Multidimensionale Arrays in C

Florian Wilkens

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

21.11.2013

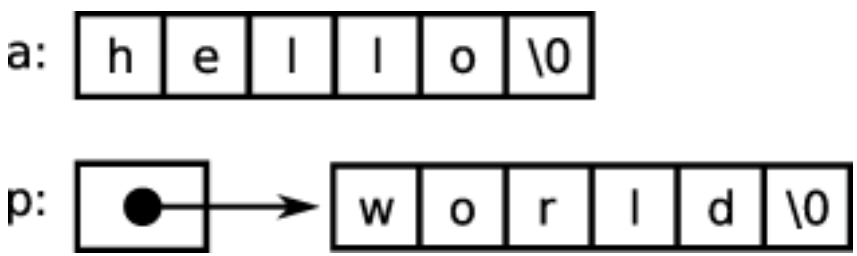
Gliederung

Einleitung

- Bekannt:
  - Indizierungsoperator: `arr[i]`
  - Pointerarithmetik: `ptr + i`
- Nicht so bekannt:
  - `arr[i]` ist definiert als `*(arr + i)`
  - Daraus folgt: `i[arr] = *(i + arr) *(arr +i) = arr[i]`
- $\Rightarrow$ Jeder (indexierte) Speicherzugriff folgt dieser Regel!

Was ist eigentlich ein Array?

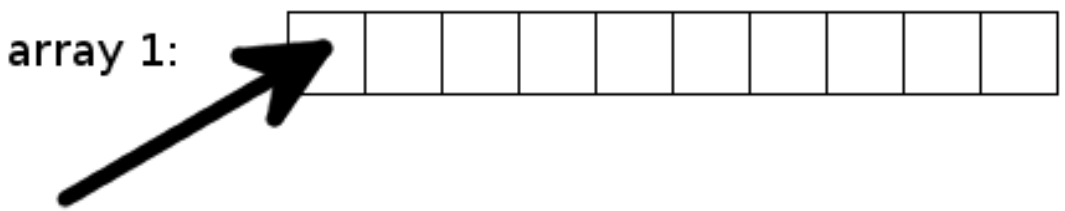
- "An array is a single, preallocated chunk of contiguous elements (all of the same type), fixed in size and location." [Sum13a]



- "second-class citizen"
  - Keine Zuweisungen nach Initialisierung
  - In Ausdrücken nahezu immer "Array-Decay"

Was ist eigentlich Array-Decay?

- Array-Decay
  - Zerfall zu Pointern auf erstes Element
  - Ausnahmen (C99): &-Operator, `sizeof()` und string-Literale
- Was bedeutet das für uns?
  - Selten tatsächliche Verwendung von Arrays
  - `[i]` ist eigentlich ein Pointer-Operator!



Indizierung von eindimensionalen Arrays

- Betrachten wir folgendes Array `int arr[5]`
- Was passiert bei der Auswertung von `arr[i]`?
  - Typ von `arr` ist `int ()[5]`
  - `arr[i]` $\Rightarrow$ `*(arr + i)`
  - Typ von `arr` im Kontext mit '+' ist `int*`
  - Zur Adresse wird `i*sizeof(int)` addiert
- $\Rightarrow$ Element ist an der Adresse "`&arr[0] + 4i`"

Arrays als Funktionsparameter

- Besonderheit: Pointer-Decay bei Definition
- $\Rightarrow$ Verlust von Typinformationen (`sizeof()`)

```
/* eindimensionale Arrays */
void array_function(int values[5])
{
    printf("%d\n", sizeof(values));
        // Erhofft: 20 Ausgabe: 8
}
/* Kompilieren ergibt */
void array_funtion(int* values)..

void array_function(int values[5])
{
    printf("%d\n", sizeof(*values)); // Erwartet: 4 Ausgabe: 4
```

Arrays als Funktionsparameter II

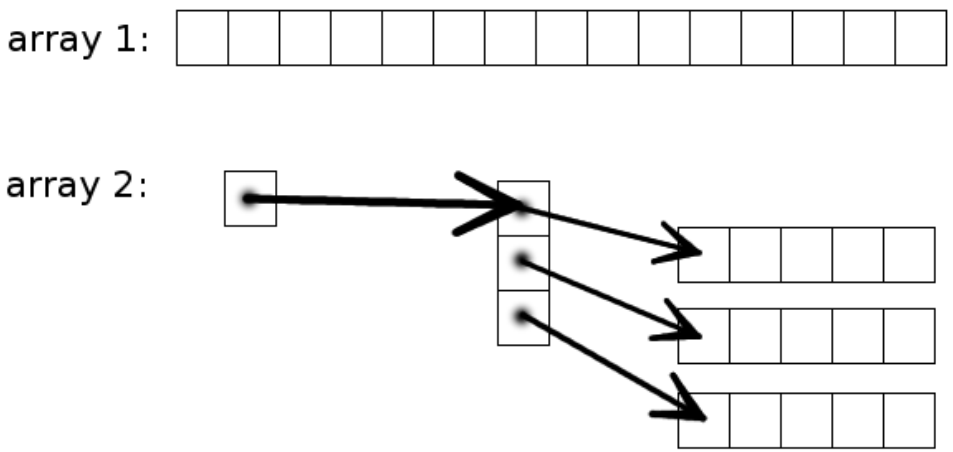
- Besonderheit: Pointer-Decay bei Definition
- $\Rightarrow$ Verlust von Typinformationen (Arraygrenzen)

```
void array_function(int values[5])
{
        printf("%d\n", sizeof(values));
}

int values[3] = { 1, 2, 3 };
```

Was ist eigentlich ein multidimensionales Array?

- Zwei wesentliche Typen
  - "Array von Arrays" `char array1[3][5]`
  - "Pointer auf Pointer" `char **array2`
- Mischformen (`char (*array3)[5]`)



Verschiedene Arten der Allokation

- Diverse Möglichkeiten je nach Speicherort

- Stack
  - (Multidimensionale) Arrays
  - Keine Freigabe notwendig
- Heap
  - Benutzung von c/malloc
  - (meist) Limitierung auf Pointer
  - Freigabe mittels free()

Allokation auf dem Stack

- Allokation

```
/* mit statischen Dimensionen -> Arrays von Arrays */
char arr[2][2][2];

/* mit dynamischen Dimensionen (seit C99) -> Arrays von Arrays */
char arr2[n][n][n];
```

- Freigabe

}

Allokation auf dem Heap I

- “Traditionelle Variante” - geschachteltes (c/malloc)
- Allokation

```
int dim1 = 2, dim2 = 2, dim3 = 2;
char ***arr3 = calloc(dim1, sizeof(*arr3));
for(i = 0; i < dim1; i++)
{
    arr3[i] = calloc(dim2, sizeof(**arr3));
    for(j = 0; j < dim2; j++)
        arr3[i][j] = calloc(dim3, sizeof(***arr3));
}
```

Allokation auf dem Heap II

- “Traditionelle Variante” - geschachteltes (c/malloc)
- Freigabe

```
for (int i = 0; i < dim2; ++i)
{
        for (int j = 0; i < dim1; ++j)
                free(arr3[i][j]);
        free(arr3[i]);
}
free(arr3);
```

Allokation auf dem Heap III

- Kontinuierlicher Speicherbereich über Pointer

- Allokation

```
int dim1 = 2, dim2 = 2, dim3 = 2;
char ***arr4 = calloc(dim1, sizeof(*arr4));
for(i = 0; i < dim1; i++)
{
        arr4[i] = calloc(dim2, sizeof(**arr4));
}
arr4[0][0] = calloc(dim1 * dim2 * dim3, sizeof(***arr4));
for(i = 0; i < dim1; i++)
{
        for (int j = 0; j < dim2; j++)
                arr4[i][j] = arr4[0][0] + (i * dim1) + (j * dim2);
}
```

Allokation auf dem Heap IV

- Kontinuierlicher Speicherbereich über Pointer
- Freigabe

```
free(arr4[0][0]);
for (int i = dim1; i > 0; --i)
{
        free(arr4[i]);
}
free(arr4);
```

Allokation auf dem Heap V

- Kontinuierlicher Speicherbereich (eindimensional)
- Allokation

```
int dim1 = 2, dim2 = 2, dim3 = 2;
char *arr5 = calloc(dim1 * dim2 * dim3, sizeof(*arr5));
```

- Zugriff

```
char c = arr5[i * dim1 + j * dim2 + k];          // mit k < dim3
```

- Optionales Macro

```
#define Arrayaccess(a, x, y, z) ((a)[(x) * dim1 + (y) * dim2 + (z)])
```

Allokation auf dem Heap VI

- Kontinuierlicher Speicherbereich (eindimensional)
- Freigabe

```
free(arr5);
```

Allokation auf dem Heap VII

- Mittels Pointer auf (multidimensionales) Array
- Allokation
- Zugriff

Allokation auf dem Heap VIII

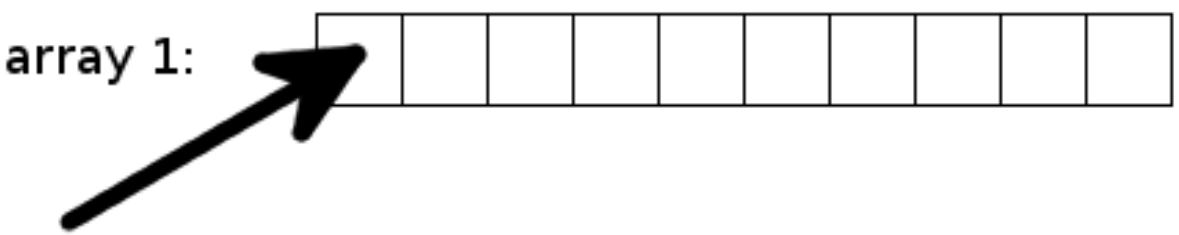
- Mittels Pointer auf (multidimensionales) Array
- Freigabe

```
free(arr6);
```

Indizierung multidimensionaler Arrays

- Warum funktioniert sowohl `arr[i][j]` als auch `ptr[i][j]`?

- Array-Decay to the rescue!
  - Jede Dimension zerfällt
  - Implizite Pointer ermöglichen korrekten Zugriff



Indizierung multidimensionaler Arrays

- Betrachten wir nun folgendes Array `char arr[3][5]`
- Was passiert bei der Auswertung von `arr[i][j]`?
  - Typ von `arr` ist `char (*)[3][5]`
  - Typ von `arr` im Kontext mit '[]' ist `char (*)[5]`
  - $\Rightarrow$ Offset zum Zielarray: `5*i`
  - Typ von `arr[i]` ist `char (*)[5]`
  - Typ von `arr[i]` im Kontext mit '[]' ist `char*`
  - $\Rightarrow$ Offset zum Zielwert im Zielarray: `1*j`
- $\Rightarrow$ Element ist an der Adresse "`&arr[0][0] + 5*i + 1*j`"

Multidimensionale Arrays als Funktionsparameter

- Analog zu eindimensionalen Arrays

```
/* multidimensionale Arrays */
void multi_array_function(int multi_values[5][5])
{
    printf("%d\n", sizeof(multi_values));
        // Erhofft: 20? Ausgabe: 8
}
/* Kompilieren ergibt */
void multi_array_function(int (*multi_values)[5])..

void multi_array_function(int multi_values[5][5])
{
    printf("%d\n", sizeof(*multi_values));
        // Erwartet: 20 Ausgabe: 20
}
```

Korrektes Casten von einfachen Pointern

- Gegeben sei der folgende Prototyp aus einer Bibliothek
- `int* lib_function_returning_simple_ptr(/*params */);`

```
/* Wie kann der Ergebnispointer korrekt gecastet werden? */
int width, height;
int *result = lib_function_returning_simple_ptr(/* params */);
int (*array)[width];
array = (int(*)[width])result;

/* Oder kompakter.. */
int width, height;
int (*array)[width] =
    (int(*)[width])lib_function_returning_simple_ptr(/* params */);
```

- Viele weitere (aber unsaubere Möglichkeiten)

Anwendungsgebiete

- Modellierung von bereits existierenden Strukturen
  - Matrizen
  - Allgemeine Rasterstrukturen (Verteilungen, Messwertreihen)
- Bei performancekritischen Anwendungen
  - Rohdaten in koninuierlichem Datenblock
  - Adressierung über multidimensionales Array
  - $\Rightarrow$ Vorteile der Adressierung kombiniert mit Performanz

Auszug aus partdiff-seq.c (Hochleistungsrechnen)

```c
/* als globale Variablen l.45f */
double ***Matrix;     // index matrix used for addressing M
double *M;            // two matrices with real values

/* aus allocateMatrices() l.78ff */
Matrix = (double ***) calloc (2, sizeof (double **));
/* allocate index matrix */
Matrix[0] = (double **) calloc ((N + 1), sizeof (double *));
Matrix[1] = (double **) calloc ((N + 1), sizeof (double *));
/* allocate actual matrices */
M = malloc (sizeof (double) * (N + 1) * (N + 1) * 2);
/* correctly set pointer in index matrix*/
for (i = 0; i <= 1; i++)
  for (j = 0; j <= N; j++)
    Matrix[i][j] = (double *)
        (M + (i * (N + 1) * (N + 1)) + (j * (N + 1)));
```

Auszug aus partdiff-seq.c (Hochleistungsrechnen) II

```
/* aus calculate() l.230 */
star = -Matrix[m2][i - 1][j] - Matrix[m2][j - 1][i] + 4 *
  Matrix[m2][i][j] - Matrix[m2][i][j + 1] - Matrix[m2][i + 1][j];

/* aus freeMatrices() l.165ff */
if (Matrix[1] != 0)
  free (Matrix[1]);
if (Matrix[0] != 0)
  free (Matrix[0]);
free (Matrix);
free (M);
```

Performanzvergleich

- Echte multidimensionale Arrays vs. Pointerarrays
  - 2 Dimensionen a 1000
  - 100 000 Iterationen a 2 Durchläufe aller "Zellen"
  - Mittelwert aus Summe aller Iterationen
- Zeitmessung mittels <time.h> / clock()

```
/* Testschleife in main() */
for (int i = 0; i < ITERATIONS; ++i)
{
        total_variable += test_funktion();
}
average_variable = total_variable / (double)ITERATIONS;
```

Performanzvergleich II

- `int[][]`: 5,95ms/Iteration vs. `int**`: 6,25ms/Iteration

```
/* aus test_funktion() */
start = clock();
for (int i = 0; i < ARRAY_SIZE; ++i)
{
        for (int j = 0; j < ARRAY_SIZE; ++j)
        {
                data[i][j] = (i + j);
        }
}
for (int i = 0; i < ARRAY_SIZE; ++i)
{
        for (int j = 0; j < ARRAY_SIZE; ++j)
        {
                data[i][j] = (i - j);
        }
}
end = clock();
```

Fazit - Multidimensionale Arrays in C

- Typen
  - Array von Arrays
  - Pointer auf Arrays/Pointer
  - Mischformen
- Besonderheiten
  - **Pointer-Decay**
  - eigenwillige Cast-Syntax
- Allokation
  - statisch
  - dynamisch
- Performanz
  - Leichter Vorsprung von [][] vs. **

Referenzen I


Summit, Steve. *comp.lang.c FAQ list - Question 6.8*. Nov. 5, 2013. URL: http://c-faq.com/aryptr/practdiff.html.


– .*comp.lang.c FAQ list - 6. Arrays and Pointers*. Nov. 5, 2013. URL: http://c-faq.com/aryptr/index.html.


StackOverflow.com. *In C arrays why is this true? a[5] == 5[a]*. Nov. 5, 2013. URL: http://stackoverflow.com/questions/381542/in-c-arrays-why-is-this-true-a5-5a.


– .*Are a, &a, *a, a[0], &a[0] and &a[0][0] identical pointers?* Nov. 5, 2013. URL: http://stackoverflow.com/questions/18361111/are-a-a-a-a0-a0-and-a00-identical-pointers.

Referenzen II


StackOverflow.com. *what is array-decaying*. Nov. 5, 2013. URL: http://stackoverflow.com/questions/1461432/what-is-array-decaying.


Hosey, Peter. *Everything you need to know about pointers in C*. Nov. 5, 2013. URL: http://boredzo.org/pointers/.