Fachhochschule Bingen

Programmieren

Zeiger / Arrays
Referenz

Prof. Dr. Maximilian Mengel,
Professur Programmiermethodik,
Grundlagen der Informatik und Multimedia
Gebäude 1, Raum 212
Tel.: 06721-409 152
E-Mail: mengel@fh-bingen.de

Pointer /Zeiger

- „Ein Zeiger ist eine Variable, die die Adresse einer anderen Variablen enthält.“ *
- Zeiger verweisen im Allgemeinen auf Variablen eines bestimmten Datentyps
  - Zeiger auf eine Integer Variable
  - Zeiger auf eine Double-Variable
- Ein Zeiger auf eine Integer-Variable wird folgendermaßen vereinbart

```
int *px;
```

*Kerninghan/Ritchie: Programmieren in C

Pointer /Zeiger

- Der Stern-Operator vor einem Zeiger bezeichnet die Variable auf die verwiesen wird; diese Variable kann „normal“ benutzt werden
  - *px = 3*5;
  - printf(“Ein Integer Wert: %d“, *px);
  - int x = *px;
  - ++(*px)
  - (*px)-- // Klammern sind notwendig

Pointer /Zeiger

- Mit dem &-Operator erhält man die Adresse einer Variablen
- Einem Zeiger kann man die Adresse einer Variablen zuweisen

```
int *px, *py;
int x = 5, y = 6;
px = &x;
py = &y;
*px = y;
*py = *px * y;
```

Vektoren

- Vektoren fassen mehrere Datenwerte gleichen Typs in einer Variablen zusammen

  int a[10];
- Ein Zeiger auf eine Integer-Variable kann auch auf Elemente in a zeigen

  int *pa;

  pa = &a[0]; // äquivalent zu pa = a;
- Durch diese Anweisung verweist pa auf das erste Element im Feld a
  - *pa == a[0] und pa == a

Adress-Arithmetik

- Mit Zeigern auf Vektoren/Felder kann Adress-Arithmetik betrieben werden

  pa+1 == &a[1] und *(pa+1) == a[1]

  int i =4;

  pa+i == &a[i] und *(pa+i) == a[i]
- Diese Adress-Arithmetik funktioniert immer, unabhängig von verwendeten Datentyp

  struct complex ca[10], *pca;

  pca = ca;

  pca+i == &ca[i] und *(pca+i) == ca[i]

Vektor-Arithmetik

- Die Analogie zwischen Vektoren und Zeigern „funktioniert“ jedoch auch in umgekehrter Reihenfolge

  int a[10], *pa;

  printf(“ A[%d] = %d\n“, 3, *(a+3));

  pa = a + 3; // entspricht pa = &a[3]
- Sogar folgendes funktioniert:

  pa =a;

  printf(“ A[%d] = %d\n“, 3, pa[3]);

Vektor / Zeiger

- Zusammenfassend kann folgendes festgehalten werden:
  - „statt Vektor Namen und Index Ausdruck kann man immer einen Zeiger und Abstand angeben und umgekehrt“ *
- Unterschied zwischen Vektoren und Zeigern:
  - Ein Zeiger ist eine Variable
  - Ein Vektor-Name ist eine Konstante und darf nicht verändert werden
    - Fehler: a++;
    - Fehler: a = pa;

***Kerninghan/Ritchie: Programmieren in C**

Zwei- und Mehrdimensionale Felder

- Ein Zweidimensionaler Vektor ist...
  - „… in Wirklichkeit ein eindimensionaler Vektor, bei dem jedes Element wieder ein Vektor ist“*
- Folgende Anweisungen sind identisch:
  - int feld[2][3], *px;
  - px=feld[0]; oder px=&feld[0][0]; oder px = (int*)feld;
- Folgendes funktioniert jedoch nicht:
  - px = feld[0]; // Noch OK
  - printf(“%d“,px[1][2]); // Jetzt gibt es einen Fehler

*Kerninghan/Ritchie: Programmieren in C

Übung

- Geben Sie die Ausgaben an:

```
main()
{
   int feld[2][3] = {{1,2,3},{4,5,6}};
   int *px;
   px = (int *)feld;
   printf("%d\n",*px);
   px = feld[0];
   printf("%d\n",px[1]);
   px = feld[1];
   printf("%d\n",*px);
   px = &feld[1][2];
   printf("%d\n",px[0]);
}
```


Felder von Zeigern

- Felder können beliebige Datentypen als Elemente besitzen; also auch Zeiger:
  - int *px[3]; // vereinbart Vektor mit 3 Zeiger auf int
  - int x0[3] ,x1[3], x2[3];
  - px[0] = x0; px[1] = x1; px[2] = x2;
  - printf(“%d“,px[1][2]); // Funktioniert!!!
- Die Vektoren, auf die die Zeiger verweisen müssen jedoch nicht gleich groß sein:
  - int y0[2] ,y1[3], y2[4];
  - px[0] = y0; px[1] = y1; px[2] = y2;
  - printf(“%d“,px[1][2]); // Funktioniert!!!
  - printf(“%d“,px[0][2]); // Fehler bzw. undefinierte Ausgabe

Zeiger als Übergabeparameter

- Das als Call by Reference eingeführte Übergabeverfahren bei Variablen wird nun auch klarer
  - Übergeben wird ein Zeiger auf eine Variable
  - über den *-Operator wird dementsprechend in der Funktion auf die außerhalb der Funktion befindliche Variable zugegriffen die „on Reference“ übergeben wurde
  - Sollte jedoch der Zeiger verändert werden, kann auch nicht mehr auf die entsprechende Variable zugegriffen werden

Vektoren als Übergabeparameter

- Die Analogie zwischen Zeigern und Vektoren trifft auch für die Übergabe als Parameter zu
  sort(int feld[10]);
- ist equivalent zu
  sort(int *feld);
- Dementsprechend funktioniert auch folgende Aufrufe
  - int f1[10], *pf1;
  - pf1=f1;
  - sort(f1);
    // oder z.B.: sort(&f1[3]); sort(f1+3); sort(&pf[3]), …

Zwei- und Mehrdimmensionale Vektoren als Parameter

- Sollen Zweidimmensionalen Vektoren übergeben werden muß klar sein wie groß die „innere“ Dimension ist:
  - test(int feld[2][3]);
  - test(int feld[][3]);
  - test(int (*feld)[3]);
  - falsch: test(int *feld[3]);
  - falsch: test(int feld[][]);
- Bei n-Dimensionalen Feldern müssen die n-1 inneren Dimensionen klar sein

C++ Erweiterungen: Schreibweisen / Referenzen

- Alternative Schreibweisen:
  - int* p; // entspricht int *p;
  - int* v[10] // entspricht …
    // Vektor der Länge 10 mit Zeigern auf int
  - int (*z)[10] // entspricht …
    // Ein Zeiger auf int-Vektor der Länge 10
- Referenz
  - int i = 5; // vereinbart Objekt i
    int& r = i; // vereinbart alternativen Namen für i
    r = 7; // sowohl r und i haben den Wert 7

Beispiele/Fehler: Schreibweisen und Referenzen

- Vorsicht:
  - int* p, z; // entspricht int *p;
    // und int z;
  - int i = 5;
    int& r = i; // r und i referenzieren für immer den
    // selben Speicherbereich
    int j = 3;
    r = j;
    j = 4; // r und i = 3 | j = 4;
    int& k = r; // k und r und i = 3 | j = 4;
    int& p = 5; // nicht erlaubt !!!

C++ Erweiterung: Funktionsparameter Call by Reference

- Call by Reference:
  - Statt Pointer (int *x) per Referenz (int& x):

```
void abs(int& x)
{
   x = (x<0)? -x : x;      // Kein * mehr nötig!!!
}
main()
{
int i = -10;
abs(i);        // kein & mehr nötig !!!
printf(%d,i);  // Ausgabe: 10 !!!
}
```

- Vorsicht Verwechslungsgefahr:
  - Call by Reference // Call by Value !!!

Dynamische Daten, Objekte und Strukturen

- Statische Daten
  - Variablen genau definierten Inhalts
  - Anzahl ist exakt bekannt oder wird durch maximale Obergrenze beschränkt
  - Anpassung an neue (größere) Problemgröße bedeutet verändern des Programms und neues Übersetzen
- Dynamische Daten
  - Anlegen von Daten bei und nach Bedarf
  - Anpassbar an beliebige Problemgrößen

Dynamische Felder

- Statische Felder
  - Genau definierte Anzahl an Elementen
    - Felder werden häufig zu groß angelegt
    - Maximale Feldgröße genügt nicht
- Dynamische Felder
  - Speicherplatz des Feldes wird zur Laufzeit angefordert, wenn die Problemgröße bekannt ist
  - Zugriff auf das Feld per Pointer
  - Speicherplatz wird wieder freigegeben wenn die Daten nicht mehr benötigt werden

Dynamischer Speicher

- C bietet folgende Funktionen zum Umgang mit dynamisch angefordertem Speicher <stdlib.h>
  - void * malloc (unsigned Groesse);
    Belegt einen Speicherbereich von Groesse Bytes. Der zurückgegebene Zeiger auf void sollte dem entsprechenden Datentyp angepaßt werden.
  - void * calloc(unsigned Anzahl, unsigned Groesse);
    Belegt einen Speicherbereich von
    Anzahl * Groesse Bytes.
  - void free(void * Verweis);
    Gibt den übergebenen Speicherbereich frei

Dynamische Felder

- Programmbeispiel (char-Feld):

```
char * pFeld;
unsigned i, groesse;
printf("Wie groß soll das Feld sein ?");
scanf("%u",&groesse);
pFeld = (char *) malloc(groesse);
for (i = 0; i < groesse; ++i)
   pFeld[i] = ' ';
…
free (pFeld);
```


Dynamische Felder

- Funktion sizeof(datentyp)
  - Gibt zu einem Datentyp die Anzahl der Bytes
- Programmbeispiel (Float-Feld):

```
float * pFeld;
…
pFeld = (float *) calloc(groesse, sizeof(float));
for (i = 0; i < groesse; ++i)
   pFeld[i] = 0;
…
free (pFeld);
```


Dynamische Felder

- Programmbeispiel (struct-Feld):

```
typedef struct {double re, im;} complex;
complex * pFeld;
…
pFeld = (complex *) calloc(groesse, sizeof(complex));
for (i = 0; i < groesse; ++i)
   {
   pFeld[i].re = 0.0;
   pFeld[I].im = 0.0;
   }
…
free (pFeld);
```


Übung

- Programmieren Sie einen FIFO-Stack für Integerzahlen
  - Neben den „üblichen" Funktionen wie push(), pop() und size() soll sich ihr Stack dynamisch an die Anzahl der Elemente anpassen
  - Wenn Ihr Stack voll ist soll automatisch eine Anpassung an die aktuelle Anzahl der Elemente erfolgen, indem die Stack-Kapazität verdoppelt wird
  - Die ursprüngliche Stack-Kapazität soll 20 Integerzahlen betragen