muss ausreichend Speicher in Form eines char-Arrays bereitstellen.

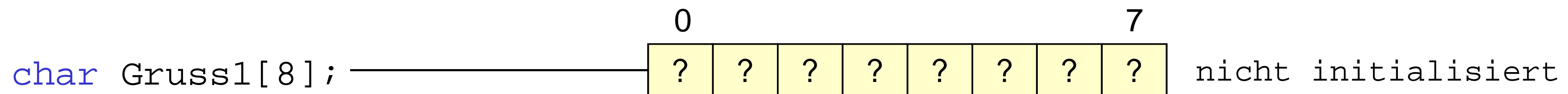
```
cin >> text;
```

Relativ sicher: Liest bis zum ersten Zwischenraumzeichen und hängt '\0' an.

```
cin.getline(text,ZMAX,'\n');
```

Sicher: Liest Benutzereingabe bis zum Trennzeichen (hier: '\n'). Das Trennzeichen wird entfernt aber nicht gespeichert. Nach max. ZMAX-1 Zeichen wird '\0' angehängt und abgebrochen.

```
char Gruss1[8];
```

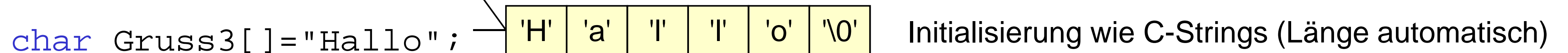


nicht initialisiert

```
char Gruss2[6]={'H','a','l','l','o','\0'};
```

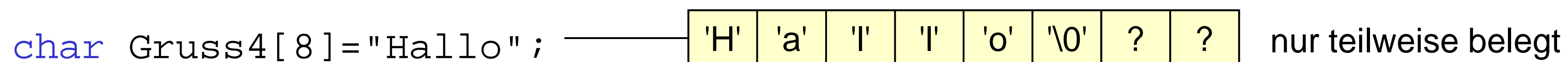
Initialisierung wie C-Arrays

```
char Gruss3[]="Hallo";
```



Initialisierung wie C-Strings (Länge automatisch)

```
char Gruss4[8]="Hallo";
```



nur teilweise belegt

```
Gruss1 = Gruss4;
Gruss3 = "Tschau";
```

(durchgestrichen)

Fehler: C-Arrays können **keine L-Werte** sein, weil **Zuweisung** bei Arrays **nicht definiert ist.**

Lernkontrolle – Was wird ausgegeben? Was ist falsch?

- Nocheinmal: Zeiger und Arrays sind in C++ eng verwandt!
  - Der Name des Arrays ist für den Compiler (im exe-File existiert der Arrayname nicht) die Adresse (= read-only-Zeiger) des ersten Array-Elements:

```
char land[10] = "Hessen";
cout << (land == &land[0]) << endl;
```

Ausgabe: true

```
float preis[6];
cout << preis << &preis[0] << endl;
```

gibt 2-mal identische Hexadressen aus

```
cout << land << &land[0] << endl;
```

gibt 2-mal "Hessen" aus

```
cout << *land << land[0] << endl;
```

gibt 2-mal 'H' aus

```
char *pc = "Gewinner ist";
```

const char *

Initialisierung eines Zeigers mit einem C-String; pc zeigt auf ‚G'

```
cout << pc << endl;
```

Ausgabe: Gewinner ist

```
cout << *pc << endl;
```

Ausgabe: ‚G'

Lernkontrolle – Was wird ausgegeben? Was ist falsch?

```
pc = "Portugal";
cout << pc << endl;
cout << *pc << endl;
*pc = "Holland";
```

// ok: Zuweisung eines `const char*` an einen `char*`

// Ausgabe: Portugal

Ausgabe: P

Fehler: Zuweisung eines `const char*` an ein `char`

```
pc = land;
cout << pc << endl;
```

//ok! Zur Erinnerung: **`char land[10] = "Hessen";`**

//Ausgabe: Hessen

```
land = pc;
```

// Fehler: C-Array kann kein L-Wert sein

```
land = new char[20];
```

// Fehler: C-Array kann kein L-Wert sein

```
pc = new char[20];
delete pc;
```

// ok!

// gibt dynamischen Speicher wieder frei

Mehrdimensionales char-Array

```
char words[3][6] = { "Pferd", "Sau", "Hund" };

cout << words[0] << endl;
cout << words[1] << endl;
cout << words[2] << endl;
```

```
Pferd
Sau
Hund
```

```
P f e r d 0
S a u 0 ? ?
H u n d 0 ?
```

```
P f e r d 0 S a u 0 ? ? H u n d 0 ?
```

```
cin >> words[0];        // Eingabe: Dackel
```

```
D a c k e l 0 a u 0 ? ? H u n d 0 ?
```

```
cout << words[0] << endl;
cout << words[1] << endl;
cout << words[2] << endl;
```

```
Dackel

Hund
```

```
D a c k e l
0 a u 0 ? ?
H u n d 0 ?
```

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

Beispiel:

- Jede Methode einer Klasse kennt zur Laufzeit das Objekt, das sie aufgerufen hat.
- Dazu wird beim Methodenaufruf, auch ein (verdeckter) Zeiger auf das aufrufende Objekt auf dem Stack übergeben. Dieser Zeiger heißt **this**!

```
int main() {
    Fraction b(1,2);
    cout << b.toFloat();
}

float Fraction::toFloat() const {
    return float(this->z)/this->n;
}
```

Zeiger `this` zeigt immer auf die Adresse des aufrufenden Objekts (hier: `this=&b`)

`this` optional, wenn eindeutig!

```
Fraction::Fraction( int z, int n ) {
    z = z;
    n = n;
    this->z = z;
    this->n = n;
}
```

(`z = z;` und `n = n;` durchgestrichen) Fehler: Nicht eindeutig

Ok, weil eindeutig

Hauptanwendungen von Zeigern - pass by reference

- Ersatz für Referenzparameter in Parameterlisten von Funktionen/Methoden ("pass by reference" gab's in C nicht!)

Beispiel:

```
void Becher::fuellen( int wieviel, Fass & *pf)
{
  menge += wieviel;
  f.abzapfen(wieviel);
  (*pf).abzapfen(wieviel);
}
```

Version mit Pointer?

Klammern, weil '.'-Operator höheren Vorrang als '*'-Op. hat

Hinweis:

```
(*pf).abzapfen(wieviel);
pf->abzapfen(wieviel);
```

und sind gleichwertig

kompakterer Dereferenzierungsoperator

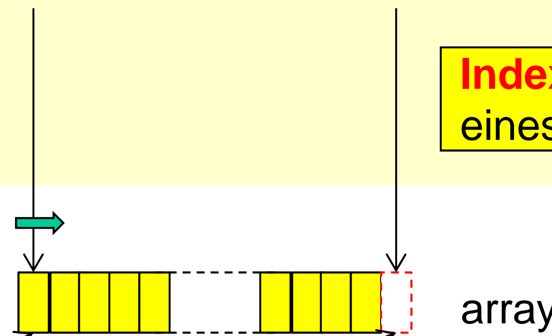
Indirektion ⇔ Zugriff auf Objekte und deren Memberfunktionen über Zeiger
Ist einSchlüsselkonzept von C++

Hauptanwendungen von Zeigern - Übergabe von (Teil-)Containern

- Mit Zeigern kann man auch Teilbereiche eines Containers übergeben.
- Beispiel:

```
void generate( int array[], int first, int last )
{
  for( ;first!=last;++first )
    array[first] = next();
}
```

Indexbasierte Spezifikation eines (Teil-)Containers

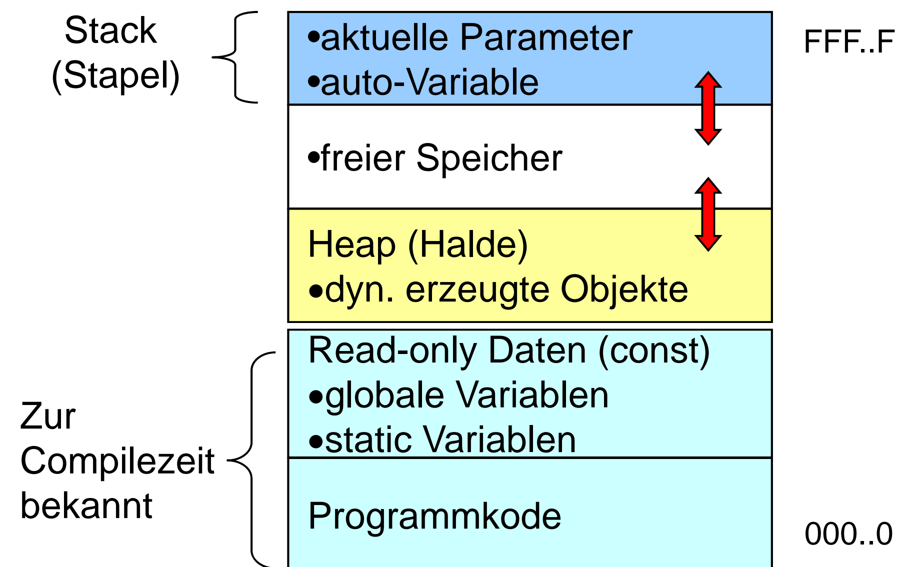



```
void generate( int* pfirst, int* plast )
{
  for( ; pfirst!=plast; ++pfirst )
     *pfirst = next();
}
```

Zeigerbasierte Spezifikation eines (Teil-)Containers (→schmalere Schnittstelle)

Hauptanwendungen von Zeigern - dynamische Speicherverwaltung

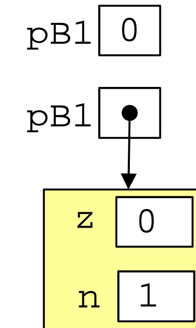
- Die bisher behandelten Datentypen waren statisch – der Compiler mußte und konnte den Speicherplatzbedarf ermitteln.
- In vielen Anwendungen ist der erforderliche Platzbedarf aber erst zur Laufzeit des Programms bekannt; z. B. wenn der Platz von der Anzahl der Benutzereingaben abhängt oder der Größe einer einzulesenden Datei.
- Die Größe eines statischen Datentyps muß in diesem Fall dem worst-case angepaßt sein. Im Regelfall bleibt dann ein Großteil des reservierten Speicherplatzes ungenutzt.
- Besser ist es, wenn man den Speicher erst zur Laufzeit, entsprechend dem tatsächlichen Bedarf anfordern (allokieren) kann.
- Ein Programm wird vom Betriebssystem (je nach Betriebssystem unterschiedlich) im Speicher etwa wie folgt abgelegt:

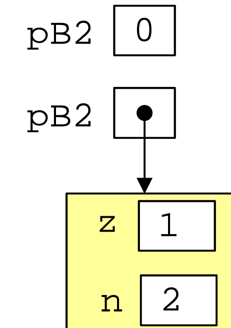


Dynamische Speicherverwaltung - Beispiel mit Zeigern


```
int main() {
Fraction *pB1 = 0, *pB2 = 0, *pB3 = 0;

//Dynamisch Objekte erzeugen
//Standard-Konstruktor wird aufgerufen
pB1 = new Fraction;
pB2 = new Fraction(1/2);
```

Speicher auf Heap belegen und **Konstruktor** aufrufen



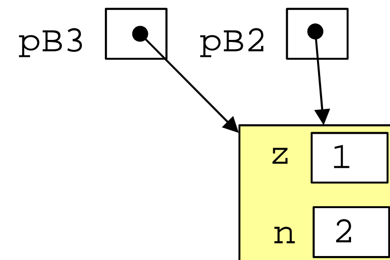



```
pB1.add(pB2);    // Fehler! Warum?
/*Richtig*/ (*pB1).add(*pB2); /*oder*/ pB1->add(*pB2);
```

Objekte statt Pointer werden erwartet!

```
*pB3 = pB1->add(*pB2);    // Fehler! Warum?
/*Richtig*/ pB3 = new Fraction; *pB3 = pB1->add(*pB2);
/*oder besser mit Copykonstruktor*/ pB3 = new Fraction(pB1->add(*pB2));
*pB3 = *pB1 + *pB2; // globaler operator+ : ohne Klammern weil Punkt- vor Strichoperator
cout << pB3->toString();
cout << *pB3; // globaler operator<<

//B3 zu Kopie von B2 machen
pB3 = pB2;    // Fehler! Warum?
*pB3 = *pB2;    // Richtig
//Speicher wieder freigeben
delete pB1; delete pB2; delete pB3; }
```

pB3 verweist noch auf kein Objekt (s.o.: pB3=0)!



Nur der Zeiger pB2 wurde kopiert, nicht aber das Objekt auf das pB2 zeigt!

Destruktoren aufrufen und Speicherplatz freigeben

Dynamische Datenstrukturen - Beispiel mit Zeiger-Arrays

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
int main() {
  const int MAX = 4
  Fraction* pB[MAX] = {0};
  //Dynamisch Objekte erzeugen
  //Standard-Konstruktor wird aufgerufen
  pB[0] = new Fraction;
  pB[1] = new Fraction(1,4);
  pB[2] = new Fraction(1,2);
  int i=MAX;
  Fraction sum(0,1);
  while(i) {
     cout << *pB[--i] << endl;
     sum = sum + *pB[i];
  }
```

Häufige Fehler!

```
  //Kopie von Bruch1 erzeugen
  pB[2] = new Fraction(*pB[1]);
  //Speicher wieder freigeben
  for( i=0; i<MAX; ++i )
     delete pB[i];
}
```

Array von Zeigern auf Fraction-Objekte wird definiert und mit Null initialisiert. Achtung: hier werden keine Fraction-Objekte erzeugt!

Fehler: `Fraction(1,2)` ist jetzt ein **"verwaistes" Objekt**, auf das kein Zeiger mehr zeigt -> kein delete mehr möglich! **Richtig**:
`delete pB[2]; pB[2] = new Fraction(*pB[1]);`

Destruktoren werden aufgerufen und Speicherplatz freigegeben

Fehler: es sind 3 "hängende Zeiger" entstanden, die auf nicht mehr reservierten Speicherplatz zeigen! **Richtig**:
`{ delete pB[i]; pB[i] = 0; }`

Hauptanwendungen von Zeigern - Zeiger auf Funktionen

- in C++ ist ein Funktionsname ein konstanter Zeiger auf den Maschinencode einer Funktion
- Funktionszeiger können einer anderen Funktion als Parameter übergeben werden. Damit ist es möglich, erst zur Laufzeit eines Programms zu bestimmen, welche Funktion ausgeführt wird (engl. **late binding**, **dynamic binding**).
- werden in C++ wie folgt definiert:

```
double func(double);
// Definition eines Zeigers auf diese Funktion
double (*pf)(double);
```

ist der Prototyp einer Funktion, die einen `double`-Parameter erwartet und einen `double`-Wert zurück gibt.

Dies ist eine Zeiger-Variable mit dem **Namen** **`pf`**, die auf eine solche Funktion zeigt.

Klammerung ist notwendig, weil der Compiler dies sonst für die Deklaration der Funktion **`double* pf(double);`** hält.

- sollten wie alle Zeigervariablen stets initialisiert werden:

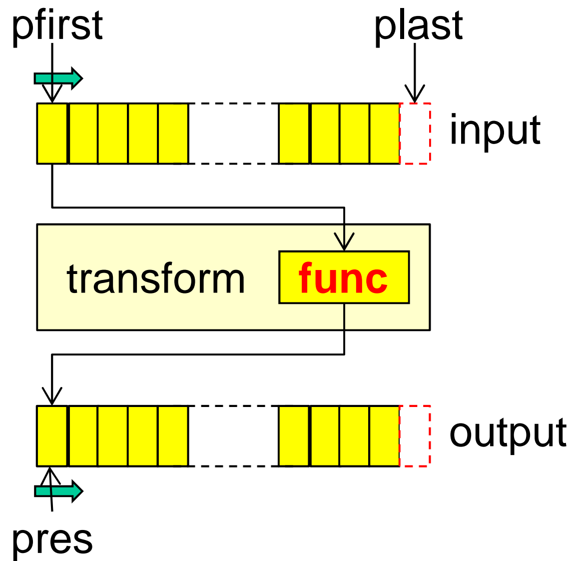
```
double (*pf)(double) = 0;
```

- können zur Laufzeit eine Adresse zugewiesen bekommen

```
pf = sin;
```

`sin` ist die Adresse des Maschinencodes der sin-Funktion `double sin(double)` aus der Mathematik-Bibliothek

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch




```
// Apply function to range
double* transform(double* pfirst, double* plast, double* pres, double (*func)(double))
{
   while( pfirst<plast ) {
      *pres = func( *pfirst );
      ++pfirst; ++pres;
   }
   return pres;
}
```

unary function (erwartet 1 Operanden)

```
#include <cmath>
#include <algorithm>
int main()
{
   double x[8] = { .4, .8, 1.2, 1.6, 2., 2.4, 2.8, 3.2 };
   double y1[8] = {0.}; double y2[8] = {0.};

   ::transform( &x[0], &x[8], &y1[0], sin );

   ::transform( &x[0], &x[8], &y2[0], cos );

   std::transform( x, &x[8], y2, cos );
}
```

Speichert sin(x[i]), i=0…8 in y1[i]

Speichert cos(x[i]), i=0…8 in y2[i]

Funktion `transform` gibt es auch in STL

Name des Arrays ist ein Zeiger auf 1. Element

Such-Algorithmen - Lineare Suche

- Prinzip der linearen Suche:
  - Betrachte jedes Element im Suchbereich
  - Vergleiche jedes Element im Suchbereich mit dem Suchwert
  - Wenn gefunden (*Suchtreffer*), gib Index oder Zeiger auf Suchtreffer zurück
  - Wenn nicht gefunden (*Suchfehler*), gib Sentinel zurück.

```
// Suche nach einem bestimmten Wert
int* find( int* pfirst, int* plast, int value ){
 while( pfirst<plast )
   if( *pfirst == value )
      return pfirst;   // Suchtreffer
   else
      ++pfirst;
 return plast;  // sentinel für Suchfehler
}
```

Die Zeiger auf Containerelemente werden auch als "**Iteratoren**" bezeichnet, weil sie durch iteratives Weiterschalten Zugriff auf jedes Element erhalten.

Welche Anforderungen stellt diese Algorithmus an die Iteratoren?

Welche Anforderungen stellt diese Implementierung an den Elementtyp?

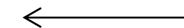
Such-Algorithmen - Lineare Suche nach einer Eigenschaft

```
// Suche nach einer bestimmten Eigenschaft
int* find_if( int* pfirst, int* plast ) {
  while( pfirst<plast )
    if(  pred( *pfirst )  )
       return pfirst;    // Suchtreffer
    else
       ++pfirst;
  return plast;  // sentinel für Suchfehler
}
```

Idee: Der Test auf Existenz einer bestimmten Eigenschaft (engl.: predicate) wird an eine Funktion delegiert!

Funktionen mit diesem Schnittstellenprofil (1 Parameter, gibt Wahrheitswert zurück) nennt man "**unäre Predicatefunktion**"

Welche Anforderungen stellt der Algorithmus an die Funktion `pred`?



```
bool pred( int i ) { return (i%2==0) ? true:false; }

int a[5] = { 1, 2, 3, 4, 5 };
int* pi = NULL;
pi = find( &a[0], &a[5] );
cout << *pi << endl;
pi = find( pi+1, &a[5] );
cout << *pi << endl;
```

Ausgabe: 2

Ausgabe: 4

Such-Algorithmen - Lineare Suche nach einer Eigenschaft

Definiere einen Funktionszeiger, der auf diese unäre Predicatefunktion zeigen kann:

```
bool (*f)( int )
```

Damit der Suchalgorithmus nach unterschiedlichen Eigenschaften suchen kann, bekommt er zusätzlich einen Zeiger auf die Pedicatefunktion übergeben:

```
int* find_if( int* pfirst, int* plast, bool (*f)( int ) ) {
   while( pfirst<plast )
      if( f(*pfirst) )
         return pfirst; // Suchtreffer
      else
         ++pfirst;
   return plast;  // sentinel für Suchfehler
}
```

Hier wird die vom Programmierer bereit-gestellte Predicatefkt. aufgerufen. Diese Technik nennt man "**callback**"

```
bool isEven( int i ) { return (i%2==0) ? true:false; }
bool isOdd ( int i ) { return (i%2==0) ? false:true; }
```

```
int a[5] = { 1, 2, 3, 4, 5 };
int* pi = NULL;
pi = find_if( &a[0], &a[5], isEven );
cout << *pi << endl;
pi = find_if( pi+1, &a[5], isOdd );
cout << *pi << endl;
```

Ausgabe: 2

Ausgabe: 3

Funktionsobjekte

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

- werden auch Funktor genannt
- sind ein Ersatz für Funktionszeiger; sind vielseitiger und können mehr.
- werden deshalb häufig von der STL verwendet
- sind Objekte ganz normaler Klassen, die jedoch den `operator()` überladen haben.
- Dadurch ist der callback eines Funktors syntaktisch nicht von dem einer Funktion zu unterscheiden.

```
class Next {
  int n;
public:
  Next(int n0) { n=n0; }
  int get() { return n++; }
  int operator()() { return n++; }
  void reset() { n=0; }     // Setzt das Gedächtnis zurück.
}
```

Die Memberviariable `n` übernimmt hier die Funktion des Gedächtnisses der Memberfunktion `Next::get()`

```
void generate( int array[], int first, int last, Next gen ) {
  for( ;first!=last;++first ) {
    array[first] = gen.get();
    array[first] = gen.operator()();
    array[first] = gen();
  }
}
```

Geänderte Abhängigkeit: Die Klasse `Next` muss den Funktionsoperator überladen!

Äquivalente callbacks!

Anwendung des Funktionsobjekts

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
int main()
{
  const int MAX = 1000;
  int a[MAX];
   Next genAscInt(5);
  generate( a, a+MAX, genAscInt );
  return 0;
}
```

- Bewertung:
  - Jeder neue Elementtyp erfordert eine Überladung von `generate`
  - Jede neue Wertefolge erfordert eine eigene Funktionsobjektklasse und eine Überladung mit dieser.

    Man benötigt für m verschiedene Elementtypen mit je n verschiedenen Wertefolgen m mal n Versionen (Überladungen) der Funktion `generate`.

- Abhilfe: Funktionstemplates