Multidimensionale Arrays in C

— Seminar *Effiziente Programmierung in C* —

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

Vorgelegt von:	Florian Wilkens
E-Mail-Adresse:	1wilkens@informatik.uni-hamburg.de
Matrikelnummer:	6324030
Studiengang:	Software-System-Entwicklung

Betreuer: Nathanael Hübbe

Hamburg, den 31. März 2014

Vorwort

Dies ist eine ergänzende Ausarbeitung zum Thema *Multidimensionale Arrays in C* im Rahmen des Seminars *Effiziente Programmierung in C* an dem ich im Wintersemester 2013/14 teilgenommen habe.

Weitere Informationen finden sich unter wr.informatik.uni-hamburg.de

Inhaltsverzeichnis

1 Einleitung

Im Folgenden werde ich mich mit multidimensionalen (oder mehrdimensionalen) Arrays in C befassen. Dabei beginne ich mit den Grundlagen und Eigenheiten eindimensionaler Arrays, um die komplexeren Konzepte später in Bezug setzen zu können. Es folgt eine kurze Beschreibung von möglichen Anwendungsgebieten und ein kleiner Performanzvergleich der zwei wesentlichen Arten multidimensionaler Arrays. Den Abschluss bildet ein kurzes Fazit, dass die wichigsten Punkte noch einmal aufgreift.

Vorrausgesetzt wird syntaktische Kenntnis der Programmiersprache C, um die erläuternden Listings verstehen zu können und grundlegendes Wissen über den Indizierungsoperator "[]".

2 Eindimensionale Arrays

Bevor die Implementationsdetails von Arrays in C betrachtet werden können, lege ich mich hier auf einen Array-Begriff fest, um spätere Unklarheiten zu vermeiden.

"An array is a single, preallocated chunk of contiguous elements (all of the same type), fixed in size and location." [6]

Dabei ist insbesondere in C wichtig, dass *alle* Elemente eines Arrays vom gleichen Typ sein müssen. Ungleich in anderen (meist höheren) Programmiersprachen ist dies absolut notwendig, da in C keine Typinformationen zu einem erstellten Array abgelegt werden. Die Typisierung des verwendeten Speichers erfolgt ausschließlich über den verwendeten Zeigertyp und die daraus resultierende Zeigerarithmetik.

Weiterhin werden Arrays in C als sogenannte "second-class citizen" bezeichnet. Dies bedeutet, dass nicht alle modifizierenden Operationen, die die Programmiersprache anbietet, auf ihnen angewendet werden können. Insbesondere ist zu beachten, dass Arrayvariablen nach ihrer Zuweisung nicht neu belegt werden können. Elemente des Arrays können natürlich ohne Probleme verändert oder ersetzt werden, aber eine komplette Neuzuweisung eines anderen Arrays an eine bereits initialisierte Arrayvariable ist (in C) nicht möglich.

2.1 array-decay

Die Benachteiligung von Arrays gegenüber "first-class ciziten" (beispielsweise primitiven Typen) wird jedoch an einer weiteren wesentlichen Stelle deutlich. Wird ein Array in einem Wertausdruck[1] verwendet, zerfällt der *Wert* des Arrays zu einem Zeiger auf sein erstes Element (engl. *array-decay*).

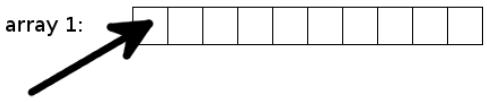

Abbildung 2.1: array-decay

In Abbildung 2.1 wird dies verdeutlicht. Ein Beispielarray ist an eine Variable mit dem Bezeichner `array1` gebunden. Der *tatsächliche* Wert dieser Variable ist also das komplette Array, also der zusammenhängende Speicherblock. Der schwarze Pfeil liefert nun den Wert der Variable im Kontext eines Werteausdrucks. Es ist leicht erkennbar, dass der symbolisierte Zeiger nur auf das erste Elements des Speicherblocks (und damit des Arrays) verweist. Es wird jedoch weder der *tatsächliche* Wert von `array1` verändert, noch eine weitere Zeigervariable mit einem neuen Wert instanziert. Der Wert der Arrayvariable wird lediglich im Kontext eines Wertausdrucks anders interpretiert. Er zerfällt von einem Array auf ein Elementzeiger (mit dem Typ der Arrayelemente), daher der Name *array-decay*.

Tatsächlich handelt es sich bei nahezu allen alltäglichen Verwendungen eines Arrays um Werteausdrücke. Nach dem Standard C99 gibt es nur drei Ausnahmen, in denen ein Array *nicht* zu einem Elementzeiger zerfällt:

- der &-Operator,
- die `sizeof()` Funktion und
- die Zuweisung von string-Literalen.

Diese Ausnahmen lassen sich relativ einfach erklären. Das Ziel des &-Operators ist es ja gerade die Adresse eines Wertes im Speicher zu erhalten. Würde in diesem Fall ein betrachtetes Array zu einem Zeiger auf sein erstes Element zerfallen, würde die Adresse dieses (temporären) Zeigers zurückgeliefert werden, anstatt der Adresse des tatsächlichen Arrays. Ähnlich verhält es sich bei der `sizeof()` Funktion. Würde hier array-decay stattfinden, würde die Funktion die Größe dieses Zeigers anstatt der (gewünschten) Größe des Arrays ermitteln.

[1] engl. *value expression* Verwendung in Zuweisungen, Zugriff auf einzelne Elemente etc.

Der dritte Fall wird an folgender Codezeile deutlich:

```
char arr[] = "Hello world!";
```

Es handelt sich hier um eine einfache Zuweisung eines string-Literals an eine neu deklarierte Array-Variable *arr*. Was auf den ersten Blick eventuell nicht sofort klar ist - auch bei dem Literal `"Hello World"` handelt es sich um ein Array. Würde dieses hier zu einem Zeiger zerfallen, müsste bei der Übersetzung unnötig komplizierter und teurer Maschinencode generiert werden.[2] Das Literal bleibt daher in diesem Fall ein Array und kann direkt der neu deklarierten Variable *arr* zugewiesen werden.

Es ist jedoch bemerkenswert, wie kurz diese Liste der Ausnahmen ist. In allen anderen Fällen (in C99) findet array-decay statt. Es ist daher als Programmierer unerlässlich dieses Konzept vestanden zu haben, da bei naiver Verwendung in nahezu allen Anwendungsfällen unerwarteten Ergebnissen auftreten können. Ein besonders wichtiger Fall wird im nächsten Abschnitt noch einmal genauer betrachtet.

2.2 Indexierter Zugriff

Nach dem Lesen des letzten Abschnitts mags sich die Frage gestellt haben, ob der Indizierungsoperator '[ ]' vergessen wurde. Schließlich ist Arrayindizierung der wohl meistverwendete Kontext von Arrays im Programmablauf. Das Fehlen des Operators ist jedoch kein Fehler, sondern beabsichtigt, denn der Inidizierungsoperator arbeitet nicht (wie oft vermutet) auf Arrays, sondern auf einfachen Zeigern. Dies können entweder ganz "normale" Zeiger oder eben Arrays sein, die während der Indizierung zu einem Zeiger auf ihr erstes Element zerfallen.

Betrachten wir beispielsweise folgendes Array `int arr[5];` und einen Zugriff auf das dritte Element mittels `arr[2]`. Der Typ von `arr` ist nun wie deklariert ersteinmal `int ()[5]`. Beim Übersetzen wird nun der Indizierungsoperator in einen zeigerarithmetischen Ausdruck umgewandelt. Aus `arr[2]` wird `*(arr + 2)`. Betrachten wir nun den Typ von `arr` im Kontext von '+', so erhalten wir `int*`, denn die zeigerarithmetische Addition ist ein Werteausdruck. Folglich zerfällt das Array auf einen Zeiger auf sein erstes Element, das natürlich vom Typ `int` ist. Abschließend wird der Ausdruck normal evaluiert. Zum Zeiger `arr` wird `2 * sizeof(int)` addiert, um die Speicheradresse des gesuchten Elements zu erhalten. Dieses liegt also an der Position "`&arr[0] + 2*4`"[3][4]

[2] Es müsste vom Zeiger ausgehend durch den Speicher iteriert werden, bis der 0-Terminator erreicht wird

[3] Basierend auf einer Integergröße von vier Byte

[4] &arr[0] meint hier lediglich den zerfallenen Pointer und nicht die tatsächliche Anwendung des &-Operators. Der Ausdruck ist logisch nicht korrekt und dient nur zur Veranschaulichung

2.3 Arrays als Funktionsparameter

Eine weitere Besonderheit bildet den Abschluss dieses einleitenden Kapitels. Array-Zerfall tritt auch bei der Verwendung von Arrays als Funktionsparametern ein.
Betrachten wir folgende Funktionsdefinition:

Listing 2.1: Eindimensionale Arrays als Funktionsparameter

```
1 void array_function(int values[5])
2 {
3     printf("%d\n", sizeof(values));
4         // Erhofft: 20 Ausgabe: 8
5 }
```

Wie annotiert ist die Ausgabe `8`, obwohl der Typ des Arguments korrekt als Array spezifiziert wurde. Array-Zerfall kann es aber auch nicht sein, wurde doch oben beschrieben, dass `sizeof()` eine der wenigen Ausnahmen ist. Wie kommt es also zu der unerwarteten Ausgabe?

Tatsächlich handelt es sich zu dem Zeitpunkt der Anwendung von *sizeof* bei `values` nicht um ein Array. Folglich kann auch keine Ausnahme in Kraft treten und die korrekte Größe ermitteln. Aus der oberen Funktionsdefinition erzeugt der Compiler (vor dem Übersetzen in Maschinencode) eine Funktion mit folgender Signatur.
`void array_funtion(int*);`

Der als `int ()[]` deklarierter Parameter zerfällt also (bereits während seiner Deklaration als Funktionsargument) zu einem `int`-Zeiger. Folglich liefert `sizeof(values)` (korrekt) `8` Byte als Zeigergröße. [5] Daraus folgt für den Programmierer, dass bei Verwendung von Arrays nahezu immer ein zweiter Größenparameter erforderlich ist, um Pufferüberläufe zu verhindern. Weiterhin bedeutet dieser Sonderfall aber auch, dass auf der Seite des Aufrufers keine Garantie für erforderliche Arraydimensionen übernommen werden kann. Folgender Code stellt also einen (nach C-Standard) validen Aufruf unserer oben definierten Funktion dar.

Listing 2.2: Probleme der Aufruferseite

```
1 int values[3] = { 1, 2, 3 };
2 array_function(values);  // syntaktisch valider Aufruf
```

Natürlich fehlt hier immer noch der empfohlene Größenparameter, aber das Problem wird deutlich. Da der Typ des Parameters auf einen Zeiger reduziert wird, kann ein Aufrufer ein beliebig großes Array (das dann zu einem Zeiger zerfällt) oder gar einen einfachen Zeiger des gewünschten Typs verwenden. Es kann somit über die Parameterdefinition keinerlei Einschränkung bezüglich erwarteten Arraygrößen übernommen werden, sodass diese auf anderen Wegen festgelegt werden müssen (etwa über weitere Parameter, globale Variablen, Konstanten etc.).

[5] Auf meiner Testmaschine unter ArchLinux x64

3 Multidimensionale Arrays

Nach diesem einleitenden Kapitel geht es also nun um das eigentliche Thema. Multidimensionale Arrays gibt es (in C) in zwei wesentlichen Varianten.

- "Arrays von Arrays" `char array1[3][5]`
- "Zeiger auf Zeiger" `char **array2`

Beide dargestellten Variablendefinitionen stellen hier Arrays mit gleichen Dimensionen dar.[1] Diese Variablen können nun ersteinmal (dank Array-Zerfall) identisch verwendet werden. Wie unterschiedlich die Position der Elemente im Speicher jedoch ist, verdeutlicht folgende Grafik.

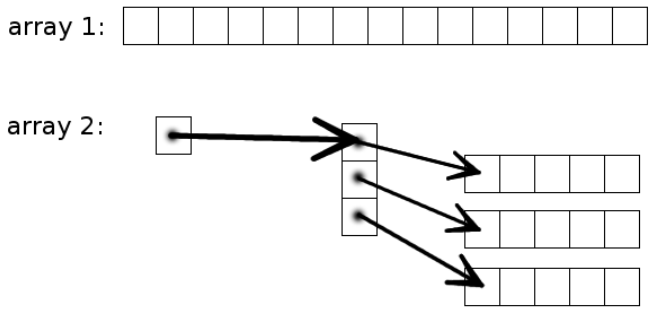

Abbildung 3.1: Speicherbelegung multidimensionaler Arrays

Es ist deutlich zu erkennen, dass `array1` einen einzelnen zusammenhängenden Speicherblock belegt, während `array2` auf mehrere über Zeiger verbundene Speicherblöcke verteilt ist. Nach der Definition aus der Einleitung ist also nur `array1` tatsächlich *ein* Array, während `array2` eher eine Kombination aus Arrays und Zeigern ist, die sich (dank der Funktionsweise des Indizierungsoerators) wie ein Array verwenden lässt. In der Praxis werden beide Grundarten sowie Mischformen verwendet, wobei das Anwendungsgebiet meist den Typ bestimmt. So werden auf dem Stack meist 'echte' multidimensionale Arrays verwendet, während auf dem Heap Variationen auf Zeigerbasis (geboren aus der Notwendigkeit der Verwendung von `c/malloc`) vorherrschen. Ich werde im Folgenden auf die verschiedenen Möglichkeiten der Allokation beider Arraytyypen eingehen und Vor- und Nachteile aufzeigen.

[1] Natürlich ist der Zeiger auf einen Zeiger in `array2` nicht auf ein Array dieser Dimensionen oder überhaupt ein Array beschränkt.

3.1 .. auf dem Stack

Die Verwendung von multidimensionalen Arrays auf dem Stack beschränkt sich auf kontinuerliche Arrays mit konstanten Größen. Die Allokation ist daher wie erwartet recht simpel und intutiv.

Listing 3.1: Allokation auf dem Stack

```
1 /* mit statischen Dimensionen -> Arrays von Arrays */
2 char arr[2][2][2];
3
4 /* mit dynamischen Dimensionen (seit C99) -> Arrays von
    ↪ Arrays */
5 char arr2[n][n][n];
```

Eine kleine Besonderheit bildet die zweite Variante. Die Allokation eines Stackarrays mit Dimensionen, die erst zur Laufzeit bestimmt werden können, ist in vielen neueren Sprachen selbstverständlich, war jedoch in C vor C99 nicht möglich. Bei der Verwendung dieser Methode ist also darauf zu achten, dass das Compilerflag für den entsprechenden C-Standard gesetzt ist.

3.2 .. auf dem Heap

Auf dem Heap hat der findige C-Programmierer eine große Auswahl an Arraytypen und damit auch an Allokationsarten. Zuerst betrachtet wird hier die traditionelle Variante mittels `c/malloc`.

Listing 3.2: Allokation auf dem Heap: Variante 1

```
1 int dim1 = 2, dim2 = 2, dim3 = 2;
2 char ***arr3 = calloc(dim1, sizeof(*arr3));
3 for(i = 0; i < dim1; i++)
4 {
5     arr3[i] = calloc(dim2, sizeof(**arr3));
6     for(j = 0; j < dim2; j++)
7         arr3[i][j] = calloc(dim3, sizeof(***arr3));
8 }
```

Diese Art der Allokation ist trotz ihrer Länge recht trivial. Die ersten “n-1” Dimensionen des Zielarrays müssen in einer Schleife mihilfe eines `c/malloc`-Calls allokiert werden. Es ist leicht zu erkennen, dass die Komplexität mit steigender Dimensionszahl rapide zunimmt. Obwohl die einzelnen Zeilen klar zu verstehen sind, verliert man bei steigender Schleifenanzahl schnell den Überblick. Dies macht die Allokationsart gerade für Copy&Paste-Fehler anfällig, wenn beispielsweise eine Schleifenvariable falsch übernommen wird.

Die Freigabe läuft quasi gespiegelt ab. Nachdem alle Elemente einer Dimension freigegeben wurden, kann das dazugehörige Element der übergeordneten Ebene freigegeben werden.

Listing 3.3: Freigabe auf dem Heap: Variante 1

```
1 for (int i = 0; i < dim2; ++i)
2 {
3     for (int j = 0; i < dim1; ++j)
4         free(arr3[i][j]);
5     free(arr3[i]);
6 }
7 free(arr3);
```

Der traditionelle Ansatz hat einen Nachteil, der jedoch nicht auf den ersten Blick ersichtlich ist. Da die Allokation der eigentlichen Datenblöcke über mehrere Funktionsaufrufe verteilt ist, sind diese natürlich nicht garantiert zusammenhängend.[2] Dies ist ersteinmal kein direktes Problem, kann aber unter Umständen die Performanz beeinträchtigen (erhöhte Anzahl von *cache misses*).
Hier kommt die zweite vorgestellte Variante ins Spiel. Die Idee ist die Größe des gewünschten Datenblocks aus den gegebenen Dimensionen auszurechnen und in einem Aufruf anzufordern. Danach wird mithilfe von etwas Zeigerarithmetik eine bekannte Arraystruktur aufgesetzt, um auf den reservierten (und hoffentlich zusammenhängenden) Datenblock wie gewohnt zugreifen zu können.

Listing 3.4: Allokation auf dem Heap: Variante 2

```
1  int dim1 = 2, dim2 = 2, dim3 = 2;
2  char ***arr4 = calloc(dim1, sizeof(*arr4));
3  for(i = 0; i < dim1; i++)
4  {
5      arr4[i] = calloc(dim2, sizeof(**arr4));
6  }
7  arr4[0][0] = calloc(dim1 * dim2 * dim3, sizeof(***arr4));
8  for(i = 0; i < dim1; i++)
9  {
10     for (int j = 0; j < dim2; j++)
11         arr4[i][j] = arr4[0][0] + (i * dim1) + (j * dim2);
12 }
```

Nun wird die Allokation schon deutlich komplexer. Die ersten 6 Zeilen legen die ersten beiden Dimensionen an. Dies ist hier notwendig, da der Datenblock an der Stelle des ersten Datenelement beginnen soll. Da es sich im Beispiel um ein dreidimensionales Array handeln soll, müssen wir also die ersten beiden Dimensionen anlegen. Zeile 7 ist nun

[2] Dies ist natürlich streng genommen auch bei "normalen" Arrays nicht der Fall, da virtuelle Adressen verwendet werden. Bei typischen Anwendungsfällen passen die allokierten Arrays jedoch auf eine Speicherseite, sodass häufig (auch physikalisch) zusammenhängende Blöcke vergeben werden.

der Funktionsaufruf, der den gesamten Speicher für die verwendeten Daten reserviert. Die Größe wird hiebei einfach über das Produkt der Dimensionen und der Größe des Datentyps bestimmt.

Die Notation `***arr4` wird hierbei (und auch bereits im vorherigen und in den folgenden Beispielen) verwendet, um vom gewählten Datentyp abstrahieren zu können. Da `arr4` bereits als dreifacher Zeiger hier auf einen `char` definiert ist, ergibt sich `sizeof(***arr4)== sizeof(char)`. Der Ausdruck mithilfe der Variable erleichtert jedoch den späteren Wechsel des Datentyps. Soll aus dem `char`-Array später ein `int`-Array werden, muss lediglich der Datentyp von `arr4` an einer Stelle (nämlich der Deklaration) verändert werden.

Die Zeilen 8 bis 12 "biegen" nun die einzelnen Elemente des Array korrekt auf den erstellten Speicherbereich um, da ja die Elemente der bereits allokierten ersten Dimension auf diesen zeigen müssen. Die geschweiften Klammern wurden nur aus Platzgründen entfernt, denn die interessante Zeile ist hier #11. Hier werden die einzelnen Zeiger des vorher initialisierten Arrays auf den Speicherbereich gedeutet. Nach diesem Codeblock kann nun `arr4` wie ein gewohntes dreidimensionales Array verwendet werden.

Die Freigabe läuft hier dafür nun etwas simpler ab. Da die letzte Dimension implizit über normale Indizierung des Speicherblocks, wurde eine Dimension weniger direkt über `c/malloc` allokiert. Es muss daher nur der Datenblock und die andere Dimension sowie die Variable freigegeben werden und zwar natürlich wie bekannt in umgekehrter Reservierungsreihenfolge.

Listing 3.5: Freigabe auf dem Heap: Variante 2

```
1 free(arr4[0][0]);
2 for (int i = dim1; i > 0; --i)
3 {
4     free(arr4[i]);
5 }
6 free(arr4);
```

Es verbleibt jedoch weiterhin das Problem, dass der Allokationsaufwand immer noch mit der Dimensionsanzahl skaliert. Die letzten beiden Variationen vermeiden dies auf unterschiedlichen Wegen. Einerseits bleibt natürlich die Möglichkeit ein eindimensionales Array wie ein multidimentionales zu verwenden. Dabei bleibt natürlich auch der Vorteil erhalten, dass weiterhin ein kontinuierlicher (virtueller) Speicherblock verwendet wird und es wird der Overhead für zusätzliche Zeigerarrays vermieden. Die Allokierung ist wie erwartet recht unspektakulär, es wird lediglich die Größe wie im vorherigen Beispiel als Produkt der Dimensionen und der Elementgröße festgelegt.

Listing 3.6: Allokation auf dem Heap: Variante 3

```
1 int dim1 = 2, dim2 = 2, dim3 = 2;
2 char *arr5 = calloc(dim1 * dim2 * dim3, sizeof(*arr5));
3
4 char c = arr5[i * dim1 + j * dim2 + k];      // mit k < dim3
```

Natürlich schlägt sich die Verwendung eines eindimensionalen Arrays im Zugriff wieder. Die Adressierung der Dimensionen muss manuell vorgenommen werden, wobei natürlich deren Größen bekannt sein müssen. Um diesen Prozess zu vereinfachen, kann es sich anbieten, ein Makro für den Variablenzugriff in der folgenden Art zu definieren.

Listing 3.7: Allokation auf dem Heap: Variante 3 - Zugriffsmakro

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

Dies bringt aber weitere Schwierigkeiten mit sich, denn das Makro ist natürlich von den Arraydimensionen abhängig. Es muss also für jedes Array mit neuen Dimensionen ein neues Makro angelegt werden. Weiterhin ist der Zugriff nicht gerade intuitiv und gerade bei verschiedenen Makros für verschiedene Arrays kann dies neue Programmierer im Projekt verwirren. Es ist daher nicht empfehlenswert, hüfig verwendete Daten eindimensionale Arrays abzulegen. Für kurze temporäre Speicher ist dies zwar noch vertretbar, allerdings kann in diesem Fall meist ein "echtes" multidimensionales Stack-Array verwendet werden. Immerhin ist aber die Freigabe wieder recht trivial. Da nur ein einziges (endimensionales) Array verwendet wurde, reicht ein einzelner Aufruf der `free`-Funktion, um den allokierten Speicher wieder freizugeben.

Listing 3.8: Freigabe auf dem Heap: Variante 3

```
1 free(arr5);
```

Die letzte Methode vereinigt nun die Vorteile des multidimensionalen Stackarrays mit der (relativen) Größenunabhängigkeit des Heaps. Der einzige Grund, der es bisher vermied, multidimensionale Arrays auf dem Stack anzulegen, war die Bindung an `c/malloc` und damit an den simplen Zeiger als Rückgabetyp. Dies lässt sich aber mit gekonnter Verwendung des C-Typsystems umgehen. Das folgende Beispiel allokiert nun also ein echtes, multidimensionales Array auf dem Heap und zwar als einen Zeiger.

Listing 3.9: Allokation auf dem Heap: Variante 4

```
1 char (*arr6)[dim1][dim2][dim3] = calloc(1, sizeof(*arr6));
2
3 char c = (*arr6)[i][j][k];
```

In einer Zeile ist das Array also vollständig allokiert und auch der Elementzugriff sieht nicht allzu fremd aus. Der größte Vorteil dieser Allokationsart besteht eigentlich darin, dass die gesamten Typinformationen wie Dimension und Elementtyp komplett in der Variablendeklaration von `arr6` stehen. Beim Aufruf von *c/malloc* wiederum ist nun

der Compiler in der Lage die Größe des Typs dieser Variable zu ermitteln und nimmt dem Programmierer so einen Großteil der Arbeit ab. Es reicht daher, wenn als erster Parameter `1` übergeben wird. Beim Zugriff ist nun lediglich darauf zu achten, dass es sich um ein Zeiger auf ein multidimensionales Array und nicht um dieses selbst handelt. Bevor also die bekannte Indizierung erfolgen kann, muss der Zeiger folglich erst dereferenziert werden.
Die Freigabe verhält sich nun ähnlich simpel wie im vorherigen Beispiel. Da wieder nur ein einziger Funktionsaufruf stattgefunden hat, muss auch diese Variable lediglich mit einem einzigen `free`-Call freigegeben werden.

Listing 3.10: Freigabe auf dem Heap: Variante 3

```
1 free(arr5);
```

Aufgrund der diversen Vorteile ist diese Methode daher meine Empfehlung für die Allokation von multidimensionalen Arrays auf dem Heap. Der (Zeilen-)Aufwand skaliert nicht mit der Dimensionsanzahl, die Freigabe ist überraschend einfach und auch der Zugriff ähnelt dem bekannten verfahren. Es sollte aber in jedem Fall abgewogen werden, ob die Nutzung eines Heap-Arrays erforderlich beziehungsweise gerechtfertigt ist. Für geringe Datenmengen sollte (gerade aufgrund der Simplizität) stets der Stack bevorzugt werden.

3.3 Funktionsweise

Nachdem nun die verschiedene Möglichkeiten zur Allokation hoffentlich etwas klarer geworden sind, geht es in diesesm Abschnitt noch einmal um die technischen Hintergründe beim Zugriff auf Elemente der höheren Dimensionen. Analog zu dem Beispiel für eindimensionale Arrays, möchte ich an dieser Stelle noch einmal erläutern, wie die Indizierung mit steigender Dimensionszahl funktioniert und wie dabei **Array-Zerfall** dabei ein weiteres Mal eine wichtige Rolle spielt.

Wie vorher werde ich den Vorgang der Indizierung anhand eines Beispiels Schritt für Schritt erläutern. Wie betrachten daher folgendes Array: *chararr*[3][5]. Was geschieht nun also bei der Auswerung von `arr[i][j]`? Zuerst wird wieder der Typ des Ausdrucks betrachtet. Der Typ von `arr` ist wie erwartet `char (*)[3][5]`. Im Kontext mit "[]" (also einer zeigerarithmetischen Addition) zerfällt der Typ der Variable jedoch zu `char (*)[5]`, also zu einem Zeiger auf `char`-Arrays der Größe 5.[3] Tritt man kurz einen Schritt zurück, wird die Bedeutung dessen schnell klar. Unser Ausgangsarray `arr` besitzt die Dimensionen "`[3][5]`". Um also durch die erste Dimension zu iterieren, müssen in jedem Schritt 5 Elemente "übersprungen" werden. Genau dies wird mit dem Typ des zerfallenen Arrays (`char (*)[5]`) erreicht.
Um also ersteinmal das korrekte Array der zweiten Dimension zu erhalten, muss ein Offset von `i*5*sizeof(char)` zum Anfangspunkt des gesamten Arrays angewendet werden.

[3] Zahlen sind in diesem Abschnitt entgegen Konventionen der deutschen Rechtschreibung als Literale verwendet, um die Verständlichkeit zu erhöhen.

Da wir das Zielarray erreicht haben, betrachten wir den Typ des Ausdrucks `arr[i]`. Dieser ist nun wie erwartet `char ()[5]`, wieder im Kontext mit dem zweiten Indizierungsoperator ergibt sich also `char*`. Von hier aus verläuft alles wie bei dem Beispiel des eindimensionalen Arrays und es ergibt sich ein Offset zum Zielwert im Zielarray von `j*sizeof(char)`.
Das gesuchte Element befindet sich also an der Stelle "`&arr[0][0] + i*5 + j*1`".[4]

3.4 .. als Funktionsparameter

Wie auch schon im Kapitel zu eindimensionalen Arrays geht es hier nocheinmal um Array-Zerfall in Funktionsdefinitionen. Es wurde bereits erläutert, wann dieser Zerfall einsetzt und welche Auswirkungen dies auf die Programmlogik haben kann. Im Prinzip ändert sich nicht an diese Verfahren, es ist nur etwas trickreicher, wenn es um mehrere Dimensionen geht. Beispielhaft wurde hier eine Funktion definiert, die ein multidimensionales Array als Argument erwartet.

Listing 3.11: Funktion mit multidimensionalem Argument

```
1 /* multidimensionale Arrays */
2 void multi_array_function(int multi_values[5][5])
3 {
4     printf("%d\n", sizeof(multi_values));
5         // Erhofft: 20? Ausgabe: 8
6 }
```

Mit dem Vorwissen aus dem Kapitel über eindimensionale Arrays wurde hier angenommen, dass 20 Byte als Größe des Arguments geliefert wird. Dies sollte sich aus der Größe der zweiten Dimension (5 Byte) und der Größe eines Integer (4 Byte) ergeben, da ja die erste Ebene zerfällt. Leider hat der Programmierer übersehen, dass die erste Ebene zwar zerfällt, aber eben zu einem Zeiger und nicht einfach zu einem eindimensionalen Array. Vor dem Übersetzen in Maschinencode erzeugt der Compiler die folgende Signatur für die gezeigt Beispielfunktion:

Listing 3.12: Compilergenerierte Signatur

```
1 /* Kompilieren ergibt */
2 void multi_array_function(int (*multi_values)[5])..
```

Es ist zu sehen, dass das Argument zu einem Zeiger auf ein `int`-Array der Größe fünf zerfällt und damit natürlich die Größe acht Byte liefert. Es ist übrigens von großer Wichtigkeit, dass die Größe der zweiten Dimension erhalten bleibt und nicht gar ein Zerfall zu einem einfachen Zeiger eintritt, denn bei der Indizierung muss diese Größenangabe verwendet werden, um korrekte Zeigerarithmetik gewährleisten zu können. Eine Iteration

[4] Auch hier steht &arr[0][0] wie im vorherigen Beispiel lediglich informell für die Startadresse des Arrays im Speicher.

durch das übergebene Array wäre damit unmöglich gemacht.

Listing 3.13: Derefenrezieren des Argumentzeigers

```
1 void multi_array_function(int multi_values[5][5])
2 {
3     printf("%d\n", sizeof(*multi_values));
4         // Erwartet: 20 Ausgabe: 20
5 }
```

Natürlich kann der zerfallene Zeiger jedoch dereferenziert werden und es wird die korrekte erwartete Größe von 20 Byte zurückgeliefert. Außerdem bedeutet der Zerfall natürlich, dass auch hier Größeninformationen über die erste Ebene verloren gehen. Daher ist analog zu eindimensionalen Arrays die Angabe der ersten Dimension keine Garantie und eigene Funktionen sollten auf zusätzliche Größenparameter zurückgreifen.

3.5 Exkurs: Korrektes Casten von einfachen Zeigern

Ein kleiner Exkurs zum Thema korrektes Casten von Zeigern bildet den Abschluss dieses Kapitels. Als Programmierer hat man oft keinen Einfluss auf die Rückgabetypen aller Funktionen, die aufgerufen werden. Gerade im Falle von Bibliotheken, die nur binär vorliegen, müssen andere Wege gefunden werden, korrekte Typen zu verwenden. Im Folgenden geht es um eine fiktive Bibliotheksfunktion `lib_function_returning_simple_pointer(..)` die einen einfachen `int`-Zeiger zurückliefert, der jedoch auf ein mehrdimensionales Array mit den Dimensionen `height` und `width` zeigt. Es stellt sich die Frage, wie der Rückgabewert korrekt in seinen semantischen Typ gecastet werden kann.

Listing 3.14: Casten von Bibliotheksfunktionen

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

Mit der richtigen Syntax ist es also leicht möglich den Rückgabewert in einen Zeiger auf ein Array zu casten, um so einen effektiven, semantisch logischen Zugriff auf die Ergebnisdaten zu erhalten. Dieses erste Beispiel ist etwas umständlich, um wirklich die korrekten Typen und den Cast zu zeigen. Natürlich kann die `result`-Variable auch direkt korrekt getypt und gecastet werden.

Listing 3.15: Casten von Bibliotheksfunktionen (kompakt)

```
1 /* Oder kompakter.. */
2 int width, height;
3 int (*array)[width] =
4     (int(*)[width])lib_function_returning_simple_ptr(/*
      ↪ params */);
```

4 Anwendungsgebiete

In diesem Abschnitt geht es nun um potenzielle Anwendungsmöglichkeiten von multidimensionalen Arrays. Da der Fokus dieser Ausarbeitung jedoch auf den technischen Aspekten liegt, ist dieser Teil bewusst kürzer gehalten und soll lediglich einen kurzen Überblick liefern.

Wie die Struktur es schon vermuten lässt, eignen sich multidimensionale Arrays bestens, um Daten zu speichern, die in Reihen oder Rasterstrukturen vorkommen. Das klassische Beispiel dabei sind Matrizen in der Mathematik. Der Datenzugriff ist dann syntaktisch sehr intutiv, da lediglich die Koordinaten der Matrix verwendet werden können. Andere Möglichkeiten wären naturwissenschaftlichen Verteilungen oder Messreichen. Diese können ohne direkten Kontext in einem multidimensionalen Array verarbeitet werden, ohne auf höhere Konzepte wie Klassen zugreifen zu müssen (die in C ohnehin erst implementiert werden müssten). Wichtig ist hier jedoch, dass das verwendete Schema in der Dokumentation festgehalten wird, da die Daten in ihren rohen Typen ohne zusätzliche Informationen verwendet werden.

Gerade bei performanzkritischen Anwendungen ist es jedoch wichtig die korrekte Allokationsvariante zu wählen. Da hier Anwendungsdaten verarbeitet werden, bleibt als Speicherort meist nur der Heap. Häufig muss dabei zwischen Leistung (Verwendung eines kontinuerlichen Datenblock) und Komfortablität (echte multidimensionale Adressierung) entschieden werden. Eine Lösungmöglichkeit habe ich bereits in meiner letzten Allokationsvariante im letzten Kapitel vorgestellt (siehe 3.9). Eine andere Möglichkeit ist die Daten in einem eindimensionalen Array abzulegen und lediglich den Zugriff über ein multidimensionales Array zu ermöglichen, indem die Zeiger korrekt auf diesen Block umgelegt werden. Dies verhindert die andauernenden zeigerarithmetischen Rechnungen wie im der Methode “eindimensionales Array”, bietet jedoch den Vorteil der kontinuierlichen Speicherbelegung.

Ein Anwendungsbeispiel dieser Methode möchte ich hier anhand von Quellcode aus dem Modul Hochleistungsrechnen vorstellen. Es handelt sch dabei um drei Ausschnitte aus einem Programm zur Berechnung von partiellen Differentialgleichungen, genannt *partdiff-seq*. In diesem Programm werden die zwei Matrizen, die zur Berechnung benötigt werden, in zwei kontinuierlichen Blöcken abgelegt (Variable `M`, der Zugriff erfolgt jedoch über einen dreifachen Zeiger, der wie bekannt adressiert wird (`Matrix`).

Listing 4.1: Auszug: *partdiff-seq.c* - Allokation

```
1  /* als globale Variablen 1.45f */
2  double ***Matrix;     // index matrix used for addressing M
3  double *M;            // two matrices with real values
4
5  /* aus allocateMatrices() 1.78ff */
6  Matrix = (double ***) calloc (2, sizeof (double **));
7
8  /* allocate index matrix */
9  Matrix[0] = (double **) calloc ((N + 1), sizeof (double *));
10 Matrix[1] = (double **) calloc ((N + 1), sizeof (double *));
11
12 /* allocate actual matrices */
13 M = malloc (sizeof (double) * (N + 1) * (N + 1) * 2);
14
15 /* correctly set pointer in index matrix*/
16 for (i = 0; i <= 1; i++)
17   for (j = 0; j <= N; j++)
18     Matrix[i][j] = (double *)
19         (M + (i * (N + 1) * (N + 1)) + (j * (N + 1)));
```

Nachdem in den ersten Zeilen normal die Allokation der einzelnen Arrays für Daten und Adressierungsmatrix durchgeführt wird, deuten die beiden Schleifen die gerade allokierten Zeiger aus `Matrix` auf den Datenblock `M`. Hier wird auch nocheinmal deutlich, dass die Rechnungen mit steigender Dimensionsanzahl extrem ausführlich und unpraktisch werden. Hier werden diese jedoch nur an einer Stelle und vor allem *vor* den folgenden Berechnungen durchgeführt. Dies erhöht die Wartbarkeit (Änderungen können zentral durchgeführt werden) und beschleunigt den Zugriff, da die zeigerarithmetischen Rechnungen bereits abgeschlossen sind und die Zeiger der Adressierungsmatrix später nur einfach dereferenziert werden müssen.

Der Zugriff auf die zu berechnenden Matrizen erfolgt nun wie bekannt über die Indizierung von `Matrix`. Dies ist hierbei der einzige Zugriffspunkt, denn `M` wird bis zur Freigabe zum Ende des Programms nicht mehr direkt verwendet.

Listing 4.2: Auszug: *partdiff-seq.c* - Zugriff

```
1 /* aus calculate() 1.230 */
2 star = -Matrix[m2][i - 1][j] - Matrix[m2][j - 1][i] + 4 *
3   Matrix[m2][i][j] - Matrix[m2][i][j + 1] - Matrix[m2][i +
    ↪ 1][j];
```

Der Vollständigkeit halber soll an dieser Stelle noch einmal die Freigabe betrachtet werden. Diese läuft im Groben wie erwartet ab. In umgekehrter Reihenfolge werden Indizierungsmatrixdimensionen, dann die Indizierungsmatrix selbst und dann die Datenmatrix

freigegeben. Eine kleine Beonderheit sind hier die Nullchecks vor der Freigabe. Diese sind überaus nützlich und verhindern eine Freigabe von Null-Zeigern (falls beispielsweise die Allokation fehlgeschlagen ist[1]).

Listing 4.3: Auszug: *partdiff-seq.c* - Freigabe

```
1 /* aus freeMatrices() l.165ff */
2 if (Matrix[1] != 0)
3   free (Matrix[1]);
4 if (Matrix[0] != 0)
5   free (Matrix[0]);
6 free (Matrix);
7 free (M);
```

Zum Abschluss ist hier noch einmal zu sagen, dass alle Codebeispiele gerade aus dem Allokationsteil sehr gekürzt und auf die Konzepte beschränkt sind. Es fehlen also viele allgemein verwendete Sicheheitsmaßnahmen, wie die oben gezeigeten Null-Checks, die natürlich bei der Verwendung dieser Konzepte angewendet werden sollten, um Defekte oder undefiniertes Verhalten auszuschließen.

[1] Dies sollte natürlich schon nach der Allokation geprüft werden

5 Performanzvergleich

Den inhaltlichen Abschluss soll nun ein kleiner Performanzvergleich zwischen den beiden Urtypen multidimensionaler Arrays bilden. Dafür habe ich einen kleinen Benchmark geschrieben, der jeweils ein Array als echte multidimensionales Array auf dem Stack und ein Zeigerarray auf dem Heap anlegt und diese zwei Mal mit 100.000 Iterationen durchläuft. Daraus wird dann die mittlere Iterationsdauer, also die Zeit für das zweifache Durchlaufen des ganzen Arrays errechnet.

Listing 5.1: Performanzvergleich - Iterationsschleife

```
1 /* Testschleife in main() */
2 for (int i = 0; i < ITERATIONS; ++i)
3 {
4     total_variable += test_funktion();
5 }
6 average_variable = total_variable / (double)ITERATIONS;
```

Die Funktion `test_funktion` führt nun also die zwei Iterationen mittels zwei aufeinanderfolgenden `for`-Schleifen durch. Die durchgeführte Rechenoperation ist jeweils verschieden, um eventuelles Caching zu vermeiden und soll lediglich einen Schreibzugriff simulieren.

Listing 5.2: Performanzvergleich - Test-Funktion

```
1  /* aus test_funktion() */
2  start = clock();
3  for (int i = 0; i < ARRAY_SIZE; ++i)
4  {
5      for (int j = 0; j < ARRAY_SIZE; ++j)
6      {
7          data[i][j] = (i + j);
8      }
9  }
10 for (int i = 0; i < ARRAY_SIZE; ++i)
11 {
12     for (int j = 0; j < ARRAY_SIZE; ++j)
13     {
14         data[i][j] = (i - j);
15     }
16 }
17 end = clock();
```

Nach einer Kompilation mit *gcc -O0* um eventuelle Optimierungen zu vermeiden, können nun die gemessenen Zeiten verglichen werden. Das multidimensionale Stackarray liegt mit 5,95 ms pro Iteration leicht vor dem heapallokierten Zeigerarray mit 6,25 ms. Obwohl keine Größenordnungen dazwischen liegen, ist dieses Ergenis historisch interessant.
Traditionell war es lange der Fall, dass multidimensionale Arrays auf dem Heap deutlich schneller waren, da weniger zeigerarithmetischen Rechnungen durchgeführt werden müssen, um die Zielposition im Speicher zu erreichen. Da das dereferenzieren wesentlich schneller war, als die Berechnungen der Dimensionen, ergab sich ein starker Performanzvorteil für Heaparrays. Heutzutage sind Prozessoren und auch Compiler jedoch soweit vorangeschritten, dass der Unterschied (in >90% der Fälle) vernachlässigbar ist und das Stackarray sogar um einen sehr kleinen Anteil schneller ist.

6 Fazit

Zum Abschluss dieser Ausarbeitung soll hier noch einmal eine kurze Zusammenfassung gezogen werden. Das Kernkonzept, auf das sich die meisten Konzepte zurückführen lassen, ist der häufig genannte **Array-Zerfall**. Letztendlich sind sowohl der Verlust der Größendimensionen bei der Parameterdefinition, als auch der Prozess der Indizierung relativ einfach zu verstehen, wenn die Regeln des Zerfalls rigoros befolgt werden.
Grundsätzlich ist wichtig, zwischen den beiden Haupttypen von multidimensionalen Arrays zu unterscheiden. Auf dem Stack leben kontinuierliche, "echte" Arrays, während auf dem Heap zeigerbasierte Arrays dominieren. Auf letzterem bleiben dem findigen Entwickler jedoch eine Vielzahl an Möglichkeiten. Die Allokationsart sollte demnach der Aufgabenstellung entsprechend gewählt werden.
Im direkten Performanzvergleich hat das Stackarray einen leichten Vorsprung, der aber so gering ist, dass er nun bei Programmen mit größter Performanzanforderung überhaupt in Betrachtung gezogen sollte. Insgesamt ist festzuhalten, dass multidimensionale Arrays in C auf den ersten Blick komplex und unverständlich wirken können, aber bei Beachtung einiger weniger Regeln eine deutliche Erhöhung der Lesbarkeit und Performanz einer Anwendung bewirken können.

Literatur

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

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

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

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

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

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

Listings

Abbildungsverzeichnis

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 31. März 2014 .....