

E 3. Übung

- Besprechung 1. Aufgabe
- Aufgabe 3: malloc
- Debugger gdb

E-1 Aufgabe 1

■ 1. Include, Deklarationen

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

void append_element(int value);
int remove_element(void);

struct listelement {
    int value;
    struct listelement *next;
};

struct listelement *first = NULL;
```

E-1 Aufgabe 1 (2)

■ Anfügen an die Liste

```
void append_element(int value) {
    struct listelement *e;

    if (value < 0) return;

    e = (struct listelement*) malloc(sizeof(struct listelement));
    if (e == NULL) {
        perror("Kann Listenelement nicht anlegen.");
        exit(EXIT_FAILURE);
    }

    e->value = value;
    e->next = NULL;

    if (first == NULL) {
        first = e;
    } else {
        /* Hinweis: man vermeidet das Durchlaufen der Liste, wenn man
           einen Zeiger auf das Listenende vorhaelt.
           Hier aber einfaches Durchlaufen.
        */
        struct listelement *p;
        for(p=first; p->next != NULL; p=p->next);
        p->next = e;
    }
}
```

E-1 Aufgabe 1 (3)

■ Entnehmen aus Liste

```
int remove_element() {
    struct listelement *e;
    int v;
    if (first == NULL) return -1;
    v = first->value;
    e = first;
    first = first->next;
    free(e);
    return v;
}
```

E-1 Aufgabe 1 (4)

- Fehlerbehandlung nicht vergessen!

```
e = (struct listelement*) malloc(sizeof(struct listelement));
if (e == NULL) {
    perror("Kann Listenelement nicht anlegen.");
    exit(EXIT_FAILURE);
}
```

- Fehlermeldungen immer auf `stderr` ausgeben!

z.B. mit `fprintf`

```
fprintf(stderr, "%s(%d): %s\n", __FILE__, __LINE__, strerror(errno));
```

oder mit `perror`

```
perror("Beschreibung wobei");
```

- ...

E-2 Speicherverwaltung mit malloc

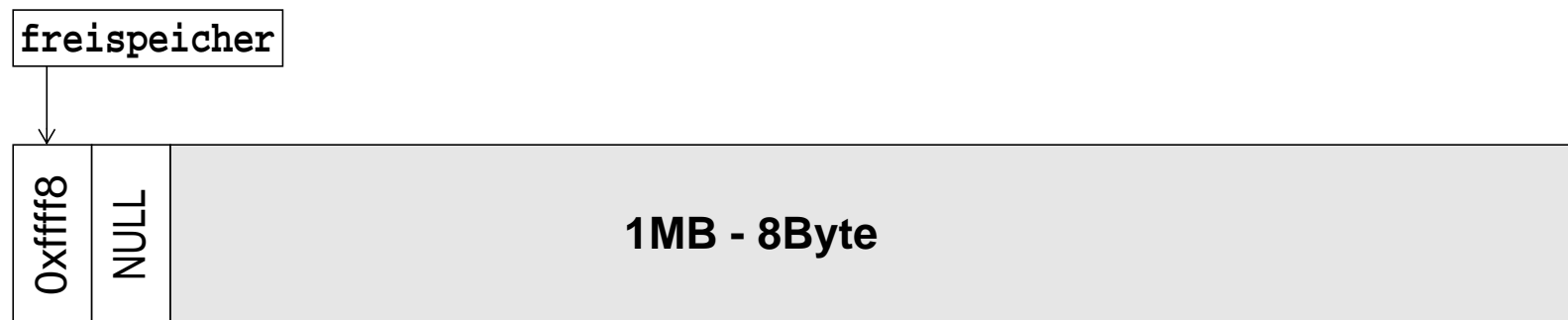
- **malloc(3), calloc(3), realloc(3), free(3):**
Funktionen der Standard-C-Bibliothek zur Anforderung und Freigabe von Speicherbereichen zur Laufzeit (dynamische Speicherverwaltung)
 - ◆ fordern Speicher in größeren Blöcken vom Betriebssystem an
 - durch Verlängerung des Datensegments (Halde)
sbrk(2)
 - typischerweise 8kB-Einheiten
 - ◆ Anforderungen des Anwendungsprogramms werden aus diesem Speicher erfüllt
 - effiziente Implementierung - auch für kleine Speichereinheiten
 - ◆ freigegebenem Speicher (**free(3)**) wird intern verwaltet
 - wird nicht an das Betriebssystem zurückgegeben
 - wird bei folgenden malloc(3)-Aufrufen wieder benutzt
 - nebeneinanderliegende freigegebene Speicherbereiche werden verschmolzen (um Fragmentierung zu vermindern)

E-3 Aufgabe 3: einfache malloc-Implementierung

1 Überblick

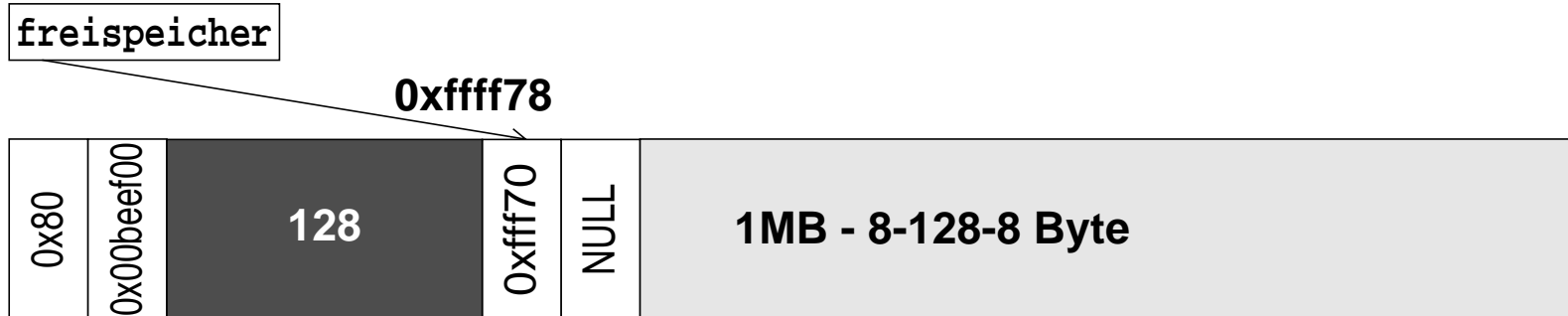
- erheblich vereinfachte Implementierung
 - nur einmal am Anfang Speicher vom Betriebssystem anfordern (1 MB)
 - freigegebener Speicher wird in einer einfachen verketteten Liste verwaltet (benachbarte freie Blöcke werden nicht mehr verschmolzen)
 - `realloc` verlängert den Speicher nicht, sondern wird grundsätzlich auf ein neues `malloc`, `memcpy` und `free` abgebildet

2 initialer Zustand

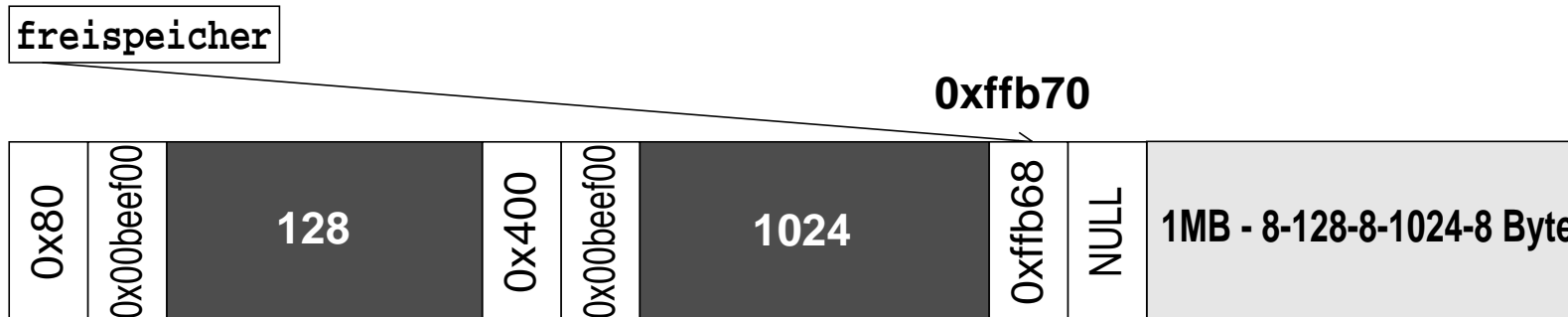


Größe next-Zeiger

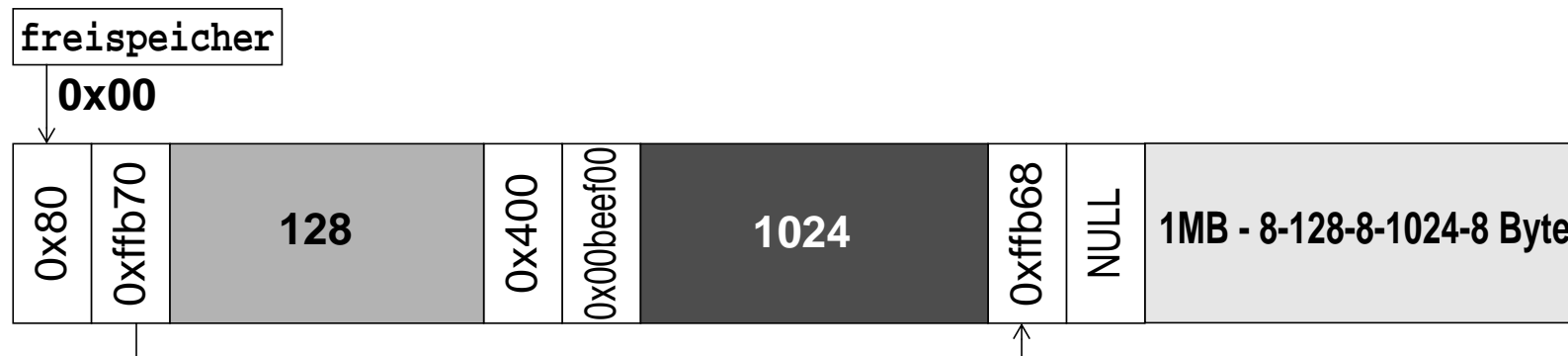
3 Anfordern von 128 (0x80) Byte



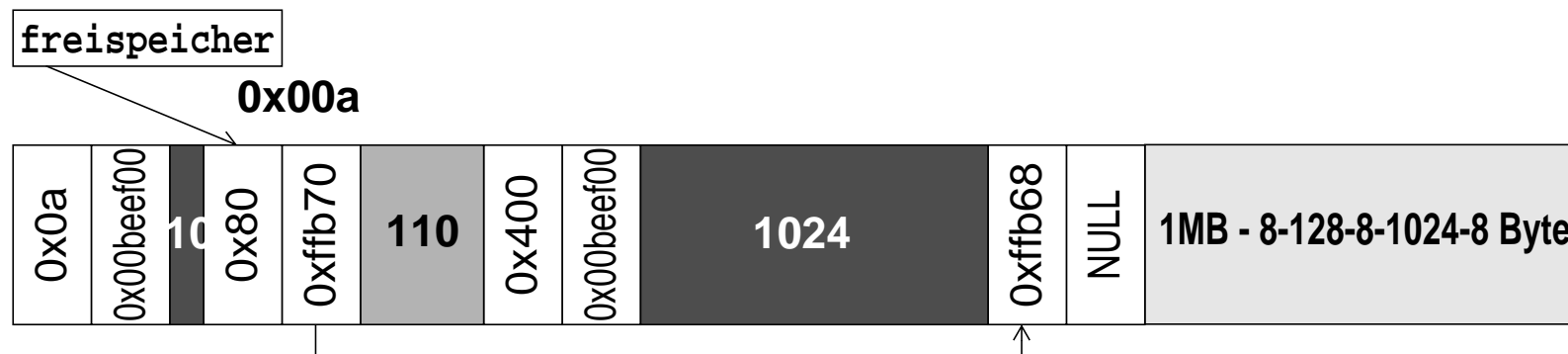
4 Anfordern von 1024 (0x400) Byte



5 Freigabe der ersten 128 Byte



6 Anfordern von 10 Byte



E.4 Debuggen mit dem gdb

- Programm muß mit der Compileroption `-g` übersetzt werden

```
gcc -g -o hello hello.c
```

- Aufruf des Debuggers mit `gdb <Programmname>`

```
gdb hello
```

- im Debugger kann man u.a.
 - ◆ Breakpoints setzen
 - ◆ das Programm schrittweise abarbeiten
 - ◆ Inhalt Variablen und Speicherinhalte ansehen und modifizieren
- Debugger außerdem zur Analyse von core dumps
 - ◆ Erlauben von core dumps:
z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`

1 Breakpoints

- Breakpoints:
 - ◆ `b <Funktionsname>`
 - ◆ `b <Dateiname>:<Zeilennummer>`
 - ◆ Beispiel: Breakpoint bei main-Funktion

```
b main
```

- Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
- Schrittweise Abarbeitung mit
 - ◆ `s` (step: läuft in Funktionen hinein) bzw.
 - ◆ `n` (next: läuft über Funktionsaufrufe ohne in diese hineinzusteppen)
- Fortsetzen bis zum nächsten Breakpoint mit `c` (continue)
- Breakpoint löschen: `delete <breakpoint-nummer>`

2 Variablen, Stack

- Anzeigen von Variablen mit `p <variablenname>`
- Automatische Anzeige von Variablen bei jedem Programmhalt (Breakpoint, Step, ...) mit `display <variablenname>`
- Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks: `bt`

3 Emacs und gdb

- gdb läßt sich am komfortabelsten im Emacs verwenden
- Aufruf mit "**ESC-x gdb**" und bei der Frage "**Run gdb on file:**" das mit der **-g**-Option übersetzte ausführbare File angeben
- Breakpoints lassen sich (nachdem der gdb gestartet wurde) im Buffer setzen, in welchem das C-File bearbeitet wird: **CTRL-x SPACE**