Sichere Strings und Speicher in C

Pablo Yánez Trujillo

Poolmanager Team
Universität Freiburg

7. August, 2006

- ANSI C ist die Spezifikation der Sprache C
- ANSI C ist eine vom ANSI Komitee entworfene Norm der Sprache C
- Jeder C Compiler **sollte** sich daran halten und ANSI C Code übersetzen
- Im Kurs verwendeter C Compiler: GNU GCC

- Die Meinungen über C gehen auseinander. Viele halten C für veraltet und unbrauchbar
- C wird nicht in kürzer Zeit aussterben (trotz Vormarsch von C++, Java, C#)
- System- und Hardwareprogrammierung findet fast auschließlich in C statt
- C ermöglich sehr nah am Speicher zu arbeiten $\Longrightarrow$ viel Macht verknüpft mit viel Verantwortung

- Eine Einführung für C Einsteiger, speziell für Hardware Programmierer
- Richtiger Umgang mit dem Speicher, speziell mit dem dynamischen Speicher
- Dazu gehört auch der Umgang mit den C Strings, die meistens auf den dynamischen Speicher aufbauen
- Am Ende des Kurses sollen wir eine kleine Bibliothek schreiben, die sicher mit den C Strings umgehen

- Die C Syntax ist einfach (C++/Java ähnlich)
- Die Sprache enthält:
  - Data structures (primitive, Felder, Zeiger, zusammengesetzte)
  - Funktionen
  - Operatoren (+,-,*,/,%)
  - Kontrollstrukturen (`for`,`if`,`while`,`switch`,`do`)

- primitive: char,int, long,double, float
- Felder: Variable, die mehrere Werte speichern kann
- Zeiger: Variable, deren Inhalt eine Adresse im Speicher ist
- zusammengesetzte: structs

Deklaration von Variablen

```
int anzahl; /* Variable vom Typ Integer */
int feld[3]; /* Feld vom Typ Integer */
int *ptr1; /* Zeiger auf eine int Variable */
int *ptr2 = &anzahl; /* Zeiger auf anzahl */
struct map { /* struct mit int und char */
    char *name;
    int wert;
};
struct map a,b; /* mehrere Variablen von Typ struct
map */
```

- Sie helfen, den Code zu strukturieren und in kurzen Abschnitten zu teilen
- Programmier können davon profitieren, z.B. in Bibliotheken
- Der Code ist einfacher zu lesen und es ist einfacher nach Fehlern zu suchen
- Wenn man sieht, dass ein Teilcode immer wieder vorkommt, dann kann man es in eine Funktion packen.

- Der Typ einer Funktion ist derselbe wie der Typ des Rückgabewertes
- Wenn eine Funktion keinen Wert zurückgibt, dann ist sie von Typ `void`
- `void` Variablen gibt es nicht, `void*` Zeiger dagegen schon (später behandelt)

type **function-name**([Parameter...]) *code*

Beispiel 1: Additionsfunktion

```
int addition(int a, int b)
{
    return a+b;
}
⋮
void foo()
{
    int a,b,c;
    a = 8; b = 10;
    c = addition(a,b);
}
```

Das "Hello World" Programm

Das "Hello World" Programm

```c
#include <stdio.h>

int main()
{
    printf("Hello, world\n");
    return 0;
}
```

- Kontrollstrukturen helfen dabei, das Programm zu steuern
- Das Verhalten hängt von den Abfragen nach Gleichheit, Schleifen, usw. ab
- if,switch Abfragen: Prüfe nach Gleichheit
- for,do,while Schleifen: führe eine Schleife solange eine Bedingung erfüllt ist

Bedingungen

- `(a == b)` Inhalt von a ist gleich dem Inhalt von b
- `(a < b)` Inhalt von a ist kleiner als Inhalt von b
- `(a > b)` Inhalt von a ist größer als Inhalt von b
- `(a <= b)` Inhalt von a ist kleiner oder gleich dem Inhalt von b
- `(a >= b)` Inhalt von a ist größer oder gleich dem Inhalt von b
- `(a != b)` Inhalt von a ist ungleich dem Inhalt von b

Achtung: C hat nativ kein `true` oder `false`. In C ist `false` 0 und `true` etwas, was nicht 0 ist. Wenn eine Bedingung erfüllt ist, so ist der Rückgabewert nicht 0.

Bedingungen: logisch verknüpfen

- *bed1* && *bed2* UND-Verknüpfung
- *bed1* || *bed2* OR-Verknüpfung
- !*bed* NOT Negation

Wenn eine Teilbedingung wahr ist, dann wird bei OR-Verknüpfungen nicht weiter getestet. Analog mit der UND-Verküpfung

IF Abfrage

if(*Bedingung*) *code*

IF Abfrage

```
int abs(int n)
{
    if(n<0)
        return -n;
    else
        return n;
}
```

SWITCH Abfrage

switch(*variable*) **case** *value*: *code* **break;**

SWITCH Abfrage

```
void print_name(int val)
{
    switch(val)
    {
        case 0:  printf("Pablo\n");
                 break;
        case 1:
        case 2:
        case 3:  printf("Michael\n");
                 break;
        default:  printf("Unbekannt\n");
    }
}
```

FOR Schleife

for(*Startzuweisung*; *Bedingung* ; *Aktion*) *code*

FOR Schleife

```
int n;
for(n=0; n<10; ++n)
    printf("%d ", n);

printf("\n");

/* Ausgabe:  0 1 2 3 4 5 6 7 8 9 */
```

WHILE Schleife

while(*Bedingung*) *code*

WHILE Schleife

```
#include <stdio.h>
#include <time.h>
:
:
int zufall = 8;
srand(time(NULL)); /* initialisiere Zufallsgenerator */

while(zufall != 1048)
{
    printf("Mist....  zufall ist %d und nicht 1048\n", zufall);
    zufall = rand() % 2000;
}

printf("Endlich raus aus dieser Schleife!\n");
```

- Die Funktion `printf` ist die wichtigste Funktion, wenn man am Bildschirm etwas ausgeben will
- Sie wird in der Datei `stdio.h` definiert und muss stets eingebunden sein
- Konstrukte von C++ oder Java wie `"a = " + a + "\n"` sind in C nicht möglich
- Bei `printf` muss man zuerst das Format eingeben und dann alle Variablen hinten anhängen

printf Beispiele

```
#include <stdio.h>

printf("Nur Text ausgeben\n");
printf("Inhalt eines Integers ausgeben, d=%d\n", d);
printf("Inhalt eines Floats ausgeben, d=%f\n", d);
printf("Inhalt eines Strings ausgeben:  %s\n", "Hallo, Welt!");
printf("Nur ein einzelnes Zeichen ausgeben:  %c\n", 'a');
printf("Gemischte Ausgabe:  3+5=%d, string=%s\n", 3+5, "Hallo, Welt!");
```

- kompiliere das "Hello World" Programm
- finde heraus, welcher der Unterschied zwischen `k++` und `++k` für eine Integer Variable `k` (zum Beispiel mit Hilfe der `printf` Funktion)
- finde heraus, was `a += 5;` für eine Integer Variable `a` bewirkt. Geht es auch mit den anderen Operatoren?
- Schreibe eine Funktion, die die ersten 30 Glieder der Fibonacci Folge auf den Bildschirm ausgibt

- Die Struktur des Codes eines Programms ist sehr wichtig, damit man sich mit dem Code beschäftigen kann
- Spaghetti-Code ist unleserlich
- Code sollte in Funktionen und in mehreren Dateien unterteil werden $\longrightarrow$ erhöht die Lesbarkeit.

- Funktionen sind "kleine" Code Einheiten (verglichen mit dem gesamten Code)
- Wenn ein Teilprogramm immer wieder vorkommt und sich an wenigen Stellen unterscheidet, sollte man es in eine Funktion packen
- Es ist einfacher Funktionen aufzurufen, der Code wird kleiner und lesbarer. Bibliotheken sind einfacher zu bedienen und zu erzeugen
- Es gibt keine Faustregel, was alles in eine Funktion gepackt wird.

- Kleine Programme schreibt man in eine einzelne Datei. Wenn man alles in eine einzelne Datei schreibt, ist es schwer den Code zu lesen
- Man sollte den Code in mehreren Dateien unterteilen
- Die üblichen Dateiendungen für C Code sind .c und .h (für Header Dateien)
- Erst beim Linken werden alle kompilierte Dateien zu einem ganzen Programm zusammen betrachtet

- Eine Header Datei ist eine Datei, die nur die Definition von Datenstrukturen und Funktionen enthält

Funktionsreihenfolge

```
void foo()
{
    bar();
}

void bar()
{
    do_something();
}
```

GCC meldet

```
$ gcc foobar.c -c
foobar.c:7:  warning:  conflicting types for 'bar'
foobar.c:3:  warning:  previous implicit declaration of 'bar' was here
```

- Prototypen sind die Lösung
- Sie sind dafür da, dass der Compiler weiß, was es gibt.

Lösung

```
void foo();
void bar();

void foo()
{
    bar();
}

void bar()
{
    do_something();
}
```

- In den Header Dateien sollten Funktionsprototypen und selbstdefinierte structs stehen
- Header Dateien sollen durch die sog. "Header guards" geschützt sein
- Die Header guard sind Anweisungen des C-Präprozessors. Sie helfen (auf primitiver Weise) Konstanten zu deklarieren und Stellen des Codes erst beim Compilieren "sichtbar" bzw. "unsichtbar" zu machen

complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H


struct complex {
    float real;
    float imaginary;
};


struct complex complex_addition(struct complex a, struct complex b);
float get_real(struct complex x);
float get_imaginary(struct complex x);


#endif
```

complex.c

```
#include "complex.h"


struct complex complex_addition(struct complex a, struct complex b)
{
    /* Code comes here */
}
...
```

main.c

```c
#include <stdio.h>
#include "complex.h"

int main(void)
{
    struct complex a,b,c;
    a = assign(4, 1);
    b = assign(6, -9.7);

    c = complex_addition(a,b);

    printf("c = (%f, %f)\n", c.real, c.imaginary);

    return 0;
}
```

- Damit ein kompilierter Code lauffähig wird, muss der Code die `main` Funktion haben
- Es sollte nur eine Datei geben, die die `main` Funktion implementiert, sonst kann es zu Linker-Fehlern kommen
- Man braucht kein Prototyp für die `main` Funktion
- Aber ... es gibt mehrere (gültige) Möglichkeiten, die `main` Funktion zu implementieren

Die `main` Funktion

```
int main();

int main(void);

int main(int argc, char *argv[]);

int main(int argc, char **arvg);
```

- Wenn das Programm fehlerlos beendet wurde, sollte `main` die 0 zurückgeben (wichtig für das Betriebsystem)
- Wenn das Programm fehlerhaft beendet wurde, sollte `main` etwas ungleich 0 zurückgeben

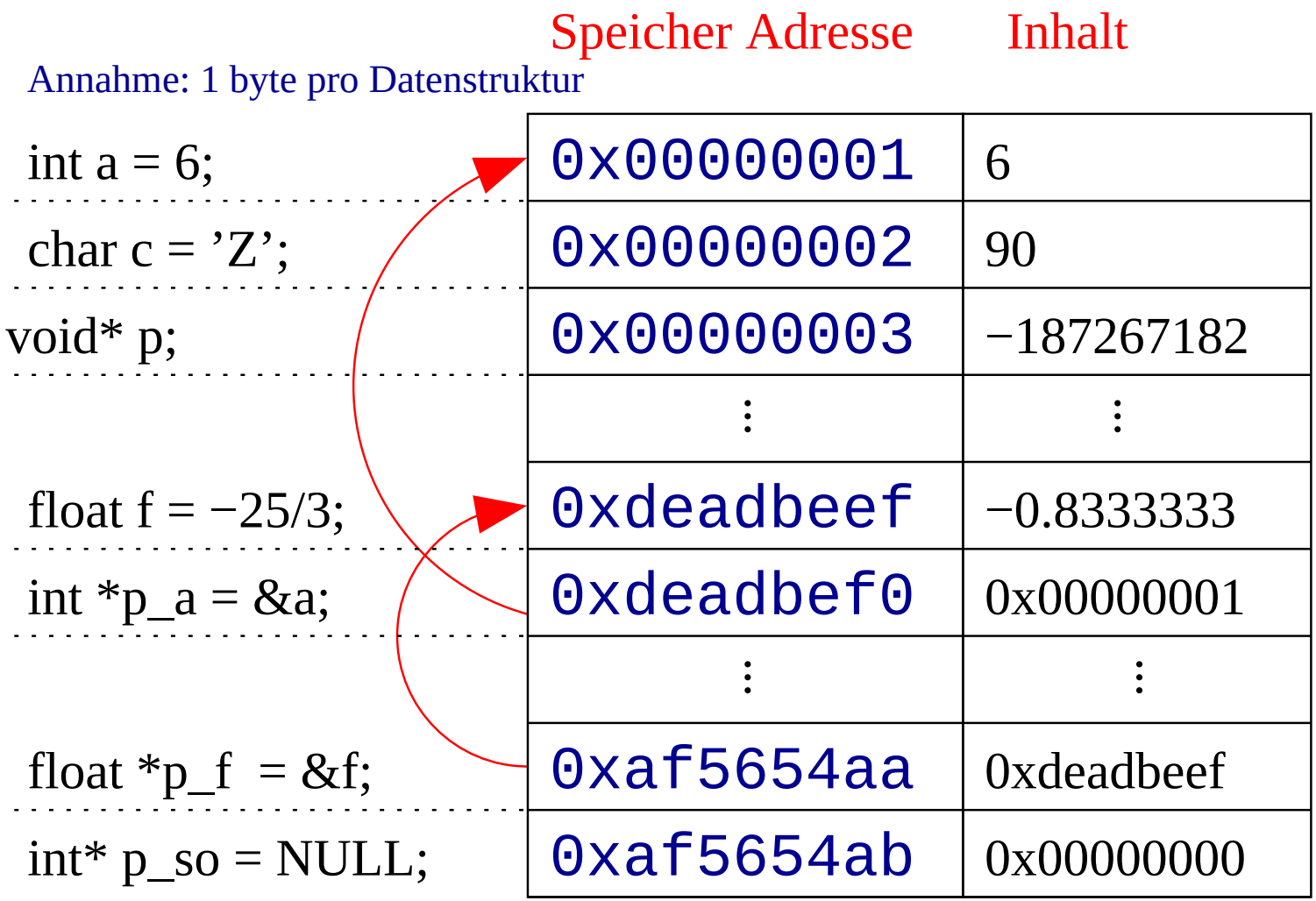
Praxis Übung

- Wir haben als Beispiel für die Unterteilung vom Code die Implementierung der komplexen Zahlen
- Man erkennt sofort, dass die Implementierung der komplexen Zahlen am besten in eigenen Datein geschrieben werden sollte, damit andere Programme ggf. davon profitieren können
- Lade dir die tar.gz/zip Datei `beispiele.tar.gz/beispiele.zip` aus der Kursseite herunter.
- Unter `code/tag01/complex` befindet sich der Code des Beispiels. Erweitere die Implementierung um die Multiplikation und Division der komplexen Zahlen. Teste sie in der `main` Funktion

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

$$\frac{(a+bi)}{(c+di)} = \frac{(ac+bd)}{(c^2+d^2)} + \frac{(bc-ad)}{(c^2+d^2)}i$$

- Zeiger sind sehr wichtig in der C Sprache. Sie ermöglichen eine am Speicher nahe Prorgrammierung
- Zeiger sind an sich Integer Variablen, deren gespeicherten Werte den Adressen des Speichers (worauf sie zeigen) entsprechen
- D.h. mit Zeigern kann man auf Stellen des Speichers zugreifen, die wir beispielsweise nicht deklariert haben (siehe `main` Funktion)
- Es gibt Zeiger auf Zeiger (manchmal ist es notwendig)


Speicher Adresse
Inhalt
Annahme: 1 byte pro Datenstruktur
int a = 6;
char c = 'Z';
void* p;
float f = −25/3;
int *p_a = &a;
float *p_f = &f;
int* p_so = NULL;
0x00000001
6
0x00000002
90
0x00000003
−187267182
0xdeadbeef
−0.8333333
0xdeadbef0
0x00000001
0xaf5654aa
0xdeadbeef
0xaf5654ab
0x00000000

- Was passiert, wenn eine Funktion mehrere Werte auf einmal zurückliefern muss?
- Angenommen: ich will eine Funktion schreiben, die eine komplexe Zahl durch eine natürliche Zahl teilt
- Wenn die natürliche Zahl 0 ist, dann kann man nicht dividieren, Die Funktion sollte in diesem Fall auf den Fehler aufmerksam machen.
- Bei Erfolg soll eine komplexe Zahl zurückgeliefert werden und mitteilen, dass die Funktion fehlerlos gearbeitet hat.

Erster Ansatz

```
int complex_divide(struct complex c, int a)
{
    if(a == 0) /* Fehler */
        return 0;

    c.real = c.real / a;
    c.imaginary = c.imaginary / a;
    return 1;
}
```

- Wenn a ungleich 0 ist, dann wird zwar richtig dividiert und c hat einen gültigen Wert
- Doch wenn `complex_divide` verlassen wird, geht der Wert von c verloren
- An die Funktion wird nur eine Kopie von c übergegeben, d.h. nur eine Kopie des Inhalts von c
- Nach dem Aufruf von `complex_divide` wird die übergegebene Variable in der Tat gar nicht geändert, sondern nur ihre Kopie, die nur innerhalb von `complex_divide` gültig ist

Richtiger Ansatz

```
int complex_divide(struct complex *c, int a)
{
    if(a == 0) /* Fehler */
        return 0;

    c->real = c->real / a;
    c->imaginary = c->imaginary / a;
    return 1;
}
```

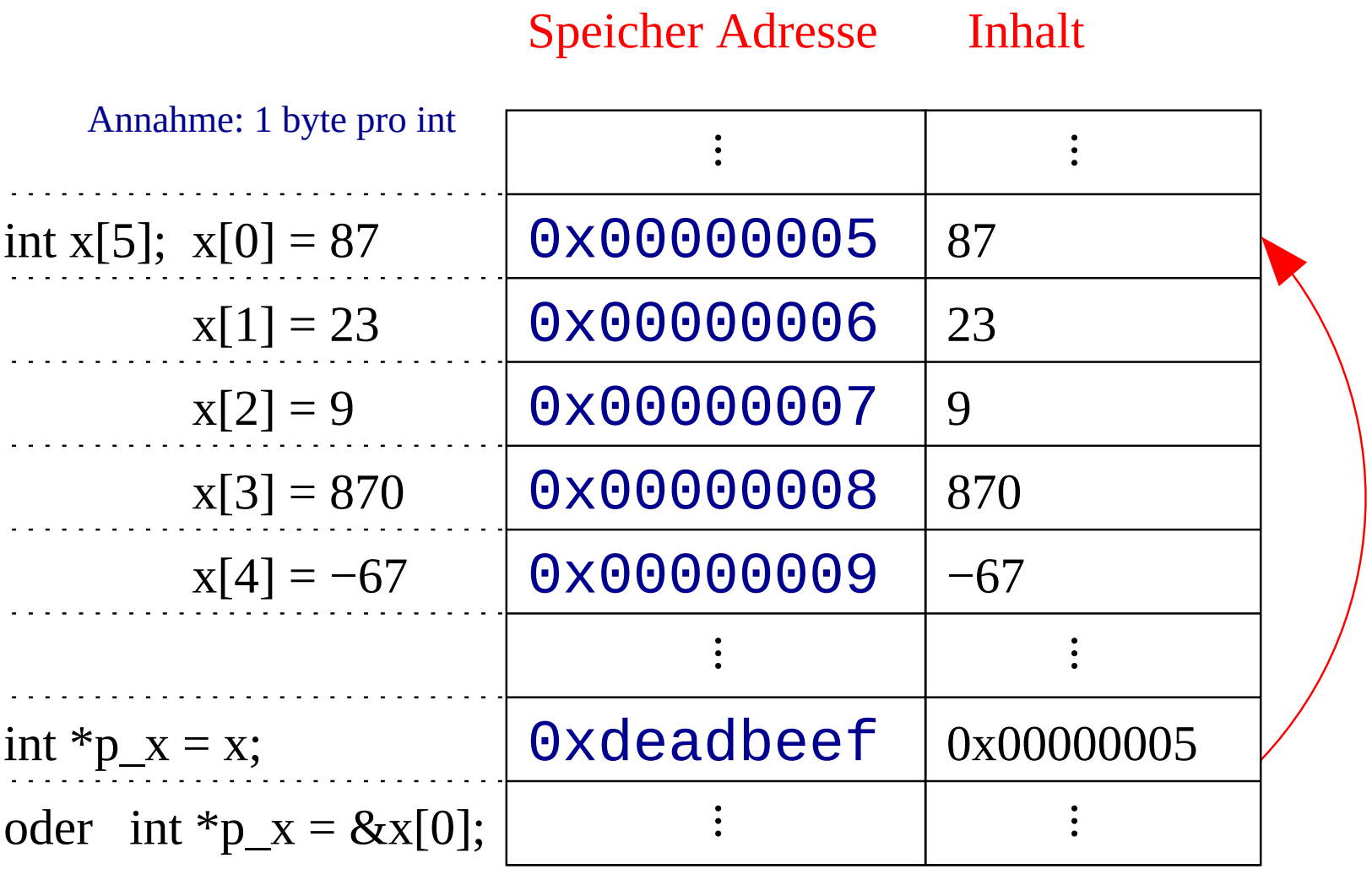
- In diesem Fall wird ebenfalls eine Kopie des Inhalts des Zeigers an die Funktion übergegeben
- Doch, der Inhalt des Zeiger ist die Adresse auf die tatsächliche Variable
- So können wir Inhalte von Variablen ändern, die innerhalb von `complex_divide` gar nicht deklariert sind

- Da Zeiger im Prinzip Integer Variablen sind, kann man mit Zeigern die Operation auf Integers ausführen.
- Was man damit bewirkt, ist dass man den Inhalt des Zeigers (also die Speicheradresse, auf die der Zeiger zeigt) ändert
- So kann man sich durch den Speicher "bewegen", erst interessant, wenn man mit dynamsichen Speicher arbeitet

Zeiger-Arithmetik

```
int a = 10; int *p_a = &a; /* Zeiger auf a */

printf("Inhalt von p_a:  %x.  Inhalt von a:  %d\n", p_a, *p_a);

*p_a = 18; /* Adresse bleibt gleich, Inhalt von a ändert sich auf 18 */

p_a++; /* Zeige auf nächste Stelle im Speicher */

(*p_a)++; /* Erhöhe Inhalt von a um 1 */

p_a = 0xdeadbeef; /* Möglich, aber wenn 0xdeadbeef nicht im Adressraum
liegt, dann kann nicht drauf zugreifen */
```

- Arrays (Felder) sind wie Tabellen, die mehrere Werte eines gleichen Typs speichern können
- Sie verhalten sich sehr ähnlich wie Zeiger. Wenn eine Funktion ein Array braucht, wird in der Tat einen Zeiger auf das erste Element des Arrays übergegeben
- Die Länge der Arrays muss vor der Kompilierungszeit bekannt sein (ab C99 nicht mehr, aber die wenigsten Compiler halten sich jetzt an C99)


Speicher Adresse
Inhalt
Annahme: 1 byte pro int
int x[5]; x[0] = 87
x[1] = 23
x[2] = 9
x[3] = 870
x[4] = −67
int *p_x = x;
oder int *p_x = &x[0];
0x00000005
87
0x00000006
23
0x00000007
9
0x00000008
870
0x00000009
−67
0xdeadbeef
0x00000005

Arrays

```
int arr1[5] = { 3, 4, 1, -2, 97 };
int arr1[] = { 3, 4, 1, -2, 97 };
int arr1[5]; arr1[0] = 3; arr1[2] = 4; ...
int *p = arr1; p[0] = 3; p[1] = 4; ...
int *p = &arr1[0]; p[0] = 3; p[1] = 4; ...
```

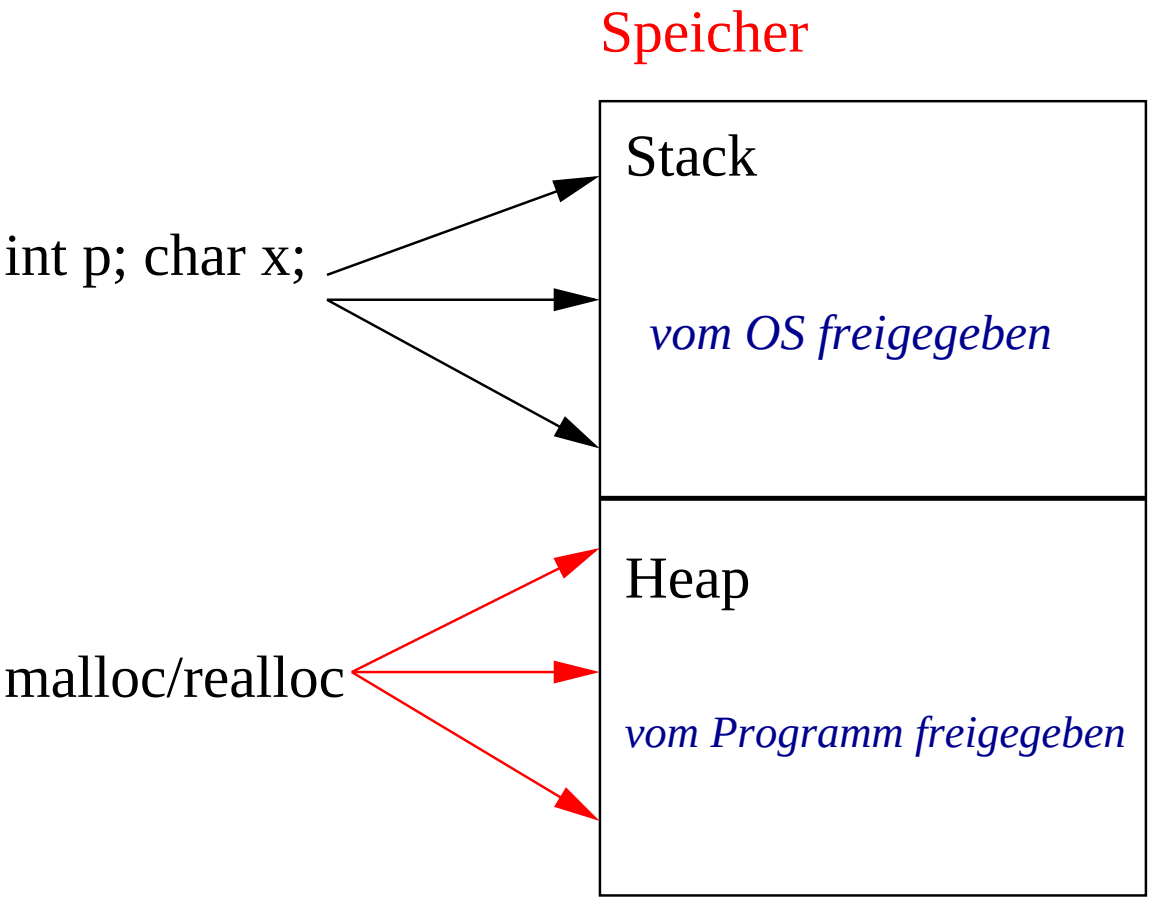
Es geht auch schief...

```
int anzahl;
printf("Gib eine Zahl ein:  ");
scanf("%d", &anzahl);

int array[anzahl];
```

- Das geht nicht immer. Erst ab C99 erlaubt, aber kein Compiler unterstützt das neue C99 Standard vollständig
- Bis C98 müssen Variablen stets am Funktionsanfang deklariert werden
- Somit wäre eine Anweisung wie `int array[anzahl];` ungültig

Lösung

```c
#include <stdlib.h>

int anzahl, *pointer;
printf("Gib eine Zahl ein:  ");
scanf("%d", &anzahl);

pointer = malloc(sizeof(int) * anzahl);
if(!pointer) {
    /* Kein freier Speicher.  Fehlerbehandlung */
}

pointer[0] = 8;
...
free(pointer);
```


Speicher
Stack
int p; char x;
vom OS freigegeben
Heap
malloc/realloc
vom Programm freigegeben

malloc

```
#include <stdlib.h>

void *malloc(size_t size);
```

- `malloc` alloziert (reserviert) Speicher vom Heap
- `size_t size` ist die Anzahl von Bytes, die man reservieren möchte
- Bei Erfolg liefer `malloc` einen Zeiger auf den reservierten Speicher zurück
- Sonst `NULL`, d.h. `malloc` war nicht in der Lage Speicher zu reservieren

Verwendung von malloc

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

int main(void)
{
    int i, j = 10, *p;
    p = malloc(sizeof(int) * j);

    if(!p){
        printf("Kein freier Speicher mehr\n");
        return 1;
    }

    for(i = 0; i < j; ++i) p[i] = i;

    free(p);
    return 0;
}
```

realloc

```
#include <stdlib.h>

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

- realloc verändert die allozierte Speicherkapazität eines allozierten Speichers
- void *ptr ist der allozierte Speicher
- size_t size ist die neue Größe (in bytes)
- ralloc verhält sich genauso wie malloc. Es kann aber passieren, dass die Adresse sich ändert

Verwendung von realloc

```
#include <stdlib.h>

int *p, *tmp;
...  /* p wurde irgendwann mit malloc alloziert */

/* Sei n eine int Variable mit der neuen Größe (in Bytes) */
tmp = realloc(p, n);
if(!tmp)
    /* Kein freier Speicher:  Fehlerbehandlung --> abbrechen */

if(p != tmp)
    p = tmp;

...
free(p); /* tmp braucht nicht gefreet werden, schon erledigt */
```

calloc

```
#include <stdlib.h>

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

- `calloc` reserviert ein Array mit `nmemb` Elementen, die `size` viele Bytes brauchen
- Im Prinzip dasselbe wie `malloc` mit dem Unterschied, dass der gesamte Speicher mit 0 initialisiert wird
- Es sind äquivalent: `calloc(5, sizeof(int))` und `malloc(sizeof(int) * 5)` mit dem oben genannten Unterschied