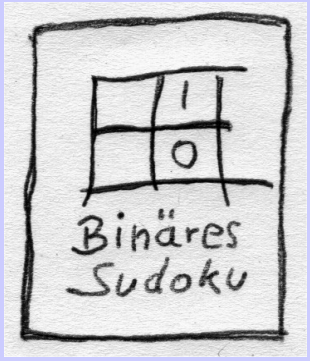
Jörn Fischer
j.fischer@hs-mannheim.de

Willkommen zur Vorlesung

Technische Informatik 2



Vorstellung

Zu meiner Person...

Überblick

Inhalt

1 - Organisatorisches

2 - Einführung

3 - Die Programmiersprache C/C++

4 - Debugger und Profiler

5 - Digitale Signalverarbeitung

6 - Prozessoren

7 - Virtuelle Maschinen

8 - Betriebssysteme

9 - Grafikkartenprogrammierung

10 - Mikrokontroller Programmierung

1 - Organisatorisches

1 - Organisatorisches

Kapitel 1 - Organisatorisches

- Wie läufts ab?
- Prüfungszulassung
- Fragen an mich

1 - Organisatorisches

Wie läufts ab?

- 2 SWS Vorlesung + 2 SWS Übungen
- Übungen in 1-3’er Gruppen, aber einzeln testiert
- Übungen sollen auch zu Hause programmiert werden!
- Unterlagen auf meiner Homepage

1 - Organisatorisches

Prüfungszulassung

- Die Übungsblätter werden online gestellt
- Testatdatum steht im Link auf das Übungsblatt
- ACHTUNG:

  Fällt beim Testat auf, dass eine Gruppe von einer anderen Gruppe die Ergebnisse kopiert hat, so verlieren beide Gruppen die Zulassung!

1 - Organisatorisches

Fragen an mich...

- Fragen können gerne auch per Email an mich gerichtet werden oder aber bei Diskussionsbedarf einfach per Mail einen Termin vereinbaren...
- Rückkopplung ist ausdrücklich erwünscht !!!!!

1 - Organisatorisches

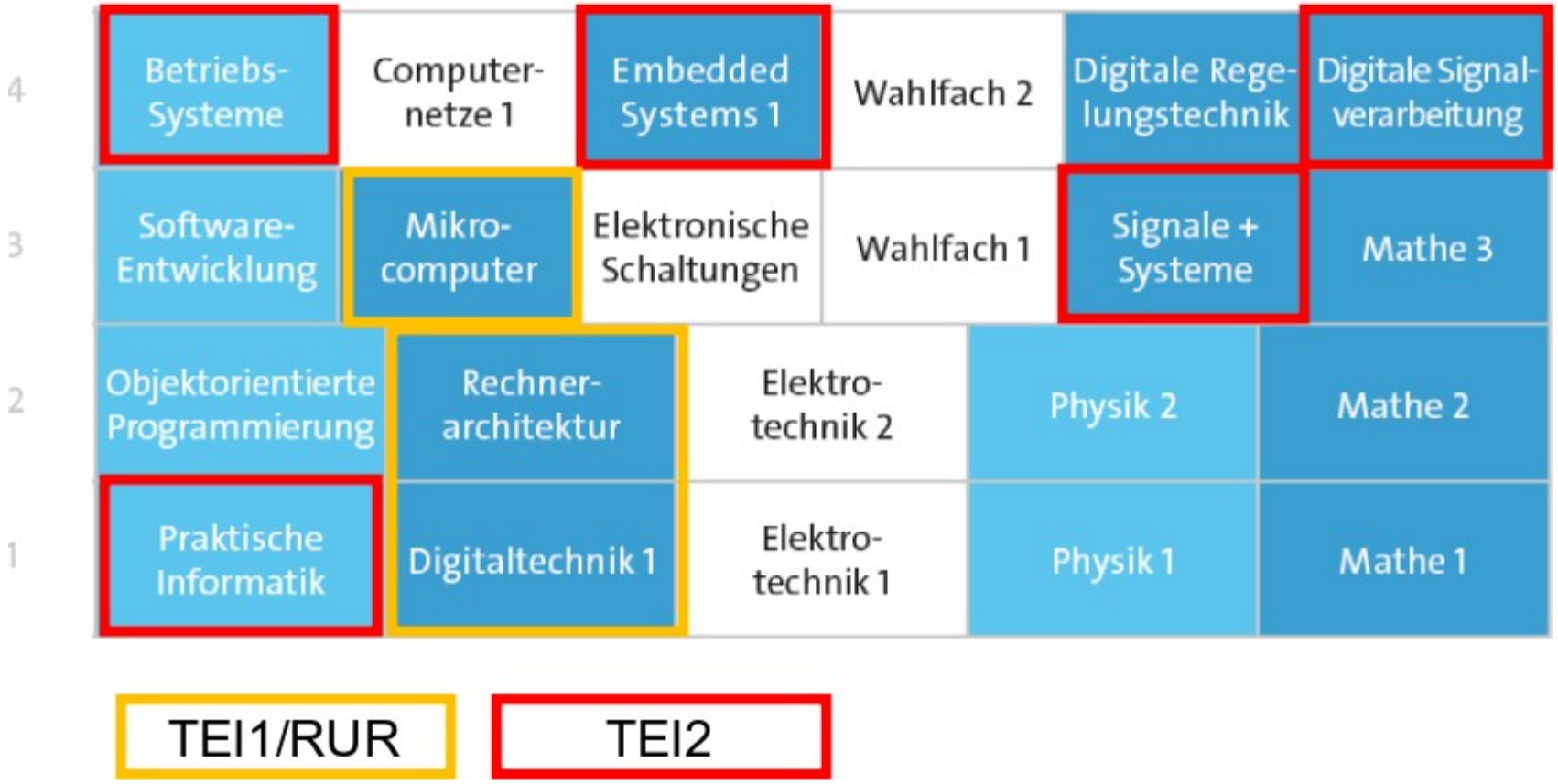
Haben Sie Fragen?

2 - Einführung

2 - Einführung

Kapitel 2 - Einführung

- Es gibt ganze Studiengänge „Technische Informatik“ z.B. HS-Mannheim



Kapitel 2 - Einführung

- Diese Vorlesung ist Fortsetzung der Technischen Informatik1
- Aus Bauteilen wie Flipflops, Zähler, Speicher, ALU etc. entstehen unterschiedliche Prozessorarchitekturen, die für unterschiedliche Anwendungsfälle optimiert sind
- So unterteilt man in CPU, µC, DSP und GPU, auf denen
  - Betriebssysteme samt Anwendungen,
  - Steuerungen der Hardware,
  - Signalverarbeitungs- oder
  - komplexe Berechnungsalgorithmen laufen

2 - Einführung

Warum sollte man als Informatiker die Hardware kennen?

- Um möglichst optimale Programme zu schreiben, müssen die folgenden Punkte berücksichtigt werden:
  - der Aufbau der Hardware
  - die Latenzzeiten und der Datendurchsatz der unterschiedlichen Speicher
  - die Größe der Speicher
  - die Ausführungszeit der unterschiedlichen Befehle

2 - Einführung

Warum sollte man als Informatiker die Hardware kennen?

- Bei hohen Anforderungen an die Rechengeschwindigkeit, bei vielen kooperierenden Threads, bei vielen Zugriffen über das Netzwerk, über den PCI-Bus etc...
  - … sollte die Performance optimiert werden
  - … sollten Zugriffe aufeinander abgestimmt sein
  - … sollten die Ressourcen optimal ausgenutzt werden
- sonst kann das System unbrauchbar werden, die Performance unerwartet zusammenbrechen, die Anforderungen an die Hardware steigen etc.

2 - Einführung

Beispiel (1):

- Aufgrund einer physikalischen Simulation soll ein Frühwarnsystem programmiert werden.
- Was, wenn die physikalische Simulation schlecht skaliert und deutlich langsamer als in Echtzeit läuft? (→ es resultiert ein „zu Spät-Warnsystem“)

2 - Einführung

Beispiel (2):

- Ein Navigationssystem soll während des Fahrens die Routenplanung neu berechnen
- Was, wenn die Planung einer Route eine halbe Stunde braucht, die Fahrtstrecke aber nur eine Viertelstunde entfernt ist?

2 - Einführung

Beispiel (3):

- Ein Roboter soll auf gesprochene Sprache reagieren.
- Was, wenn der eingebaute Prozessor nicht einmal die notwendige Fouriertransformation in Echtzeit berechnen kann?

2 - Einführung

- Darum ist mein MOTTO:

  „Sich auf das Wesentliche beschränken“

  d.h.

- Kein unnötiger Overhead was die Daten angeht
- Keine Rechenzeit vergeuden (,es sei denn die Entwicklungszeit steigt unverhältnismäßig)
- Typische Hardware kennen... und erst denken, dann programmieren!

3 - Die Programmiersprache C/C++

3 – Die Programmiersprache C/C++

Kapitel 3 – Die Programmiersprache C

- Literatur
- Unterschiede Java und C/C++
- Datentypen
- Operatoren
- Pointer (Wie und Warum)
- Speicher reservieren
- Typedefs, Structs und Unions

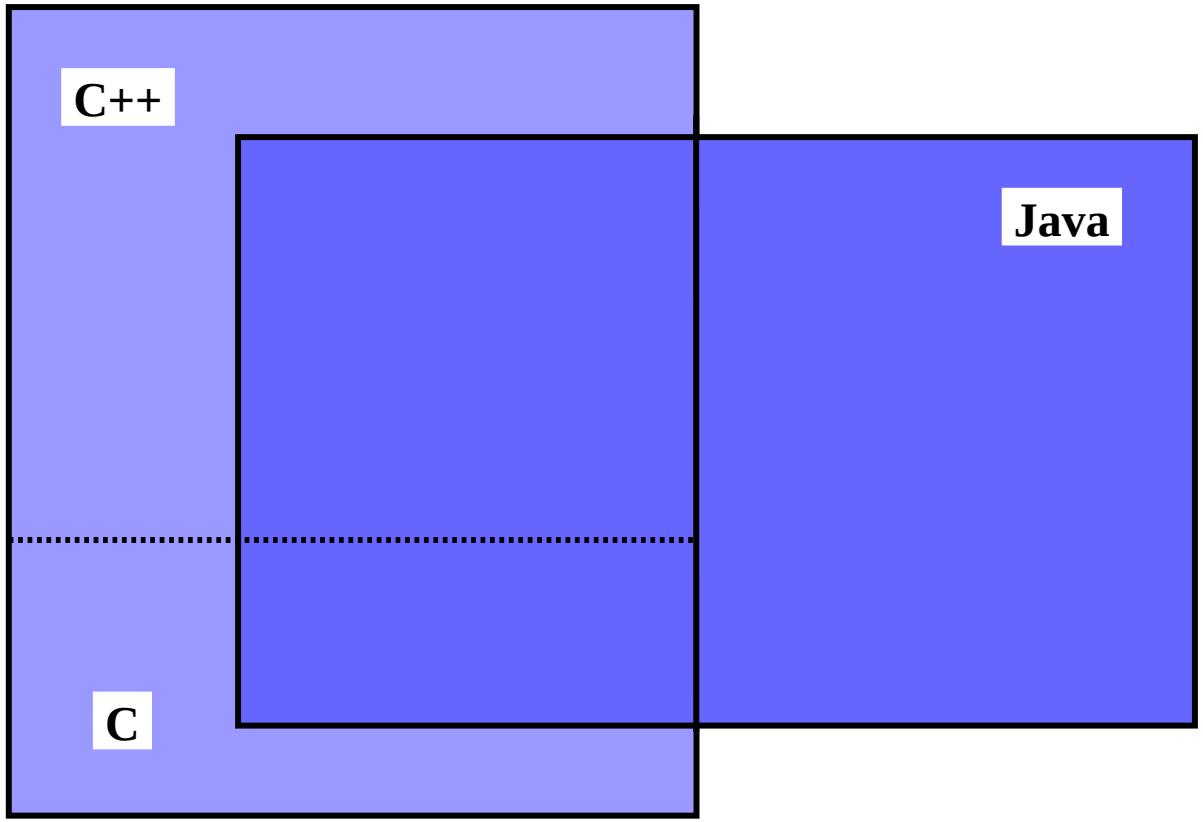
3 – Die Programmiersprache C/C++

Literatur

- B. Kernigan, D. Richie: Programmieren in C; Hanser Verlag 1990
- Bjarne Stroustrup: Die C++ Programmiersprache; Adison-Wesley, 4. Auflage, 2009
- Ulrich Breymann: Der C++-Programmierer; Carl Hanser Verlag, 2. Auflage, 2011
- Dietrich May: Grundkurs Software-Entwicklung mit C++;Springer, 2. Auflage, 2006
- de.wikibooks.org/wiki/C-Programmierung
- de.wikibooks.org/wiki/C++-Programmierung
- www.cplusplus.com
- www.cppreference.com
- www.cpp-tutor.de

3 – Die Programmiersprache C/C++

Unterschiede Java und C/C++



3 – Die Programmiersprache C/C++

Was fehlt in C/C++ (was es in Java gibt)

- In C gibt es keine Klassen und keine Objekte
- In C++ gibt es Klassen aber keine
  - Interfaces
  - Reflections
  - Garbage Collection
- Nur mit Zusatzbibliotheken gibt es
  - Grafikbibliotheken
  - GUI
  - Netzwerk

3 – Die Programmiersprache C/C++

Was fehlt in Java (was es in C++ gibt)

- Präprozessor
- Zeiger
- Effizientes Speichermanagement
- Operatorüberladung
- Mehrfachvererbung
- Namensräume
- Möglichkeiten der systemnahen Programmierung

3 – Die Programmiersprache C/C++

Einschränkungen in C:

- C ist rein prozedural (nicht objektorientiert, keine Vererbung, keine Templates...)
- Es gibt in C keine Referenzen, sondern nur Zeiger
- Lokale Variablen müssen in C immer am Anfang des Blocks definiert werden!
- Es gibt keine Klassen, sondern nur Structs (enthalten keine Funktionen)

3 – Die Programmiersprache C/C++

Welche Synonyme gibt es (Java → C/C++):

- Methoden → Funktionen
- Attribute → Variablen
- Generics → Templates
- abstrakte Methoden → virtuelle Methoden
- Interfaces → rein abstrakte Klassen (kommt dem am nächsten)

3 – Die Programmiersprache C/C++

Zum Einstieg ein Hello World Programm:

C

```
#include <stdio.h>
int main(int argc, char **argv){
     printf("Hello World!");
     return 0;
}
```

C++

```
#include <iostream>
class Hello{
   static void myMain(){
          std::cout << "Hello World!";
   }
};
int main(int argc, char**argv[]){
   Hello::myMain();
   return 0;
}
```

3 – Die Programmiersprache C/C++

- Die „printf“ Anweisung

```
#include <stdio.h>
int main(int argc, char** argv)
{
 printf("Ganze Zahl = %d \n",5);
 printf("Komma Zahl = %f \n",5.5);
 printf("Buchstabe  = %c \n",'A');
}
```

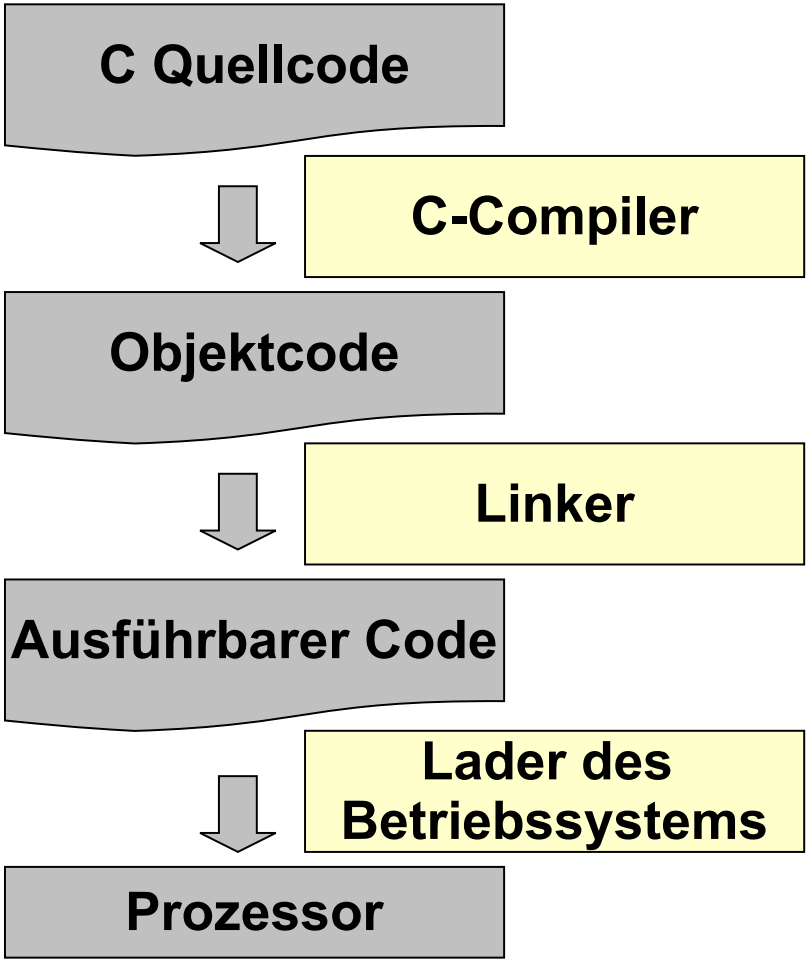
Fromatierungszeichen	Beschreibung
%d	ganzzahlig
%f	Fließkomma
%c	Character (Zeichen)
%s	String
\n	Zeilenvorschub

- Die „scanf“ Anweisung

```
 #include <stdio.h>
 int main(int argc, char** argv)
 {
  int zahl; float kommaZahl; char buchstabe; // declare variable
  scanf("%d",&zahl);                         // read integer
  scanf("%f",&kommaZahl);                    // read float
  scanf("%c",&buchstabe);                    // read char
 }
```

3 – Die Programmiersprache C/C++

Der Compiliervorgang:



- C-Programme liegen als Quellcode vor
- Ein Programm kann aus mehreren Quellcodedateien bestehen.
- Nur Files mit Änderung seit dem letzten Kompiliervorgang werden übersetzt
- Zunächst läuft ein Präprozessor über das Programm
- .c Dateien werden zu .o Object Files
- Dann wird mit Hilfe des Linkers aus den Object Files eine ausführbare Datei erstellt
- int main(int argc, char argv[][])wird beim Starten automatisch aufgerufen

3 – Die Programmiersprache C/C++

Der C/C++ Kompiliervorgang

- Kompilieren (einschließlich Präprozessor)

  gcc -c Hello.c

  (erzeugt Hello.o bzw. .obj)

- Linken:

  gcc -o Hello Hello.o [ggf. weitere .o Dateien]

  (erzeugt ausführbare Datei Hello bzw. Hello.exe)

- Ausführen

  Hello (unter Windows)

  ./Hello (unter Unix/Linux)

3 – Die Programmiersprache C/C++

MakeFiles zur Steuerung des Kompiliervorganges

- Im Makefile werden die Abhängigkeiten definiert

```
CC = /usr/bin/gcc
CFLAGS = -Wall -g -D_REENTRANT
LDFLAGS = -lm -lpthread

OBJ = datei1.o datei2.o datei3.o datei4.o

prog: $(OBJ)
	$(CC) $(CFLAGS) -o prog $(OBJ) $(LDFLAGS)

%.o: %.c
	$(CC) $(CFLAGS) -c $<
```

Mehr dazu: http://www.ijon.de/comp/tutorials/makefile.html

3 – Die Programmiersprache C/C++

Der Präprozessor

- Der Präprozessor ersetzt Quelltext im Programm

```
#include <stdio.h> // hier wird das Headerfile reinkopiert
#include "meinFile.h" // Header aus eigenem Pfad
#define PI 3.1415927  // definiert PI als 3.1415927
#define MAX(a,b) a>b?a:b // Macro Ersetzung
#if DEBUG_MODE // bedingte Übersetung
#else
#elif          // entspricht elseif
#endif
// mehrzeilige Präprozessoranw. mit \ am Ende der Zeile
```

3 – Die Programmiersprache C/C++

#include

Was macht der #include Befehl?

Was ist der Unterschied zwischen

```
#include "string.h"  // sucht zunächst im lokalen Ordner
#include <string.h>  // (c) sucht im standard Include Pfad
#include <string>    // (c++) sucht im standard Include Pfad
```

3 – Die Programmiersprache C/C++

#define

- In C werden Konstanten mit define deklariert.
- Makros ersparen durch Ersetzung im Quelltext den Funktionsaufruf.

```
#define TEXT "Hallo Welt"
#define TRUE 1
#define FALSE 0
#define DEBUG TRUE
#define MAX(a,b) a>b?a:b;
#define MAXX(a,b) if (a>b)                          \
                      printf("a ist groesser");     \
                  else                              \
                      printf("b ist groesser");
```

3 – Die Programmiersprache C/C++

#if #else #endif

Bedingte Compilierung:

```
#ifndef BERECHNUNG_H
#define BERECHNUNG_H
// obige Zeilen sogen dafür, dass das File
// nur einmal eingebunden wird, weil ab dann
// H_BERECHNUNG definiert ist.
...
#if DEBUG == TRUE
   printf("debugvalue=%d\n",debugvalue);
#else
   printf("\n");
#endif
...
#endif // BERECHNUNG_H
```

3 – Die Programmiersprache C/C++

#pragma

Was macht der Befehl #pragma?

- Der #pragma Befehl veranlasst den Präprozessor eine implementierungsspezifische Aktion (z.B. den Coprozessor mit zu benutzen)

```
// Beispiel: Parallelisierung mit Open-MP
#include <cstdio>
#include <omp.h>
int main(int, char*[]){
   #pragma omp parallel
   printf("Dies ist Thread %i von %i.\n",
       omp_get_thread_num(),
       omp_get_num_threads());
   return 0;
}
```

```
Ausgabe auf 2-Core-
Prozessor:

Dies ist Thread 1 von 2
Dies ist Thread 2 von 2
```

3 – Die Programmiersprache C/C++

Die Headerdatei

- In der Headerdatei werden Funktionen, Klassen, Typen deklariert
- Funktionen werden dort i.Allg. nur als Signatur hinterlegt
- So hat man pro Klasse immer ein .cpp und ein .h File
- Auch in c gibt es ein .c und ein.h File, welche gleich heißen
- Dem Compiler reicht das Headerfile mit den Signaturen um Funktionsaufrufe oder Objekt instanziierungen compilieren zu können.
- Libraries liegen dann nur noch als kompilierter Code und Headerfile vor.

3 – Die Programmiersprache C/C++

Die Headerdatei

Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
public:
    Vector();
    float getLength();
private:
    float x,y;
};
#endif // VECTOR_H
```

Vector.cpp

```
#include "Vector.h"
#include <math.h>

Vector::Vector()
{
  x=0; y=0;
}
float Vector::getLength()
{
  return sqrt(x*x+y*y);
}
```

3 – Die Programmiersprache C/C++

Der C/C++ Quelltext

- Der Compiler unterscheidet an der Endung des Files, ob das Programm als C Programm (.c Datei) oder als C++ Programm (.cpp Datei) compiliert werden soll
- Im Allgemeinen lassen sich .c Dateien auch als C++ Programm compilieren, indem man die Dateiendung auf .cpp ändert
- C++ Programme, die in einer .c Datei stehen lassen sich typischerweise nicht als .c Datei compilieren

3 – Die Programmiersprache C/C++

Schlüsselwörter:

asm	**auto**	**break**	**case**	*catch*
char	*class*	**const**	**continue**	**default**
delete	**do**	**double**	**else**	**enum**
except	**extern**	*finally*	**float**	**for**
friend	**goto**	**if**	*inline*	**int**
long	*new*	*operator*	*private*	*protected*
public	**register**	**return**	**short**	**signed**
sizeof	**static**	**struct**	**switch**	*template*
this	*throw*	*try*	**typedef**	**union**
unsigned	*virtual*	**void**	**volatile**	**while**

Zusätzlich in C++

3 – Die Programmiersprache C/C++

Elementare Datentypen

Datentyp	Wertebereich	Beschreibung
char	-128..127	ganzzahlig (1 ASC II Zeichen)
short	Bitbreite halb so groß oder gleich int meist: -32768..32767	ganzzahlig
int	Bitbreite des Targets mindestens 16 Bit	ganzzahlig
long int	Bitbreite doppelt so groß wie int	ganzzahlig
float	32 Bit, IEEE 754	Fließkommazahl
double	64 Bit	Fließkommazahl

3 – Die Programmiersprache C/C++

Datentypen

In C99 sind noch folgende Typen definiert in der <stdint.h>:

Datentyp	Wertebereich	Beschreibung
int8_t	-128..127	ganzzahlig
int16_t	-32768..32767	ganzzahlig
int32_t	-2147483648..2147483647	Ganzzahlig

- Zusätzlich gibt es die include Files: <limits.h> und <float.h> in denen die Min und Maxwerte der Datentypen stehen.
- Steht ein unsigned vor dem Datentyp, so beschreibt der neue Datentyp eine positive Zahl, dessen Wertebereich sich in den positiven Bereich verschiebt.

3 – Die Programmiersprache C/C++

Operatoren (Auswertreihenfolge)

Bezeichnung	Operatorsymbol	Priorität
Klammern	() []	13
Negation Inkrement Dekrement	- ! ~ ++ --	12
Arithmetische Operatoren	* / %	11
	+ -	10
Shift Operatoren	<< >>	9
Vergleichsoperatoren	> >= < <=	8
	== !=	7
Bitweise Operatoren	&	6
	^	5
	|	4
Logische Operatoren	&&	3
	||	2
Zuweisungsoperator	= += -= *= /= %= >>= <<= &= ^= |=	1

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Operator	Beschreibung
|	bitweise oder (OR)
&	bitweise und (AND)
!	bitweise nicht (NOT)
^	Ausschliessendes oder (XOR)
<<	links Verschiebung (shift left)
>>	rechts Verschiebung (shift right)

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Bitweise ODER (OR):

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Beispiel:

$$\begin{array}{ll} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} = 553_{dez} \\ | & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 1188_{dez} \\ = & 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1_{bin} = 1709_{dez} \end{array}$$

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Bitweise UND (AND):

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel:

$$\begin{array}{ll} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} = 553_{dez} \\ \& & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 1188_{dez} \\ = & 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0_{bin} = 32_{dez} \end{array}$$

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Bitweise NICHT (NOT):

A	~A
0	1
1	0

Beispiel:

~ 0 1 0 0 0 1 0 1 0 0 1 $_{bin}$ = 553_{dez}

= 1 0 1 1 1 0 1 0 1 1 0 $_{bin}$ = 1494_{dez}

Bitbreite ist entscheidend für das Ergebnis

➔ Wird hauptsächlich für `true == !false` verwendet

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Bitweise ausschließendes ODER (XOR):

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Beispiel:

$$\begin{array}{rl} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} = 553_{dez} \\ \mathbf{\hat{}} & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 1188_{dez} \\ \mathbf{=} & 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1_{bin} = 1677_{dez} \end{array}$$

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Links verschieben (SHIFT LEFT):

Nach links verschieben um x Stellen entspricht der Multiplikaton mit 2^x

Beispiel:

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} << 1 = 553_{dez} * 2$

$0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0_{bin} = 1106_{dez}$

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} << 2 = 553_{dez} * 4$

$1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 2212_{dez}$

3 – Die Programmiersprache C/C++

Bitweise Operatoren

Rechts verschieben (SHIFT RIGHT):

Nach rechts verschieben um x Stellen entspricht der Integer-Division durch 2^x

Beispiel:

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin}$ >> 1 = 553_{dez} / 2

$0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0_{bin}$ = 276_{dez}

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin}$ >> 2 = 553_{dez} / 4

$0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0_{bin}$ = 138_{dez}

3 – Die Programmiersprache C/C++

Wie kann man Bits extrahieren?

Gegeben sei eine vorzeichenlose Integerzahl $0x7A_{hex}$.

Wie kann ich die Zahl extrahieren, die zwischen Bit 4 und Bit 7 steht:

Bit	7	6	5	4	3	2	1	0
Wert 0x7A=	0	1	1	1	1	0	1	0
&	1	1	1	1	0	0	0	0
=	0	1	1	1	0	0	0	0
>>4 =	0	0	0	0	0	1	1	1

Man schiebt die Zahl um 4 Bits nach rechts, damit sie rechtbündig ist und verundet sie mit der Zahl, die sich ergibt, wenn man alle zu extrahierenden Bits gleich eins setzt 0b1111: ergebnis=(0x7A >> 4) & 0b1111; // Ergebnis = 7

3 – Die Programmiersprache C/C++

Wie kann man Bits setzen?

Gegeben sei eine vorzeichenlose Integerzahl $0x5A_{hex}$.

Wie kann ich Bit 5 bis Bit 7 setzen?

Bit	7	6	5	4	3	2	1	0
Wert 0x5A =	0	1	0	1	1	0	1	0
| 0xE0	1	1	1	0	0	0	0	0
=	1	1	1	1	1	0	1	0

0x5A | 0xE0 = 0xFA

3 – Die Programmiersprache C/C++

Wie kann man Bits löschen?

Gegeben sei eine vorzeichenlose Integerzahl $0x5A_{hex}$.

Wie kann ich Bit 5 bis Bit 7 löschen?

Bit	7	6	5	4	3	2	1	0
Wert 0x5A =	0	1	0	1	1	0	1	0
& 0x1F	0	0	0	1	1	1	1	1
=	0	0	0	1	1	0	1	0

0x5A &0x1F = 0x1A

3 – Die Programmiersprache C/C++

Pointer

- Zeiger (englisch: pointer) sind Variablen, deren Inhalt eine Speicheradresse ist
- Sie sind auch von einem definierten Typ, wobei der Typ nichts mit dem Inhalt der Variablen zu tun hat, sondern nur mit deren Inkrementierung und Dekrementierung

z.B. wenn double auf einem 8 Bit Prozessor 4 Byte einnimmt

```
double *pointer1 = &variable;

char   *pointer2 = &variable;

pointer1++; // pointer1 wird um 8 inkrementiert

pointer2++; // pointer2 wird um 1 inkrementiert
```

3 – Die Programmiersprache C/C++

Pointer

- Da ein Pointer nach der Deklaration irgendwo in den Speicher zeigt birgt das die Gefahr, dass man über den Pointer irgendwo in den Speicher schreibt.
- Darum ist es wichtig den Zeiger zu initialisieren.
- Falls die Variable, auf die gezeigt werden soll noch nicht bekannt ist sollte der Pointer mit NULL initialisiert werden:

```
int *pZeiger;

pZeiger = NULL;
```

3 – Die Programmiersprache C/C++

Adressoperator

Mit dem Adressoperator & bekommt man die Adresse einer Variablen:

```
int a,b,*pZeiger;
a=5;
pZeiger=&a; // pZeiger zeigt auf die Adresse von a
*pZeiger=6; // schreibt in die Adresse die Zahl 6
            //(äquivalent zu a=6;)
pZeiger=&b; // pZeiger zeigt auf die Adresse von b
*pZeiger=2; // schreibt 2 in b (äquivalent zu b=2;)
```

3 – Die Programmiersprache C/C++

Pointer auf pointer

Es können auch Pointer auf Pointer definiert werden:

```
int a,*pZeiger,**ppZeiger;

pZeiger=&a;                 // zeigt auf Adresse von a

ppZeiger=&pZeiger;          // zeigt auf Adresse von p Zeiger a=5;

*pZeiger=6;                 // entspricht a=6;

**ppzeiger=7;               // entspricht a=7;
```

3 – Die Programmiersprache C/C++

Pointer (Operatorreihenfolge)

Wert an der Stelle, auf die pointer zeigt auslesen und pointer erhöhen:

```
wert=*pointer++
```

...das ist die gebräuchliche Notation

Wert an der Stelle, auf die pointer zeigt auslesen und diesen Wert erhöhen:

```
wert=++*pointer
```

…bitte so NICHT verwenden

…bitte nur einfache Ausdrücke verwenden...Negativbeispiel:

```
char variable[4] = {0,1,2,3};

char *pointer = &variable[0];

*pointer = *pointer++ + ++*pointer;
```

3 – Die Programmiersprache C/C++

Speicher reservieren

Statt Speicher mit z.B. *int varArray[100]* zu reservieren kann man auch die Funktion *malloc()* benutzen.

```
int *pVarArray;

int groesse = sizeof(int)*numberOfElements;

pVarArray = malloc(groesse);

if (pVarArray){

   // Speicher wurde reserviert!

}
```

Da der Speicher nicht automatisch freigegeben wird, muss er mit free() wieder freigegeben werden. Sonst können Memory Leaks (Speicherlecks) entstehen.

```
free(pVarArray);
```

3 – Die Programmiersprache C/C++

c-Strings

Zunächst c-String Bibliothek einbinden:

```
#include <string.h>

char* myText = "Hallo Welt"

printf("%s\n",myText);
```

3 – Die Programmiersprache C/C++

c-Strings

Länge eines Strings abfragen mit der Funktion
strlen(<String>)

- liefert Anzahl der Zeichen im übergebenen String als int-Wert.

Beispiele:

```
char *name = "Jörn Fischer";
printf ("%d\n",strlen(name));
```

→ **12**

```
char *a = "a";
printf ("%d\n",strlen(a));
```

→ **1**

3 – Die Programmiersprache C/C++

c-Strings

String kopieren mit der Funktion
strcpy(<kopie>, <quelle>)

```
// Speicher für neuen String belegen
char *quelle = "Beispieltext";
char *kopie = malloc(strlen(quelle)+1);   // +1 Nullbyte
// String kopieren
strcpy(kopie,quelle);
```

3 – Die Programmiersprache C/C++

c-Strings

n Zeichen des Strings kopieren mit der Funktion
strncpy(<kopie>, <quelle>, <n>)

```
// reserve memory for the string

char *kopie = malloc(4);

// copy string

strncpy(kopie,quelle, 4);
```

3 – Die Programmiersprache C/C++

c-Strings

Zwei Strings aneinanderhängen (konkatenieren) mit der Funktion
strcat(<ziel>, <quelle>)

- Zielstring bekommt Quellstring hinten drangehängt

```
char *vorname = "Jörn";

char *nachname ="Fischer";

// Speicher für Zielstring belegen

char *name = malloc(20);       // Platz für 19 Zeichen plus
Nullbyte

strcpy(name,vorname);          // Vornamen kopieren

strcat(name, " ");             // Leerzeichen dazwischen

strcat(name, nachname);        // Nachnamen hinten dran hängen

printf("%s\n",name);           // Gibt „Jörn Fischer" aus
```

3 – Die Programmiersprache C/C++

c-Strings

n Zeichen des Quellstrings an Zielstring anhängen mit der Funktion
strncat(<ziel>, <quelle>, <n>)

- Zielstring bekommt n Zeichen des Quellstrings hinten drangehängt

```
char *vorname = "Jörn";

char *nachname ="Fischer";

// Speicher für Zielstring belegen

char *name = malloc(20);       // Platz für 19 Zeichen plus
Nullbyte

strcpy(name,vorname);          // Vornamen kopieren

strcat(name, " ");             // Leerzeichen dazwischen

strncat(name, nachname, 3);    // 3 Buchst. des Nachnamens anh.

printf("%s\n",name);           // Gibt „Jörn Fis" aus
```

3 – Die Programmiersprache C/C++

c-Strings

Inhalt zweier Strings vergleichen mit der Funktion
strcmp(<String1>, <String2>)

- liefert **0**, wenn beide Strings zeichenweise gleich
- liefert +1, wenn String1 größer String2 (lexikalisch nach String2)
- liefert -1 wenn String1 kleiner String2 (lexikalisch vor String2)

!strcmp(<String1>, <String2>)

- liefert **1**, wenn beide Strings zeichenweise gleich, sonst 0.

```
char *t1="Jörn";
char *t2="Jörn";
char *m = "Markus";
printf("!strcmp(%s, %s)=%d\n", t1, t2, !strcmp(t1,t2)); // returns 1
printf("!strcmp(%s, %s)=%d\n", t1, m, !strcmp(t1,m));   // returns 0
```

3 – Die Programmiersprache C/C++

c-Strings

Suchstring im Quellstring herausfinden
char *strstr(<quellString>, <suchString>)

- liefert den Quellstring ab der Stelle, an der der Suchstr. gefunden ist
- liefert NULL, wenn Suchstring im Quellstring nicht enthalten

Beispiel:

```
char *name = "Jörn Fischer";
char *fischer = strstr(name,"Fischer");
printf ("%s\n",fischer);      // Fischer
char *fischer2 = strstr(name,"fischer");
printf ("%s\n",fischer2);     // (null)
char *fischer3 = strstr(name,"Jörn");
printf ("%s\n",fischer3);     // Jörn Fischer
```

3 – Die Programmiersprache C/C++

Dateizugriffe (Fopen(), fprintf(), fclose())

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

int main(int argc, char *argv[])
{
   int t;
   FILE *out;
   out = fopen("Filename.txt","wb");
   if (out != NULL){
      for (t=0;t<100;t++){
          fprintf(out, "Test\n");
      }

      fclose(out);//closes the File
   }
  return 0;
}
```

```
open("file",xxx);
xxx = w für schreiben
xxx = r für lesen(File existiert)
xxx = r+ für lesen + schreiben
xxx = w+ für lesen + schreiben
xxx = a+ für lesen und erweitern
xxx = wb für binäres schreiben
xxx = rb für binäres lesen

fprintf() schreibt das, was
bei printf() auf dem Bildschirm

landet ins File!
```

3 – Die Programmiersprache C/C++

Dateizugriffe (Fgetc())

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

int main(int argc, char *argv[])
{
   char letter;
   FILE *in;
   in = fopen("Filename.txt","rb");
   if (in != NULL){
      while(!feof(in)){
         letter=fgetc(in);
         printf("%c",letter);
      }
      fclose(in);
   }
   return 0;
}
```

fgetc liest ein char aus dem File

3 – Die Programmiersprache C/C++

Dateizugriffe (Fputc())

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

int main(int argc, char *argv[])
{
   int t;
   FILE *out;
   out = fopen("Filename.txt","wb");
   if(out != NULL){
        for (t=0;t<100;t++){
            fputc('A',out);
        }
        fputc('\n',out);
        fclose(out);
   }
   return 0;
}
```

fputc schreibt ein char ins File

3 – Die Programmiersprache C/C++

Dateizugriffe (Fseek())

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   FILE *out;
   out = fopen("Filename.txt",“w");
   if(out != NULL){
      fputs("Dies ist ein Irrtum.",out);
      fseek(out, 13, SEEK_SET);
      fputs("Beispiel!" ,out);
   }
   fclose(in);
   return 0;
}
```

```
fseek(FILE*stream, long int
offset, int origin)
```

offset = Anzahl der bytes vom Ausgangspunkt

origin = SEEK_SET (Fileanfang)
= SEEK_CUR (aktuelle Pos)
= SEEK_END (Fileende)

3 – Die Programmiersprache C/C++

Typedefs

Mit der typedef Anweisung kann man einem Datentyp einen anderen Namen geben:

```
#if GPU_CODE
    typedef float FLOATING;
#else
    Typedef double FLOATING
#endif
FLOATING var1;
```

3 – Die Programmiersprache C/C++

Strukturen

Strukturen werden in C dazu benutzt, Daten verschiedenen Typs, die zusammen gehören, in einer Datenstruktur zusammenzufassen.

```
struct schwein
{
 char name[20];
 float groesse;
 float gewicht;
}rudi,nele; // rudi und nele sind Instanzen von Schwein
```

3 – Die Programmiersprache C/C++

Strukturen

Eine Instanz eines structs benutzt man folgendermassen:

```
struct schwein berta; // berta is an instance of schwein
strcpy(berta.name,"Berta");
berta.groesse=0.95;
berta.gewicht=200;

struct schwein *pZeigerAufBerta;
pZeigerAufBerta=&berta;
pZeigerAufBerta->groesse += 0.1; // berta.groesse is enlarged
pZeigerAufBerta->gewicht  = 210; // berta.gewicht = 210 kg
```

3 – Die Programmiersprache C/C++

Unions

Im Gegensatz zu einer Struktur kann eine Union in einer Variablen unterschiedliche Typen speichern:

```
union eineVariable
{
 // Instance of eineVariable may be of type int or array
 // of float, but not both

 int intWert;
 float fliesskomaWert[10];
};
```

3 – Die Programmiersprache C/C++

...bis hier ging alles in C

… ab hier gibts C++

3 – Die Programmiersprache C/C++

Kompatibilität C/C++

- Ein C++ Compiler erzeugte früher im ersten Schritt C Code, der erst in weiteren Schritten in Assembler umgewandelt wurde.
- So ergibt sich automatisch eine gewisse Kompatibilität von C++ zu C.
- C++ kann sowohl prozedural als auch objektorientiert programmiert werden.
- Die Datenkapselung ist wesentlich konsequenter handhabbar als in C.

3 – Die Programmiersprache C/C++

Was man in C++ anders macht...

In C++ sollten folgende Dinge vermieden werden, da vor allem die Typsicherheit nicht gewährleistet ist:

- #define Textersetzung ohne Typenprüfung
- malloc liefert void pointer zurück, der typenlos ist
- unions
- void * ist typenlos
- casts Typekonvertierung kann Fehler provozieren

3 – Die Programmiersprache C/C++

Was man in C++ anders macht...

Statt Konstanten mit #defines zu definieren können wir mit const eine Konstante definieren. Speichereffizienter ist jedoch die enum Anweisung:

```
#include <stdio.h>

int main(int argc, char **argv){
  enum Color { red, green, blue };
  Color r = red; // Value range check by compiler
  switch(r)
  {
    case red  : printf("red\n");   break;
    case green: printf("green\n"); break;
    case blue : printf("blue\n");  break;
  }
}
```

3 – Die Programmiersprache C/C++

Was man in C++ anders macht...

Die const Anweisung kann auch in Zusammenhang mit Pointern verwendet werden.

```
char c1 = 'A', c2='B';
char* p1
const char* p2;                 // pointer is protected
char* const p3=&test;           // contents is protected
const char* const p4= &test;    // both protected

p1 = &c1;
*p1 = 'X';
p2 = &c1;     // not possible! Pointer would be changed
*p2 = 'X';
p3 = &c1;
*p3 = 'X';    // not possible! Contents would be changed
p4 = &c1;     // not possible! Pointer would be changed
*p4 = 'X';    // not possible! Contents would be changed
```

3 – Die Programmiersprache C/C++

Was man in C++ anders macht...

In C++ kann auch eine Methode (Funktion) als const definiert werden. Dann können die Klassenattribute nicht durch diese Funktion verändert werden. Beispiel dafür sind getter Methoden.

```
int fkt() const; // class-attributes can not be changed!
```

Neue Sprachmittel

3 – Die Programmiersprache C/C++

Neue Sprachmittel

- Deklaration für einen Block
- Struct
- Typenkonversion
- Defaultparameter
- Referenzen
- New und delete
- Namespaces

3 – Die Programmiersprache C/C++

Deklaration für einen Block

Zunächst kann man in C++ eine Variable auch im laufenden Programmcode deklarieren z.B.:

```
for (int i=0; i<10; i++){  // variable i is only valid insode of
                           // the „for“ loop
}
// or it might be declared inside of a Block
{
 int zahl=0;
 zahl+=1;
} // after the block the variable does not exist anymore!
```

3 – Die Programmiersprache C/C++

Struct

Der Ausdruck struct kann bei der Instanziierung (Deklaration) weggelassen werden

```
struct Schwein{
    int groesse;
    int gewicht;
};
Schwein Berta; // in C++ style ...
               // in plain C a struct would be necessary before the
               // Schwein
```

3 – Die Programmiersprache C/C++

Typenkonversion

Die Typenkonversion zwischen bool und int ist definiert:

```
bool wahr_iable=100; // äquivalent zu wahriable=true;

int ergebnis=false;// false->0, true->1

bool bIsOk=(wahr_iable!=ergebnis); // ergibt in diesem Fall true
```

3 – Die Programmiersprache C/C++

Default Parameter

Man kann bei Funktionen "Default Parameter" angeben. Sind "Default Parameter" angegeben, so kann man die letzten Argumente beim Aufruf der Funktion weglassen. Diese werden dann mit den Defaultwerten belegt:

```
// Funktionsdeklaration

int Funktion(int a, int b=1, int c=2);

// Aufruf mit a=1,b=2,c=2

Funktion(1,2);
```

3 – Die Programmiersprache C/C++

Referenzen

Mit vorangestelltem & wird die Deklaration einer Variable zur Deklaration einer Referenz:

```
int Zahl=5;
int &verweisAufZahl = Zahl; // Deklaration einer Referenz!
int pointer*;

verweisAufZahl=6; // damit ändert man die Variable Zahl auf 6
pointer=&verweisAufZahl; // nun zeigt auch die Variable pointer
                         // auf die Adresse der Variablen "Zahl"
```

3 – Die Programmiersprache C/C++

Besonders häufig werden Referenzen bei Funktionsübergaben verwendet:

```
void addiereUndSubtrahiere(int& ergebnis1, int& ergebnis2,int zahl1,
   int zahl2)
{
    ergebnis1=zahl1+zahl2;
    ergebnis2=zahl1-zahl2;
}
int main(void)
{
  int a=1,b=1;
  int erg1,erg2;
  addiereUndSubtrahiere(erg1,erg2,a,b); // equivalent: erg1=a+b;
                                        //             erg2=a-b;
}
```

3 – Die Programmiersprache C/C++

new und delete

Statt wie in ANSI-C mit malloc Speicher zu reservieren bietet C++ die Operatoren "new" und "delete" bzw. "new[]" und "delete[]".

```
int *feld;

feld = new int[100]; // Speicher reservieren

feld[0]=123;         // Speicher nutzen

delete[] feld;       // Speicher freigeben
```

3 – Die Programmiersprache C/C++

namespaces

- Wenn in einem grossen Projekt z.B. in zwei Packages jeweils eine Funktion mit gleichem Namen und gleichen Übergabeparametern existieren, so kann ein C wie ein C++ Compiler beim Aufruf dieser Funktion nicht entscheiden, welche der zwei Implementationen verwendet werden soll.
- Um existierenden Code so abzukapseln, dass später doch beide Funktionen aufgerufen werden können, benutzt man Namespaces.
- Der Aufruf wird dann mit dem scope operator :: realisiert.
- Anonyme namespaces sind nur im Modul sichtbar: namespace{...}

3 – Die Programmiersprache C/C++

```
namespace package1
{
 void f(){
   // implemetation von package1
 }
}
namespace package2
{
 void f(){
   // implemetation von package2
 }
}
int main(void)
{
 package1::f(); // ruft f von package1 auf
 package2::f(); // ruft f von package2 auf
}
```

Klassen, Objekte und Methoden

3 – Die Programmiersprache C/C++

Klassen Objekte und Methoden

- Unterschiede zwischen Class und Struct
- Zugriffsschutz
- Explizit und implizit inline
- Statische Attribute und statische Methoden
- Konstruktoren und Destruktoren
- Der Kopierkonstruktor (Teil 1)
- Weiterleiten von Initialisierungen

3 – Die Programmiersprache C/C++

Unterschied zwischen Class und Struct

- In C++ kann "struct" durch "class" ersetzt werden.
- Der wesentliche Unterschied zum struct ist, dass innerhalb einer Klasse normalerweise auch die Funktionen implementiert werden, die das Objekt modellieren und auf die Membervariablen zugreifen.
- Ausserdem sind Variablen, die in einer Klasse implementiert werden nur von deren Funktionen zugreifbar, es sei denn man schreibt das Schlüsselwort `public:` davor.
- Membervariablen sind die Variablen, die innerhalb der Klasse deklariert werden und das Objekt beschreiben.

3 – Die Programmiersprache C/C++

Zugriffsschutz

- Streng genommen greift man in C++ von ausserhalb einer Klasse nur mit Hilfe sogenannter setter und getter Funktionen auf eine Membervariable der Klasse zu.
- Damit ist die Klasse optimal gekapselt.
- Schreibt man private: vor Membervariablen oder Memberfunktionen, so ist der Zugriff von ausserhalb des instantiierten Objektes nicht möglich.
- Die Variablen und Funktionen sind geschützt. Schreibt man public: vor die Variablen oder Funktionen, so kann auch von auserhalb auf die Funktionen und Variablen zugegriffen werden.

3 – Die Programmiersprache C/C++

Explizit und implizit inline

- Inline Methoden können vom compiler wie ein Makro an die Stelle des Aufrufs kopiert werden.
- Das spart den Overhead beim Aufruf, braucht aber mehr Programmspeicher.
- Wenn die Methoden innerhalb der Klassendeklaration implementiert sind, sind sie implizit inline.
- Mit dem Schlüsselwort **inline** vor einer Memberfunktion kann eine Funktion auch explizit als inline deklariert werden.

3 – Die Programmiersprache C/C++

```
class CSchwein
{
 private: // kann von extern nur über setter und getter Funktionen zugegriffen werden
 int m_groesse; // Variablen oder Attribute
 int m_gewicht; // Variablen oder Attribute

 public: // mit public können die Funktionen auch von ausserhalb genutzt werden

 CSchwein(int groesse=0,int gewicht=0) // Konstruktor wird beim instantiieren eines Objekts
                                       // aufgerufen
 {                                     // wenn im Headerfile implementiert, dann automatisch
                                       // inline (wie ein Makro)
  m_groesse=groesse; // Variablen initialisieren
  m_gewicht=gewicht; // Variablen initialisieren
 }
 ~CSchwein(); // Destruktor wird aufgerufen, wenn die Instanz des objektes zerstört wird
 void setGroesse(int groesse);  // Deklaration der Methode,Memberfunktion oder Elementfunktion
 void setGewicht(int gewicht);  // Deklaration der Methode,Memberfunktion oder Elementfunktion
 int getGroesse();             // Deklaration der Methode,Memberfunktion oder Elementfunktion
 int getGewicht();             // Deklaration der Methode,Memberfunktion oder Elementfunktion
};
```

3 – Die Programmiersprache C/C++

```
CSchwein::~CSchwein() // Destruktor wird aufgerufen, wenn die Instanz des Objektes zerstört wird
{
// typischerweise stehen hier die delete Aufrufe
}
void CSchwein::setGroesse(int groesse)  // Methode,Memberfunktion oder Elementfunktion
{
  m_groesse=groesse;
}
void CSchwein::setGewicht(int gewicht)  // Methode,Memberfunktion oder Elementfunktion
{
  m_gewicht=gewicht;
}
int CSchwein::getGroesse()              // Methode,Memberfunktion oder Elementfunktion
{
  return m_groesse;
}
int CSchwein::getGewicht()              // Methode,Memberfunktion oder Elementfunktion
{
  return m_gewicht;
}
```

3 – Die Programmiersprache C/C++

```
// --- Main.cpp File ------------------------------------------------

int main(void)
{
 CSchwein Berta; // Berta wird instantiiert und ist dann ein Objekt oder eine Instanz
                 // der Klasse CSchwein
 int Gewicht;

 Berta.setGewicht(200);// Berta ist 200kg schwer
 Gewicht=Berta.getGewicht();
}
```

3 – Die Programmiersprache C/C++

Statische Attribute und statische Methoden

- Methode, die nicht als static deklariert werden bekommen vom Compiler als erstes Argument den **this** pointer, einen Zeiger auf die eigene Instanz der Klasse.
- Statische Methoden bekommen vom compiler keinen **this** pointer.
- Sie können nur auf statische Attribute zugreifen!
- Statische Attribute (Variablen) in einer Klasse existieren genau einmal pro Klasse schon bevor überhaupt eine Instanz der Klasse existiert.

Beispiel: Keine Instanz von einem Konto, aber Zinssatz kann schon festgelegt werden, da setZinssatz() und f_zinssatz als static deklariert sind! Statische Attribute müssen explizit initialisiert werden.

3 – Die Programmiersprache C/C++

```
class Ckonto //im Header File
{
 float m_Kontostand;
 static float m_Zinssatz; // für alle Instanzen ist der Zinssatz
                          // gleich, da diese Variable nur einmal
                          // im Speicher existiert!
 public:
 static void setZinssatz(float zinssatz){
    m_Zinssatz=zinssatz;
 }
 static float getZinssatz(){
    return m_Zinssatz;
 }
}
int main(void){ // im cpp File
 CKonto::setZinssatz(0.3);
 printf("Zinssatz = %f\n",CKonto::getZinssatz());
}
```

Konstruktoren und Destruktoren

Der Konstruktor ist eine Memberfunktion, die aufgerufen wird, wenn eine Instanz eines Objekts instantiiert (bzw. deklariert) wird.

Dort können Membervariablen initialisiert werden. Wird das Objekt zerstört, z.B. durch beenden des Programms, so wird der Destruktor aufgerufen.

Im Destruktor sollten alle mit "new" reservierten Speicher mit "delete" wieder freigegeben werden.

3 – Die Programmiersprache C/C++

```
CSchwein::CSchwein(int groesse=0,int gewicht=0)
          // Konstruktor wird beim instantiieren eines Objekts
          // aufgerufen
{ // int *feld; wird im Headerfile der Klasse deklariert!
  feld = new int[100];// Speicher reservieren
  feld[0]=123;        // Speicher initialisieren
  ...
}
CSchwein::~CSchwein() // Destruktor wird aufgerufen, wenn die
                      // Instanz des Objektes zerstört wird
{
delete[] feld;    // Speicher freigeben
}
```

Sowohl ein Konstruktor als auch ein Destruktor hat keinen Rückgabewert!

Der Kopierkonstruktor (Teil 1)

Eine Besonderheit stellt der Kopierkonstruktor dar. Es gibt ihn ohne dass man ihn implementiert:

```
Schwein Berta(200,100);
Schwein Olaf(Berta);  // hier kopiert der Copy-Konstruktor
                      // die Daten von Berta in die Instanz Olaf
Schwein Rudolf=Berta; // hier kopiert der Copy-Konstruktor
                      //die Daten von Berta in die Instanz Rudolf

Rudolf=Olaf; // kopiert alle Variablen von Olaf in Rudolf, aber
             // mittels des = Operators!

fuettere(Olaf); // Call by value: Kopierkonstruktor wird benutzt
Rudolf=groesstesSchwein(); // Return value: Kopierkonstruktor benutzt
```

3 – Die Programmiersprache C/C++

Der Kopierkonstruktor kann zu unerwarteten Ergebnissen führen, vor allem wenn im eigentlichen Konstruktor mehr gemacht wird, als die Variablen zu initialisieren.

z.B. werden Pointer auch nur kopiert und zeigen dann auf die gleichen Speicherbereiche wie die Original Instanz

Man kann sich z.B. einen eigenen Kopierkonstruktor schreiben (siehe Teil 2):

```
Schwein::Schwein(const Schwein& T) {
  // here the copy-constructor is implemented!!!
}
```

Möchte man verhindern, dass ein Kopierkonstruktor benutzt wird, so definiert man sich eine Kopierkonstruktor Funktion als private: …

Weiterleiten von Initialisierungen

Werden Objekte (wie im Unteren Beispiel departure und arrival) innerhalb einer Klasse instantiiert und deren Konstruktor aufgerufen, so kann vom Klasseneigenen Konstruktor Daten über die "Member Initialization List" (mit Komma getrennte Liste hinter Konstruktor) weitergeleitet werden

3 – Die Programmiersprache C/C++

```
class CflightConnection{
 public:
 //
            Daten werden an weitere Konstruktoren weitergeleitet
 CflightConnection(const char*depart, const char* arrive,
              int flighnumber):departure(depart), arrival(arrive)
 {
 }
 private:
 Ctime departure; // hier wird der Konstruktor mit den Daten
                  // der Weiterleitung des obigen eigenen
                  // Konstruktors gefüttert.
 Ctime arrival;   // hier wird der Konstruktor mit den Daten
                  // der Weiterleitung des obigen eigenen
                  // Konstruktors gefüttert.
}
```

3 – Die Programmiersprache C/C++

Dabei könnten „departure“ und „arrival“ auch const oder eine Referenz & sein:

```
const Ctime departure; // const variablen können nur mit Member
                       // Initialization List oder direkt
                       // initialisiert werden
Ctime& arrival;        // referenz kann nur mit Member
                       // Initialization List oder direkt
                       // initialisiert werden
```

Überladen von Funktionen und Operatoren

3 – Die Programmiersprache C/C++

Überladen von Funktionen und Operatoren

- Überladen von Funktionen
- Der Kopierkonstruktor (Teil 2)
- Name Mangeling
- Extern “c“
- Überladen von Operatoren

3 – Die Programmiersprache C/C++

Überladen von Funktionen

In C war es nicht möglich Funktionen mit gleichem Funktionsnamen aber unterschiedlichen Parametern zu definieren.

In C++ ist dies möglich:

```
int addiere(int a,int b);          // 1. Funktion
int addiere(float a, float b);     // 2. Funktion
int addiere(char a, double b);     // 3. Funktion
```

Für den Aufruf entscheidet dabei der Compiler welche der deklarierten Funktionen aufgerufen wird. Z.B. ruft addiere(1.2f,1.3f) die 2.Funktion auf.

3 – Die Programmiersprache C/C++

Überladen von Funktionen

So kann auch der Konstruktor und der Kopierkonstruktor überladen werden.

```
class myClass
{
 myClass(); // constructor
 myClass(int a); // overloaded constructor
}
```

Der Kopierkonstruktor (Teil 2)

Der Kopierkonstruktor wird aufgerufen, sobald eine Instanz einer Klasse einer anderen Instanz der selben Klasse zugewiesen wird.

Das kann entweder durch den = Operator geschehen oder bei der Instanziierung der Klasse:

```
cSchein berta;

cSchein heinrich(berta); // Kopiert berta nach heinrich

cSchwein heinrich = berta; // nutzt auch den Kopierkonstruktor
```

Oder

```
heinrich = berta; // Kopiert berta nach heinrich mittels
                  // der Operator=() Funktion
```

3 – Die Programmiersprache C/C++

- Wurde kein Kopierkonstruktor implementiert, so wird der Standard Kopierkonstruktor verwendet. Dieser kopiert einfach die Inhalte aller Variablen der Quelle in die entsprechenden Variablen des Zielobjektes.
- Mit Vorsicht zu genießen ist die Kopie von Zeigern.
- Im obigen Beispiel wird Schwein `heinrich` als eine Kopie von `Berta` erzeugt. Falls `Berta` Zeiger auf eigene Variablen beinhaltet, so werden diese Zeiger auch kopiert.
- Sie zeigen dann in der Kopie (in diesem Fall Zeiger von `Heinrich`) immer noch auf Variablen von `Berta`.

3 – Die Programmiersprache C/C++

Eine Implementation eines Kopierkonstruktors kann beispielsweise folgendermassen aussehen:

```
class Date
{
 public:

 int tag,monat,jahr;

 Date();                  // Konstruktor
 ~Date();                 // Destruktor
 Date(const Date& d){     // Kopierkonstruktor
    tag   = d.tag;
    monat = d.monat;
    jahr  = d.jahr;
 }
}
```

3 – Die Programmiersprache C/C++

NameMangeling

- Der Compiler führt ein NameMangling durch, d.h. die Funktion wird mit den aufgerufenen Datentypen in einem String codiert...z.B. `addiere@@@int@@@int`.
- Da die Syntax des NameMangling von Compiler zu Compiler unterschiedlich ist, kann das zu Problemen bei Bibliotheken führen.
- Nutzt man `extern"c"` so können Funktionen nicht überladen werden, aber ein NameMangling findet auch nicht statt.

3 – Die Programmiersprache C/C++

extern ”c“

```
extern "c"
{
 int addiere(int a, int b);
}
```

- So ist es möglich auch C oder C++ Funktionen zu nutzen, die mit einem anderen Compiler geschrieben wurden.

Überladen von Operatoren

Nicht nur Funktionen können überladen werden, sondern auch Operatoren.

Die Syntax einer Funktion, die einen Operator überlädt lautet wie folgt:

```
Rückgabewert operator+ (const datentyp leftHandSide, const datentyp
   rightHandSide)
```

3 – Die Programmiersprache C/C++

```
class vector{
  enum {SIZE=3};
  double v[SIZE];
public:
  vector(double x,double y,double z)
  {
  v[0]=x;   v[1]=y;    v[2]=z;
  }
  // --------------- Operator überladen ------------ //
  vector operator+(const vector& v2){
    vector t(0,0,0);
    for (int i=0;i<SIZE;i++){
       t.v[i]=v[i]+v2.v[i];
    }
    return t;
  }
};
// ---------------------- main ---------------------//
int main(void) {
 vector v1(1,2,3), v2(4,6,5), v3(0,0,0);
 v3 = v1+v2; // funktioniert nur weil der Operator+ überladen ist
}
```

Überladen von Operatoren: Bedingungen

Nicht überladen werden können:

```
?:      Bedingter Ausdruck
sizeof  Sizeof Operator
.       Punktoperator
::      Scopeoperator
.*      Komponentenzeiger
```

3 – Die Programmiersprache C/C++

Überladen von Operatoren: Bedingungen

- Mindestens ein Operand muss abstrakter (z.B. selbst definierter) Datentyp sein.
- Die Regeln der C Operatoren sollten eingehalten werden (z.B.Auswertereihenfolge).
- Die Anzahl der Parameter ist vorgegeben.
- Überlädt man beipielsweise den = Operator, so sollte man darauf achten, dass die operator= Methode auch einen Rückgabewert hat (in diesem Fall *this), denn sonst ist obj1=obj2 möglich, nicht aber obj1=obj2=obj3.

3 – Die Programmiersprache C/C++

Überladen von Operatoren: Bedingungen

- Für den Fall, dass der lefthand Operator nicht vom Typ der Klasse ist, gibt es keine Möglichkeit innerhalb der Klasse eine Operatorüberladung zu implementieren.
- Für einen solchen Fall kann man die Operatorüberladung auch ausserhalb der Klasse schreiben.
- Die Klasse, von dessen Typ der erste operator ist, wird dann normalerweise als friend deklariert, und kann somit auch auf private: Elemente zugreifen.
- Auch hier gilt, dass man für call by value und für den Rückgabewert einen Kopierkonstruktor braucht.

3 – Die Programmiersprache C/C++

Beispiel: Skalar mal Vektor, Operatorüberladung ausserhalb der Klasse

```
class doubleVector{
public:
  enum {SIZE=3};
  double v[SIZE];
  doubleVector(double x,double y,double z){
  v[0]=x;   v[1]=y;    v[2]=z;
  }
};// --- overloading the operator outside of the class ---
doubleVector operator*(int k, const doubleVector& vec){
  doubleVector t(0.0,0.0,0.0);
  for (int i=0;i<doubleVector::SIZE;i++){
     t.v[i] = k*vec.v[i];
  }
  return t;
}
int main(void) {
 doubleVector v(1.1, 2.2, 3.3);
 doubleVector v2 = 5*v; // the function operator*() is called
}
```

3 – Die Programmiersprache C/C++

Man kann z.B. auch einen Typecast überladen/ einbauen:

```
class Bruch
{
 int m_zaehler,m_nenner;
public:
 Bruch(int zaehler, int nenner) : m_zaehler(zaehler),
                                  m_nenner(nenner)
 {}// Konstruktor mit initialisierung von m_nenner und m_zaehler

 operator float()   // typecast implementieren!!!
 {
   return ((float)(m_zaehler)/(float)(m_nenner));
 }
 operator double()   // typecast implementieren!!!
 {
   return ((double)(m_zaehler)/(double)(m_nenner));
 }
};
```

3 – Die Programmiersprache C/C++

Zusätzlich könnten die Operatoren überladen werden, die das Bruchrechnen übernehmen:

```
Bruch operator*(Bruch b)
 {
   Bruch ergebnis;
   ergebnis.m_zaehler=this->m_zaehler*b.m_zaehler;
   ergebnis.m_nenner=this->m_nenner*b.m_nenner;
   return ergebnis;
 }
 Bruch operator+(Bruch b)
 {
   Bruch ergebnis;
   ergebnis.m_zaehler=this->m_zaehler*b.m_nenner +
                      b.m_zaehler*this->m_nenner;
   ergebnis.m_nenner=this->m_nenner*b.m_nenner;
   return ergebnis;
 }
```

Vererbung

3 – Die Programmiersprache C/C++

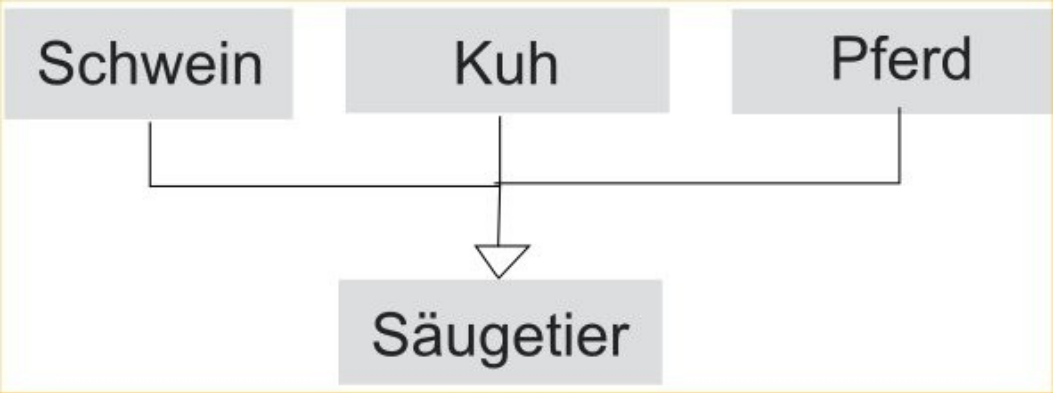
Vererbung in C++

- Grundlagen wie in Java
- Überschreibung von Funktionen
- Aufruf der Konstruktoren
- Friend
- Virtuelle Funktionen und Polymorphismus
- Zugriffsrechte
- Nachtrag Kopierkonstruktor (Teil 3)

3 – Die Programmiersprache C/C++

Grundlagen wie in Java

Besonders effizient programmieren kann man mit Hilfe von Abstraktion. Wenn zum Beispiel "Kuh", "Schwein" und "Pferd" = "Saeugetiere" sind könnte man zunächst die Klasse Säugetiere implementieren und davon "Kuh", "Schwein" und "Pferd" ableiten.



3 – Die Programmiersprache C/C++

- In anderen Worten: Man kann die Attribute und Methoden, die für alle Säugetiere gelten an die einzelnen Saeugetiere vererben.
- Einzelne Methoden, wie z.B. das "grunzen()" des Schweines, könnten dann für das vom Säugetier abgeleitete Schein extra implementiert werden
- Konstruktoren und Destruktoren werden nicht vererbt

3 – Die Programmiersprache C/C++

```
class CSaeugetier // Basisklasse
{
 friend Class CLieberGott; // Freunde können auf alle Attribute
          // der Basisklasse zugreifen, auch auf die privaten!
 public:
 void fressen();
 void wachsen();
 protected:       // auf protected: Attribute kann von der
                  // abgeleiteten Klasse zugegriffen werden
 int groesse;
 int gewicht;
 private:         // auf private: Attribute kann von der abgeleiteten
                  // Klasse nicht zugegriffen werden
}
class Kuh : public CSaeugetier  // von CSaeugetier  abgeleitete Klasse
{
 public:
 void muhen();
}
```

3 – Die Programmiersprache C/C++

Und noch das Schwein und das Pferd…

```
class Schwein : public CSaeugetier    // von CSaeugetier
                                      // abgeleitete Klasse
{
 public:
 void grunzen();
}

class Pferd : public CSaeugetier     // von Csaeugetier
                                     // abgeleitete Klasse
{
 public:
 void wiehern();
}
```

3 – Die Programmiersprache C/C++

Überschreiben von Funtionen

- Zusätzlich kann man in der abgeleiteten Klasse eine Funktion der Basisklasse überschreiben.
- Das heißt man kann eine Funktion mit gleichem Namen und gleichen Parametern implementieren.
- Bei einer Instanz der abgeleiteten Klasse wird dann diese Funktion statt der überschriebenen Basisklassenfunktion aufgerufen.

Aufruf der Konstruktoren

- Bei Instanziierung eines Objektes einer abgeleiteten Klasse wird zuerst der parameterlose Konstruktor der Basisklasse aufgerufen, dann der Konstruktor der abgeleiteten Klasse.
- Bei mehrfach vererbten Klassen wird zuerst der parameterlose Konstruktor der Basisklasse, dann der der abgeleiteten Klasse, dann der der davon abgeleiteten Klasse... bis zum Aufruf des Konstruktors der aufgerufenen Klasse (der auch Parameter beinhalten kann).

3 – Die Programmiersprache C/C++

Friend

- Im obigen Beispiel wird die Klasse "CLieberGott" als "friend" deklariert.
- Dadurch kann jede Methode der Klasse "CLieberGott" alle Attribute der Klasse "CSaeugetier" ändern und lesen, auch wenn sie **private** sind!

  Anmerkung: Mit **friend:** definierte Klassen werden nicht als friend mit vererbt.

3 – Die Programmiersprache C/C++

Virtuelle Funktionen und Polymorphismus

- Arbeitet man mit Zeigern vom Typ Basisklasse "CSaeugetier" auf eines seiner abgeleiteten Klasseninstanzen z.B. "Schwein", so kennt der Compiler nicht die Funktion "grunzen()", da es keine Funktion der Basisklasse ist.
- Wenn man die Methode "grunzen()" als virtuell in der Klasse CSaeugetier deklariert und imlementiert, so kennt der Compiler auch diese Funktion. (z.B. virtual void grunzen();)
- Schreibt man in der Basisklasse hinter einer virtual void Funktion()=0, so muss diese Funktion in jeder abgeleiteten Klasse implementiert werden! Man spricht dann von rein virtuellen Funktionen.
- Eine Klasse, die ein oder mehrere rein virtuelle Methoden enthält kann nicht instanziiert werden

3 – Die Programmiersprache C/C++

- **Anmerkung:** Mit virtuellen Methoden werden Virtuelle Methodentabellen angelegt und für jede Instanz der Klasse Zeiger auf die Virtuelle Methodentabelle. So erkauft man sich die Flexibilität der virtuellen Methoden besonders bei vielen Instanzen mit Speicherplatz und Laufzeit.

- **Anmerkung:** Klassen, die mindestens eine virtuelle Funktion mit "=0" implementieren heißen abstrakte Klassen!

3 – Die Programmiersprache C/C++

```
class CSaeugetier
{

 private:
 int groesse;
 int gewicht;

 public:
 void fressen();
 void wachsen();
 virtual void grunzen();
 virtual void wiehern();
 virtual void muhen();

}
```

3 – Die Programmiersprache C/C++

Zugriffsrechte

public: Auf public Member kann von überall drauf zugegriffen werden!

protected: Auf protected Member kann von der Klasse, in der sie deklariert sind und deren abgeleiteten Klassen zugegriffen werden!

private: Auf private Member kann nur von der Klasse zugegriffen werden, in der sie deklariert sind!

Hinter der abgeleiteten Klasse steht mit : getrennt die Art der Ableitung. Die Art kann private, protected oder public sein. Damit kann man die geerbten Funktionen und Variablen noch mal extra schützen. Mit public werden die Zugriffsrechte der Basis beibehalten, mit protected sind sie mindestens protected, mit private sind sie alle private!

3 – Die Programmiersprache C/C++

Ableitung	Privileg der Basisklasse	Privileg der abgeleiteten Klasse
:public	public:	public
	protected:	protected
	private:	Kein Zugriff
:protected	public:	protected
	protected:	protected
	private:	Kein Zugriff
:private	public:	private
	protected:	private
	private:	Kein Zugriff

Nachtrag Kopierkonstruktor (Teil 3)

Wenn man bei der Instanziierung einer Basisklasse die von ihr abgeleiteten Klassen Instanz zuweist, so hat der Compiler alle Daten zur Verfügung um die Basisklasse zu füllen. Andersrum würde der Compiler eine Fehlermeldung ausgeben, da bei der abgeleiteten Klasse mehr Attribute sein könnten, die dann nicht aus der Basisklasse kopiert werden können.

```
// so geht's...

CAbgeleiteteKlasse abgeleitet;

CBasisKlasse basis=abgeleitet; // ruft den Kopierkonstruktor und
   kopiert alle Attribute die die Basisklasse benötigt
```

Templates

3 – Die Programmiersprache C/C++

Templates

- Grundlagen
- Funktionstemplates
- Klassentemplates

3 – Die Programmiersprache C/C++

Grundlagen

- Manchmal wünscht man sich Funktionen, die mit einem beliebigen Datentyp arbeiten. Diese Möglichkeit bieten Templates (Schablonen)
- Es gibt Funktionstemlates und Klassentemplates

3 – Die Programmiersprache C/C++

Funktionstemplates

Hier ein Beispiel einer Funktion, die das Maximum zweier Instanzen zurückgibt. Wird die Templatefunktion mit verschiedenen Datentypen aufgerufen, so erzeugt der Compiler die Funktion für jeden der genutzten Datentypen neu.

```
template <typename T>
T max(T parameter1, T parameter2)
{
 if (parameter1>parameter2){
    return parameter1;
 }
 else{
    return parameter2;
 }
}
```

Werden mehr als ein Templatedatentyp verwendet, so kann man diese mit Komma innerhalb der spitzen Klammern abtrennen.

```
template <typename T, typename U>
```

Funktionstemplates

Der Aufruf dieser Funktionstemplates kann so aussehen wie bisher:

```
int a = max(5,7);
```

In dem Fall wird 5 und 7 als Zahlen des Typs Integer interpretiert.

Man kann den Datentyp jedoch auch explizit angeben:

```
int a = max <int> (5,7);
```

Klassentemplates

- Bei Klassentemplates können Attribute der Klasse und Funktionsparameter und Rückgabewerte der Methoden vom Template-Typ sein.
- Templateklassen können vererbt werden und können virtuelle Funktionen enthalten!

3 – Die Programmiersprache C/C++

```
template <class T>
class CMathVector
{
public:
  enum{MAX=3}
  T mathVector[MAX];

  T mittelwert(void)
  {
    T sum=0;
    for (int i=0;i<MAX;i++){
        sum+=mathVector[i];
    }
    return sum/MAX;
  }
}
```

```
// im cpp File Instantiierung:

int main(void)
{
CMathvector <double> dblVekt;
double mittel;

dblVekt.mathVector[0] = 1.234;
dblVekt.mathVector[1] = 0.5643;
dblVekt.mathVector[2] = 0.543;
mittel = dblVekt.mittelwert();
return 0;
}
```

3 – Die Programmiersprache C/C++

```
// alternative Implementierung
// ausserhalb der Klassendefinition
template <class T>
T CMathVector<T>::mittelwert(void)
{
  T sum;
  for (int i=0;i<MAX;i++){
      sum+=mathVector[i];
  }
  return sum/MAX;
}
```

Streams

3 – Die Programmiersprache C/C++

Streams

- Grundlagen
- Datei schreiben
- Datei lesen

3 – Die Programmiersprache C/C++

Grundagen

Bindet man <iostream> ein, so kann man Zeichen von der Tastatur einlesen und auf dem Bildschirm ausgeben. Dazu nutzen wir den << und den >> Operator:

```
#include <iostream>
using namespace std;
int main(void)
{
 int zahl;
 cout << "geben Sie eine Zahl ein:"; // Der Text in Anführungszeichen wird
   auf
                                     // dem Bildschirm ausgegeben
 cin >> zahl;      // hier wird eine Zahl von der Tastatur eingelesen,
                   // die mit der Eingabetaste bestätigt werden muss
 cout << "Sie haben folgende Zahl eingegeben:" << zahl << endl;
                   // endl ist Zeilenumbruch und forciert das ausgeben
                   // bzw. leeren des Ausgabebuffers
 return 0;
}
```

3 – Die Programmiersprache C/C++

Datei schreiben

In C++ gibt es eine Stream Erweiterung, die für das File-Handling zuständig ist:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
 ofstream outFile; // für Dateien, die geschrieben werden
 outFile.open("test.txt",ios::out); // öffnen der Datei zum schreiben
 if (!outFile){
    cout << "Ausgabedatei kann nicht geöffnet werden!\n\n";
 }
 else{
   outFile << "Dies ist ein BeispielTEXT" << flush;
 }
 outFile.close();
}
```

3 – Die Programmiersprache C/C++

Datei lesen

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
 string inp;
 ifstream inFile;  // für Dateien, die geöffnet werden

 inFile.open("test.txt",ios::in); // öffnen der Datei zum lesen
 if (!inFile){
    cout << "Eingabedatei kann nicht geöffnet werden!\n\n";
 }
 else
 {
    while(inFile.getline(inp,1)){
       cout << inp << flush; // flush bedeutet Buffer lehren
    }
 }
 inFile.close();
}
```

Strings

3 – Die Programmiersprache C/C++

Strings

- Vorteile gegenüber C-Strings
- Beispiel

3 – Die Programmiersprache C/C++

Grundlagen

Der Vorteil von String zum ursprünglichen C-Sting oder array von char ist:

- \0 kann im string vorkommen ohne das es das Ende des Strings makiert!
- es gibt zahlreiche Möglichkeiten, mit denen man Strings manipulieren kann z.B. kann man strings addieren
- Man kann den String mit rezise vergrößern
- alle Methoden sind sicher!!!

Um die Stringklasse benutzen zu können muss sie zunächst includiert werden.

3 – Die Programmiersprache C/C++

Beispiel

```
#include <string>
using namespace std;
int main(void)
{
 string Nachname("Walter");
 string Vorname("Dieter");
 string Name;
 Name=Vorname+" "+Nachname;
 if (Vorname=="Otto"){
    cout << Name << endl;
 }
 if (Vorname.find("ie")>0){
    cout << "Die Buchstaben ie befinden sich an Stelle "
         << Vorname.find("ie") << "im Vornamen" << Vorname
         << "." << endl;
 }
 return 0;
}
```

Casting und Fehlerbehandlung

3 – Die Programmiersprache C/C++

Casting und Fehlerbehandlung

- Casting
- Fehlerbehandlung

3 – Die Programmiersprache C/C++

Casting

Es gibt in C++ weitere Cast Operatoren:

```
static_cast<xyz*>(p)              // prüft Verwandschaftsbeziehung

reinterpret_cast<char*>(&org);    // hier kann zu etwas komplett anderem gecastet
                                  // werden.

const_cast<char*>(q)='B';         // selbst konstanten können überschrieben werden!

dynamic_cast<CKlasse*>(pA)        // Prüft zur Laufzeit ob cast korrekt und gibt
                                  // sonst NULL zurück
```

Beim dynamic_cast wird eine vtable (virtuelle Methodentabelle) in das Programm eingebunden. Damit ist der Dynamic_cast für embedded Systeme ungeeignet.

3 – Die Programmiersprache C/C++

Fehlerbehandlung

Es gibt zweierlei Arten der Fehlerbehandlung in C++:

1. Mittels assertion

```
#include <assert.h>
...
assert(true);  // führt keine Fehlerbehandlung aus
assert(false); // führt eine Fehlerbehandlung aus
```

3 – Die Programmiersprache C/C++

2. Mittels Exceptions try-catch:

```
#include <iostream>
int i=0,j=0;
try
{
   i=i/j;
}
catch(...) // statt ... kann hier auch ein Ausnahmetyp stehen
{  // springt mittels einer exception hier rein um den Fehler zu behandeln!
   std::cout << "Division durch Null!"<<flush;
}
```

Der neue C++ Standard

3 – Die Programmiersprache C/C++

Der neue C++ Standard

- Neue Schlüsselworte
- Die range basierte For Schleife
- Funktionale Programmierung (Lambda Funktionen)
- generische Programmierung (zusätzliche Möglichkeiten)
- Multithreading
- Atomare Datentypen
- Asynchrone Aufgaben
- Reguläre Ausdrücke

3 – Die Programmiersprache C/C++

Neue Schlüsselworte

Um die Programmierung zu vereinfachen wurden neue Schlüsselworte eingeführt:

Beim neuen Datentyp **auto** wird der eigendliche Datentyp erst bei der Initialisierung der Variablen definiert. Das macht besonders bei Templates Sinn.

```
auto var = x + y; // sind x und y Objekte einer Klasse,
                  // so ist auch var Objekt dieser Klasse
```

Um den Datentyp zu ermitteln liefert decltype den Datentyp zurück.:

```
typedef decltype(var) myDataType;
myDataType var2; // var2 hat den selben Datentyp wie var
```

3 – Die Programmiersprache C/C++

Die range basierte For- Schleife

Um sich wenig um die Größe von Feldern kümmern zu müssen gibt es nun die range basierte for Schleife:

```
std::vector<int> vec({ 1, 2, 3, 4, 5, 6 });
for (const auto x : vec){
     std::cout << x << std::endl;
}
```

Die Schleife iteriert über alle Elemente vec.

3 – Die Programmiersprache C/C++

Funktionale Programmierung

Ab C++11 sind auch die sogenannten Lambda-Expressions der funktionalen Programmierung möglich.

```
std::vector<int> v = {15, -2, 4, -1};
std::sort(v.begin(), v.end(), [](int a, int b) {
      return abs(a)<abs(b); });
```

Innerhalb der Funktionen kann man zwar Membervariablen auslesen und sich Hilfsvariablen deklarieren und benutzen. Es können aber keine Membervariablen beschrieben werden.
In Eckigen Klammern [] wird die Bindung zu den lokalen Variablen angegeben, in runden Klammern () die Parameter

3 – Die Programmiersprache C/C++

Generische Programmierung

Als generischer Programmiererweiterung werden ab C++11 folgende eingeführt, hier aber nicht näher beschrieben:

- Templates, die beliebig viele Parameter annehmen
- Zusicherungen, die zur Compilezeit ausgewertet werden
- Konstanten, die zur Compilezeit ausgewertet werden
- Aliase Templates, um einfache Namen für teilweise gebundene Templates zu definieren

Multithreading

Es gibt im Wesendlichen 3 Varianten einen Thread zu erzeugen:

1. mit Hilfe einer Funktion
2. mit Hilfe eines Objektes
3. mit Hilfe einer Lambda Expression

3 – Die Programmiersprache C/C++

Multithreading (mit Funktion)

```
#include <iostream>
#include <thread>
void helloFunction() {
  std::cout << "C++11 Thread with function"<< std::endl;
}
int main() {
 std::thread t1(helloFunction);
 t1.join();
}
```

3 – Die Programmiersprache C/C++

Multithreading (mit Objekt)

```
#include <iostream>
#include <thread>
class HelloFunctionObject {
  public:  void operator()() const {
        std::cout << "C++11 Thread with Object" << std::endl;
  }
}
int main() {
 HelloFunctionObject helloFunctionObject;
 std::thread t2(helloFunctionObject);
 t2.join();
}
```

3 – Die Programmiersprache C/C++

Multithreading (mit Lambda Ausdruck)

```
#include <iostream>

#include <thread>

int main() {

 std::thread t3([]{std::cout << "C++11 Thread with lambda
   function" << std::endl;});

 t3.join();

}
```

3 – Die Programmiersprache C/C++

Atomare Datentypen

- Atomare Datentypen sind Datentypen, die nicht aus anderen Datentypen zusammengesetzt sind.
- Auf diese Datentypen können atomare Operationen ausgeführt werden.
- Atomare Operationen sind Operationen, die nicht unterbrochen werden können.
- Es gibt bereits viele atomic Datentypen, die einen äquivalenten build in Typ haben: z.B. atomic_bool, atomic_char, atomic_int etc.
- Mit Hilfe des `std::atomic-Klassen-Template` ist es möglich eigene atomare Typen zu implementieren.

3 – Die Programmiersprache C/C++

Asynchrone Aufgaben

- Sie sind unter dem Namen Futures bekannt
- Der Compiler kümmert sich selbst um die Verwaltung und prüft ob es z.B. sinnvoll ist einen neuen Thread zu erzeugen
- std::async liefert ein std::future Objekt zurück, von welchem man zu einem späteren Zeitpunkt das Ergebnis abrufen kann
- Sollte das Ergebnis noch nicht vorliegen blockiert der Thread

3 – Die Programmiersprache C/C++

Asynchrone Aufgaben(Beispiel)

```
#include <future>

int product(int a, int b) {

 return a*b;

}

int main() {

 int a = 20;

 int b = 10;

 std::future<int> futureSum = std::async([=](){return a+b;});

 auto futureProduct = std::async(&product,a, b);

 std::cout << futureSum.get() << std::endl;

 std::cout << futureProduct.get() << std::endl;

}
```

Reguläre Ausdrücke

Mit Hilfe der regukären Ausdrücke können Zeichenfolgen durchsucht und ersetzt werden.

Ab C++11 werden reguläre Ausdrücke der folgenden Grammatiken unterstützt:

- ECMAScript
- basic
- extended
- awk
- grep
- egrep

4 – Debugger und Profililer

4 – Debugger und Profiler

Debugger

- Ein Debugger ist ein Tool zum Finden von Fehlern (Bugs).
- Typischerweise ist es möglich sogenannte Breakpoints zu setzen (Markierungen für eine Codezeile), an denen man den Programmablauf stoppen und später wieder fortsetzen kann
- Wenn der Programmablauf gestoppt wird, kann man z.B. Variablen auslesen oder ändern
- Manche Debugger lassen auch Breakpoints auf Variablen zu. Dann stoppt der Programmablauf, sobald sich die Variable ändert

4 – Debugger und Profiler

Profiler

Vorgehensweise

- Profiler starten und ein typisches Anwendungsszenarium durchspielen. Dies kann auch automatisiert werden, da man unter Umständen nicht so viele Eingaben machen kann wie erforderlich
- Man bekommt meist für eine Programmzeile die Rechenzeit in % oder Zyklen

4 – Debugger und Profiler

Profiler

Die Zeilen mit der meisten Rechenzeit nimmt man sich vor:

- Aus Schleifen zieht man unnötige Mehrfachberechnungen
- Funktionsaufrufe kosten Zeit und können z.B. als inline Funktion oder Makro (#define) verarbeitet werden
- Manchmal sind Tabellen schneller als mathematische Funktionen oder es gibt eine vereinfachte gröbere Berechnung der mathematischen Funktion
- Passt das Memory in Schleifen in den Cache? In den Hauptspeicher?
- Gibt es einen schnelleren Algorithmus (z.B. mit log(n) Aufwand)?
- Sind die benötigten Datentypen auf der Maschine schnell?
- Muss die Routine in dem Zeitraster aufgerufen werden?
- Muss die Berechnung in dem Raster durchgeführt werden?

5 – Digitale Signalverarbeitung

Kapitel 5 – Digitale Signalverarbeitung

- Einführung
- Faltung, was ist das?
- Anwendung in der Bildverarbeitung (Laplace Filter)
- Anwendung in der Audioverarbeitung (FIR Filter)
- Adaptive Filter
- Fourier Transformation

.

5 – Digitale Signalverarbeitung

Einführung

- Unter Digitalen Signalen versteht man Signale, die auf diskrete Meßwerte abgebildet sind
- Um Signale zu digitalisieren verwendet man üblicherweise Analog-Digitalwandler (AD-Wandler).

Die einfachste Umsetzung ist ein Zähler, der zählt, wie lange z.B. ein Kondensator, der vom Signal in einem kurzen Zeitintervall aufgeladen wurde, braucht, um sich wieder über einen Widerstand zu entladen. Der Zählerwert entspricht dann in etwa der Amplitude des analogen Signals im Mittel über das Zeitintervall.

5 – Digitale Signalverarbeitung

- Möchte man ein digitales Signal wieder in ein analoges umwandeln, so dient dazu ein Digital-Analog Wandler (DA-Wandler)

Die einfachste Umsetzung ist ein tiefpass gefiltertes pulslängen moduliertes Signal.

Je länger das Signal auf „High“ steht, desto größer ist der Wert der Amplitude.

5 – Digitale Signalverarbeitung

Was ist eine Faltung?

- Eine Faltung ist eine mathematische Operation bei der der Stern * der Faltungsoperator ist!
- Die Faltung ist folgendermaßen definiert:

$$(f * g)(t) = \int f(\tau)\, g(t-\tau)\, d\tau$$

5 – Digitale Signalverarbeitung

Beispiel:

$$(f*g)(t)=\int f(\tau)\,g(t-\tau)\,d\tau$$

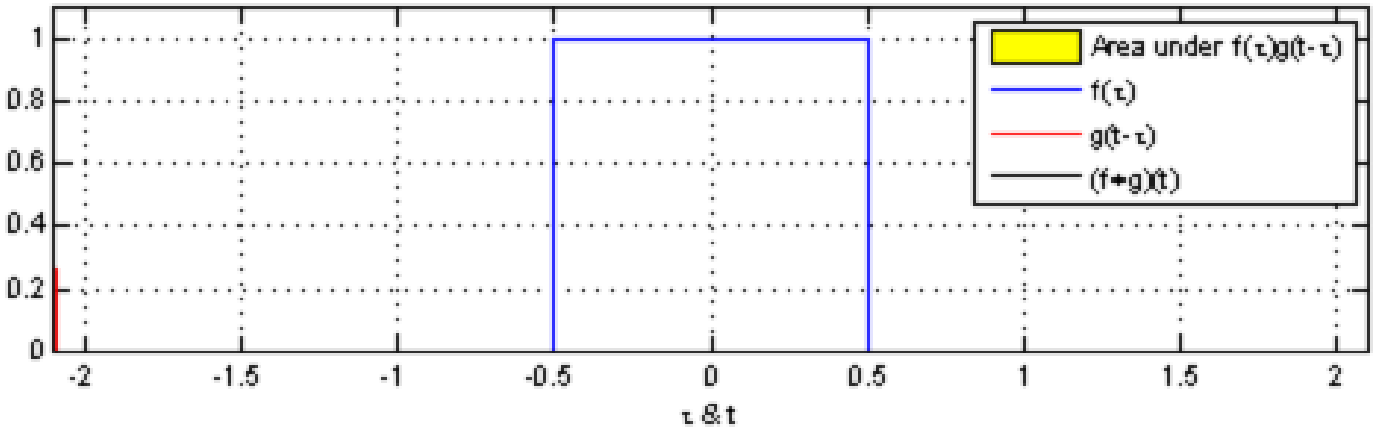


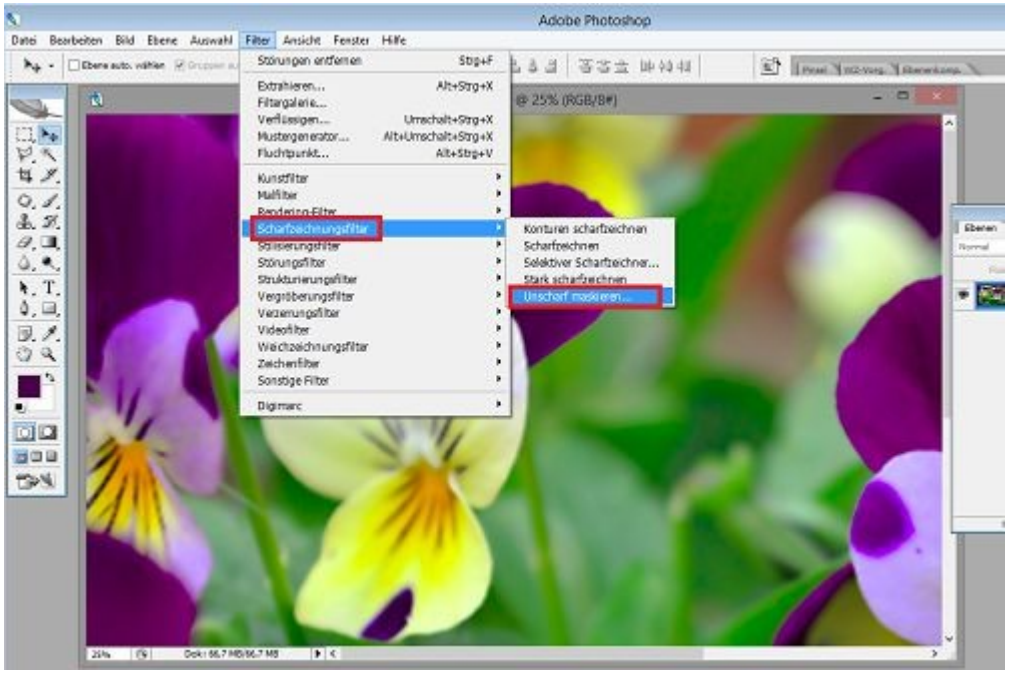
Abb.1) Diese Abbildung zeigt den Vorgang einer Faltung zweier

Rechteckfunktionen. Das Integral an der Stelle t wird hier durch die gelbe Fläche dargestellt. Quelle: https://de.wikipedia.org/wiki/Faltung_(Mathematik)

5 – Digitale Signalverarbeitung

Bildverarbeitung:

- Wenn man die Faltung mittels eines Bildes und einer 3x3 Matrix berechnet, so kann man damit z.B. Fotos unschärfer machen

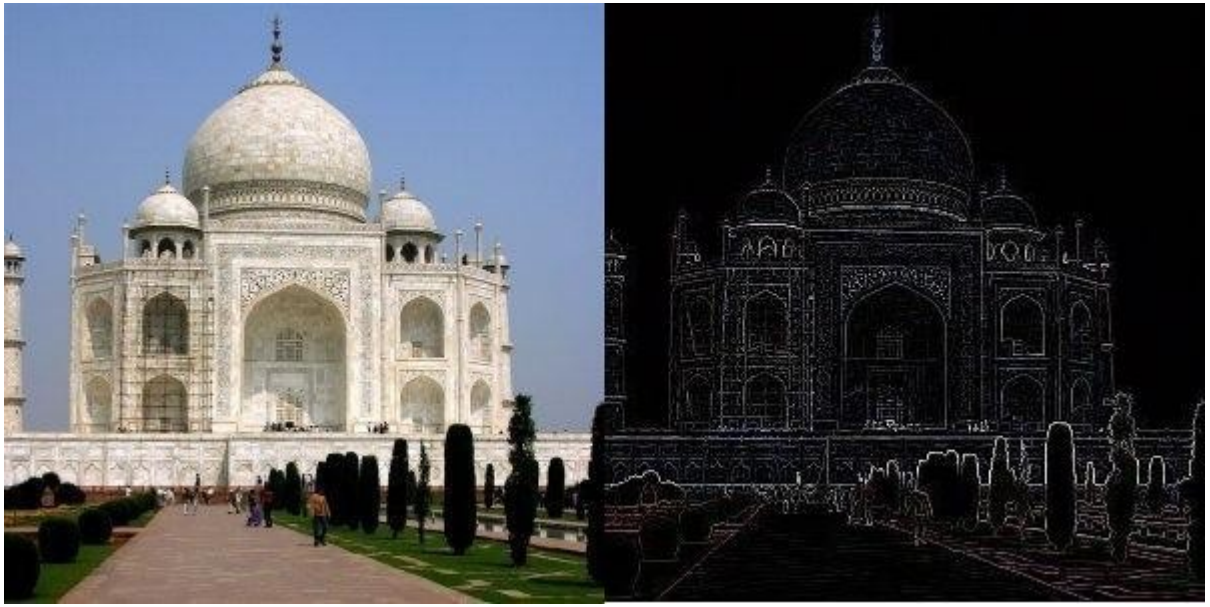


$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

5 – Digitale Signalverarbeitung

Bildverarbeitung:

- Oder Kanten extrahieren mittels Laplace Filter!

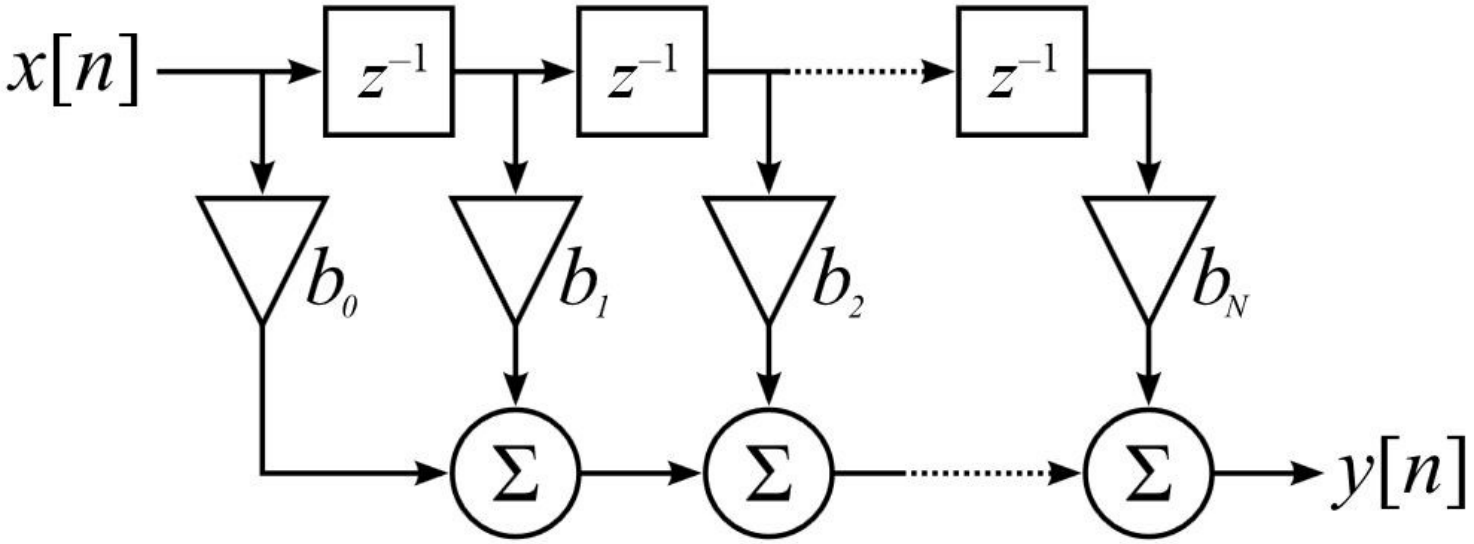


$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

5 – Digitale Signalverarbeitung

Audioverarbeitung

- Die Faltung eines eindimensionalen diskretisierten Audiosignals wird in Form eines FIR-Filters umgesetzt!



5 – Digitale Signalverarbeitung

Audioverarbeitung

- Mit festen Koeffizienten kann man so Filter mit beliebiger Übertragungsfunktion konstruieren.
- Aber was, wenn man z.B. einen anhaltenden Hintergrundton wegfiltern möchte, aber dessen Frequenz nicht kennt?
- Dann braucht man ein adaptives Filter

5 – Digitale Signalverarbeitung

Adaptive Filter

- Gegeben sei ein FIR Filter:

$$y[t] = \sum_i b_i x[t-i]$$

- Hat man eine Zielausgabe Y_{target}, die der Filter lernen soll, so lautet der Fehler:

$$E^2 = (y_{target} - y[t])^2$$

- Um den Gradienten zu berechnen leitet man den Fehler nach den Koeffizienten ab:

$$\Delta b_i = -\eta \frac{\partial E^2}{\partial b_i} = -\eta 2 (y_{target} - y[t]) x[t-i]$$

η ist die Lernrate

5 – Digitale Signalverarbeitung

Adaptive Filter

- Definiert man nun als y_{target} das nachfolgende Ausgabesample, so gibt es eine Vorhersage.
- Diese Vorhersage kann man z.B. zur Kompression verwenden, denn die Differenz zwischen Vorhersage und realem Sample läßt sich mit weniger Bits speichern.

- Wenn man nun die Lernrate sehr klein wählt, dann werden hauptsächlich die lang anhaltenden Signale mit grosser Amplitude gelernt
- Zieht man nun die gelernte Vorhersage des Hintergrundsignals vom Ursprünglichen Signal ab, so filtert man dauerhafte Nebengeräusche raus. (Patente 1,2)

1) DE 102005039621 Verfahren und Vorrichtung zur adaptiven Reduktion von Rausch- und Hintergrundsignalen in einem sprachverarbeitendem System

2) US 2007/43559; US 2011/22382 ADAPTIVE REDUCTION OF NOISE SIGNALS IN A SPEECH-PROCESSING SYSTEM

5 – Digitale Signalverarbeitung

Adaptive Filter

```
length = readAudioSamples(inArray[])

for t=4 to length

    y = w[1]*inArray[t-1] +
        w[2]*inArray[t-2] +
        w[3]*inArray[t-3] +
        w[4]*inArray[t-4];

    error = inArray[t].f - y;

    w[1] += learnRate * error * inArray[t-1];

    w[2] += learnRate * error * inArray[t-2];

    w[3] += learnRate * error * inArray[t-3];

    w[4] += learnRate * error * inArray[t-4];

    outArray[t] = inArray[t] - y;

writeAutioSamples(outArray[])
```

5 – Digitale Signalverarbeitung

Diskrete Fourier Transformation

- Die diskrete Fourier Transformation (DFT) ist eine Faltung:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi n}{N}k} \quad k = 0, 1, 2, \ldots, (N-1)$$

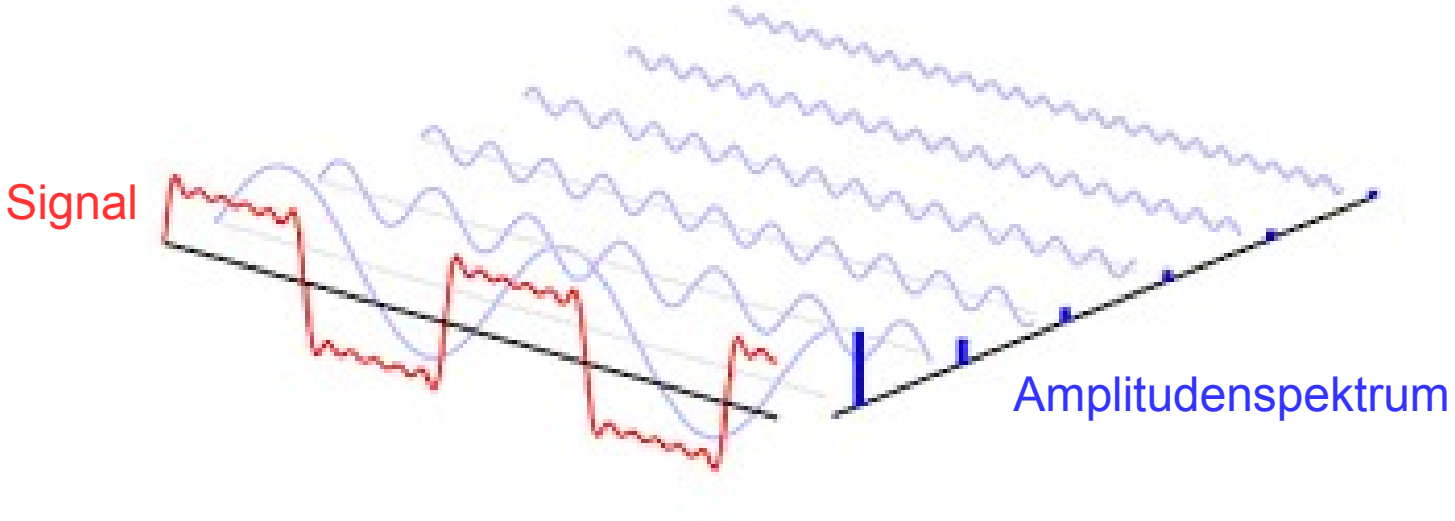
Beispiel für $N = 4$ (also für 4 Frequenzen aus 4 Samples berechnet) :

$$X[k] = \sum_{n=0}^{3} \underbrace{x[n]\{\cos(2\pi nk/4)}_{\text{Realteil}} - j\underbrace{\sin(2\pi nk/4)\}}_{\text{Imaginärteil}}$$

- Diese Gleichung muss nun für k = 0..4 nerechnet werden um alle 4 komplexen Zahlen zu bekommen. Das Amplitudenspektrum bekommt man nun, indem man x[k] quadriert!

5 – Digitale Signalverarbeitung

Diskrete Fourier Transformation



6 - Prozessoren

Kapitel 6 - Prozessoren

- Was bedeutet CPU,GPU,µC,DSP?
- Grundlegender Aufbau CPU - System
- Grundlegender Aufbau GPU - System
- µC und DSP basierte Systeme

.

6 – Prozessoren

Was bedeutet CPU,GPU,µC,DSP?

- CPU = Central Processing Unit, eine Zentrale Steuereinheit, wie sie heute in jedem PC zu finden ist. Es werden immer mehr Einheiten in die CPU integriert, so dass neben Speicher (Cache), auch Fließkommaeinheiten oder Grafikkarten bereits auf dem Chip integriert sind
- GPU = Graphics Processing Unit, Prozessoren, die auf Grafikkarten zur hochparallelen Verarbeitung der vor allem in modernen Spielen benötigten imensen Anzahl von Dreiecken. Mittlerweile ist die Rechenkapazität gemessen in Flops der GPUs um ein vielfaches höher als die der CPU's und sie lassen sich auch in Hochsprachen wie C/C++ oder Fortran programmieren

6 – Prozessoren

Was bedeutet CPU,GPU,µC,DSP?

- µC =Mikrokontroller, sicher die häufigste Art von Prozessoren, die vor allem für Haushaltsgeräte, Autos und Unterhaltungselektronik eingesetzt werden. Sie haben oft einen integrierten Flash Speicher für Programme, integrierte Wandler (AD/DA) und sind mit wenig Beschaltungsaufwand lauffähig.
- DSPs = Digitale Signal Prozessoren, Rechenknechte, meist mit Fließkomma- oder Festkomma Einheiten ausgestattet. (Festkomma z.B.: -1.0 bis 0.9999) Oft parallel zu Mikrokontrollern z.B. für Audio oder Videoverarbeitung.

Grundlegender Aufbau CPU-System

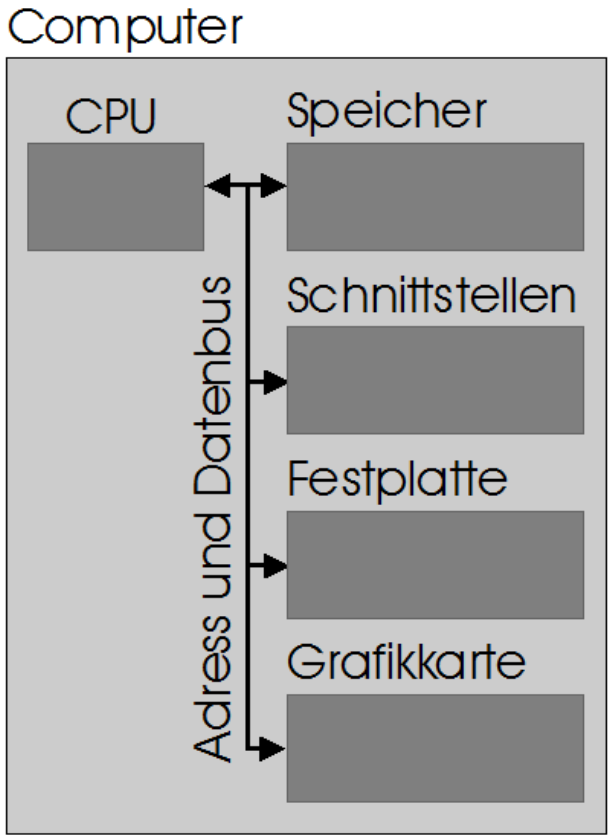
6 – Prozessoren

Grundlegender Aufbau CPU-System

- Aus TEI1 kennen Sie das EVA Prinzip: Eingabe, Verarbeitung, Ausgabe
- Im Computer gibt es i.a.
  - CPU (Central Processing Unit), die Recheneinheit
  - RAM (Random Access Memory)
  - ROM (Read only Memory) bzw. Flash-Memory für das BIOS (Basic Input Output System)
  - Festplattenspeicher oder SSD (Solid State Disk)
  - Grafikkarte
  - Interfaces (Schnittstellen)
  - Bus Systeme (z.B. USB = Universal Serial Bus, PCI-Express) meistens bestehend aus Daten- und Adressbus

6 – Prozessoren

Grundlegender Aufbau CPU-System

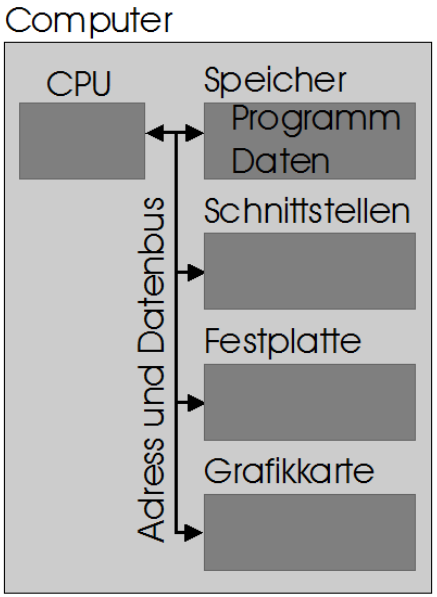


6 – Prozessoren

Grundlegender Aufbau CPU-System

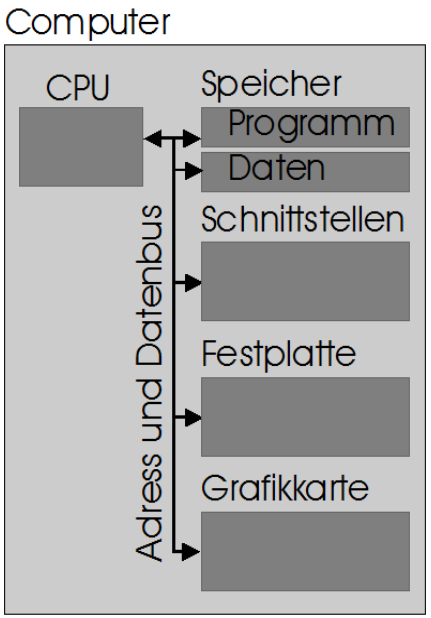
- Es gibt unzählige Computerarchitekturen. Wichtig für die Programmierung ist beispielsweise die Unterteilung in

Von Neumann Architektur



Moderner PC

Harvard Architektur



manche Digitalen Signal Prozessoren

6 – Prozessoren

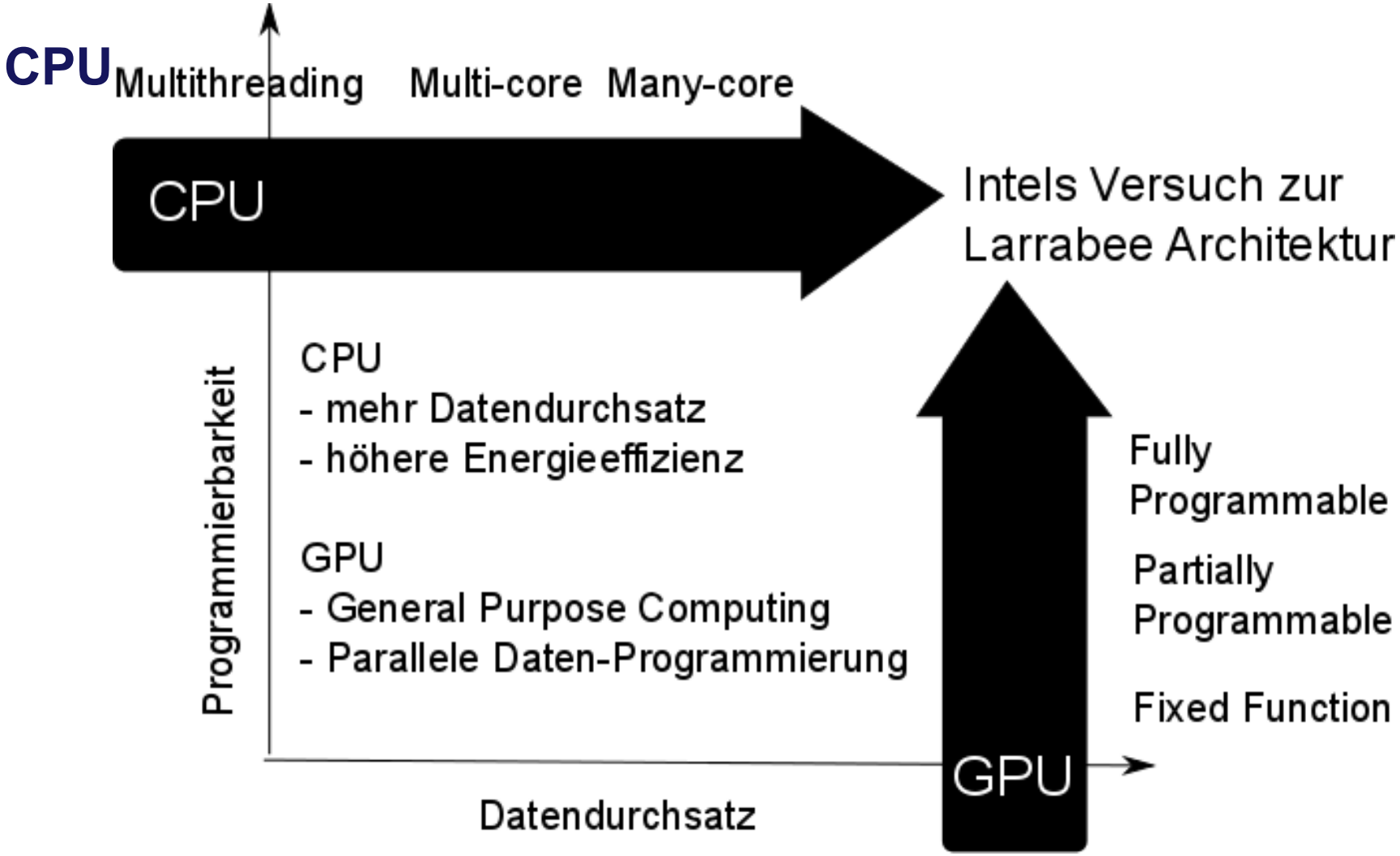
Schneller mit Cache

- Die Möglichkeit der Harvard Architektur gleichzeitig auf Programmspeicher und Datenspeicher zuzugreifen wird bei der von Neumann Architektur durch zusätzliche Speicher im Prozessor kompensiert.
- Dieser Cache (Daten-cache oder Programm-Cache) ist zwar wesendlich kleiner und schneller als der Hauptspeicher, kann aber Programmteile bzw einen Teil der Daten zwischenspeichern und so gleichzeitig Befehle aus dem Programm-Cache und Daten aus dem Daten-Cache laden.

→ Folgerung: Um Programme schnell zu bekommen, sollten die rechenaufwendigen Programmteile möglichst in den Cache passen! Von Vorteil ist es auch, wenn die Daten in den Daten-Cache passen.

Grundlegender Aufbau GPU-System

6 – Prozessoren



6 – Prozessoren

Grundlegender Aufbau GPU-System

- Moderne Grafikkarten führen heute zwischen 10^{11} und 10^{12} Fließkommaberechnungen pro Sekunde aus.
- Würden solche Operationen sequentiell verarbeitet müßte man die Prozessoren im Terraherz Bereich Takten und die Elektronik in subatomaren größen fertigen oder eben Signale mit über Lichtgeschwindigkeit senden.
- Darum basieren Grafikkarten auf hochparalelen Prozessorarchitekturen.

6 – Prozessoren

µC und DSP basierte Systeme

- µC (sprich: Mikrocontroller) gibt es in 4 Bit bis 64 Bit Architektur.
- Der Trend geht zur ARM basierten RISC (Risc Instruction Set Computer) Architektur
- In den meisten µC wird im Befehlssatz vorwiegend Integerarithmetik implementiert
- Im Zuge der immer höheren Integrationsdichte werden bei den kleineren µC ADC's (Analog Digitalwandler) und DAC's (Digital Analog Wandler), bei größeren werden Displaytreiber, Bildschirmspeicher etc. eingebaut
- Ausserdem werden Flash-Programmspeicher und RAM fortlaufend größer
- Typisches Einsatzgebiet: Küchenmaschinen, Autos, elektronische Geräte

6 – Prozessoren

µC und DSP basierte Systeme

•DSP's (sprich: Digitale SignalProzessoren) sind dafür optimiert große Datenmengen an Festkomma- oder Fließkommazahlen zu verarbeiten.

• Ihr Einsatzgebiet ist insbesondere in der Audio- und Video-Verarbeitung zu finden

• i.A. haben die Prozessoren Multiply-Add befehle, die vor allem FIR und IIR Filter schnell berechnen können

7 - Virtuelle Maschinen

7 - Virtuelle Maschine

Kapitel 7 - Virtuelle Maschinen

- Was ist eine VM?
- Stack-basierte/ Register-basierte VM
- Die Tiny-VM, Realisierung einer Register-basierten VM

7 - Virtuelle Maschine

Was ist eine VM?

- Eine Virtuelle Maschine ist ein emulierter Prozessor oder Computer
- Eine VM kann als Sandbox dienen, die verhindert, dass Programme einen kompletten Rechner zum Absturz bringen (VM-Ware oder Virtual-Box). So können auf einem Server mehrere virtuelle Maschinen aufgesetzt werden, die sich gegenseitig nicht beeinflussen können.
- Eine VM kann als Runtime Umgebung für eine Programmiersprache dienen (Java-VM, C# etc.). Damit ist sie leicht portierbar und garantiert, dass die Programme unter unterschiedlichen Betriebssystemen gleich ablaufen.
- Hauptvorteil von Virtuellen Maschinen ist die Plattform unabhängigkeit.
- Es gibt Registerbasierte VM's (z.B. VM-Ware) und Stackbasierte VM's (JAVA-VM)

7 - Virtuelle Maschine

Stackbasierte/ Registerbasierte VM

- Eine Stack-basierte VM (z.B.JAVA-VM) schiebt die Operanden, die in einer Rechnung später noch benötigt werden auf den Stack. Vorteil ist, dass die Anzahl der benötigten Speicherplätze für eine Rechnung nicht vorher bekannt zu sein muss.
- Eine Registerbasierte VM (z.B. Dalvik für Java-Prog.) hat Register, ähnlich einer CPU, mit denen gerechnet werden kann. Sind mehr Register für eine Berechnung nötig, als vorhanden, so müssen die Register zwischengespeichert werden. Für die optimierte Ausnutzung der Register gibt es aber schon sehr effiziente Algorithmen bzw. Compiler.

8 - Betriebssysteme

8 - Betriebssysteme

Kapitel 8 - Betriebssysteme

- Überblick
- Architektur von Betriebssystemen
- Prozesse und Threads
- Speicherverwaltung
- Ein-/Ausgabe Systeme
- Schutz und Sicherheit

8 - Betriebssysteme

Überblick: Wofür braucht man ein Betriebssystem

- Es sorgt für die ordentliche Konfiguration der Hardware
- Es bietet i.Allg. Treiber um bestimmte Hardware-Komponenten anzusprechen
- Es beschränkt Hardwarezugriffe im USER-Mode
- Es implementiert das Memory-Management
- Es bietet oftmals eine Grafische Oberfläche oder eine Konsole um die Kommunikation mit dem Nutzer zu erlauben
- Es managed das Multitasking von Prozessen*
- Es bietet Funktionen für Ein-/Ausgabe z.B. um auf das Filesystem zuzugreifen
- Es verwaltet mehrere Nutzer*

* Diese Funktionalität ist nicht in allen Betriebssystemen enthalten

8 - Betriebssysteme

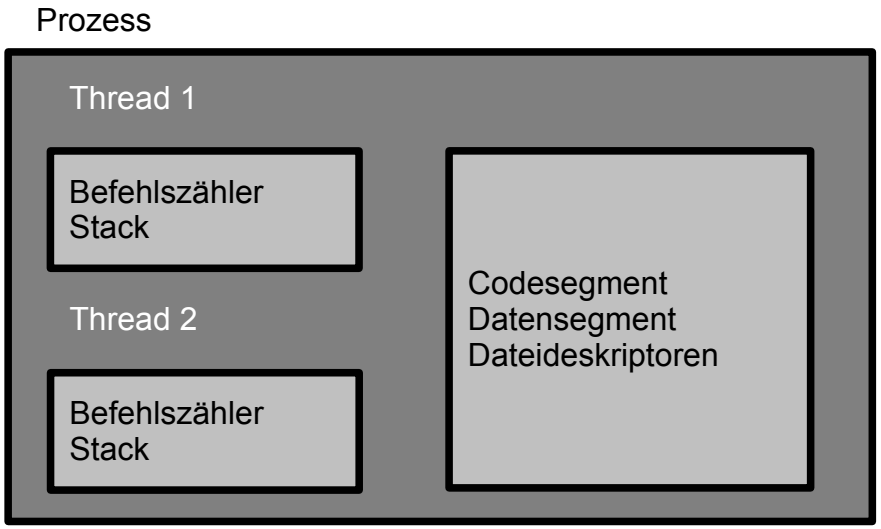
Architektur von Betriebssystemen

- Monolithische Architektur: alle wesentlichen Komponenten in einem homogenen Code, effizient aber wenig flexibel anpassbar
- Kern-Schale-Architektur: wichtigsten Komponenten im Kern (z.B. ist bei Linux die Prozessverwaltung im Kern (engl. Kernel) und der Kommando-Interpreter in der Schale (Shell))
- Hierarchische Schichten (Mehrschichtenmodell): jede Schicht bietet der darüberliegenden Schicht Dienstleistungen, Schichten austauschbar
- Mikrokern (micro kernel): bildet nur eine Art Infrastruktur mit minimalem Funktionsumfang, z.B. bei Client-Server-Modellen

8 - Betriebssysteme

Prozesse und Threads

- Prozesse sind Programme. Im Multitasking Betriebssystem können mehrere Prozesse (engl. task) parallel (gleichzeitig) laufen.
- Threads sind parallel ablaufende Routinen oder Unterprogramme. Sie laufen meist innerhalb eines Prozesses und teilen sich Codesegment, Datensegment, Dateideskriptoren



Memory Management

- Ein üblicher Computer hat im Prozessor eine Memory Management Unit (MMU)
- Diese ermöglicht es Programme in einem virtuellen Adressraum zu legen
- Das bedeutet, dass die reale Adresse an der ein Program liegt eine andere ist, als die virtuelle Adresse, so dass absolute Sprünge (an eine Adresse), Sprünge an die virtuelle Adresse sind
- Zusätzlich gibt es meist einen DMA (Direct Memory Access), mit Hilfe deren es möglich ist Speicher zu kopieren, ohne Rechenressourcen zu blockieren.

Scheduling/ Multitasking

- Multitasking heisst, dass man vom Prozessor mehrere Prozesse parallel oder quasiparallel ausführen lassen kann
- Das funktioniert z.B. für einen Prozessor mit einem Rechenkern folgendermassen:

  - Ein Prozess wird ausgeführt

  - Nach einer vorgegebenen Zeit wird dieser von einem Timer-Interrupt unterbrochen

  - Die Register und der Program Counter (PC) werden gesichert

  - Die Register und der PC vom nächsten geplanten Prozess werden geladen

  - Der nächste Prozess wird ausgeführt

8 - Betriebssysteme

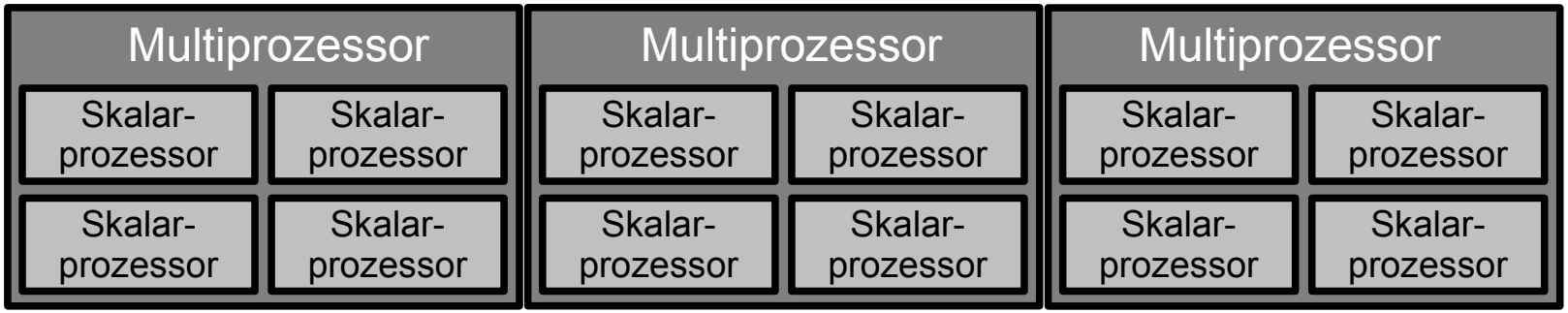
Dateisystem

- Ein Filesystem ermöglicht das anlegen und beschreiben von Ordnern und Dateien
- Dafür hat jedes Filesystem eine File Allocation Table, eine art Tabelle, in der steht, wie der Ordner oder das File heisst, wie groß das File ist und an welcher Adresse der Festplatte/SSD/SD-CARD etc. das File beginnt.
- Oft wird ein Order wie eine Datei behandelt, in denen Referenzen auf die enthaltenen Dateien und Ordner stehen.
- Oftmals werden in der File Allocation Table auch der Nutzer, der die Datei angelegt hat und die Rechte einer Datei festgelegt

9 – Grafikkarten Programmierung

9 – Grafikkarten programmierung

Grundlegender Aufbau GPU-System



- Die Hardware ist mehrstufig parallel:
  - (GPU - Multiprozessoren - Skalarprozessoren)
- Das Programmiermodell ist mehrstufig parallel:
  - (grid - blocks - threads)
- Es gibt unterschiedliche Speicherarten
  - Globale Memory, Shared Memory, Texture Memory, Constant Memory, Register

9 – Grafikkarten programmierung

Einführung CUDA (Programmiermodell)

- CUDA Programme bestehen aus:
  - Ein „Host“ Programm (auf CPU)
  - Kontrolliert den wesentlichen Programmablauf
  - Ist für den Datenaustausch zwischen CPU und GPU verantwortlich

- „Accelerator“ Programmiermodell
  - CPU = Master
  - GPU = Slave
  - CPU und GPU arbeiten parallel, CPU hat die Kontrolle über die GPU

9 – Grafikkarten programmierung

Einführung CUDA (Programmiermodell)

- Kernels
  - Ist nur vom Host aus aufrufbar
  - Beispiel: __global__ void getSum(float* inputA, float* sum);
  - Man nennt diese Funktionen CUDA Kernel
  - Die Funktion muss immer vom Typ void sein.
- Device Funktionen
  - Ist nur vom device aus aufrufbar
  - Beispiel: __device__ float max(float x, float y);
  - Kann nicht rekursiv sein für compute capability < 2.0
- Andere Funktionen
  - Alle anderen Funktionen werden als Host Code compiliert

9 – Grafikkarten programmierung

Einführung CUDA (Programmiermodell)

- Kernels können vom Hostcode wie jede andere Funktion aufgerufen werden.
- Bemerkung:
  - Kernel aufrufe sind asynchron d.h. , dass die Funktion sofort zurückkehrt bevor die Bearbeitung der Funktion erfolgt ist.

- Beispiel:

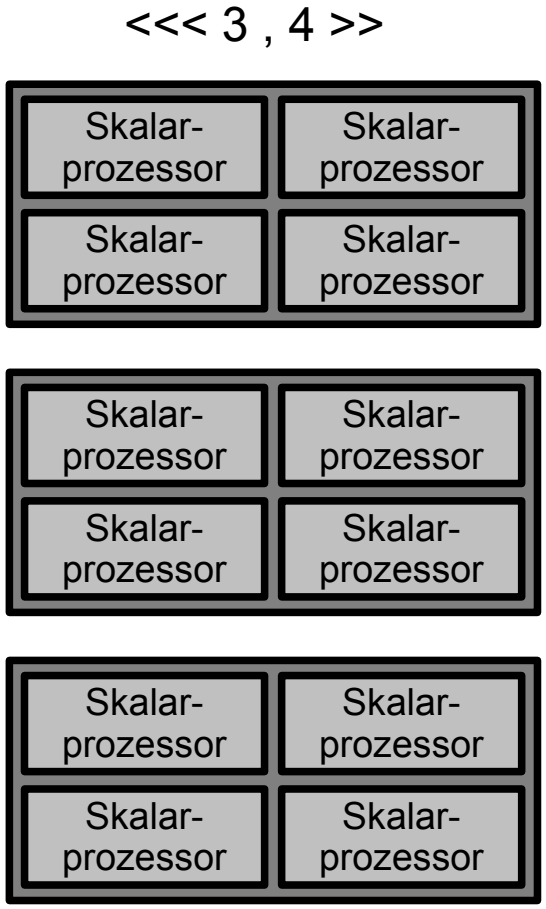
```
int main(int argc, char** argv){
    // hier Funktionsaufrufe für „Speicher reservieren“
    // und Speicher übertragen CPU->GPU…

    myCudaFunction<<<gridSize, blockSize>>> (inputA, inputB,
    output, size);

    cudaThreadSynchronize();
    // hier Funktionsaufrufe für „Speicher übertragen“
    // GPU->CPU, dann Speicher freigeben
    return 0;
}
```

9 – Grafikkarten programmierung

Einführung (Kernel Aufrufskonfiguration)

- Was heißt Kernel Aufrufskonfiguration?
  <<<gridSize, blockSize >>>

- Block von logischen Threads
  - Werden parallel ausgeführt
  - Werden in unterschiedlichen Skalarprozessoren des selben Multiprozessors ausgeführt
- Grid of Blocks
  - Werden unabhängig von-einander seriell oder parallel ausgeführt
  - Werden auf unterschiedliche Multiprozessoren verteilt

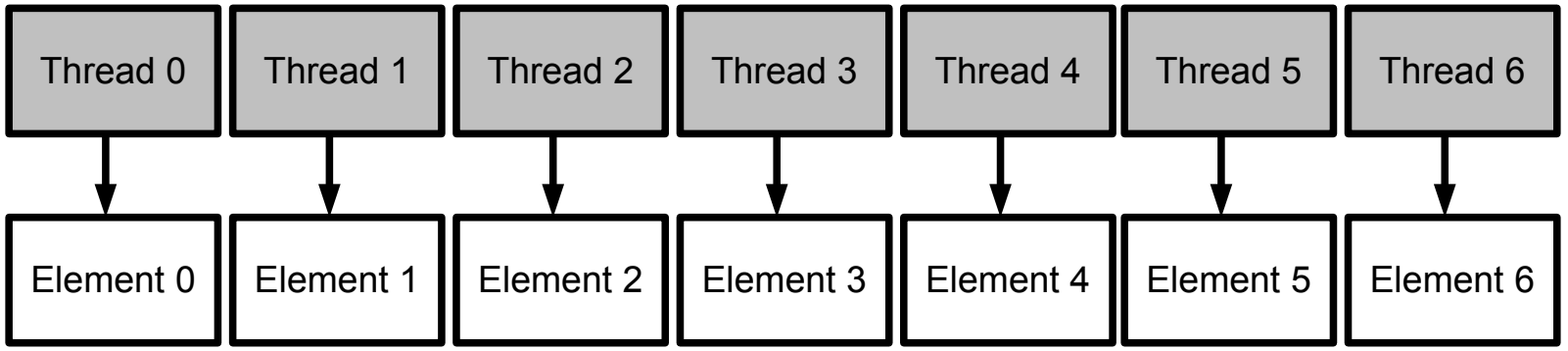


9 – Grafikkarten programmierung

Beispiel: addiere 2 Vektoren auf einem Multiprozessor

- Wenn vektor size = block size:

```
__global__ void VectorSum(float* inputA, float* inputB, float* output){
    output[threadIdx.x] = inputA[threadIdx.x] + inputB[threadIdx.x]
}
```



9 – Grafikkarten programmierung

Speicher reservieren und kopieren

- Device Memory Allozieren: `cudaError_t cudaMalloc( void** pointer, size_t size );`
- Device Memory freigeben: `cudaError_t cudaFree( void* pointer );`
- Memory kopieren
```
cudaError_t cudaMemcpy( void* dat, const void* arc,
                        size_t count, enum cudaMemcpyKind kind );
```
  - kind kann sein: `cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice`
- Reservieren und freigeben von page-locked host Memory

```
cudaError_t cudaMallocHost(void** pointer, size_t size);
cudaError_t cudaFreeHost(void* pointer);
```

- Warum page locked memory?
  - Memory kann schneller von und zum Device kopiert werden
  - Zugriff vom Device ist möglich
  - Asynchrones kopieren ist möglich

10 – Mikrokontroller Programmierung

10 – Mikrokontroller Programmierung

Inhalt

- Raspberry PI (praktische Übung)
- Arduino (praktische Übung)