Technische Informatik 2

C/C++ Programmierung
2. Zeiger, Arrays, Zeichenketten

Prof. Dr. Ivo Wolf

Institut für Medizinische Informatik

hochschule mannheim

Überblick

2. Zeiger und Arrays
    1. Zeiger
    2. Dynamisch angelegte Variablen und Objekte
    3. Zeiger und Klassen/Strukturen
    4. Speicherklassen und Typ-Qualifizierer*
    5. Arrays
    6. Zeichenketten

2. Zeiger, Arrays, Zeichenketten

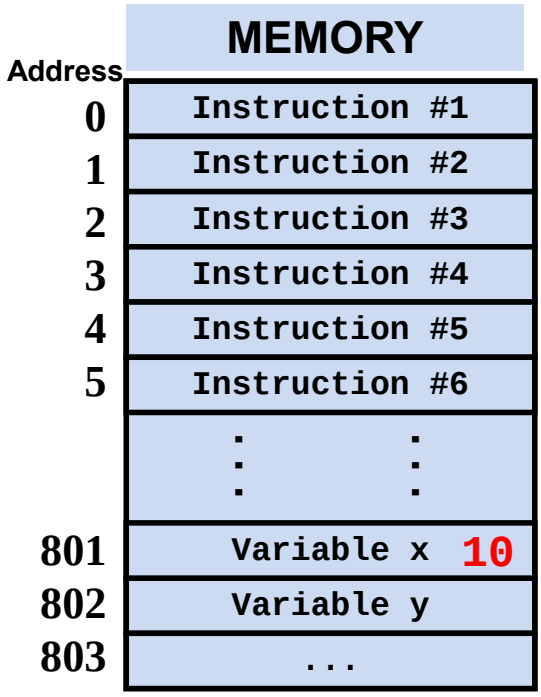
1. Zeiger

Variablen haben Adressen

- Jede Speicherstelle hat eine Adresse
- Variablen brauchen Speicher
  → haben Adressen

```
int x = 10;
```

→ Setzt den Inhalt der Speicherstelle 801 auf den Wert 10



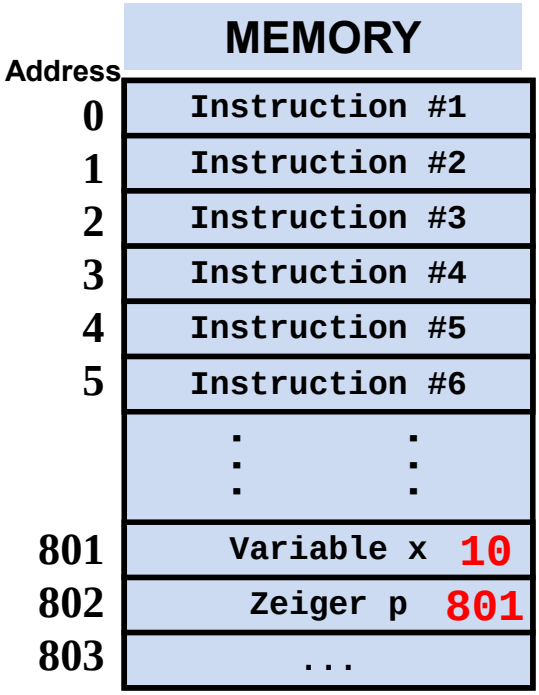
Zeiger (engl. pointer)

Zeiger (engl. pointer):

- sind (auch) Variablen
- enthalten Adressen

Beispiel:

- Zeiger-Variable p
- soll auf x zeigen



Zeigervariablen in C

Syntax: **Typ*** **Zeigervariablenname**

Beispiel: **int*** **ip;**

lies: ip ist ein Zeiger auf eine int-Variable

Darstellung:

ip Zeiger

***ip** int-Variable

Zugriff auf referenzierten Wert: mittels Dereferenzierungsoperator *

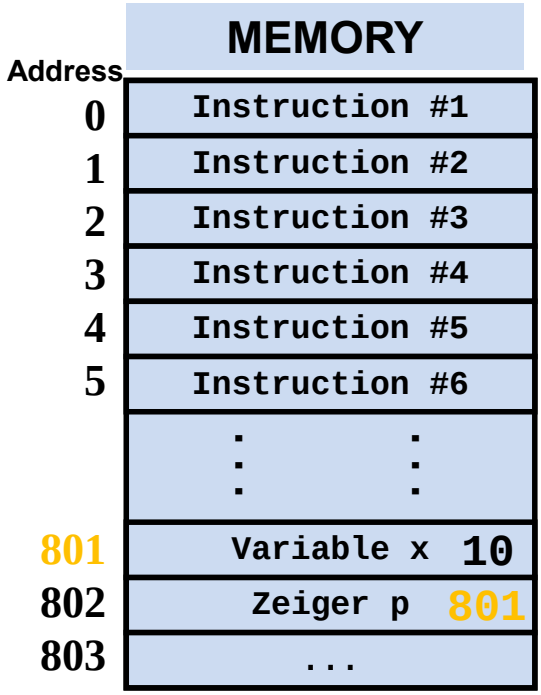
besonderer Zeiger: NULL-Zeiger

Address operator &

Beispiel:

- Zeiger-Variable p
- soll auf x zeigen

```
int  x = 10;
int* p;

p = &x;
// p speichert Adresse von x,
// also 801
```

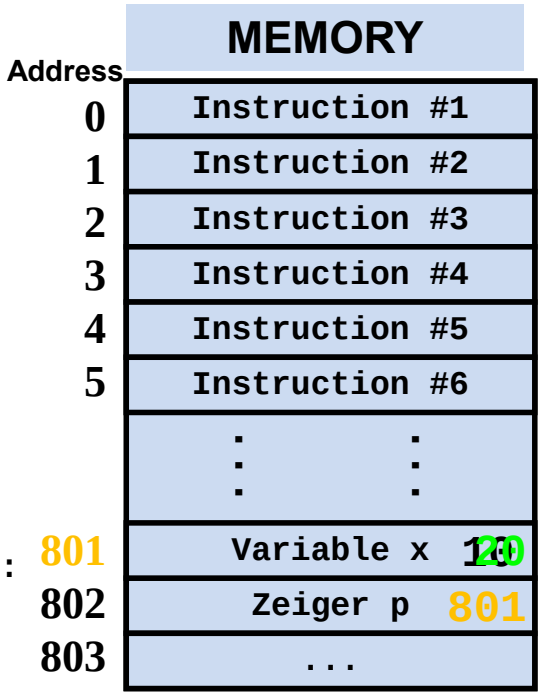


Dereferenzierungsoperator *

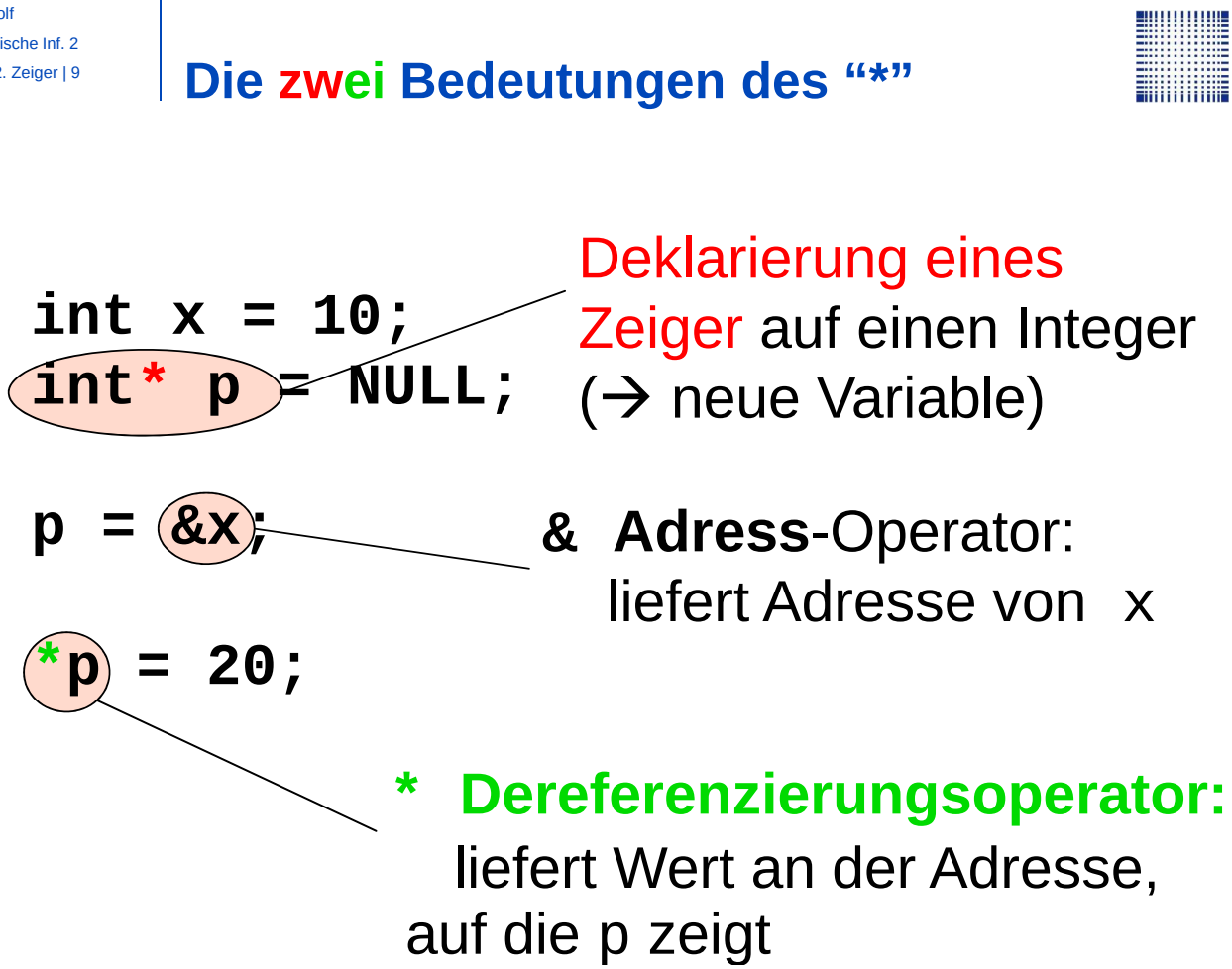
Beispiel:

- Zeiger-Variable p
- soll auf x zeigen

```
int  x = 10;
int* p;

p = &x;

// Das Folgende gibt 10 aus
printf("%i", *p);
// Auch ändernder Zugriff möglich:
*p = 20;
```



Die zwei Bedeutungen des “*”



int *p;

- You can use the pointer in a C++ expression:

```
int x = 123;
int *p = &x;

int y = *p + 17;
*p = *p +1;
printf("%d %d“, x, y);
```

- Expected result?

```
124 140
```

Assigning a value to a *dereferenced* pointer

A pointer must have a value before you can *(should) dereference* it (follow the pointer).

```
int *p;
*p=3;
```

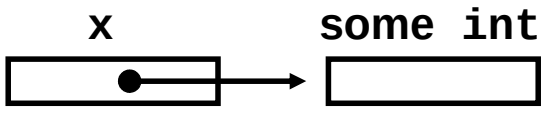
ERROR!!!
x doesn't point to anything!!!

```
int foo;
int *p;
p = &foo;
*p=3;
```

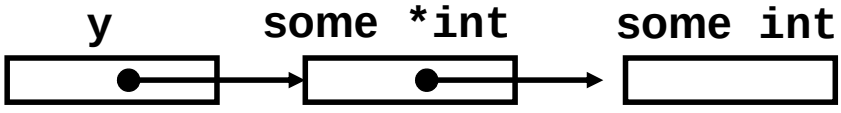
this is fine
x points to foo

Pointers to anything

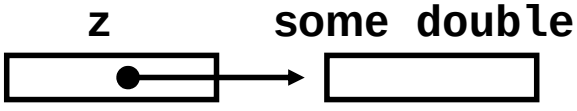
```
int *x;
```




```
int **y;
```




```
double *z;
```



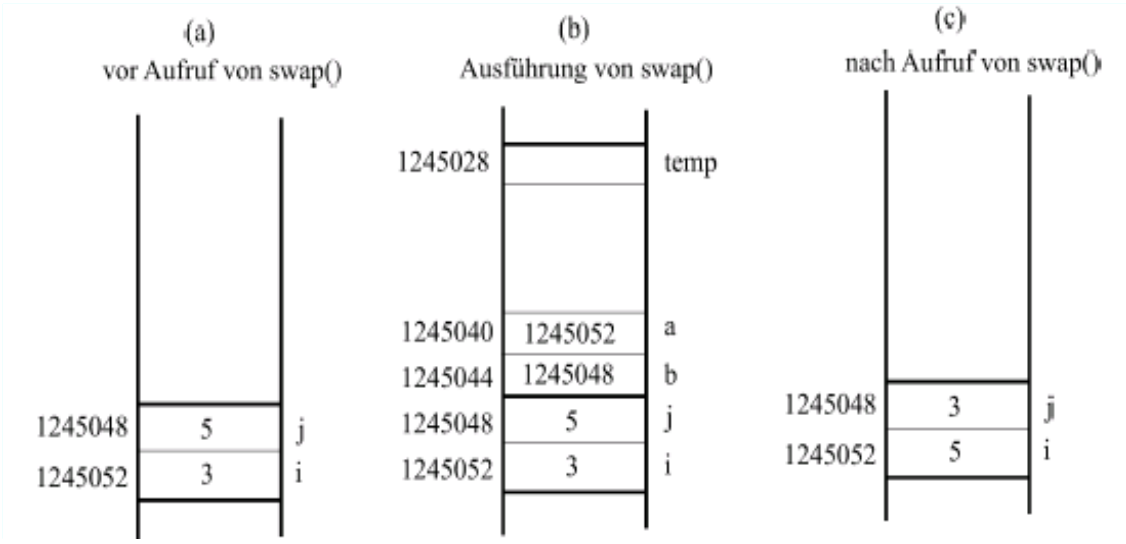
Adressparameter

- In C werden Parameter immer als Wert-Parameter (call-by-value) übergeben:
  - Der Wert des Arguments wird in den formalen Parameter kopiert.
  - Keine Änderung des Original-Werts möglich!
- Mit Hilfe von übergebenen Zeiger kann die aufgerufene Funktion auf die adressierte Speicherstelle (den eigentlich interessierenden Wert) zugreifen und diese verändern (call-by-reference)
- Java:
  - Für einfache Datentypen: call-by-value
  - Objekte = Objektreferenzen: call-by-reference
    (die Referenz/der Pointer wird by-value übergeben, für das Objekt selbst ist das ein call-by-reference)

Adressparameter - Beispiel

```
#include <stdio.h>

void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main () {
  int i = 3, j= 5;

  printf ("Vor swap: i = %d, j =%d\n", i, j);
  swap(&i, &j);
  printf ("Nach swap: i = %d, j =%d\n", i, j);

  return 0;
}
```

Adressparameter - Speicherabbild



Adressparameter – Verwendung

Adressparameter werden verwendet, wenn

- Eine Funktion mehrere Ergebnisse liefern soll (s. z.B. scanf()).
- Wenn komplexe Datentypen (z.B. Arrays, Objekte) übergeben werden, um die Effizienz zu steigern.

- Vorsicht: Es dürfen keine Verweise auf lokale (automatische) Variablen zurückgegeben werden:

```
int * Mist()
{
    int dasGibtProbleme = 7;
    return &dasGibtProbleme; //NEIN!!
}
```

2. Dynamisch angelegte Variablen und Objekte

Dynamisch reservierter Speicher

- In einem speziellen Speicherbereich, dem Heap, (engl. für Haufen) kann dynamisch Speicher reserviert werden
- Der **Heap**(bereich) wird im Gegensatz zum **Stack**(bereich), der zur Aufnahme der statischen Daten dient, beim Verlassen eines Unterprogramms nicht wieder (ab-) geräumt:
  - dynamische Objekte sind nach Verlassen eines Unterprogramms noch zugreifbar.
  - auch nicht, wenn kein Zeiger mehr darauf vorhanden ist: dann hat man ein Speicherleck (Java hat dafür seinen Garbage-Collector)
  - dynamisch reservierter Speicher muss also explizit wieder freigegeben werden!

2. Zeiger, Arrays, Zeichenketten

2. Dynamisch angelegte Variablen und Objekte

- C: malloc und free
- C++: new und delete
- Gefahren

Speicher reservieren auf dem Heap

Reservieren von Speicher:

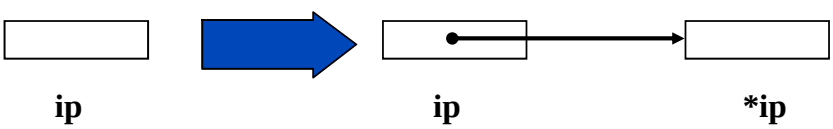
- C-Funktion: `void* malloc(size_t)`

  (in `stdlib.h`)
- Menge an Speicher wird in Bytes angegeben
- Beispiel:

```
int *ip;
ip  =  (int *) malloc(sizeof(int));
```

oder

```
ip  =  (int *) malloc(sizeof(*ip));
```



Speicher auf dem Heap freigeben

- In C kann (sollte) der Programmierer nicht mehr benötigten Platz auf dem Heap explizit freigeben.
- Dazu: Funktion `free` (in `stdlib.h`)
- Beispiel:

```
free(ip); // crash, falls ip==NULL
ip=NULL;  // ist eine gute Idee…
```

2. Zeiger, Arrays, Zeichenketten

2. Dynamisch angelegte Variablen und Objekte

- C: malloc und free
- C++: new und delete
- Gefahren

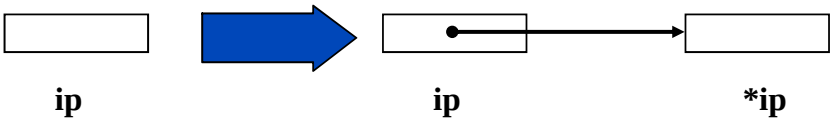
Speicher reservieren auf dem Heap

Reservieren von Speicher:

- C++-Operator: `new`

  (Funktion (Operator) mit etwas seltsamer Syntax)
- Menge an Speicher wird durch Typ angegeben
- Beispiel:

```
int *ip;
ip  =  new int;
(vgl: ip=  (int *) malloc(sizeof(int)))
```



Speicher auf dem Heap freigeben

- Auch in C++ sollte der Programmierer nicht mehr benötigten Platz auf dem Heap explizit freigeben.
- Dazu: Operator `delete`
- Beispiel:

```
delete ip; // kein crash bei ip==NULL
ip=NULL;   // ist eine gute Idee…
```

Allocating memory using new for objects

```
StudentRecord* sptr= new StudentRecord;
Point *pptr = new Point(5, 5);
```

- **new** calls the object's constructor.
- Do not use **malloc** for classes!

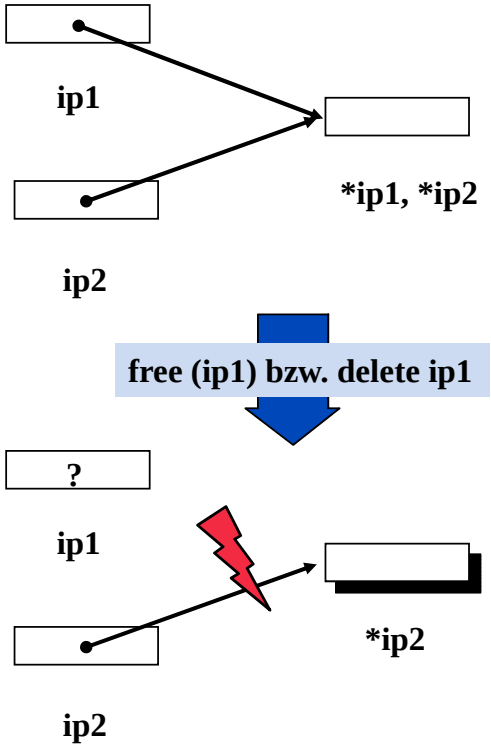
New vs Malloc

- ***Never*** mix new/delete with malloc/free

Malloc	New
Standard C Function	Operator (like ==, +=, etc.)
Used sparingly in C++; used frequently in C	Only in C++
Used for allocating chunks of memory of a given size without respect to what will be stored in that memory	Used to allocate instances of classes / structs / arrays and will invoke an object's constructor
Returns void* and requires explicit casting	Returns the proper type
Returns NULL when there is not enough memory	Throws an exception when there is not enough memory
Every malloc() should be matched with a free()	Every new/new[] should be matched with a delete/delete[]

2. Dynamisch angelegte Variablen und Objekte

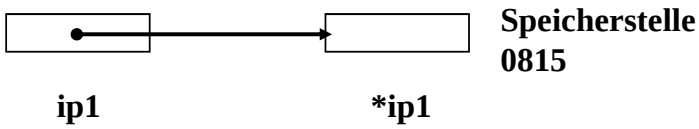
- C: malloc und free
- C++: new und delete
- Gefahren

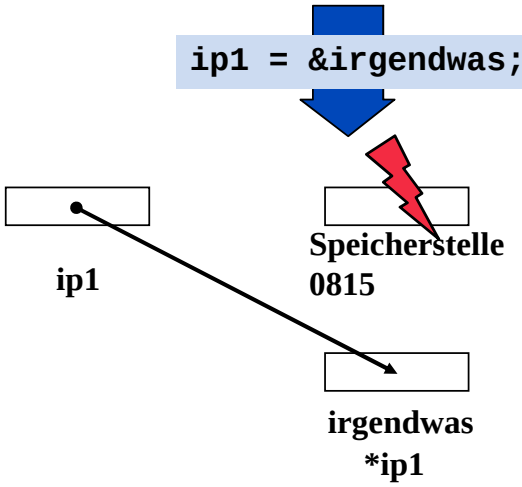


- ip2 zeigt auf freigegebenen Platz!
- Wird bei nächster Speicherreservierung wiederverwendet!
- Bei Zugriff über ip2: unerwartete Effekte!

Gefahr 2: Probleme beim Vergessen von free / delete

```
ip1 = new int; // Bsp.: Reserviert
               // Speicherstelle 0815
```





- auf die Speicherstelle 0815 zeigt kein Zeiger mehr!
- Keine Chance, die Speicherstelle jemals wieder freizugeben!
- Speicherleck=memory leak

Gefahr 3

- ***Never*** mix new/delete with malloc/free !

```
int *ip = new int;
...
free(ip); // crash! (sooner or later)
```

```
int *q = (int*)malloc(sizeof(int));
...
delete q; // crash! (sooner or later)
```

2. Zeiger, Arrays, Zeichenketten

3. Zeiger und Strukturen/Klassen

Zeiger auf Strukturen/Objekte

- Zeiger auf Strukturen/Objekte sind häufig:

```
struct Data {
  int x,y;
};
struct Data* dataptr =
  (struct Data*) malloc(sizeof(struct Data));
```

- Zugriff auf Member-Variable (oder Methode):

```
printf("x: %i",(*dataptr).x);
```

➔ umständlich! ☹

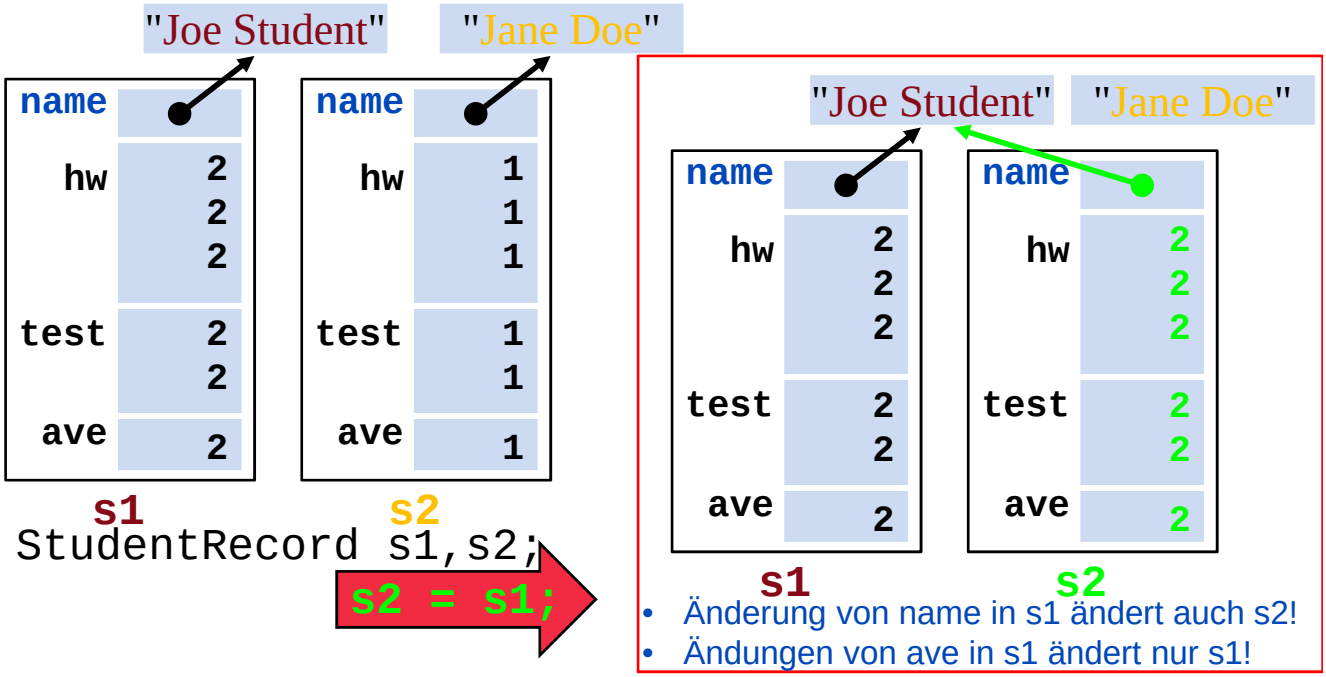
- Daher spezieller "*member access operator* ": `->`

```
printf("x: %i",dataptr->x);
```

Sieht aus wie ein "Zeiger"!

Vorsicht bei Zeigern als Member

- Ist eine Member-Variable ein Zeiger, bedeutet *kopieren* das Kopieren des Zeigers!



2. Zeiger, Arrays, Zeichenketten

4. Speicherklassen und Typ-Qualifizierer*

Typ-Qualifizierer / Modifier in C/C++

- Werden dem Typ-Spezifizierer eines Objekts vorangestellt.
- Beschreiben zusätzliche Eigenschaften, die es normalerweise nicht hat.

Speicherklassen und Typ-Qualifizierer*

extern	Ankündigung einer globalen Variable, die in einer anderen Kompiliereinheit angelegt wurde.
extern "C"	deklariert Variablen und Funktionen mit C-Symbolbezeichnungen. So deklarierte C-Funktionen/ Variablen können damit auch in C++ Programmteilen genutzt werden. Siehe z.B. www.cpp-tutor.de/cpp/le07/extern_c.html
static	Variablen: haben eine statische Lebensdauer (= der des Programms). Können global oder lokal sein. Globale Variablen und Funktionen: nur innerhalb der aktuellen Kompiliereinheit (.c/.cpp Datei) zugreifbar.
volatile	Garantiert keine Änderung der Zugriffsreihenfolge auf volatile qualifizierte Variablen bei Optimierung (z.B. wird der Wert einer Variablen vor jedem Zugriff neu ermittelt).
const	bewirkt, dass ein Objekt nach seiner Deklaration nicht mehr verändert werden darf, schützt das Objekt also vor (unbeabsichtigter) Veränderung. Auf const-Objekte dürfen nur const-Methoden angewandt werden
mutable	Dieser Modifier erlaubt es, dass das so qualifizierte Attribut eines const-Objekts verändert werden darf.

2. Zeiger, Arrays, Zeichenketten

5. Arrays

Deklaration von Arrays mit fester Größe

Syntax:

```
Datentyp Bezeichner[konstanter Wert];
```

Beispiel: `int vec[10];` (**NICHT**: `int vec[n]`)

- Die Größe des Arrays muss (bei nicht wenigen Compilern noch#) eine Konstante sein, also zum Übersetzungszeitpunkt bekannt sein!!
- Indexbereich: 0 – N-1, wenn N der Wert der konstanten Größenangabe ist.
- Die Länge eines Arrays kann nicht abgefragt werden ☹ (anders als in Java)
- Zugriff auf Elemente wie man es von Java her erwartet:

```
vec[7]=3;
```

#Bei lokalen Variablen sind seit C99 auch variable Größen erlaubt. Innerhalb von Strukturen gibt es dafür (zwangsläufig) starke Einschränkungen, s. z.B. https://en.wikipedia.org/wiki/Flexible_array_member.

Arrays in C/C++

- Arrays können bei der Vereinbarung initialisiert werden, ggf. mit impliziter Längenangabe.
  Beispiel: `int vec[] = { 2, 3, 7 };`
- Der Zugriff auf Arrays als Ganzes ist problematisch:
  - insbesondere können die Inhalte von Arrays nicht als Ganzes zugewiesen werden.
- char-Arrays sind in C besonders wichtig. Da es in C keinen Datentyp string gibt, werden Strings über char-Arrays realisiert.

Andere Zeigerinterpretation

Bisher: Verwendung von Zeigern wie in anderen Programmiersprachen auch.

In C/C++: Zeiger und Arrays sind eng miteinander verwandt.

Arrays = zusammenhängende Speicherbereiche mit Objekten desselben Typs.

Zeiger = Verweis auf ein Objekt.
Dieses Objekt kann auch das erste Element einer zusammenhängenden Folge von Objekten desselben Typs sein.

Dynamische Arrays in C

- Beispiel: Definition eines Vektors mit n Elementen. Die Anzahl der Elemente n wird zur Laufzeit eingelesen.

```
#include <stdlib.h>
int n;      /* number of elements */
int * vec;  /* dynamic vector */

vec = (int *) malloc (n * sizeof(int));
```

- Zugriff auf das Element mit Index i von vec:
  `vec[i] oder *(vec + i)`

Zeiger und Arrays

- Array-Name ist im Wesentlichen ein *const* Zeiger!
- Eckige Klammern auch für Zeiger nutzbar:

```
int vec[10];
int* p;
p = vec;
p[7]=5;

p = &vec[2];
for (int  i=0;i<3;i++)
  p[i]++;
```

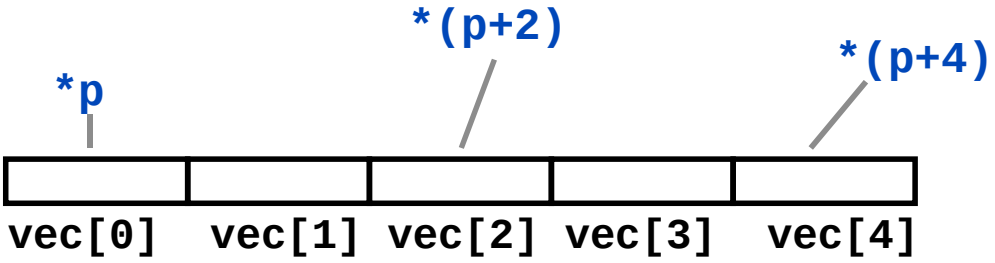
p is “the address of **vec[2]**”

p[i] is the same as **vec[i+2]**

Zeiger Arithmetik

- Mit Zeigern kann gerechnet werden
- Dabei Änderung des Zeigers immer um Vielfache der Größe des Datentyps, auf den der Zeiger zeigt.

```
double vec[5];
double *p = vec;
```



Zeigerarithmetik 1/2

- Addition Zeiger mit `int` :
  - Positionierung um die angegebene Anzahl von Elementen.
  - Sei p ein Zeiger auf `typ` und `i` eine int-Variable. Dann wird mit p + i die um (i * sizeof(typ)) Bytes verschobene Adresse bezeichnet.
- Subtraktion eines `int` von Zeiger:
  - Positionierung um die angegebene Anzahl von Elementen davor (nicht kommutativ).

Zeigerarithmetik 2/2

- Subtraktion Zeiger von Zeiger:
  - Anzahl der Elemente zwischen den beiden Zeigern.
- Inkrement / Dekrement von Zeigern:
  - Fort- bzw. Zurückschalten auf das nächste bzw. das vorhergehende Element

Zusammenfassung: Arrays und Zeiger

Name einer Array-Variablen = konstanter Zeiger auf das erste Element des Arrays.

Zugriff auf Arraykomponenten ist möglich über 2 Operatoren:

- Indexoperator [ ]
- Dereferenzierungsoperator *

Beispiel:

```
int vec [10];
```

Es gelten folgende Äquivalenzen:

```
vec[i] ≡ *(vec + i)
&vec[i] ≡ vec + i
vec  ≡&vec[0]
*vec ≡*&vec[0] ≡vec[0]
```

Unterschied zwischen Zeiger und Array auf Stack

- Ein Zeiger ist eine Variable.
- Ein Array auf dem Stack ist dagegen eine Konstante.

```
int vec [10];
int * p;
p = vec;   /* zulässig */
vec = p;   /* nicht zulässig */
```


2. Zeiger, Arrays, Zeichenketten

6. Zeichenketten

Strings in C = char Arrays

- Stringvereinbarung:

```
char string [len+1];
```

- Am Ende eines Strings muss der `char`-Wert 0 stehen = String-Terminator (‘\0‘)
- Für die Nutzdaten werden `len` Felder benötigt, für den String-Terminator (‘\0‘) ein Feld.
- Routinen zur Stringverarbeitung müssen immer auf ‘\0‘ auf Ende des Strings abfragen.

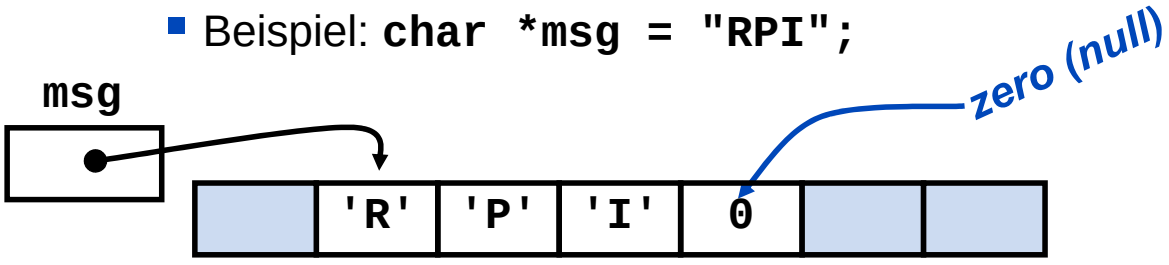
  Grund dafür?
  - Array-Länge kann nicht abgefragt werden!
  - Das Array darf auch länger sein als die Nutzdaten!
- Funktionen zur Stringverarbeitung sind in `string.h` definiert, Funktionsnamen beginnen mit `str` (z.B. `strlen`, `strcpy`)

Strings in C = char Arrays

- Stringvereinbarung:

```
char string [strlen+1];
```

- Am Ende eines Strings muss der `char`-Wert 0 stehen = String-Terminator (‘\0‘)
- Für die Nutzdaten werden `strlen` Felder benötigt, für den String-Terminator (‘\0‘) ein Feld.
- Beispiel: **`char *msg = "RPI";`**



Strings in C = char Arrays

- Für Strings ist die implizite Längenangabe sehr bequem:

```
char *string = "Dies ist ein String";
```

oder

```
char string [] = "Dies ist ein String";
```

entspricht:

```
char string [] = {'D', 'i',... 'n', 'g','\0'};
```


Stringverarbeitung - Beispiel

```
#include <stdio.h>
main ()
{
   char  string1 [] = "2 fast 4 u ?";
   int i;
   for(i=0; string1[i] != '\0'; i++)
     if (string1[i] >= 'a' &&
         string1[i] <= 'z')
       string1 [i] -= 32;
   printf ("string1 = %s\n", string1);
}
```

Bibliotheksfunktionen zur Verarbeitung von Zeichenketten 1/3

Header: `<string.h>`

strcpy `char * strcpy (char * dest, char * scr);`
Kopiert die Zeichenkette src an den durch dest adressierten Speicherbereich.
Rückgabewert: Zeiger auf das erste Zeichen von dest.

strncpy `char * strncpy (char * dest, char * scr, int n);`
Kopiert die ersten n Zeichen der Zeichenkette src an den durch dest adressierten Speicherbereich.
Rückgabewert: Zeiger auf das erste Zeichen von dest.

Bibliotheksfunktionen* zur Verarbeitung von Zeichenketten 2/3

strcat `char * strcat (char * dest, char * scr);`
Hängt die Zeichenkette src an die Zeichenkette dest an.
Rückgabewert: Zeiger auf das erste Zeichen von dest.

strncat `char * strncat (char * dest, char * scr , int n);`
Hängt die ersten n Zeichen der Zeichenkette src an die Zeichenkette dest an.
Rückgabewert: Zeiger auf das erste Zeichen von dest.

Bibliotheksfunktionen* zur Verarbeitung von Zeichenketten 3/3

strlen `int strlen (char * s);`

Bestimmt die Anzahl der Zeichen von s (ohne \0).

Rückgabewert: Anzahl der Zeichen von s.

strcmp `int strcmp (char * s1, char * s2);`

Vergleicht die Inhalte von s1 und s2 alphabetisch.

Rückgabewert: < 0, falls s1 < s2

= 0, falls s1 == s2

> 0, falls s1 > s2