

Hydroinformatik I - WiSe 2019/2020

V10a: Pointer

Prof. Dr.-Ing. habil. Olaf Kolditz

¹Helmholtz Centre for Environmental Research – UFZ, Leipzig

²Technische Universität Dresden – TUD, Dresden

³Center for Advanced Water Research – CAWR

Dresden, 20.12.2019

Semesterfahrplan

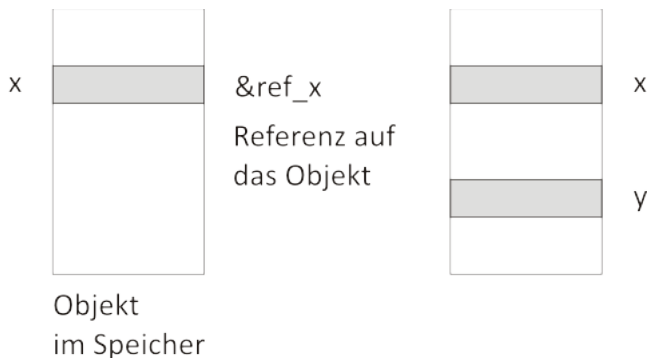
WiSe 2019/2020: Hydroinformatik I, Freitag (3. DS) 11:10-12:40, HÜL/S186/H					
No	KW	Datum	ID	Vorlesung	Dozent
1a	42	19.10.2019	BHYWI-08-01	Hydroinformatik - Einführung	Kolditz
1b	42	19.10.2019	BHYWI-08-02	Compiler (Installation)	Kolditz
2	43	25.10.2019	BHYWI-08-03	Datentypen	Kolditz
3	44	01.11.2019	BHYWI-08-05	Hausaufgaben	Kolditz
5	45	08.11.2019	BHYWI-08-04	Klassen	Kolditz
4	46	15.11.2019	BHYWI-08-06	Programmieren in Nat-Ing	Kalbacher
6	47	22.11.2019	BHYWI-08-07	Input-Output (I/O)	Kolditz
7	48	29.11.2019	BHYWI-08-08	Strings - Textverarbeitung	Kolditz
8	49	06.12.2019	BHYWI-08-09	Hydrologische Modellierung	HA
9	50	13.12.2019	BHYWI-08-10	Hydrologische Modellierung	Kolditz
10	51	20.12.2019	BHYWI-08-11	Pointer & Container	Kolditz
12	2	10.01.2020	BHYWI-08-12	BigData & Water 4.0	Kolditz
13	3	17.01.2020	BHYWI-08-13	Neuronale Netzwerke	Kolditz
14	4	24.01.2020	BHYWI-08-14	ANN / Bayes'sche Netzwerke	Kolditz
15	5	31.01.2020	BHYWI-08-15	BN / Maschinelles Lernen	Kolditz
16	6	07.02.2020	BHYWI-08-16	Klausurvorbereitung	Kolditz

Referenzen und Zeiger

Referenzen und Zeiger (pointer), d.h. die Symbole & und * sind uns bereits mehrfach begegnet, z.B. in der Parameterliste der `main()` Funktion ... Generell können Zeiger und Referenzen als einfache Variable, Parameter oder Rückgabewert (return) einer Funktion auftauchen.

In der Übung E63 haben wir bereits mit Referenzen gearbeitet und festgestellt, dass Referenzen eigentlich nichts anderes als andere Bezeichner (Alias) für eine bereits existierende Instanz eines Objekts sind. Es kann also mehrere solcher Referenzen geben. Bei der Definition eines Zeigers wird also kein neuer Speicher reserviert. Die Instanz des Objekts muss physikalisch (also speichermäßig) natürlich vorhanden sein, sonst 'verabschiedet' sich unser Programm gleich wieder.

Referenzen auf Objekte



Die Abbildung soll das Konzept des Referenzierens noch einmal verdeutlichen.

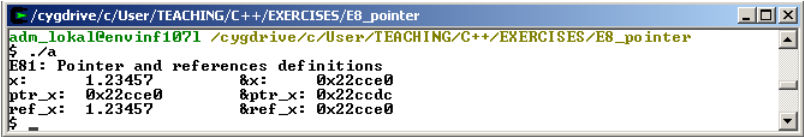
- ▶ `x` ist die Instanz eines Datentyps `T`
- ▶ `&ref_x` ist eine Referenz (Verweis) auf `x`

Übung E711

In der ersten Übung sehen die Definition von Referenzen und Zeigern im Quelltext

```
double x = 1.23456789;  
double* ptr_x;  
double& ref_x = x; // initialisation is needed
```

Die Bildschirmausgabe der ersten Übung zeigt Folgendes:



```
/cygdrive/c/User/TEACHING/C++/EXERCISES/E8_pointer  
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E8_pointer  
$ ./a  
E81: Pointer and references definitions  
x:      1.23457      &x:      0x22cce0  
ptr_x:  0x22cce0    &ptr_x:  0x22ccd0  
ref_x:  1.23457    &ref_x:  0x22cce0  
$ _
```

Abbildung: Referenzen auf Objekte

Zwischen-Summary#1

Das Verstehen und Arbeiten mit Zeigern und Referenzen ist nicht ganz einfach und braucht eine ganze Zeit der praktischen Anwendung. Der *Operator kann beispielsweise als Zeiger auf etwas und gleichzeitig für dynamische Vektoren benutzt werden. Dies scheint zunächst absolut verwirrend zu sein, beim genaueren überlegen aber: * ist lediglich die Startadresse für den Vektor. Merken sie sich erst einmal nur, dass es geschickter (und schneller) ist, nicht mit den Daten-Objekten direkt sondern mit Verweisen darauf (also deren Adressen) zu arbeiten. Die nachfolgende Tabelle und Abbildung sollen das Thema 'Referenzen und Zeiger' noch einmal illustrieren.

Zwischen-Summary#2

Syntax	Bedeutung
<code>double x</code>	Definition einer doppelt-genauen Gleitkommazahl (Speicherbedarf 8 Byte)
<code>double* ptr</code>	Zeiger auf eine doppelt-genauen Gleitkommazahl (Speicherbedarf 4 Byte)
<code>double& ref</code>	Referenz auf eine doppelt-genaue Gleitkommazahl (Speicherbedarf 8 Byte)
<code>double* dVektor</code>	Zeiger auf einen Gleitkommazahl-Vektor

Tabelle: Zeiger und Referenzen

Nochmal eine Grafik ...

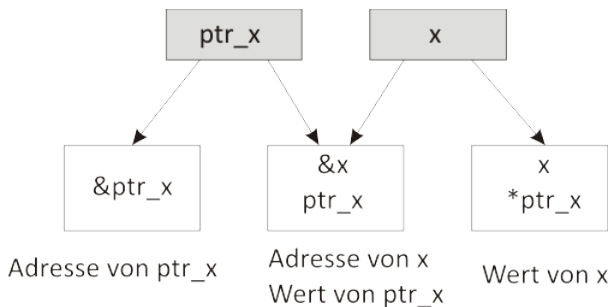


Abbildung: Referenzen und Zeiger

Zeiger

Zeiger sind eine der Grundideen moderner Programmiersprachen. Dabei arbeitet man nicht direkt mit den Daten-Blöcken im Speicher sondern mit deren Adressen. Dies macht insbesondere Sinn, wenn es um große Daten-Objekte geht oder bei Objekten, die erst zur Laufzeit angelegt werden (z.B. Vektoren, Listen, Strings). Das Klassen-Prinzip unterstützt die Zeigertechnik optimal, da die Klasse ihre Daten ja selber verwaltet (siehe Klassenkonstruktor). Der Zeiger (pointer) auf ein Objekt repräsentiert dessen Adresse und Daten-Typ.

Übung E72

Die nachfolgende Übung zeigt, wie wir Zeiger definieren und mit ihnen arbeiten können. Für den Zugriff auf Zeiger gibt es den sogenannten Verweisoperator '*':

```
int main()
{
    int i, *ptr_i; // Definitionen
    i = 100;       // Zuweisung
    ptr_i = &i;   // Dem Zeiger wird die Adresse der integer Variable i
                  // zugewiesen
    *ptr_i += 1;  // Die integer Variable i wird um Eins erhöht (i += 1;
}
```

Geben sie Werte und Adressen der Integer-Variable i und den Zeiger auf i (i_ptr) aus.

NULL Zeiger

Wir haben gesehen, dass Referenzen auf Daten-Objekte zwingenderweise initialisiert werden müssen (sonst streikt der Compiler). Zeiger müssen eigentlich nicht initialisiert werden, ihnen wird bei der Definition eine 'vernünftige' Adresse zu gewiesen. Dies bedeutet noch lange nicht, dass sich hinter dieser Adresse etwas Vernünftiges verbirgt. Daher ist es ratsam, auch Zeiger zu initialisieren, um Nachfragen zu können, ob sie tatsächlich auf existierende Objekte zeigen. Zum Initialisieren von Pointern gibt es den NULL-Zeiger.

```
double* ptr_x = NULL;
ptr_x = &x;
if(!ptr_x) cout << "Warning: ptr_x does not have a meaningful adress";
```

Der linke Operand der Operators = muss immer auf eine 'sinnvolle' Speicherstelle verweisen. Dieser wird daher auch als so genannter L-Wert (L(ef)t value) bezeichnet. Wenn eine Fehlermeldung 'wrong L-value' erscheint, wissen sie, dass keine 'sinnvolle' Adresse vergeben wurde, dies spricht wiederum für eine Initialisierung von Zeigern mit Zero-Pointer.

Zeiger und Arrays

Wie bereits gesagt, die Verzeigerungs-Technik ist eine der Besonderheiten von objekt-orientierten Sprachen, wie z.B. von C++. Mit solchen Pointern können wir auch größere Daten-Objekte wie Vektoren verwalten. Die zentrale Idee ist: Wenn Startpunkt (*id* Adresse) und Länge des Vektors bekannt sind, wissen wir eigentlich alles und können auf alle Vektor-Daten zugreifen.

Statische Objekte

Bei der Einführung der String-Klasse hatten wir bereits mit Zeichenketten zu tun. Aus der Sicht des Speichers ist eine Zeichenkette ein Vektor der eine bestimmte Anzahl von Zeichen enthält. Vektoren können natürlich für (fast) alle Datentypen definiert werden. Im nächsten Kapitel (??) beschäftigen wir uns mit sogenannten Containern, damit können z.B. Vektoren, Listen etc. für ganze Klassen generiert werden.

```
char Zeichenkette[80];  
int iVektor[50];  
double dVektor[50];
```

Dynamische Objekte#1

Bisher haben wir uns fast ausschließlich mit Datentypen beschäftigt, deren Speicherbedarf bereits während der Kompilierung fest steht und reserviert wird (bis auf Strings-Objekte). Es gibt aber auch die Möglichkeit, den Speicherbedarf während der Programmausführung zu ändern - dies geht mit dynamischen Daten-Objekten. Die werden wir folgt deklariert.

```
char* Zeichenkette;  
int* iVektor;  
double* dVektor;
```

Dynamische Objekte#2

Nun müssen wir uns allerdings selber um die Speicherreservierung kümmern. Dies erfolgt mit dem `new`-Operator und wird in der nächsten Übung gezeigt. Wir ahnen schon, wenn es etwas zum Vereinbaren von Speicher gibt, muss es wohl auch das Gegenstück - freigeben von Speicher - geben. Natürlich habe sie Recht. Dies erledigt der `delete`-Operator.

```
double* dVektor;  
dVektor = new double[1000];  
delete [] dVektor;
```

Übung

Die Anwendung beider Operatoren zum Speichermanagement sowie eine Möglichkeit, den verbrauchten Speicher zu messen, werden in der folgenden Übung gezeigt.

```
#include <iostream> // for using cout
#include <malloc.h> // for using malloc_usable_size
using namespace std;

int main()
{
    char* Zeichenkette;           // Definitions
    long size_memory;            // Auxiliary variable for memory size
    Zeichenkette = new char[1000]; // Memory allocation
    size_memory = malloc_usable_size(Zeichenkette); // calculation of mem
    delete [] Zeichenkette;      // Memory release
    return 0;
}
```


Mehrdimensionale Objekte

Zeiger können auch verschachtelt werden: Zeiger auf Zeiger ...
Damit lassen sich mehrdimensionale Datenstrukturen schaffen, z.B.
Matrizen.

```
**matrix
```

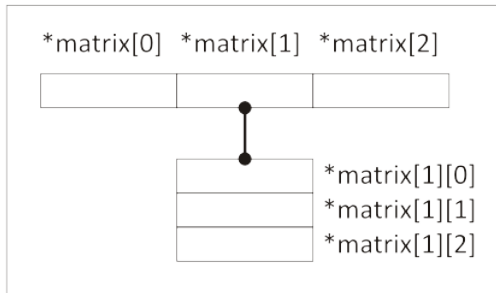


Abbildung: Die Definition einer Matrix mittels Pointer