Dynamische Speicherverwaltung

Einführung in die Programmierung

Michael Felderer (QE)

Institut für Informatik, Universität Innsbruck

Dynamische Speicherverwaltung

- Bisher haben wir nur statischen Speicher in unseren Programmen verwendet.
- Sehr oft kommt aber in der Praxis der Fall vor, dass beim Programmieren noch nicht bekannt ist, wie viele Daten gespeichert werden.
- Die einfachste Lösung für Arrays
  - Ein Array mit besonders viel Speicherplatz anlegen.
  - Nachteile
    - Es werden viele Ressourcen verschwendet.
    - Die Gültigkeit des Speicherbereichs des Arrays verfällt, sobald der Anweisungsblock verlassen wurde.
    - Nur globale und statische Arrays bleiben über die gesamte Programmlaufzeit erhalten.
- Lösung: Speicherplatz dynamisch zur Laufzeit anfordern.

malloc

- Dynamische Reservierung von Speicherplatz zur Laufzeit erfolgt mit der Funktion `malloc()` (`<stdlib.h>`).
- Form
  - `void *malloc(size_t size)`
  - `size_t` ist ein eigens definierter vorzeichenloser Ganzzahltyp.
  - Es wird ein zusammenhängender Speicherbereich von `size` Bytes am **Heap** reserviert.
    - Liefert die Anfangsadresse dieses Speicherbereichs.
      - Ist ein `void`-Pointer – Compiler macht aber implizites Typecasting!
    - Liefert `NULL`, wenn der Speicher nicht reserviert werden kann.
  - Beispiel (für `double`, andere Typen auch möglich!)

```
#include <stdlib.h>
…
size_t max_numbers = …;
double *numbers = malloc(max_numbers * sizeof(*numbers));
```

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

int *iarray(unsigned int n) {
      int *iptr = malloc(n * sizeof(*iptr));     // oder iptr = (int*) malloc(n * sizeof(int));
      if (iptr != NULL) {
            for (int i = 0; i < n; i++) {
                  iptr[i] = i * i;           // Alternativ: *(iptr + i) = …
            }
      }
      return iptr;
}

int main(void) {
      unsigned int val;
      printf("Wie viele int-Elemente benötigen Sie: ");
      scanf("%u", &val);
      int *arr = iarray(val);
      if (arr == NULL) {
            printf("Fehler bei der Speicherreservierung!\n");
            return EXIT_FAILURE;
      }
      printf("Ausgabe der Elemente\n");
      for (int i = 0; i < val; i++) {
            printf("arr[%d] = %d\n", i, arr[i]);
      }
      free(arr);
      return EXIT_SUCCESS;
}
```

Ausgabe:

```
Wie viele int-Elemente benötigen Sie: 3
Ausgabe der Elemente
arr[0] = 0
arr[1] = 1
arr[2] = 4
```

Interaktive Aufgabe

- Welcher Fehler wurde im folgenden Codeausschnitt gemacht?

```
double *dvals;
int *ivals;

dvals = malloc (n * sizeof(double*));
ivals = malloc (n * sizeof(ivals));
```

calloc

- Anzahl und Größe werden als Parameter übergeben.
  - `void *calloc(size_t nmeb, size_t size)`
- Verhalten ähnlich zu `malloc`
  - Es werden `nmeb*size` Bytes am Heap reserviert.
  - Liefert die Anfangsadresse des Speicherbereichs zurück.
  - Liefert `NULL`, wenn der Speicher nicht reserviert werden kann.
- ABER
  - `calloc` initialisiert den reservierten Speicher mit 0.
    - `malloc` macht das nicht!
  - `calloc` benötigt mehr Zeit als `malloc`.
    - Jede Speicherzelle muss ja initialisiert werden!
- Beispiel von vorher

```
iptr = (int*) calloc(n, sizeof(*iptr));
```

realloc

- Form
  - `void *realloc(void *ptr, size_t size)`
- Mit dieser Funktion ist es möglich, den reservierten Speicherplatz während des laufenden Programms an den aktuellen Bedarf anzupassen.
  - Damit wird das Programm wirklich vollständig dynamisch.
- Vorgangsweise
  - Es wird die Größe des durch `ptr` adressierten Speicherblocks verändert.
    - Dieser Speicherblock muss zuvor auch dynamisch erzeugt worden sein!
  - Es wird ein Zeiger auf die Anfangsadresse des (möglicherweise neu) reservierten Speicherblocks mit der neuen Größe `size` zurückgegeben.
    - Der ursprüngliche Inhalt von `ptr` bleibt aber erhalten.
    - Falls notwendig, wird der Inhalt von `ptr` kopiert.
  - Diese Funktion sollte man immer nur für größere Speicherblöcke benutzen, da das Umkopieren aufwändig werden kann (je nach Umfang der Daten)!

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#define BLKSIZE 8
int main(void) {
	int n = 0, max = BLKSIZE, z, i;
	int *numbers = calloc(BLKSIZE, sizeof(*numbers));
	if (numbers == NULL) {
		return EXIT_FAILURE;
	}
	printf("Insert numbers here --- 0 = quit\n");
	while (1) {
		printf("Number (%d): ", n + 1);
		scanf("%d", &z);
		if (z == 0) break;
		if (n >= max - 1) {
			max += BLKSIZE;
			numbers = realloc(numbers, max * sizeof(*numbers));
			if (numbers == NULL) {
				return EXIT_FAILURE;
			}
			printf("New storage : %d Bytes\n", (int) (sizeof(int) * BLKSIZE));
			printf("Overall     : %d Bytes\n", (int) (sizeof(int) * max));
			printf("Space for   : %d elements\n", max);
		}
		numbers[n++] = z;
	}
	printf("Numbers given ->\n\n");
	for (i = 0; i < n; i++)
		printf("%d ", numbers[i]);
	printf("\n");
	free(numbers);
	return EXIT_SUCCESS;
}
```

realloc – Besonderheiten

- Folgende Aufrufe sind identisch
  - `ptr = realloc(NULL, groesse);`
  - `ptr = malloc(groesse);`
- Verkleinerung des Speicherbereiches
  - Es wird der hintere Abschnitt des ursprünglichen Blocks freigegeben.
    - Der Inhalt des vorderen Abschnitts bleibt unverändert.
  - Mit `realloc(ptr,0)` kann ein Speicherbereich freigegeben werden.
    - Dafür gibt es aber eigentlich die Funktion `free` (siehe nächste Folie).
- Umkopieren bei der Vergrößerung
  - Eventuell muss aufgrund der neuen Speicheranforderung der alte Speicherbereich in einen neuen Speicherbereich umkopiert werden.
  - Zeigt noch ein Zeiger auf eine Adresse aus dem ursprünglichen Speicherbereich, dann ist diese Adresse nicht mehr gültig!
- Scheitert `realloc`, bleibt der ursprüngliche Speicherblock erhalten!

free (1)

- Nicht mehr benötigter Speicher **muss immer frei gegeben** werden.
- Erfolgt mit `free`
  - `void free(void *ptr)`
- Dabei muss man beim Aufruf beachten
  - Der übergebene Zeiger muss auf einen Speicher zeigen, der zuvor reserviert wurde (durch `malloc`, `calloc`, `realloc`) .
  - Die Funktion darf nur einmal ausgeführt werden.
  - Wird ein falscher Zeiger verwendet, oder wurde der Speicherplatz schon freigegeben, dann ist das weitere Verhalten des Programms undefiniert!
  - `free` setzt den Zeiger nicht auf `NULL`!
    - Man kann danach noch immer darauf zugreifen (Verhalten ist aber undefiniert).
    - Daher sollte man den Zeiger sofort auf `NULL` setzen!

free (2)

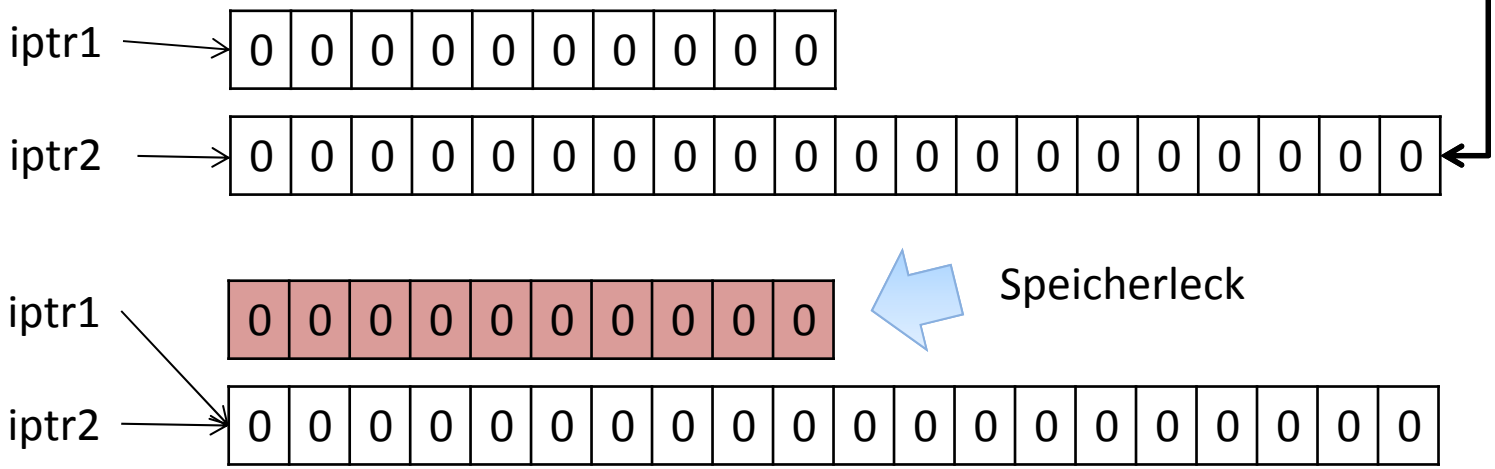
- Am Ende eines Programmes wird der reservierte Speicherplatz meist freigegeben.
- ABER
  - Das hängt von der Implementierung der Speicherverwaltung des Betriebssystems ab!
  - Man sollte sich nicht darauf verlassen!

Speicherlecks (Memory Leaks)

- Speicherleck
  - Ist ein Speicherbereich, der zwar belegt ist, aber zur Laufzeit weder verwendet noch freigegeben werden kann.
- Wie entstehen Speicherlecks?
  - Speicher wird zum Beispiel mittels `malloc()` reserviert.
  - Ein Zeiger verweist auf die Anfangsadresse des Speicherblocks.
  - Geht dieser Zeiger „verloren“ (z.B. durch eine neue Zuweisung) bevor `free` ausgeführt werden kann, dann entsteht ein Speicherleck.
    - Man kann nicht mehr darauf zugreifen bzw. bei `free` den Zeiger auf diesen Speicherbereich übergeben.

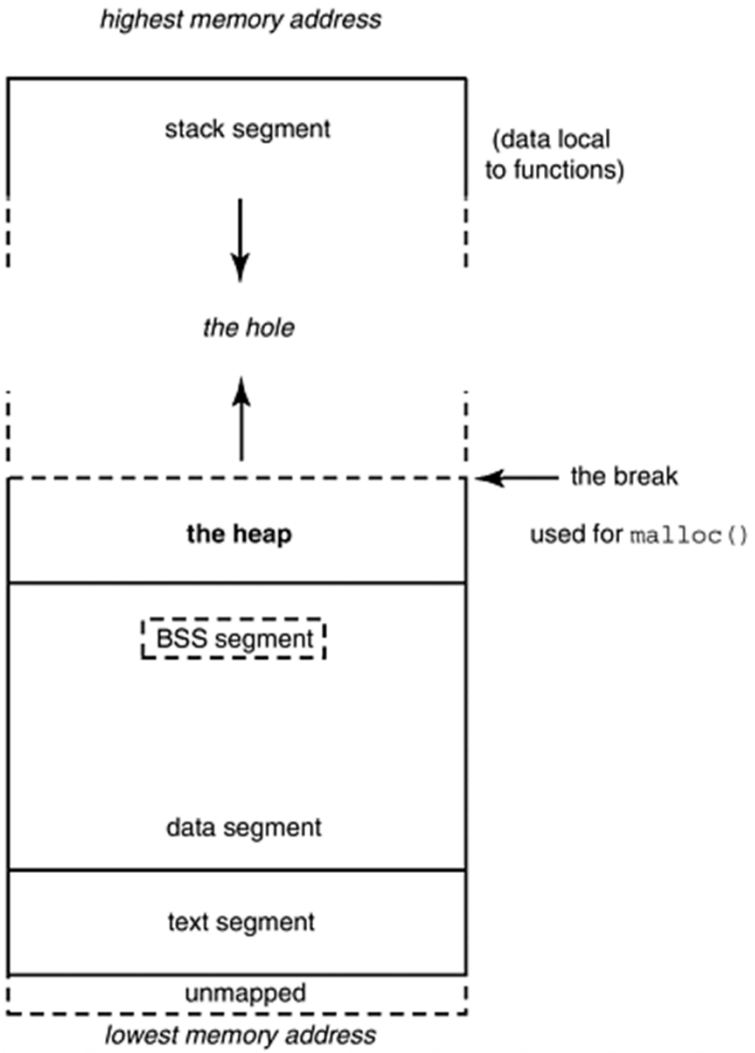
Beispiel (Speicherleck)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *iptr1, *iptr2;
    iptr1 = calloc(10, sizeof(*iptr1));
    iptr2 = calloc(20, sizeof(*iptr2));
    iptr1 = iptr2; // Speicherleck erstellt !!!
    //... nach vielen Zeilen Code, Speicher freigeben
    free(iptr1); // Gibt Speicher frei
    free(iptr2); // Fehler, Speicher wurde schon freigegeben!!!
    return EXIT_SUCCESS;
}
```



Heap

- Bei der dynamischen Speicherverwaltung kommt der Speicher vom Heap.
- Dieser Speicher wird dynamisch zugewiesen und ist bis zu einem `free` auf den entsprechenden Zeiger gültig!
  - Das kann und wird über die Lebensdauer von Funktionsaufrufen hinausgehen.
  - Damit kann man in einer Funktion Speicher reservieren, belegen und dann an die aufrufende Funktion zurückgeben.
- Heap wird **nicht** nach dem LIFO-Prinzip verwaltet.
  - Speicherbelegungen werden ja erst bei einem `free` freigegeben und das kann unterschiedlich lange dauern!



[van der Linden 94]

Beispiel (Problem mit Stack, Heap ok)

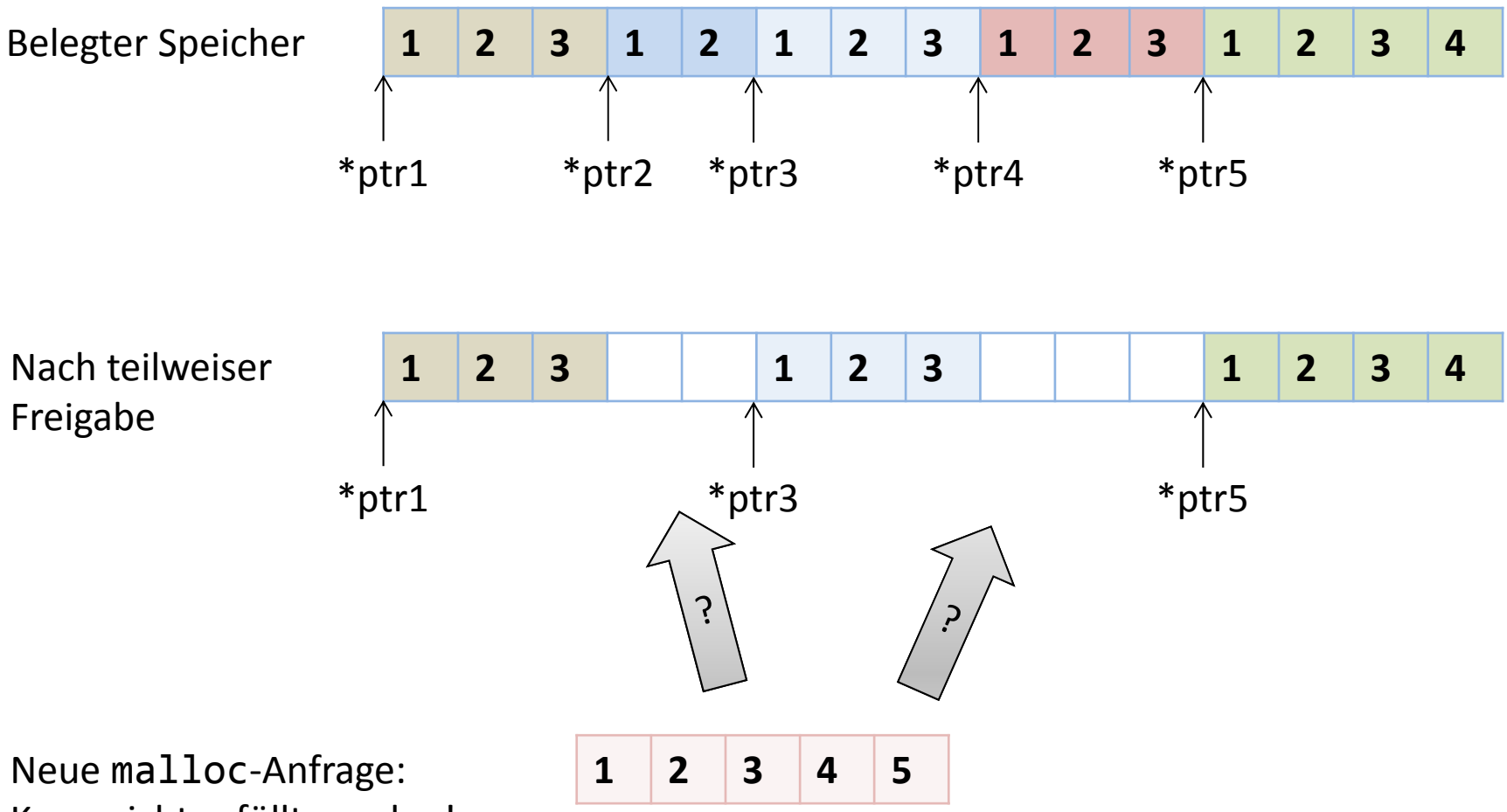
```c
#include <stdio.h>
#include <stdlib.h>
int *test1() {
	int *a, b = 5;
	a = &b;
	return a;
}
int *test2() {
	int *a, b = 5;
	a = malloc(sizeof(*a));
	*a = b;
	return a;
}
void test3() {
	double x = 10000000.0;
	double y = 20000000.0;
	x = x + 10.0;
	y = y + 20.0;
}
int main(void) {
	int *x;
	x = test1();
	test3();
	printf("%d\n", *x);
	x = test2();
	test3();
	printf("%d\n", *x);
	free(x);
}
```

Ausgabe:
1098060497
5

Heap-Fragmentierung

- Bei einer Speicheranforderung wird ein **zusammenhängender** Block angefordert.
- Wird sehr oft Speicher angefordert und freigegeben, dann kann es unter Umständen zu einer Fragmentierung kommen.
  - Die Freispeicherverwaltung des Betriebssystems merkt sich die Stellen im Speicher, die als „freier Speicher“ zur Verfügung stehen und reserviert werden können.
  - Da nicht verwendeter Speicher auch wieder freigegeben wird, entstehen Lücken im Speicherbereich.
  - Kann eine Speicheranforderung nicht durch diese Lücken bedient werden, dann schlägt die Speicheranforderung fehl.
    - Die Summe der Lücken kann dabei viel größer sein, als die eigentliche Speicheranforderung.

Heap-Fragmentierung – Beispiel



Heap-Fragmentierung – Gegenmaßnahmen

- Man sollte sich immer überlegen, ob eine dynamische Speicheranforderung überhaupt Sinn macht oder ob man nicht auch mit einem statischen Speicher auskommt.
- Speicherreservierungen sollten möglichst sparsam eingesetzt werden.
  - In besonderen Fällen sinnvoll: Gleich größere Blöcke anfordern und dann Teile davon verwenden (Pooling).
- Funktion `alloca()` benutzen, die den Speicher vom Stack und nicht vom Heap anfordert.
  - Dieser Speicher muss nicht explizit freigegeben werden, da am Ende der Funktion der Stack-Frame (und damit die Speicherallokation) vom Stack entfernt wird.
  - Nachteile
    - Am Stack ist nicht so viel Speicher vorhanden.
    - Am Ende der Funktion ist der Speicher weg.
    - Diese Funktion ist nicht ANSI-konform!

Interaktive Aufgabe

- Welcher Fehler wurde im folgenden Codeabschnitt gemacht?

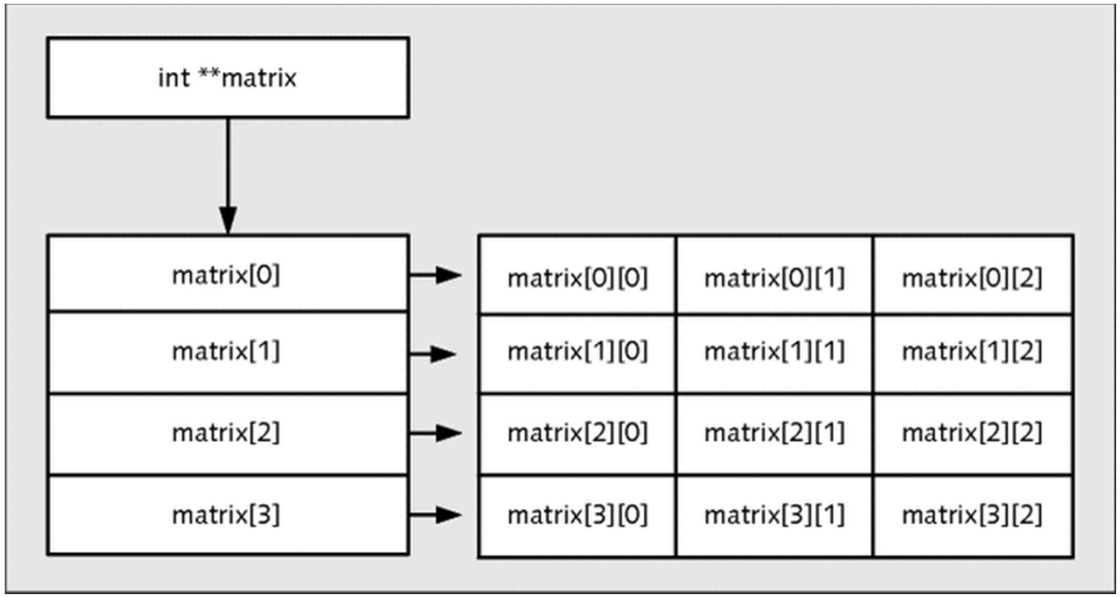
```
int *iarray1, *iarray2, i;
iarray1 = malloc(n * sizeof(int));
iarray2 = malloc(n * sizeof(int));
for( i=0; i<n; i++) {
  iarray1[i] = i;
  iarray2[i] = i+i;
}
iarray1 = iarray2;
iarray2 = iarray1;
```

Wenn die Speicherallokation fehlschlägt …

- Speicheranforderung reduzieren
  - Nicht gleich Programm beenden
  - Benutzer Alternativen anbieten (z.B. Hälfte der Anforderung etc.)
- Speicheranforderungen aufteilen und teilweise mit `realloc` probieren
- Einen Puffer konstanter Größe verwenden
  - Wenn zum Beispiel immer wieder Daten kopiert werden
- Zwischenspeichern auf der Festplatte vor der Allokation
- Nur benötigten Speicher anfordern

Dynamische Speicherverwaltung bei zweidimensionalen Arrays

- Die Matrix wird als `int**` realisiert.
- Beispiel 4×3-Matrix:



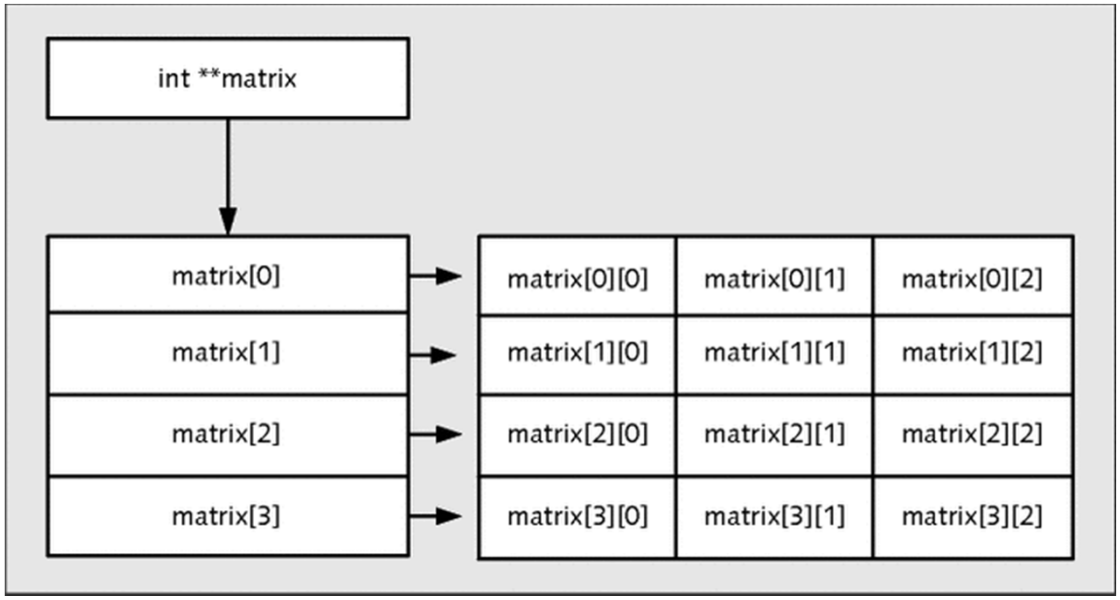
- Es muss zunächst Speicherplatz für die Zeilen (erste Dimension) und danach für jede Spalte (zweite Dimension) reserviert werden
- Beim Freigeben muss dasselbe in umgekehrter Reihenfolge durchgeführt werden (zuerst Spalten-, dann Zeilenfreigabe)

Speicherallokation und Freigabe bei zweidimensionalen Arrays

```
int **init_matrix(int dim1, int dim2) {
  int **matrix;

  if ((matrix = calloc(dim1, sizeof(int *))))
  for (int i = 0; i < dim1; i++) {
    if (!(matrix[i] = calloc(dim2, sizeof(int)))) {
      printf("No memory for line %d", i);
      exit(EXIT_FAILURE);
    }
  }
  else {
    printf("Out of memory ...");
    exit(EXIT_FAILURE);
  }

  return matrix;
}

void free_matrix(int **matrix, int dim1) {
  for (int i = 0; i < dim1; i++)
    free(matrix[i]);
  free(matrix);
}
```

Beispiel (Zugriff auf die Matrix)

Zugriff auf …	Möglichkeit 1	Möglichkeit 2	Möglichkeit 3
0. Zeile, 0. Spalte	`**matrix`	`*matrix[0]`	`matrix[0][0]`
i. Zeile, 0. Spalte	`**(matrix+i)`	`*matrix[i]`	`matrix[i][0]`
0. Zeile, i. Spalte	`*(*matrix+i)`	`*(matrix[0]+i)`	`matrix[0][i]`
i. Zeile, j. Spalte	`*(*(matrix+i)+j)`	`*(matrix[i]+j)`	`matrix[i][j]`



Valgrind (1)

- **Valgrind**
  - Ist ein quelloffenes Analyseframework für Unix-basierte Systeme zum Debuggen, Profilen und zur dynamischen Fehleranalyse von Programmen.
- Einsatzbereiche
  - **Auffinden von Speicherlecks**
  - Cacheanalyse
  - Erstellen von Call-Graphen
  - Ermitteln des Ressourcenbedarfs
  - Auffinden von Race Conditions/Deadlocks
- Unterstützte Plattformen
  - Linux: *x86, AMD64, PPC32, PPC64*
  - Mac OS X: *x86, AMD64*

Valgrind (2)

- Analysiert die Maschinenbefehle des zu untersuchenden Programms vor deren Ausführung, oft zusammen mit ergänzendem Code („Instrumentierung“).
- Arbeitet auf dem **Binärcode** der Anwendung, daher unabhängig von der verwendeten Programmiersprache und dem Compiler bzw. Interpreter.
- **Optimiert für C/C++**, jedoch auch für andere Sprachen wie Fortran, Java, Perl, Python, usw. geeignet.
- Eingeschleppte Fehler durch Compiler bzw. Interpreter auffindbar.
- Kommandozeilenorientiertes Programm

```
$ valgrind --tool=<NAME> ./binary [ARGUMENTS]
```

<NAME> = Name des Tools *(**memcheck**, cachegrind, callgrind, massif, helgrind)*
./binary = das zu untersuchende Programm
[ARGUMENTS] = optionale Argumente für das zu untersuchende Programm

Memcheck

- Eines der meistverwendeten Tools von Valgrind.
- Erkennt
  - Potentiell **fehlerhafte Zugriffe** auf Speicherbereiche
    - Lese- und Schreibzugriffe auf freigegebenen Speicher
    - Schreiben über die Speichergrenzen hinaus
    - Verwendung von Adresszeigern mit ungültigem Wert (*dangling pointers*)
  - die Benutzung von **nicht initialisierten Variablen**
  - **falsches Freigeben** von Speicher (z.B. doppeltes Freigeben)
  - **Speicherlecks**
  - bei Verwendung von `memcpy` oder ähnlichen Funktionen, wenn sich **Quell- und Zielbereich überlappen**
  - …

Memcheck-Beispiel (1)

- Source Code:

leaktest.c

```
1: #include <stdlib.h>
2:
3: int main(int argc, char *argv[]) {
4:     int i, *p = (int*) calloc(10, sizeof(int));
5:     for (i = 0; i <= 10; ++i)
6:         p[i] = i;
7:     return EXIT_SUCCESS;
8: }
```

```
$ gcc –Wall -Werror –std=c99 –g leaktest.c -o leaktest
```

- GDB Debugger:

```
[c703288@zid-gpl Vorlesung]$ gdb ./leaktest
GNU gdb (GDB) Fedora (7.3-43.fc15)
Copyright (C) 2011 Free Software Foundation, Inc.
…
…
(gdb) r
Starting program: /afs/zid1.uibk.ac.at/home/c703/c703288/EidP/Vorlesung/leaktest
[Inferior 1 (process 9028) exited normally]
```

Memcheck-Beispiel (2)

- Valgrind:

```
[c703288@zid-gpl Vorlesung]$ valgrind --tool=memcheck ./leaktest
==9141== Memcheck, a memory error detector
==9141== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==9141== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==9141== Command: ./leaktest
==9141==
==9141== Invalid write of size 4
==9141==    at 0x4004FF: main (leaktest.c:6)
==9141==  Address 0x4c51068 is 0 bytes after a block of size 40 alloc'd
==9141==    at 0x4A04B84: calloc (vg_replace_malloc.c:467)
==9141==    by 0x4004E1: main (leaktest.c:4)
==9141==
==9141==
==9141== HEAP SUMMARY:
==9141==     in use at exit: 40 bytes in 1 blocks
==9141==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==9141==
==9141== LEAK SUMMARY:
==9141==    definitely lost: 40 bytes in 1 blocks
==9141==    indirectly lost: 0 bytes in 0 blocks
==9141==      possibly lost: 0 bytes in 0 blocks
==9141==    still reachable: 0 bytes in 0 blocks
==9141==         suppressed: 0 bytes in 0 blocks
==9141== Rerun with --leak-check=full to see details of leaked memory
==9141==
==9141== For counts of detected and suppressed errors, rerun with: -v
==9141== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Weitere Tools

- **cachegrind**
  - Profiling Tool zum Aufspüren von *Cache Misses* und falscher *Branch Prediction*.
- **callgrind**
  - Profiling Tool für Funktionsaufrufe, wobei neben der Simulation des Caches auch *CPU-Instruktionen* gezählt und *Call-Graphen* zur Darstellung der Funktionsabhängigkeiten erzeugt werden können.
- **massif**
  - Speicher Profiling Tool zum Ermitteln des *Speicherverbrauchs* (Heap und Stack) einzelner Funktionen zur Laufzeit.
- **helgrind**
  - Hilft beim Aufspüren von *Synchronisationsfehlern* in Multi-Threaded Anwendungen welche auf POSIX Threads (Pthreads) basieren.

Literatur

- [van der Linden 94] Peter van der Linden, **Expert C Programming – Deep C Secrets**, Prentice Hall, 1994