SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

Speicherverwaltung, 0, const

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Speicherverwaltung

§7.20.3

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Speicherverwaltung kein Teil der Sprache, wird mittels Bibliotheksfunktionen realisiert
- Damit sind Reihungen mit dynamischer Größe realisierbar
- Benutzung über Zeiger, Größe muss manuell verwaltet werden
- Deklarationen in `<stdlib.h>`

malloc()

§7.20.3.3

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

- Fordert `n` Byte Speicher an
- Bei Fehlschlag: Nullzeiger
- Rückgabwert testen, niemals `assert()` verwenden!
- Speicher ist uninitialisiert

Beispiel: malloc()


```
XY* p = malloc(sizeof(*p)); // Ein XY-Objekt
XY* p = malloc(sizeof(XY)); /* Aequivalent, aber
                             * fehleranfaellig */

XY* a = malloc(sizeof(*a) * 10); // 10 XY-Objekte
a[3].x = 23;          // Setze x von 4tem Objekt
(*(a + 3)).x = 23; // Dasselbe mit Zeigerarithmetik
(a + 3)->.x  = 23;
```

memset()

§7.21.6.1


```
void* memset(void* p, int c, size_t n);
```

- Beschreibt `n` Byte beginnend bei `p` mit Byte-Wert von `c`
  → Typ `int` von `c` historisch bedingt
- Häufigste Verwendung: `c = 0`
- Gibt `p` zurück
- Achtung: Nullbytes nicht unbedingt korrekte Darstellung für Nullzeiger
  → In der Praxis heutzutage doch

```
struct Node {
  struct Node* prev;
  struct Node* next;
} n;
memset(&n, 0, sizeof(n)); // Eigentlich nicht richtig
```

calloc()

§7.20.3.1


```
void* calloc(size_t n, size_t size);
```

- Fordert Speicher für `n` Objekte der Größe `size` an
  $\longrightarrow$ Schlicht `n * size`
- Speicher ist ausgenullt
- Fast äquivalent zu:

```
memset(malloc(n * size), 0, n * size)
```

calloc()

§7.20.3.1

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void* calloc(size_t n, size_t size);
```

- Fordert Speicher für `n` Objekte der Größe `size` an
  $\longrightarrow$ Schlicht `n * size`
- Speicher ist ausgenullt
- Fast äquivalent zu:

```
memset(malloc(n * size), 0, n * size)
```

  - Test auf Fehlschlag nach `malloc()`
  - Test auf Überlauf von `n * size`

free()

§7.20.3.2


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

- Gibt Speicher frei, der mittels `malloc()`, `calloc()` oder `realloc()` beschafft wurde
- Zeiger muss unverändert sein
- Speicher darf nicht mehrfach freigegeben werden
- Muss manuell erledigt werden, sonst Speicherleck
  → Bessere Mittel in C++ (später)
- Nullzeiger als Argument erlaubt (tut nichts)

realloc()

§7.20.3.4

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void* realloc(void* p, size_t size);
```

- Vergrößert/verkleinert Speicherblock, auf den `p` zeigt auf `size` Bytes
- Beim Vergrößern: Zusätzlicher Speicher uninitialisiert
- Gibt eventuell anderen Zeiger zurück
  $\longrightarrow$ Kopiert Inhalt bei Bedarf
- Achtung: Alter Zeiger darf in keiner Weise mehr verwendet werden!
  $\longrightarrow$ Falle: In alten Block zeigende Zeiger in jedem Fall ungültig!
- Bei Fehlschlag: Nullzeiger, dann alter Zeiger weiterhin gültig
  $\longrightarrow$ Nicht vergessen Speicher freizugeben

Mehrdimensionale Speicherallokation

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
// Eine feste Dimension: n x 10-Reihung
int (*p)[10] = malloc(sizeof(*p) * n);
p[3][7] = 42;

// Keine feste Dimension: a x b-Reihung
int* q = malloc(sizeof(*q) * a * b);
// Adressrechnung muss manuell vorgenommen werden
q[y * b + x] = 23;
```

Flexible Array Member

§6.7.2.1:16

```
typedef struct X {
  unsigned count;
  char     data[];
} X;

X* p = malloc(sizeof(*p) + sizeof(*p->data) * n);
p->count   = n;
p->data[0] = 23;
```

- Spezialfall: Reihung mit unbekannter Länge (`[]`) am Ende eines Verbunds erlaubt
- Mittels `malloc()` Kann Objekt mit beliebig vielen Elementen erzeugt werden
- `sizeof` liefert nur Größe bis zur Reihung am Ende
  $\longrightarrow$ Auch eventuelle Füllung

Nullzeiger

- Ein Zeiger, der auf nichts zeigt
- Ungültiger Zeiger
  $\longrightarrow$ Darf nicht dereferenziert werden
- Spezieller ungültiger Zeiger
  $\longrightarrow$ Der einzige ungültige Zeiger, der verglichen werden darf
- Andere ungültige Zeiger z.B. mittels ungültiger Zeigerarithmetik erzeugbar oder Zeiger nach Aufruf von `free()`
  $\longrightarrow$ Test mit diesen ist undefiniert
- Ist damit die einzige Art ungültiger Zeiger, die teilweise definiertes Verhalten hat

0


```
0               // Nullzeigerkonstante
(void*)0        // Nullzeigerkonstante
23 / 42         // ebenso
int* p = 0; // p ist ein Nullzeiger
if (p != 0) // Vergleich mit Nullzeigerkonstante
if (p)          /* Impliziter Vergleich mit
                 * Nullzeigerkonstante */
```

- Jeder ganzzahlige konstante Ausdruck mit dem Wert `0` ist eine Nullzeigerkonstante
  → Oder solch ein Ausdruck umgewandelt nach `(void*)` (nicht C++)
- Nullzeigerkonstanten werden in den meisten Situationen in Nullzeiger umgewandelt
  → z.B. bei Vergleich mit Zeigern
  → Nicht bei variadischen Parametern (später)
  → Überladene Funktionen ebenso problematisch (C++)
- Nullzeiger bestehen nicht notwendigerweise nur aus Nullbits (heutzutage zumeist doch)

NULL

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#define NULL 0
#define NULL (void*)0
#define NULL 0L
```

- Ist ein Makro, definiert in `<stddef.h>` (und einigen anderen, z.B. `<stdlib.h>`)
- Evaluiert zu einer implementierungsabhängigen Nullzeigerkonstante
  $\longrightarrow$ Kein Nullzeiger

Nullzeigerkonstante und C++

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void f(int);
void f(char*);

#define NULL 0 // Nullzeigerkonstante fuer C++
f(NULL);
f(0); // Nach Makroexpansion
```

- In C++ darf Nullzeigerkonstante nicht Zeigertyp haben
  $\longrightarrow$ C++ verbietet implizites Umwandeln von `void*` in andere Zeigertypen
- (Diese) Nullzeigerkonstante hat Typ `int`
  $\longrightarrow$ Achtung: `f(int)` wird aufgerufen

Nullzeigerkonstante[1]

```
    !       !  !      !  !       !
  !  !    !   !      !  !       !
 !    !  !    !      !  !       !
!       !     !!!!!!  !!!!!  !!!!1;
```

[1] Aus „The C++ Programming Language“

Alle Sorten von Null

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Nullzeiger
- Nullzeigerkonstanten (0)
- NUL-Zeichen (`'\0'`)
  $\longrightarrow$ Ist auch eine gültige Nullzeigerkonstante, ansonsten siehe Zeichenketten

const

- `const` an Objekt: Objekt unveränderbar
- `const` an referenziertem Typ: Referenziertes Objekt durch diesen Zeiger nicht veränderbar
  $\longrightarrow$ Veränderungen können aber beobachtet werden
- Regel: `const` bezieht sich auf direkt linkes Element, außer wenn es ganz links steht, dann rechts

```
int                 i;     // Veraenderbarer int
int const           i;     // Unveraenderbarer int
const int           i;     // s.o.
int* const          pi;    /* Unveraenderbarer Zeiger auf
                               * aenderbaren int */
int const*          pci;   /* Veraenderbarer Zeiger auf
                               * unveraenderbaren int */
const int*          pci;   // s.o.
int const* const    cpci;  /* Unveraenderbarer Zeiger auf
                               * unveraenderbaren int */
```

const und Zugriffspfade

Achtung: `const` ist kein Schutz vor Änderung über andere Zugriffspfade!

```
int         i = 42;
int const* p = &i;
// *p ist 42
i = 23;
// *p ist jetzt 23!
```

const und Zuweisung/Initialisierung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int const      ci;
int const*   pci = &ci;
int       *   pi;
int       ** ppi  = &pi;
int const** ppci = ppi;
*ppci = pci;
```

Nach der letzten Zeile zeigt pi (Zeiger auf `int`) auf einen `int const`.
Wo liegt der Fehler?

const und Zuweisung/Initialisierung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int const      ci;
int const*   pci = &ci;
int       *   pi;
int       ** ppi  = &pi;
int const** ppci = ppi;
*ppci = pci;
```

Nach der letzten Zeile zeigt pi (Zeiger auf `int`) auf einen `int const`.
Wo liegt der Fehler?
$\rightarrow$ ppci = ppi ist nicht zulässig!

const und Zuweisung/Initialisierung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Außen: hinzufügen und entfernen

```
int const a = 0;
int       b = a;
int const c = b; // Nur lokale Variablen
```

- Erste Zeigerstufe: hinzufügen

```
int         i;
int*        pi  = &i;
int const*  pci = pi;  // Oder direkt &i
int*        pi2 = pci; // Fehler!
```

- Sonst nicht möglich

```
int**        ppi;
int const**  ppci = ppi;  // Fehler!
int**        ppi2 = ppci; // Fehler!
```

Zeiger: Probleme

- Unsachgemäße Handhabung führt zu teilweise schwer auffindbaren Fehlern
- Speicherlecks
- Zugriff außerhalb von gültigen Bereichen
- Zugriff auf bereits freigegebenen Speicher

Zugriff außerhalb gültigen Bereichs

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
int main(int argc, int** argv)
{
  int a[10];
  int idx = atoi(argv[1]); /* Wandelt Zeichenkette in
                            * Zahl um */
  a[idx] = 23; // Ungeprueft!
}


int* p = malloc(sizeof(*p) * n);
p[n] = 23; // FEHLER: Ungueltiger Index!
```

- Benutzereingabe immer prüfen
  → Tastatur, Dateien, Netzwerk
- Interne Konsistenz mittels `assert()` sicherstellen
  → Nur für interne Fehler, niemals für Eingaben

Zugriff außerhalb gültigen Bereichs

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
int main(int argc, int** argv)
{
  int a[10];
  int idx = atoi(argv[1]); /* Wandelt Zeichenkette in
                            * Zahl um */
  a[idx] = 23; // Ungeprueft!
}


int* p = malloc(sizeof(*p) * n);
p[n] = 23; // FEHLER: Ungueltiger Index!
```

- Benutzereingabe immer prüfen
  → Tastatur, Dateien, Netzwerk
- Interne Konsistenz mittels `assert()` sicherstellen
  → Nur für interne Fehler, niemals für Eingaben

Weitere Fehler:

Zugriff außerhalb gültigen Bereichs

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
int main(int argc, int** argv)
{
  int a[10];
  int idx = atoi(argv[1]); /* Wandelt Zeichenkette in
                            * Zahl um */
  a[idx] = 23; // Ungeprueft!
}


int* p = malloc(sizeof(*p) * n);
p[n] = 23; // FEHLER: Ungueltiger Index!
```

- Benutzereingabe immer prüfen
  → Tastatur, Dateien, Netzwerk
- Interne Konsistenz mittels `assert()` sicherstellen
  → Nur für interne Fehler, niemals für Eingaben

Weitere Fehler:

- Fehlender Test ob `malloc()` fehlschlug
- Möglicher Überlauf bei Größenberechnung

Benutzung nach Freigabe

Use after free

```
typedef struct X X;
struct X { int data; X* next };

void free_list(X* p)
{
  while (p) {
    free(p);
    p = p->next; // FEHLER: Benutzung nach Freigabe!
  }
}

void free_list(X* p)
{
  while (p) {
    X* next = p->next; // Richtig
    free(p);
    p = next;
  }
}
```