3. Die Programmiersprache C - Vertiefung深化

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

3.1 Vorschau

- Pointer Arithmetik
- Unterschied Array vs. Pointer
- Zweifach Zeiger
- Dreifach Zeiger
- Funktions-Pointer

- 指针算术
- 数组与指针的区别
- 双指针
- 三层指针
- 功能指针
- 堆和。堆栈
- 数组和结构的内存布局
- 工会

- Heap und. Stack
- Memory-Layout von Arrays und Structs
- Unions

3.2 Pointer-Arithmetik

- 指针可以在运行时改变
- 注意：指针总是由值*类型改变的

- Zeiger können zur Laufzeit verändert werden
- Achtung: der Zeiger immer um `Wert*Typ` verändert
- Beispiele

Ausdruck	Zeiger-Typ	Addierter Wert auf die Adresse
ptr+1	`char`	1
ptr+1	`int32_t`	4
ptr+1	`double`	8
ptr+2	`char`	2
ptr+2	`int32_t`	8
ptr+2	`double`	16
...	...	...

3.2 Pointer-Arithmetik

- Beispiel:

```c
/* ptr_arith.c */
#include <stdio.h>

int main(void) {
   int array[] = { 45, 67, 89 };
   int *array_ptr = array;  /* abgekürzte Schreibweise */

   printf(" first element: %d\n", *(array_ptr++) );
   printf("second element: %d\n", *(array_ptr++) );
   printf(" third element: %d\n", *array_ptr );
}
```

3.2 Pointer-Arithmetik

- Beispiel: `*(array_ptr++)` **vs** `(*array_ptr)++`
- Was ist hier der Unterschied?

3.2 Pointer-Arithmetik

- Beispiel:

```
*(array_ptr++)        vs      (*array_ptr)++
```

Wert an Adresse auslesen
Danach Zeiger inkrementieren

读取地址上的值
然后增加指针

Wert an Adresse auslesen
Danach Wert an der Adresse inkrementieren

读取地址上的值 然后增加地址上的值

3.2 Pointer-Arithmetik

- Manchmal ist Zeigerarithmetik besser lesbar

- 有时，指针式运算更易读

- Beispiel: ohne Zeigerarithmetik不含指针算术

```
char buffer[1024];

strcpy(buffer, "hello ");
strcpy(&buffer[6], "world!");
```

- Und das gleiche Beispiel mit Zeigerarithmetik

```
char buffer[1024];

strcpy(buffer, "hello ");
strcpy(buffer + 6, "world!");
```

3.2 Pointer-Arithmetik

- Aber oft auch Fallstricke
- Was ist hier falsch?

```c
/* ptr_sizeof.c */
#include <stdio.h>

int main(void) {
   int array[] = { 45, 67, 89, 0 };
   int *ptr    = array;

   while (*ptr != 0) {
      printf("ptr=%lx, %d\n", (long unsigned int)ptr, *ptr );
      ptr += sizeof(int);
   }
}
```

3.2 Pointer-Arithmetik

- Zeigeraddition berücksichtigt bereits die Größe des Zeiger-Datentyps!
- Richtige Lösung:

```c
/* ptr_sizeof.c */
#include <stdio.h>

int main(void) {
   int array[] = { 45, 67, 89, 0 };
   int *ptr = array;

   while (*ptr != 0) {
      printf("ptr=%lx, %d\n", (long unsigned int)ptr, *ptr );
      ptr++;
   }
}
```

3.3 Array vs. Pointer

- Ein Array ist nicht das Gleiche wie ein Pointer
- Ursprung dieses Irrtums: Array- und Zeigerdeklarationen als formale Parameter einer Funktion sind austauschbar.

- Unterschiede:
  - Array speichert Daten; Pointer speichert eine Adresse von Daten
  - Ein Zeiger muss nicht auf den Anfang eines Arrays zeigen, ein Array-Name tut dies jedoch immer.

- 数组与指针不一样
- 这种误解的起源：数组和指针的声明作为函数的正式参数是可以互换的。
- 差异。
- 阵列存储数据；指针存储数据的地址。
- 指针不一定要指向一个数组的开头。
但一个数组名称总是如此。

3.3 Array vs. Pointer

- 数组变量的名称对应于指向第0个元素的指针。

- Name einer Array-Variable entspricht einem Pointer zum 0-ten Element.
- **a[i] ≡ *(a + i)**

```c
/* arrptr.c */
#include <stdio.h>

int main(void) {
    int array[] = { 1, 2, 3 };
    int *ptr;
    int i;

    ptr = &array[0];
    for (i=0; i<3; i++) {
        printf("%d. element: %d (%d)\n", i, array[i], *(ptr+i) );
    }
    return 0;
}
```

3.3 Array vs. Pointer

从数组到指针的赋值通常以缩写的形式书写。

- Zuweisung von einem Array zu einem Pointer wird meist abgekürzt geschrieben:

```
/* arrptr.c */
#include <stdio.h>

int main(void) {
    int array[] = { 1, 2, 3 };
    int *ptr;
    int i;

    ptr = array;
    for (i=0; i<3; i++) {
        printf("%d. element: %d (%d)\n", i, array[i], *(ptr+i) );
    }
    return 0;
}
```

3.3 Array vs. Pointer

作为参数的数组以指针形式传递→大小丢失

- Array als Parameter wird als Pointer übergeben → Größe geht verloren

```c
/* arrptr_size.c */
#include <stdio.h>

void foo(int array[], unsigned int size) {
    printf("sizeof(array)=%lu, size==%u\n", sizeof(array), size );
}

int main(void) {
    int array[] = { 1, 2, 3 };

    printf("sizeof(array)=%lu\n", sizeof(array) );

    foo( array, 5 );

    return 0;
}
```

3.3 Array vs. Pointer

- Was ist der Unterschied hier?

```
1. char *ptr  = "A string";
2. char arr[] = "Another string";
```

  - 1. deklariert einen Zeiger auf ein String-Literal
    - Literal ist konstant, schreibende Zugriffe resultieren in undefiniertem Verhalten
  - 2. deklariert ein Array, welches mit einem String-Literal initialisiert wird
    - Array darf geändert werden

*ptr应该指向数组，然后改变内容

- `*ptr` dürfte auf das Array zeigen und dann den Inhalt ändern `ptr = arr;`
- `arr[]` kann nicht dem Zeiger zugewiesen werden ~~`arr = ptr;`~~
  → erlaubt der Compiler nicht

Was ist hier falsch?

```c
/* arrptr2.c */
#include <stdio.h>

char *a  = "A string";
char b[] = "Another string";

int main(){
    *a = '*';
     a = b;
    *a = '*';
     b = a;
    printf("%s\n", b);
    return 0;
}
```

http://codinghighway.com/2013/09/07/the-shocking-unbelievable-truth-arrays-and-pointers-are-not-the-same-thing/

Was ist hier falsch?

```
/* arrptr2.c */
#include <stdio.h>

char *a  = "A string";
char b[] = "Another string";

int main(){
    *a = '*';   /* error -> writing in string constant */
   a = b;
  *a = '*';    /* OK -> writing to array now */
   b = a;      /* error by compiler (not allowed)
  */printf("%s\n", b);
  return 0;
}
```

http://codinghighway.com/2013/09/07/the-shocking-unbelievable-truth-arrays-and-pointers-are-not-the-same-thing/

3.3 Array vs. Pointer

数组和指针大多是可以互换的，但不一定。

- Array und Pointer sind meist austauschbar, aber nicht immer

file1.c

```
#include <stdio.h>

#define BUFFER_SIZE 10000

char buff[BUFFER_SIZE];
```

file2.c

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

extern char *buff;

int main(void) {
    buff[0] = 'A';
    buff[1] = '\0';

    printf("%s\n", buff);
    return 0;
}
```

3.3 Array vs. Pointer

- Array und Pointer sind meist austauschbar, aber nicht immer

file1.c

```
#include <stdio.h>

#define BUFFER_SIZE 10000

char buff[BUFFER_SIZE];
```

file2.c

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

extern char *buff;

int main(void) {
    buff[0] = 'A';
    buff[1] = '\0';

    printf("%s\n", buff);
    return 0;
}
```

3.3 Array vs. Pointer

- Array und Pointer sind fast immer austauschbar verwendbar

file1.c

```c
#include <stdio.h>

#define BUFFER_SIZE 10000

char buff[BUFFER_SIZE];
```

file2.c

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

extern char buff[];

int main(void) {
    buff[0] = 'A';
    buff[1] = '\0';

    printf("%s\n", buff);
    return 0;
}
```

3.3 Array vs. Pointer

- Assembler:

```
gcc –o test file1.c file2.c
objdump -D -M x86-64,intel-mnemonic --no-show-raw-insn test
```

```
extern char *buff;
```

```
00000000 <main>:
 ...
 673:   mov    rax,QWORD PTR [rip+0x2009c6]  # <buff>
 67a:   mov    BYTE PTR [rax],0x41           # 'A'
 67d:   mov    rax,QWORD PTR [rip+0x2009bc]  # <buff>
 684:   add    rax,0x1
 688:   mov    BYTE PTR [rax],0x0            # '\0'
 68b:   mov    rax,QWORD PTR [rip+0x2009ae]  # <buff>
 692:   mov    rdi,rax
 695:   call   510 <puts@plt>
 69a:   mov    eax,0x0
 69f:   pop    rbp
 6a0:   ret
```

```
extern char buff[];
```

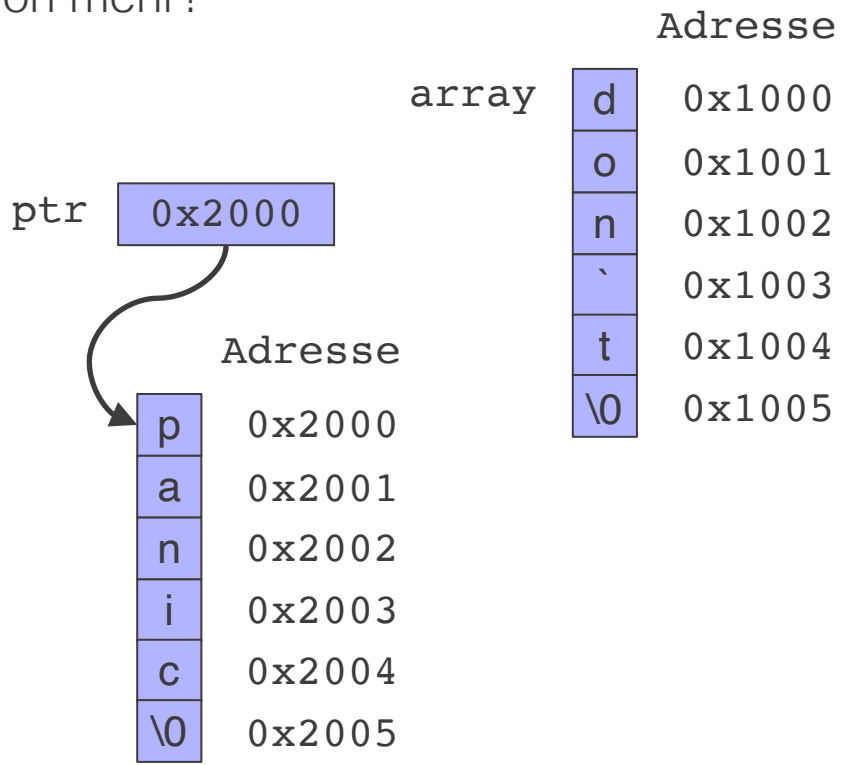
```
00000000 <main>:
 ...
 673:   mov    BYTE PTR [rip+0x2009c6],0x41  # 'A'
 67a:   mov    BYTE PTR [rip+0x2009c0],0x0   # '\0'
 681:   lea    rdi,[rip+0x2009b8]
 688:   call   510 <puts@plt>
 68d:   mov    eax,0x0
 692:   pop    rbp
 693:   ret
```

- Beobachtung: Der Zugriff über den Pointer hat eine Indirektion mehr

3.3 Array vs. Pointer

- Warum hat der Pointer eine Indirektion mehr?

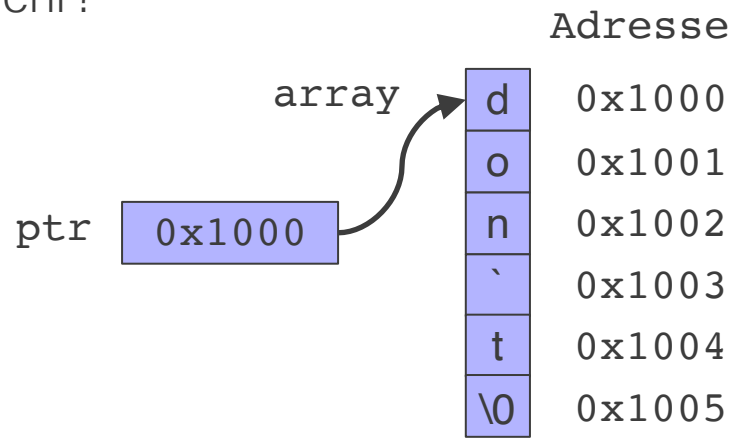
```
char array[100] = "don`t";
char *ptr       = "panic";
```



3.3 Array vs. Pointer

- Warum hat der Pointer eine Indirektion mehr?

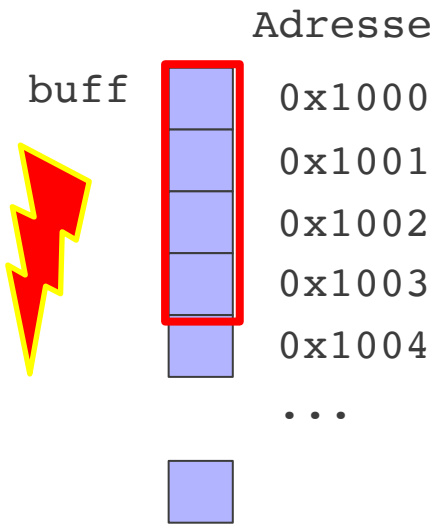
```
char array[100] = "don't";
char *ptr       = &array[0];
```



3.3 Array vs. Pointer

- Was passiert nun in unserem Beispiel mit zwei Dateien?
- Durch die Deklaration als Pointer generiert der Compiler einen indirekten Zugriff
- Es werden nun die ersten 4 Byte unseres char-Arrays ausgelesen und dieser Wert als Pointer verwendet und der Wert dann dereferenziert
- Da hier kein gültiger Pointer steht stürzt das Programm an.

```
extern char *buff;
```

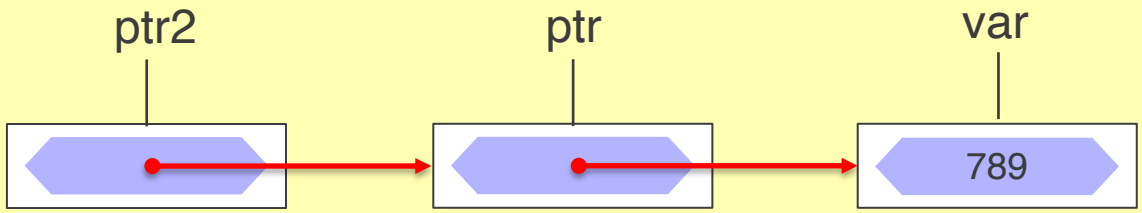


- Bem.: Sonderfall im Zusammenhang mit extern!

3.4 Double Pointers

- Variable die einen Zeiger auf eine Zeigervariable speichert

```c
/* doubleptr.c */
#include <stdio.h>

int main(){
    int var    = 789;
    int *ptr   = &var; /* Zeiger auf var */
    int **ptr2 = &ptr; /* Zeiger auf ptr */

    printf("Wert von var = %d\n", var);
    printf("Wert von var via single pointer = %d\n", *ptr );
    printf("Wert von var via double pointer = %d\n", **ptr2 );
    return 0;
}
```



3.4 Double Pointers

- Double Pointer werden seltener benötigt, sind aber manchmal sehr elegant
- Beispiel: Löschen eines Elementes in einer einfach verketteten Liste
- Datenstruktur:

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

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

typedef struct element_prototype element_type;

element_type *head;
```

3.4 Double Pointers

- Löschen eines Elementes in einer einfach verketteten Liste
- Lösung mit single Pointer
  - Sonderfall falls nur ein Element in der Liste
  - In der Schleife suchen wir den Vorgänger des zu löschenden Elements
  - Austragen & Speicher freigeben

```
void remove_element(int v) {
     element_type  *del  = head;
     element_type  *prev = head;

     if (head->value == v) {
          head = head->next;
          free( del );
          return ;
     }

     while ( prev->next != NULL) {
          if (prev->next->value == v ) {
               del = prev->next;
               prev->next = prev->next->next;
               free ( del );
               return ;
          } else {
               prev = prev->next;
          }
     }
}
```

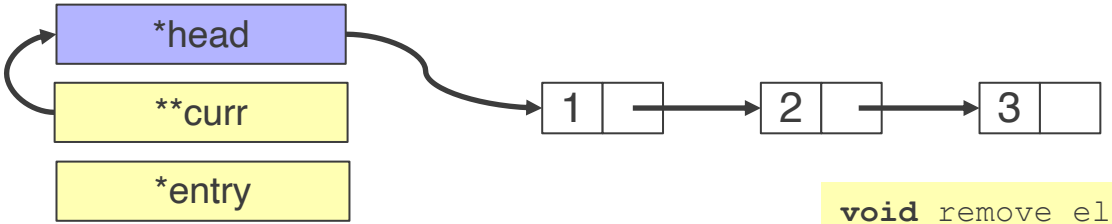
3.4 Double Pointers

- Löschen eines Elementes in einer einfach verketteten Liste
- Lösung mit double Pointer
  - Sonderfall entfällt
  - Code kürzer

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

- Aus: „Two star programming“, siehe hier http://wordaligned.org/articles/two-star-programming

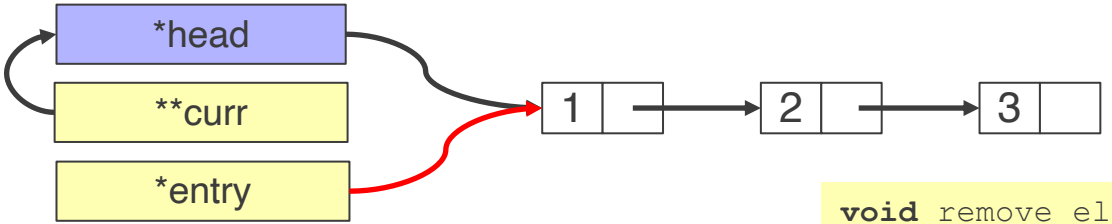
Beispiel: Löschen des 1. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

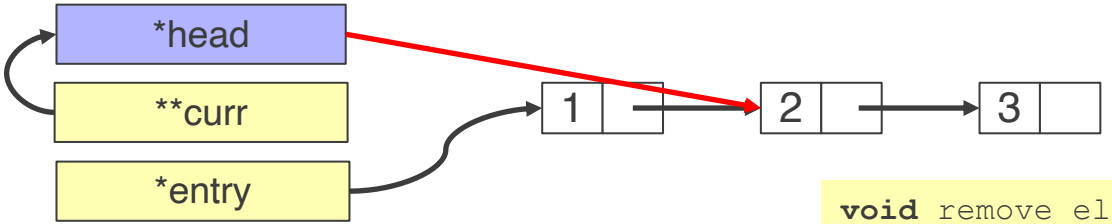
Beispiel: Löschen des 1. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

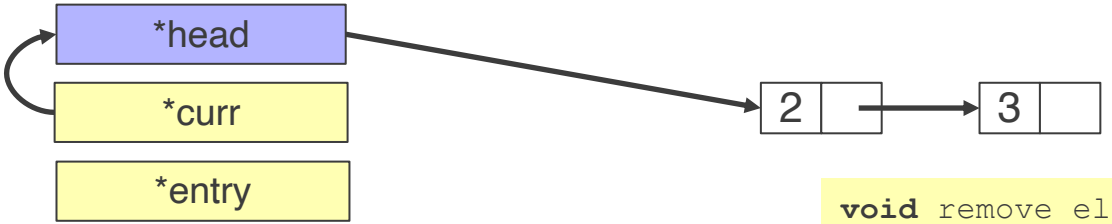
Beispiel: Löschen des 1. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

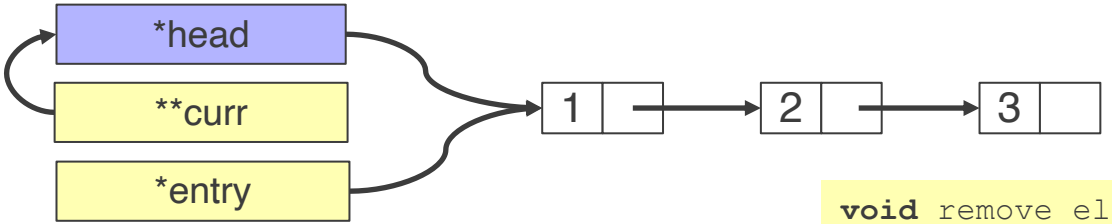
Beispiel: Löschen des 1. Elementes



STOP

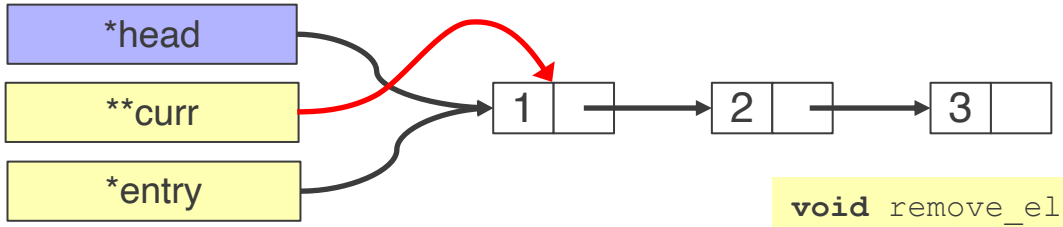
```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

Beispiel: Löschen des 2. Elementes




```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

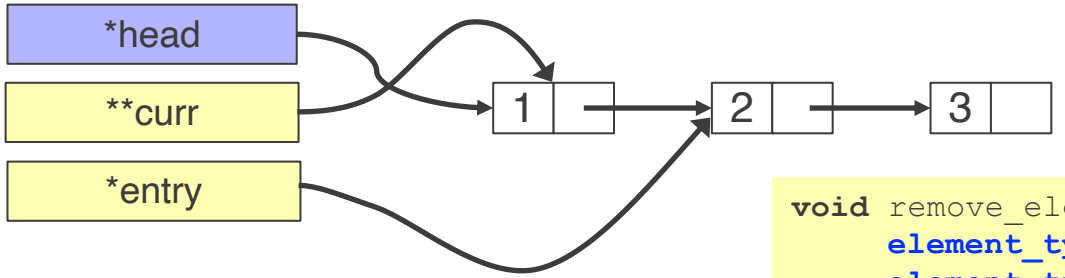
Beispiel: Löschen des 2. Elementes




```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

STOP

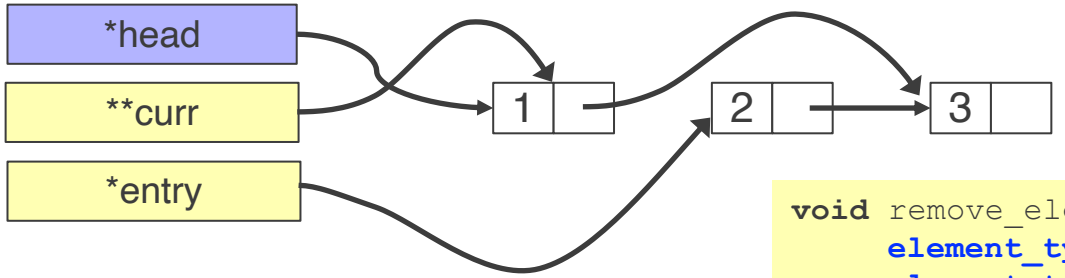
Beispiel: Löschen des 2. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

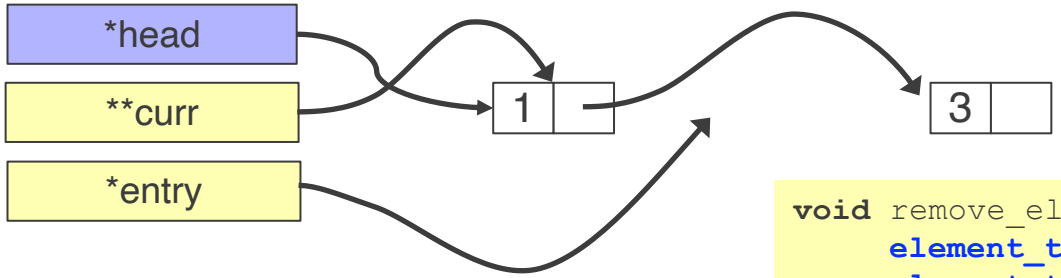
Beispiel: Löschen des 2. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

Beispiel: Löschen des 2. Elementes



STOP

```
void remove_element(int v) {
     element_type  **curr  = &head;
     element_type  *entry;

     while ( *curr != NULL) {
          entry = *curr;
          if (entry->value == v) {
               *curr = entry->next;
               free(entry);
               entry = NULL;
               return ;
          } else {
               curr = &entry->next;
          }
     }
}
```

3.5 Triple Pointers

- „The more indirect your pointers are (i.e. the more * before your variables), the higher your reputation will be“
  → http://wiki.c2.com/?ThreeStarProgrammer

- Beispiel: Rückgabe eines Zeiger-Arrays in dem jeder Zeiger-Eintrag eine variabel lange Zeichenkette referenziert

```
void array_of_strings(int *num_strings, char ***string_data) { ... }
```

- In der Praxis kaum verwendet und außerdem schwer verständlich
  → besser vermeiden

3.6 Funktions-Pointer

- Funktions-Pointer = Typ einer Variable, welche als Adresse eine Funktion hat. Die Signatur dieser Funktion ist im Typ definiert.
- Beispiel: Quick-Sort in **<stdlib.h>** sortiert Objekte in einem Array
  - **base**: Referenz auf das 1. Element des Arrays
  - **nel**: Anzahl an Elementen im Array
  - **width**: Größe jedes Objektes im Array
  - **compar**: Zeiger auf die Vergleichsfunktion.
    Der Rückgabewert hat
    folgende Bedeutung:
    - 0: Objekt 1 == Objekt 2
    - <0: Objekt 1 < Objekt 2
    - >0: Objekt 2 < Objekt 1

```
void qsort( void *base,
            size_t nel,
            size_t width,
            int(*compar)(const void *, const void *)
          );
```

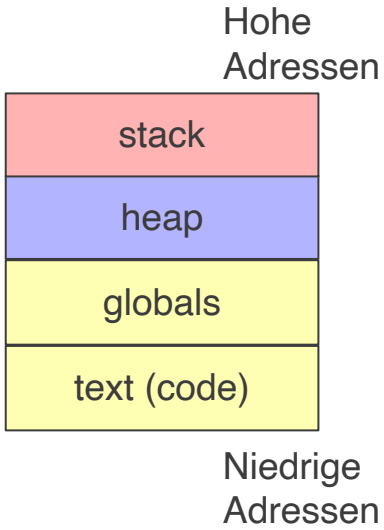
3.6 Funktions-Pointer

- Beispiel: Elemente eines Integer-Array mit `qsort` sortieren

```
/* funcptr.c */
#include <stdio.h>
#include <stdlib.h>

#define NR_OF_ENTRIES   5

int array[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int main () {
   qsort(array, NR_OF_ENTRIES, sizeof(int), &cmpfunc);
   return 0;
}
```

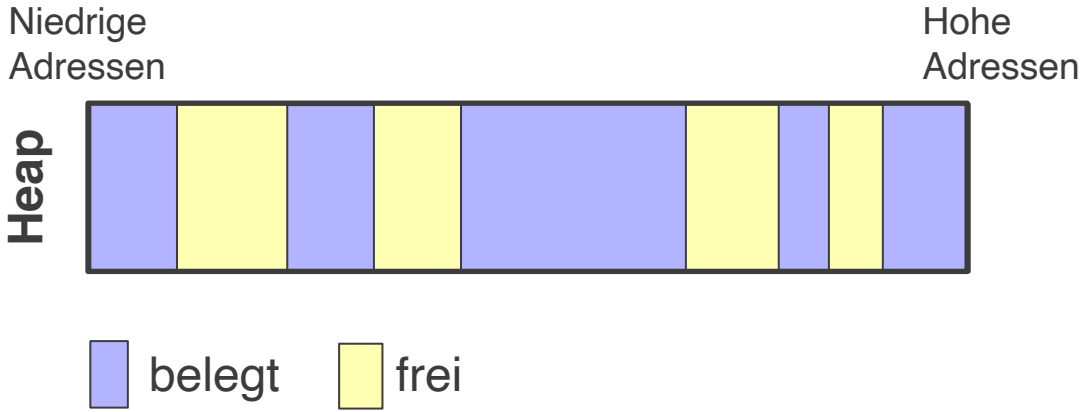
3.7 Heap und Stack

- Heap (dt. Halde):
  - Explizite Allokation und Freigabe durch `malloc` und `free`
  - Für dynamische Datenstrukturen, Instanzen, Puffer etc.
- Stack (dt. Keller, Stapelspeicher):
  - Implizite Allokation und Freigabe durch Programmfluss
  - Für Parameter & lokale Variablen, Rückkehradressen
- Globale Variablen (auch static in C)
  - Automatische Allokation, wenn Programm startet
  - Automatische Freigabe, wenn Programm terminiert



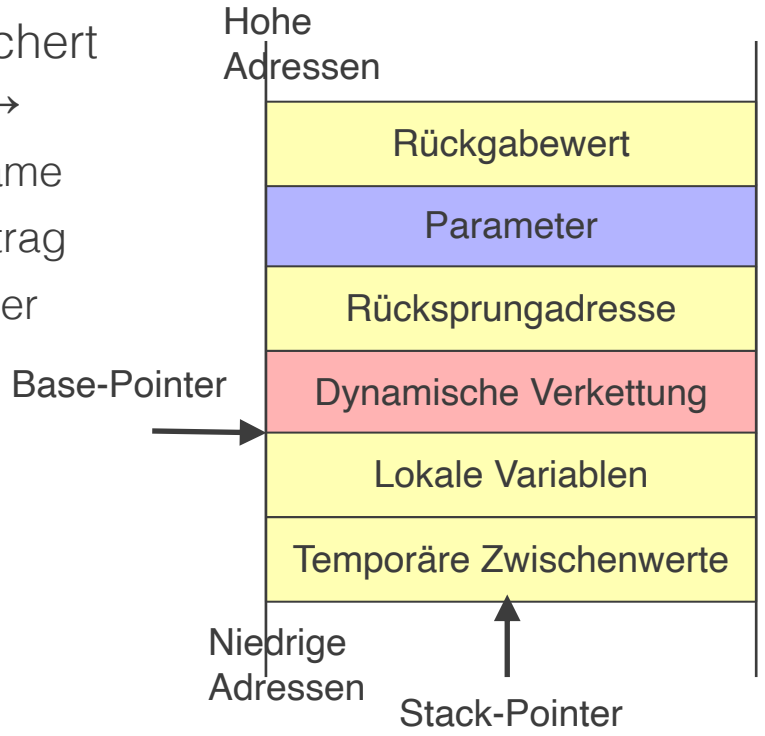
Heap

- Ein großer Speicherblock, welcher je nach Speicheranfragen (`malloc`) in kleinere Stücke zerteilt wird.
- Werden Speicherblöcke freigegeben (`free`), so wird versucht angrenzende freie Blöcke zu verschmelzen, damit wieder größere Blöcke entstehen
- Weitere Details zur Heap-Verwaltung später.



Stack (dt. Keller)

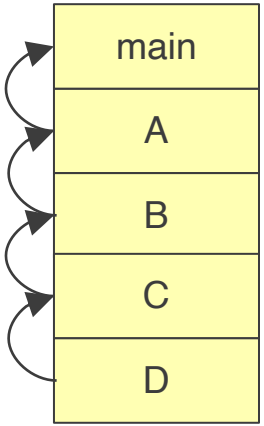
- Wächst und schrumpft abhängig von Funktionsaufrufen
- Ein Kellerrahmen (engl. stackframe) speichert Informationen zu einem Funktionsaufruf →
  - **Base-Pointer**: zeigt auf aktuellen Stackframe
  - **Stack-Pointer**: zeigt auf letzten Stack-Eintrag
  - **Dynamische Verkettung**: alter Base-Pointer des Aufrufers → zeigt auf Stackframe der aufrufenden Funktion
  - **Rückgabewert** manchmal auch in einem Register, statt auf Stack



Dynamische Verkettung der Stackframes

- Kette wächst durch Funktionsaufrufe und schrumpft beim Rücksprung
- Beispiele von Funktionsaufruffolgen

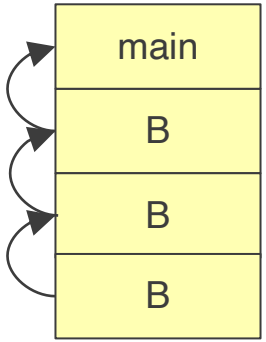
main → A → B → C → D



Hohe Adressen

Niedrige Adressen

main → B → B → B



Stackaufbau an einem konkreten Beispiel

- C-Beispiel:
  - **`mpush-args`**: erzwingt die Parameterübergabe auf dem Stack (funktioniert nur bei 32 Bit)
  - **`m32`**: 32 Bit Code-Generiung
  - **`fno-pie`**: no position indpependent code

```
/* gcc -mpush-args -m32 -fno-pie
       -o stack stack.c */
#include <stdio.h>

int add(int a, int b) {
   return a + b;
}

int main() {
   int e;

   e = add(2,3);

   return 0;
}
```

Stackaufbau an einem konkreten Beispiel (2)

- Re-Assemblieren der Objektdatei mithilfe von **`objdump`**
- Alternativ stack.s ansehen (AT&T Assembly Syntax)
  - **`--save-temps`**: Zwischendateien

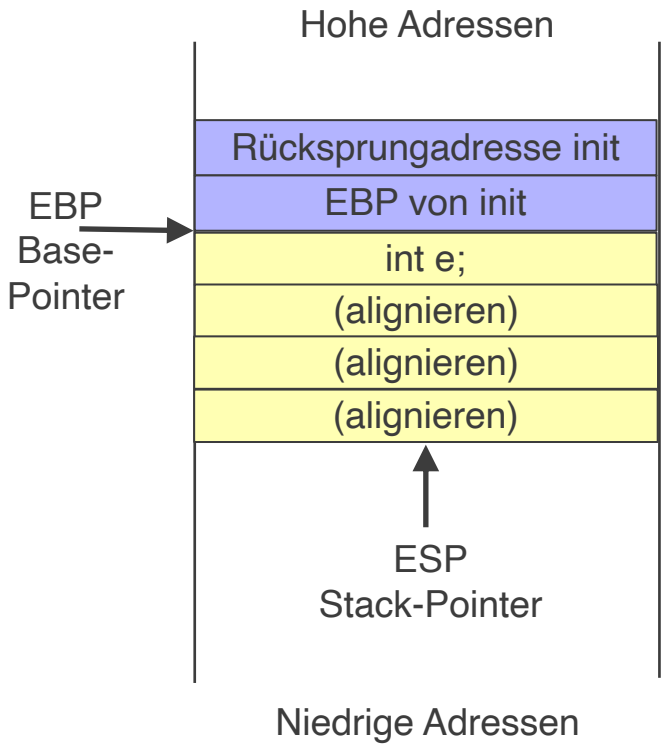
```
objdump -D -M i386,intel-mnemonic
        --no-show-raw-insn stack
```

```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

Speicher für `int e;` + `Padding`

Releases Stackframe set by ENTER

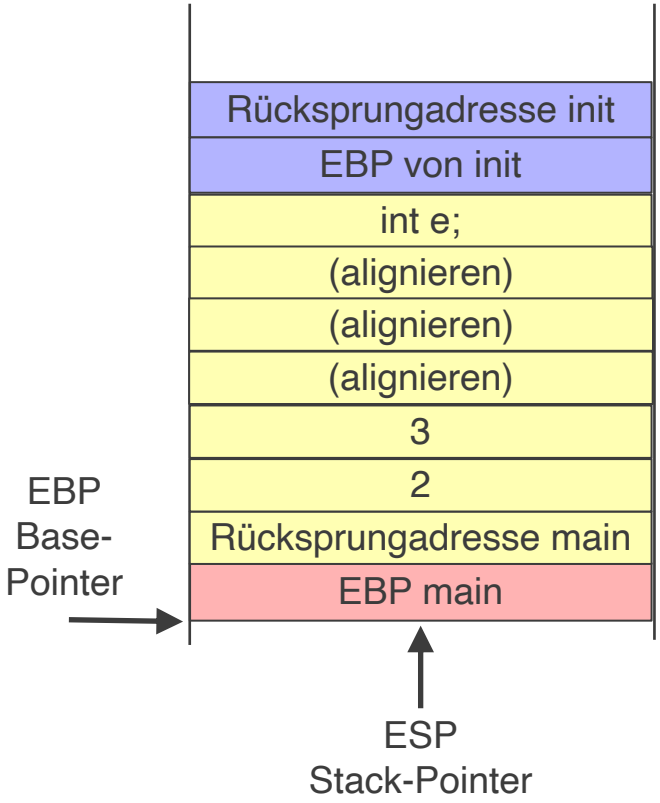
Stackaufbau an einem konkreten Beispiel (2)




```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

STOP

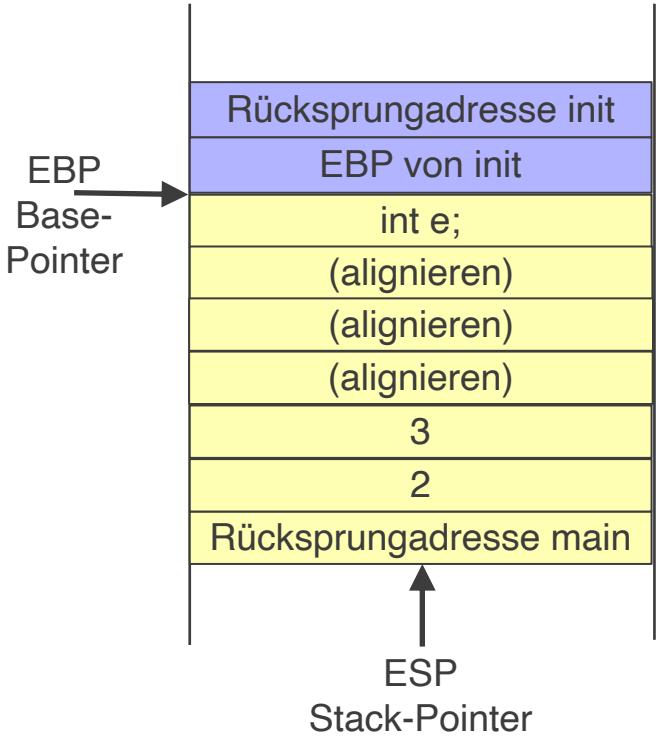
Stackaufbau an einem konkreten Beispiel (2)



STOP

```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

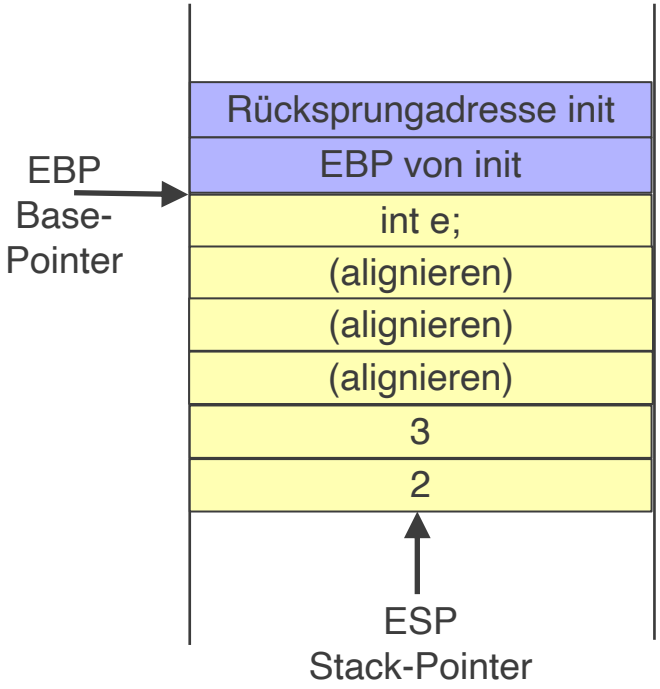
Stackaufbau an einem konkreten Beispiel (2)



STOP

```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

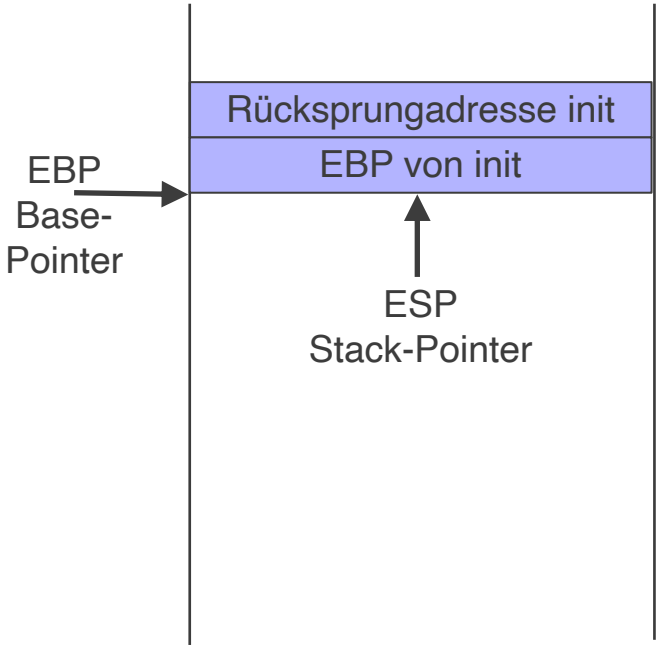
Stackaufbau an einem konkreten Beispiel (2)



STOP

```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

Stackaufbau an einem konkreten Beispiel (2)




```
stack:     file format elf32-i386

Disassembly of section __TEXT,__text:

000004ed <add>:
 4ed:   push   ebp
 4ee:   mov    ebp,esp
 4f0:   mov    edx,DWORD PTR [ebp+0x8]
 4f3:   mov    eax,DWORD PTR [ebp+0xc]
 4f6:   add    eax,edx
 4f8:   pop    ebp
 4f9:   ret

000004fa <main>:
 4fa:   push   ebp
 4fb:   mov    ebp,esp
 4fd:   sub    esp,0x10
 500:   push   0x3
 502:   push   0x2
 504:   call   4ed <add>
 509:   add    esp,0x8
 50c:   mov    DWORD PTR [ebp-0x4],eax
 50f:   mov    eax,0x0
 514:   leave
 515:   ret
```

STOP

Stack - Bemerkungen

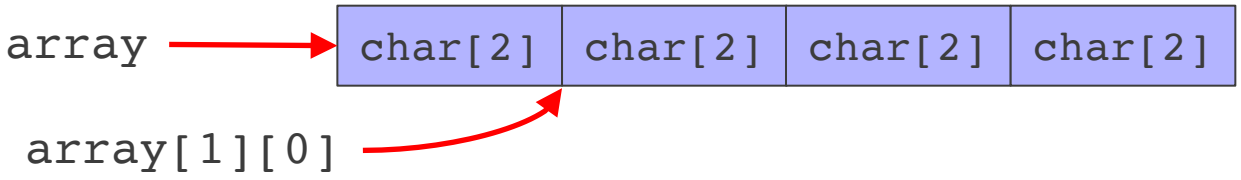
- Stack benötigt i.d.R. wenig Speicher; Ausnahme: rekursive Funktionen
- Structs oder Arrays mit fester Größe, deklariert als lokale Variablen, werden auf dem Stack alloziert, zum Beispiel `int array[5];`
- Compiler versucht Parameter möglichst in Registern zu übergeben, insbesondere ab IA32-64
- Kleine Funktionen mit wenig Code werden kopiert und dupliziert eingefügt
  - Hierdurch wird der Aufruf wegoptimiert
  - Nennt sich „function inlining“

3.8 Memory-Layout von Arrays

- Eindimensionales Array wird als ein Speicherblock alloziert

- Mehrdimensionales Array:
  - Arrays von Arrays
  - Werden aber auch in einem Speicherblock abgelegt (im Gegensatz zu Java)

- Beispiel: `char array[4][2];`



3.8 Memory-Layout von Arrays

- Beispiel: Teil 1

```c
/* array2D.c */
#include <stdio.h>

#define DIM0    4
#define DIM1    2

void dump(char array[DIM0][DIM1]) {
    int  x,y;

    for (y=0; y<DIM0; y++) {
        for (x=0; x<DIM1; x++) {
            printf("%c", array[y][x]);
        }
        printf("\n");
    }
}
```

3.8 Memory-Layout von Arrays

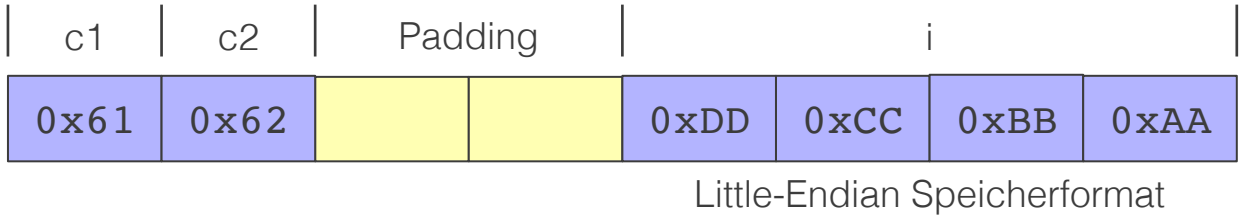
- Beispiel: Teil 2

```
int main() {
    char array[DIM0][DIM1];
    char *array_ptr = array[0];
    char ch = 'A';
    int  i;

    for (i=0; i<DIM0*DIM1; i++) {
        *array_ptr++ = ch++;
    }

    dump(array);

    return 0;
}
```

3.9 Memory-Layout von Structs

- Prozessoren und deren Caches arbeiten am schnellsten wenn die Adressen auf Wortgrenzen oder einem Vielfachen davon sind
- GCC berücksichtigt dies und verwendet bei `structs` das sogenannte Padding

```
/* padding.c */
struct SomeData {
    char  c1;
    char  c2;
    int   i;
}foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xAABBCCDD;
```

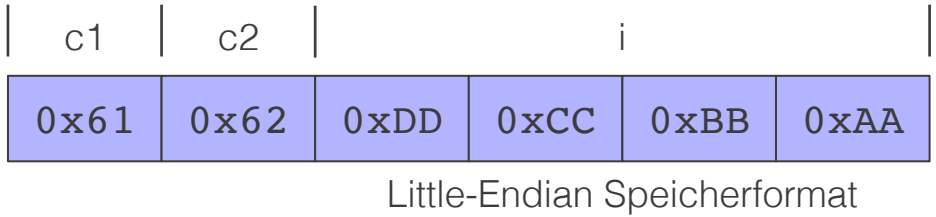


Little-Endian Speicherformat

3.9 Memory-Layout von Structs

- Kann mit dem Attribut **`packed`** unterbunden werden
- Dies ist sinnvoll, wenn viele Millionen von diesen structs angelegt werden
- Oder das Speicherlayout vorgegeben ist, beispielsweise Paketformate bei Netzwerken

```
/* padding.c */
struct SomeData {
    char  c1;
    char  c2;
    int   i;
}__attribute__((packed)) foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xAABBCCDD;
```



Little-Endian Speicherformat

3.9 Memory-Layout von Structs - Bitmodifier

- Beispiel: Bitmodifier

```c
/* bitmodifier.c */
typedef union {
    struct {
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char reserved:4;
    } bits;
    unsigned char byte;
} HW_Register;
```

```c
int main() {
   HW_Register reg;

   reg.byte = 0;     /* init */

   reg.bits.b1 = 1; /* set bit 1 */
   reg.bits.b3 = 1; /* set bit 3 */

   printf("reg=%x\n", reg.byte);
   return 0;
}
```

3.10 Unions

- Ähnlich einem Struct
- Alle Komponenten (Variablen oder Structs) überlappen, aber derselben Adresse
- Die Gesamtgröße ist immer so groß wie die größte Komponente

```c
/* union.c */
union Data {
    char ch;
    int i;
    char str[20];
};

int main( ) {
    union Data data;

    printf( "sizeof union : %lu\n",
                                sizeof(data));

    data.ch = 'A';
    data.i  = 10;
    strcpy( data.str, "C programming");
```

3.10 Unions mit Tags

- Tags sind optional und dienen dazu zur Laufzeit abfragen zu können, welche Variablen in einer Union genutzt werden
- Muss in C manuell nachgebildet werden
  - Union wird hierzu in eine Struct geschachtelt
  - Eine Variable in der umgebenden Struct zeigt an, wie die Union zu verwenden ist
  - Meist werden die Konstanten für den Tag mithilfe eines Aufzählungstyps definiert

3.10 Unions mit Tags: Beispiel

```c
/* uniontag.c */
enum UnionTags {CHAR, INT, STRING};

struct TaggedUnion {
   enum UnionTags tag;

   union Data {
        char ch;
        int i;
        char str[20];
   } value;
};
```

```c
union Data {
   char ch;
   int i;
   char str[20];
} value;


enum UnionTags {CHAR, INT, STRING};

struct TaggedUnion {
   enum UnionTags tag;
   union Data value;
};
```

3.11 Literaturhinweis

- Spezielle Themen zu C werden in folgendem Buch behandelt.
- „Expert C Programming: Deep C Secrets“, Peter Van der Linden, Prentice Hall, 1994.