Übungen zu Systemprogrammierung 1 (SP1)

VL 4 – Freispeicherverwaltung

Jens Schedel, Christoph Erhardt, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2012/13 – 19. November bis 23. November 2012

http://www4.cs.fau.de/Lehre/WS12/V_SP1

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

04-halde_handout

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

04-halde_handout

Dynamische Speicherverwaltung

Auszug aus Wikipedia

„Der dynamische Speicher, auch Heap (engl. für ‚Halde', ‚Haufen'), Haldenspeicher oder Freispeicher ist ein Speicherbereich, aus dem zur Laufzeit eines Programms zusammenhängende Speicherabschnitte angefordert und in beliebiger Reihenfolge wieder freigegeben werden können."

- In C
  - Anforderung des Speichers mit Hilfe von `malloc(3)`
    - Parameter: Größe des angeforderten Speichers
    - Rückgabewert: Zeiger auf einen Speicherbereich
  - Explizite Freigabe mit Hilfe von `free(3)`
    - Parameter: Zeiger auf freizugebenden Speicherbereich
    - Rückgabewert: –

04-halde_handout

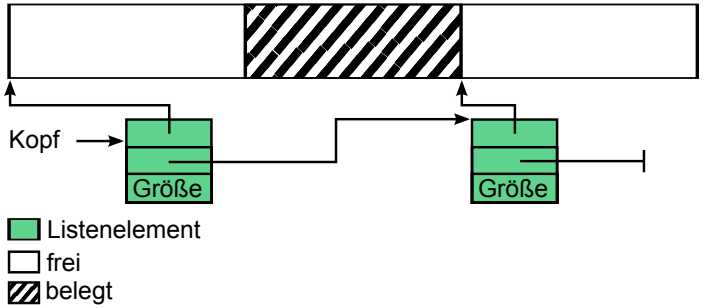
Anforderungsanalyse

- Ziel: Speicherbereiche, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps

  - Welche Informationen muss eine Freispeicherverwaltung bereit halten?
    - Lage aller freien Blöcke
    - für freie Blöcke: Größe und Lage des Speicherbereichs
    - für belegte Blöcke: Größe des Speicherbereichs
  - Welche Datenstruktur ist für eine Freispeicherverwaltung geeignet?
    - KISS (Keep it small and simple): einfach verkettete Liste

Konzept: Verkettete Liste zur Allokation

- Konzept einer Freispeicherverwaltung auf Basis einer verketteten Liste (ohne Berücksichtigung der belegen Blöcke!)

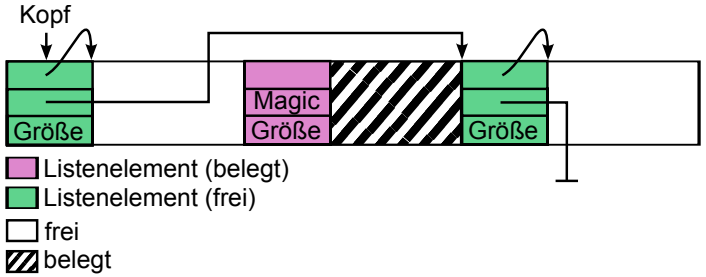


  - Freie Blöcke werden in einer verketteten Liste gespeichert
- Wiederholung Aufgabe 1 (lilo)
  - Wie wird eine verkettete Liste in C implementiert?

    `insertElement()` → `malloc()` → `insertElement()` → `malloc()` → `insertElement()` → `malloc()` → `insertElement()` → `malloc()` → `insertElement()` → `...`

Speicher für die Listenelemente

- Woher den Speicher für die Listenelemente nehmen?



  - Listenelemente werden innerhalb des verwalteten Speichers am Anfang des jeweiligen Speicherbereichs abgelegt
  - Listenelemente auch in belegten Blöcken vorhanden, aber nicht verkettet
    - Verweis auf nächstes Listenelement wird zur Realisierung eines Schutzmechanismusses eingesetzt
    - Abspeichern eines wohldefinierten magischen Wertes und Überprüfung des Wertes vor dem Freigeben

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

Implementierung

- Listenelementdefinition in C

```
typedef struct mblock {
  struct mblock *next; // Zeiger zur Verkettung
  size_t size;         // Größe des Speicherbereichs
  char memory[];       // Zeiger auf Speicherbereich
} mblock;
```

  - Verwendung von FAM (Flexible Array Member):
    - `memory` ist eigentlich ein Feld beliebiger Länge
    - In unserem Fall: `memory` ist ein konstanter „Zeiger" auf das Ende der Struktur
    - `memory` selbst hat die Größe 0

Beispiel auf den Folien

- Schrittweises Abarbeiten des folgenden Codestückes

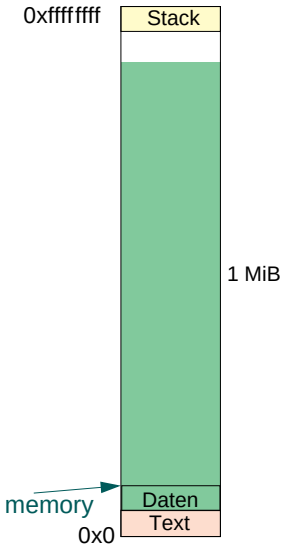
```
char *m1;
char *m2;
char *m3;
...
m1 = (char *) malloc(128);
m2 = (char *) malloc(512*1024);
m3 = (char *) malloc(1024);
...
free(m2);
```

- Annahmen:
  - Freispeicherverwaltung verwaltet 1 MiB statisch allokierten Speicher
  - Verwendung von absoluten Größen (Annahme: 32-Bit Architektur)
    - Größe eines Zeigers: 4 Byte
    - Größe der `struct mblock`: 8 Byte

Halde

- initialer Zustand
  - Speicher statisch allokiert

```
static char memory[1048576];
```



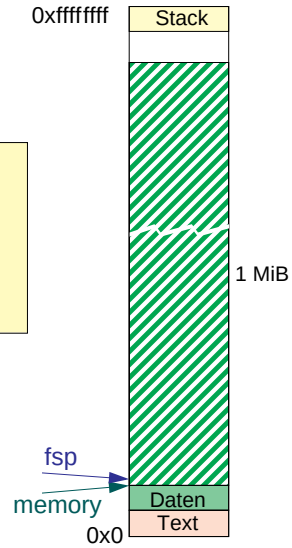
Halde

- initialer Zustand
  - Speicher statisch allokiert

```
static char memory[1048576];
```

  - struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
```



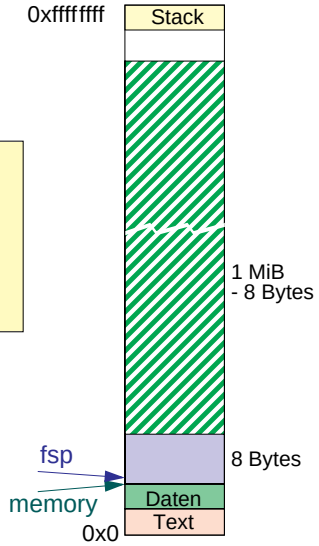
Halde

- initialer Zustand
  - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```

  - ◆ struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
```



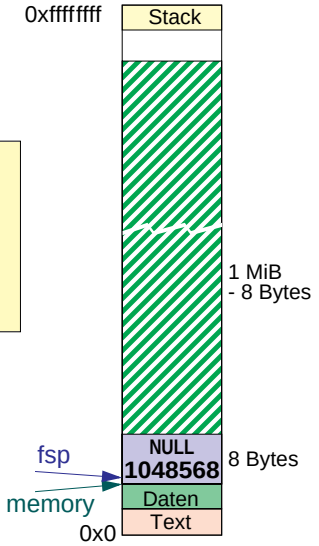
Halde

- initialer Zustand
  - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```

  - ◆ struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
fsp->size = sizeof(memory)-sizeof(mblock);
fsp->next = NULL;
```

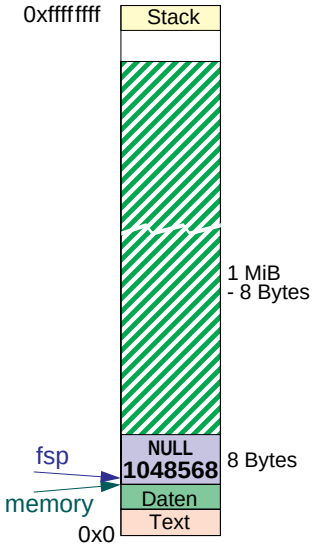


Halde

- initialer Zustand

➥ zwei Zeiger mit unterschiedlichem Typ zeigen auf den gleichen Speicherbereich

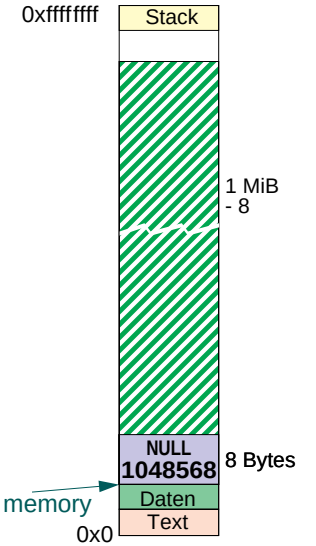
➤ unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

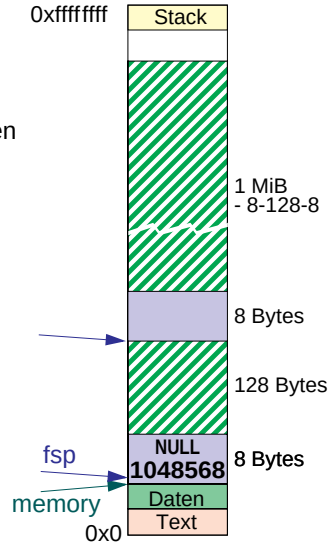


Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

- 128 Bytes hinter dem fsp-mblock reservieren
- neuen mblock dahinter anlegen

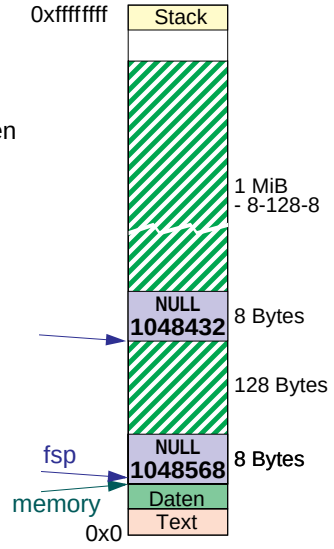


Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

- 128 Bytes hinter dem fsp-mblock reservieren
- neuen mblock dahinter anlegen und initialisieren

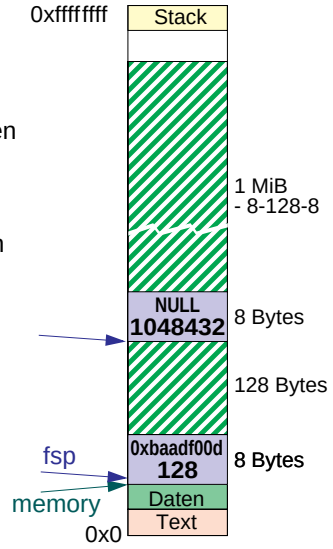


Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

- 128 Bytes hinter dem fsp-mblock reservieren
- neuen mblock dahinter anlegen und initialisieren
- bisherigen fsp-mblock als belegt markieren

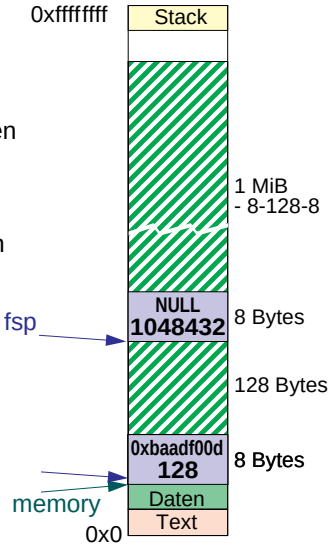


Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

- 128 Bytes hinter dem fsp-mblock reservieren
- neuen mblock dahinter anlegen und initialisieren
- bisherigen fsp-mblock als belegt markieren
- fsp-Zeiger auf neuen mblock setzen

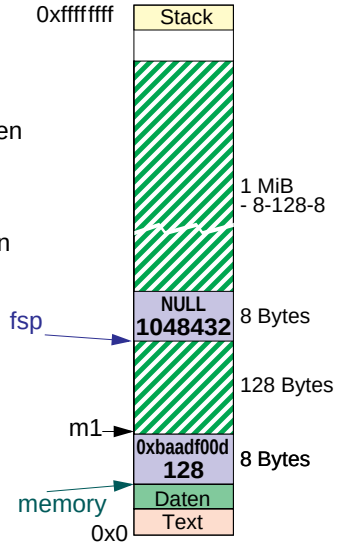


Halde

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *) malloc(128);
```

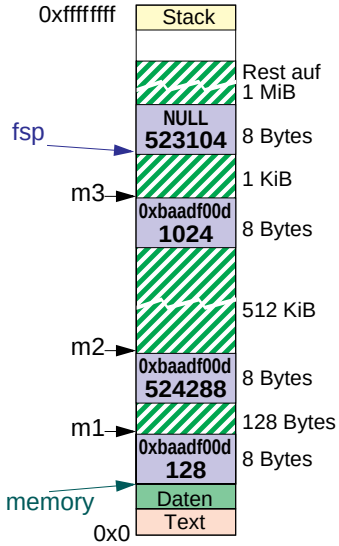
- 128 Bytes hinter dem fsp-mblock reservieren
- neuen mblock dahinter anlegen und initialisieren
- bisherigen fsp-mblock als belegt markieren
- fsp-Zeiger auf neuen mblock setzen
- Zeiger auf die reservierten 128 Bytes zurückgeben



Halde

- Situation nach 3 malloc-Aufrufen

```
...
char *m1,*m2,*m3;
...
m1 = (char *) malloc(128);
m2 = (char *) malloc(512*1024);
m3 = (char *) malloc(1024);
...
free(m2);
```

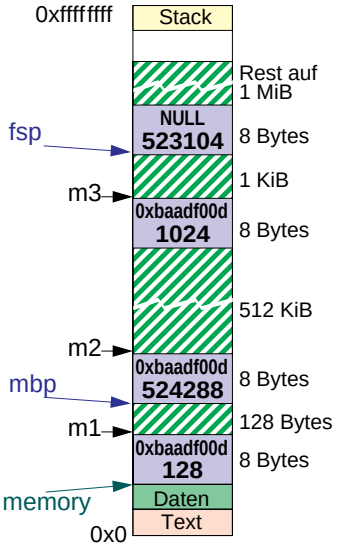


Halde

- Freigabe von m2 - Aufgaben

```
...
char *m1,*m2,*m3;
...
m1 = (char *) malloc(128);
m2 = (char *) malloc(512*1024);
m3 = (char *) malloc(1024);
...
free(m2);
```

- Zeiger `mbp` auf zugehörigen mblock ermitteln
- überprüfen, ob ein gültiger, belegter mblock vorliegt (0xbaadf00d)

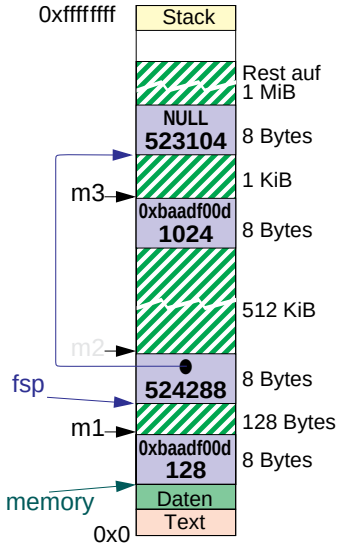


Halde

- Freigabe von m2 - Aufgaben

```
...
char *m1,*m2,*m3;
...
m1 = (char *) malloc(128);
m2 = (char *) malloc(512*1024);
m3 = (char *) malloc(1024);
...
free(m2);
```

- Zeiger `mbp` auf zugehörigen mblock ermitteln
- überprüfen, ob ein gültiger, belegter mblock vorliegt (0xbaadf00d)
- `fsp` auf freigegebenen Block setzen, bisherigen fsp-mblock verketten



Zusammenfassung

- sehr einfache Implementierung - in der Praxis problematisch
  - Speicher wird im Laufe der Zeit stark fragmentiert
    - Suche nach passender Lücke dauert zunehmend länger
    - eventuell keine passende Lücke mehr vorhanden, obwohl insgesamt genug Speicher frei ist
- sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
  - Implementierung erheblich aufwändiger - Resultat aber entsprechend effizienter
  - Strategien werden im Abschnitt Speicherverwaltung in der Vorlesung SP2 behandelt (z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren)

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

Ziele der Aufgabe

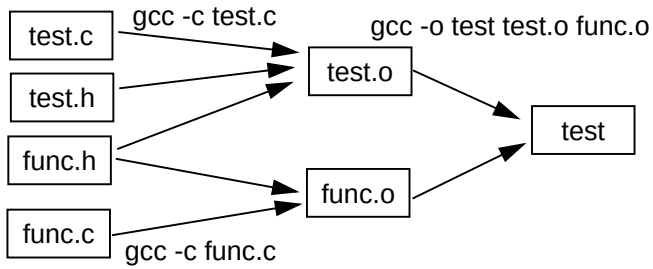
- Ziele der Aufgabe
  - Zusammenhang zwischen „nacktem Speicher" und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Vereinfachungen
  - First-Fit-ähnliche Allokationsstrategie
  - 128 MiB Speicher statisch allokiert
  - freier Speicher wird in einer einfach-verketteten Liste (unsortiert) verwaltet
  - benachbarte freie Blöcke werden nicht verschmolzen
  - `realloc` wird grundsätzlich auf `malloc`, `memcpy` und `free` abgebildet

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

Schrittweises Übersetzen

- Rechner beim Erzeugen von ausführbaren Dateien „entlasten”



- Zwischenprodukte verwenden und somit Übersetzungszeit sparen

Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
  - so deklarierte Targets werden immer gebaut, auch wenn eine gleichnamige Datei bereits existiert, die aktueller als die Abhängigkeiten ist
  - Deklaration als Abhängigkeit des Spezial-Targets `.PHONY` nötig
- Beispiel: Aufräumen mit make clean

```
clean:
  rm -f test.o test
```

- Beispiel: Projekt bauen mit make all

```
all: test
```

  - Konvention: all ist immer erstes Target im Makefile

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
- Aufruf des Debuggers mit `gdb <Programmname>`

Beispiel

```
void initArray(int *array, unsigned int size) {
  int i;
  for ( i=0; i<=size; i++ ) {
    array[i] = 0;
  }
}

int main(int argc, char *argv[]) {
  int *array;
  int buf[8];
  array = buf;

  initArray(buf,8);

  while ( array != buf+8 ) {
    printf("%d\n", *array);
    array++;
  }

  exit(EXIT_SUCCESS);
}
```


Befehlsübersicht

- Programmausführung beeinflussen
  - Breakpoints setzen:
    - `b [<Dateiname>:]<Funktionsname>`
    - `b <Dateiname>:<Zeilennummer>`
  - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - `s` (step: läuft in Funktionen hinein)
    - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: `info breakpoints`
  - Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints

Agenda

4.1 Freispeicherverwaltung
4.2 Implementierung
4.3 Aufgabe 3: halde
4.4 Makefiles – Teil 2
4.5 gdb
4.6 Gelerntes Anwenden

Aktive Mitarbeit!

„Aufgabenstellung“

- Skizzieren Sie den Aufbau des verwalteten Speicherbereichs (hier: 64 Byte, `sizeof(mblock)` = 8 Byte) nach jeden Schritt des jeweiligen Szenarios
  - Szenario 1:

```
char* c1 = malloc(5);
char* c2 = malloc(7);
free(c1);
```

  - Szenario 2:

```
char* c1 = malloc(12);
free(c1);
char* c2 = malloc(4);
```

  - Szenario 3:

```
char* c1 = malloc(26);
char* c2 = malloc(22);
free(c1);
```