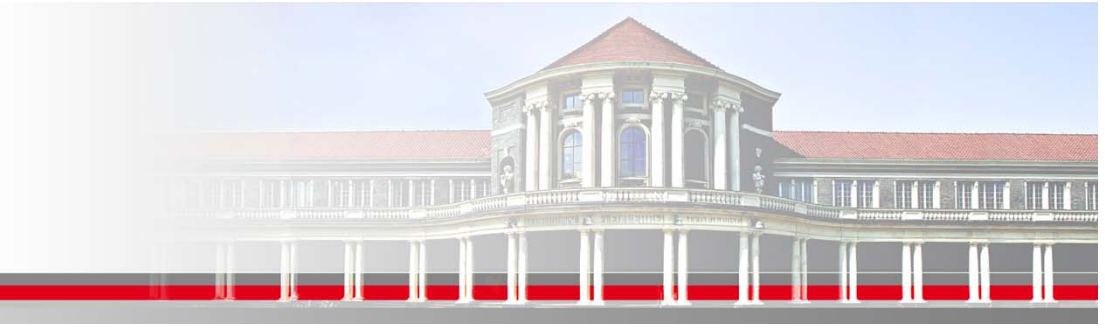
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Übung:
Datenvisualisierung und GPU-Computing

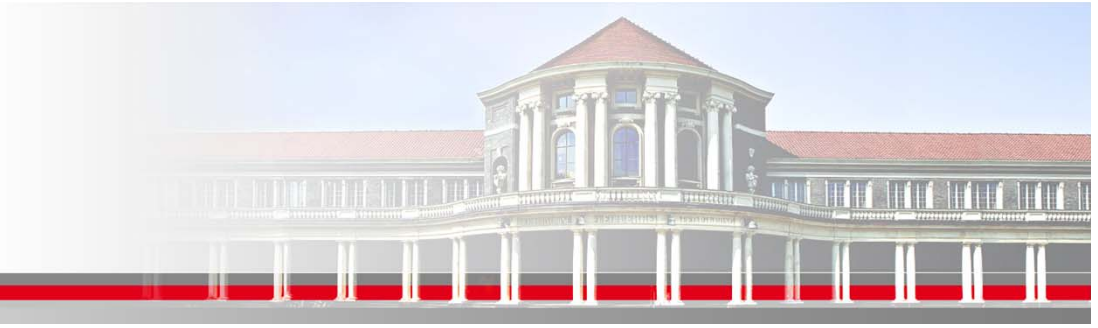
Programmieren in C und C++
Teil 1

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



Agenda

- Ablauf der Übung
- Der C++-Compiler
- Datentypen und Operatoren
- Zeiger und Referenzen
- Kontrollfluss
- Eingabe- und Ausgabeoperationen



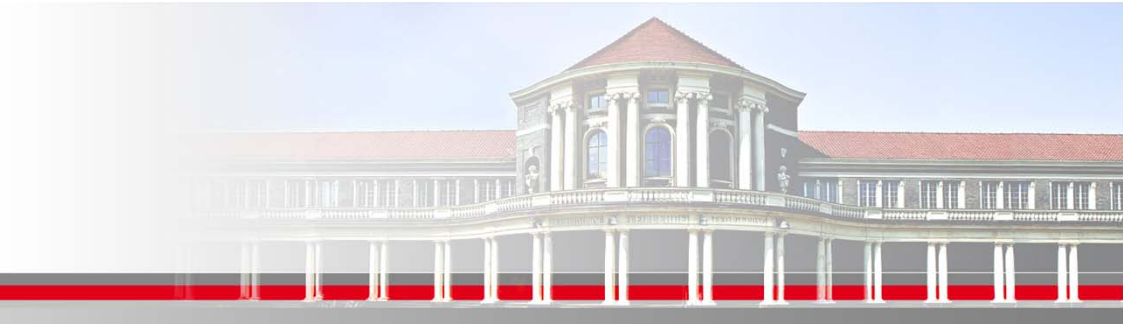
Vorstellung der Runde

Michael Vetter
Regionales Rechenzentrum der Universität Hamburg
Scientific Visualization & Parallel Processing

michael.vetter@rrz.uni-hamburg.de

Schlüterstr. 70, D-20146 Hamburg
Raum 314

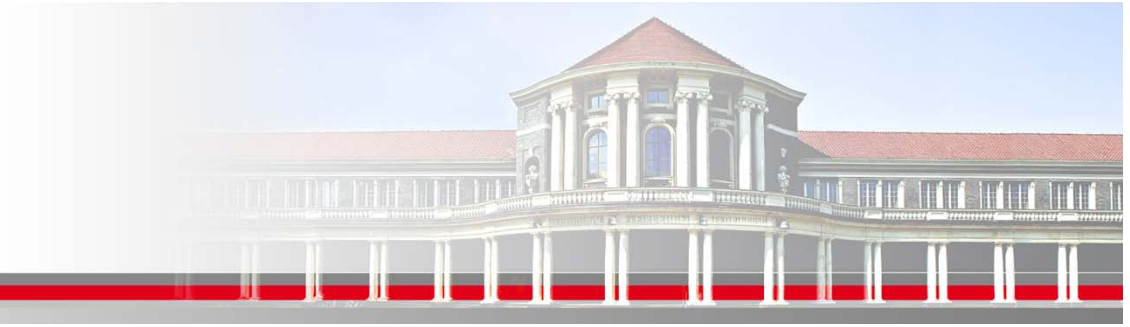
Sprechstunde: nach Vereinbarung



Vorläufiger Lehrplan

Vorlesung	Übung	Vorlesung	Übung
01.04.2013	02.04.2013	Ostermontag	fällt aus
08.04.2013	09.04.2013	Organisatorisches, Einführung, Szenarien	C++ Teil 1
15.04.2013	16.04.2013	Datenquellen, Display, Rendering	C++ Teil 2
22.04.2013	23.04.2013	Grafikprogrammierung: OpenGL	fällt aus (Dies Academicus)
29.04.2013	30.04.2013	Volumenvisualisierung, Strömungsvisualisierung	C++ Teil 3
06.05.2013	07.05.2013	Parallele Architekturen, PThreads	OpenGL 1
13.05.2013	14.05.2013	OpenMP, MPI	OpenGL 2
20.05.2013	21.05.2013	Pfingstferien	Pfingstferien
27.05.2013	28.05.2013	CUDA, OpenCL	PThreads
03.06.2013	04.06.2013	Farb- und 3D-Darstellung, Interaktion	OpenMP
10.06.2013	11.06.2013	VTK 1	MPI
17.06.2013	18.06.2013	fällt aus (ISC)	fällt aus (ISC)
24.06.2013	25.06.2013	VTK 2	CUDA / OpenCL
01.07.2013	02.07.2013	Zusammenfassung und Demo	OpenCL
08.07.2013	09.07.2013	Exkursion	VTK

Einzelne Themen können sich noch ändern!

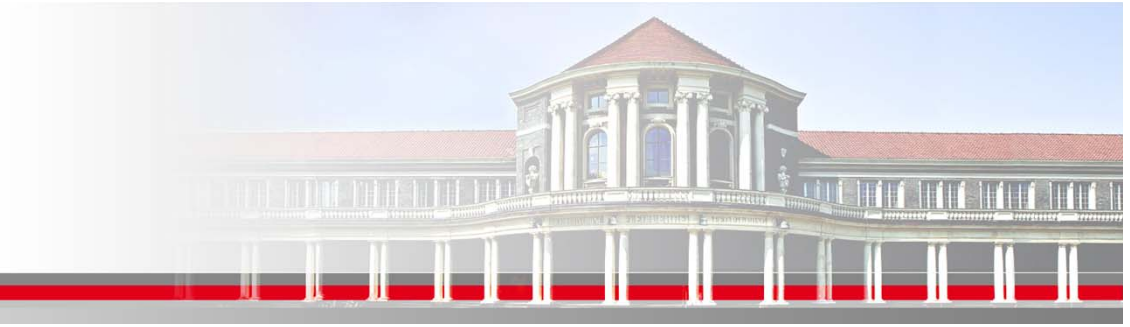


Links

http://openbook.galileocomputing.de/c_von_a_bis_z/

http://www.cplusplus.com/

www.google.de



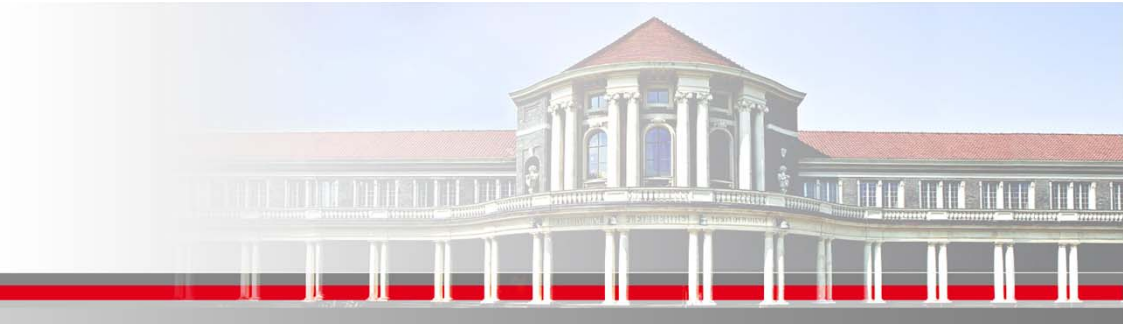
Kurze Einführung in die Programmiersprachen C und C++

Was ist C:

- Prozedurale Programmiersprache
- Ursprünglich für Unix Systemprogrammierung entwickelt
- Compiler für sehr viele Plattformen verfügbar

Was ist C++:

- Objektorientierte Programmiersprache
- Entwicklung aus der Programmiersprache C

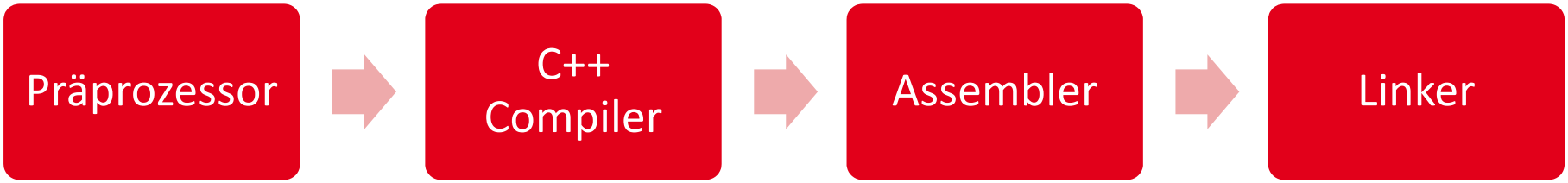


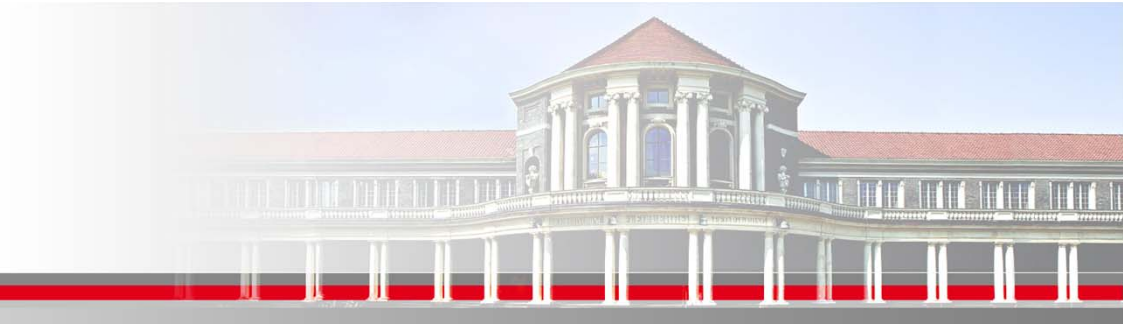
Kurze Einführung in die Programmiersprachen C und C++

C++-Compiler:

- GNU Compiler Collection (unter Windows MinGW)
- Microsoft Visual C++
- Intel C++ Compiler
- Borland C++ Builder

Ablauf des C++-Compilers

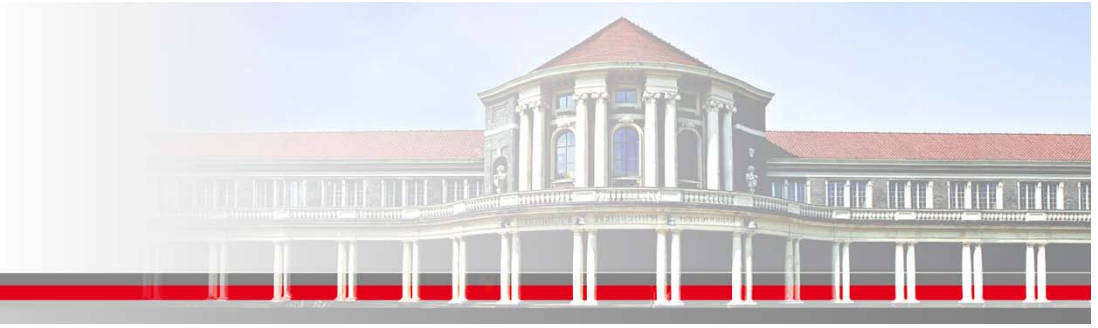




Einstiegspunkt

Grundgerüst eines C-Programmes:

```
1  int main()
2  {
3      // Ein Kommentar
4      /* noch ein Kommentar */
5      return 0;
6  }
```

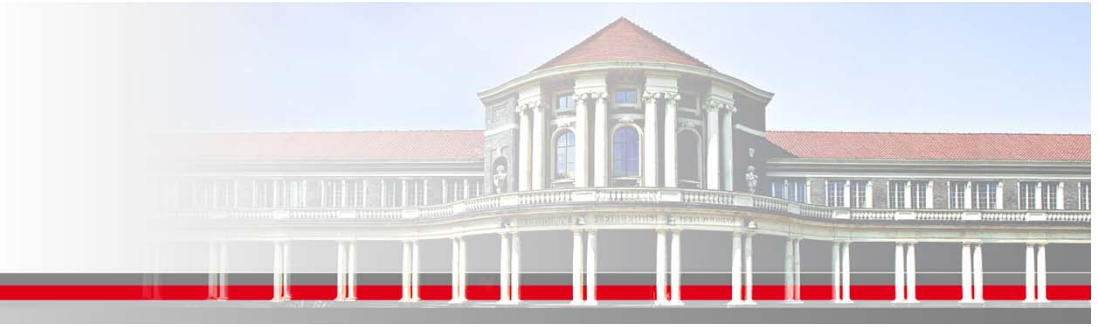


Präprozessordirektiven (1)

Erweitertes Grundgerüst eines C-Programmes:

Fügt den Inhalt der angegebenen Datei in dieses Dokument ein.

```
1   #include <eine_Headerdatei>
2   #include <noch_eine_Headerdatei.h>
3
4   int main()
5   {
6       …
7   }
```



Datentypen und Operatoren (1)

Datentypen bestehen aus der Menge der möglichen Werte

sowie die mit diesen Werten möglichen Operationen.

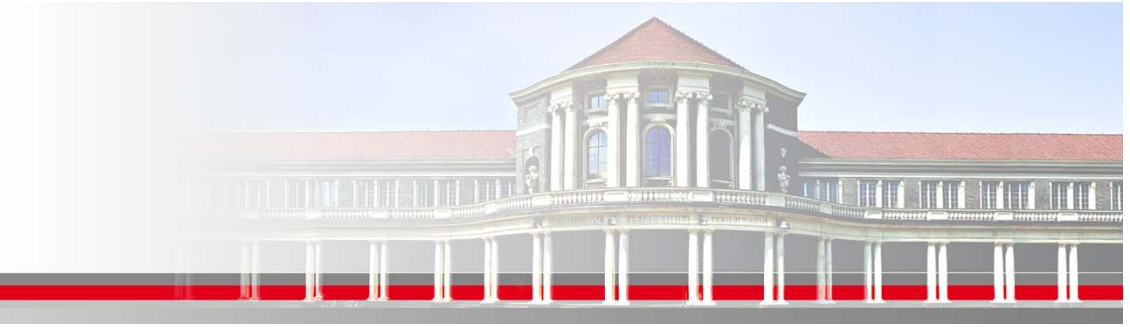
Grunddatentypen: int, char, bool, float, double

Spezifisierung: short, long, signed, unsigned

Zuweisungs-Operatoren: =

Arithmetrische-Operatoren: +, -, /, *, %

Vergleichs-Operatoren: ==, <, >, <=, >=, !=



Datentypen und Operatoren (2)

Deklaration:

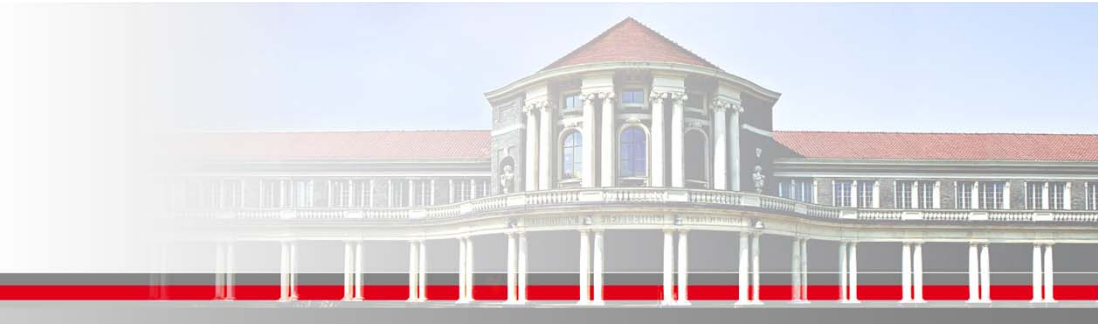
```
int a;
int b = 0;
```

Operationen:

```
a = 12;
a = 5 * b;
```

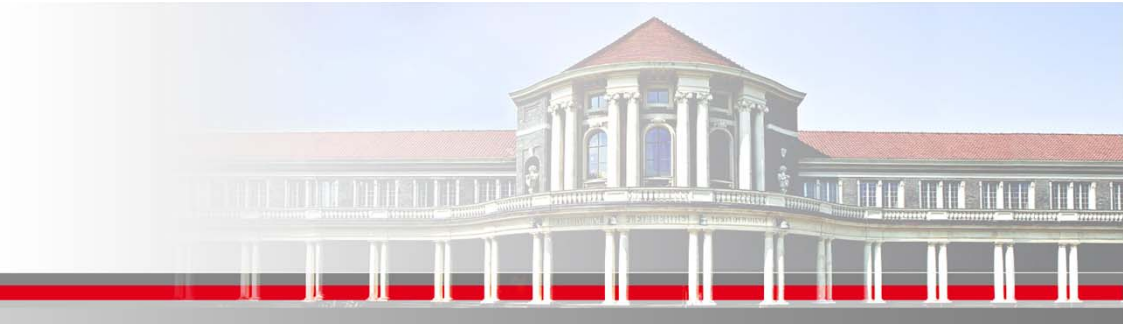
Inkrementieren einer Variable:

```
a = a + 1;
a++;
a += 1;
```



Bitoperationen

unsigned int bit_value = 0x1257;	0001 0010 0101 0111	
unsigned int mask = 0x3122;	0011 0001 0010 0010	
unsigned int b = bit_value | mask;	0011 0011 0111 0111	OR
unsigned int c = bit_value & mask;	0001 0000 0000 0010	AND
unsigned int d = ~mask;	1100 1110 1101 1101	NOT
unsigned int e = bit_value & ~mask;	0000 0010 0101 0101	AND NOT
unsigned int f = bit_value ^ mask;	0010 0011 0111 0101	XOR



Verzweigung

- IF-Verzweigung

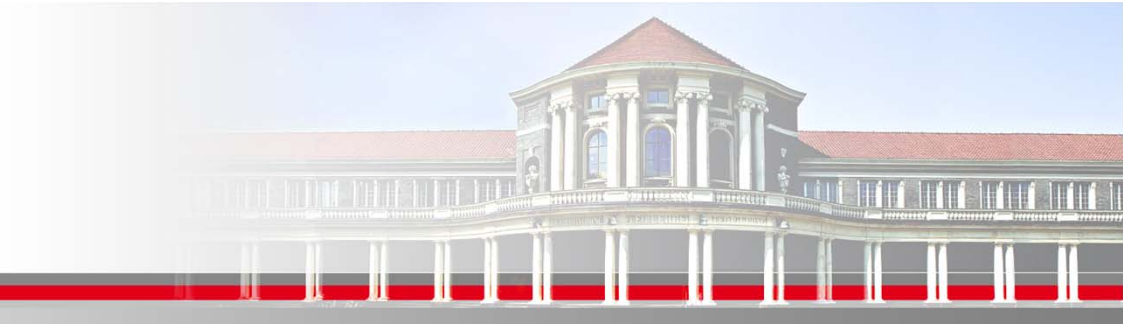
```
if (condition)
  statement
```

- IF-ELSE-Verzweigung

```
if (condition)
  statement
else
  statement
```

- SWITCH-Verzweigung

```
switch (selector)
{
      case label:
            statement
      default:
            statement
}
```



Schleifen

- WHILE-Schleife

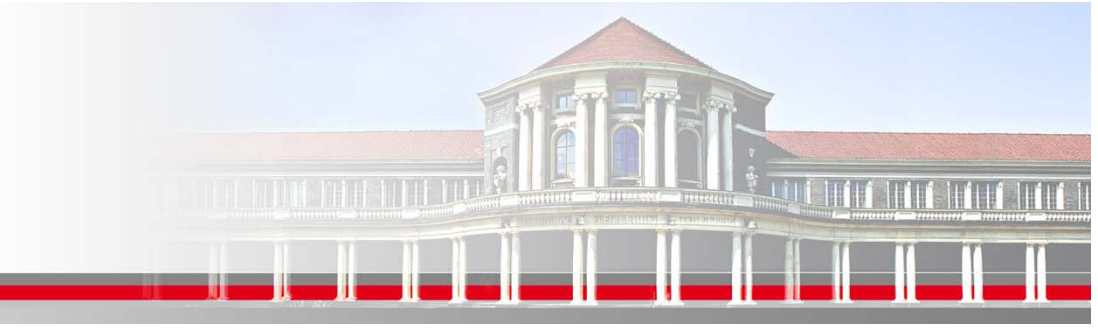
```
while (condition)
  statement
```

- FOR-Schleife

```
for (initializer; condition; update)
  statement
```

- Vorzeitiges verlassen einer Schleife oder einer SWITCH-Verzweigung mittels

```
break;
```



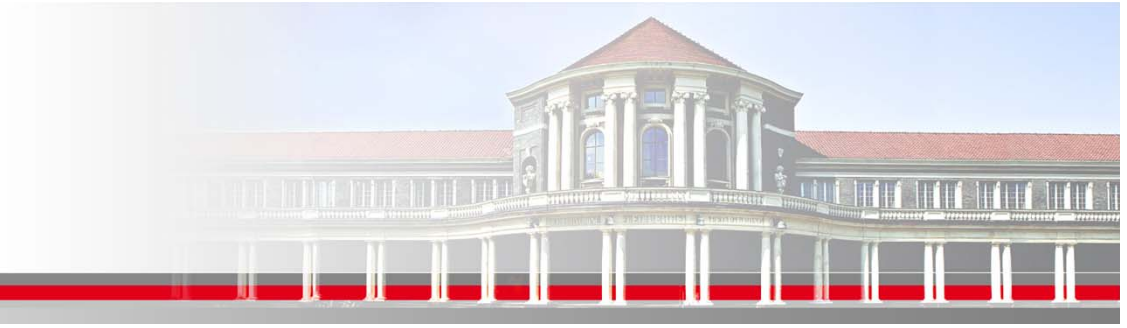
IO

- C++: IO-Stream

```
#include <iostream>
std::cout << „Text“ << std::endl;
std::cin >> variable;
```

- C: Std.-IO

```
#include <stdio.h>
printf(format_description,variables);
scanf(format_description,variables);
```

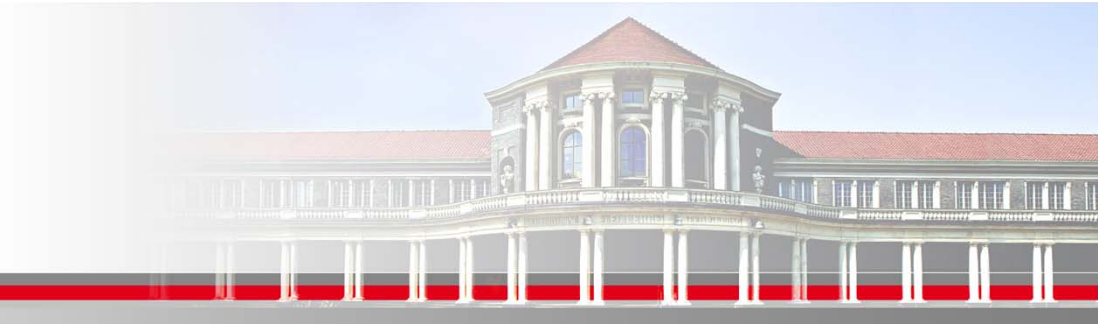


IO - Formatbeschreiber

Integer:

%d	int dezimal
%hd	short dezimal
%ld	long dezimal
%o	int oktal
%ho	short oktal
%lo	long oktal
%x	int hexadezimal
%hx	short hexadezimal
%lx	long hexadezimal

%u	unsigned int dezimal
%hu	unsigned short dezimal
%lu	unsigned long dezimal
%i	int dezimal 12, oktal 012, hexadezimal 0x12
%hi	short dezimal 12, oktal 012, hexadezimal 0x12
%li	long dezimal 12, oktal 012, hexadezimal 0x12



IO - Formatbeschreiber

Float:

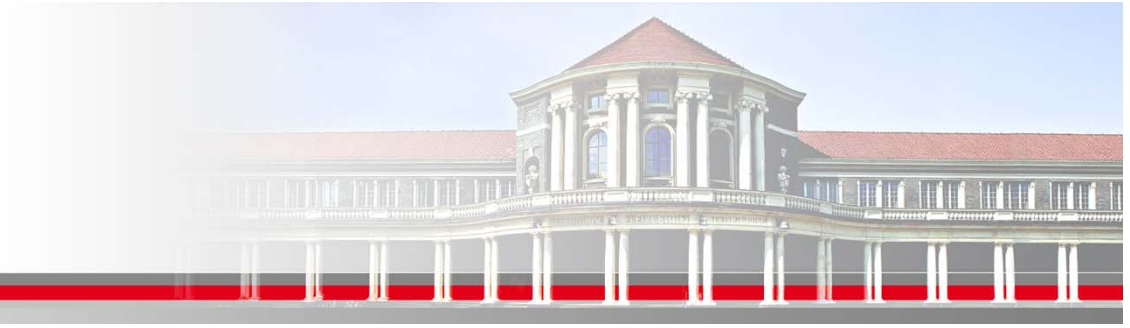
- %f float fixed
- %lf double short fixed
- %LF long double fixed
- %e float scientific
- %le double short scientific
- %Le long double scientific
- %g float fixed und scientific
- %lg double short fixed und scientific
- %Lg long double fixed und scientific

Character:

- %c Zeichen

C-String:

- %s Zeichenkette



Arrays

- Die Deklaration

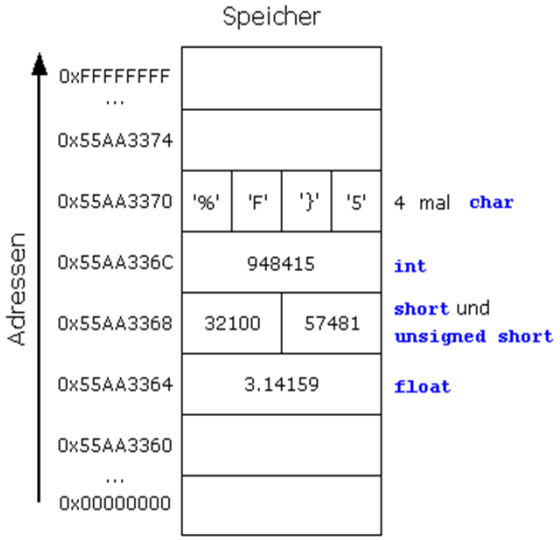
```
int a[10];
```

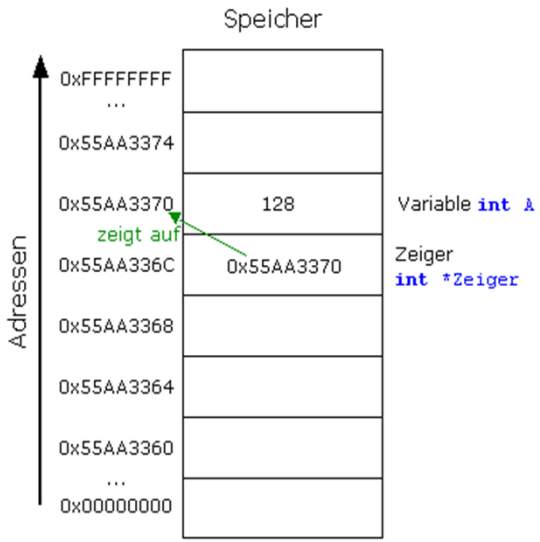
definiert ein Array `a` mit 10 Elementen vom Typ `int`. Auf die einzelnen Elemente kann mit `a[0]` bis `a[9]` zugegriffen werden.

- Auf diese Art lassen sich nur Arrays definieren, deren Größe zur Compilezeit bekannt ist (variable Größe → später)
- Mehrdimensionale Arrays definiert man analog:

```
int a[10][20];
a[5][2] = 19;
```

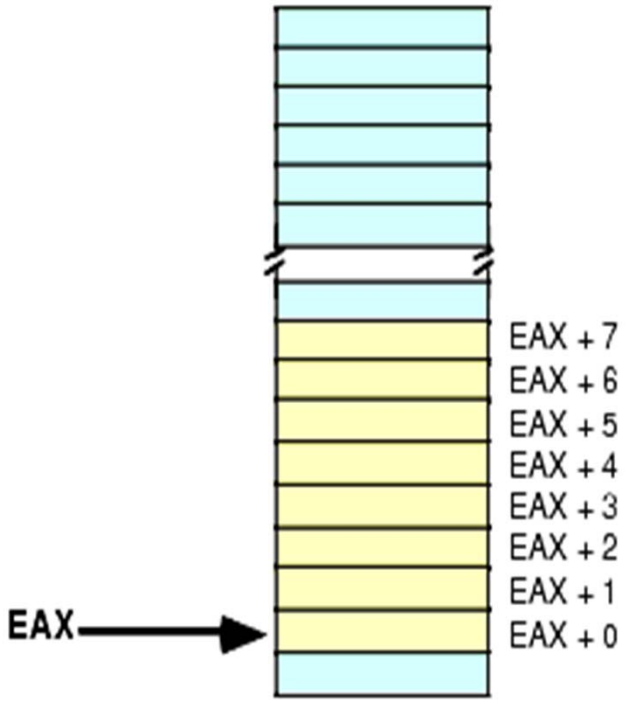
Speicher , Adressen , Zeiger (Pointer)

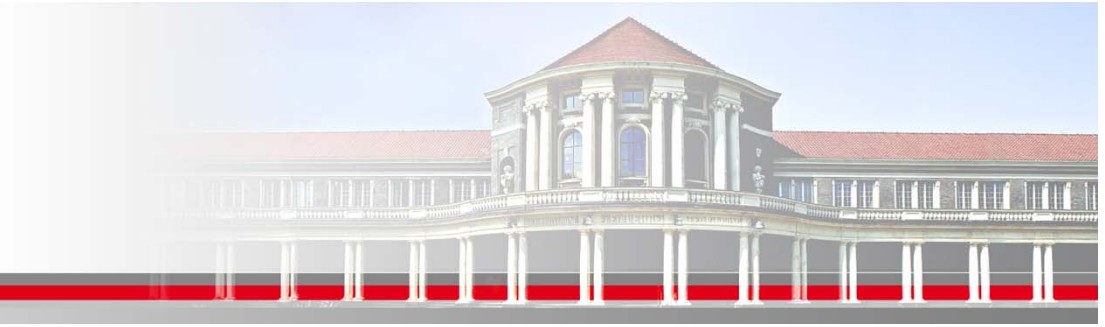




Zeiger: Speicheranforderung eines Arrays

```
int EAX[8];
```

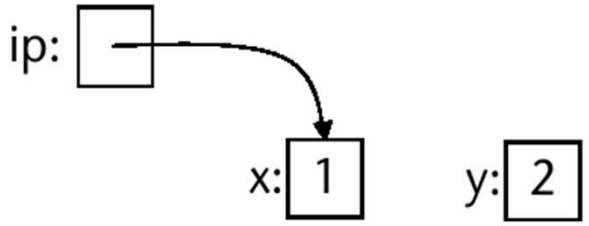




- Für einen beliebigen Datentyp `type` bezeichnet `type*` den Typ "Zeiger auf `type`", d.h. eine Variable vom Typ `type*` kann die *Adresse* eines "Objektes" vom Typ `type` aufnehmen
- Wichtige Operatoren:
  - Adressoperator `&`
  - Dereferenzierungsoperator `*` (Inhaltsoperator)
- Beispiel:

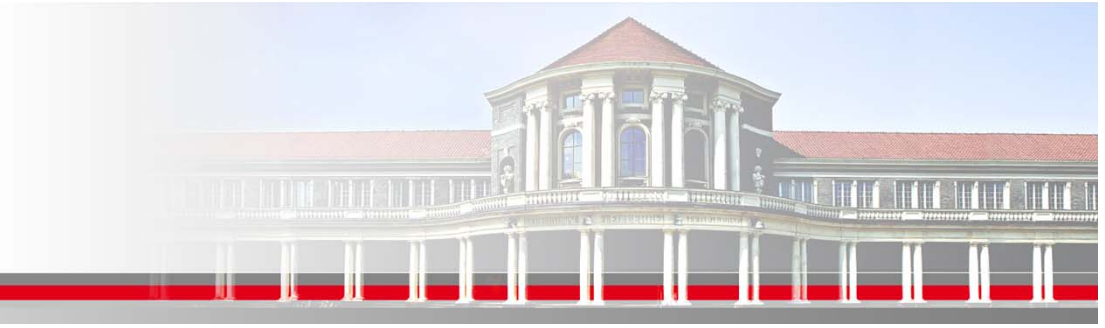
```
int  x = 1;
int  y = 2;
int* ip;        // "Zeiger auf int"

ip  = &x;       // ip enthält Adresse von x
y   = *ip;      // y ist jetzt 1
*ip = 0;        // x ist jetzt 0
ip  = &y;       // ip zeigt jetzt auf y
```

- Illustration:



- Zeigern kann der Wert 0 zugewiesen werden (Null-Pointer). So ist prüfbar, ob ein Zeiger belegt ist oder nicht:

```
int  x  = 1;
int  y  = 2;
int* ip = 0;

if (ip) y = *ip;    // Keine Zuweisung
ip = &x;            // ip enthält Adresse von x
if (ip) y = *ip;    // y = 1
```

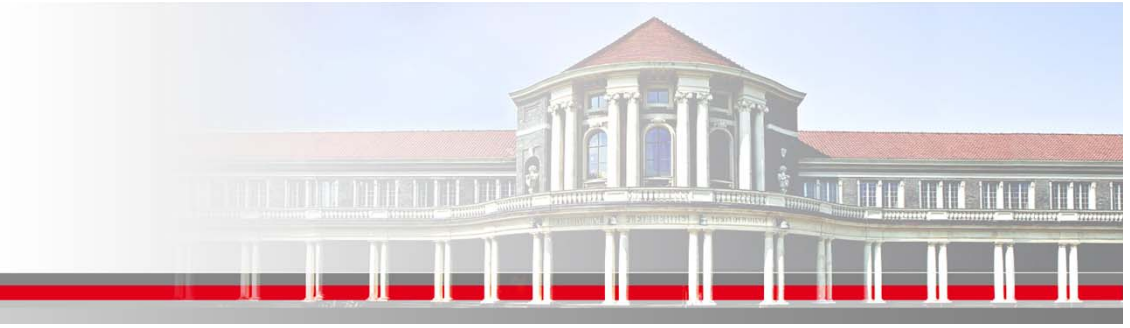


Referenzen

- "Eine Referenz ist ein alternativer Name für ein Objekt"
- Vorstellung: Eine Referenz ist ein (nicht veränderbarer) Zeiger auf ein Objekt, der bei jeder Benutzung dereferenziert wird
- Da eine Referenz immer an ein Objekt gebunden ist, muss man sie zwingend initialisieren
- Beispiel:

```
int  x  = 17;
int& xr = x;

int y = x;      // y = 17
int z = xr;     // z = 17, da xr Synonym für x
```

- Anwendung: Call-by-reference



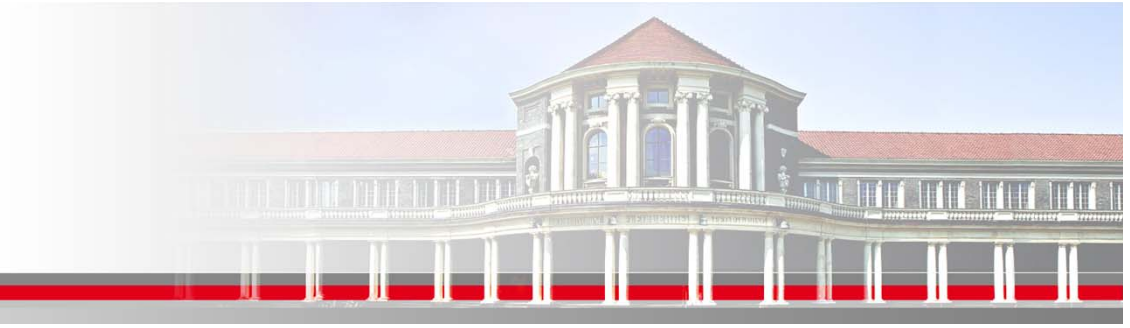
Funktionen

- Funktionen erlauben es Code auszulagern und in kleine, zusammengehörenden Teile zu zerschneiden.

```
rückgabe_type
funktions_name(0_bis_beliebige_anzahl_an_parametern);
```

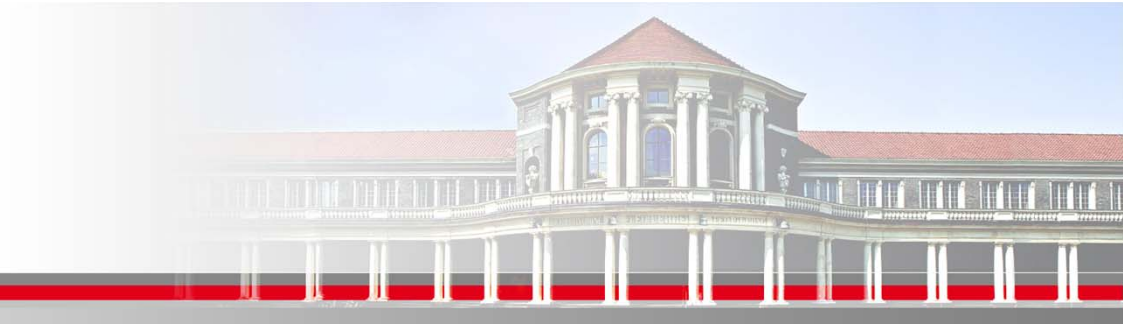
- Beispiel

```
int Add(int lhs, int rhs) //Definition einer Funktion
{
      return lhs + rhs;
}
```



Programmaufbau

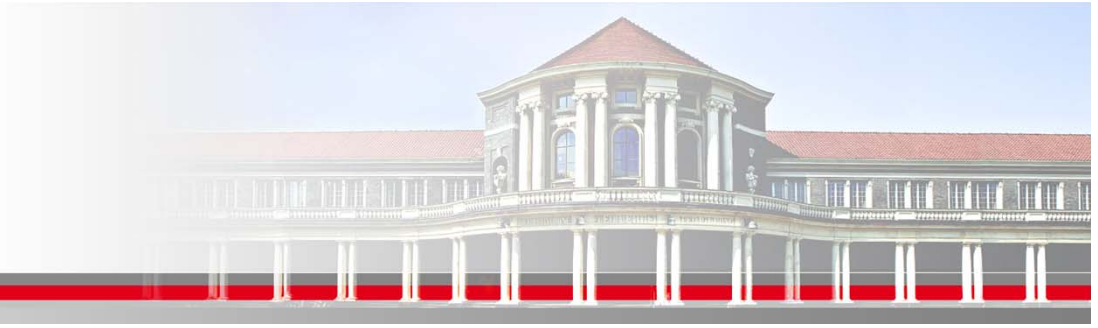
```
#include <stdlib.h>

void print();

int main() {
    print();
    return 0;
}

void print() {
    cout << "Hello world!" << endl;
}
```



Parameterübergabe (1)

- In Java erfolgt die Parameterübergabe durch übergeben von Werten
- In C++ kann die Parameterübergabe erfolgen durch Übergabe von:
  - Einem Wert
  - Einer Referenz
  - Einer konstanten Referenz

```
void f( int iA, int &riB, const int &criC );
```

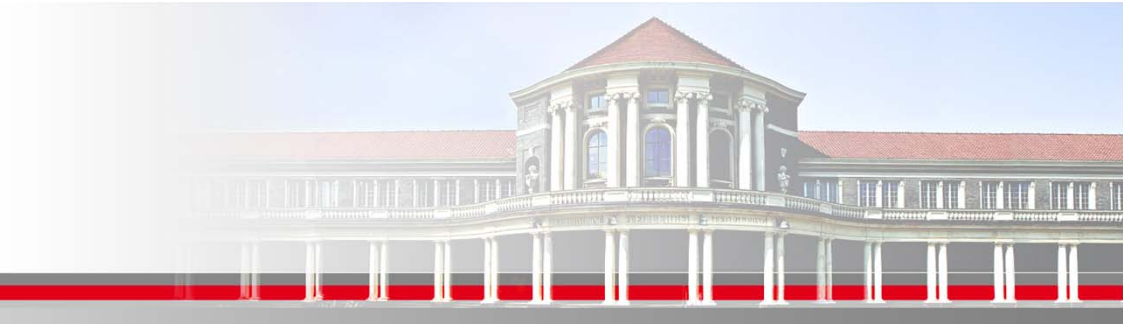


Parameterübergabe (2)

- Erfolg die Übergabe durch einen Wert, wird eine Kopie des Parameters angelegt

```
void f(int iN) { iN++; }
int main() {
    int i_x = 2;
    f(i_x);
    printf("Wert der Variable i_x ist: %d"
,i_x);
}
```

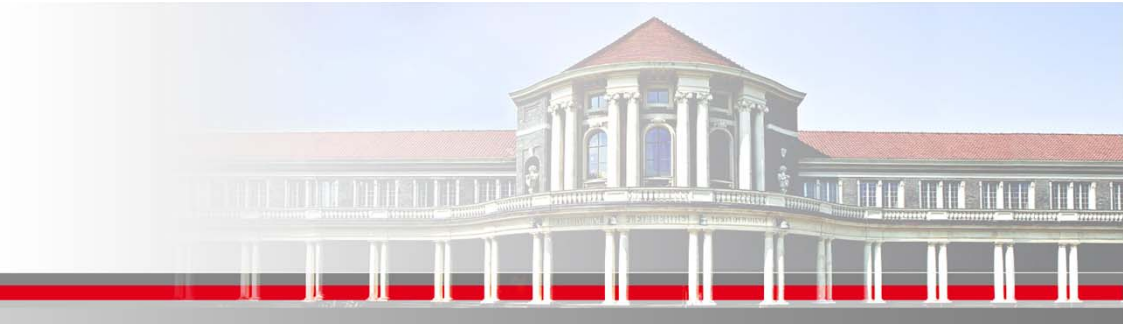
- `f(.)` arbeitet mit einer Kopie (nicht mit original Variable), somit ist die Ausgabe für `x_i` „2“



Parameterübergabe (3)

```
void f(int *piP) {
    *piP = 5;
    piP = NULL;
}
int main() {
    int i_x=2;
    int *pi_q = &i_x;
    f(pi_q); // hier, i_x == 5, aber pi_q != NULL
}
```

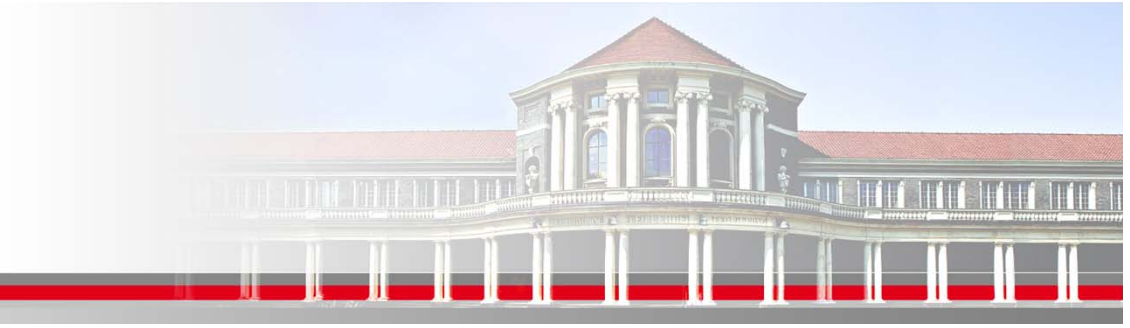
- Der Zeiger wird als ein Wert übergeben, aber das Objekt auf das er verweist kann sich ändern
- Wird auch „call by reference“ genannt



Parameterübergabe (4)

```
void f(int &riN) { riN++; }
int main() {
    int i_x = 2;
    f(i_x);
    cout << i_x;
}
```

- Der Parameter wurde geändert (wie auch bei der Übergabe von Zeigern)
- Eigentlich wurde hier ein Zeiger übergeben (keine Kopie!!!)



C/C++-Exkurs: Parameterübergabe (5)

- Problem: einer Referenz sieht man nicht an, daß sie in der Methode verändert wird!
- Guideline:
  - Referenzparameter immer nur mit **const** verwenden, z.B.

```
void doIt (const int& x);
```

  - Falls (Call-by-reference) Parameter verändert werden soll, dann Pointer verwenden z.B.

```
void doIt (const int* x);
```