

## Blöcke und Variablen

- In Blöcken definierte Variablen sind nur innerhalb dieses Blockes sichtbar.
  - ↪ auch in enthaltenen Blöcken
  - ↪ aber **nicht** in umfassenden Blöcken
- In Blöcken definierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken definierte Variablen werden beim Verlassen des Blockes wieder ungültig.
  - ↪ überdeckte Variablen werden wieder sichtbar

# Konstanten

- Literale
- Variablen, die mit Typattribut `const` definiert sind
  - `const Datentyp Bezeichner;`
  - Variable nach Initialisierung schreibgeschützt
  - Beispiel: `const double PI = 3.1415927`
  - Konvention: *Bezeichner* aus Großbuchstaben
- symbolische Konstanten
  - `#define Name Wert`
  - Präprozessor ersetzt *textuell* alle Vorkommen von *Name* durch *Wert*
  - Beispiel: `#define PI 3.1415`
  - Konvention: *Name* aus Großbuchstaben

## Rückgabebetyp `void`

- Funktionen ohne Rückgabewert  
   $\rightsquigarrow$  reine Prozeduren
- dienen z.B. zur bloßen Datenausgabe
- keine `return`-Anweisung *oder* `return`;
- `void prozedur_funktion ( ... ) { ... }`

## Parameterübergabe mit call-by-value

- Vereinbarung einer lokalen Variablen:  
Name und Typ des formalen Parameters
- *Kopie* des aktuellen Parameterwertes als Wert dieser lokalen Variablen
- Im Funktionsrumpf wird immer die lokale Variable verwendet.
  - ↪ **kein Einfluss auf aktuellen Parameter**
  - ↪ keine Seiteneffekte
- Ausdruck des aktuellen Parameters wird nur einmal ausgewertet
- verwendet in: C, C++, Java, C#, PASCAL, ...

## Andere Methoden der Parameterübergabe — nicht in C

- **call-by-value-result**

- zunächst wie call-by-value:
  - \* lokale Variable mit Wert des aktuellen Parameters initialisiert
  - \* keine Änderung am aktuellen Parameter
- am Ende (z.B. bei `return;`):
  - eine weitere automatische Wertzuweisung
  - ↔ Wert der lokalen Variablen → aktueller Parameter

- **call-by-name**

- textuelle Übergabe des aktuellen Bezeichners
- Im Funktionsrumpf wird jedes Vorkommen des formalen Parameters textuell ersetzt.
  - ↪ Seiteneffekt: kann aktuellen Parameter beeinflussen
- Ausdruck des aktuellen Parameters wird mehrfach ausgewertet
- kann zu Konflikten mit globalen Variablen führen  
(falls Variablen des Funktionsrumpfes einen Bezeichner verwenden, der Teil eines aktuellen Parameters ist)

- **call-by-reference**

- Vereinbarung von lokalen Variablen für alle Variablen in den aktuellen Parametern
- referenzieren die Speicheradressen der Variablen in den aktuellen Parametern
- Zuweisungen beeinflussen die Werte der aktuellen Parameter direkt

## Vier Kennzeichen einer Variablen

- Datentyp
- Variablenname
- Wert
- Adresse im Arbeitsspeicher (Primärspeicher)

In C: Zugriff über Namen oder Adresse auf den Wert



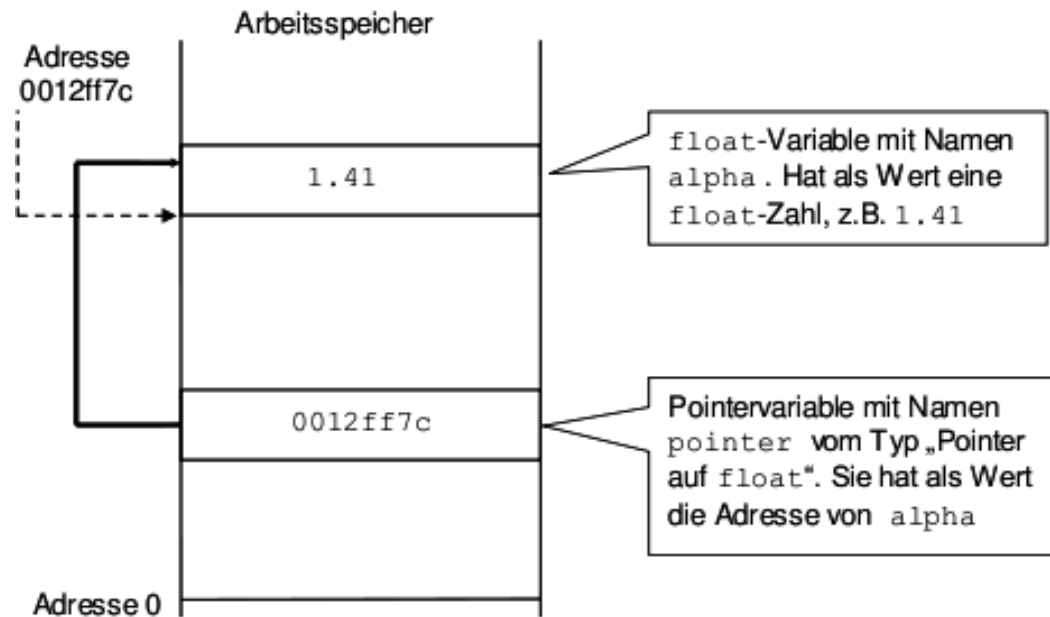
## Adressen von Variablen

- Arbeitsspeicher ist in (gleich große) *Speicherzellen* eingeteilt (z.B. je 1 Byte)
- Speicherzellen sind durchnummeriert (Hexadezimalzahlen)
- **Adresse** einer Variablen ist *Nummer* der Speicherzelle, in der ihr Speicherplatz beginnt.



# Pointer (Zeiger)

- **Pointer:** Variable, deren Wert *Adresse* einer Variablen (oder Funktion) ist



## Pointervariablen

- Definition als Pointer auf *Datentyp*  
↪ Pointertyp und Datentyp des Speicherobjekts sind gekoppelt!
- *Typname \* Pointername;*  
↪ Datentyp: Pointer auf *Typname*
- Beispiel: `float * pointer1;`  
↪ Pointer auf `float` mit Namen `pointer1`

- Vorsicht:

Definition	entspricht
<code>int * pointer, alpha;</code>	<code>int * pointer;</code> <code>int alpha;</code>
<code>int * pointer1, * pointer2;</code>	<code>int * pointer1;</code> <code>int * pointer2;</code>

# Wertzuweisung an einen Pointer

## 1. Adressoperator

- Adressoperator **&** liefert Pointer auf Speicherplatz einer Variablen
- $\&Variable \rightsquigarrow$  Pointer auf *Datentyp* von *Variable*
- kann an Variable vom Typ Pointer auf *Datentyp* zugewiesen werden:  
`pointer = &alpha;`

## 2. Wert einer anderen Pointervariablen

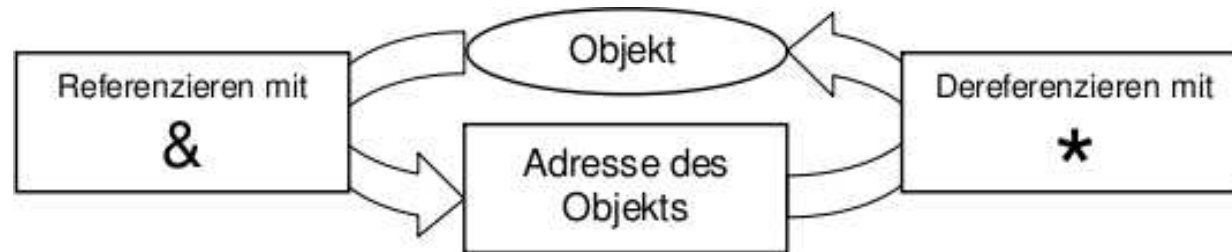
- `pointer2 = pointer1;`
- Übergabe der Adresse von einer Pointer-Variablen an eine weitere
- nur bei Pointern auf den gleichen Typ!

## 3. Konstante NULL

- vordefinierter Pointer, zeigt auf Adresse 0
- zeigt nie auf ein Speicherobjekt

# Dereferenzieren

- **Inhaltsoperator** \*: *\*Pointer*
- \* und & sind invers:  $*\&\alpha$  ist äquivalent zu  $\alpha$

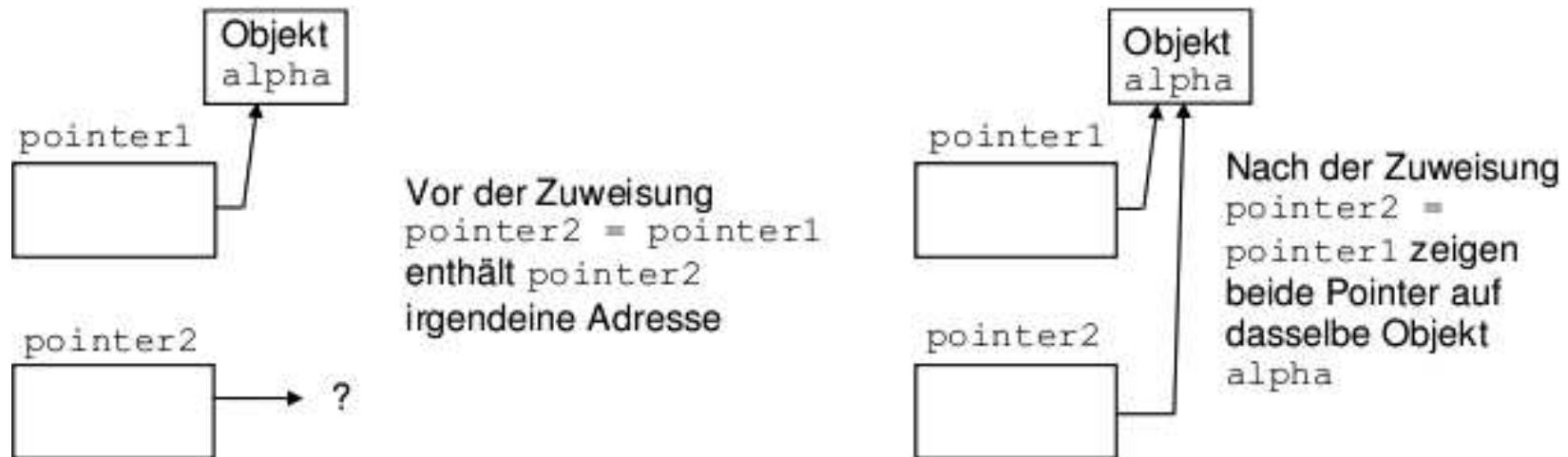


- $*\text{pointer} = \text{Wert}$ ; erlaubt  
     $\rightsquigarrow$  nicht bevor pointer gültige Adresse speichert!

## Arbeit mit Pointern

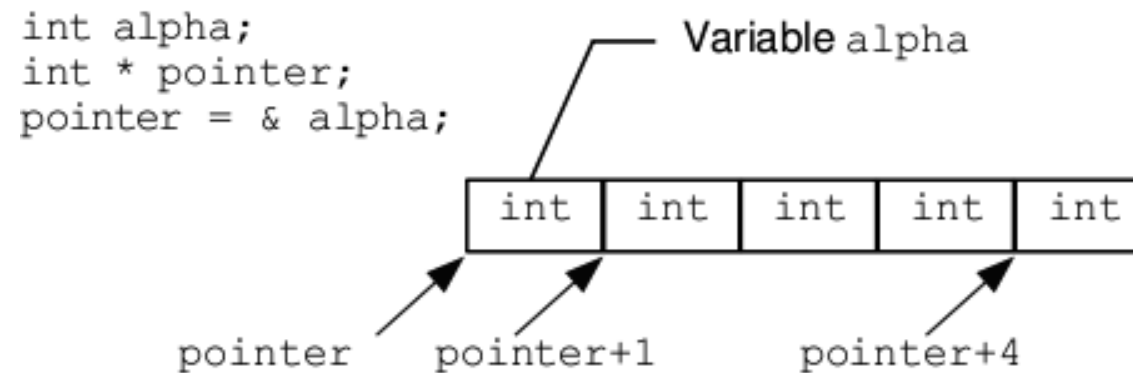
- nicht initialisierte Variablen haben *irgendeinen* Wert!

↪ nicht initialisierte Pointer verweisen auf *irgendeine* Adresse!



# Pointerarithmetik

- Vergleich (`==` und `!=`)
- Addition und Subtraktion (einer ganzen Zahl  $n$ )  
Verschieben des Zeigers um  $n$  Speicherobjekte des Typs, auf den der Pointer zeigt



## Pointer auf void

- `void * Pointername;`
- **untypisierter Pointer**: Datentyp steht nicht fest
- darf nicht dereferenziert werden (zeigt nie auf Speicherobjekte)
- kann in jeden Pointertyp umgewandelt werden (Zuweisung)  
     $\rightsquigarrow$  kein Verlust an Information oder Genauigkeit
- für die Zuweisung von Pointern auf anderen Typ verwenden