

# Programmiersprache 1 (C++)

Prof. Dr. Stefan Enderle  
NTA Isny

# 9. Zeiger

# Arbeitsspeicher / Adressen

- Der Arbeitsspeicher des Computers (RAM) besteht aus einem Feld von Speicherzellen, beginnend bei Adresse 0. (Bild)
- Einen Teil des Speichers nimmt das Betriebssystem in Anspruch. Der Rest steht für Programme zur Verfügung.
- Wenn ein C-Programm eine Variable deklariert, reserviert der Compiler eine Speicheradresse (mit evtl. mehreren Bytes) für den Wert der Variablen.

# Zeiger

- Ein Zeiger ist eine Variable, die die Speicheradresse einer anderen Variable beinhaltet.
- Zeiger (engl. Pointer) „zeigen auf eine Speicherstelle“.
- Zeiger haben trotzdem einen Typ!  
Der Typ gibt an, wie der Wert an der entsprechende Speicherzelle interpretiert wird.
- Ein Zeiger, der auf Nichts zeigt besitzt den Wert NULL.

# Zeiger - Deklaration

- Deklaration eines Zeigers:

```
int* i;  
char* c;  
double* d;  
string* s;
```

- „Traditionelle Schreibweise“:

```
int *i;
```

- Achtung: Uninitialisierte Zeiger zeigen „irgendwo“ hin!!!

# Zeiger - Vorsicht

- Bei „traditioneller Schreibweise“ Mehrfachdeklaration möglich:

```
int *i, *j, *k;
```

- Achtung: Bei „schöner Schreibweise“ geht dies nicht:

```
int* i, j, k;
```

Hier sind `j` und `k` keine Pointer sondern `int`!!

- Sauberste Lösung:

```
int* i;  
int* j;  
int* k;
```

# Adress-Operator &

- Der Adress-Operator & bestimmt die Speicheradresse der nachfolgenden Variablen.

```
int variable;  
cout << &variable;    // Speicheradresse  
  
string name;  
cout << &name;        // Speicheradresse
```

Ausgabe (z.B.): 0xbfebd600

# Adressen zuweisen

- Einem Zeiger kann die Adresse einer Variablen zugewiesen werden:

```
int variable;  
int* zeiger;  
zeiger = &variable;           // Adresse
```

- Ebenso kann einem Zeiger eine beliebige Speicherstelle zugewiesen werden:

```
int* zeiger;  
zeiger = (int*) 1000;         // Adresse 1000
```

(Hier wird ein Cast benötigt, da 1000 ein Integer ist und erst zu einem int-Pointer gemacht werden muss.)

# Zeiger initialisieren

- Ein Zeiger kann wie andere Variablen direkt bei der Deklaration initialisiert werden:

```
int variable;  
int* zeiger = &variable; // Adresse
```

- Einen Zeiger sollte man grundsätzlich initialisieren!!
- Vor der Initialisierung zeigt ein Zeiger „irgendwo“ hin!!!

```
int* i; // Zeiger nach Irgendwo!  
int* j = NULL; // Saubere Initialisierung
```

# Indirektionsoperator \*

- Zeigt ein Zeiger auf einen Wert im Speicher, so kann dieser Wert mit Hilfe des „Indirektionsoperators“ \* ausgelesen werden:

```
int   variable = 5;
int*  zeiger   = &variable;           // Adresse
cout << zeiger;                       // Adresse ausgeben
cout << *zeiger;                       // Wert ausgeben
```

- Der Indirektionsoperator \* liefert den Wert an der Speicheradresse, auf die der nachfolgende Zeiger zeigt.

# Indirektionsoperator \*

- Als Typinformation steht dem Indirektionsoperator der Typ des Zeigers zur Verfügung:

```
int variable = 5;
int* zeiger = &variable;
cout << *zeiger; // als int ausgeben
```

```
float variable2 = 5.1;
float* zeiger2 = &variable2;
cout << *zeiger2; // als float ausgeben
```

# Zeigertypen casten

- Manchmal möchte man einen Wert an einer Speicherstelle anders interpretieren.
- Beispiel: Man möchte eine Zahl vom Typ float als einzelne Bytes auslesen:

```
float f = 5.0;
float* zeiger = &f;    // Adresse von f
cout << *zeiger;      // Indirekter Zugriff
```

```
char* z;
z = (char*) zeiger;    // Zeiger auf Bytes
cout << (int) *z;      // Ausgabe als int
cout << (int) *(z+1);  // Ausgabe als int
```

# Zeiger und Arrays

- Zwischen Zeigern und Arrays besteht eine enge Beziehung:

Der Name eines Arrays ist eigentlich ein Zeiger auf das erste Element:

```
int array[10] = {1,2,3,4,5,6,7,8,9,0};  
cout << *array;           // Ausgabe: 1
```

# Zeiger und Arrays

- Als Zugriff auf die Array-Elemente haben wir bisher den Operator [ ] benutzt:

```
int array[10] = {1,2,3,4,5,6,7,8,9,0};  
cout << array[4];           // Ausgabe: 5
```

- Da der Array-Name Zeiger auf Element 0 ist, wäre auch folgende möglich:

```
int array[10] = {1,2,3,4,5,6,7,8,9,0};  
cout << *(array+4);        // Ausgabe: 5
```

- Allgemein:  $a[n]$  entspricht  $*(a+n)$ , wobei für  $n$  die korrekte Größe der Array-Objekte eingesetzt wird!

# Zeiger und Strukturen

- Zeigt ein Zeiger `p` auf eine Struktur, so kann auf deren Elemente mit dem **Pfeil-Operator** zugegriffen werden:

```
struct Person
{
    string name;
    int  alter;
}
```

```
Person hans;
Person* p = &hans;           // Pointer auf Person
p->name = "hans";           // Zugriff auf name
p->alter = 29;               // Zugriff auf alter
```

# C-Strings

- In C gibt es keinen Datentyp „String“
- Stattdessen wird als String ein Array von Zeichen benutzt:  

```
char name[100];
```
- Bei der Deklaration wird ein Feld mit 100 Zeichen Platz allokiert. `name` enthält den Zeiger darauf.  
(Bild)
- Der Typ von `name` ist also „Zeiger auf ein Zeichen“:  

```
char* name;
```

# C-Strings

- Initialisierung eines „C-Strings“ funktioniert:

```
char name[100]="Anna";
```

- Sogar ohne Angabe der Länge automatisch:

```
char name []="Anna";
```

(Hier werden 5 (!) Zeichen allokiert, nämlich „Anna“ mit abschließender 0.)

# C-Strings

- Problem: Da `name` nur ein Zeiger ist, kann man ihm keinen Wert (also String) zuweisen!

- Beispiel:

```
char name1[100]="Anna";  
char name2[100]="Bernd";
```

```
name1=name2; // Was passiert hier?
```

- In C zeigt nun der Zeiger `name1` ebenfalls auf die Speicherstelle, wo „Bernd“ steht.
- In C++ ist die Zuweisung von Array-Pointern nicht erlaubt und gibt einen Fehler.

# C-Strings

- Für String-Operationen in C nutzt man die Funktionen der Bibliothek `<string.h>`:
  - `char *strcpy(s, ct)`  
Zeichenkette `ct` nach `s` kopieren, inklusive `'\0'`.
  - `char *strcat(s, ct)`  
Zeichenkette `ct` hinten an die Zeichenkette `s` anfügen.
  - `int strcmp(cs, ct)`  
Zeichenketten `cs` und `ct` vergleichen.  
Liefert `<0` wenn `cs<ct`, `0` wenn `cs==ct`, oder `>0`, wenn `cs>ct`.
  - `char * strchr(cs, c)`  
Liefert Zeiger auf das erste `c` in `cs` oder `NULL`, falls nicht vorhanden.
  - `char * strstr(cs, ct)`  
Liefert Zeiger auf erste Kopie von `ct` in `cs` oder `NULL`, falls nicht vorhanden.
  - `size_t strlen(cs)`  
Liefert Länge von `cs` (ohne `'\0'`).

# Vergleich: C++-Strings

- In C++ gibt es den Datentyp `string`
- Er kann wie andere interne Datentypen benutzt werden:
  - Deklaration: `string name;`
  - Initialisierung: `string name="Anna";`
  - Auslesen: `cout << name << endl;`

# Vergleich: C++-Strings

- Auf C++-Strings sind weitere Operationen definiert:
  - Zuweisung:

```
string name1="Anna";  
string name2;  
name2 = name1; // name2 enthält Kopie!
```
  - Verkettung: `name = name1 + name2;`
  - Vergleiche:
    - `name1 == name2`
    - `name1 != name2`
    - `name1 > name2`
    - `name1 < name2`

# Vergleich: C++-Strings

- Länge eines Strings:  
`len = name.size();`  
`len = name.length();`
- Verlängern/Verkürzen:  
`name.resize(n, '*');`