Pointer & Speicherverwaltung

Vom Umgang mit Speicheradressen

Übersicht

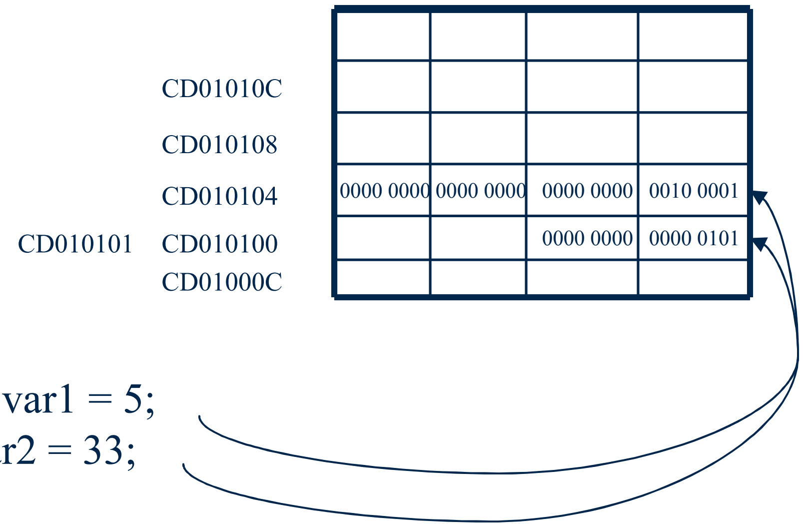
- Rückblick – Datentypen
  - Basis Typen
  - Strukturen
- Zeiger Grundlagen und Anwendung
- Dynamische Speicherzuweisung
  - new() und delete()
- Speichertypen
  - heap, stack
- Typische Fehlerquellen

Rückblick einfache Datentypen

- Einfache Variablen
  - Ganzzahlig (short, int, long)
  - Reelle Zahlen (flaot, double)
  - Zeichen (char)

- Darstellung im Speicher
  - binär (000100101….)
  - Je nach Datentyp mit der Länge
    - 1 byte (char), 2byte (short)
    - 4 byte (int , long , float)
    - 8 byte (double)

Ausschnitt aus dem Hauptspeicher
je ein byte in 4 byte Blöcken



decimal 33 = 0010 0001 binär
2**5 + 2**0
32 + 1

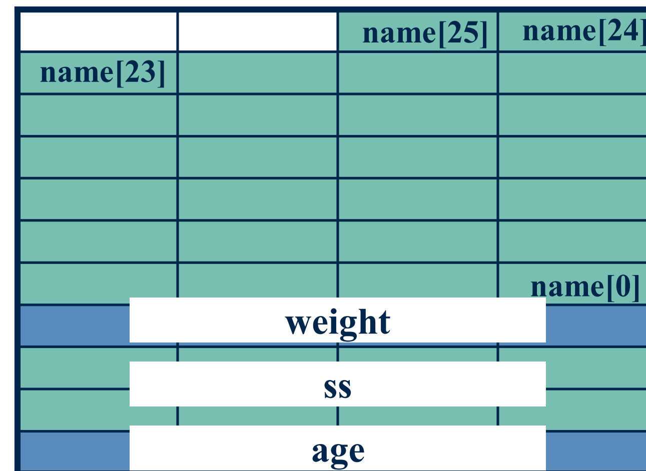
Rückblick – komplexe Datentypen

- Arrays
  - int Feld[10] = { 1,10,…..
  - char Text[10]
    (Sonderfall C-Strings…)

- Komplexe Datentypen/-strukturen
  - struct MeineStrukt {…} Variable
  - Zugriff auf Member durch "." Operator

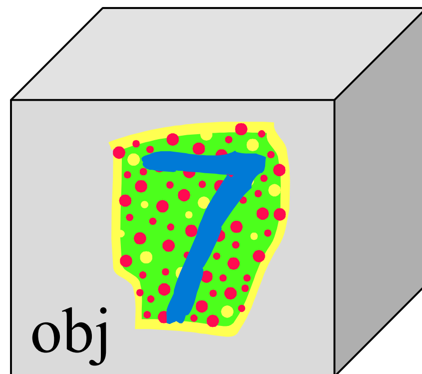
- Strukturen belegen (u.U.) einen sehr großen Speicherbereich

```
struct PERSON       // Structur vom typ PERSON
{
      int   age;           // Member age vom typ int
      long  ss;            //  weitere Member
      float weight;
      char  name[25];
} vater;                   // Variable vater vom Typ
                           // struktur PERSON
struct PERSON bruder;  // weitere Variable C-Stil
PERSON schwester;      // weitere Variable C++
bruder.age = 12;       // Zugriff durch "."
```



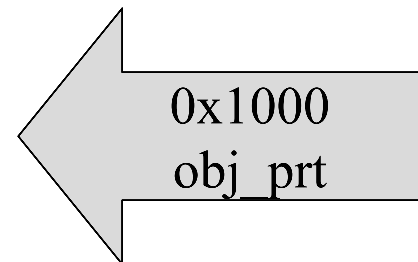
Begriff und Anwendungen

- Zeiger (pointer) sind Speicheradressen
  - Wertebereich und Darstellung maschinenabhängig
    - 20 Bit (8086), 24 Bit (68000), 32 Bit (80386, Pentium, PowerPC)
    - linear (Motorola), Segment + Offset (Intel bis 80286)
- Zeigervariable enthalten Speicheradressen



0x1000

Objekt an Speicherstelle



Zeiger auf Objekt

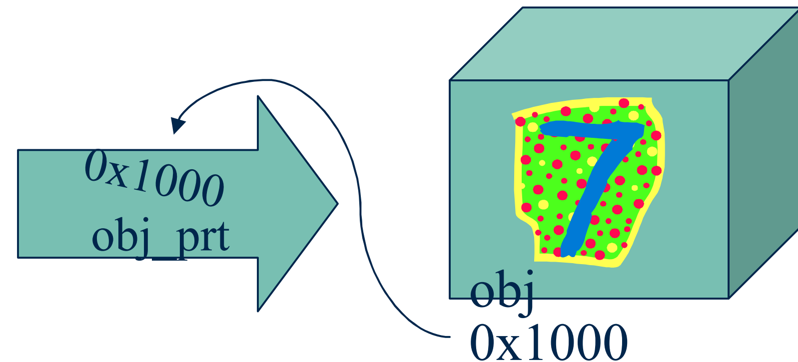
Begriff und Anwendungen

- Hauptanwendungen:
  - Umgang mit Zeichenketten (strings)
  - Zugriff auf dynamisch allokierte Speicherbereiche
  - Verkettung von Objekten (z.B. Liste aller existierenden Sprites)
  - alternative Form mit Arrays zu arbeiten: Adreßarithmetik
  - in C: Rückgabeparameter von Funktionen (*call by reference*) ☹

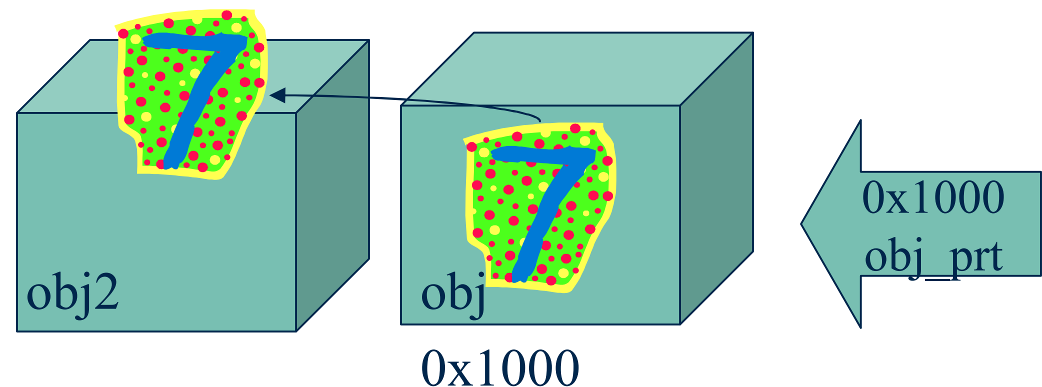
- Zur Unterscheidung zwischen Variable eines Datentyps und einem Zeiger auf einen Datentyp wird der "*" als Vorsatz genommen
  - int Variable // Variable vom Typ int
  - int *Pointer // Zeiger auf einen Speicherplatz für int

Zeiger Operationen (zur Veranschaulichung)

- Zuweisung der Adresse eines Objektes an einen Zeiger
  - int Obj = 7;
  - int *obj_prt;
  - obj_ptr = &obj;

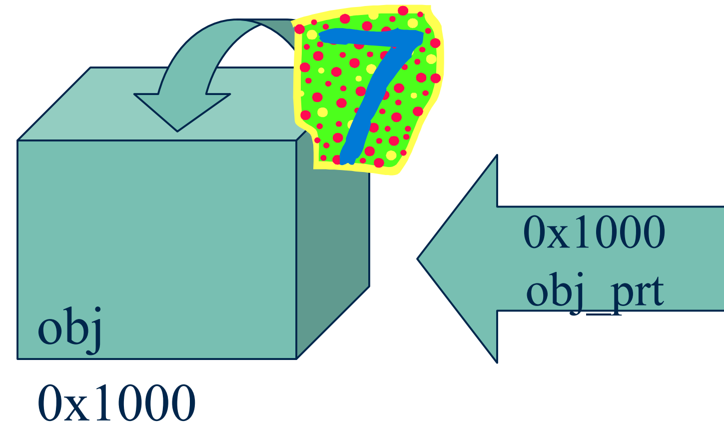


- De-Referenzzierung
  Den Inhalt an der Adresse des Zeigers einem anderen Objekt zuweisen
  - int obj2;
  - obj2 = *obj_prt;



Zeiger Operationen (zur Veranschaulichung)

- Wert an der (Speicher)-Stelle Ändern auf die der Zeiger zeigt.
  - int Obj;
  - int *obj_prt;
  - obj_ptr = &obj;
  - *obj_ptr = 7;



Deklaration und Operatoren

immer gleich initialisieren, damit die Abstürze zumindest reproduzierbar sind

- Deklaration durch *
  - reserviert Speicherplatz für <u>eine</u> Speicher<u>adresse</u>
  - ordnet dem Zeiger einen Typ zu, auf dessen Objekte er zeigen darf

```
int* Zeiger = NULL;
int Wert;
```

- Adresse einer Variablen (`int Wert;`)

```
Zeiger = &Wert;
```

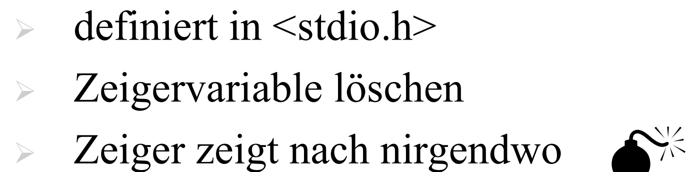
- Dereferenzierung einer Zeigervariablen

```
Wert = *Zeiger;
```

- Vergleich `==` `!=`

```
(Zeiger1==Zeiger2)
(Zeiger1!=Zeiger2)
```

- Konstante
  - definiert in <stdio.h>
  - Zeigervariable löschen
  - Zeiger zeigt nach nirgendwo



```
#define NULL 0
Zeiger = NULL;
(Zeiger==NULL)
```

- Initialisierung

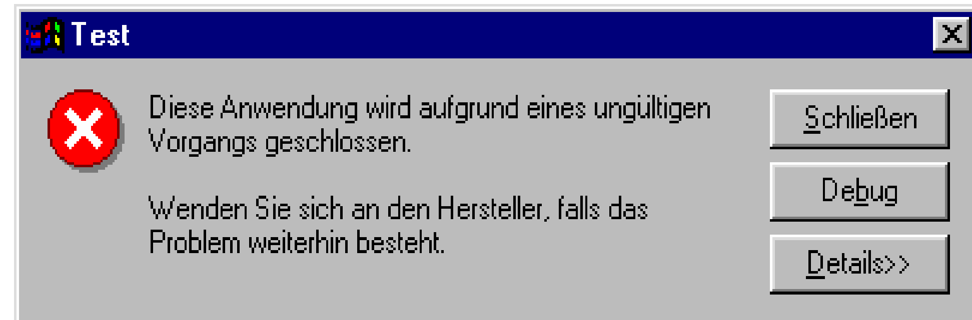
```
int* Zeiger = &Wert;
```

Die berühmte Schutzverletzung unter Windows

- so einfach ist das: Zugriff über einen illegalen Zeiger
- **das ist kein Fehler des Betriebssystems, sondern des Anwendungsprogrammierers**
  - das Betriebssystem erfüllt hier gerade seine Aufgabe
- leider zu oft macht auch Windows Fehler:
  - es entdeckt nicht alle illegalen Zugriffe
  - mitunter läßt es sich von einer Anwendung abstürzen

... aber meistens sind es die Anwendungsprogrammierer

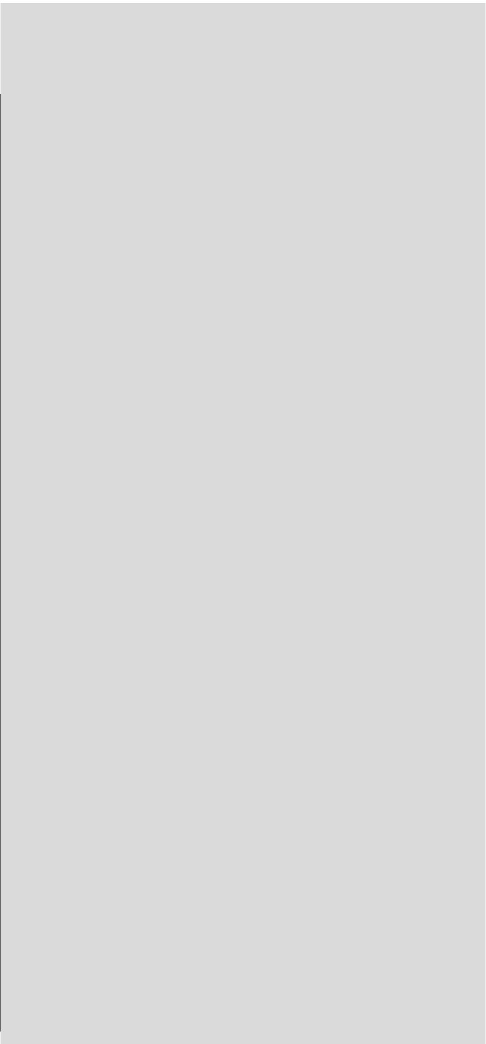
```
#include <stdio.h>
void main ()
{
    int *Zeiger = NULL;
    int x = *Zeiger;
}
```



Hardware-Hintergrund Beispiel

```
int x = 5, y = 0x17A2;
int *zeiger;
int *z1 = NULL;
int vek [2];
                    // Zustand ①
    // Adresse von x zuweisen:
zeiger = &x;
                    // Zustand ②
    // indirekter Zugriff auf x:
y = *zeiger;
                    // Zustand ③
 // indirekte Zuweisung auf x:
*zeiger = 0x1234;
                    // Zustand ④
   // Adresse von Vektorelem.:
z1 = &vek[1];
                    // Zustand ⑤
```

Name	*Adresse*	①
	3AC81	???
vek[1]	3AC80	???
	3AC7F	???
vek[0]	3AC7E	???
	3AC7D	00
	3AC7C	00
z1	3AC7B	00
	3AC7A	???
	3AC79	???
zeiger	3AC78	???
	3AC77	17
y	3AC76	A2
	3AC75	00
x	3AC74	05



Hardware-Hintergrund Beispiel

```
int x = 5, y = 0x17A2;
int *zeiger;
int *z1 = NULL;
int vek [2];
                    // Zustand ①
   // Adresse von x zuweisen:
zeiger = &x;
                    // Zustand ②
    // indirekter Zugriff auf x:
y = *zeiger;
                    // Zustand ③
 // indirekte Zuweisung auf x:
*zeiger = 0x1234;
                    // Zustand ④
  // Adresse von Vektorelem.:
z1 = &vek[1];
                    // Zustand ⑤
```

Name	*Adresse*	①	②
	3AC81	???	???
vek[1]	3AC80	???	???
	3AC7F	???	???
vek[0]	3AC7E	???	???
	3AC7D	00	00
	3AC7C	00	00
z1	3AC7B	00	00
	3AC7A	???	03
	3AC79	???	AC
zeiger	3AC78	???	74
	3AC77	17	17
y	3AC76	A2	A2
	3AC75	00	00
x	3AC74	05	05

Hardware-Hintergrund Beispiel

```
int x = 5, y = 0x17A2;
int *zeiger;
int *z1 = NULL;
int vek [2];
                    // Zustand ①
   // Adresse von x zuweisen:
zeiger = &x;
                    // Zustand ②
    // indirekter Zugriff auf x:
y = *zeiger;
                    // Zustand ③
 // indirekte Zuweisung auf x:
*zeiger = 0x1234;
                    // Zustand ④
  // Adresse von Vektorelem.:
z1 = &vek[1];
                    // Zustand ⑤
```

Name	*Adresse*	①	②	③
	3AC81	???	???	???
vek[1]	3AC80	???	???	???
	3AC7F	???	???	???
vek[0]	3AC7E	???	???	???
	3AC7D	00	00	00
	3AC7C	00	00	00
z1	3AC7B	00	00	00
	3AC7A	???	03	03
	3AC79	???	AC	AC
zeiger	3AC78	???	74	74
	3AC77	17	17	00
y	3AC76	A2	A2	05
	3AC75	00	00	00
x	3AC74	05	05	05

Hardware-Hintergrund Beispiel

```
int x = 5, y = 0x17A2;
int *zeiger;
int *z1 = NULL;
int vek [2];
                   // Zustand ①
    // Adresse von x zuweisen:
zeiger = &x;
                   // Zustand ②
    // indirekter Zugriff auf x:
y = *zeiger;
                   // Zustand ③
 // indirekte Zuweisung auf x:
*zeiger = 0x1234;
                   // Zustand ④
   // Adresse von Vektorelem.:
z1 = &vek[1];
                   // Zustand ⑤
```

Name	*Adresse*	①	②	③	④
	3AC81	???	???	???	???
vek[1]	3AC80	???	???	???	???
	3AC7F	???	???	???	???
vek[0]	3AC7E	???	???	???	???
	3AC7D	00	00	00	00
	3AC7C	00	00	00	00
z1	3AC7B	00	00	00	00
	3AC7A	???	03	03	03
	3AC79	???	AC	AC	AC
zeiger	3AC78	???	74	74	74
	3AC77	17	17	00	00
y	3AC76	A2	A2	05	05
	3AC75	00	00	00	12
x	3AC74	05	05	05	34

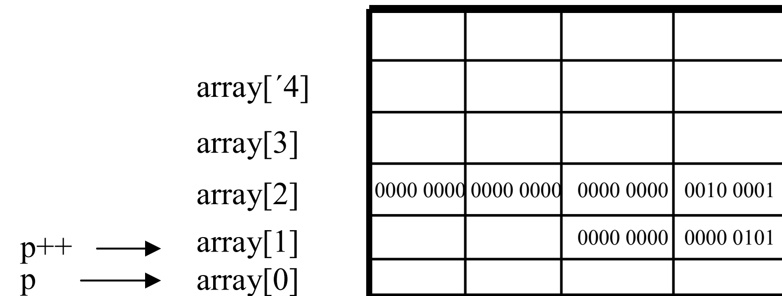
Hardware-Hintergrund Beispiel

```
int x = 5, y = 0x17A2;
int *zeiger;
int *z1 = NULL;
int vek [2];
                  // Zustand ①
   // Adresse von x zuweisen:
zeiger = &x;
                  // Zustand ②
   // indirekter Zugriff auf x:
y = *zeiger;
                  // Zustand ③
 // indirekte Zuweisung auf x:
*zeiger = 0x1234;
                  // Zustand ④
 // Adresse von Vektorelem.:
z1 = &vek[1];
                  // Zustand ⑤
```

Name	*Adresse*	①	②	③	④	⑤
	3AC81	???	???	???	???	???
vek[1]	3AC80	???	???	???	???	???
	3AC7F	???	???	???	???	???
vek[0]	3AC7E	???	???	???	???	???
	3AC7D	00	00	00	00	03
	3AC7C	00	00	00	00	AC
z1	3AC7B	00	00	00	00	80
	3AC7A	???	03	03	03	03
	3AC79	???	AC	AC	AC	AC
zeiger	3AC78	???	74	74	74	74
	3AC77	17	17	00	00	00
y	3AC76	A2	A2	05	05	05
	3AC75	00	00	00	12	12
x	3AC74	05	05	05	34	34

Adreßarithmetik

- Erbschaft aus C (maschinennah)
  - nur definiert und portabel, wenn Zeiger auf Array zeigt
  - meist effizienterer Code gegenüber Arrayadressierung bei linearer Abarbeitung
  - größte Vorsicht geboten ! Compilerwarnungen ernst nehmen !
- Addition und Subtraktion von int (auch ++ -- += -=)
  - +1 inkrementiert um Größe des Objekttyps (nicht um 1 Byte !)




```
int array[10];
int *p = &array[0];

for (i=0;i<9;i++) array[i] = 0;

for (i=0;i<9;i++,p++) *p=0;
```

Adreßarithmetik

- Adreßbereich des Arrays beachten
  - vom ersten bis (letzten+1) Element
  - wird nicht vom Compiler geprüft
- Vergleich zweier Zeiger   < <= >= >
- Subtraktion zweier Zeiger (keine Addition !) liefert `int`

Achtung - bei Pointer aufpassen

- Geht in der Regel auch ohne!

```
char* findchar(char* text,
               char c)
{
   char *p = text;

   while (*p) {
     if (*p == c)
       return p;
     p++
   }
}


char text[] = "Hallo";


cout << findchar(text,'a');
```

Typumwandlung

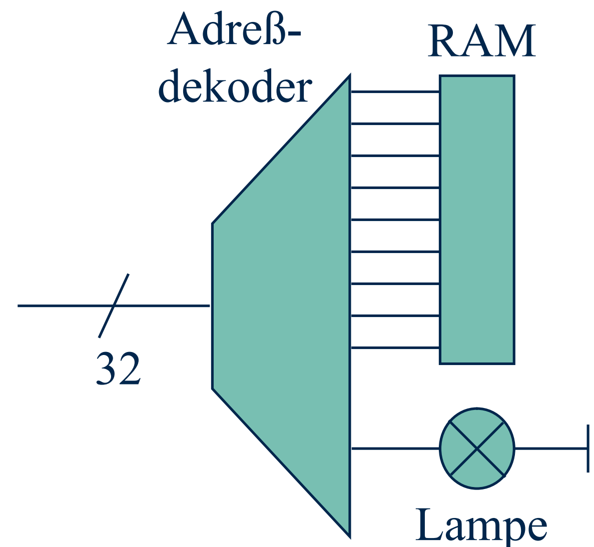
- automatisch
  - 0 ⇨ Zeigertyp ok (NULL-Zeiger)
  - Zeiger auf abgeleitete Klasse ⇨ Zeiger auf Basisklasse *später*
- explizit durch cast-Operator
  - Zeiger auf abgeleitete Klasse ⇦ Zeiger auf Basisklasse *später*
  - Zeigertyp ⇔ Integer
    - implementierungsabhängig (Alignment)
    - Anwendung: Zugriff auf absolute Adressen
  - Zeigertyp1 ⇔ Zeigertyp2
    - implementierungsabhängig (Alignment)
  - Zeigertyp ⇨ void* ok (generischer Zeiger)
  - Zeigertyp ⇦ void* implementierungsabhäng

nur in Embedded Control Anwendungen für memory mapped I/O

braucht man nicht in ordentlichen Programmen

Memory-Mapped I/O

- *eine* Möglichkeit zum Anschluß von Peripheriegeräten an Mikrocontroller
  - Lampen, Motorsteuerung, ...
- die Adressen der Peripheriegeräte liegen fest durch Verdrahtung der Chips
- Problem:
  absolute Adresse von Variablen in C++ eigentlich unbekannt
- Lösung:
  Zeiger mit int-Konstante initialisieren (⇨Typkonversion)




```
bool* pLampe =
0x3000ACF0;
*pLampe = true;      // Lampe
ein
*pLampe = false;     // Lampe
aus
```

veraltet: Ersatz für *call by reference*

falls Ihnen mal ein C Programm begegnet

Funktion zum Austauschen zweier Werte

```
void swap (int* pWert1, int* pWert2)
{
  int temporaer;

  temporaer = *pWert1;
  *pWert1 = *pWert2;
  *pWert2 = temporaer;
}
```

zur Namenswahl:
`*pWert1` suggeriert den Umgang mit Wert;
`*pointerW1` würde den Zeiger betonen

```
int Hugo, Fred;
swap (&Hugo, &Fred); // vertauscht die Inhalte von Hugo und Fred
```

Gegenüberstellung: Referenzen und Zeiger

- syntaktisch

	Referenz	Zeiger
➢ Deklaration	int& x;	int* x;
➢ Initialisierung (mit *int a*)	int& x = a;	int* x = &a;
➢ aktueller Parameter	a	&a
➢ Dereferenzierung	x	*x
➢ Konstante		NULL

notwendig als Endemarke in verketteten Strukturen

- semantisch
  - ➢ Referenz kann nie undefiniert sein

➢ Operator == vergleicht	Inhalte	Adressen

Zugriff auf Member von Strukturen (Objekte)

- abkürzende Schreibweise für Zugriff auf Attribute oder Funktionen eines Objekts
- "reguläre" Notation:
  (* Zeiger **) .**
- übliche Abkürzung:
  Zeiger **->**

```
struct Point {
int x;
int y; } P1;

Struct Point* P_P1 = & P1;

P1.x = 1;    // via Variable

(*P_P1).y = 2; // via Pointer

P_P1->y = 3;    // Kurzform
```

Kommandozeilenparameter char * array[]

- Das Betriebssystem übergibt beim Aufruf eine variable Anzahl von Parametern aus der Kommandozeile an `main`
  - `argc` liefert die Anzahl
  - `argv` ist ein Array von Zeigern auf Zeichenketten (liegen irgendwo)
  - `argv[0]` zeigt auf den Programmnamen aus der Kommandozeile

```
#include <iostream.h>

int main (int argc, char* argv[])
{

  cout << "Programmname: " << argv[0];
  for (int i=1; i<argc; i++) {
    cout << i << ". Parameter: " << argv[i] << endl;
  }
  return 0;                    // Fehlercode ans Betriebssystem: 0=ok
}
```

Dynamische Speicherverwaltung

- Bisher waren alle Variablen und Felder zur Laufzeit bekannt. Die Frage ist: Kann Speicher auch zur Laufzeit dynamisch angefordert werden, ohne dass die Größe zum Zeitpunkt der Programmerstellung bekannt ist?
- C++ - Stil
  - new - zum Anfordern von Speicher
  - delete – zum freigeben des Speichers
- C Stil (nicht mehr verwenden)
  - malloc / calloc - zum Anfordern von Speicher
  - free – zum freigeben

new() und delete()

- new liefert als Ergebnis der Speicheranforderung einen Zeiger auf den entsprechenden Datentyp
  - einfache Datentypen
  - Felder
  - Klassen
- Der Speicher muss (sollte) hinterher wieder mit delete freigegeben werden.
  - C++ hat keine Garbage collection
  - Erst nach Probrammende wird (verlorener) Speicher freigegeben

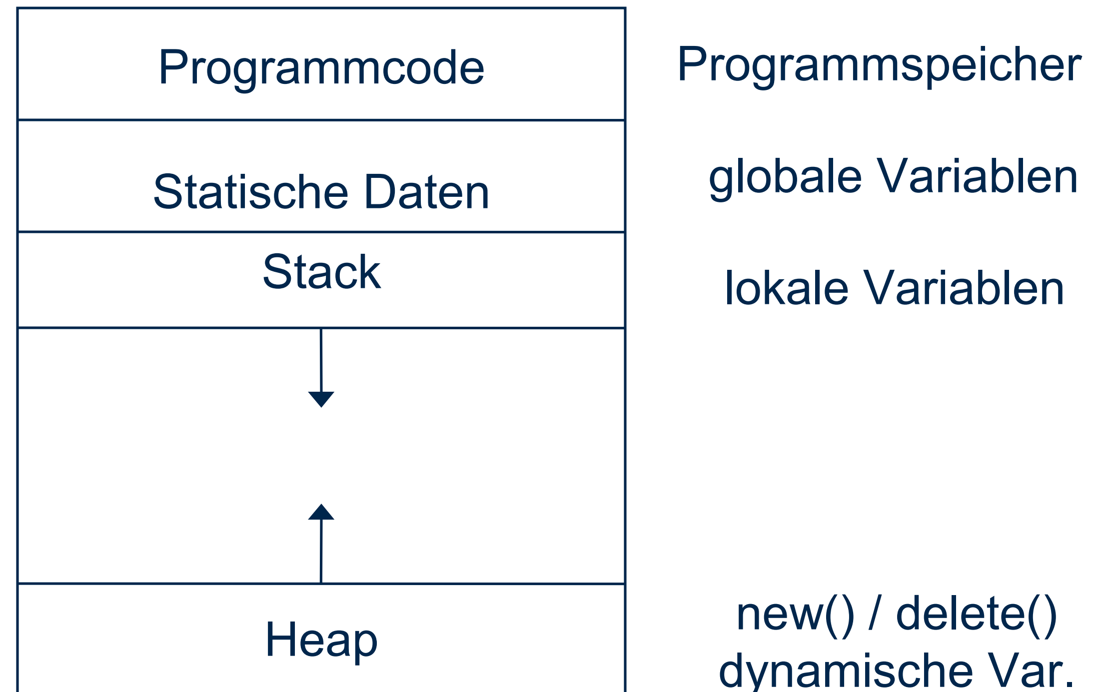
```
int *P_int = new int;
struct Demo {
   int i;
   long l;
   double d;
} var;
Demo *p_var = new Demo;
delete p_var;

p_var->l = 10;
int *parray = new int [100];

funktion () {
   int *p = new int [1000];
   …
   delete [] p;
   return;
}
```

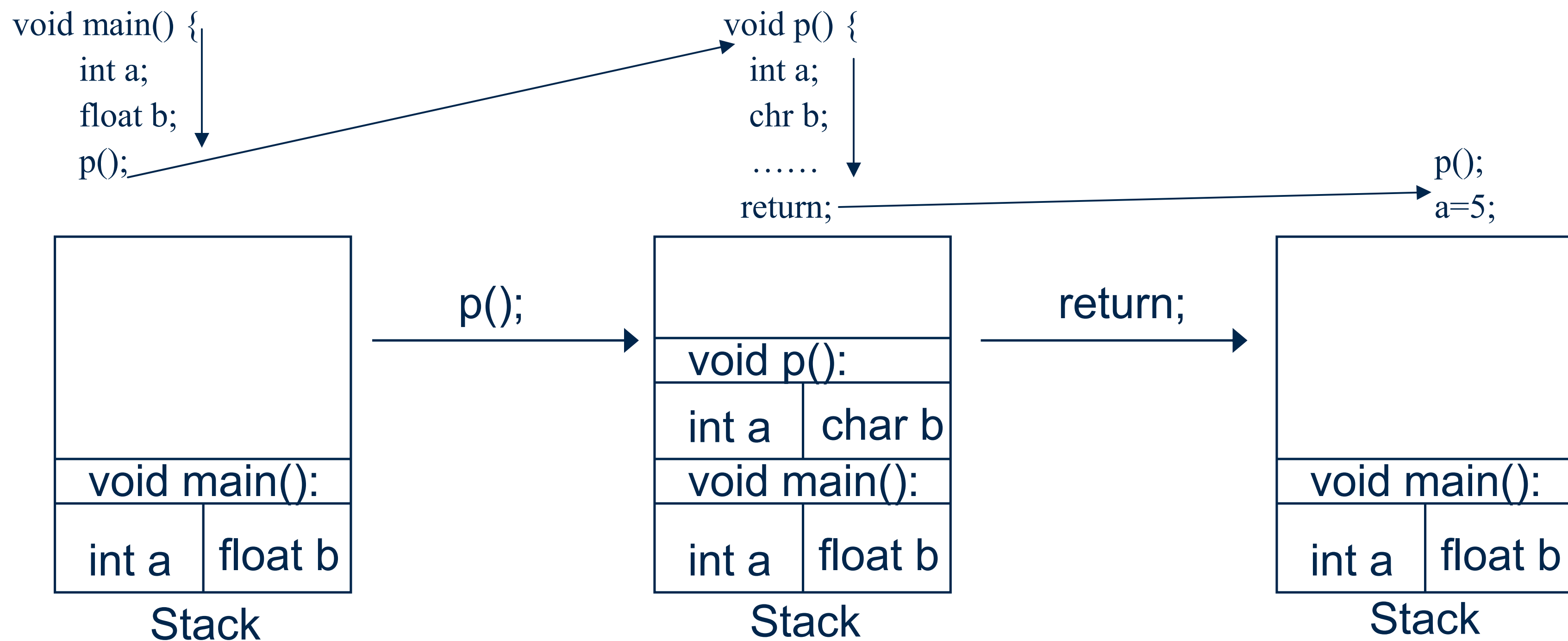
Verschieden Speicherstellen für …

- Ein Programm kann einen gewissen Adressraum (Speicher verwenden)
  - phys./virt. Speicher
- Statische Daten sind:
  - globale Variablen
  - statische Variablen
- Programmcode und statischer Bereich sind durch Compilation festgelegt
- Stack und Heap ändern sich während der Ausführung



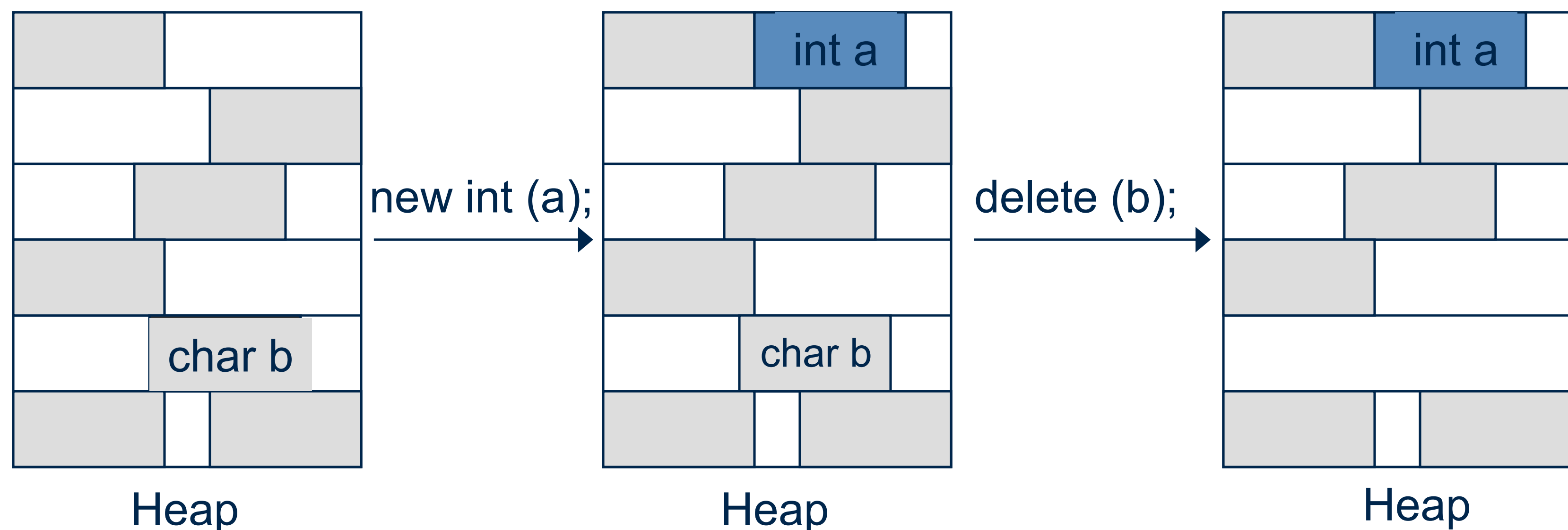
Der Stack

- Verwaltung durch Laufzeitsystem
- Arbeitet nach dem Prinzip „last-in-first-out"
- Verwaltung von Funktionsaktivierungen
- Speicherbereich für lokale Variablen



Der Heap

- Programmkontrollierte Verwaltung
- Speicherzuweisung per Anweisung (new/delete)
- Zugriff über „Zeiger-Variablen“ (Referenz-Variablen)
- Unterstützung durch Laufzeitsystem (Garbage Collection, ...) abhängig von Programmiersprache (JAVA ja!)



Typische Fehlerquellen

- Fehlendes delete ("memory leak")
  - Dynamische Objekte werden nicht freigegeben, auch wenn sie nicht mehr gebraucht werden. Das Programm allokiert immer mehr Speicher und wird bei genügend langer Laufzeit an erschöpften Systemreserven scheitern.
- delete ohne new
  - delete verläßt sich darauf, daß das Argument ein von new gelieferter Zeiger ist: Ansonsten stürzt das Programm i.d.R. ab.
- delete eines beliebigen Zeigers
  - wie vorhergehender Punkt.
- Zeigerzugriff nach delete ("dangling pointer")
  - Zeiger verweist nach delete auf Speicherplatz, an dem kein Objekt mehr existiert. Lesen aus diesem Speicherbereich liefert zufällige Daten, Schreiben führt meist zu einem Programmabbruch.
- Speicherzerstückelung ("heap fragmentation")
  - Nach häufigem Allokieren und Freigeben ist der Heap zerstückelt: Es gibt keine Lücke für eine größere Anforderung mehr, obwohl die Summe aller (kleinen) Lücken ausreichend freien Platz ausweist.

Maßnahmen gegen Fehler

- new und delete zusammenziehen
  - new und delete für eine dynamische Variable im gleichen Block aufrufen
- Zeiger zurücksetzen
  - Jeden Zeiger sofort nach delete mit Nullzeiger überschreiben
- Memory profiling
  - Spezieller Code, der new- und delete-Aufrufe protokolliert und am Ende abgleicht. Bremst den normalen Programmablauf beträchtlich ab.