DS-Systeme

Kapitel 10

Strukturen

Inhaltsverzeichnis

Struktur (struct) – Begriffsdefinition

- Eine Struktur (**struct**) ist die Zusammenfassung von **verschiedenen** Datenelementen in einem zusammenhängenden Speicherbereich.

Beispiel mit Memory-Layout

Beispiel:

C/C++ - Code:

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

Wie bei Arrays:

Sobald eine Variable so einer Struktur angelegt wurde, steht die kleinste Adresse im Speicher unten (Richtung Adresse **NULL**).

Memory-Layout:

Deklaration bzw. Definition

Strukturen werden ähnlich definiert wie Arrays. Mit dem Unterschied dass die einzelnen Elemente (Member) nun auch unterschiedliche Datentypen haben können.

Beispiel: **Initialisierte** Struktur

C/C++ - Code:

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};

struct rec svar = {1, 2, {1, 2}, NULL};

int main()
{
  //...
  return 0;
}
```

Strukturvariable **global** definiert.

Assembler - Code:

```
.section .data
  .align 16
  .globl svar
  .type svar, @object
  .size svar, 24
svar:
# i:
  .int 1
# j:
  .int 2
# a:
  .int 1, 2 # Array
# p:
  .quad 0
```

Deklaration bzw. Definition

Beispiel: **Nicht initialisierte** Struktur

C/C++ - Code:

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};

struct rec svar;

int main()
{
  //...
  return 0;
}
```

Assembler - Code:

```
#.lcomm <symbolname>, <lengthinbyte>, <alignment>
.section .bss
.align 16
  .lcomm svar, 24
```

bzw.

```
#.comm <symbolname>, <lengthinbyte>, <alignment>
.section .bss
.align 16
  .comm svar, 24
```

Strukturvariable **global** definiert.

Deklaration bzw. Definition

Beispiel: Struktur auf dem Stack

C/C++ - Code:

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};

int main()
{
  struct rec svar;
  //...
  return 0;
}
```

Strukturvariable **lokal** in main() definiert.

Assembler - Code:

```
main:
  #...
  subq $24, %rsp
  #...
```

Lokale Variablen werden immer entweder in Registern oder auf dem Stack abgelegt.

Lokale Strukturvariablen werden – genau wie Arrays - wegen ihrer Größe immer auf dem Stack abgelegt.

Zugriff auf Elemente

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

Zugriff auf Strukturelemente **pr->i** und **pr->j**:

Anmerkung:

```
struct rec svar;
struct rec *pr = &svar; // in %rdi
movl (%rdi), %eax  # Get pr->i
movl %eax, 4(%rdi) # Store in pr->j
```

Zugriff auf **&(pr->a[1])**:

Anmerkung:

```
struct rec svar;
struct rec *pr = &svar; // in %rdi, Array-Index von a in %rsi (0 oder 1)

leaq 8(%rdi,%rsi,4), %rax  # Set %rax to &(pr->a[i])
```

8 (Bytes) ist das Offset (die Summe der Größen der Variablen **i** und **j**), das übersprungen wird.

Zugriff auf Elemente - Aufgabe

C/C++-Code:

```
struct S1{
  long u;
  short v;
  char w;
} sv1; // adress in rsi

struct S2{
  int a[2];
  char *p;
} sv2; // adress in rdi

int main()
{
  //...
}
```

Aufgabe:

rsi: Pointer auf **S1**
rdi: Pointer auf **S2**
al, **ax**, **eax** bzw. **rax**:
Register für Ergebnis **x**

rax wird am Anfang mit **0** initialisiert

Füllen Sie unter diesen Voraussetzungen die folgende Tabelle aus:

Expr	ASM-Code
x=sv1.u;	movq (%rsi), %rax
x=sv1.v;	
x=sv1.w;	
x=sv2.a[1];	
x=sv2.p;	

Zugriff auf Elemente - Aufgabe

Zugriff auf Strukturelement **pr->p = &pr->a[pr->i + pr->j]**:

Anmerkung:

```
struct rec *pr = &svar; // in %rdi
```

```
movl 4(%rdi), %eax          # Get pr->j
addl (%rdi), %eax           # Add pr->i
cltq                        # Extend to 8 bytes
leaq 8(%rdi,%rax,4), %rax   # Calc address (pr->a[pr->i + pr->j])
movq %rax, 16(%rdi)         # Store in pr->p
```

8 (Bytes) ist die Summe der Größen der Variablen **i** und **j**, die übersprungen wird.

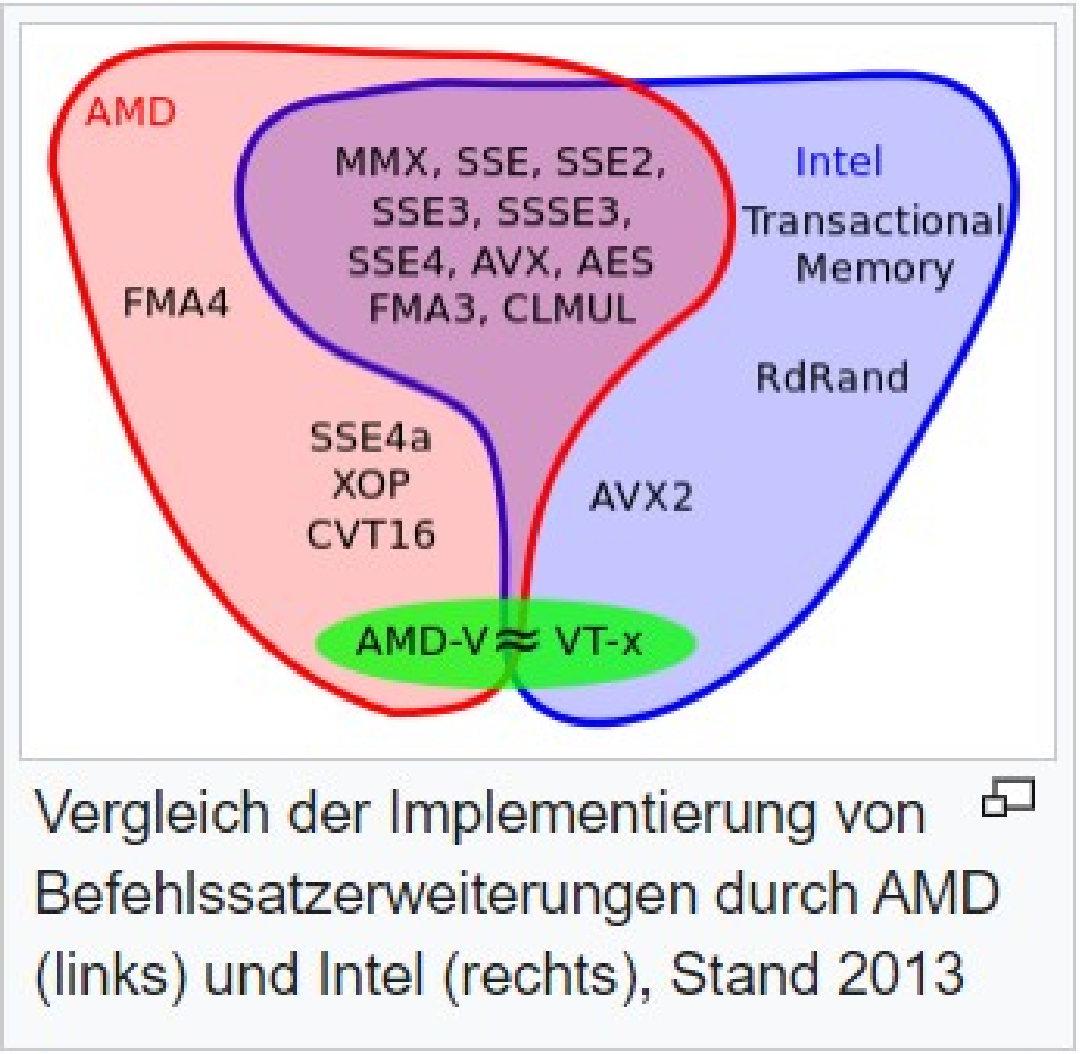
Aufgabe:

Beantworten Sie folgende Fragen:

a)Worauf zeigt jetzt die in **p** gespeicherte Adresse?
b)Welchen Wert hat diese Adresse?

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};

struct rec svar
            = {1, 2, {1, 2}, NULL};

int main()
{
  //...
  return 0;
}
```

Alignment (Ausrichtung)



Vergleich der Implementierung von Befehlssatzerweiterungen durch AMD (links) und Intel (rechts), Stand 2013

Quelle: Wikipedia
https://commons.wikimedia.org/wiki/File:X86_extensions_2013.svg

Die **S**treaming **SIMD** **E**xtensions (**SSE**) ist eine von Intel entwickelte Befehlssatzerweiterung der **x86-Architektur**, die mit der Einführung des Pentium-III-Prozessors 1999 vorgestellt wurde.

Zweck ist es, Programme durch Parallelisierung auf Instruktionslevel zu beschleunigen, genannt **S**ingle **I**nstruction **M**ultiple **D**ata (**SIMD**).

Alignment (Ausrichtung)

- Architekturen wie **x86-64** haben Einschränkungen für erlaubte Adressen bzgl. primitiver Datentypen.
- Werden diese nicht eingehalten, kann die Performance erheblich abnehmen oder zu Ausnahmen führen (SSE-Befehle).
- Generell sollten die Datentypen im Speicher an einer Adresse ausgerichtet werden, die ein Vielfaches von ihrer Größe (in Bytes) sind (**natural alignment**):
  - 64-Bit / 8-Byte Daten an Hex-Adressen, die auf 0 oder 8 enden (8-byte alignment)
  - 32-Bit / 4-Byte Daten auf Hex-Adressen, die auf 0, 4, 8 oder C enden (4-byte alignment)
  - 16-Bit / 2-Byte Daten auf Hex-Adressen die auf 0, 2, 4, 6, 8, A, C oder E enden (2-byte alignment)
  - 8-Bit / 1-Byte Daten beliebig

Alignment (Ausrichtung)

- Diese Ausrichtung ist möglich mit der **.align** Direktive (vor den entsprechenden Daten):
  - z.B. 8-Byte alignment: **.align 8**
- Bei großen Daten-Agglomerationen (Anhäufungen) wie Arrays und Strukturen sollte der Anfang auf 8 Byte ausgerichtet werden.
- Große Strukturen (d.h Strukturen > 64 Byte) sollten auf 16 Byte ausgerichtet werden (Hex-Addressen enden auf 0)
- Bei Strukturen führt das Einhalten des **natural alignments** zusätzlich dazu, dass ggf. **zero padding** zwischen den Datenelementen eingeführt werden muss.
- SSE-Befehle erfordern, dass alle dynamischen bzw. Stack-Adressen ein Vielfaches von 16 sind.
  - ==> **.align 16**

Alignment (Ausrichtung)

Beispiel: **minimal gepackte** Datenstruktur

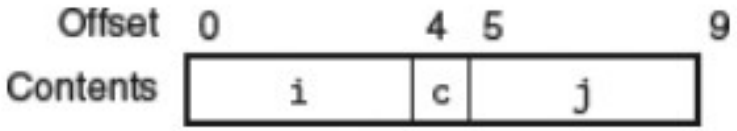
C/C++-Code:

```
struct S1 {
  int  i;
  char c;
  int  j;
};

struct S1 svar = {4, 'C', 8};

int main(){
  //...
  return 0;
}
```

Assembler-Code:

```
.section .data
.align 8
.globl svar
  .type svar, @object
  .size svar, 9
svar:
# i:
  .long 4
# c:
  .byte 67
# j:
  .long 8
```

Memory-Layout:



Entspricht der Verwendung der Direktive

```
#pragma pack(push)*
#pragma pack(1)
// …
#pragma pack(pop)
```

in C / C++.

* **#pragma pack()** ist compilerabhängig

Alignment (Ausrichtung)

Beispiel: **natural alignment**

C/C++-Code:

```
struct S1 {
  int  i;
  char c;
  int  j;
};

struct S1 svar = {4, 'C', 8};

int main(){
  //...
  return 0;
}
```

Assembler-Code:

```
  .section .data
   .align 8
   .globl svar
   .type svar, @object
   .size svar, 12
svar:
# i:
   .long 4   # value
# c:
   .byte 67 # value is ASCII-Code
   .zero 3   # number of bytes
# j:
   .long 8   # value
```

Memory-Layout:

Entspricht der Verwendung der Direktive

```
#pragma pack(push)
#pragma pack(3) // 2^3 = 8
// …
#pragma pack(pop)
```

in C / C++.

Strukturen – Alignment (Ausrichtung)

Beispiel: **Optimierung des Alignments durch Umordnung**

C/C++-Code:

```
struct S1 {
  int  i;
  int  j;
  char c;
};

struct S1 svar =  {4, 8, 'C'};

int main(){
  //...
  return 0;
}
```

Assembler-Code:

```
.section .data
.globl svar
  .align 8
  .type svar, @object
  .size svar, 9
svar:
# i:
  .long 4
# j:
  .long 8
# c:
  .byte 67
```

Memory-Layout: