

# Programmierwerkstatt



## Arrays, Pointer und Referenzen



## Zum Inhalt

- Wir wollen Euch:
  - das Wesentliche vermitteln
  - Fehlerquellen verdeutlichen
  - Verständnis ist uns wichtig
  - „programming by coincidence“ vermeiden
- Themen dieses Vortrags sind:
  - Wozu braucht man Pointer?
  - Wie verwendet man das Schlüsselwort `const` richtig?
  - Wie arbeitet man korrekt mit Arrays?
  - Für was sind Referenzen gut?

20.12.05

Arrays, Pointer & Referenzen

2/27



## Was ist ein Pointer?

- Variable mit Speicheradresse als Inhalt
- NULL-Pointer
- Variable => Pointer mit Adressoperator „&“

```
int a = 5;  
int *p_a = &a;
```

20.12.05

Arrays, Pointer & Referenzen

3/27



## Pointerarithmetik (1)

```
char c = 'D';  
int i = 20;  
int* pi = &i;  
int j = *pi;  
  
cout << pi << endl;  
cout << j;  
  
/*ergibt Ausgabe:  
  
14502  
  
20  
  
*/
```

| Adresse | Inhalt   | Wert  |
|---------|----------|-------|
| 0000000 | 01011010 | ?     |
| 0000001 | 11100001 | ?     |
| 0000002 | 00001110 | ?     |
| 0000003 | 11000100 | ?     |
| ...     | ...      | ...   |
| 0014501 | 01000100 | D     |
| 0014502 | 00000000 |       |
| 0014503 | 00000000 |       |
| 0014504 | 00000000 | 20    |
| 0014505 | 00010100 |       |
| 0014506 | 00000000 |       |
| 0014507 | 00000000 |       |
| 0014508 | 00111000 | 14502 |
| 0014509 | 10100110 |       |
| 0014510 | 00000000 |       |
| 0014511 | 00000000 |       |
| 0014512 | 00010100 | 20    |
| 0014513 | 11100001 |       |
| 0014514 | 10111011 | ?     |

20.12.05

Arrays, Pointer & Referenzen

4/27



## Warum sind Pointer so toll?

- einfacher und effizienter Zugriff auf Daten
- Pointerarithmetik ermöglicht Verwendung von Arrays
- dynamische Speicherverwaltung (Speicher zur Laufzeit reservieren und freigeben)



## Wozu kann man Pointer benutzen?

- Pointer sind typisiert (wegen Pointerarithmetik) - Adresse allein reicht nicht aus

```
<Typ> *a;
a++; //Adressänderung per Postinkrement
--a; //Adressänderung per Predekrement
```

- dereferenzieren mit \* entspricht: Inhalt/Wert der Speicherzelle auslesen

```
int a[] = {1,2,3,4,5,6,7,8,9};
int *b = a+5;
*b == 6 //liefert true
```



## Pointerarithmetik (2)

```
char* wort = „Bla“;
char* wort2 = wort + 2;

cout << wort << endl;
cout << wort2 << endl;
cout << *wort;

/*ergibt Ausgabe:

Bla
a
B
*/
```

| Adresse | Inhalt   | Wert |
|---------|----------|------|
| 0001200 | 01000010 | B    |
| 0001201 | 01101100 | l    |
| 0001202 | 01100001 | a    |
| 0001203 | 00000000 | \0   |
| ...     | ...      | ...  |
| 0014501 | 00000000 |      |
| 0014502 | 00000000 | 1200 |
| 0014503 | 00000100 |      |
| 0014504 | 10110000 |      |
| 0014505 | 00000000 |      |
| 0014506 | 00000000 | 1202 |
| 0014507 | 00000100 |      |
| 0014508 | 10110000 |      |
| 0014509 | 00001100 | ?    |
| 0014510 | 10000000 | ?    |
| 0014511 | 10101010 | ?    |
| 0014512 | 11111110 | ?    |
| 0014513 | 00000001 | ?    |
| 0014514 | 00001110 | ?    |



## Pointerarithmetik (3)

```
short zahlen[2] = {0, 1};
short* pZahl = zahlen + 1;

cout << *pZahl << endl;
cout << pZahl;

/*ergibt Ausgabe:

1
14503
*/
```

| Adresse | Inhalt   | Wert  |
|---------|----------|-------|
| 0000000 | 00010100 | ?     |
| 0000001 | 01000101 | ?     |
| 0000002 | 10011000 | ?     |
| 0000003 | 00100101 | ?     |
| ...     | ...      | ...   |
| 0014501 | 00000000 | 0     |
| 0014502 | 00000000 |       |
| 0014503 | 00000100 | 1     |
| 0014504 | 10110000 |       |
| 0014505 | 00000000 |       |
| 0014506 | 00000000 | 14503 |
| 0014507 | 00000100 |       |
| 0014508 | 10110000 |       |
| 0014509 | 10001001 | ?     |
| 0014510 | 00111000 | ?     |
| 0014511 | 11100000 | ?     |
| 0014512 | 10011111 | ?     |
| 0014513 | 10001111 | ?     |
| 0014514 | 01110000 | ?     |



## Speicher für Pointer reservieren

- falsch:  

```
char *zeichen;  
*zeichen = 'A';
```
- **Ganz böse! irgendein Speicherbereich wird überschrieben.**
- richtig:  

```
char *zeichen = NULL;  
zeichen = new char;  
*zeichen = 'A';  
//sauber beenden:  
delete zeichen;
```



## Das Schlüsselwort const (1)

- `const` macht eine Variable beliebigen Typs konstant, d.h. der Wert kann nicht mehr verändert werden
- beide Zeilen sind äquivalent:  

```
const <Typ> = wert;  
<Typ> const = wert;
```
- bessere Schreibweise:  

```
<Typ> const = wert; //(Erklärung später)
```



## Das Schlüsselwort const (2)

- konstanter Inhalt:  

```
<Typ> const * VarName;
```
- Erlaubte Operationen:
  - neuen Speicher zuweisen

```
char const* a = "Hallo";  
a = new char[5];  
a = "Test";
```



## Das Schlüsselwort const (3)

- konstanter Pointer:  

```
<Typ> * const VarName;
```
- Erlaubte Operationen:
  - Inhalt verändern

```
char * const a = "Wurst";  
a[0] = 'D';
```



## Das Schlüsselwort const (4)

- konstanter Pointer und konstanter Inhalt  
`<Typ> const * const VarName;`
- Es kann weder Inhalt noch Pointer verändert werden
- Eselsbrücke:  
 Das was vor „const“ steht ist konstant.
- DEMOPROGRAMM (pointer.cpp)

20.12.05

Arrays, Pointer & Referenzen

13/27



## const Pointer

```
char c = 'D';
char const * pcC = &c;
char * const cpC = &c;
char const * const cpcC = &c;
```

| Adresse | Inhalt   | Wert  |
|---------|----------|-------|
| 0000000 | 01011010 | ?     |
| 0000001 | 11100001 | ?     |
| 0000002 | 00001110 | ?     |
| 0000003 | 11000100 | ?     |
| ...     | ...      | ...   |
| 0014501 | 01000100 | D     |
| 0014502 | 00000000 |       |
| 0014503 | 00000000 |       |
| 0014504 | 00111000 | 14501 |
| 0014505 | 10100101 |       |
| 0014506 | 00000000 |       |
| 0014507 | 00000000 | 14501 |
| 0014508 | 00111000 |       |
| 0014509 | 10100101 |       |
| 0014510 | 00000000 |       |
| 0014511 | 00000000 | 14501 |
| 0014512 | 00111000 |       |
| 0014513 | 10100101 |       |
| 0014514 | 10111000 | ?     |

20.12.05

Arrays, Pointer & Referenzen

14/27



## Arrays und Pointer

- Arrays haben feste Startadressen
- Der Laufindex geht immer von 0 bis N-1  
`<Typ> Varname[N];`  
`for (i=0; i<N; i++) { //nicht „<=“`  
`cout << Varname[i];`  
`// Alternative`  
`cout << *(Varname+i);`  
`}`
- Für zweidimensionale Arrays gilt folgende Gleichheit:  
`c[2][3] == *((*c+2)+3); //liefert true`

20.12.05

Arrays, Pointer & Referenzen

16/27



## Das geht mit Arrays nicht (so einfach)

```
int a[2][3] = {{0,1,2},{3,4,5}};
int b[2][3];
a++; // 1)
b = a; // 2)
```

- 1) + 2) funktionieren nicht, da Arrays feste Startadressen haben
- Zuweisung 2) geht
  - mit dem Befehl: `memcpy(b, a, sizeof(a));`
  - oder wirklich mit einem Pointer auf ein Array:  
`int (*b)[3] = a; // Klammern sind wichtig`  
`int *b[3]; //erstellt: {int*, int*, int*}`

20.12.05

Arrays, Pointer & Referenzen

17/27



## Speicher für dynamische Arrays

- Der Speicherplatz kann mit dem Befehl `new[size]` bereitgestellt werden
  - geht nur wenn noch genügend freier Speicher da ist
- nicht mehr benötigter Speicher muss mit `delete[]` wieder freigegeben werden
  - Der Speicher könnte sonst evtl. volllaufen
- Beispiel:

```
int *a = new int[20];  
// Auf Array arbeiten...  
a[4] = 5;  
delete [] a;
```

20.12.05

Arrays, Pointer & Referenzen

18/27



## Arrays - häufige Fehler

- falsch:

```
char a[6]; //5 Zeichen + 1 Nullbyte  
a = "Hallo"; //geht nicht !!!
```
- richtig:
  - per Funktion: `strcpy(a, "Hallo");`
  - Oder direkt mit Pointer arbeiten:

```
char *c = "Hallo";  
c = "Test"; //=> das geht!!!
```

20.12.05

Arrays, Pointer & Referenzen

19/27



## Binärbaum

- Linksseitige, n-äre Bäume sind als Array darstellbar (siehe Vorlesung)
- Array hat Größe max. Knotenanzahl ( $N = 2d - 1$ )
- erstes Element speichert Größe des Arrays
  - wird sowieso nicht genutzt
  - spart die separate Übergabe dieses Wertes
- Baumstruktur kann über einen Breitendurchlauf (breadth-first) in ein Array linearisiert werden
- DEMOPROGRAMM (main.cpp + Tree.h)

20.12.05

Arrays, Pointer & Referenzen

20/27



## Referenzen

- Alle Bezeichnernamen sind implizit Referenzen
- explizite Referenzen dienen als alternative Namen für bereits bestehende Variablen

```
<Typ> var = value;  
<Typ> &ref = var;
```
- `var` und `ref` zeigen nun auf den selben Wert im Speicher
- Wertänderung bei einer der Variablen wirkt sich auf alle anderen gleichermaßen aus

```
ref = newValue; //var ist nun auch newValue
```

20.12.05

Arrays, Pointer & Referenzen

21/27



## Referenzen – häufige Fehler

- eine Variable, die referenziert werden soll muss schon deklariert und initialisiert sein!  

```
int b;  
int &a = b;
```
- nicht zu verwechseln mit dem Adressoperator „&“
- Zweck: Einfachheit und Zugriffssicherheit (erst später in C++ eingeführt)



## Call by Value

- Gängigste Parameterübergabe an Funktionen  

```
<Typ> funkName(<Typ> var) {...}
```
- übergebene Ausdrücke werden zuerst ausgewertet  
– Beispiele:  $(i + 1)$ ,  $(2 != 5)$
- Zuweisung an die im Funktionskopf deklarierten (neuzuerstellenden!) Variablen  
– entsprechende Speicherbereiche werden kopiert  
– häufige Aufrufe bremsen Geschwindigkeit
- Berechnungen auf den Variablen gehen verloren
- es sind keine Seiteneffekte möglich



## Call by Reference

- Verwendung von Referenzen eigentlich nur bei Funktionsaufrufen sinnvoll (oft in APIs verwendet)  

```
<Typ> funkName(<Typ>& var) {...}
```
- übergebene Variablen werden direkt manipuliert
- es können keine Ausdrücke übergeben werden
- Seiteneffekte sind mittels `const` vermeidbar
- Zweck: Effizienz  
– große Datentypen (z.B. Klassen) müssen nicht kopiert werden
- DEMOPROGRAMM (main.cpp)



## Fragerunde

# FRAGEN?



## Zum Schluss...

- Es ist noch kein Meister vom Himmel gefallen!
- Am Anfang ist es immer schwer in eine Programmiersprache einzusteigen
- Stellt daher bitte Fragen wenn euch etwas unklar ist  
→ Wir werden schließlich dafür bezahlt!
- Erreichbar sind wir
  - persönlich in den Pools oder
  - über die Mailingliste  
[progwerkstatt@informatik.uni-ulm.de](mailto:progwerkstatt@informatik.uni-ulm.de)
- Gebt bitte auch denen Bescheid die nicht da sind



## ...Danke für die Aufmerksamkeit!

- Wir wünschen allen frohe Weihnachten und einen guten Rutsch ins neue Jahr!

