Dynamische Speicherverwaltung

Sebastian Stabinger, Thomas Hausberger

SS2021

Stack vs. Heap

Stack vs. Heap

Stack

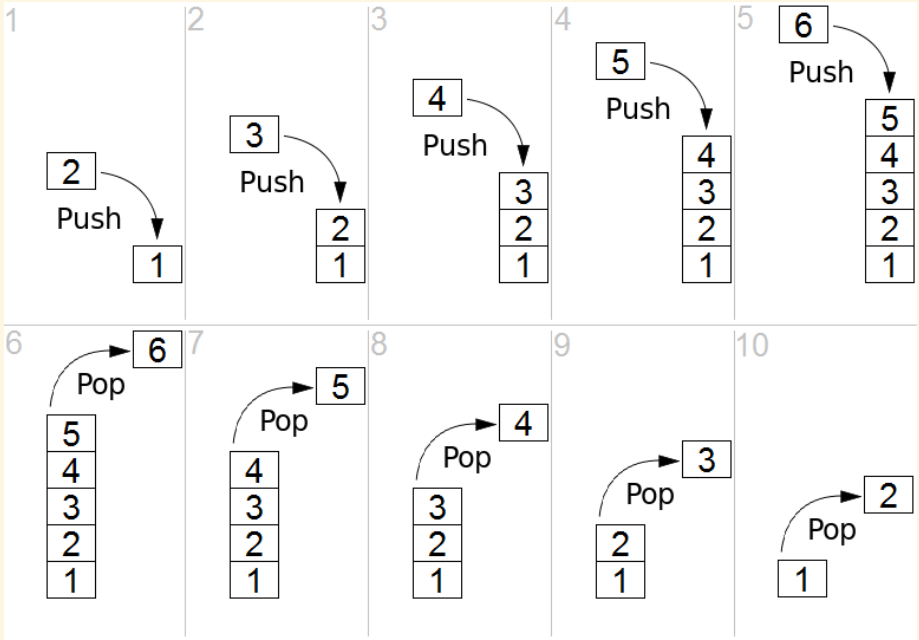
- Enthält lokale Variablen, übergebene Parameter, intern benötigte Informationen (z.B. Rücksprungadresse beim Aufruf von Funktionen, ...)
- Wird vom Compiler/System automatisch verwaltet
- In der Größe beschränkt

Heap

- Muss vom Programmierer manuell verwaltet werden
- Es kann so viel Speicher verwendet werden wie das System erlaubt (RAM, Swap, ...)

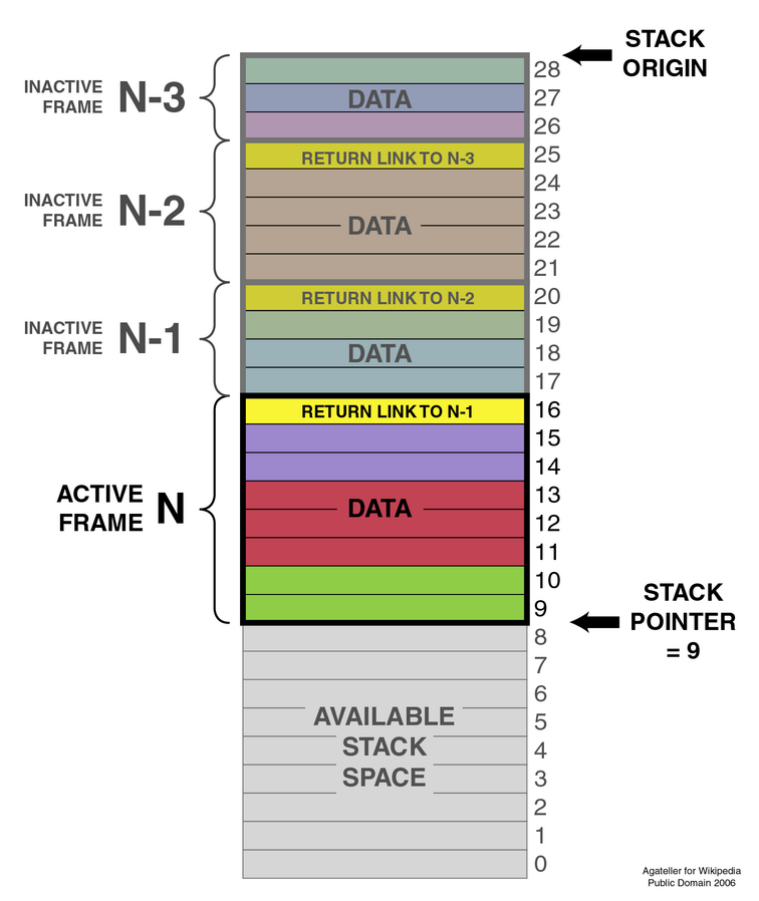
Stack (Stapel) — Datenstruktur

- Häufig verwendete Datenstruktur
- Wir können Elemente oben auf einen Stapel legen (push)
- Wir können das oberste Element des Stapels entfernen (pop)



Stack (Stapel) — Verwaltung von Speicher von Funktionen

- Sämtlicher Speicher der für eine Funktion benötigt wird bezeichnet man als Stackframe
- Diese Stackframes werden auf einem Stack verwaltet
- Wird eine Funktion aufgerufen, kommt ein neuer Stackframe auf den Stapel
- Wenn eine Funktion fertig ist (`return`), wird der oberste Stackframe vom Stapel entfernt



Der Stack verwaltet sich selbst

- Nachdem der Stackframe der aktuell laufenden Funktion vom Stapel fliegt wenn die Funktion beendet ist, wird nicht mehr benötigter Speicher auf dem Stack automatisch freigegeben

Beispiel

```c
#include <stdio.h>

int square(int n) {
  int tmp = n * n;
  return tmp;
}

int main() {
  int a = 2;
  int b = square(a);
  printf("%d\n", b);
}
```

- Stackframe für `square`
  - Speicher für `n`
  - Speicher für `tmp`
- Stackframe für `main`
  - Speicher für `a`
  - Speicher für `b`
  - Speicher für Antwort von `square`

Einschränkungen des Stack

- Die Größe des Stacks ist beschränkt
- Wie viel Speicher für einen Stackframe reserviert ist, muss fix sein
  - d.h. wir können nicht während der Laufzeit z.B. ein Array größer machen, wenn wir bemerken, dass es nicht groß genug ist
- Speicher in einem Stackframe den wir verwenden können ist immer an einen Namen gebunden.
  - Was wenn wir Speicher nur dann reservieren wollen wenn er wirklich gebraucht wird?

Einschränkungen des Stack — Beispiel

- Wir haben eine Struktur (`Model`) definiert in der wir 3D-Modelle für ein Spiel speichern wollen und jede dieser Strukturen benötigt 8 MiB Speicher
- Unser Spiel soll gleichzeitig maximal 1024 Modelle unterstützen

Naive Lösung

- Wir reservieren Speicher für 1024 Modelle: `Model modelstore[1024];`

Problem:

- Wir reservieren immer 8 GiB Speicher, selbst wenn wir tatsächlich nur eine Hand voll Modelle laden

Einschränkungen des Stack — Beispiel

- Wir haben eine Struktur (`Model`) definiert in der wir 3D-Modelle für ein Spiel speichern wollen und jede dieser Strukturen benötigt 8 MiB Speicher
- Unser Spiel soll gleichzeitig maximal 1024 Modelle unterstützen

Lösungsansatz

- Wir reservieren Speicher für 1024 Zeiger auf Modelle: `Model* modelstore[1024];`

Vorteil:

- Ein Zeiger ist immer gleich groß und recht klein (z.B. 64 Bit $\rightarrow$ 8 Byte). Das Array braucht also z.B. nur 8 KiB.
- Wir können die Zeiger mit `NULL` initialisieren und wissen immer welcher Platz im Array wirklich ein echtes Modell enthält

Neues Problem:

- Wie können wir jetzt aber neue Strukturen erzeugen und einen Zeiger darauf in unserem Array speichern?

Einschränkungen des Stack — Falsche Lösung 1

```c
typedef struct Model {
  double data[1024 * 1024];
} Model;

Model *load_model(char *filename) {
  Model loadedmodel;
  // ...
  // Hier wird das Modell von der Festplatte geladen
  // und die Daten in loadedmodel geschrieben
  return &loadedmodel;
}

int main() {
  Model *modelstore[1024];
  modelstore[0] = load_model("player.3d");
  modelstore[1] = load_model("enemy.3d");
  modelstore[2] = load_model("tree.3d");
  // ...
}
```

- Diese Lösung funktioniert nicht, weil der Speicher für `loadedmodel` nach Beenden von `load_model` automatisch frei gegeben wird. D.h. der Zeiger ist nicht mehr gültig!

Einschränkungen des Stack — Falsche Lösung 2

```c
typedef struct Model {
  double data[1024 * 1024];
} Model;

Model load_model(char *filename) {
  Model loadedmodel;
  // ...
  // Hier wird das Modell von der Festplatte geladen
  // und die Daten in loadedmodel geschrieben
  return loadedmodel; // Wir gegen direkt eine Kopie zurück
}

int main() {
  Model *modelstore[1024];
  Model m = load_model("player.3d");
  modelstore[0] = &m;
  m = load_model("enemy.3d");
  modelstore[1] = &m;
  m = load_model("tree.3d");
  modelstore[2] = &m;
  // ...
}
```

- Funktioniert nicht, weil der Inhalt von `m` jedes mal überschrieben wird

Einschränkungen des Stack — Problematische Lösung

```c
typedef struct Model {
  double data[1024 * 1024];
} Model;

Model load_model(char *filename) {
  Model loadedmodel;
  // ...
  // Hier wird das Modell von der Festplatte geladen
  // und die Daten in loadedmodel geschrieben
  return loadedmodel; // Wir gegen direkt eine Kopie zurück
}

int main() {
  Model *modelstore[1024];
  Model m1 = load_model("player.3d");
  modelstore[0] = &m1;
  Model m2 = load_model("enemy.3d");
  modelstore[1] = &m2;
  Model m3 = load_model("tree.3d");
  modelstore[2] = &m3;
  // ...
}
```

- Diese Lösung funktioniert, ist aber nicht Dynamisch → Da man für jedes Modell eine Variable anlegen muss, muss man beim Compilieren schon wissen wie viele Modelle man laden will

Heap (Haufen)

- Als Lösung für solche Probleme verwendet man statt dem Stack den sogenannten Heap (auch Free Store genannt) um Daten zu speichern
- Der Heap ist der Teil von einem Programm, wo der größte Teil des verfügbaren Speichers zu finden ist.
  - Wenn ihr z.B. 7 GiB freien RAM habt könnt ihr diese 7 GiB über den Heap verwenden. Der Stack ist gewöhnlich viel kleiner.
- Der Heap ist ein Stück Speicher ohne weitere Struktur (daher der Name)
- Der Heap wird mittels dynamischer Speicherverwaltung verwendet

Dynamische Speicherverwaltung

Allgemeines

- Um die gleich erwähnten Funktionen verwenden zu können muss `stdlib.h` mit `#include` eingebunden werden

Reservieren von Speicher

- Speicher wird mit der Funktion `malloc` reserviert
- Als einzigen Parameter nimmt die Funktion die Größe des zu reservierenden Speichers in Byte entgegen
- Die Funktion gibt die Adresse des ersten Bytes des reservierten Speichers zurück
- Falls etwas schief gelaufen ist, wird `NULL` zurück gegeben

Reservieren von Speicher — Beispiele

Reservieren von Speicher für einen Integer

```
int *ip = malloc(sizeof(int));

if (ip != NULL) {
  *ip = 23;
  printf("%d\n", *ip);
} else
  printf("Etwas ist schief gelaufen!\n");
```

Reservieren von Speicher für 10 Double

```
double *double_arr = malloc(sizeof(double) * 10);

if (double_arr) {
  double_arr[8] = 23;
  printf("%f\n", double_arr[8]);
} else
  printf("Etwas ist schief gelaufen!\n");
```

Freigeben von Speicher

- Speicher wird mit der Funktion `free` freigegeben
- Als einzigen Parameter nimmt die Funktion die Adresse des ersten Bytes eines vorher reservierten Speicherbereichs entgegen
- Falls die übergebene Adresse `NULL` ist, macht die Funktion nichts
- Ein Speicherbereich darf nur ein mal mit `free` freigegeben werden!

Freigeben von Speicher — Beispiele

Reservieren von Speicher für einen Integer mit Freigabe

```c
int *ip = malloc(sizeof(int));

if (ip != NULL) {
  *ip = 23;
  printf("%d\n", *ip);
} else
  printf("Etwas ist schief gelaufen!\n");

free(ip);
```

Reservieren von Speicher für 10 Double mit Freigabe

```c
double *double_arr = malloc(sizeof(double) * 10);

if (double_arr) {
  double_arr[8] = 23;
  printf("%f\n", double_arr[8]);
} else
  printf("Etwas ist schief gelaufen!\n");

free(double_arr);
```

Vergrößern/Verkleinern von reserviertem Speicher

- Bereits reservierter Speicher kann mit der Funktion `realloc` vergrößert/verkleinert werden
- Werte die schon im Array stehen bleiben erhalten (bis auf Werte die beim Verkleinern verloren gehen)
- Die Funktion nimmt als Parameter die Adresse des ersten Bytes eines vorher reservierten Speicherbereichs und die neue Größe in Byte entgegen
- Die Funktion liefert entweder die ursprüngliche Adresse des ersten Bytes zurück, oder eine neue falls der Speicher aus Platzgründen umkopiert werden musste
- Falls etwas schief gelaufen ist, wird `NULL` zurück gegeben

Vergrößern/Verkleinern — Beispiele

Beispiel 1

```
int *arr = malloc(sizeof(int) * 10);
// arr hat jetzt Platz für 10 Integerwerte
arr = realloc(arr, sizeof(int) * 20);
// arr hat jetzt Platz für 20 Integerwerte
```

Mit kompletter Fehlerbehandlung

```
int *arr = malloc(sizeof(int) * 10);
if (arr) {
  // arr hat jetzt Platz für 10 Integerwerte
  int *newarr = realloc(arr, sizeof(int) * 20);
  if (newarr) {
    arr = newarr;
    // arr hat jetzt Platz für 20 Integerwerte
  } else {
    printf("Vergrößern des Speichers hat nicht geklappt!\n");
    // arr ist noch gültig und hat immer noch Platz für nur 10 Integer
  }
} else
  printf("Reservierung des Speichers hat nicht geklappt!\n");
```

Memory Leak / Speicherleck

- Das Problem von dynamischer Speicherverwaltung ist, dass hier leicht Fehler passieren können
- Wenn man die Adresse zu einem dynamisch reservierten Speicherbereich verliert, kann man nicht mehr darauf zugreifen und den Speicher auch nicht mehr mittels `free` frei geben
- Der Speicherplatz ist damit bis zum Programmende verloren!
- Man bezeichnet so etwas als Speicherleck (auf Englisch Memory leak)
- Sehr häufiger Fehler in C/C++ Programmen die nach einiger Zeit zum Programmabsturz führen weil der Speicher ausgeht

Memory Leak — Beispiel

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

typedef struct Complex {
  double real, imag;
} Complex;

Complex *randcomplex() {
  Complex *res = malloc(sizeof(Complex));
  res->imag = rand() % 1000;
  res->real = rand() % 200;
  return res;
}

int main() {
  double realsum = 0;
  double imagsum = 0;
  for (int i = 0; i < 100; i++) {
    Complex *c = randcomplex();
    realsum += c->real;
    imagsum += c->imag;
    // Wir müssten hier eigentlich free(c) aufrufen!
  }
  // Wir haben in der for-Schleife 800 Byte Speicher verloren
  printf("Durchschnitt real=%f, imag=%f\n", realsum / 100, imagsum / 100);
}
```

Praktische Anwendungen

Dynamisches Erzeugen von einem Array

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

int main() {
  // Reserviert Speicher für 1024 Integer
  int *dynarr = malloc(sizeof(int) * 1024);
  if (dynarr != NULL) {
    // Kann danach verwendet werden wie jedes andere Array auch
    dynarr[23] = 42;
    dynarr[47] = 2;
    printf("dynarr[23] = %d\n", dynarr[23]);
    printf("dynarr[47] = %d\n", dynarr[47]);
  } else {
    printf("Etwas ist beim Erzeugen des Arrays schief gelaufen!\n");
  }

  // Wenn wir fertig sind, wird der Speicher des Arrays wieder frei gegeben
  free(dynarr);
}
```

Rückgabe eines neuen Arrays von einer Funktion

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

double *reserve_and_init(int size, double val) {
  double *arr = malloc(sizeof(double) * size);
  if (arr != NULL) {
    for (int i = 0; i < size; i++)
      arr[i] = val;
  }
  return arr;
}

int main() {
  double *dynarr = reserve_and_init(1024, 23.42);
  if (dynarr) {
    printf("dynarr[23] = %f\n", dynarr[23]);
    printf("dynarr[123] = %f\n", dynarr[147]);
  } else
    printf("Etwas ist beim Erzeugen des Arrays schief gelaufen!\n");

  // Wenn wir fertig sind, wird der Speicher des Arrays wieder frei gegeben
  free(dynarr);
}
```

Größenänderung eines Arrays

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

int main() {
  // Reserviert Speicher für 128 Integer
  int *dynarr = malloc(sizeof(int) * 128);
  if (dynarr != NULL) {
    // Kann danach verwendet werden wie jedes andere Array auch
    dynarr[23] = 42;
    dynarr[47] = 2;
    // Wir vergrößern das Array auf eine Größe von 256 Integer
    dynarr = realloc(dynarr, sizeof(int) * 256);
    // Wir haben jetzt mehr Platz!
    dynarr[230] = 11;
    // Die Alten Werte sind noch da
    printf("dynarr[23] = %d\n", dynarr[23]);
    printf("dynarr[47] = %d\n", dynarr[47]);
    // Neuer Index funktioniert auch
    printf("dynarr[230] = %d\n", dynarr[230]);

  } else {
    printf("Etwas ist beim Erzeugen des Arrays schief gelaufen!\n");
  }
  // Wenn wir fertig sind, wird der Speicher des Arrays wieder frei gegeben
  free(dynarr);
}
```

Dynamischer Speicher für Strukturen

- Wir wollen eine Menge an zufällig erzeugten komplexen Zahlen in einem Array speichern, wobei der Speicherplatz im Array nur dann benötigt werden soll wenn an dieser Stelle tatsächlich eine komplexe Zahl gespeichert ist.
- Das Beispiel ist mehr oder weniger äquivalent zu dem am Anfang erwähnten Beispiel bei dem wir geladene 3D Modelle für ein Spiel laden wollen

Übung

Verwendung von dynamischem Speicher für Monster

Erweitern Sie unser Spieleprojekt folgendermaßen:

- Entfernen Sie die zweite Spielfigur da sie aktuell nicht mehr benötigt wird
- Verwenden Sie ein Array von 10 Zeigern auf `Figure` Strukturen welches zu Anfang mit `NULL` initialisiert ist. Wir werden dieses Array verwenden um Monster in unser Spiel zu bringen.
- Bei jedem Schleifendurchlauf soll das folgende passieren:
  - Jeder Platz im Array in dem aktuell noch kein Monster gespeichert ist hat eine Chance von 1:10, dass ein neues Monster mit zufälliger Position erzeugt wird
  - Alle Monster die sich im Array befinden haben eine Chance von 1:10, dass sie sterben und aus dem Array entfernt werden
  - Alle Monster die sich im Array befinden werden gezeichnet