PERSONAL - Grundlagenpraktikum: Rechnerarchitektur

- x86-64 ASM vs x86-32 ASM
  - Registerüberblick
  - Immediate-Operanden
- IA-64 und IA-32 Software Development Manual
  - Instruktionen lesen und verstehen
- System Calls
  - System Calls auf x86-64 Linux Systeme
- Layout einer Programmbinary
- Start eines Programms
  - Stack bei Programmstart
- Textaufgabe auf der Konsole
- Beenden eines Programms
- Grundlagen - C
- Datentypen in C
  - Integers
    - Beispielcode
  - Floating-Point-Zahlen
  - void
  - Pointer-Datentypen
- Funktionen in C
  - Die main-Funktion
- Variablen in C
  - Scopes
  - Zuweisung von Werten
- Arithmetische und logische Operatoren
- Kontrollflussstrukturen
  - if-else-Bedingungen
  - while und do-while Schleifen

    - for-Schleifen
    - switch-Statements
- Der C-Präprozessor
    - Makros
    - if-else-Konstrukte
    - #include-Direktiven
- Header-Dateien
    - Sichtbarkeit
    - Standard-Header
    - stdint.h und stdbool.h
- printf
    - Format Strings
    - Conversion Specifiers
- Disassemblieren mit objdump
    - Generierter Maschinencode
    - Optimierungsstufen
        - Stufen
        - Weitere Optimierung
    - Extra: Godbolt Compiler Explorer
- Debugging mit GDB
    - Ausführen von Programme in GDB, Breakpoints, Programmfluss
        - Bsp: Array von Pointer und Strings
- Optimierungen
    - Optimierung der Fibonacci-Reihenfolge
        - Basis-Fibonaccifunktion
    - Laufzeitklassen
    - Optimierung für kleine Eingabewerte
    - Fibonacci: Lineare Schleife statt Doppelter Rekursion
    - Fibonacci: Logarithmische Laufzeit mit Formel von Binet
    - Fibonacci: Lookup-Table (LUT)
    - Speicherplatzoptimierung: LUT verkleinern
- Komplexe Datenstrukturen in C
    - sizeof-Operator

- Alignment
- Pointer
  - Pointerarithmetik
  - Pointerarithmetik - Arraysubscript
  - void-Pointer und implizite Pointer-Casts
  - Explizite Pointer-Casts
- Arrays
  - Arrays als Parameter
- Strings
- structs
  - struct als Parameter
- struct vs. union
  - struct mit union
  - union als Parameter
- Mehrdimensionale Arrays
- Speicherbereiche
  - Stack vs. Heap
  - Speicherverwaltung auf dem Heap
- Effizientes Debugging
  - printf-Debugging
  - assert()
- Buffer Overflows
- Segmentation Fault
- Inhärent Unsichere Funktionen
- Überprüfung von malloc()
- Format String Injection
- Memory Leak
- Use After Free und Double Free
- Undefined Behavior
- Vermeidung von Fehlern - Sanitizer
- Kommandozeilenargumente in C
- System V Application Binary Interface
  - Registertypen

  - Stack-Alignment
  - Struct-Layout
  - Structs als Funktionsparameter
  - Structs als Rückgabewerte
- Calling Convention: Zusammenfassung
- Fixkommazahlen / Festkommazahlen
  - Fixkommazahlarithmetik
- Fließkommazahlen
  - Aufbau
  - Genauerer Aufbau
  - Datentypen: float und double
  - Konvertierung zu einer Fließkommazahl und umgekehrt
  - Addition und Subtraktion von Fließkommazahlen
  - Multiplikation und Divison von Fließkommazahlen
  - Probleme bei Genauigkeit
  - Denormale / Subnormale Zahlen
  - Null mit Vorzeichen
  - Unendlich / Infinity
  - Not A Number / NaN
  - Weitere Floating Point Formate
- Fließ- und Fixkommazahlen: Zusammenfassung
- Streaming SIMD Extensions (SSE)
  - SSE Register
  - Konstanten
  - Arithmetik
  - Vergleiche
  - Codeeispiel ()
  - Erweiterte Calling Convention
- SIMD - SSE
  - SIMD - Single Instruction Multiple Data(stream)
  - SSE-Instruktionen für SIMD
  - Integer-Instruktionen
  - Vektoraddition: 32- und 64-bit
  - Inkrementieren eines Elements mittels SIMD

- Gleitkomma-Instruktionen
  - Addition eines Vektors auf sich selbst mit SIMD
  - SIMD-Alignment
  - Aligned Zugriff
  - SIMD-Stolperfallen
  - Compiler und Vektorisierung
- SIMD-Intrinsics
  - Codebeispiel - Saxpy
  - Andere Datentypen
  - Vor- und Nachteile von Intrinsics
  - Automatische Vektorisierung
- Optimierungen
  - Optimierung von Berechnungen
  - Optimierung von Schleifen
  - Optimierung von Funktionsaufrufen
  - Interprozedurale Optimierungen
  - Low-Level Optimierungen
  - Optimierte Funktionen
  - Builtins
  - Funktionsattribute (Beispiele)
  - Layout von Datenstrukturen
  - Layout von structs
  - Pointer-Aliasing
  - Beispiele für Pointer-Aliasing
- Vergleiche mit SIMD
  - Vergleichsbefehle
  - Anwendung der Bitmaske
- SIMD mit General Purpose Registern
- Wann ist SIMD sinnvoll?
- SIMD mit AVX
  - Erweiterte SSE-Instruktionen in AVX
  - Neue AVX-Instruktionen
  - Alignment in AVX
  - SSE und AVX - Adressierungsschemata

- Zeitmessung
  - Zeitmessung im Code
  - Messen eines Zeitpunktes
- Umgebungsbedingungen
- Dokumentation
- Zeitmessung - Zusammenfassung
- Profiling mit perf

x86-64 ASM vs x86-32 ASM

- 64-bit Wortbreite
  - 64-bit Adressen $\rightarrow 2^{64}$ Byte bzw. 16 EB Hauptspeicher adressierbar
  - 64-bit Register
- in GRA: Kommentare mit `//`

Registerüberblick

64-bit	32-bit	16-bit	8-bit	Special Purpose for functions	When calling a function	When writing a function
rax	eax	ax	ah,al	Return Value	Might be changed	Use freely
rbx	ebx	bx	bh,bl		Will not be changed	Save before using!
rcx	ecx	cx	ch,cl	4th integer argument	Might be changed	Use freely
rdx	edx	dx	dh,dl	3rd integer argument	Might be changed	Use freely
rsi	esi	si	sil	2nd integer argument	Might be changed	Use freely
rdi	edi	di	sil	1st integer argument	Might be changed	Use freely
rbp	ebp	bp	bpl	Frame Pointer	Maybe Be Careful	Maybe Be Careful
rsp	esp	sp	spl	Stack Pointer	Be Very Careful!	Be Very Careful!
r8	r8d	r8w	r8b	5th integer argument	Might be changed	Use freely
r9	r9d	r9w	r9b	6th integer argument	Might be changed	Use freely
r10	r10d	r10w	r10b		Might be changed	Use freely
r11	r11d	r11w	r11b		Might be changed	Use freely
r12	r12d	r12w	r12b		Will not be changed	Save before using!
r13	r13d	r13w	r13b		Will not be changed	Save before using!
r14	r14d	r14w	r14b		Will not be changed	Save before using!
r15	r15d	r15w	r15b		Will not be changed	Save before using!

- **unteren 8 Bits** von `esp, ebp, esi, edi` jetzt adressierbar
- **8 neue GPR** hinzugefügt und nach Größe beschriftet (`d` - *DWORD*, `w`- *WORD*, `b` - *BYTE*)
- **EFLAGS** auch auf 64-bit erweitert
- **(!)** bei **schreibendem Zugriff** auf 32-bit Register (*und nur auf 32-bit Register, nicht auf kleinere*) werden die oberen 32-bits von dem dazugehörigen 64-bit Register auf 0 gesetzt
  - z.B. `mov eax, 0xffffffff` führt dazu, dass im Register `rax` jetzt `0x00000000ffffffff` liegt

Immediate-Operanden

- 64-bit Immediates nur bei `mov` erlaubt
  - `mov rax, 0xaaaabbbbccccdddd` erlaubt
  - `add rax, 0xaaaabbbbccccdddd` **nicht** erlaubt
- 32-bit Immediates sind **sign-extended** (*das oberste Bit der unteren 32 Bits wird an allen Stellen der oberen 32 Bits kopiert*)

    - z.B. `add rax, 0xffffffff` führt dazu, dass man eigentlich `0xffffffffffffffff` auf `rax` "addiert"

IA-64 und IA-32 Software Development Manual

Instruktionen lesen und verstehen

- Kapitel **3 bis 6** beinhalten alle Befehle der ISA
- **relevant**: Instruction, 64-bit Mode, Description
    - *Instruction*: Syntax der Operation und Operanden
        - `r`: Registeroperand mit Größe in Bits
        - `m`: Speicheroperand
        - `imm`: Immediate
    - *64-bit Mode*: Operation kann (oder kann nicht) auf x86-64 verwendet werden
        - alles andere als *Valid* bedeutet, dass die Operation nicht verwendet werden kann
    - *Description*: kurze Erklärung der Operation

System Calls

- aus Sicherheitsgründen unterliegen Programme auf Linux gewisse **Restriktionen** (*z.B. R / W*)
- Zugriff wird über **Anfragen an das Betriebssystem** (*Syscalls*) geregelt
    - **Interface** zwischen Programm und OS
    - Syscalls kann man sich als **Funktionsaufruf** vorstellen
    - Syscalls werden eindeutig über **System Call Numbers** identifiziert

System Calls auf x86-64 Linux Systeme

- solche ISAs haben die Instruktion `syscall`
    - übergibt Kontrolle an Kernel des Betriebssystems und führt Syscall aus
    - danach wird normale Programmausführung fortgesetzt
- *Reihenfolge der Argumente (Konvention)*: `rdi, rsi, rdx, r10, r8, r9`
- Nummer des Syscalls wird in `rax` weitergeleitet
- Die Adresse, an der die Programmausführung i.A. nach dem System Call fortgesetzt wird, wird in `rcx` gespeichert
- Ergebnis des Syscalls wird in `rax` geschrieben
    - ein Ergebnis zwischen `-4095` (*inkl.*) und `-1` (*inkl.*) bedeutet, dass ein Fehler aufgetreten ist

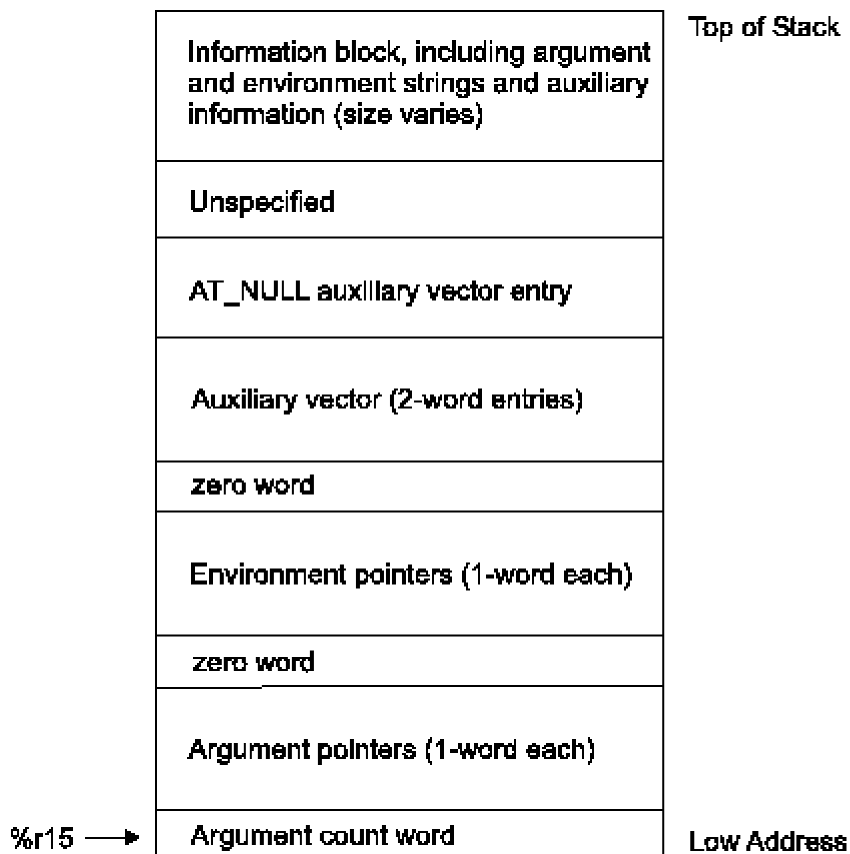
Layout einer Programmbinary

- Binaries liegen als **ELF-Datei** (*Executable and Linkable Format*) vor
  - enthalten **Programmcode** und **Metadata**
- beim Laden des Programmbinaries sind für das OS folgende Informationen wichtig:
  - **Adressen** an welche **Segmente** geladen werden sollen (*im Program-Header*)
  - **Adresse** an der die **Programmausführung beginnen** soll (*Label:* `_start`) (*im Datei-Header*)
- Binaries lassen sich mit `readelf` anzeigen

Start eines Programms

- mittels eines **System Calls**: `execve`
  - liefert bei erfolgreicher Ausführung kein Rückgabewert
  - stattdessen wird das aktuelle Programm durch das neue "ersetzt" (*Programmbinary wird im Speicher geladen und ein neuer Stack initialisiert*)
- **Syntax**: `./testprog arg1 arg2 ...`
- Inhalt der General-Purpose-Register am Anfang unspezifiziert

Stack bei Programmstart



- Programmargumente (!)

    - `argc`: Argument Count (hier zeigt `rsp`!)
    - `argv[argc]`: Argument Vector
        - `argv[0]`: Programmpfad
        - `argv[1], ... , argv[argc-1]`: Programmargumente
- Nullbyte
- Umgebungsvariablen (*irrelevant für GRA*)
- Auxiliary Vector (*irrelevant für GRA*)

Textaufgabe auf der Konsole

- mittels **System Calls** `write` und `stdout`

Beenden eines Programms

- `ret` oder keine weiteren Instruktionen **reichen nicht aus**
- mittels System Calls `_exit` (60) oder `exit_group` (231)
    - jeweils **1 Parameter** (*Exit Code des Programms, 0 = erfolgreiche Ausführung des Programms*)
    - **kein Rückgabewert**

Grundlagen - C

- **imperativ und prozedural**
    - nicht objekt-orientiert, basiert auf Funktionen und einfache Datentypen
- **standardisiert**
    - definiert Anforderungen an **konkrete Implementierung** des Standards
        - umfasst Compiler, Standardbibliothek, Betriebssystem und Hardware
        - meistens wird auf Rückwertskompatibilität geachtet
    - man unterscheidet zw. das durch den Standard definierte Verhalten und dem “implementation-defined behavior” (*z.B. konkrete Größen der Basisdaten*)
    - in GRA: **C17**

Datentypen in C

Type	Size (bits)	Size (bytes)	Range
char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int	16	2	-2^{15} to $2^{15}-1$
unsigned int	16	2	0 to $2^{16}-1$
short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int	32	4	-2^{31} to $2^{31}-1$
unsigned long int	32	4	0 to $2^{32}-1$
float	32	4	3.4E-38 to 3.4E+38
double	64	8	1.7E-308 to 1.7E+308
long double	80	10	3.4E-4932 to 1.1E+4932

Integers

- genaue Größen nicht garantiert (*siehe* `stdint.h`)
- standardmäßig **vorzeichenbehaftet** außer `char` und `_Bool`
    - durch `unsigned` wird angegeben, dass das Integer **nicht vorzeichenbehaftet** ist
        - bei `unsigned int` kann `int` weggelassen werden
    - analog für `signed`
- `_Bool`s sind standardmäßig 0 und werden bei der Zuweisung eines Wertes ungleich 0 wird dieser automatisch zu 1 konvertiert
- vorzeichenlose Zahlen haben **größeren positiven Wertebereich**
- **Overflows** nur für **vorzeichenlose / unsigned Zahlen** definiert

Beispielcode

```
unsigned long l = 42;
signed char c = -1;
unsigned i = UINT_MAX; // impl. unsigned int
```

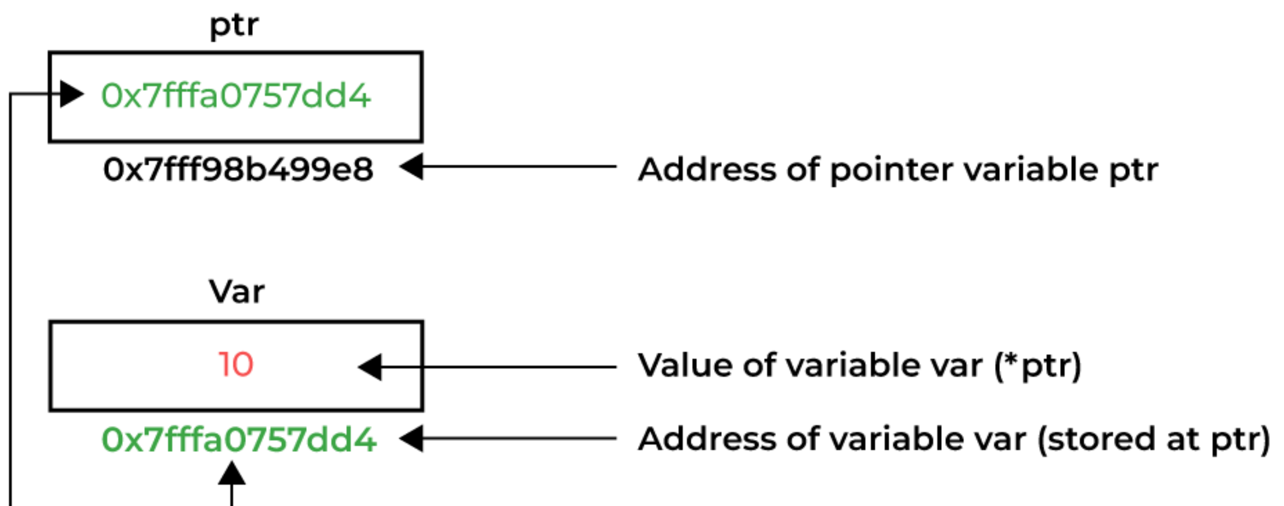
Floating-Point-Zahlen

- **stets vorzeichenbehaftet**

`void`

- **leerer Datentyp** (*z.B. kein Rückgabewert / keine Parameter*)

Pointer-Datentypen



- **Speicheradresse**, die mit einem bestimmten Datentypen verbunden ist
- zeigt auf **ein oder mehrere Elemente** eines bestimmten Datentyps
  - kann auch selber auf **anderen Pointer** zeigen
- Größe ist plattformabhängig (*LP64: 64 bits*)
- syntaktisch durch einen `*` hervorgerufen
- **Arrays** und **Strings** werden mittels Pointer umgesetzt

Funktionen in C

- enthalten **ausführbare Programmlogik**
- müssen **deklariert** und **definiert** werden

```
void foo ( int n ) ; // <-- Deklaration
void foo ( int n ) { // <-- Definition
    ...
}

void bar ( unsigned n ) { // <-- Deklaration + Definition
    ...
}
```

- **(!) Deklaration von Funktionen ohne Parameter**: `void` als **Parameterliste**

```
int foo ( void ) ; // <-- RICHTIG : akzeptiert keine Parameter

int bar () ; // <-- FALSCH : kann mit beliebigen Parametern
             // definiert / aufgerufen werden
```

- **Verlassen von Funktionen**: mittels `return` und Rückgabewert (*optional und ohne Rückgabewert bei `void`-Funktionen*)

```
void foo ( unsigned n , short s ) {
    ...
    return; // <-- kein Rückgabewert, hier optional
}

int bar ( long long multi_word_parameter ) {
    ...
    return -42; // <-- int als Rückgabewert
}
```

Die `main`-Funktion

- **Eintrittspunkt** des Programms
  - Programm beginnt in `main` und endet, sobald `main` fertig ist
- **Rückgabewert**: Exit Code
  - Standardisierte Konstanten `EXIT_SUCCESS` und `EXIT_FAILURE` (*unabhängig von Implementierung*)

```
int main ( void ) { // keine Parameter
    ...
    return EXIT_SUCCESS ; // 0
}

int main ( int argc , const char ** argv ) { // 2 Parameter (Anz. der Kommandozeilenargumente in argc und die Argumente als Array von Strings in argv)
// erstes Kommandozeilenargument ist üblicherweise Name des Programms
    ...
    return 1; // Implementation-defined error code
}
```

Variablen in C

- Variablen müssen vor Nutzung deklariert werden (*alloziert Speicherplatz für diese*)
  - Wert bis zur Zuweisung undefiniert
- bei konstanten Variablen muss die Zuweisung **gleichzeitig** mit der Deklaration geschehen

```
TYPE_NAME [ = VALUE ]; // Deklaration
const TYPE_NAME = VALUE; // Deklaration und Zuweisung einer konstanten
```

```
Variable
```

```
const int a;
a = 4; // COMPILER-FEHLER
```

- **Vorsicht bei `const` mit Pointer!**

```
const TYPE* PTR [= ADDR ]; // Pointer auf konstante Daten

TYPE* const PTR = ADDR ; // Konstanter Pointer auf
                         // variable Daten

const TYPE* const PTR = ADDR ; // Konstanter Pointer auf
                               // konstante Daten
```

Scopes

- Code-Blocks innerhalb von **geschweiften Klammern** definieren einen Scope
- beeinflusst **Sichtbarkeit** der Variablen innerhalb und außerhalb des Scopes

```
void foo () {
    int a = 42;
}

void bar () {
    int b = a ; // FEHLER : a ist nur in foo Sichtbar

    {
        int c = b ; // OK
    }

    int d = b ; // OK
    int e = c ; // FEHLER : c ist hier nicht mehr sichtbar
}
```

Zuweisung von Werten

```
int i ;
i = -2;             // negative Konstante im Dezimalsystem
i = 0xDEADBEEF ;    // Konstante im Hexadezimalsystem
i = 011;            // Konstante im Oktalsystem ( führende Null !!)
i = 'A';            // "character literal" - hier wird automatisch
                    // der entsprechende numerische Wert für den
                    // Buchstaben "A" eingefügt
                    // Siehe 'man 7 ascii' für eine Tabelle.
```

```
double d;
d = 2.0;           // double - Konstante
d = 2.0 f ;        // float - Konstante

int a = 2.0        // gibt a den Wert 2 als Integer
```

Arithmetische und logische Operatoren

- sei `unsigned a = 42;`

Operation	direkte Zuweisung	Bedeutung	Ergebnis	Ergebnis-Typ
`a = a + 42`	`a += 42`	Addition	84	unsigned
`a = a - 42`	`a -= 42`	Subtraktion	0	unsigned
`a = a * 42`	`a *= 42`	Multiplikation	1764	unsigned
`a = a / 5`	`a /= 5`	Division	8	unsigned
`a = a % 5`	`a %= 5`	Modulo	2	unsigned
`a = a && 0`	-	logisches UND	0	int
`a = a || 0`	-	logisches ODER	1	int
`a = !a`	-	logisches NOT	0	int
`a = a << 2`	`a <<= 2`	Linksshift	168	unsigned
`a = a >> 2`	`a >>= 2`	Rechtsshift	10	unsigned
`a = a & 0x3`	`a &= 0x3`	bitweises UND	2	unsigned
`a = a | 0x5`	`a |= 0x5`	bitweises ODER	47	unsigned
`a = a ^ 0xff`	`a ^= 0xff`	bitweises XOR	213	unsigned
`a = ~a`	-	bitweises NOT	4294967253	unsigned

Kontrollflussstrukturen

if-else-Bedingungen

```
if (x > 2.4) {
    ...
} else if (x < 0 x123456789) { // else if branch optional
    ...
} else { // else branch optional
    ...
}
```

while und do-while Schleifen

```
// while
while (n-- > 0) {
    ...
    if ( x == y ) {
        break; // beendet schleife
    }
    ...
}

// do...while
// code im schleifenkörper wird mindestens 1 mal ausgeführt
do {
    ...
    if ( x == y ) {
        continue; // bricht aktuelle iteration der schleife
    }
    ...
} while (--n > 0) ;
```

for-Schleifen

```
// Variante 0 - standard
for (int i = 0; i < 42; i++) { ... }

// Variante 1 - init. mehrere Variablen des gleichen Typs
for (int i = 0, j = 0; ...) { ... }

// Variante 2 - bereits deklarierte Variable
int k;
for (k = 0; k < 42; k++) { ... }

// Variante 3 - Schleife ohne Abbruchbedingung
for (;;) { ... } // analog zu "while (1) { ... }""

// Variante 4 - i-- im 2. Teil
for (unsigned i = n ; i-- > 0;) { ... }
```

switch-Statements

- Fallunterscheidung über Wert einer Integer-Variable

```
switch (x) {
    case -42:
```

```c
        ...
        break;
    case 'A':
        ...
        /* fall through */
    case 'B':
        ...
        break;
    default:
        ...
        break;
}
```

Der C-Präprozessor

- **Preprocessing** vor dem Kompilieren durch `cpp`
    - Auflösen von **Makros**
    - Kombination mehrerer **Dateien**
- Präprozessorbefehle beginnen mit `#`
- $\text{src.c} \xrightarrow{\text{Präprozessor}} \text{src.i} \xrightarrow{\text{Compiler}} \text{src.s} \xrightarrow{\text{Assembler}} \text{src.o} \xrightarrow{\text{Linker}} \text{a.out}$

Makros

```c
#define NUMBER 42    // Ersetze NUMBER durch 42
#define MYNUM 2 + 3 // Ersetze MYNUM durch 2 + 3

int a = NUMBER ;     // = 42
int b = MYNUM * 2;   // = 2 + 3 * 2 = 8 ( nicht (!) 10)

#undef MYNUM          // Mache Definition ruckgängig
```

- Präprozessor **kopiert** Inhalt an allen Stellen wo das Makro benutzt wurde

if-else-Konstrukte

```c
#define MYFLAG 0

#if MYFLAG
const char c = 'A ';
#else
const char c = 'B ';
#endif
```

```c
#if 0
int x = 42; // auskommentierter Code
#endif
```

`#include`-Direktiven

```c
#include <system_header.h> // Copy - paste Inhalte von system_header.h an diese Stelle
#include "local_header.h" // Copy - paste Inhalte von local_header.h an diese Stelle
```

- in `<>` wenn C-Standardbibliothek
- in `""` wenn Datei im Rahmen des eigenen Projekts

Header-Dateien

```c
// foo.h
void func(void);

// foo.c
#include "foo.h"

void func(void) {...}

// main.c
#include "foo.h"

int main(void) {
    func();
    return 0;
}
```

- **deklarieren Makros oder Funktionen** an einer Stelle, um an vielen Stellen benutzt zu werden
- Definitionen von Funktionen werden **nicht** in Header-Dateien vorgenommen

Sichtbarkeit

```c
// foo.h
void func(void);

// foo.c
#include "foo.h"

static void helper(void) {...}
void func(void) {...}
```

```c
// main.c
#include "foo.h"

static void helper(void){...}
int main(void) {
    func();
    return 0;
}
```

- **Problem**: `helper` wird sowohl von `main.c`, als auch von `foo.c` **definiert**, so dass `static` notwendig ist
- mittels Storage-Class-Specifier `extern` *(impl.)* und `static`
    - `extern`: Funktionen sind extern für andere C-Dateien sichtbar und können genutzt werden
    - `static`: beschränkt Sichtbarkeit nur auf eigene Datei (*in der die Funktion deklariert / definiert wird*)

Standard-Header

- kein `import`-System, die Standardbibliothek wird über Header benutzt

```c
// Systemweite Bibliotheksheader
# include < stdio.h >  // Input - Output Funktionalität
# include < string.h > // Funktionen zur Stringmanipulation

# include < stddef.h > // Definiert u.a. size_t (unsigned Typ, max. Größe von Objekten im Speicher).
                       // m.a.W. gibt es kein Speicherobjekt mit einer größeren Größe als size_t
                       // Bereits indirekt durch stdio.h eingebunden.

// Lokaler Header des Projekts
# include "myheader.h"
```

`stdint.h` und `stdbool.h`

- `stdint.h` definiert fixed-width Integer Typen

Signed	Unsigned	Größe
`int8_t`	`uint8_t`	8 Bit
`int16_t`	`uint16_t`	16 Bit
`int32_t`	`uint32_t`	32 Bit
`int64_t`	`uint64_t`	64 Bit

- `stdbool.h` enthält syntaktischen Zucker für boolsche Werte
    - `bool` für `_Bool`
    - `true` und `false` für 1 und 0

`printf`

```
// Hello World in C
# include < stdio .h > // <-- Wir brauchen die Deklaration von printf

int main (void) {
// Schreibe "Hello World!" gefolgt von einer Newline
    printf ("Hello World \n");
    return 0;
}
```

Format Strings

- `printf` bietet viele Ausgabemöglichkeiten

```
unsigned a = 0x42;
printf(" The value of a is : % u\n", a);
```

Conversion Specifiers

Specifier	Argumenttyp	Ausgabe
d	signed int	Dezimaldarstellung
u	unsigned int	Dezimaldarstellung
x / X	unsigned int	Hex-Darstellung
c	signed int	ASCII-Zeichen
s	`const char*`	String

- Optionale Angabe eines **Length Modifiers** vor dem Conversion Specifier
    - Bedeutung **abhängig** von Conversion Specifier
    - z.B. `%ld` für einen `long int`
- (*) bei `printf("%s", "test");` kann es möglicherweise zu keinem Output führen, da ohne Newline die Ausgabe ggf. in den Zwischenspeicher kommt

Disassemblieren mit `objdump`

- `objdump PROGNAME -d -M intel | less`

- `-d`: disassemble
- `-M intel`: Intel-Syntax statt AT&T
- `less`: übersichtlicher

Generierter Maschinencode

```
objdump -d -M intel gauss
```

```
000000000000064a <main>:
 64a:   55                      push  rbp                       ┐ alter Basepointer
 64b:   48 89 e5                mov   rbp,rsp                   ┘ gesichert
 64e:   48 83 ec 20             sub   rsp,0x20                  ] alloc. 32 Bytes für lok.Var.
 652:   89 7d ec                mov   DWORD PTR [rbp-0x14],edi  ┐ übergegebene Param. an main()
 655:   48 89 75 e0             mov   QWORD PTR [rbp-0x20],rsi  ┘ werden in lok. Var. gesichert
 659:   c7 45 f8 00 00 00 00    mov   DWORD PTR [rbp-0x8],0x0   ┐ init. sum und i
 660:   c7 45 fc 00 00 00 00    mov   DWORD PTR [rbp-0x4],0x0   ┘ mit 0
 667:   eb 0a                 ┌ jmp   673 <main+0x29>           springe an abbruchbedingung
 669:   8b 45 fc              │ mov   eax,DWORD PTR [rbp-0x4]   lade Wert von i in eax
 66c:   01 45 f8              │ add   DWORD PTR [rbp-0x8],eax   addiere Wert auf sum (sum += i)
 66f:   83 45 fc 01  schleife │ add   DWORD PTR [rbp-0x4],0x1   inkrementiere i (i++)
 673:   83 7d fc 64           │ cmp   DWORD PTR [rbp-0x4],0x64  führe schleife aus bis i <= 100
 677:   7e f0                 └ jle   669 <main+0x1f>
 679:   8b 45 f8                mov   eax,DWORD PTR [rbp-0x8]   ┐
 67c:   89 c6                   mov   esi,eax                   │
 67e:   48 8d 3d 9f 00 00 00    lea   rdi,[rip+0x9f]            │ printf
 685:   b8 00 00 00 00          mov   eax,0x0                   │
 68a:   e8 91 fe ff ff          call  520 <printf@plt>          ┘
 68f:   b8 00 00 00 00          mov   eax,0x0     Rückgabewert von main()
 694:   c9                      leave   räume Stack auf (equiv. zu "mov rsp, rbp; pop rbp")
 695:   c3                      ret     end
```

- links nach rechts…
  - **1. Spalte**: **Zeilennummer** (*in Bytes*) der Befehle in Hex
    - manche Befehle größer als 1 Byte
  - **2. Spalte**: Inhalte der **Binärdatei** (*Opcodes der Befehle*)
  - **3. Spalte**: Assemblercode

Optimierungsstufen

- **Usage (z.B)**: `gcc -O2 -o gaussO2 gauss.c`

Stufen

- `-O0`: keine Optimierung (*default*)
  - schneller Compilevorgang, gut lesbarer Maschinencode
- `-O1`: “Optimize”
  - bisschen Optimierung schadet nie
- `-O2`: “Optimize even more”
  - “best of both worlds”

- `-O3`: “Optimize yet more”
    - kann zu großem Maschinencode führen

Weitere Optimierung

- `-Ofast`: `-O3` + float-Optimerungen
    - kann zu Rundungsfehler führen
- `-Os`: wie `-O2`, aber möglichst kleine Ausgabedatei
- `-Og`: wie `-O1`, stattdessen gut debugbarer Code

```
0000000000000560 <main>:
 560:   48 8d 35 bd 01 00 00    lea  rsi,[rip+0x1bd]
 567:   48 83 ec 08             sub  rsp,0x8
 56b:   ba ba 13 00 00          mov  edx,0x13ba
 570:   bf 01 00 00 00          mov  edi,0x1
 575:   31 c0                   xor  eax,eax
 577:   e8 c4 ff ff ff          call 540 <__printf_chk@plt>
 57c:   31 c0                   xor  eax,eax
 57e:   48 83 c4 08             add  rsp,0x8
 582:   c3                      ret
```

Extra: Godbolt Compiler Explorer

- generiert Maschinencode online aus Source Code

```c
int testFunction(int* input, int length) {
  int sum = 0;
  for (int i = 0; i < length; ++i) {
    sum += input[i];
  }
  return sum;
}
```

```
testFunction:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     DWORD PTR [rbp-28], esi
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-8]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        add     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-8], 1
.L2:
        mov     eax, DWORD PTR [rbp-8]
        cmp     eax, DWORD PTR [rbp-28]
        jl      .L3
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Debugging mit GDB

- mit `gcc`: `-O0 -g debug.c`
    - `-O0`: for obvious reasons

- `-g`: generiert alle notwendigen Dateien für Debugging (*nur mit Debugging benutzen!*)
- z.B. `gcc -o debug -O0 -Wall -Wextra -g debug.c`

Ausführen von Programme in GDB, Breakpoints, Programmfluss

- mit `gdb PROGNAME`
  - ladet Debugginginformationen aus Executable
- GDB-Befehle:
  - `run / r argv[0] argv[1]...`: führt Programm (*mit Argumenten*) aus
    - kann auch zum Neustart des Debuggen verwendet werden
  - `break / b PROGNAME:LINENUM` / `b LABEL`: setzt Breakpoint an Zeilennummer / Label / Funktion im Code
    - GDB stoppt Ausführung bei Breakpoint und zeigt die als nächstes auszuführende Zeile
  - `print / p VARIABLE / $REGISTER`: zeige Variablen- bzw. Registerinhalt an
    - `p (len == $rax)`: bestätigt, dass `len` und `$rax` entsp. Calling Convention denselben Wert haben
    - `p POINTER`: gibt Adresse von Pointer aus
  - `x ADDR`: zeigt Speicherinhalt an Addresse
    - `x ARRAY_NAME`: zeigt Addresse und erstes Element eines Arrays
    - `x (ARRAY_NAME + 1)`: zeigt 2. Element in Array
    - `x/LEN ARRAY_NAME`: zeigt alle Elemente des Arrays
    - `x/a`: gibt Adresse aus
    - `x/s`: gibt String aus (*alle Zeichen bis zum Ende des Strings*)
    - `x/d`: gibt Dezimalzahl aus
    - wenn kein Format Specifier angegeben wird, verwendet GDB das letzte verwendete Format Specifier
  - `step / s`: führt nächste Zeile im Code aus und stoppt erneut Ausführung (*step-into*)
    - wenn nächste Zeile Funktionsbeginn markiert, wird in die Funktion gesprungen
  - `stepi / si`: führt einzelne Assembler-Instruktion aus
    - `s` und `si` bei ASM-Debugging identisch
  - `info break`: zeigt alle Breakpoints
  - `delete`: löscht alle Breakpoints
    - `delete BREAKPOINT_NUM`: löscht bestimmten Breakpoint

    - `disable / enable BREAKPOINT_NUM`: temporäres (de)aktivieren eines Breakpoints
    - `next / n`: springt direkt in die nächste Zeile nach Funktionsaufruf (*step-over*)
    - `continue / c`: setzt Programmausführung bis zum nächsten Breakpoint fort
    - `finish`: setzt Ausführung bis zum Verlassen der aktuellen Funktion fort
    - `quit`: beendet Debugging und löscht alle Breakpoints
- `~./gdbinit`: zeigt Inhalte aller General-Purpose-Register an

Bsp: Array von Pointer und Strings

```
(gdb) p argc
$5 = 3

(gdb) x/3a argv
0x...: 0x... 0x...
0x...: 0x...

(gdb) x/s argv[1]
0x...: "hello"
```

Optimierungen

- **Optimerungsansätze**:
    - algorithmische / mathematische Optimierungen
    - Wahl der Programmiersprache
    - Compiler-spezifische Optimierungen
    - Hardware-spezifische Optimierungen
- **Optimierungsziele** (*beispielshalber*):
    - Laufzeit
    - Speicherplatz
- Optimierter Code ist meistens…
    - aufwändiger zu schreiben
    - schwerer zu lesen
    - komplizierter zu testen

Optimierung der Fibonacci-Reihenfolge

Basis-Fibonaccifunktion

```
# include <stdint.h>
uint64_t fib1 ( uint64_t n ) {
```

```
    if(n <= 1) {
        return n ; // fib (0) = 0 , fib (1) = 1
    }
    return fib1(n - 1) + fib1(n - 2);
}
```

- sehr lange Laufzeit für $n > 40 \ldots$

Laufzeitklassen

- (Komplexe) **Laufzeit** eines Algorithmus: $f(n)$
- $f(n)$ wächst vergleichbar zu einer “simplen” Funktion $K(n)$
    - $K(n)$: **Laufzeitklasse** des Algorithmus
- **beste Optimierung**: Laufzeitklasse des Algorithmus ist entscheidend
    - andere Mikrooptimierungen sind zunächst **unnötig**

Optimierung für kleine Eingabewerte

- **schlechtere Laufzeitklassen** sind **möglicherweise schneller**
    - konstante Faktoren und Offsets sind ausschlaggebend
- muss **individuell** getestet werden

Fibonacci: Lineare Schleife statt Doppelter Rekursion

```
uint64_t fib2 ( uint64_t n ) {
    if (n == 0) { // base case
        return 0;
    }
    if (n > 93) { // integers greater than 93 cannot be represented using
uint64_t
        return UINT64_MAX ;
    }

    uint64_t a = 0;
    uint64_t b = 1;
    uint64_t i = 1;
    for (; i < n ; i++) {
        uint64_t tmp = b ;
        b += a ;
        a = tmp ;
    }
```

```
    return b ;
}
```

Fibonacci: Logarithmische Laufzeit mit Formel von Binet

- $fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$
    - **+**: Logarithmische Laufzeit
    - **-**: Fließkommazahlen mit begrenzten Nachkommastellen

Fibonacci: Lookup-Table (LUT)

- da nur die ersten 94 Fibonaccizahlen darstellbar sind, kann man diese vorberechnen und in einer LUT speichern
- der Algorithmus schlägt Werte einfach in LUT nach

```
// All 94 64 - bit fibonacci numbers ( n = 0 ,... ,93)
uint64_t lut[] = {0 ,1 ,1 ,2 ,3 ,5 ,8 ,13 ,21 ,34 ,55 ,89 ,144 ,233 ,377,
..., 7540113804746346429 ,12200160415121876738};

uint64_t fib3 (uint64_t n) {
    if (n > 93) {
        return UINT64_MAX;
    }

    return lut[n];
}
```

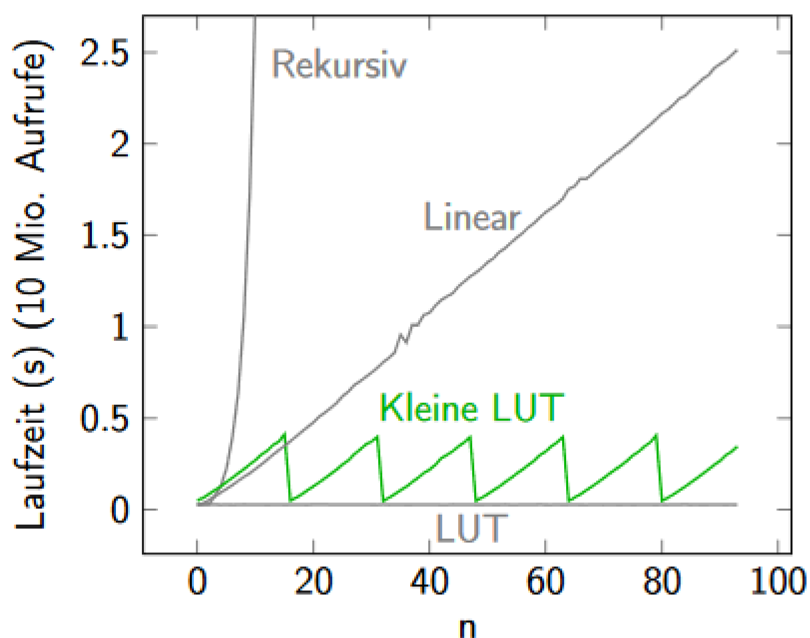
- **+**: schnellster Ansatz (*konstante Laufzeit*)
- **-**: deutliche Vergrößerung des Programms $\longrightarrow$ Speicherplatzprobleme

Speicherplatzoptimierung: LUT verkleinern

- Unterteilen der LUT in **Abschnitte**
    - **erste zwei Werte** jeden Abschnitts speichern
    - restliche Werte **dynamisch** zur Laufzeit berechnen
        - z.B. 6 Abschnitte mit je 16 Zahlen

```
# include < stdint .h >

// LUT for n = {0 ,16 ,32 ,48 ,64 ,80}
uint64_t lut0 [] = {0 ,987 ,2178309 ,4807526976 ,10610209857723 ,
23416728348467685};

// LUT for n = {1 ,17 ,33 ,49 ,65 ,81}
```

```
uint64_t lut1 [] = {1 ,1597 ,3524578 ,7778742049 ,17167680177565 ,
37889062373143906};

uint64_t fib4 ( uint64_t n ) {
    // case for numbers exceeding 64-bit limit
    if ( n > 93) {
        return UINT64_MAX ;
    }

    // calculate index in interval to find first 2 numbers of interval
    uint64_t index = n / 16;
    uint64_t a = lut0 [ index ];
    uint64_t b = lut1 [ index ];

    // get pos. of first fibonacci number in interval
    index *= 16;
    if ( index == n ) // if number is already saved, return it
        return a ;

    // calculate fibonacci number using standard loop
    index++;
    for (; index < n ; index ++) {
        uint64_t tmp = b ;
        b += a ;
        a = tmp ;
    }

    return b ;
}
```



Komplexe Datenstrukturen in C

- **zusuammengesetzte Datentypen**: Arrays, `struct`
- `union`: Zugriff auf gleichen Speicherbereich über unterschiedliche Identifier
- **“Speicherobjekt” / “Objekt”**: Speicherbereich eines bestimmten Typs (*Variablen, Parameter etc.*)

`sizeof`-Operator

- wird **wie eine Funktion** genutzt
- ermittelt die **Größe** des als Argument übergegebenen Datentyps in **Byte**
    - ermittelte Werte sind **plattformabhängig** (*Ausnahme:* `sizeof(char) == 1`)
- kann auch auf **Variablen** aufgerufen werden

```
size_t size;
size = sizeof(char);    // 1
size = sizeof(size_t); // 8
size = sizeof(void *); // 8

size_t a = sizeof(size_t);
size_t b = sizeof(a);
// a == b, da 'a' vom Typ 'size_t' ist
```

Alignment

- Anforderung an **Ausrichtung der Speicheradresse** eines Objekts
    - **Speicheradresse** muss **Vielfaches** vom Alignment sein
    - **Größe eines Datentyps** ist **Vielfaches** des Alignments
        - bzw. Alignment muss **Teiler der Größe** sein
- **implementation-defined**: abhängig vom Compiler-Toolchain und Hardware
- **Beispiel**: `int`
    - **Größe**: 4 Byte
    - **Alignment** entweder 1, 2 oder 4 Byte (*üblicherweise 4*)
- **Beispiel**: `char`
    - **Größe**: 1 Byte
    - **Alignment**: 1 Byte

Pointer

- **Pointer**: Adresse eines Speicherobjekts, “zeigt” auf das Objekt

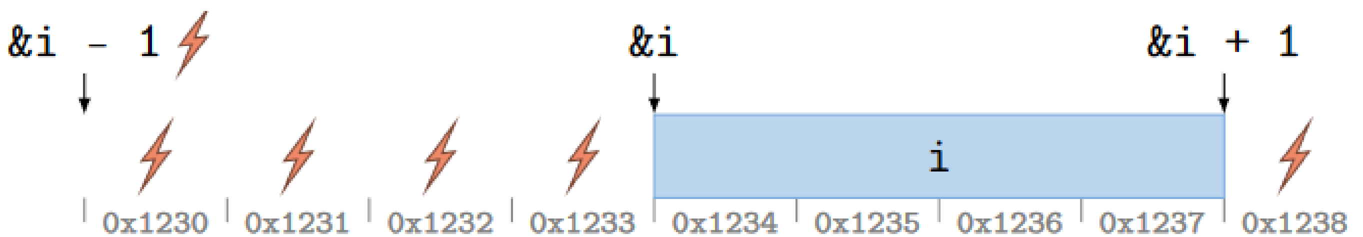
- **Arrays**: zusammenhängender Speicherbereich, enthält Datenobjekte gleichen Typs direkt aufeinanderfolgend
    - Arrays können ebenfalls über Pointer genutzt Werden
- `&`: nimmt Adresse eines Objekts
- `*`: dereferenziert Pointer

```
int i = 0;
int* i_ptr = &i; // declare pointer variable of type int (int*) that points to address of i (&i)
int i_new = *i_ptr; // deref. pointer, i_new == i
```

Pointerarithmetik

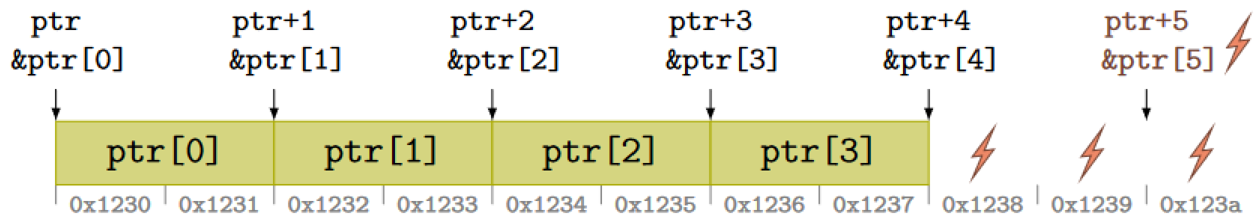
- Pointerarithmetik geschieht immer mit dem **Vielfachen der Größe des Datentyps**
    - z.B. bei Inkrementierung eines `int*`s wird **4** darauf addiert

```
// Pointerarithmetik
int i = 0;
int* i_ptr = &i; // 0x1234
i_ptr++;         // 0x1238
i_ptr -= 2;      // 0x1230 (undefined behavior, i_ptr points to element outside of array)
```



Pointerarithmetik - Arraysubscript

- `ptr[0] == *ptr`
    - `ptr = &ptr[0]`
- `ptr[3] == *(ptr + 3)`
    - `ptr + 3 = &ptr[3]`
- *alte Syntax*: `ptr[index] == index[ptr]`
- valide Pointer zeigen immer auf ein Objekt, eine Stelle in einem Array oder an das Ende eines Arrays (dann nicht dereferenzierbar), andernfalls ist das Verhalten undefiniert!



`void`-Pointer und implizite Pointer-Casts

- **(!)** `void`-Pointer **nicht** dereferenzierbar
- **Inkrementieren und Dekrementieren** eines `void`-Pointers ist **undefiniert**
- **Casting** zwischen `void`-Pointer und Pointer eines anderen Typs ist **implizit**

```
int* i_ptr = ...;
void* v_ptr = i_ptr; // every pointer can become a void-pointer
int* i_ptr2 = v_ptr; // a void-pointer can become any pointer
int i = *v_ptr;      // compiler error!
```

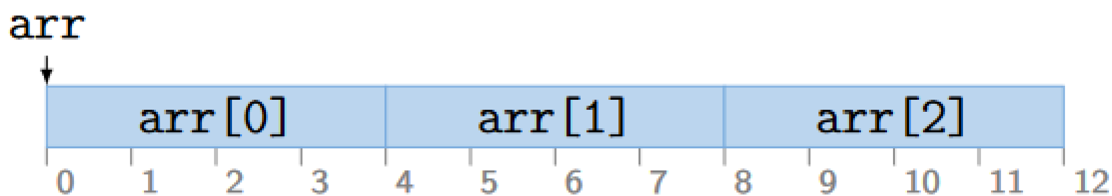
Explizite Pointer-Casts

- explizite Typumwandlung
- neuer Datentyp darf **kein strengeres Alignment** fordern
- Derefenzierung von umgewandelten Pointern ist **undefined behaviour** (*Ausnahme:* `char*`)
    - explizite Casts sollten daher vermieden werden

```
int* i_ptr = ...;
char* c_ptr = (char*) i_ptr;   // zugriff möglich
short* s_ptr = (short*) i_ptr; // zugriff undefined
long* l_ptr = (long*) i_ptr;   // cast undefined wegen strengeren alignment-anforderungen von long
```

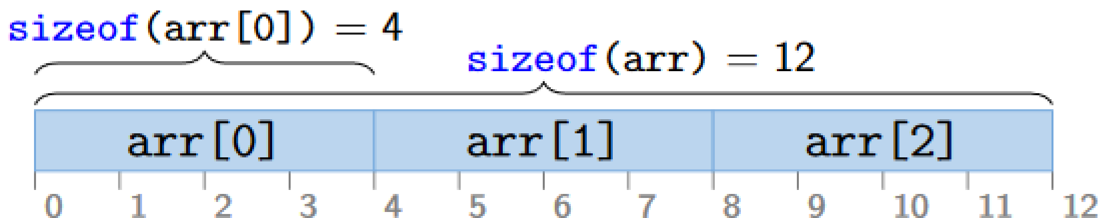
Arrays

- Speicherbereich mit **Datenobjekten gleichen Typs direkt aufeinanderfolgend**
- **Zugriff**: syntaktisch **Analog zu Pointern** (z.B. `arr[0]` oder `*arr`)
- **Deklaration**:
    - mit impliziter Größenangabe: `int arr[3]`
    - Deklaration + Definition in einem Statement mittels “Compound Literal”: `int arr[3] = {1,2,3}`
    - Verzicht auf explizite Größenangabe: `int arr[] = {1,2,3}`



- **Bestimmung der Größe** eines Arrays mittels `sizeof`
    - (!) **gesamte Größe** muss durch **Elementgröße dividiert** werden

```
int arr[3] = {1,2,3};
size_t arr_len = sizeof(arr) / sizeof(arr[0]); // 12 / 4 = 3
```



- Arrays mit **variabler Größe**: `void func(size_t size){ char buf[size]; }`

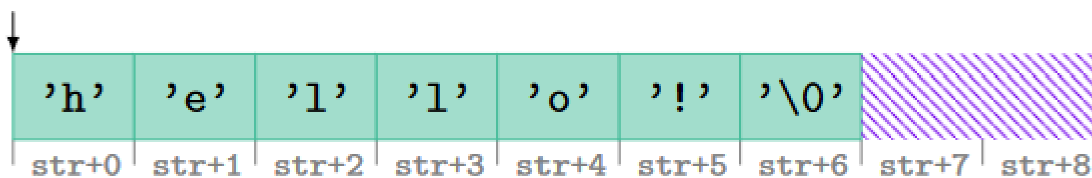
Arrays als Parameter

```
// pointersyntax
void func ( int * arr , size_t arr_length ) { // sizeof(arr) ermittelt NICHT
größe des arrays!
    for ( size_t i = 0; i < arr_length ; i ++) {
        printf ( " % d \ n " , arr [ i ]) ;
    }
}

int main ( void ) {
    int arr [3] = {1 ,2 ,3};
    func ( arr , sizeof ( arr ) / sizeof ( arr [0]) ) ;
    // ...
}

// arraysyntax
// parameter sind immer noch pointer!
void func1 ( int arr [ arr_length ] , size_t arr_length ) {
    for ( size_t i = 0; i < arr_length ; i ++) {
        printf ( " % d \ n " , arr [ i ]) ;
    }
}
void func2 ( int arr [3]) { // sizeof(arr) würde immer noch größe des
pointers zurückgeben!
```

```
    for ( size_t i = 0; i < 3; i ++) {
        printf ( " % d \ n " , arr [ i ]) ;
    }
}
int main ( void ) {
    int arr [3] = {1 ,2 ,3};
    func1 ( arr , sizeof ( arr ) / sizeof ( arr [0]) ) ;
    func2 ( arr ) ; // ...
}
```

Strings

- repräsentiert als `char`**-Array**
  - **Länge implizit**: 0-Byte `\0` als Terminal
- ASCII-Zeichensatz



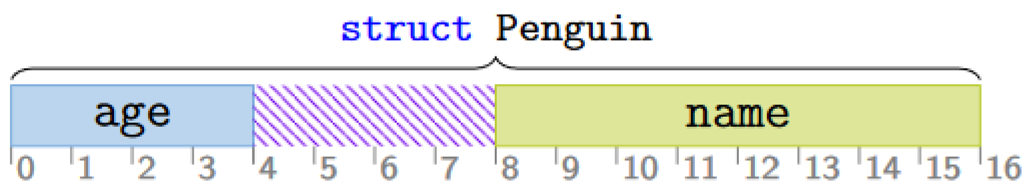
- **Definitionsmöglichkeiten**:
  - Array mit explizitem Null-Byte: `char str[] = {'t','u','x','\0'};`
  - String-Literal mit implizitem Null-Byte: `char str[] = "tux";` (*äquiv. zur vorherigen Variante*)
  - Pointer statt Array: `char* str = "tux";` (*String hier nicht modifizierbar, sollte als* `const char*` *definiert werden!*)

`struct`s

- **zusammengesetzter Datentyp** mit **Elementen verschiedenen Typs**
- Elemente liegen **aufeinanderfolgend** (*ggf. mit Padding, also kgV der Alignment-Anforderungen im Zusammenhang zur Deklarationsreihenfolge*) **im Speicher**

```
struct Penguin{
    int age;
    char* name;
}

struct Penguin1 { // size: 8
    int id; // 4
    unsigned char age; // 1
```

```
    char color; // 1
    // 2 byte padding...
    // 4 + 1 + 1 + 2 = 8
}

struct Penguin2 { // size: 12
    unsigned char age; // 1
    // 3 byte padding...
    int id; // 4
    char color; // 1
    // 3 byte padding...
    // 1 + 3 + 4 + 1 + 3 = 12
}
```



- **Zugriff** direkt oder über einen Pointer

```
penguin.age = 0; // type of penguin is struct Penguin
penguin_ptr -> age = 0; // type of penguin_ptr is struct Penguin*
```

- **Initialisierung** eines `struct`s über "**Compound Literal**" oder indem alle Werte auf **null** gesetzt werden

```
// compound literal
struct Penguin penguin1 = { .age = 0, .name = "Tux" };
struct Penguin penguin2 = { .age = 0 }; // impl. penguin2.name = NULL
struct Penguin penguin3 = { 0, "Tux" }; // festgelegte Reihenfolge

// NULL
struct Penguin penguin4 = { 0 };
```

`struct` als Parameter

- **by value** oder über **Pointer**
    - wenn `struct` by Value übergegeben wird und in der Funktion modifiziert wird, wird **nur die Kopie** modifiziert und nicht das eigentliche `struct`-Objekt

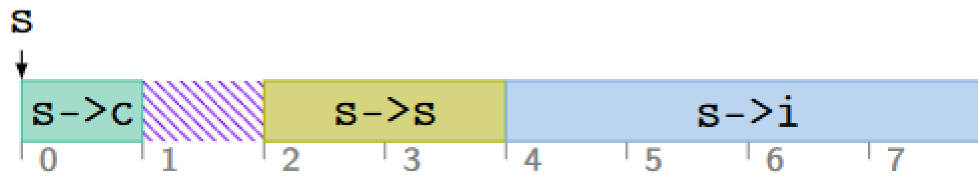
```
struct Penguin {
    char * name ;
    unsigned age ;
};
```

```c
void print_penguin_name1 ( struct Penguin * penguin ) {
    printf ( " name : % s \ n " , penguin - > name ) ;
}
void print_penguin_name2 ( struct Penguin penguin ) {
    printf ( " name : % s \ n " , penguin . name ) ;
}
int main ( void ) {
    struct Penguin penguin = { " tux " , 5 };
    print_penguin_name1 (& penguin ) ;
    print_penguin_name2 ( penguin ) ;
}
```

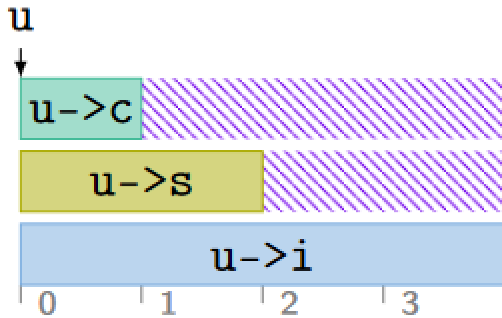
`struct` vs. `union`

- `union` erlaubt Zugriff auf Speicherbereich mit unterschiedlichen Datentypen
  - sinnvoll, wenn von verschiedenen Datentypen **nur maximal einer** benutzt wird
  - Objekte können an **der selben Speicheradresse** gespeichert werden → Speicherplatz wird gespart
  - nur das **zuletzt geschriebene Element** wird gespeichert
- `struct` speichert immer **alle Elemente** an **aufeinanderfolgenden Adressen**
- **(!)** bei **anonymen `struct`s** und bei **anonymen `union`s** werden die Member zu Membern der übergeordneten Struktur

```c
struct Penguin {
    struct {
        unsigned height ;
        unsigned age ;
        unsigned id ; // oops
    };
    char id [256]; // compiler error: attribute id already "defined"
};
```



union { char c; short s; int i; }* u;



`struct` mit `union`

- Kombination mit `struct` mit Indikator für Gültigkeit der `union`-Elemente

```
struct Dimension { ... };
struct Shape {
    int shape_kind; // 1 = circle, 2 = rect
    union {
        int circle_radius;
        struct Dimension rect;
    };
};

struct Shape my_circ = { .shape_kind = 1, { .circle_radius = 10 }};
```

`union` als Parameter

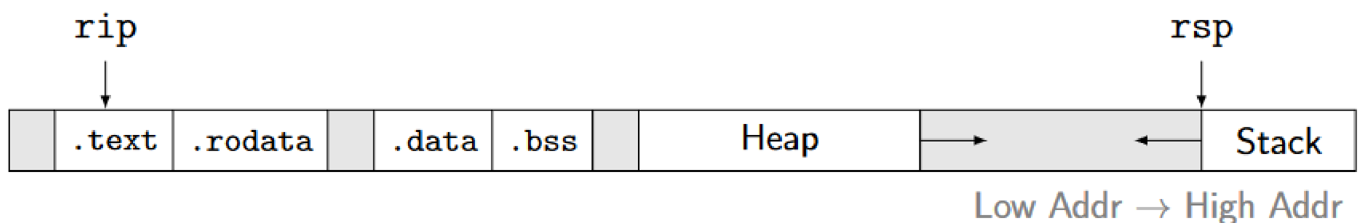
- Zugriff **syntaktisch analog** zu `struct`

```
void f(union Number num){
    // Zugriff auf 'a' mit 'num.a'
}

void g(union Number* num){
    // Zugriff auf 'a' mit 'num->a'
}
```

Mehrdimensionale Arrays

```c
unsigned matrix [3][4] = {
    { 1 , 2 , 3 , 4 } , { 5 , 6 , 7 , 8 } , { 9 , 10 , 11 , 12 }
};
for ( size_t i = 0; i < sizeof ( matrix ) / sizeof (* matrix ) ; i ++) {
    for ( size_t j = 0; j < sizeof ( matrix [ i ]) / sizeof (* matrix [ i ])
; j ++) {
        printf ( " % u " , matrix [ i ][ j ]) ;
    }
}
printf ( " \ n " ) ;

// Auslassen der expliziten Größenangabe der "obersten Ebene":
unsigned matrix [][4] = {
    { 1 , 2 , 3 , 4 } , { 5 , 6 , 7 , 8 } , { 9 , 10 , 11 , 12 }
};
```

Speicherbereiche



- `.text`: beinhaltet **Programmcode** (*read-only, executable*)
  - `rip` zeigt auf den zunächst auszuführenden Befehl innerhalb des Codes
- `.rodata`: beinhaltet **globale konstante initialisierte Variablen** (*read-only*)
  - z.B. `const int i = 42;` (*global*)
- `.data`: beinhaltet **globale initialisierte Variablen** (*read-write*)
  - z.B. `int i = 42;` (*global*)
- `.bss`: beinhaltet **globale Variablen**, die mit **0** initialisiert sind
  - (!) **nur globale Variablen** können so initialisiert werden, da die selbe Deklaration innerhalb einer Funktion der Variable einen **undefinierten** Wert zuweisen würde
  - z.B. `int i;` (*global*)

Stack vs. Heap

- **Stack**:
  - für **kleine Datenmengen** (*zw. 8 und 16 MB in LP64*)

    - wächst **von oben nach unten**
    - **LIFO**
    - enthält **lokale Variablen** einer Funktion
    - **automatische Speicherfreigabe** (*Referenzen zu Elementen auf dem Stack dürfen nicht zurückgegeben werden*)
- **Heap**:
    - für **größere Datenmengen** (*z.B. Liste mit abstrakten Datentypen*)
    - **dynamische Allokation und Freigabe**, Größe in der Regel nur vom physisch vorhandenen Speicherplatz eingeschränkt
    - Allokationen **global verwendbar**

Speicherverwaltung auf dem Heap

- **Speicherallokation**:
    - `void* malloc(size_t size)` und `void* calloc(size_t nmemb, size_t size)` aus `stdlib.h` reservieren Speicher auf dem **Heap** und **müssen freigegeben werden**!
        - Pointer zeigt auf **Beginn des Speicherbereichs**
        - `void* alloca(size_t size)` reserviert Speicher auf dem **Stack** und muss nicht wieder freigegeben werden
    - **(!)** im Falle eines **Fehlers**: `NULL`-Pointer als Rückgabewert
- **Speicherfreigabe**:
    - **kein Garbage Collector**
    - `void free(void* ptr)` aus `stdlib.h`
        - **(!)** nur **original** `ptr` von `malloc` bzw. `calloc` freen!
        - **(!)** nach `free` soll `ptr` **nicht mehr** für Speicherzugriffe bzw. Speicherverwaltung verwendet werden!

```c
// Beispiel: Allokation auf dem Heap
char* p = malloc(256 * sizeof(char));
if ( p == NULL ) {
    // Behandlung von Fehler bei Speicherallokation
    abort();
}
// ... arbeite mit p
free(p);
```

- `void* realloc(void* ptr, size_t size)` vergrößert / verkleinert bereits reservierter Speicher
    - alter Speicherbereich wird bei Erfolg automatisch freigegeben

    - *Vergrößerung*: neue Daten uninitialisiert
    - `realloc(NULL, size)` equiv. zu `malloc(size)`
    - **(!)** im Falle eines **Fehlers**: `NULL`-Pointer als Rückgabewert und alter Speicherbereich wird **nicht** freigegeben
- `void* aligned_alloc(size_t alignment, size_t size)`
    - `alignment` muss **Zweierpotenz** sein
    - `size` muss Vielfaches des `alignment`s sein

Effizientes Debugging

- mittels `-Wall` und `-Wextra`

`printf`-Debugging

- `printf`s sind gebuffert
    - `\n` leert den Buffer
    - `fflush(stdout)` leert ihn auch

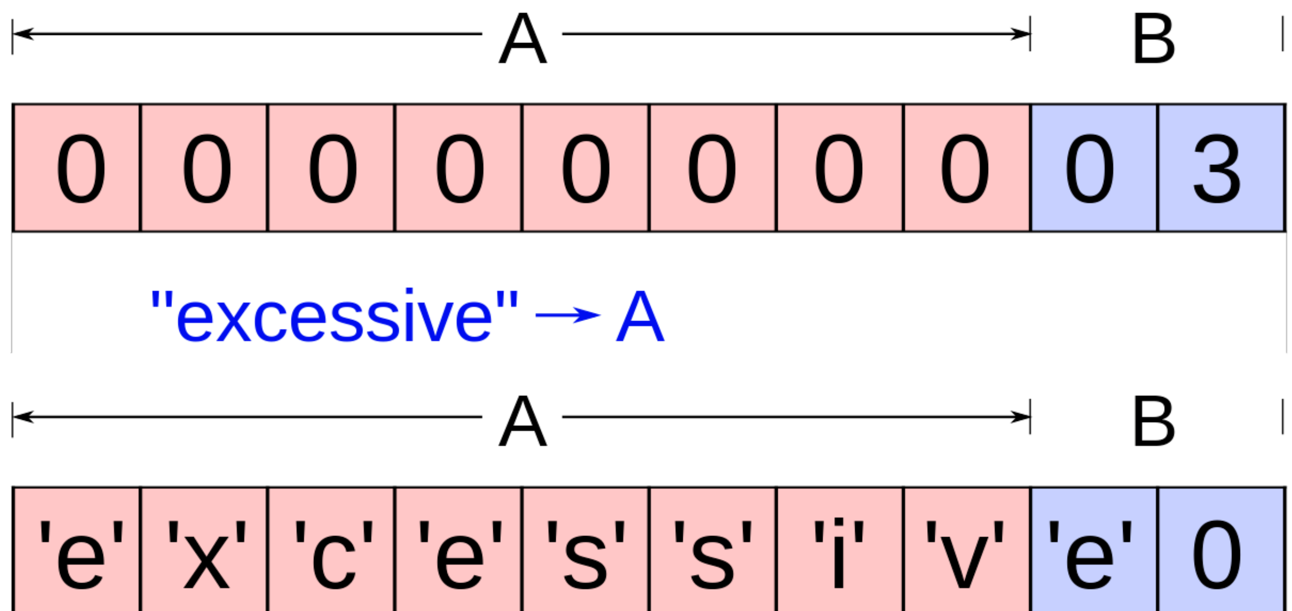
`assert()`

- in `assert.h`
    - mit `-DNDEBUG` wird festgestellt, dass `assert()` als NOP funktioniert
    - ansonsten wird das Programm abgebrochen, wenn `assert()` false zurückliefert
- **(!)** man soll **nie** annehmen, dass Assertions den Code abbrechen
    - `assert` soll nur in der Testphase verwendet werden

Buffer Overflows

- **Buffer** sind **ordinäre Speicherberieche** und sind in C notwendig
    - z.B. Nutzereingaben oder Daten ablegen etc.
- **Buffer Overflow**: Lese- bzw. Schreibzugriff außerhalb der Grenzen des Buffers
    - häufig auftretende Sicherheitslücke
    - *beispielhafter Angriff*: Überschreiben der Rücksprungadresse auf dem Stack
    - ermöglicht Sprünge zu beliebigen Funktionen im Programm

- Overflow von nur einem Byte kann die Sicherheit beeinträchtigen



- `A` hat platz nur für 4 `char`s (inkl. `\0`)
- `strcpy()` darf jedoch aufgerufen werden, der Code kompiliert noch (*obwohl die meisten Compiler den Overflow erkennen und davor warnen werden*) und wird ausgeführt
- **Die Bytes in `B` werden somit überschrieben**

Segmentation Fault

- **Speicherschutzverletzung**, ausgelöst durch…
  - …schreibenden Zugriff auf read-only Daten
  - …Zugriff auf Daten mit fehlenden Berechtigungen (*z.B. Kernel-Daten als Benutzer*)
  - …NULL-Pointer-Dereferenzierung
  - …fehlerhafte Speicherzugriffe durch Verletzung der Calling Convention (*ASM*)
  - etc.
- **Ansätze zur Vermeidung**:
  - **Pointer** vor Benutzung überprüfen (*z.B. auf NULL-Wert*)
    - z.B. `fopen` gibt bei Fehler `NULL` zurück
  - Fehler bei Einhaltung der **Array-Grenzen**
    - Off-By-One Fehler (*Index beginnt bei `0` und endet bei `arrLen - 1`*)
    - Hardcoding von Arraygrößen
  - Beachtung des **Nullterminals** bei C-Strings überprüfen
    - Vergessen des Nullterminals
    - Speicher zu klein für Nullterminal

Inhärent Unsichere Funktionen

- wenn möglich, Funktionen verwenden, die **Schutzmechanismen** erzwingen
  - z.B. `memcpy(void* dest, const void* src, size_t n)`, die eine explizite Angabe der Datengröße `n` fordert
  - solche Schutzmechanismen sollte man auch in den eigenen Funktionen einbauen
- **C-Standardbibliothek bietet viele potentiell gefährliche Funktionen an**!
  - `gets(char* buf)`
    - ab C11 nicht mehr Teil des Standards
    - liest Daten in `buf`, bis `EOF` oder `'\n'` erkannt wird
  - `scanf()`
    - Angabe eines Format-Strings kann maximale Zeichen einschränken (*z.B.* `scanf("%5s", buf)`)
    - *Problem*: `scanf("%s", buf)` (*ohne Einschränkung*) ist weiterhin erlaubt
    - stattdessen sollte **immer** `fgets(char* dest, int n, FILE* stream)` verwendet werden
      - höchstens `n - 1` oder EOF bzw. `\n` Zeichen werden eingelesen
      - String in `dest` wird **nullterminiert** $\rightarrow$ Buffer-Overflows durch Nullterminal werden vermieden
  - `strcpy(char* dest, const char* src)`
    - kopiert `src` nach `dest`, inklusive Nullterminal von `src`
    - *Problem*: `size(src) > size(dest)` $\rightarrow$ Buffer-Overflow
    - *bessere Alternative*: `strncpy(char* dest, const char* src, size_t n)`
      - `n` bestimmt, wie viele Bytes maximal kopiert werden
      - **Resultat ist nur nullterminiert, wenn ein Nullterminal in den `n` Bytes von `src` existiert!**
        - z.B. `char d[3]; char* s = {'1','2','3'}`
        - `strncpy(d,s,3)` ergibt `d = {'1','2','3'}`
  - `printf(buf)`, wobei `buf` User-Input enthält
    - da der Inhalt von `buf` aufgrund des Einlesens mit `scanf` durch den Nutzer frei wählbar ist, kann dieser dort auch Format Specifier wie `%s`, `%d`, etc. angeben
    - `printf` wird diese Format Specifier dann wie üblich interpretieren und als Konsequenz evtl. den Inhalt von Registern- und/oder des Stack-Speichers ausgeben

Überprüfung von `malloc()`

- Rückgabewert von `malloc()` bei Fehler ist NULL-Pointer
- Derferenzierung des NULL-Pointers ist undefiniertes Verhalten
- *Vermeidung*: immer überprüfen, ob `malloc()` NULL zurückgibt

Format String Injection

- `printf()` benutzt pro Format Specifier einen Parameter
- Parameter sind laut Calling Convention automatisch **immer die Register und danach der Stack**
  - bei **Einlesen von Format Specifiern** werden **Register / Stack als Parameter interpretiert**
    - User kann somit **Speicher mit `%x`, `%s` etc. leaken** oder mit `%n` **schreiben**
- *Vermeidung*: Kombinieren der `printf`-Aufrufe und Benutzung eines Formatstrings: `printf("Hello %s!\n", buf)`

Memory Leak

- angeforderter Speicher wird nicht freigegeben
- in größeren Programmen kann es dazu führen, dass der Speicherbedarf unkontrolliert wächst
- *Vermeidung*: Freigabe von unbenötigtem Speicher durch `free` vor `return`-Statements
  - *Ausnahme*: Rückgabe von heapalloziertem Speicher (*Rückgabe von Pointer durch `malloc` oder `calloc`, da man diesen vllt. später noch braucht*)

Use After Free und Double Free

- **undefiniertes Verhalten**

```
if (! strlen ( buf ) ) {
    printf ( " You didn 't enter your name !\ n " ) ;
    return 1;
} else if ( strlen ( buf ) > 20) {
    printf ( " You have a really long name , % s !\ n " , buf ) ;
    free ( buf ) ;
}
// buf gets used after being freed -> use after free
printf ( " Thank you for introducing yourself , % s !\ n " , buf ) ;
// buf gets freed again -> double free
free ( buf ) ;
```

Undefined Behavior

- **Programm weicht vom C-Standard ab**
    - Dereferenzierung von Nullpointer
    - Double Free
    - Use after Free
    - Lesen uninitialisierter Variablen
    - **Signed (!)** Integer Overflow
    - Shift um Länge eines Integerwerts (*oder mehr oder negativ*)
    - Flushen eines Inputstreams, z.B. `fflush(stdin)`
    - Fehler bei Pointercasts (*meist unnötig/obsolet*)
    - *etc…*

Vermeidung von Fehlern - Sanitizer

- **Sanitizer** in GCC können über Compilerflags aktiviert werden
    - `-fsanitize=address` für Buffer Overflows und Dangling Pointer
    - `-fsanitize=leak` für Memory Leaks
    - `-fsanitize=undefined` für Undefined Behavior
- **Nachteile** der Verwendung von Sanitizer:
    - erschwert Debugging mit anderen Tools
    - Performanz des Programms wird deutlich verringert
    - Compiler erkennt bei Weitem nicht alle Fehler
    - funktioniert mit handgeschriebenem Assembly nicht

Kommandozeilenargumente in C

- `int argc` und `char** argv`
    - werden bei Programmstart mit Argumenten, die dem Programm übergegeben werden, gefüllt (*Start durch Terminal* $\rightarrow$ *Kommandozeilenargumente*)
- `argv`: Kommandozeile, an Leerzeichen aufgetrennt
- `argc`: Länge des Arrays `argv`
    - *Ausnahme*: zwischen `""` wird **nicht** getrennt
- **keine besondere Bedeutung** für `-...`
    - *dazu*: `getopt`
        - wenn `getopt` auf eine Option trifft, die nicht im `optstring` spezifiziert wurde, wird der character `'?'` zurückgegeben

        - dieser und alle anderen Rückgaben außer dem Charakter der Option führen in den *default case* und damit zum Ausdrucken einer *Fehlernachricht* und vorzeitigem Beenden mit einem Fehlercode
    - `strtol`: *Parsen von Zahlen aus Strings*
        - `**endptr` zeigt nach Ausführung auf das erste Zeichen, das nicht konvertiert werden konnte
        - wenn das gesamte Argument aus konvertierbaren Zeichen bestand, zeigt daher `**endptr` auf das letzte Zeichen des übergebenen Strings - das Nullbyte das hinter dem String abgelegt wird → `*strtol_err == '\0'` gilt, um zu prüfen, dass das Argument für Option `n` eine Ganzzahl ist

```
#include <stdio.h>

int main(int argc, char** argv){
    for(int i = 0; i < argc; i++){
        printf(";%s;\n", argv[i]);
    }
    return 0;
}
```

```
$ ./myprog.o -o filename -g
;./myprog.o;
;-o;
;filename;
;-g;

$ ./myprog.o -n "Hallo Welt"
;./myprog.o;
;-n;
;Hallo Welt;
```

System V Application Binary Interface

- **Application Binary Interface (ABI)**: Schnittstelle eines Software-Moduls auf Maschinencodeebene (*z.B. Funktion oder Betriebssystem*)
    - Datentypen (*Größe, Alignment, Layout*)
    - **Calling Convention**
        - Ort der Funktionsparameter
        - Ort der Rückgabewerte
        - welche Register gesichert bzw. verändert / überschrieben werden dürfen
    - Speicherstruktur

- Betriebssystemschnittstelle

Registertypen

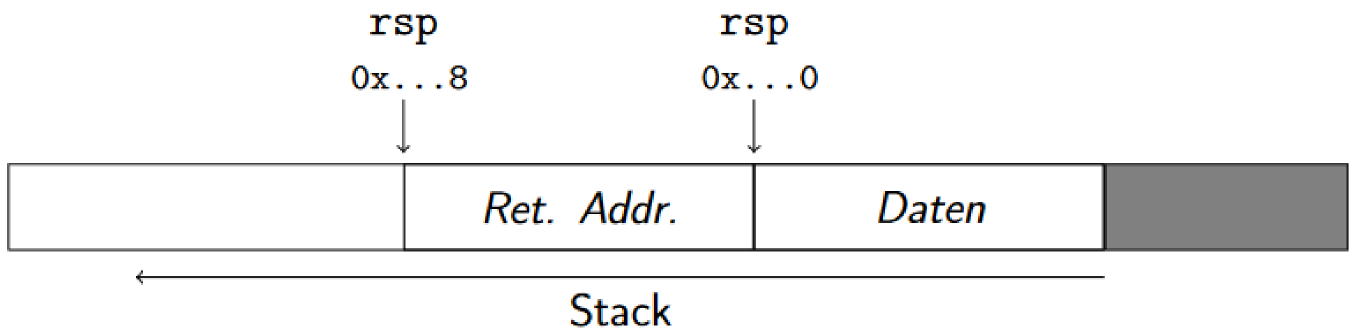
- Callee-Saved Register (*speichern z.B. Variablen über Funktionsaufrufe*)
  - gehören aufrufender Funktion
  - müssen vom callee gesichert werden
  - nach Funktionsende unverändert
  - `rbx, rbp, r12 - r15, rsp`
- Temporäre / Caller-Saved Register
  - gehören aufgerufener Funktion
  - dürfen frei verwendet werden
  - nach Funktionsende undefiniert
  - `rax, rcx, rdx, rsi, rdi, r8 - r11`

```
myfun:
  push rbx
  mov rbx, [rdi]
  ...
  pop rbx
  ret
```

Stack-Alignment

- Calling Convention stellt **Anforderung an Stack-Alignment**
  - **16-Byte-Alignment** vor einem Funktionsaufruf (*`rsp % 16 == 0`, bzw. die letzten 4 bit der Adresse sind 0*)
  - `call` legt weitere 8 Byte auf den Stack $\longrightarrow$ um 8 Byte verschoben (*`rsp % 16 == 8`*)
- bevor man eine weitere Funktion aufrufen kann, **muss man das Stack Alignment wiederherstellen** (z.B. mit `sub` oder `push`)

```
myfun:
  push rbx
  mov rbx, [rdi]
  ...
  call do_sth
  add rax, rbx
  pop rbx
  ret
```



Struct-Layout

- wie sind Felder eines `struct`s im Speicher angeordnet?
  - abhängig von **Reihenfolge und Alignment der Felder**
  - **Padding**, um Alignment der Felder sicherzustellen
  - Alignment des `struct`s ist das seines Felds mit dem größten Alignment
  - Größe des `struct`s ist ein Vielfaches davon → Padding am Ende

Structs als Funktionsparameter

- wie werden `struct`s als Funktionsparameter übergeben?
  - *größer als 16 Byte oder Felder, die nicht aligned sind?* → über dem Stack
  - *sonst*:
    - Aufteilen in max. 2 Teile mit jeweils 64 bit
    - diese werden als `int`-Parameter behandelt
    - mehrere benachbarte Felder können zusammengefasst werden

```
//2 64-Bit Integer (16 Byte)
void func(struct { uint64_t a; uint64_t b; } param);
// a -> rdi, b -> rsi

// 2 32-Bit Integer (8 Byte)
void func(struct { uint32_t a; uint32_t b; } param);
// a -> rdi[31:0], b -> rdi[63:32]

// mehrere zusammengefasste Felder (12 Byte)
void func(struct { uint16_t a; uint16_t b; uint8_t c; } param);
// a -> rdi[15:0], b -> rdi[31:16], c -> rdi[39:32]

// struct größer als 16 Byte (24 Byte)
void func(struct { uint64_t a; uint64_t b; uint64_t c; } param);
// a -> [rsp], b -> [rsp + 8], c -> [rsp + 16]
```

Structs als Rückgabewerte

- Ort **wie bei Parameterübergabe** bestimmt
- **Register**: `rax` für ersten 64 Bit, `rdx` für die zweiten 64 Bit
- **Stack**:
  - Caller reserviert ausreichend Speicher
  - Pointer darauf wird “verdeckt” in `rdi` übergeben
  - Funktion gibt Pointer in `rax` zurück

```
struct ComputeRes { uint64_t a, b, c; };
struct ComputeRes compute(int param);

// verhält sich wie:
struct ComputeRes* compute(struct ComputeRes* retval, int param);
```

Calling Convention: Zusammenfassung

- **Parameter**: `rdi, rsi, rdx, rcx, r8, r9, Stack`
- **Rückgabewert**: `rax, rdx (bei 128-bit)`
- **Callee-Saved**: `rbx, rbp, rsp, r12 - r15`
  - *gleicher Wert nach Funktionsende, ggf. sichern und wiederherstellen*
- **Caller-Saved**: `rax, rcx, rdx, rsi, rdi, r8 - r11`
  - *frei verwendbar, Wert nach Funktionsende unbekannt*

Fixkommazahlen / Festkommazahlen

- Aufteilung der Bits in **Vor- und Nachkommastellen**
  - z.B. 16.16: `0000000000000000.0000000000000000` (*Vorkommastellen, dann Nachkommastellen*)
- Gewicht der Bits ab Komma: $2^{-1}, 2^{-2}, 2^{-3}...$
  - z.B. `00000000001101.11010000000000`
    - *Vorkommastellen*: $1101_2 = 13_{10}$
    - *Nachkommastellen*: $2^{-1} + 2^{-2} + 2^{-4} = 0.5 + 0.25 + 0.0625 = 0.8125$
    - *insgesamt*: 13.8125_{10}

Fixkommazahlarithmetik

- **Addition und Subtraktion**:

    - **Komma** muss an **gleicher Stelle** sein (*Anzahl der Nachkommastellen ist gleich*), ansonsten nichts besonderes notwendig
- **Multiplikation und Division**:
    - beim Multiplizieren zweier Fixkommazahlen **addieren sich die Kommapositionen**
        - z.B. `110.100 * 1110.11 = 1011111.11100` (3.3 * 4.2 = 7.5)
    - **Shifts** sind notwendig, um das Ergebnis wieder in das richtige Format zu bringen
    - **Division** ist lediglich **Multiplikation mit Kehrwert**
- *bei Overflows muss geachtet werden*

Fließkommazahlen

- **Warum?**
    - Größe ist vom Anfang an fix
        - was, wenn man sehr große und sehr kleine Zahlen speichern möchte?
        - z.B. 2^{29} und 2^{-30} $\rightarrow$ 30 Vorkommastellen und 30 Nachkommastellen notwendig $\rightarrow$ 60-bit lange Zahl, wo man evtl. nur das oberste bzw. unterste Bit benötigt
    - teils unnötiger Speicherverbrauch

Aufbau

- `S | EEEEEEEE | MMMMMMMMMMMMMMMMMMMMMMM`
- **Exponentenschreibweise**
    - **Sign Bit** `S`: 0 wenn positiv, 1 wenn negativ
    - **Exponent** `E`: um wie viele Stellen shiften wir?
    - **Mantisse** `M`: Kommazahl
- *Vorteile*:
    - **feste Anzahl** an signifikanten Stellen
    - **größerer Wertebereich** als Fixkommazahlen
- *Konsequenz*: höhere Genauigkeit bei kleinen Zahlen
- $x = (-1)^S \cdot M_2 \cdot 2^{Exp-Bias}$

Genauerer Aufbau

- *Normalisiert*: $1 \leq$ Mantisse ≤ 2
- führende Eins **nicht abgespeichert** (*es liegt immer eine Eins vor der Mantisse*)
- Exponent (gespeichert) = Exponent (real) + Bias: *negative Exponente werden nicht im Zweierkomplement gespeichert*

- **Bias** statt Zweierkomplement (*immer positiv*)
- **Lexikographischer Vergleich** statt Subtraktion und Vergleich mit 0 (*Fließkommazahlen werden als vorzeichenbehaftete Zahlen verglichen*)
- theoretisch **weniger Operationen**

Datentypen: `float` und `double`

	Größe	Dezimalziffern	Abs. Min	Abs. Max
`float`	32	≈ 7	$\approx 1.18 \cdot 10^{-38}$	$\approx 3.4 \cdot 10^{38}$
`double`	64	≈ 15	$\approx 10^{-308}$	$\approx 10^{308}$

- `float`:
  - **Exponent**: 8
  - **Mantisse**: 23
  - **Bias**: 127
- **Bias** `double`:
  - **Exponent**: 11
  - **Mantisse**: 52
  - **Bias**: 1023

Konvertierung zu einer Fließkommazahl und umgekehrt

- *Beispiel*: 37.75
  - *Konvertierung zu einer Fixkommazahl*: `100101.11`
  - *Komma so weit verschieben, sodass nur noch eine 1 vor dem Komma steht*: `1.0010111`
  - *Nachkommastellen der Fixkommazahl = Mantisse; Sign-Bit setzen, falls Zahl negativ; Anz. Rechtsshifts + Bias = Exponent*: `0 10000100 00101110000000000000000`
- *Beispiel*: `0 10000100 01010000000000000000000`
  - Exponent (*132*) - Bias (*127*) = 5
  - Mantisse (*1.0101 in Binär*) = 1.3125 im Dezimalsystem
  - Somit ergibt sich die Zahl 42 (*um auf die Zahl 42 zu kommen, wird 1.3125 x 2^5 gerechnet*)

Addition und Subtraktion von Fließkommazahlen

- **kleinere** Wert muss auf den selben Exponenten wie der **größere** Wert gebracht werden (*denormalisieren der kleinen Zahl*)
- **Mantissen** können wie reguläre Zahlen **addiert bzw. subtrahiert** werden

- **Mantisse** wird entsprechend der Genauigkeit **gerundet** $\rightarrow$ Ergebnis normalisieren (*Mantisse soll Wert zwischen 1 und 2 repräsentieren*)

Multiplikation und Divison von Fließkommazahlen

- **Exponenten** addieren bzw. subtrahieren
- **Mantissen** multiplizieren bzw. dividieren (*führende Eins beachten*)
- **Mantisse** wird entsprechend der Genauigkeit **gerundet** $\rightarrow$ Ergebnis normalisieren

Probleme bei Genauigkeit

- **Rundung**: da Fließkommazahlen nur beschränkte Genauigkeit haben, muss entsprechend gerundet werden
  - *verschiedene Modi*: "round to nearest, ties to even" (*runde Zahl zum nächst liegenden abspeicherbaren Wert oder zum nächsten geraden Wert wenn die Zahl genau dazwischen liegt*)…
- **Absorption**: Addition bzw. Subtraktion von sehr großer und sehr kleiner Zahl
  - *keine Veränderung der großen Zahl* wegen Rundung
  - z.B. `1000000.00f + 0.01f = 1000000.00f`
- **Auslöschung**: Subtraktion großer ähnlicher Zahlen
  - Subtraktion verstärkt Rundungsfehler
  - z.B. `1000000.1f - 1000000.0f = 0.125f != .1f`, weil `1000000.1f` tatsächlich als `1000000.125` dargestellt wird
- **Regeln**: Arithmetische Operationen sind **weder assoziativ, noch distributiv**
  - `(x + y) + z != x + (y + z)`
  - `(x * y) * z != x * (y * z)`
  - `x * (y + z) != (x * y) + (x * z)`
- **(!)** `-ffast-math` (`-Ofast`) in GCC **ignoriert diese** zwecks Geschwindigkeit

Denormale / Subnormale Zahlen

- wegen der 1 vor der Mantisse können einige Zahlen nicht dargestellt werden (*Zahlen deren Exponent kleiner ist, als eine normalisierte Darstellung zulassen würde*)
  - z.B. single precision: $1.0_2 * 2^{-127}$ (*Exp. kann bei single precision keinen Wert kleiner als -126 annehmen*)
  - *denormalisiert*: $0.1_2 * 2^{-126}$ (*Exp. hat speziellen Wert, damit erkannt wird, dass es sich um eine denormalisierte Zahl handelt: alle Bits 0*)
  - *implizite 1 wird zur impliziten 0*
- wenn Exponent und Mantisse 0 sind, handelt es sich um 0

Null mit Vorzeichen

- **Null**: Exponent und Mantisse haben alle Bits 0
- **Sign-Bit kann gesetzt werden**: ± 0 möglich
  - allgemein $x + 0 = x$, **aber** $-0 + 0 = 0$ für $x = -0$

Unendlich / Infinity

- wenn alle Bits in **Exponent 1** und alle Bits in **Mantisse 0**: *Unendlich*
  - *je nach Sign-Bit*: $\pm\infty$
- Ergebnis bei der **Division einer Zahl durch 0** (*Ausnahme:* $0/0 = NaN$ *und* $\infty - \infty = NaN$)

Not A Number / NaN

- wenn alle Bits in **Exponent 1** und **Mantisse** $\neq$ **0**: *NaN*
  - Sign-Bit irrelevant, $-NaN$ existiert nicht
- bei **undefinierten Operationen**, z.B. Teilen von 0 durch 0 oder Subtraktion von Infinity und Infinity
- jede Operation mit NaN ergibt NaN
- jeder Vergleich mit NaN ist `false`, außer $\neq$

Weitere Floating Point Formate

- 16-Bit half precision / `half`
  - `S | EEEEE | MMMMMMMMMM` (1 / 5 / 10)
- Brain Floating Point / `bfloat`
  - `S | EEEEEEEE | MMMMMMM` (1 / 8 / 7)
- Extended Formate

Fließ- und Fixkommazahlen: Zusammenfassung

- **Fließkommazahlen**:
  - **Großer Wertebereich** bei **weniger Speicherplatzverbrauch**
  - **Ungenauigkeiten** durch **Rundung** (*Absorption, Auslöschung*)
  - **Arithmetik** nicht trivial
- **Fixkommazahlen**:
  - **Schnelle Arithmetik**
  - **Gleichbleibende Genauigkeit** im gesamten Wertebereich
  - **Kommaposition fest** → **Hoher Speicherplatzverbrauch** bei **großen Wertebereichen**

Streaming SIMD Extensions (SSE)

- SIMD Erweiterung für x86-Prozessore

SSE Register

- 16 weitere Register: `xmm0` bis `xmm15`
  - **128-bit** groß, können z.B. 4 32-bit Werte speichern
  - für **skalare Berechnungen** sind nur die unteren 32 bzw. 64 Bit von Relevanz
- `xmm`-Registern verwendbar für **Floating-Point-Berechnungen**

Konstanten

- 0 kann mit `pxor dst, src` genutzt werden (*wobei* `dst == src`)
- **Floating-Point Konstanten** können aus dem Speicher (z.B. `.rodata`) mit `movss dst, src` (*move single precision float*) geladen werden
  - z.B. `movss xmm0, [rip + .Lconstx]`
- **Moves** sind zwischen General-Purpose und xmm-Registern mit `movd / movq` möglich
  - keine Konvertierung! $\longrightarrow$ Werte werden Bit für Bit zw. Registern ausgetauscht
  - gut für Bitmanipulationen

Arithmetik

- **Namenskonvention bei Instruktionen**: `ss` für Scalar Single, `sd` für Scalar Double (*ss nicht mit sd anwendbar*)
- **Addition**: `addss dst, src`
  - `src`: Register und Speicher erlaubt
  - untere 32 bit für Addition mit `addss` verwendet
- **Subtraktion**: `subss dst, src`
  - *analog zur Addition*
- **Division**: `divss dst, src`
  - Operanden werden im Vergleich zu `div` nicht implizit gefolgert, sondern werden explizit angegeben
- **Multiplikation**: `mulss dst, src`
  - *analog zur Division*
- *Konstanter Divisor*: Multiplikation mit **Kehrwert** bevorzugen

Vergleiche

- `ucomiss op1, op2`: skalarer Vergleich zweier Floating-Point Werte
    - untersten single precision Floats werden verglichen
- **Flags** werden in **Abhängigkeit des Ergebnisses** gesetzt
    - ermöglicht Sprünge mit `jcc`
    - **aber**: Condition Codes für **vorzeichenlose** Vergleiche

```
cmpFloat :
    ucomiss xmm1 , xmm0
    jp _Lunordered ; xmm0 or xmm1 NaN
    jb _Llesser ; xmm1 < xmm0
    ja _Lgreater ; xmm1 > xmm0
    je _Lequal ; xmm1 == xmm0
```

- ist einer der Operanden NaN, wird dies mit `jp` oder `jnp` behandelt

Codeeispiel ($func \rightarrow 1/x$)

```
#include <stdio.h>

extern float func(float x);

int main(int argc, char **argv){
    float res = func(2.0);
    printf("Result: %f\n", res);
    return 0;
}
```

```
.intel_syntax noprefix
.global func
.text
func:
    mov r8, 1
    cvtsi2ss xmm1, r8
    divss xmm1, xmm0
    movss xmm0, xmm1
    ret
```

Erweiterte Calling Convention

- vermeiden, dass in General Purpose Register die Floating Point Argumente übergeben werden
- **Float-Rückgabewert**: `xmm0`
- **Float-Argumente**: `xmm0 -> xmm7` (*weitere auf Stack*)
- **(!) alle Register sind caller-saved / temporär**

- *bei Kombinationen von FP und int/ptr-Argumente*: **separate Durchnummerierung** der Register

```
   float fn(int, float, char*, float);
// xmm0      edi   xmm0    rsi     xmm1
```

```
ex:
    mov rdi, 1
    movss xmm0, [rip + .Lconstx]
    mov rs1, 2
    movss xmm1, [rip + .Lconsty]
    call fn
    ...
```

SIMD - SSE

- **Konzept**: parallele Verarbeitung von Daten ohne Threading
- **bisher - SISD (Single Instruction Single Data)**
  - Instruktion arbeitet auf *Daten bestimmter Länge* (*byte* $\longrightarrow$ *qword*)
  - *Verarbeiten eines Arrays*: Iterieren über Speicherbereich
  - z.B. Addition zweier Vektoren (*Arrays mit Elementen eines bestimmten Typs*)

e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12
0	1	2	3	4	5	6	7	8	9	10	11	12
+	+	+	+	+	+	+	+	+	+	+	+	+
0	1	2	3	4	5	6	7	8	9	10	11	12
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	2	4	6	8	10	12	14	16	18	20	22	24

SIMD - Single Instruction Multiple Data(stream)

- **Nachteil SISD**: hohe Sprunganzahl, viele Rechenoperationen
- Lösung: **Vektorisierung**
  - Anwendung der selben Instruktion auf einen ganzen Satz aus Datenobjekten
  - eingeführt durch SSE-Befehle für Vektorverarbeitung
  - dafür werden 128-bit XMM-Register verwendet

SSE-Instruktionen für SIMD

- **neue Instruktionen** für **parallele Datenverarbeitung**, die auf ganze XMM-Register **zur gleichen Zeit** arbeiten
- **keine Speicher-zu-Speicher** Operationen
- **unterschiedliche Stufen** der SSE-Erweiterung (*auf modernen CPUs immer SSE2 verfügbar*)

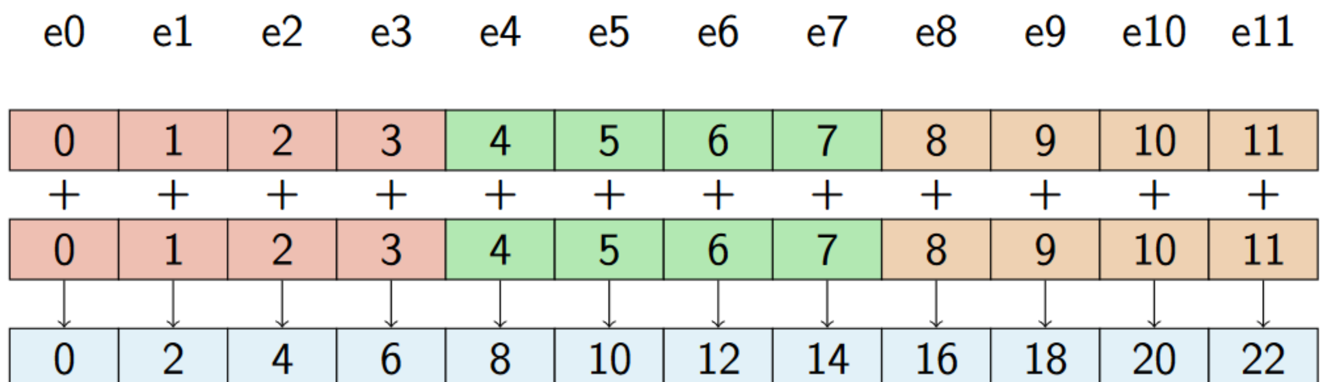
- **gesonderte Befehle** für Integer- und Floating-Point-Berechnung
  - "**Packed (P)**" - keine einheitliche Position für Int und Float (`ADDPD` / `PADDD`)
  - **kein Effekt auf nebenstehende Daten** im XMM-Register (*keine Überlaufe von einer Addition im Befehl zur nächsten, etc.*)
  - **Ziel** i.d.R. XMM-Register
- ideal ist es wenn die Daten, welche mittels SIMD verarbeitet werden sollen, **nebeneinander im Speicher liegen**
  - es ist natürlich durchaus möglich ein XMM-Register aus **unterschiedlichen Quellen** zu füllen
  - allerdings ist es umso besser, **je weniger teure Speicher und Schiebeoperationen** dafür nötig sind.

Integer-Instruktionen

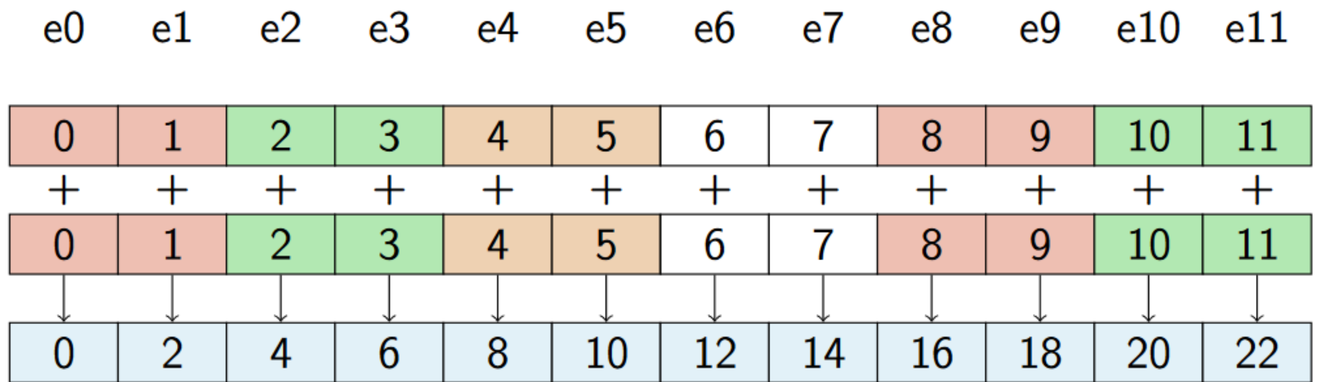
- `PADDD xmm1, xmm2/m128` (*ADD Packed Integer DWORD*)
  - addiert 4 DWORD-Integer auf 4 andere DWORD-Integer (*32-bit*)
- `PADDB xmm1, xmm2/m128` (*ADD Packed Integer BYTE*)
  - addiert 16 BYTE-Integer auf 16 andere BYTE-Integer (*8-bit*)
- `PADDQ xmm1, xmm2/m128` (*ADD Packed Integer QWORD*)
  - addiert 2 QWORD-Integer auf 2 andere QWORD-Integer (*64-bit*)

Vektoraddition: 32- und 64-bit

- `PADDD` (*jeder Kasten ist 32-bit groß $\longrightarrow$ es können 4 Werte zur selben Zeit addiert werden*)



- `PADDQ` *(jeder Kasten ist 64-bit groß $\rightarrow$ es können 2 Werte zur selben Zeit addiert werden)*



Inkrementieren eines Elements mittels SIMD

```
; void add_one(int x[4])
add_one:
    mov eax, 1
    movd xmm0, eax
    pshufd xmm0, xmm0, 0x00
    movdqu xmm1, [rdi]
    paddd xmm1, xmm0
    movdqu [rdi], xmm1
    ret
```

- `MOVD` kopiert DWORD aus EAX in XMM0
- `PSHUFD` kopiert den niedrigsten DWORD in XMM0 mittels der Maske 0x00 in die drei höherwertigen DWORD in XMM0
  - $[0, 0, 0, 1] \rightarrow [1, 1, 1, 1]$
- `MOVDQU` lädt 4 Integer aus dem Speicher ab RDI in XMM1
- `PADDD` addiert 4 Integer aus XMM0 auf XMM1
- `MOVDQU` schreibt das Resultat in den Speicher (4 Integer aus XMM1 in Speicher ab RDI)

Gleitkomma-Instruktionen

- `ADDSS xmm1, xmm2/m32` (*ADD **Scalar** Single-Precision Floating-Point Values*)
  - addiert **einen float** auf **einen anderen float**
  - **keine SIMD-Instruktion (packed)**, sondern eine **scalar**-Instruktion
- `ADDPS xmm1, xmm2/m128` (*ADD **Packed** Single-Precision Floating-Point Values*)
  - addiert **4 float** auf **4 andere float**
- `ADDPD xmm1, xmm2/m128` (*ADD **Packed** Double-Precision Floating-Point Values*)
  - addiert **2 double** auf **2 andere double**

Addition eines Vektors auf sich selbst mit SIMD

```
; void add_self(float *, size_t)
; assume that rdi & 3 == 0
add_self:
    movups xmm0, [rdi]
    addps xmm0, xmm0
    movups [rdi], xmm0
    add rdi, 16
    sub rsi, 4
    jnz add_self
    ret
```

- `MOVUPS` lädt 4 Floats ab Speicheradresse RDI in das Register XMM0
- `ADDPS` addert 4 Floats in XMM0 paarweise auf sich selbst
- `MOVUPS` schreibt 4 Floats vom Register XMM0 an Speicheradresse RDI
- erhöhe Zeiger in RDI um 16 Byte und reduziere Zähler in RSI um 4

SIMD-Alignment

- spezielle **neue `MOV`-Instruktionen** für XMM-Register
- **(!) zu berücksichtigen**: Data Alignment
- *bisher*: genaue Position der Daten im Speicher (Startadresse) nicht relevant
- *bei SIMD-SSE*: aufgrund der Performanz müssen **neue Alignment-Anforderungen** erfüllt werden
    - die meisten SIMD-Instruktionen fordern ein **16-Byte-Alignment**
        - Startadresse muss 16-Byte-aligned sein (*muss teilbar durch 16 sein, niederwertigsten 4 Ziffern in Binärdarstellung sind 0*)
        - CPU wirft bei non-aligned Speicherzugriff eine **Exception** (*z.B. als SEGFAULT*)
        - Alignment von Variable, Array und Structure-Fields in C mittels `__attribute__((aligned(16)))`
            - z.B. `int x[12] __attribute__((aligned (16)));`
        - **skalare Instruktionen** fordern dies **nicht**

Aligned Zugriff

- `MOVAPS xmm/m128 xmm/m128` (*Move Aligned Packed Single-Precision*)
    - kopiert **4 floats** vom Ursprung in das Ziel
    - **Speicheroperanden** müssen **aligned** sein, sonst Fehler
    - kann auch **4 int, 2 double oder 2 long** kopieren, obwohl es dafür gesonderte Instruktionen wie `MOVDQA` gibt

- **(!) Speicheroperanden fordern bei allen SIMD-Instruktionen das Alignment**
- `MOVUPS xmm/m128 xmm/m128` (*Move Unaligned Packed Single-Precision*)
  - kopiert **4 floats** vom Ursprung in das Ziel
  - langsamer als `MOVAPS`, auch auf aligned Speicher
    - das Zusammensetzen von Daten aus zwei Cache-Lines erfordert zusätzliche Arbeit durch die CPU, welche sich in der Ausführungszeit für den Befehl niederschlägt
    - auf modernen CPUs genauso schnell wie `MOVAPS`

SIMD-Stolperfallen

- SIMD kann Rechenzeit sparen und Verarbeitung beschleunigen
  - auf günstigen CPUs kann es evtl. langsamer sein
  - höhere Leistungsaufnahme / Temperatur mit möglicher Taktreduzierung wegen paralleler Operationen
- SIMD eignet sich nicht für jedes Problem
  - SIMD findet außerhalb der Standard-Rechenwerke statt $\rightarrow$ Mischen von Standardoperationen und SSE-Operationen kann Program verlangsamen
  - SIMD macht einen sub-optimalen Algorithmus nicht automatisch optimal
  - variierender Control-Flow kann SIMD-Vorteile minimieren
- Verwendung von Erweiterungen wie SSE, AVX und FMA kann Kompatibilität einschränken

Compiler und Vektorisierung

- Compiler versuchen Programmcode automatisch zu vektorisieren, machen es aber nicht optimal
- man sollte sich nicht auf den Compiler darauf verlassen
  - **Möglichkeit 1**: SIMD direkt in ASM zu programmieren
  - **Möglichkeit 2**: SIMD Intrinsics (*abstrahieren SIMD-Befehle für den Programmierer*)

SIMD-Intrinsics

- neuer Datentyp für Variablen: `__m128` für float-Datentypen
  - 128-bit, kann ein ganzes xmm-Register füllen, es wird aber kein konkretes Register geschrieben, sondern es wird nur eine solche Variable definiert
- für jede SSE-Instruktion gibt es eine zugehörige Intrinsic, die typischerweise auf Variablen des Typs `__m128` definiert sind
  - **ASM**: `addps xmm, xmm` $\rightarrow$ **C**: `__m128 _mm_add_ps (__m128 a, __m128 b)`
    - Ergebnis wird von der Funktion zurückgegeben und nicht direkt in Register geschrieben
  - **ASM**: `mulss xmm, xmm` $\rightarrow$ **C**: `__m128 _mm_mul_ss (__m128 a, __m128 b)`

- **ASM**: `mov*s`
  - **Load (ASM)**: → **C**: `__m128 _mm_load_ps (float const* mem_addr)`
    - *z.B. wenn b vom Typ float*: `__m128 x = _mm_load_ss(&b)`
  - **Store (ASM)**: → **C**: `void _mm_store_ps (float* mem_addr, __m128 a)`
- **weitere Befehle**: [Intel Intrinsics Guide]

Codebeispiel - Saxpy

```
#include <immintrin.h>

void saxpy(long n, float alpha, float *x, float *y){
    __m128 valpha = _mm_load1_ps(&alpha);

    // simd
    for(size_t i = 0; i < (n - (n % 4)); i += 4) {
        __m128 vx = _mm_loadu_ps(x + i);
        __m128 vy = _mm_loadu_ps(y + i);

        vy = _mm_add_ps(_mm_mul_ps(valpha, vx), vy);

        _mm_storeu_ps(y + i, vy);
    }

    // remaining elements
    for(size_t i = (n - (n % 4)); i < n; i++) {
        y[i] = alpha * x[i] + y[i];
    }
}
```

Andere Datentypen

- `__m128d` (Double) und `__m128i` (Integer)
  - `__m256`: AVX
  - `__m64`: Legacy
- Suffixe für Funktionen ändern sich auch
  - **ASM**: `addpd xmm, xmm` → **C**: `__m128d _mm_add_pd (__m128d a, __m128d b)`
  - **ASM**: `paddb xmm, xmm` → **C**: `__m128i _mm_add_epi8 (__m128i a, __m128i b)`

Vor- und Nachteile von Intrinsics

- **+**: Abstraktion von Assembly

    - Verbesserung der Lesbarkeit und Flexibilität des Codes
    - Freiraum für Compileroptimierungen
- **-**: Verlust der Plattformunabhängigkeit

Automatische Vektorisierung

- GCC automatisch ab `-O3`
- `-march` spezifiziert, welcher Erweiterungssatz verwendet werden soll
    - `-march=native`: alle Erweiterungssätze unterstützen die lokale Architektur
- `-fopt-info-vec(-missed)`: prüfe, ob / welche Vektorisierungen erfolgreich waren (und warum auch nicht)
- Resultat **nicht immer so effizient** wie eigene Implementierung $\rightarrow$ `objdump` zu analysieren

Optimierungen

Optimierungsstufe	Beschreibung
`-O0`	keine Optimierungen
`-O1`	Optimierungen mit wenig Compile-Zeit
`-Og`	O1 mit Fokus auf debugbaren Code
`-O2`	alle Optimierungen ohne Space-Speed-Tradeoff
`-Os`	O2 mit Fokus auf minimaler Code-Größe
`-O3`	alle Optimierungen
`-Ofast`	O3 mit Float-Optimierungen (disregard strict standard compliance)

- bereits ab `-O1` kann debugging schwerer werden, da Statements verschoben oder Variablen wegoptimiert wurden
- ab `-O2` kann das Kompilieren länger dauern
- ab `-O3` kann die größe der komplierten Executable wachsen
- Compiler-Optimierungen sind nur dann valide, wenn das **Verhalten des Programms unverändert** bleibt

Optimierung von Berechnungen

- **Constant Folding**: Konstante werden zur Compile-Zeit berechnet
- **Constant Propagation**: Variablen werden mit ihren tatsächlichen Werten ersetzt
    - *bei Funktionen*: Ergebnis des Funktionsaufrufs wird bereits berechnet
- **Common Subexpression Elimination (CSE)**: wenn dieselbe Berechnung mehrmals durchgeführt wird, kann sie einmal berechnet und gespeichert werden

```
int x = a * b * 24;
int y = a * b * c;

// optimized
int tmp = a * b;
int x = tmp * 24;
int y = tmp * c;
```

Optimierung von Schleifen

- mittels **Loop Unrolling**
    - Anzahl an Schleifendurchläufe wird reduziert / Schleife wird gänzlich entfernt
    - **+**: erhöhte Geschwindigkeit (*Loop Conditions werden weniger bzw. gar nicht mehr getestet*)
    - **-**: Executable-Größe wächst (*mehr Instruktionen nötig, was auch den Code wegen mehr Speicherzugriffe verlangsamen kann*)

```
for(int i = 0; i < 6; i++){
    arr[i] = 2 * arr[i];
}

// optimized: -floop-unroll-and-jam, ab -O3
for(int i = 0; i < 6; i+= 2){
    arr[i] = 2 * arr[i];
    arr[i + 1] = 2 * arr[i + 1];
}
```

- mittels **Jamming / Loop Fusion**
    - **+:** vermeidung von doppeltem Schleifen-Overhead
    - **+**: evtl. mehr Optimierungen in Schleifenkörper möglich (*z.B. Zusammenfassung von Rechnungen*)

```
for(int i = 0; i < 6; i++){
    arr[i] = 2 * arr[i];
}

for(int i = 0; i < 6; i++){
    arr2[i] = arr2[i] + 24;
}

// optimized: -floop-unroll-and-jam
for(int i = 0; i < 6; i++){
    arr[i] = 2 * arr[i];
```

```
    arr2[i] = arr2[i] + 24;
}
```

- mittels **Loop-Invariant Code Motion**
  - falls `x == 0`, kann n jeden Wert haben; daher ist die optimierte Variante korrekt
    - Beispiel für Ausnutzung von *Undefined Behavior* für Optimierungen
  - **+**: Vermeidung redundanter Berechnungen

```
int n;
for(int i = 0; i < x; i++){
    n = sizeof(arr) / sizeof(int);
    arr[i] = 2 * arr[i];
}

// optimized: -fmove-loop-invariants, ab -O1
int n = sizeof(arr) / sizeof(int);
for(int i = 0; i < x; i++){
    arr[i] = 2 * arr[i];
}
```

- mittels **Vertauschung**
  - **+**: Verbessert Cache-Verhalten (*weniger Cache-Misses*)
  - **+**: ermöglicht evtl. Vektorisierung

```
int sum = 0;
for(int i = 0; i < 6; i++){
    for(int j = 0; j < 9; j++){
        sumt += arr[j][i];
    }
}

// optimized: -floop-interchange, ab -O3
int sum = 0;
for(int j = 0; i < 9; j++){
    for(int i = 0; j < 6; i++){
        sumt += arr[j][i];
    }
}
```

Optimierung von Funktionsaufrufen

- mittels **Inlining**

    - **+**: kein Overhead durch Funktionaufruf (`call` *speichert Rücksprungadresse, Code liegt an anderem Ort in Speicher*)
    - **-**: kann Code massiv vergrößern und verlangsamen
    - der Specifier `inline` hat nur bedingt etwas zu tun

```c
static int square(int x){
    return x*x;
}

for(int i = 0; i < n; i++){
    arr[i] = square(i);
}

// optimized: -finline-functions-called-once, ab -O1 bzw. -finline-functions, ab -O2
for(int i = 0; i < n; i++){
    arr[i] = i * i;
}
```

- mittels **Tail Call Optimization**
    - dann, wenn **NUR** Funktionsaufruf letzte Operation vor Rücksprung ist
    - nicht notwendig, Daten auf dem Stack abzulegen
    - ersetzt Funktionsaufruf (`call + ret`) durch `jmp`

```c
int fac(int k, unsigned n){
    if(n <= 0) return k;
    return fac(k * n, n - 1);
}

// optimized: -foptimize-sibling-calls, ab -O2
int fac(int k, unsigned n){
    fac:
        if(n <= 0) return k;
        k *=n;
        n--;
        goto fac;
}
```

Interprozedurale Optimierungen

- Entfernen unnötiger Funktionsparameter
    - nur bei `static`-Funktionen möglich

- Funktionsspezifische Calling Convention
  - z.B. mehr callee-saved Register, andere Argumentregister
  - nur bei `static`-Funktionen ohne externe Nutzung möglich
- Spezialisierung von Funktionen bei mehreren verschiedenen Aufrufen
  - Duplikation der Funktion und Optimierung für verschiedene Parameterwerte (*z.B. Konstanten, die schon zur Compile-Zeit bekannt sind*)

Low-Level Optimierungen

- **Instruction Selection**: Übersetzung von Statement zu Instruktionen
  - z.B. Ersetzen der Multiplikation mit `lea`
- **Instruction Scheduling**: Reihenfolge der Instruktionen
  - Verringern von Abhängigkeiten zwischen Instruktionen $\longrightarrow$ Begünstigung von Pipelining und OoO
  - bessere Ausnutzung von Instruction-Level Parallelism im Prozessor
- **Register Allocation**: Speichern der Variablen in Register / Stack
  - *Ziel*: Verringern / Vermeiden von Stack-Zugriffen

Optimierte Funktionen

- `libc` stellt häufig benutzte Funktionen hochoptimiert bereit, wobei die beste Funktion zur Laufzeit mittels `IFUNC_SELECTOR`s ausgewählt wird

Builtins

- für bestimmte Anwendungszwecke bietet GCC `builtin`s an (*nicht Teil der Standardbibliothek*)
  - `__builtin_clz(unsigned x)`: gibt Anzahl der führenden 0 eines unsigned ints zurück
    - auf x86 mittels `bsr` implementierbar
    - notwendig, da Compiler selbst auf höchster Optimierungsstufe nicht erkennt, dass es sich um diesen Befehl handelt
  - `__builtin_expect(long exp, long c)`
    - `exp` wird wahrscheinlich zu `c` auswerten
    - Compiler versucht mit **Branch Prediction** performanten Code zu generieren
    - sollte aber nur nach Profiling des Codes benutzt werden

Funktionsattribute (Beispiele)

- **Hinweise** für den Compiler

```c
__attribute__((always_inline))
void addTwo(uint8_t* element){
    *element += 2;
}

__attribute__((noinline))
void addTwo(uint8_t* element){
    *element += 2;
}
```

- `const`: Ausgabe **nur** durch Eingabe bestimmt
    - Ergebnis ist unabhängig vom Zustand des Programms
        - nicht-read-only Speicher darf den Rückgabewert nicht beeinflussen
    - `const`-Funktion darf nur andere `const`-Funktionen aufrufen
    - Funktion verändert Programmzustand nicht (`void`-Rückgabewert sinnlos)
    - nur notwendig bei Funktionen, deren Definition nicht verfügbar ist
    - **Ziel**: Compiler kann Ergebnisse einfach wiederverwenden

```c
__attribute__((const))
extern uint32_t mulPi(uint32_t n); // n * pi
```

- `pure`: ähnlich zu, aber wenig restriktiver als `const`
    - Rückgabewert darf von Dereferenzierung der übergebenen Pointer abhängen
    - `pure`-Funktionen dürfen `pure`- und `const`-Funktionen aufrufen

```c
__attribute__ (( pure ) )
int my_memcmp ( const void * ptr1 , const void * ptr2 , size_t n ) {
    while (! n - -)
        if (* ptr1 ++ != * ptr2 ++)
            return * ptr2 - * ptr1 ;
    return 0;
}
```

- `hot`: besonders oft aufgerufene Funktionen
    - höhere Optimierung auf Geschwindigkeit
    - größerer Code
    - eigener Speicherbereich für bessere Cachelokalität
- `cold`: besonders selten aufgerufene Funktionen
    - kleinerer Code
    - langsamer

- eigener Speicherbereich für besseres Cacheverhalten des restlichen Programms

Layout von Datenstrukturen

- Größe der verwendeten Datentypen soll **so groß wie nötig, so klein wie möglich** sein
  - für `{0,1,...,12800}` wäre `unsigned short` besser als `unsigned int`
  - für `{0.00,0.25,...,100.00}` wäre `float` besser als `double`
- genaue Größe von `int, short` etc. ist implementation-defined, so dass die Verwendung von fixed-width Integern sinnvoll ist (*z.B.* `uint8_t`, `int16_t` *etc.*)

Layout von `struct`s

```
struct PenguinBad {       // alignment: 8 (char*)
    char type;            // offset: 0
    char *name;           // offset: 8
    uint8_t age;          // offset: 16
}                         // size (mult of alignment): 24

struct PenguinBad {       // alignment: 8 (char*)
    char type;            // offset: 0
    uint8_t age;          // offset: 1
    char *name;           // offset: 8
}                         // size (mult of alignment): 16
```

- manuelles Umordnen sinnvoll, da es nicht automatisch vom Compiler gemacht werden kann
- häufiges Kopieren / Umwandeln von Daten soll möglichst vermieden werden

Pointer-Aliasing

- Pointer zeigen grundsätzlich nur auf Speicherobjekte **eines** bestimmten Typs
  - Pointer-Casts sind möglich, aber eine Dereferenzierung dieses casted Pointers allgemein undefined behavior (Ausnahme: `char* / unsigned char*`)
- ein Pointer, der auf denselben Speicherbereich wie ein anderer Pointer zeigt, heißt **Alias**
  - nicht jeder Pointer kann Alias für jeden anderen sein (`unsigned int*` kann Alias von `int*` sein, aber `unsigned int*` kann nicht Alias von `double*` sein)
  - nur gültige Aliase sollten auf gleichen Speicherbereich zeigen (Pointer, die nicht Aliase voneinander sind)
- mittels `restrict`
  - spezif., dass die übergegebenen Pointer auf unterschiedliche Speicherbereiche zeigen (*bzw. man greift auf Speicherobjekte über genau einem der beiden Pointer zu*)

```c
void foo(unsigned* a, int* b){
    ...
}

void foo2(unsigned* restrict a, int* restrict b){
    ...
}
```

Beispiele für Pointer-Aliasing

```c
// arr und sum zeigen nicht auf gleichen speicher
// compiler kann das aber nicht wissen (char* kann alles aliasen)
void count_a(const char *arr, int *sum){
    while(*arr){
        *sum += *arr++ == 'a';
    }
}

// fixed
void count_a(const char* restrict arr, int *sum){
    while(*arr){
        *sum += *arr++ == 'a';
    }
}
```

```c
// arr und sum können keine gültigen aliase sein
// diese zeigen also nicht auf gleiche speicherbereiche
// optimierungen erst ab -o2 oder mit -fstrict-aliasing
void count_a_short(const short arr[4], int *sum){
    for(size_t i = 0; i < 4; i++){
        *sum += arr[i] == 'a';
    }
}
// optimierung:
// sum muss nicht bei jeder iteration in den speicher geschrieben werden
```

Vergleiche mit SIMD

```asm
; if(a[i] != b[i]) a[i] = 0;
mov ecx, dword ptr [rsi]
cmp dword ptr [rdi], ecx
je .Lskip
mov byte ptr [rdi] 0
```

```
.Lskip:
...
```

- **Probleme** bei Vergleiche mit SIMD:
  - kein Flag-Register für jedes Vektorelement
  - Instruktion kann nur für alle Elemente übersprungen werden
- **Vergleiche** mit SIMD funktionieren anders
  - es werden **keine Flags** gesetzt, die dann mit `Jcc` überprüft werden
  - stattdessen muss man bereits beim Vergleich entscheiden, was man prüfen will (*verschiedene Instruktionen für* `a == b`, `a > b` *etc.*)
  - **Ergebnis** wird nicht in Flag-Register, sondern **XMM-Register** gespeichert
  - Ergebnis hat Form einer **Bitmaske**
    - falls Bedingung für ein Element **erfüllt** ist, wird jedes Bit des Elements auf **1** gesetzt
    - falls Bedingung für ein Element **nicht erfüllt** ist, wird jedes Bit des Elements auf **0** gesetzt

Vergleichsbefehle

- `PCMPEQB xmm1, xmm2/m128` (*Compare Packed Data for Equal*)
  - vergleicht Bytes von xmm1 mit xmm2 / m128 auf Gleichheit
  - Ergebnis als Bitmaske in xmm1 (`0xfff...` wenn wahr, `0x000...` wenn falsch)
  - `PCMPEQW xmm1, xmm2/m128`: 8 x 16-bit
  - `PCMPEQD xmm1, xmm2/m128`: 4 x 32-bit
  - `PCMPEQQ xmm1, xmm2/m128`: 2 x 64-bit
- `PCMPGTB xmm1, xmm2/m128` (*Compare Packed Signed Integers for Greater Than*)
  - vergleicht, ob ein vorzeichenbehaftetes Element des ersten Vektors größer als das des zweiten ist
  - `PCMPGTW xmm1, xmm2/m128`: 8 x 16-bit
  - `PCMPGTD xmm1, xmm2/m128`: 4 x 32-bit
  - `PCMPGTQ xmm1, xmm2/m128`: 2 x 64-bit

Anwendung der Bitmaske

- **generierte Bitmaske** kann auf die Inputs der nächsten Instruktion anwenden, so dass diese nur dann ein Effekt hat, wenn die Bedingung erfüllt ist

```
; if (a[i] != b[i])
; a[i] = 0;
; xmm0 = a [i]
movdqa xmm0, xmmword ptr [rdi]
```

```
; xmm1 = b[i]
movdqa xmm1, xmmword ptr [rsi]
pcmpeqd xmm1, xmm0                  ; create bit mask
pand xmm0, xmm1                     ; set a[i] to 0 with logical AND
movdqa xmmword ptr [rdi] , xmm0 ; replace a[i] in memory
```

SIMD mit General Purpose Registern

- *bis jetzt*: SIMD mit **SSE und AVX**
- SIMD geht auch mit **General Purpose Registern**
  - hier muss man manuell beachten, dass ein Element keinen Einfluss auf ein anderes hat
- *Beispiel*: Elemente eines `uint8_t`-Arrays durch 2 teilen
  - 8 Elemente in `rax` laden, Division durch 2 mit Bitshift (`shr rax, 1`)
  - höchste Bits auf 0 setzen mit Bitmaske (`01111111 | 01111111 | ...`) in `r8`, da das niedrigste Bit von einem Element ins nächste geschoben wird (`and rax, r8`)

```
mov rax, [rdi]
shr rax, 1
mov r8, 0x7F7F7F7F7F7F7F7F
and rax, r8
```

Wann ist SIMD sinnvoll?

- große Datenmengen (*Bildbearbeitung, Matrixoperationen*)
- gleiche oder ähnliche Instruktionen auf Elementen (*nicht zu viele konditionale Operationen auf den Elementen ausführen!*)
- Daten müssen geschickt im Speicher liegen (*z.B. Linked Lists schlechter vektorisierbar als Arrays*)
- keine anderen Funktionen dürfen in der Schleife aufgeruft werden, außer dann, wenn die Funktion inline-bar ist und selbst vektorisierbar ist (*z.B.* `a[i] = foo(i)`)
- auf *loop carried dependencies* muss geachtet werden (*z.B. `a[i] += a[i-1

SIMD mit AVX

- **AVX-Register**: 256-bit
  - Platz für **8 x 32-bit single precision** oder **4 x 64-bit double precision** floats
- **Erweiterung der 16 xmm-Register**: `ymm0` bis `ymm15`
  - unteren 128 bit beinhalten die `xmm`-Register für Kompatibilität

Erweiterte SSE-Instruktionen in AVX

- **Drei-Operanden-Format** (`OP dest, src1, src2`)
  - `a = b + c` ⟶ mehr Flexibilität
- **Präfix** `V`
- `VADDPS xmm/ymm, xmm/ymm, xmm/m128/ymm/m256`: Vektoraddition von Gleitkommazahlen
  - bei 128-bit Register werden die oberen 128-bit des Zielregisters auf 0 gesetzt
  - bei `ADDPS` würde der Wert beibehalten
- `VMOVSD xmm, xmm, xmm`: merged 2 x 64-bit Gleitkommazahlen in ein xmm-Zielregister

Neue AVX-Instruktionen

- `VBROADCASTSS xmm/ymm, xmm/m32`
  - **kopiert** die Gleitkommazahl an den unteren 32-bit des Quelloperanden **in alle 32-bit Blöcke des Zielregisters**
  - ähnlich zu `PSHUFD` in SSE
- `VPSLLVD xmm/ymm, xmm/ymm, xmm/m128/ymm/m256`
  - Shift Logical Left von 32-bit Blöcken, vorne mit 0 aufgefüllt
- `VPSRLVD xmm/ymm, xmm/ymm, xmm/m128/ymm/m256`
  - Shift Logical Right von 32-bit Blöcken, vorne mit 0 aufgefüllt
- `VPSRAVD xmm/ymm, xmm/ymm, xmm/m128/ymm/m256`
  - Shift Arithmetic Right von 32-bit Blöcken, vorne mit Vorzeichenbit aufgefüllt

Alignment in AVX

- **nicht mehr verpflichtend**
  - gilt auch für die meisten SSE-Instruktionen, außer bei denen, wo das Alignment explizit gefordert wird (*z.B.* `MOVAPD`)
- Alignment trotzdem wegen **höherer Performanz** empfohlen
  - *best practice*: 16-Byte Alignment für 128-bit Daten, 32-Byte Alignment für 256-bit Daten

SSE und AVX - Adressierungsschemata

- was passiert, wenn man im Code häufig zwischen SSE und AVX wechselt?
  - sind die oberen 128-bit eines ymm-Registers nicht null, wird jede SSE-Instruktion zu einem Merge, wobei der obere Teil beibehalten muss, was teuer ist
  - *Vermeidung*: `VZEROALL` (*nullt alle ymm-Register und markiert diese als ungenutzt*) und `VZEROUPPER` (*nullt die oberen 128-bit aller ymm-Register*) sollten vor jedem SSE/AVX-Wechsel

ausgeführt werden

- unterschiedliche Prozessorfrequenzen für verschiedene Instruktionsklassen
    - *Non-AVX*: reguläre und SSE-Instruktionen und einfache Integer Vektor-Operationen (*CPU: normale Basis- und Turbofrequenz*)
    - *AVX2-heavy*: AVX-Instruktionen (*CPU: AVX2 Basis- und Turbofrequenz*)
- **Fazit**: VEX-Instruktionen **nicht** mit nicht-VEX-Befehlen **mischen** (*um Performanzverschlechterung zu vermeiden*)
    - AVX Operationen sorgen dafür, dass CPU mit einer niedrigeren Frequenz operiert $\rightarrow$ Geschwindigkeit kann nur weniger als erwartet zunehmen

Zeitmessung

- mit Konsolenbefehl `time cmd` (`cmd` *kann kompiliertes Programm oder Befehl sein*), z.B. `time ./matr` oder `time ls -a`

```
real    0m.024s <- Abstand zwischen call und finish
user    0m.016s <- CPU-Zeit im User-Mode
sys     0m.016s <- CPU-Zeit im Kernel-Mode
```

- **Nachteil**: es wird alles gemessen (*inkl. Wartezeit auf Eingabe*) $\rightarrow$ keine selektive Messung möglich, `time` ungeeignet für Performancemessung

Zeitmessung im Code

- **Idee**: Messpunkt `end` - Messpunkt `start` = Dauer
- **Messung der Dauer der Berechnung** $\rightarrow$ **I/O Funktionen** sollten **nie** im Messbereich liegen
- zwischen den beiden Messungen soll **mindestens eine Sekunde Abstand** liegen $\rightarrow$ genauere Messung durch genügend Workload durch z.B. Schleifen
- mindestens Optimierungsstufe 2

Messen eines Zeitpunktes

- **Genauigkeit der Messung** abhängig von **Genauigkeit der Zeitpunkte**
- **Linearität der Uhrzeit**: Uhr darf nicht durch Zeitumstellungen oder Interrupts beeinflusst werden (*z.B. bei 1 Sekunde, die in der Uhr vergeht, vergeht genau 1 Sekunde in der Realität*)
- `time.h` liefert `int clock_gettime(clockid_t clk_id, struct timespec *tp)`
    - **Rückgabewert**: 0 falls erfolgreich, sonst -1
    - `clk_id`: bestimmt die Uhr, die verwendet wird; `clockid_t CLOCK_MONOTONIC` (*basiert auf in der Realität vergehenden Zeit, durch NTP-Syncs und* `adjustTime()` *beeinflusst*)
    - `*tp`: `struct timespec{time_t tv_sec; long tv_nsec;}`
        - `tv_sec`: aktuelle Zeit in Sekunden

    - `tv_nsec`: aktuelle Zeit in Nanosekunden
    - **Umwandlung in Sekunden**: `double sec = tv_sec + 1e-9 * tv_nsec;`

```
#include <time.h>
...
struct timespec start;
clock_gettime(CLOCK_MONOTONIC, &start) ;
for(int i = 0; i < iterations ; ++i)
    foo();
struct timespec end;
clock_gettime(CLOCK_MONOTONIC, &end);

double time = end.tv_sec - start.tv_sec + 1e-9 * (end.tv_nsec -
start.tv_nsec);
double avg_time = time / iterations;
```

Umgebungsbedingungen

- Überprüfung auf **Korrektheit der Berechnung**
- *Ergebnisse mit Varianz sind garantiert*: Messung soll **mindestens 3 mal** wiederholt werden, dann soll der Durchschnitt genommen werden
- *Abweichungen minimieren*: exklusive Nutzung der Ressourcen (*kein anderer Nutzer / anderes Programm darf Ressourcen beanspruchen*)
- Messung auf **echter Hardware** ohne **CPU-Features** (*e.g. Turbo, Hyperthreading*)

Dokumentation

- *Hintergrund*: **Reproduzierbarkeit** der Ergebnisse
- **Setup** des Systems (*genaue Bezeichnung des Prozessors und Frequenz, verfügbarer Speicher, Version des Betriebssystems*)
- **Kompiliervorgang** (*verwendeter Compiler, Version und Optionen*)
- **Testvorgang** (*wie oft, wie viel und mit welchen Eingaben wurde getestet*)
- **Aufbereitung** der Ergebnisse in **Diagramm** (*ohne logarithmische Skalierung*)

Zeitmessung - Zusammenfassung

- Zeitmessung mit `CLOCK_MONOTONIC`
- **Vermeidung unnötiger Operationen** im Messbereich
- mindestens **eine Sekunde Abstand** zwischen Messungen
- mindestens **Optimierungsstufe 2**
- mindestens **drei Wiederholungen der Messung**

- Bereitstellung **maximaler Ressourcen**
- möglichst **genaue Dokumentation**

Profiling mit `perf`

- **profiling**: Analyse der Laufzeit (+ Geschwindigkeit) eines Programms, um ineffiziente Bereiche aufzudecken
  - es lohnt sich, dort zu optimieren, wo ein Großteil der Rechenarbeit durchgeführt wird (*e.g. dort, wo 50% der Laufzeit verwendet wird, kann ein 2x-Speedup gewonnen werden*)
  - auch Speicherzugriffe kann mit profiling analysiert werden
- **Tracing Profiler**: fügen Instruktionen zum Programmcode hinzu (*entweder im Source Code, in ASM oder zur Laufzeit*)
  - **+**: präziser, bei jedem Funktionswechsel wird Zeit gemessen
  - **-**: Ausführung braucht länger
- **Sampling Profiler (`perf`)**: Programmcode wird nicht verändert
  - *stattdessen*: Profiler greift in regelmäßigen Abständen in Programmablauf ein und protokolliert stichprobenartig die Ereignisse seit der letzten Stichprobe (*e.g. aktuelle Adresse*)
  - **+:** schnellere Ausführung des Programms
  - **-:** ungenauere Analyse (*e.g. fehlende Protokollierung eines Funktionsaufrufs*)
- `time ./prog`: Messung der Zeit
- `perf record ./prog`: sammelt Daten und schreibt sie in `perf.data`
  - *standard*: jeder Thread wird protokolliert
    - kann aber per Prozess oder systemweit gemessen werden
- `perf report`: zeigt protokollierte Daten an
  - *1. Spalte*: wie häufig befand sich das Programm in einer bestimmten Methode
  - *2. Spalte*: Prozess (*unterschiedliche Einträge nur dann, wenn `perf` systemweit aufgerufen wird*)
  - *3. Spalte*: Name des zugehörigen ELF-Images
  - *4. Spalte*: `[Privilege]` (`k` *- Kernel,* `.` *- Nutzermodus*) + Name des Symbols / der Methode
- `perf list`: zeigt Events an, die protokolliert werden können
  - z.B. `perf record -e cache-misses ./prog`: zeigt in `Annotate x` auch die Anzahl an Cache-Misses an

---

Summary by Flavius Schmidt, ge83pux, 2023.
https://home.in.tum.de/~scfl