Dynamische Speicherverwaltung

Tim Dobert

17.05.2013

Inhaltsverzeichnis

1 Allgemeines zur Speichernutzung

In modernen Systemen sind oft verschiedene Arten von Speicher verbaut. Generell gilt dabei, die schnellen Speicherbausteine bieten wenig Platz, die langsameren viel. Beim Schreiben von einfachen Programmen muss man sich darum allerdings keine Gedanken machen welche Daten in welchen Speicher kommen. Das übernimmt meistens das Betriebssystem. Dieses stellt einen kontinuierlichen, virtuellen Adressraum zur Verfügung, der meist mehrere Speicherarten in der Hardware überspannt. Jedes Programm, was gestartet wird, bekommt einen Teil dieses Adressraums zugeteilt und kann diesen frei nutzen.
Der Speicherplatz, den ein Programm nutzen kann, enthält drei wichtige Teile: Konstanten-Pool, Stack und Heap. Der Konstanten Pool enthält vor allem Daten, die mit `const` (in C) gekennzeichnet werden. Dieser Teil ist statisch, da Größe und Inhalt dieser Daten zur Übersetzungszeit feststehen. Auf dem Stack werden unter Anderem lokale Variablen und Parameter gespeichert. Er ist der wichtigste Teil für Berechnungen und Funktionsaufrufe. In der Assemblerprogrammierung kommt man nicht daran vorbei, ihn intensiv zu nutzen. Der dritte Teil, der Heap, ist ein frei nutzbarer, ungeordneter Speicherraum. Wenn über dynamische Speicherverwaltung gesprochen wird, ist damit in der Regel die Benutzung des Heaps gemeint. In der Praxis bedeutet das, Speicher wird reserviert, beschrieben bzw. gelesen und wieder freigegeben, wenn er nicht mehr benutzt wird.

2 Ziel und Nutzen

Generell ist es immer dann sinnvoll, Speicher dynamisch zu nutzen, wenn zur Übersetzungszeit noch nicht feststeht, wie viel Speicherplatz an einer Stelle gebraucht wird. Ist zum Beispiel nicht absehbar, wie groß ein Array sein muss, würde es unnötig Speicherplatz verschwenden, provisorisch ein viel zu großes Array anzulegen. Das dynamische Erzeugen von Arrays auf dem Heap löst dieses Problem und ist in den meisten Fällen sehr einfach. Ein weiteres Anwendungsgebiet sind Structs. Durch die dynamische Verwendung von Structs kann in vielen Bereichen der Funktionalität von objektorientierten Sprachen nahekommen.

3 Anwendung in C

Generell wird der Heap in C so genutzt: Mithilfe eines Befehls wird nach Speicher von einer bestimmten Größe gefragt. Das System alloziert dann einen Speicherblock dieser Größe und gibt seine Adresse zurück. Über diese Adresse kann der Speicher dann beschrieben und ausgelesen werden.

Im nächsten Abschnitt werden die vier Grundbefehle aus der Standardbibliothek (stdlib) vorgestellt.

3.1 malloc

Der einfachste Befehl um Speicher zu allozieren ist *malloc*.

```
void* malloc(int)
```

Die Methode nimmt eine Anzahl Bytes als Parameter, alloziert Speicher dieser Größe und gibt die Adresse dann als Zeiger zurück. Der Rückgabewert sollte gut aufgehoben werden, da er der einzige Zugang zu dem Speicher ist. Hier ist eine beispielhafte Verwendung:

```
int* number = (int*) malloc(sizeof(int));
*number = 3;
//Es wurde eine 3 abgespeichert
```

Die Typumwandlung des Rückgabewertes ist nicht explizit notwendig, sie findet auch implizit statt (in C++ ist sie aber notwendig). Die Angabe der Größe sollte in der Regel immer mithilfe von `sizeof` geschehen. Das ist praktischer, als sich die Größe selbst auszurechnen, vor allem bei Structs und macht es außerdem einfacher, den Code auf andere Systeme zu portieren.

3.2 calloc

malloc reserviert lediglich Speicher, ändert aber seinen Inhalt nicht. Wird gleich nach dem Aufruf nachgesehen was drinsteht, erhält man mehr oder weniger zufällige Werte. Falls koplett leerer Speicher benötigt wird, ist es am praktischsten, die Methode *calloc* zu benutzen.

```
void* calloc(int, int)
```

Diese Methode funktioniert ähnlich wie malloc. Der reservierte Speicher wird hier aber gleich mit Null initialisiert. Die Parameter sind auch etwas anders: der Erste besagt wie viele Blöcke reserviert werden sollen, der Zweite wie groß jeder Block sein soll (ebenfalls in Byte). Nützlich ist calloc zum Beispiel bei Char-Arrays, da diese dann gleich Null-terminiert sind.

3.3 realloc

Ein reservierter Speicherblock kann nicht ohne weiteres vergrößert werden, da der Speicher um ihn herum möglicherweise für etwas anderes reserviert ist. Wird erst bei der Ausführung des Programms klar, das bereits allozierter Speicher nicht ausreicht, kann die Methode *realloc* benutzt werden.

```
void* realloc(void*, int)
```

realloc nimmt einen Zeiger auf bereits reservierten Speicher als Parameter, sowie eine Größe in Bytes. Dann wird der Speicher vergrößert oder neu alloziert, sodass der zurückgegebene Zeiger anschließend auf Speicher mit der neuen Größe zeigt. Daten, die zuvor im Speicher standen gehen nicht verloren. Diese Methode zu benutzen ist um einiges einfacher, als neuen Speicher mit malloc zu reservieren un den Inhalt zu kopieren.

3.4 free

Zusammen mit malloc ist *free* mit der wichtigste Befehl. Speicher, der reserviert wurde bleibt reserviert, bis er explizit freigegeben wird. Die Freigaben erfolgt mit dieser Methode.

```
void free(*void)
```

free gibt den Speicherplatz, auf den der übergebene Zeiger zeigt, wieder frei. Jede Art von Speicherallokation sollte irgendwann von einem free gefolgt werden.

4 Beispielprogramm

Im Anhang (Teil 7) befindet sich der Quellcode zu einem Beispielprogramm. Es implementiert ein dynamisches Array, das automatisch größer wird, wenn es voll ist. Es dient nur der Demonstration und implementiert keine vollständige Liste.

4.1 Funktionsweise

Das Array ist eine Struktur, bestehend aus einem Zeiger auf den Speicher, der momentanen Größe des reservierten Speichers und dem Index der nächsten freien Zelle. Dieses Array wird initialisiert (`intitArray()`), indem eine Startgröße festgelegt, Speicher dieser Größe reserviert und der Index auf null gesetzt wird. Das Einfügen eines neuen Elements (`add(int element)`) geschieht wie gewohnt, solange noch Platz ist. Ist kein Platz mehr, wird zunächst mit realloc der Speicherplatz auf das Doppelte vergrößert. Vor Beenden des Programms, wird der Speicherplatz wieder freigegeben und der Zeiger des Arrays vorsichtshalber auf NULL gesetzt.

5 Vergleich mit anderen Sprachen

5.1 Java

Der Vergleich zu Java ist aus mehreren Gründen interessant. Java bietet keine Befehle, mit denen sich direkt Speicher einer bestimmten Größe reservieren oder freigeben lässt. Beim erzeugen eines Objekts (mithilfe des

new-Operators) wird automatisch Speicherplatz reserviert und von dem Objekt belegt. Der andere große Unterschied zu C liegt bei der Freigabe des Speichers. In C bleibt Speicher so lange reserviert, bis er manuell wieder freigegeben wird. geschieht das nicht, kann es irgendwann dazu kommen, dass es keinen freuen Speicher mehr gibt und das Programm abstürzt. Java hingegen benutzt einen Garbage Collector. Dieser sorgt während der Laufzeit automatisch dafür, dass Speicher, auf den den mehr zugegriffen wird, freigegeben wird. Das macht das Programmieren teilweise angenehmer, gibt einem aber auch weniger direkte Kontrolle.

5.2 C++

C++ stellt eine Obermenge zu C dar und bietet deswegen auch alle vorgestellten Befehle an. In der Praxis werden sie allerdings kaum benutzt. Stattdessen benutzt man auch hier überwiegend den new-Operator, der die Allokation automatisch übernimmt. Freigegeben wird der Speicher dann mit der Methode delete(void*), die in dem Referenzierten Objekt eine Destruktormethode aufruft, es anschließend löscht und dem von ihm besetzten Speicher freigibt.

6 Praktische Tipps

Bei dynamischer Speichernutzung ist immer ein wenig Vorsicht geboten. Fehlerhafte Handhabung von Speicher kann unvorhersehbares Verhalten auslösen. Deshalb sollte man einige Dinge immer im Kopf behalten:

1. Reservierter Speicher sollte am Ende des Sichtbarkeitsbereiches des Pointers freigegeben werden.
   Der Pointer ist der einzige Weg, auf den Speicher zuzugreifen. Wenn er verloren geht, z.B. weil der Sichtbarkeitsbereich verlassen wird, bleiben nutzlose Daten im Speicher liegen.

2. Der Pointer sollte nach free auf NULL gesetzt werden.
   Der Versuch, auf bereits Freigegebenen Speicher zuzugreifen, kann zu bösen Fehlern führen. Damit es gar nicht dazu kommen kann, sollte nach dem Freigeben des Speichers der dazugehörige Zeiger eine NULL-Referenz erhalten.

Eine weitere Möglichkeit Zugriffsfehler vor allem bei Arrays zu vermeiden ist, beim reservieren des Speichers die Größe irgendwo zu sichern. Falls diese Größe überschritten werden soll, kann mit realloc sonst einfach erst mal der Platz vergrößert werden.

7 Quellen und Anhang

7.1 Quellen

Literatur

[1] Rechnerstrukturen Skript Kapitel 17,
http://tams.informatik.uni-hamburg.de/lectures/2012ws/vorlesung/rs/doc/rsWS12-18.pdf

[2] C dynamic memory allocation,
http://en.wikipedia.org/wiki/C_dynamic_memory_allocation

[3] Advanced Memory Management: Dynamic Allocation,
http://www.cprogramming.com/tutorial/dynamic_memory_allocation.html

7.2 Code des Beispielprogramms

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

struct DynamicArray
//Ein Integer Array
{
    int *point;
    unsigned int size;
    unsigned int nextFree;
};

struct DynamicArray array;

//Reserviert Speicher und stellt den Startzustand her
void initArray();

//Gibt den Speicher frei und und setzt den Pointer auf Null
void finalize();

//Fuegt einen Integer am Ende hinzu. Vergroeßert ggf. das Array
void add(int element);

//Gibt das gesamte Array auf der Konsole aus
void printAll();

//Liest Einen Integer vom Benutzer ein
int getInput();


int main(/* int argc, char** argv */)
{
    initArray();
    //Benutzerinformationen
    printf("Was soll getan werden?\n 0: Beenden\n 1: ");
    printf("Element hinzufuegen\n 2: Array ausgeben\n ");
    printf("3: Speicher ausgeben\n 4: Array zuruecksetzen\n");
```

```
    int input = 0;         //Für Benutzerinputs
    bool running = true;  //Wurde das Programm schon beendet?

    //Main Loop
    do  //while(running)
    {
        printf("Eingabe: ");
        input=getInput();

        switch(input)
        {
            case 0:
                running = false;
                break;
            case 1:
                printf("Fuege diesen Integer hinzu: ");
                add(getInput());
                break;
            case 2:
                printAll();
                break;
            case 3:
                printf("Das Array belegt %d Byte.\n",
                       (array.size * sizeof(int)));
                break;
            case 4:
                finalize();
                initArray();
                break;
            default:
                printf("Ungueltige Eingabe. ");
                printf("Bitte etwas anderes eingeben.\n");
        }
        input=0;
        printf("\n");
    }
    while(running);

    finalize();
    //Programm endet erfolgreich, wenn der Speicher freigegeben wurde
    if(array.point != NULL)
        return 1;
    return 0;
}

void initArray()
{
    printf("Array und Speicherplatz wird initialisiert...\n");
    array = (struct DynamicArray{NULL, 2, 0});
    array.point = malloc(sizeof(int) * array.size);
}

void finalize()
{
    printf("Speicherplatz wird freigegeben...\n");
    free(array.point);
    array.point = NULL;
}

void add(int element)
{
```

```
    //Wenn kein Platz mehr ist: Vergroeßern
    if(array.nextFree >= array.size)
    {
        array.size *= 2;
        array.point = realloc(array.point, sizeof(int) * array.size);
    }
    //Hinzufuegen des Elements
    array.point[array.nextFree] = element;
    printf("%d hinzugefuegt\n", array.point[array.nextFree]);
    ++array.nextFree;
}

void printAll()
{
    if(array.nextFree != 0)
    {
        printf("Array wird ausgegeben... \n");
        for(unsigned int i=0; i < array.nextFree; ++i)
        {
            printf( "Platz %d: %d\n", i, array.point[i]);
        }
    }
    else
    {
        printf("Das Array ist momentan leer.\n");
    }
}

int getInput()
{
    int input = 0;
    char buff[40];
    fgets(buff, 40, stdin);
    buff[10] = 0x00;     //die ersten 10 Zeichen
    sscanf(buff, "%d", &input);
    return input;
}
```