Prozedurale Programmierung

Dynamische Speicherverwaltung

Hochschule Rosenheim - University of Applied Sciences

WS 2018/19

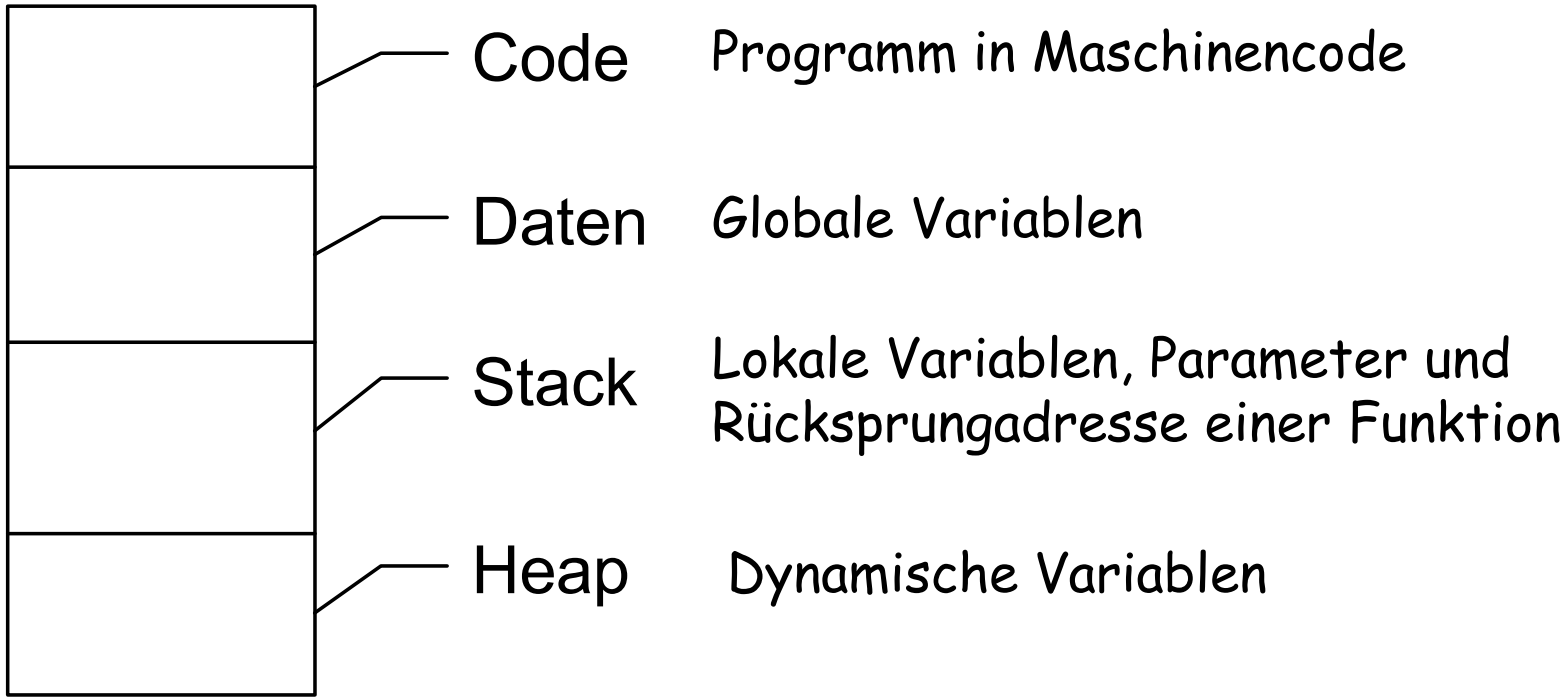
Prof. Dr. F.J. Schmitt

Überblick

- Allgemeines zur Speicherverwaltung
- Anfordern von Speicher
- Verändern der Größe von Speicherblöcken
- Freigeben von Speicher
- Typische Fehler
- Funktionen zur Manipulation von Speicherblöcken

Adressraum eines Programms

- Adressraum eines ablauffähigen Programms besteht aus vier verschiedenen Bereichen:



Speicherbereiche

Drei Speicherbereiche für Variablen:

- Statischer Speicher
  - Teil des ausführbaren Programms
  - Variablen außerhalb von Funktionen (global)
  - Statische Variablen innerhalb von Funktionen
  - Größe ändert sich während der Laufzeit nicht
- Automatischer Speicher (Stack, Keller)
  - Automatische Variablen (lokal)
  - Aufrufparameter von Funktionen
  - Pulsiert während des Programmablaufs
- Freier Speicher (Heap, Halde)
  - Wird vom Programmierer über Bibliotheksfunktionen selbst verwaltet (dynamisch)

Dynamische Speicherverwaltung

- Vereinbarung eines Felds
  - Ein konstanter Ausdruck für dessen Länge muss angegeben werden (starre Feldlänge)
  - Maximale Anzahl der Elemente ist beschränkt

- Was wird gemacht, wenn zur Zeit der Kompilierung noch nicht bekannt ist wie viele Elemente bzw. Speicherplatz wirklich benötigt wird?
  - Reservierung und Freigabe des entsprechenden Speicherplatz zur Laufzeit notwendig
  - Dynamische Speicherverwaltung mit speziellen Bibliotheksfunktionen (`#include <stdlib.h>`)

Anfordern von Speicher (1)

- Funktion `malloc` („memory allocate“)
  - Beliebiger (zusammenhängender) Speicherblock kann angefordert werden
  - Parameter: Länge des Speichers in Byte
  - Rückgabewert:
    - Adresse des Speicherblocks (Anfang) oder
    - Nullzeiger, falls nicht genügend Speicher verfügbar war
  - Angeforderter Speicher muss auch wieder freigegeben werden

- Prototyp:

```
void *malloc(size_t groesse);
```

Anfordern von Speicher (2)

- Beispiel: Speicher für eine Variable eines bestimmten Datentyps

```
datentyp *zeiger = NULL;
//...

zeiger = (datentyp*)malloc(sizeof(datentyp));

if (zeiger != NULL)
{
   // alles ok
}
else
   // Fehlerbehandlung
```

Anfordern von Speicher (3)

Beispiel: Anfordern eines Felds von Variablen eines bestimmten Datentyps

```
datentyp *feldzeiger = NULL;
int laenge = 100;
//...

feldzeiger = (datentyp*)malloc(laenge * sizeof(datentyp));

if (feldzeiger != NULL)
{
   // alles ok
}
else
   // Fehlerbehandlung
```

Anfordern von Speicher (4)

Beispiel: Anfordern Speicher für ein Feld von Strukturen

```
struct Adresse_s *db = NULL;
int laenge = 100;
//...

db = (struct Adresse_s *)
     malloc(laenge * sizeof(struct Adresse_s));

if (db != NULL)
{
   // alles ok
}
else
   // Fehlerbehandlung
```

Anfordern von Speicher (5)

- Funktion `calloc` („clear allocate“)
  - Ähnlich wie `malloc`, jedoch wird der allozierte Speicher mit 0 vorinitialisiert
  - Zwei Parameter: Anzahl der Elemente des zu allozierenden Feldes und die Größe eines Elements in Byte
  - Rückgabewert:
    - Adresse des Speicherblocks (Anfang) oder
    - Nullzeiger, falls nicht genügend Speicher verfügbar war

- Prototyp:

```
void *calloc(size_t anzahl, size_t groesse);
```

Anfordern von Speicher (6)

- Beispiel: Anfordern eines mit 0 initialisierten Speicherbereichs

```
struct Adresse_s *db = NULL;
int laenge = 100;
//...

db = (struct Adresse_s *)
       calloc(laenge, sizeof(struct Adresse_s));

if (db != NULL)
{
   // alles ok
}
else
   // Fehlerbehandlung
```

Verändern der Größe von Speicherblöcken (1)

- Funktion `realloc`
  - reicht bspw. die Länge eines mit `malloc` oder `calloc` allozierten Felds nicht aus, kann Speicherblock verlängert werden
  - Parameter: Zeiger auf einen bereits existierenden dynamischen Speicherbereich und Größe des gewünschten neuen Speicherbereichs
  - Rückgabewert:
    - Zeiger auf den reservierten Speicherbereich (falls erfolgreich) oder
    - Nullzeiger sonst

- Prototyp:

```
void *realloc(void *zeiger, size_t groesse);
```

Verändern der Größe von Speicherblöcken (2)

- Beispiel:

```
datentyp *neuerBlock = NULL;
datentyp *alterBlock = NULL;

...

neuerBlock = (datentyp *)
                realloc(alterBlock, neueLaenge);
if (neuerBlock != NULL)
{
   // ...
}
else
   // Fehlerbehandlung
```

Verändern der Größe von Speicherblöcken (3)

```
long laenge;
struct Adresse_s *db = NULL;
// … benötigte Länge ermitteln

db = (struct Adresse_s *) malloc(laenge * sizeof(struct Adresse_s));
if (db ==  NULL) //Fehlerbehandlung
{
  printf("Speicheranforderung fehlgeschlagen!\n");
  exit(1);  // Programm beenden, Fehlercode 1
}
// … Verwenden von db

//Verdoppeln der Feldlänge
laenge = 2 * laenge;
db = (struct Adresse_s *) realloc(db, laenge * sizeof(struct Adresse_s));
if (db ==  NULL) //Fehlerbehandlung
{
  printf("Speichervergroesserung fehlgeschlagen!\n");
  exit(2);  // Programm beenden, Fehlercode 2
}

//...
```

Freigeben von Speicher (1)

- Funktion `free`
  - Nicht mehr benötigter dynamischer Speicher sollte/muss wieder freigegeben werden
  - Parameter: Adresse des Speicherblocks, der von `malloc`, `calloc` oder `realloc` zurückgegeben wurde

- Prototyp:

```
void free (void *zeiger);
```

```
free(db);  // Freigabe des Speichers db
```

Freigeben von Speicher (2)

- Beachte:
  - Auf Speicher, der bereits freigegeben wurde, darf nicht mehr zugegriffen werden!
  - Speicher darf nicht zweimal freigegeben werden !
  - Daher:
    - vor free() prüfen, ob Zeiger gültig ist
    - nach Freigabe Zeiger auf NULL setzen

```
if (db != NULL)    // Zeiger gültig?
 {
    free(db);      // Freigabe des Speichers
    db = NULL;
 }
```

  - Gewissenhafte Planung, wann angeforderter Speicher wieder freigegeben werden kann

Beispiel (1)

Was wird erzeugt?

```
double farr[10];
```

- Feld (Array) von 10 doubles: farr[0] … farr[9]
- Feld mit fester Länge:
  Länge steht zum Kompilierungszeitpunkt fest

Beispiel (2)

Was wird erzeugt?

```
int n = 27;
//...
double *varr = (double*) calloc(n, sizeof(double));

varr[5] = 7.2;

//...

varr = (double*) realloc(varr, sizeof(double) * 2 * n);
```

- Feld mit variabler Länge: Länge wird zur Laufzeit festgelegt und kann geändert werden

Beispiel (3)

- Beispiel: Definition lokaler Variablen
  String mit fester Länge 5

```
char s[5] = { '0', 't', 't', 'o', '\0' };

char s[5] = "otto";
char s[]   = "otto";
```

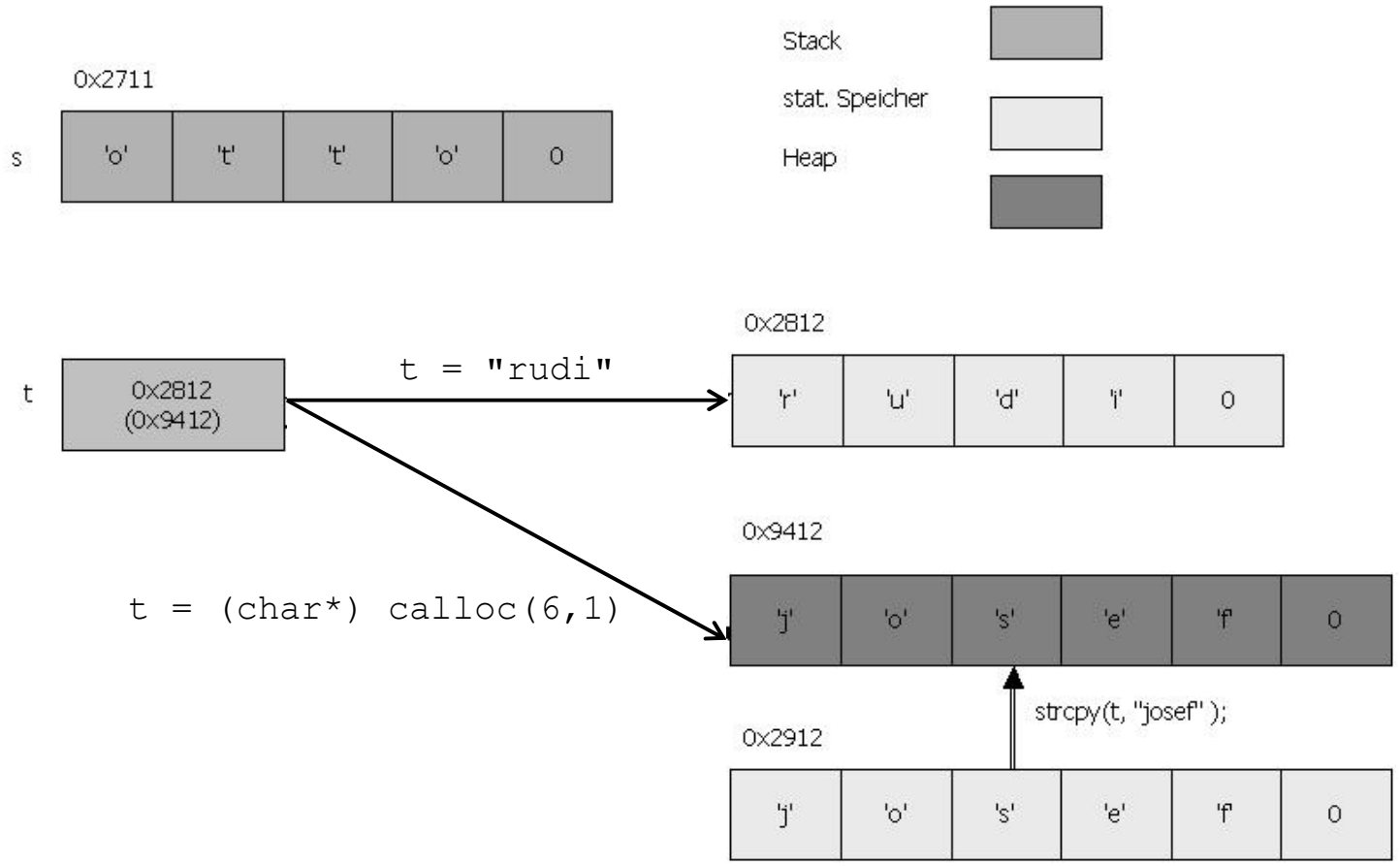
Nullterminiertes Feld mit 4+1 Zeichen

- Beispiel: Definition eines Strings mit variabler Länge

```
char *t    = "rudi";  // t zeigt auf Speicher mit 5 Zeichen

t = (char*) calloc(6, sizeof(char));
                      // legt neuen Speicher mit 6 Zeichen an
strcpy(t, "josef");
```

Beispiel (4)


Stack
stat. Speicher
Heap
0x2711
s
'o'
't'
't'
'o'
0
t
0x2812
(0x9412)
t = "rudi"
0x2812
'r'
'u'
'd'
'i'
0
0x9412
t = (char*) calloc(6,1)
'j'
'o'
's'
'e'
'f'
0
strcpy(t, "josef");
0x2912
'j'
'o'
's'
'e'
'f'
0

Typische Fehler

- es wird vergessen Speicher zu allozieren
- Speicher wird nach der Freigabe noch benutzt
- allozierter Speicher wird mehrfach freigegeben
- allozierter Speicher wird nicht freigegeben
- Speicher wird in zu großen Portionen alloziert
- Speicher wird in zu kleinen Portionen alloziert

Weitere Funktionen

- Kopieren von Speicher
  `memcpy()`
- Verschieben von Speicher
  `memmove()`
- Byteweiser Vergleich
  `memcmp()`
- Byteweise Initialisierung
  `memset()`

`#include <string.h>` notwendig

memcpy()

- Prototyp:
  ```
  void *memcpy(void *dest, const void *src, size_t count);
  ```

- kopiert `count` Byte des Speicherinhalts von Adresse `src` nach Adresse `dest`
- die Speicherbereiche dürfen sich ***nicht*** überlappen!
  - in diesem Fall muss `memmove()` verwendet werden
- Speicherbereich `dest` >= Speicherbereich `src` muss sichergestellt sein

memcpy() - Beispiele

```
// Zuweisung von Arrays
int z1[20], z2[20];
// …
memcpy(z2, z1, 20 * sizeof(int));
```

```
// z2 = z1;
int z1 = 1, z2 = 2;

memcpy(&z2, &z1, sizeof(int));
```

```
long laenge = 20;
struct Adresse_s *db = NULL;
struct Adresse_s *dbCpy = NULL;

db = (struct Adresse_s *) malloc(laenge * sizeof(struct Adresse_s));
if (db ==  NULL) …; //Fehlerbehandlung

// db mit Daten füllen

dbCpy = (struct Adresse_s *) malloc(laenge * sizeof(struct Adresse_s));
if (dbCpy ==  NULL) …; //Fehlerbehandlung

memcpy(dbCpy, db, sizeof(laenge * struct Adresse_s));
```

memmove()

- Prototyp:
```
void *memmove(void *dest, const void *src, size_t count);
```

- Verwendung wie `memcpy()`
- kopiert `count` Byte des Speicherinhalts von Adresse `src` nach Adresse `dest`
- die Speicherbereiche dürfen sich überlappen
- Speicherbereich `dest` >= Speicherbereich `src` muss sichergestellt sein

memcmp()

- Prototyp:

```
int memcmp(const void *buf1, const void *buf2, size_t count);
```

  - vergleicht `count` Byte des Speicherinhalts von Adresse `buf1` mit dem an Adresse `buf2`
  - der Vergleich erfolgt byteweise!
  - Rückgabewert `0` wenn die Speicherbereiche gleich sind

- Beispiel: Vergleich von zwei Feldern

```
int z1[20], z2[20];
int ret;
// …
ret = memcmp(z1, z2, 20 * sizeof(int));

if( ret == 0)
  printf("z1 und z2 sind gleich\n");
else
 printf("z1 und z2 sind nicht gleich\n");
```

memset()

- Prototyp:

```
void *memset(void *dest, int c, size_t count);
```

  - initialisiert `count` Byte des Speicherinhalts von Adresse `dest` mit dem Wert `c`
  - Achtung: die Initialisierung erfolgt byteweise!
    - `c` muss zwischen 0 und 255 liegen

- Beispiel: Initialisiere Feld

```
int z[20];
char c[20];

memset(z, 0, 20 * sizeof(int));    // init z mit 0

memset(c, 0, 20 * sizeof(char));   // init c mit 0

memset(c, 'A', 20 * sizeof(char)); // init c mit 'A'
```

Aufgabe

- Schreiben Sie ein C-Programm, das im Heap eine Strukturvariable vom Typ `struct point_s` mit den Komponenten `x` und `y` mit den Werten `1.5` und `4.5` anlegt.
  Geben Sie anschließend die beiden Komponenten am Bildschirm aus.

Zusammenfassung

- Speicherbereiche
  - Code- und Datensegment, Stack, Heap
- Anlegen von Speicher
  - malloc()
  - calloc()
  - realloc()
- Freigeben von Speicher
  - free()
- Funktionen zum Kopieren, Vergleichen, Initialisieren von Speicherbereichen
  - memcpy(), memmove(), memcmp(), memset()