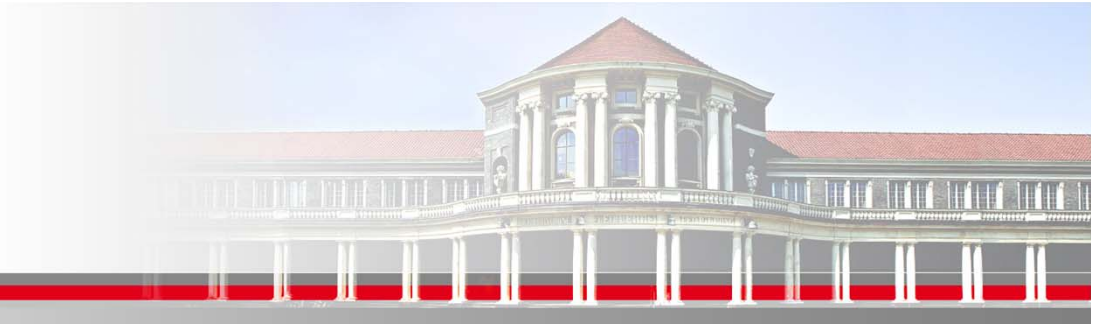
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Übung:
Datenvisualisierung und GPU-Computing

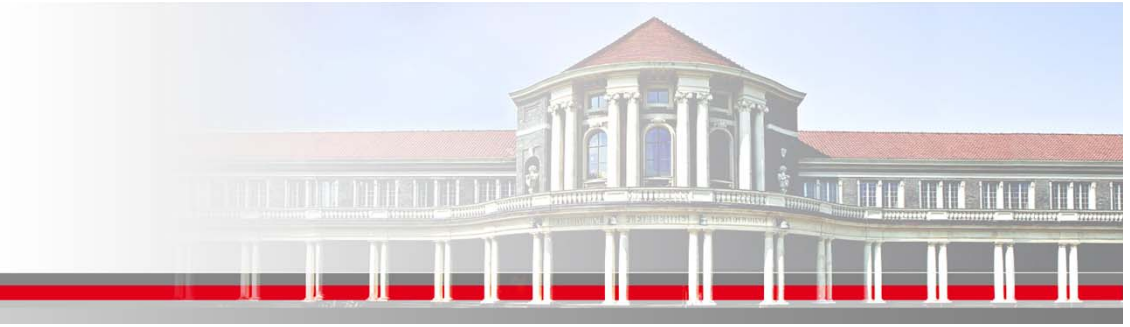
Programmieren in C und C++
Teil 2

Michael Vetter
michael.vetter@rrz.uni-hamburg.de



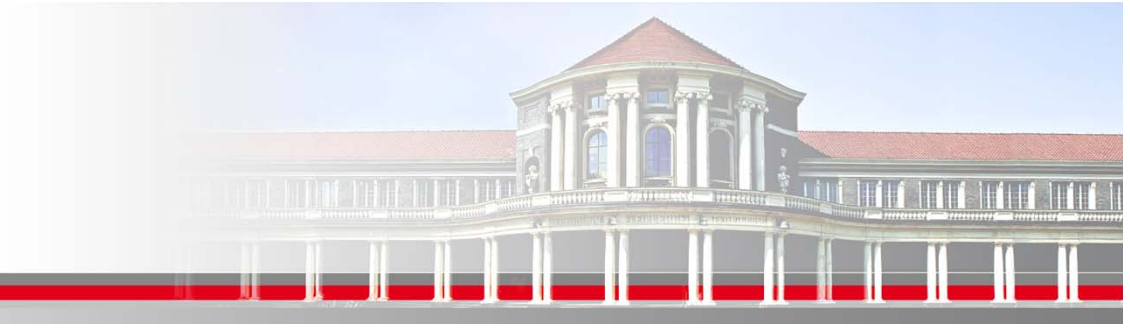
Agenda

- Überladen von Funktionen
- Sichtbarkeit und Lebensdauer von Variablen
- Komplexe Datentypen
- Dynamische Speicherverwaltung
- File-I/O (2)
- Codestrukturierung



Überladen von Funktionen (1)

- In C++ ist es möglich Funktionen zu definieren, die sich nur in der Anzahl bzw. den Typen der Parameter unterscheiden (*overloading*) → Unterscheidung nur durch den Rückgabetyp reicht jedoch nicht!



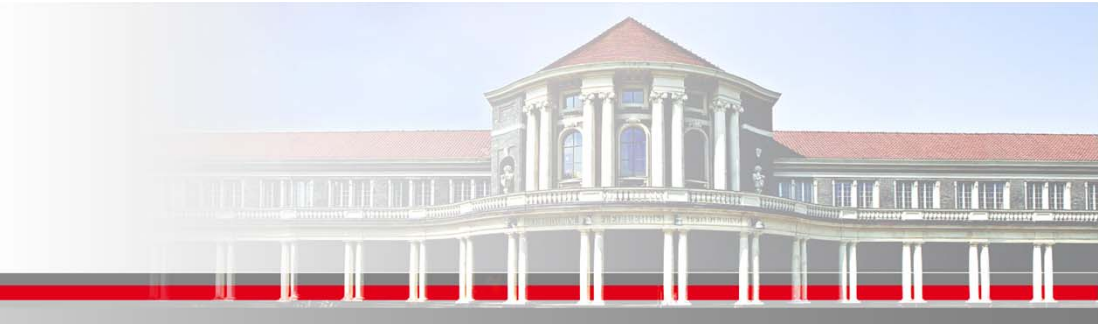
Überladen von Funktionen (2)

- Beispiel:

```
float  sqrt(float  value);
double sqrt(double value);
```

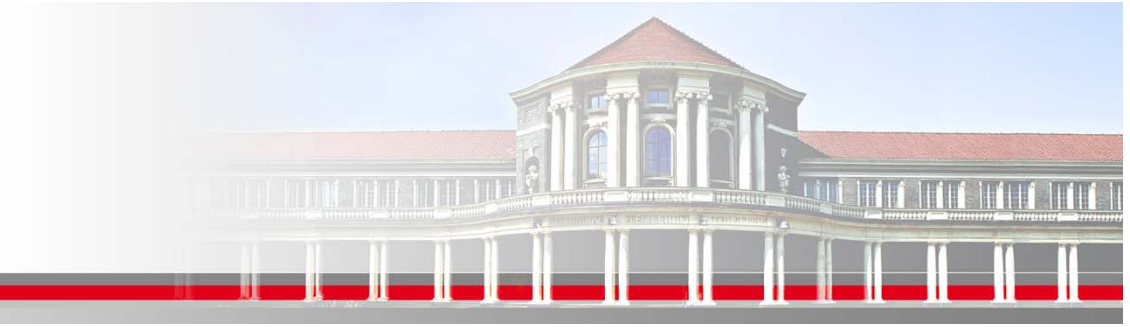
- Bei einem Aufruf wird anhand der realen Parameter entschieden, welche Variante auszuführen ist:

```
float  f = 3.14159f;
double d = 2.71828;

float  x = sqrt(f);   // Ruft float-Variante auf
double y = sqrt(d);   // Ruft double-Variante auf
```



Sichtbarkeit von Variablen

- Variablen sind grundsätzlich nur innerhalb des Blockes sichtbar, in dem sie definiert wurden.
- Variablen die in keinem Block definiert wurden sind global sichtbar (innerhalb der Quellcodedatei).
- Variablen, die über Dateigrenzen hinweg global sichtbar seien sollen werden mit dem Schlüsselwort **extern** deklariert und in nur einer Quellcodedatei ohne das Schlüsselwort **extern** definiert (Speicherzuweisung). Externe Variablen können auch innerhalb eines Blockes definiert werden.



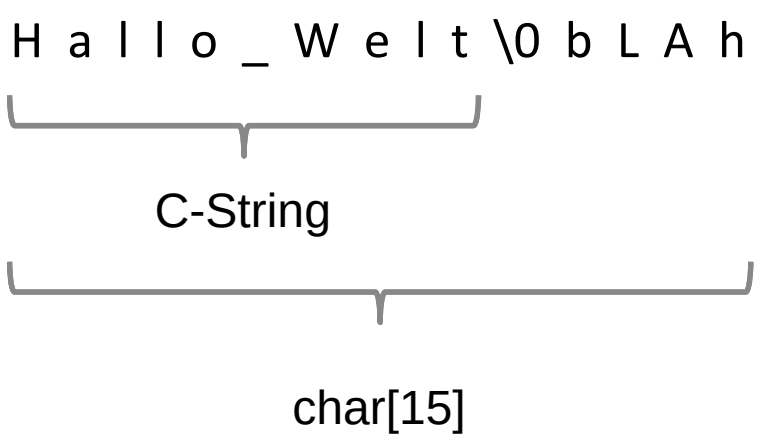
Lebensdauer von Variablen

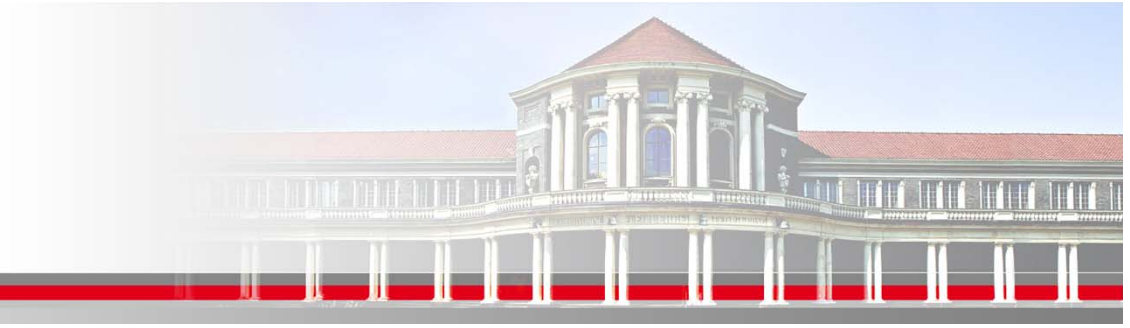
- Die Lebensdauer von Variablen, die innerhalb einer Funktion definiert wurden, endet in der Regel mit dem Ende der Funktion.
- Soll eine lokale Variable über mehrere Aufrufe einer Funktion hinweg existieren, so ist sie mit dem Schlüsselwort **static** zu definieren.

C-Strings

Definition:

Ein C-String ist eine Zeichenkette in einem char-Array, welche mit dem Zeichen ‘\0‘ terminiert wird.

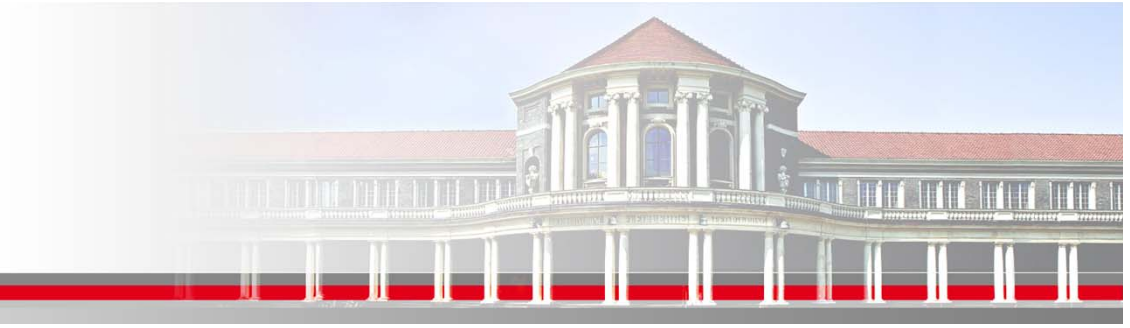




Structs

- Ein C-Struct
  - Gruppieren von semantisch zusammenhängenden Daten
  - Ähnlich einer Klasse

```
struct Student {
    int m_iId;
    bool m_IsGrad;
}; // the declaration must end with ';'
```

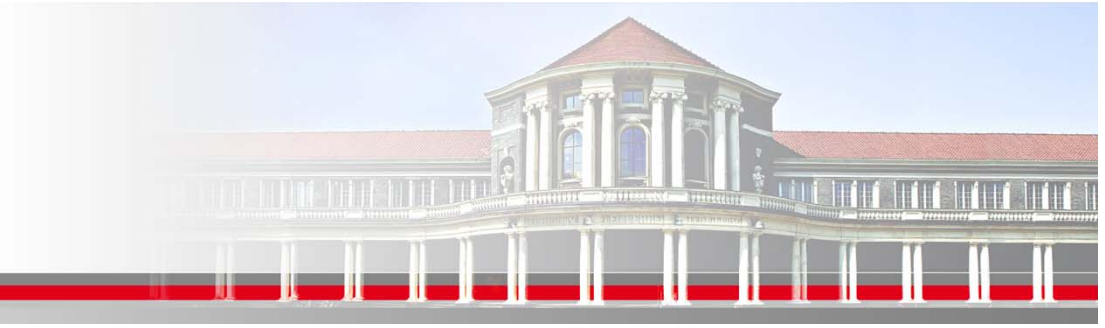


Aufzählungen

- Neue Datentypen mit festen Werten können wie folgt definiert werden:

```
enum Color { RED, BLUE, YELLOW };

Color col;
col = BLUE;
```

- Der Wert der Variablen vom Typ Color kann einen der Folgenden Werte annehmen { RED, BLUE, YELLOW }

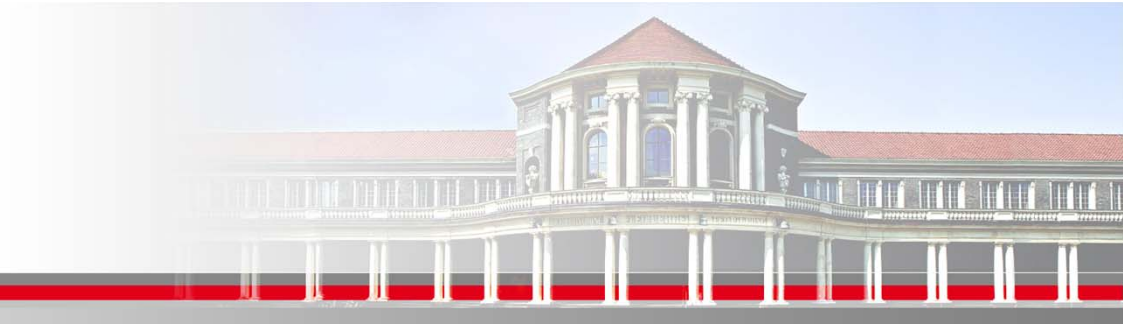


- Zugriff auf Variablen einer Struktur:

```
t_student1.m_iId = 123;
t_student2.m_iId = t_student1.m_iId + 1;
```

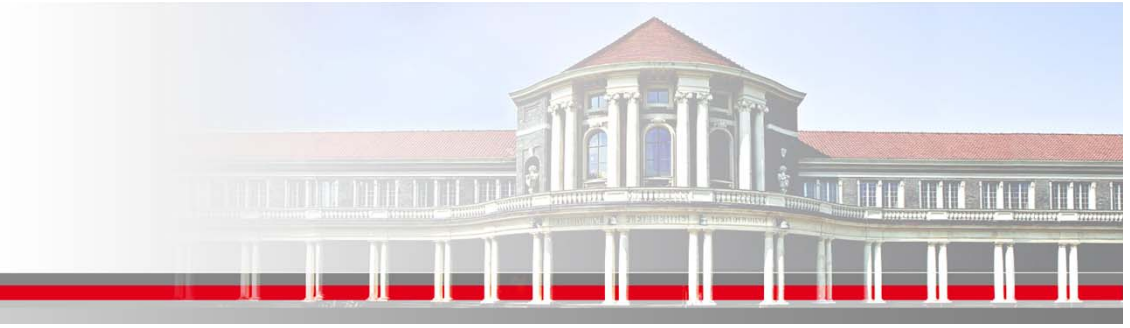
- Verschachtelte Strukturen:

```
struct Address {
        string m_City;
        int m_Zip;
};
struct Student {
        int m_iId;
        bool m_bIsGrad;
        Address m_Address;
};
```



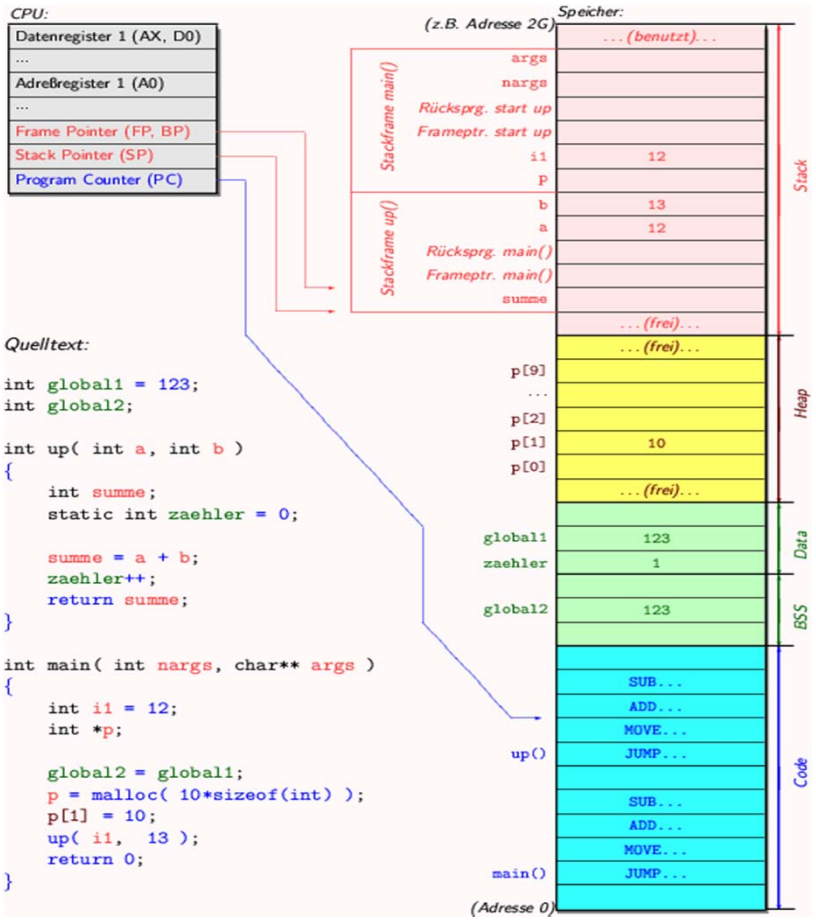
- Neue Datentypen können durch bereits existierende Datentypen definiert werden:

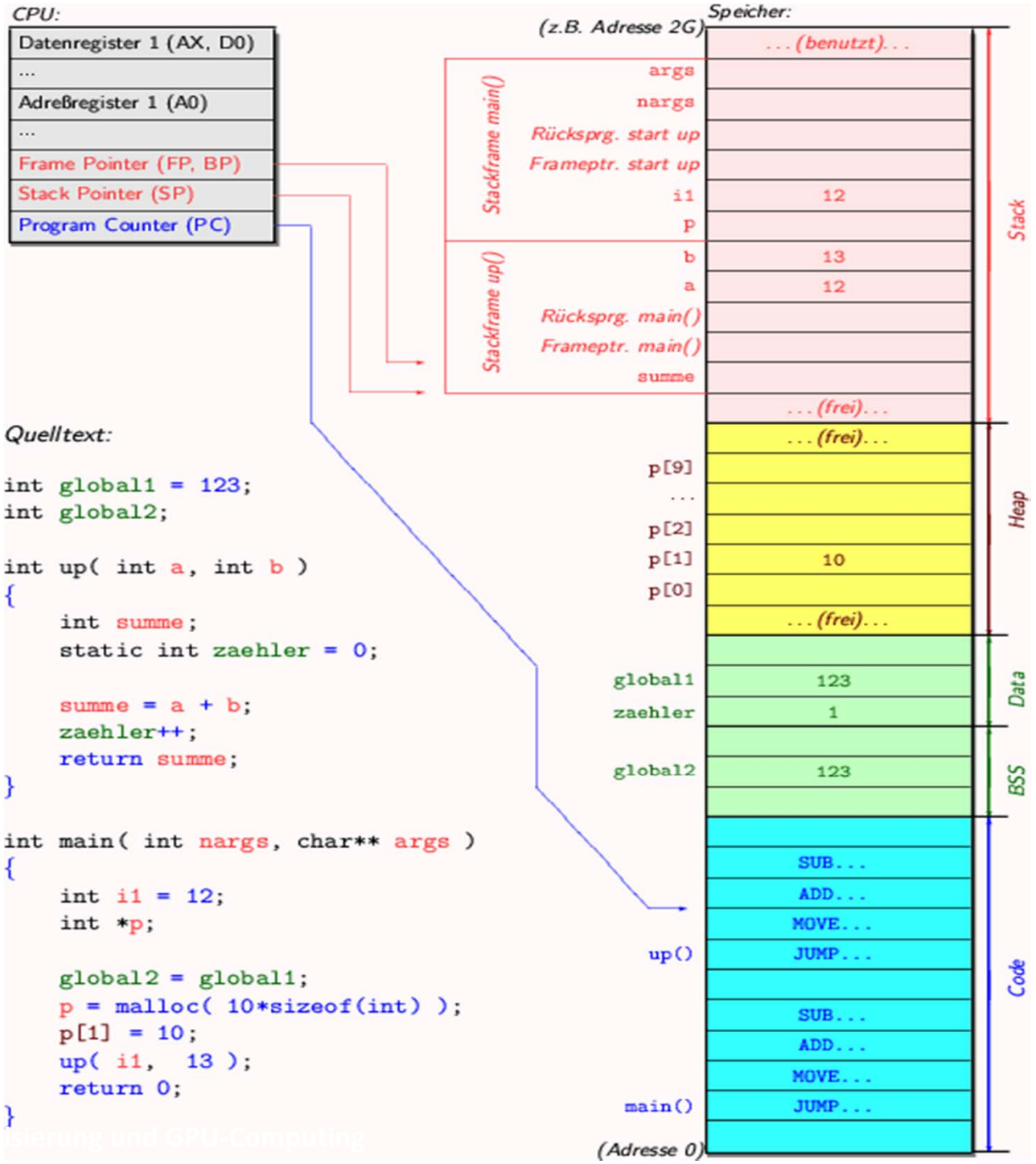
```
typedef double EuroType;
EuroType hourSalary = 10.50;
```

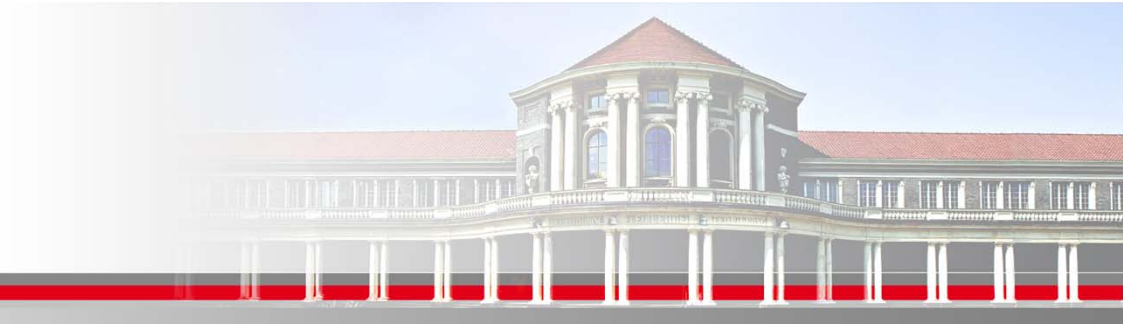


Speicher eines Prozesses:

- *Stack*
- *Heap*
- *...*




CPU:
Datenregister 1 (AX, D0)
...
Adreßregister 1 (A0)
...
Frame Pointer (FP, BP)
Stack Pointer (SP)
Program Counter (PC)
Speicher:
(z.B. Adresse 2G)
...(benutzt)...
Stackframe main()
args
nargs
Rücksprg. start up
Frameptr. start up
i1
12
p
Stackframe up()
b
13
a
12
Rücksprg. main()
Frameptr. main()
summe
...(frei)...
Stack
...(frei)...
p[9]
...
p[2]
p[1]
10
p[0]
...(frei)...
Heap
global1
123
zaehler
1
Data
global2
123
BSS
SUB...
ADD...
MOVE...
up()
JUMP...
SUB...
ADD...
MOVE...
main()
JUMP...
Code
(Adresse 0)
Quelltext:
int global1 = 123;
int global2;
int up(int a, int b)
{
int summe;
static int zaehler = 0;
summe = a + b;
zaehler++;
return summe;
}
int main(int nargs, char** args)
{
int i1 = 12;
int *p;
global2 = global1;
p = malloc(10*sizeof(int));
p[1] = 10;
up(i1, 13);
return 0;
}



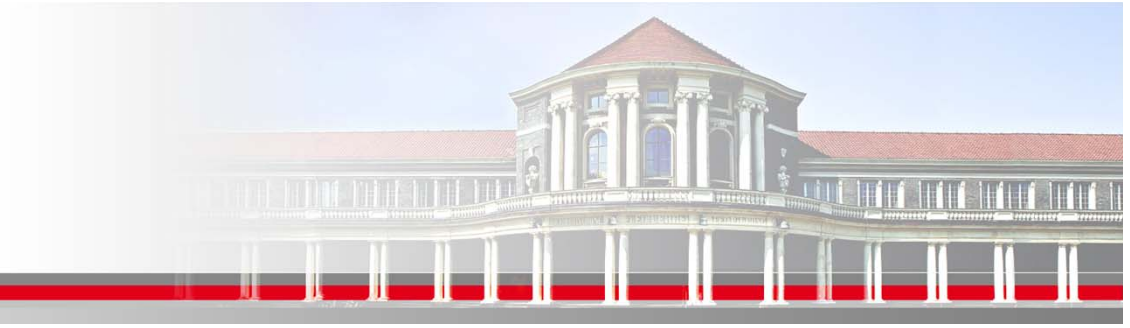
Dynamische Speicherverwaltung (C++-Variante)

- In C++ gibt es keine automatische Speicherbereinigung

```
int *pi_p = new int;          // pi_p zeigt auf
neuen                         // Speicherplatz
...
delete pi_p; // Speicher MUSS wieder
             // freigegeben werden
```

- Aber niemals

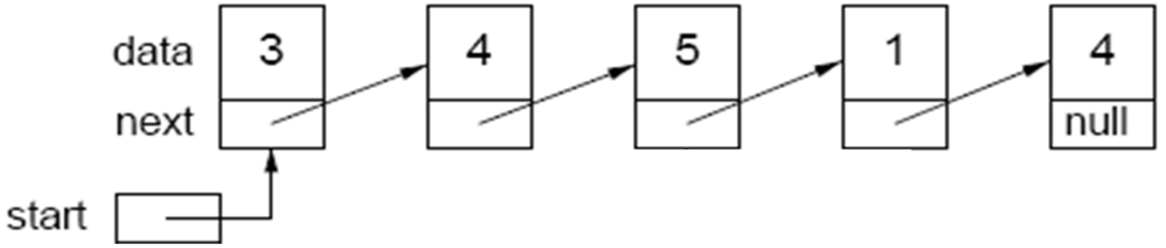
```
int *pi_p; // pi_p ist nicht initialisiert!!
*pi_p = 3;

int *pi_p=NULL; // pi_p ist ein null-Zeiger
                // (ungültige Adresse!!!
*pi_p = 3;
```

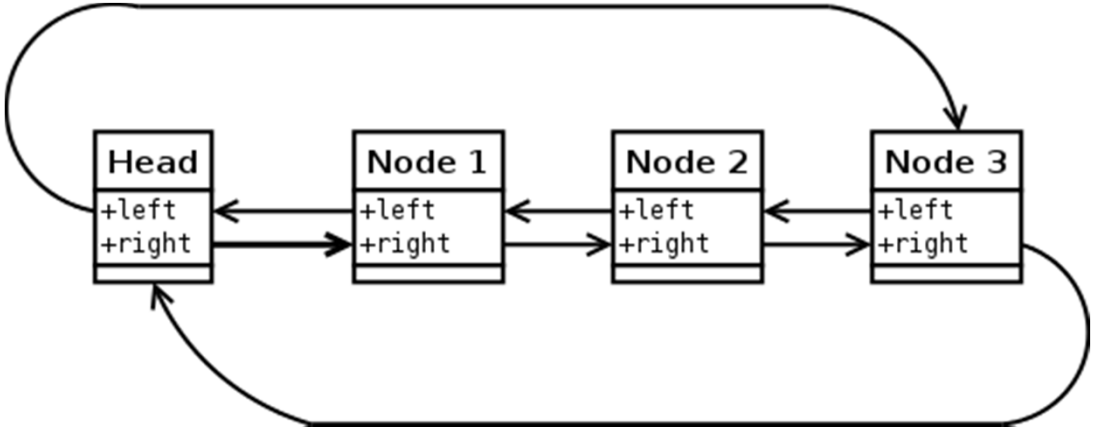


- Mache niemals:

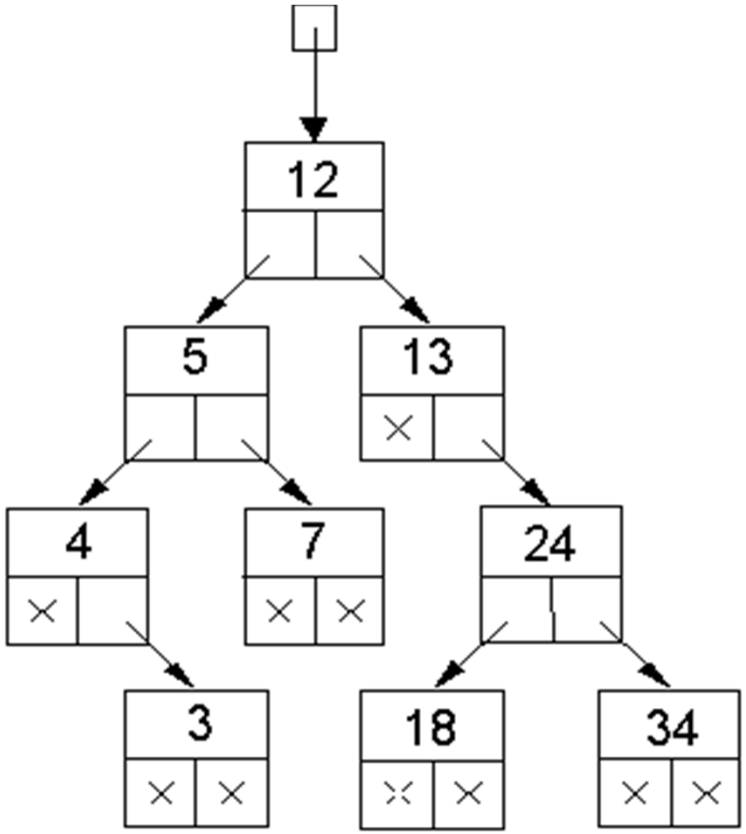
```
int *pi_p = new int;
delete pi_p;
*pi_p = 5; // pi_p ist nicht mehr gültig!!

int *pi_p, *pi_q;
pi_p = new int;
pi_q = pi_p; // pi_p zeigt auf selbe Adresse wie pi_q
delete pi_q;
*pi_p = 3; // Adresse pi_q = pi_p is nicht mehr
           // reserviert!! (abschüssiger Zeiger)

int *pi_p = new int;
pi_p = NULL; // ändere keinen Zeiger ohne den
             // Speicherplatz freizugeben auf den er
             // vorher gezeigt hat (Speicherüberlauf)
```

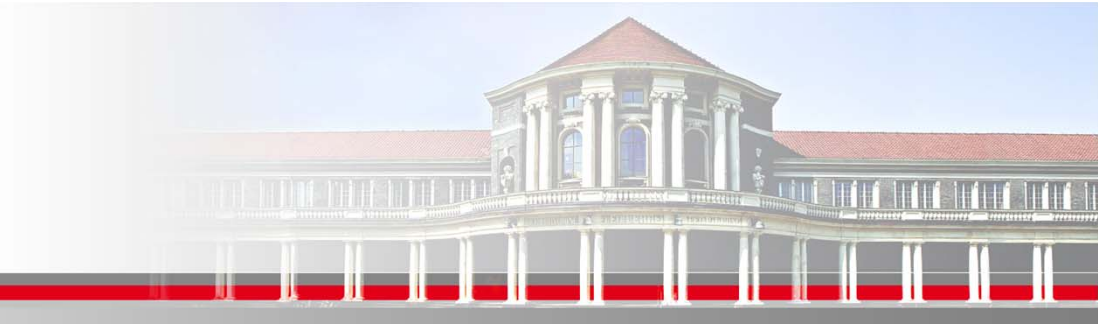
Dynamische Datenstrukturen: Listen





Dynamische Datenstrukturen: Bäume





File-I/O – C-Variante (1)

Einbinden des Header-Files:

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

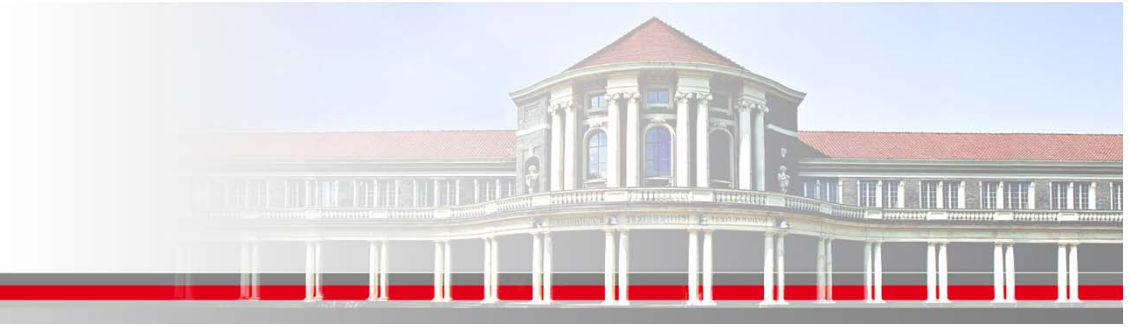
Öffnen der Datei:

```
FILE *file = fopen(filename, parameter);
```

„r“, „w“, „rw“, „a“

Schließen der Datei:

```
fclose(file);
```



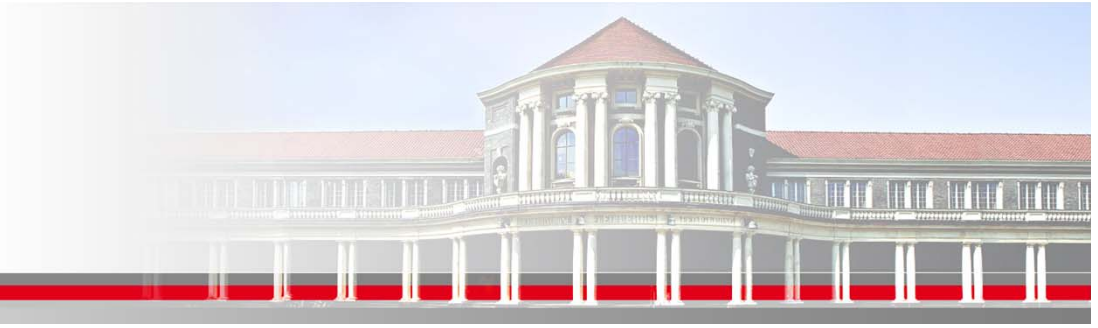
File-I/O – C-Variante (2)

Formatierte Ausgabe:

fprintf(*file*, *formatbeschreiber*, *liste_von_variablen***);**

Formatierte Eingabe:

fscanf(*file*, *formatbeschreiber*, *liste_von_variablen***);**



File-I/O – C++-Variante (1)

Einbinden des Header-Files:

```
#include <fstream>
```

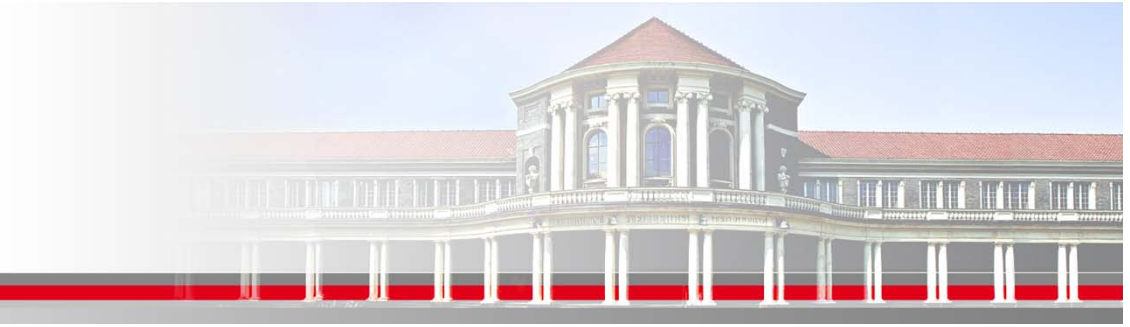
Öffnen der Datei:

```
std::ifstream   infile (filename, parameter);
std::ofstream   outfile (filename, parameter);
std::fstream    file (filename, parameter);
```

Parameter: std::ios::in, std::ios::out, std::ios::app, std::ios::trunc, …

Schließen der Datei:

```
file.close();
```



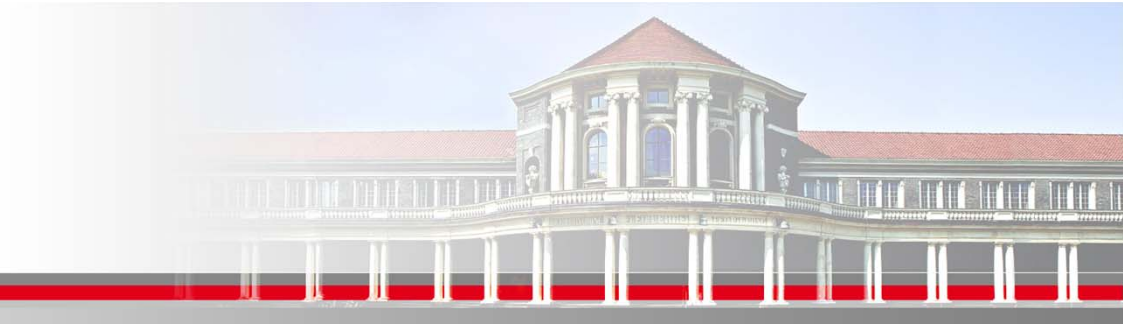
File-I/O – C++-Variante (2)

Formatierte Ausgabe:

```
file << variable;
```

Formatierte Eingabe:

```
file >> variable;
```



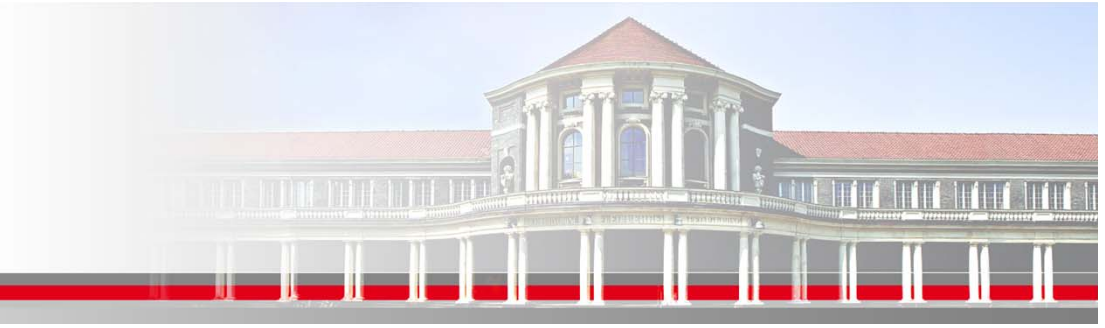
Codestrukturierung

- Der Code besteht aus Direktiven, Deklarationen und Definitionen
- Übliche Aufteilung des Codes:
  - Header-Dateien
  - Quellcode-Dateien

Präprozessor-Direktiven
- Compilerschalter
- Includes
  - Systemincludes
  - Individuelle Includes
- Symbolische Konstanten
- Makros

Deklarative Teile
- Datendeklaration
  - Deklaration von Datenstrukturen
  - Deklaration von Datentypen
- Variablendeklarationen
  - Externverweise auf globale Variablen
- Funktionsprototypen
  - Externverweise auf globale Funktionen
  - Vorwärtsverweise auf lokale Funktionen

Code
- Variablen außerhalb von Funktionen
  - lokale Variablen
  - globale Variablen
- Funktionen
  - lokale Funktionen
  - globale Funktionen



Präprozessordirektiven

```
#define name
#define name wert
#define name(platzhalter) makro
```

```
#ifdef name
...
#endif
```

```
#ifndef name
...
#endif
```

```
#ifdef name
...
#else
...
#endif
```