Computational Astrophysics

Helge Todt

Astrophysik
Institut für Physik und Astronomie
Universität Potsdam

SoSe 2014

Schriftkonventionen

Bedeutung der verwendeten Schrifttypen (Fonts)

Darstellung	Bedeutung	Beispiel
`xvzf` (Typewriter)	wörtlich einzugebender Text (z.B. Befehle)	`man ls`
argument (kursiv)	Platzhalter für selbst zu ersetzenden Text	`file` *meinedatei*

Ziele und Inhalte

- Vertiefen und Erweitern von Programmierkenntnissen in C/C++
- Einführung in Fortran $\rightarrow$ weite Verbreitung in der Astrophysik
- Behandlung astrophysikalischer Fragen, die Computerhilfe erfordern:
  - Lösen von Bewegungsgleichungen

    $\rightarrow$ Vom Zwei-Körper-Problem zu N-Teilchen-Simulationen
  - Datenanalyse

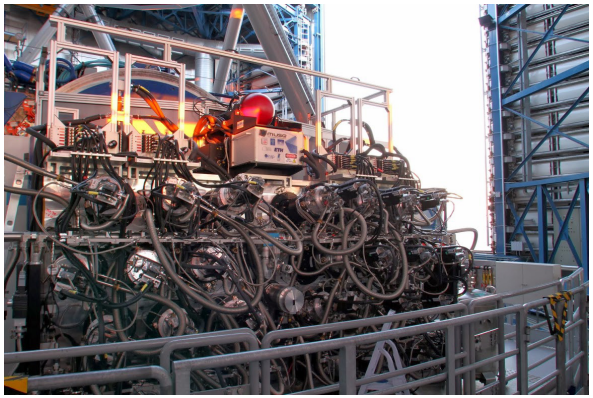
    $\rightarrow$ Datenanalyse und Simulation mit Fortran
  - Simulation von Prozessen

    $\rightarrow$ Monte-Carlo-Simulationen und Strahlungstransport

\+ Einführung in die Parallelsierung (OpenMP, Fortran2003)

Wofür Computer in der Astrophysik?

- Steuerung von Instrumenten/Teleskopen/Satelliten:



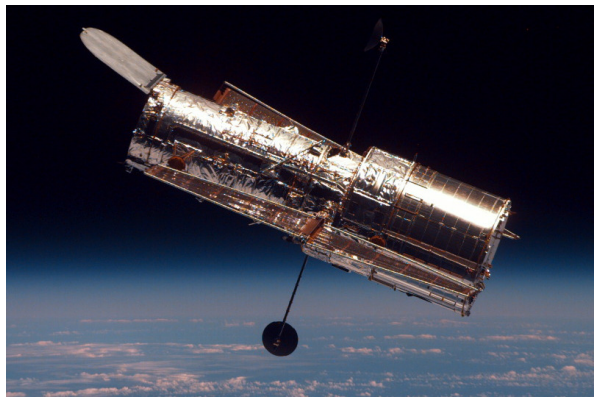
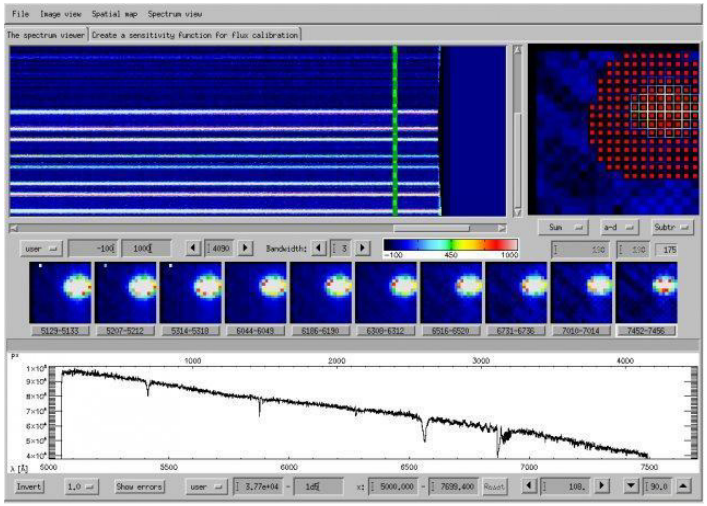
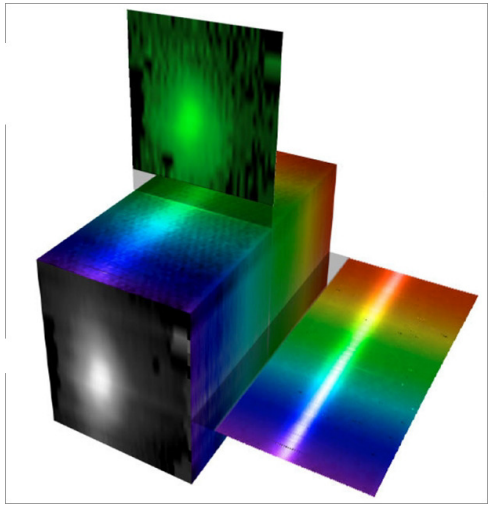


Abbildung: MUSE, VLA, HST

- Datenanalyse / Datenreduktion





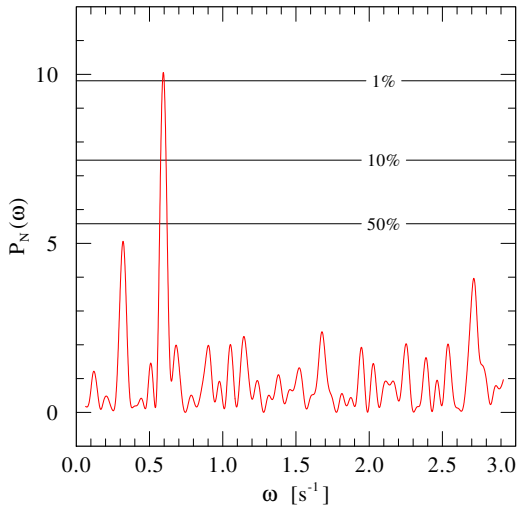
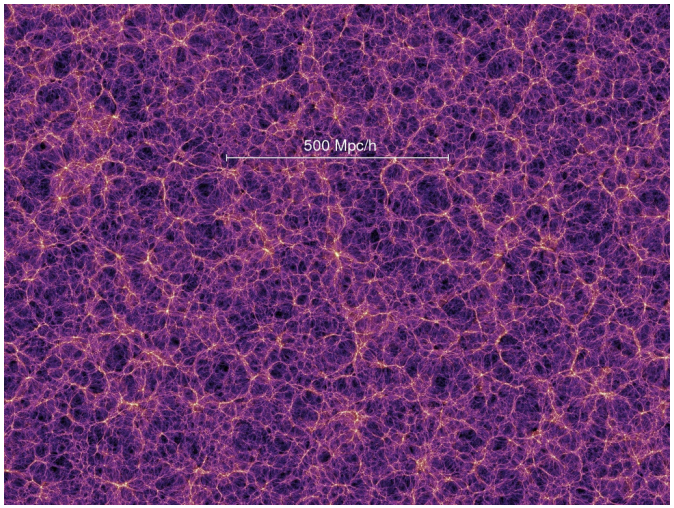
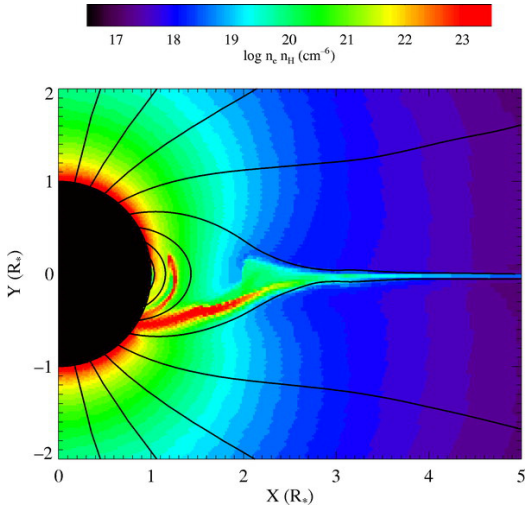


Abbildung: IDL, 3dCube / FITS, Fourieranalyse

- Modellierung / Numerische Simulationen





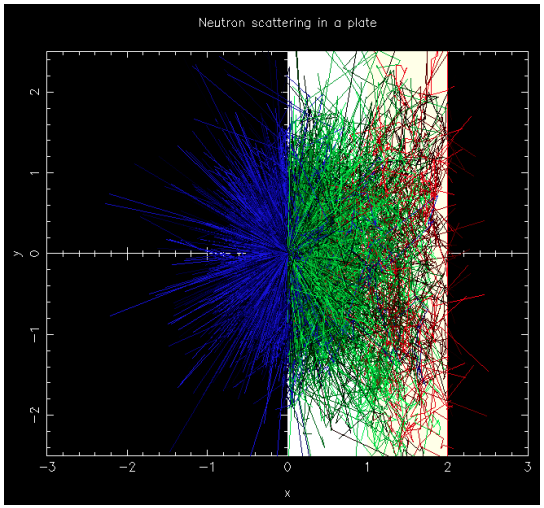


Abbildung: N-Body-Simulationen, Hydro, Monte-Carlo

Programmiersprachen

Man kann z.B. unterscheiden:

Skriptsprachen

- bash, csh $\rightarrow$ Unix-Shell
- Perl, Python
- IRAF, IDL, Midas $\rightarrow$ spezifisch für Datenreduktion in der Astrophysik

Compilersprachen

- C/C++ $\rightarrow$ sehr verbreitet, daher unser Favorit
- Fortran $\rightarrow$ sehr verbreitet in der Astrophysik, besonders für Strahlungstransport

Wiederholung: C/C++ I

- C ist eine *prozedurale* (imperative) Sprache
- C++ ist eine *objektorientierte* Erweiterung von C mit derselben Syntax
- C++ ist durch seine Sprachstrukturen (template, class) $\gg$ C

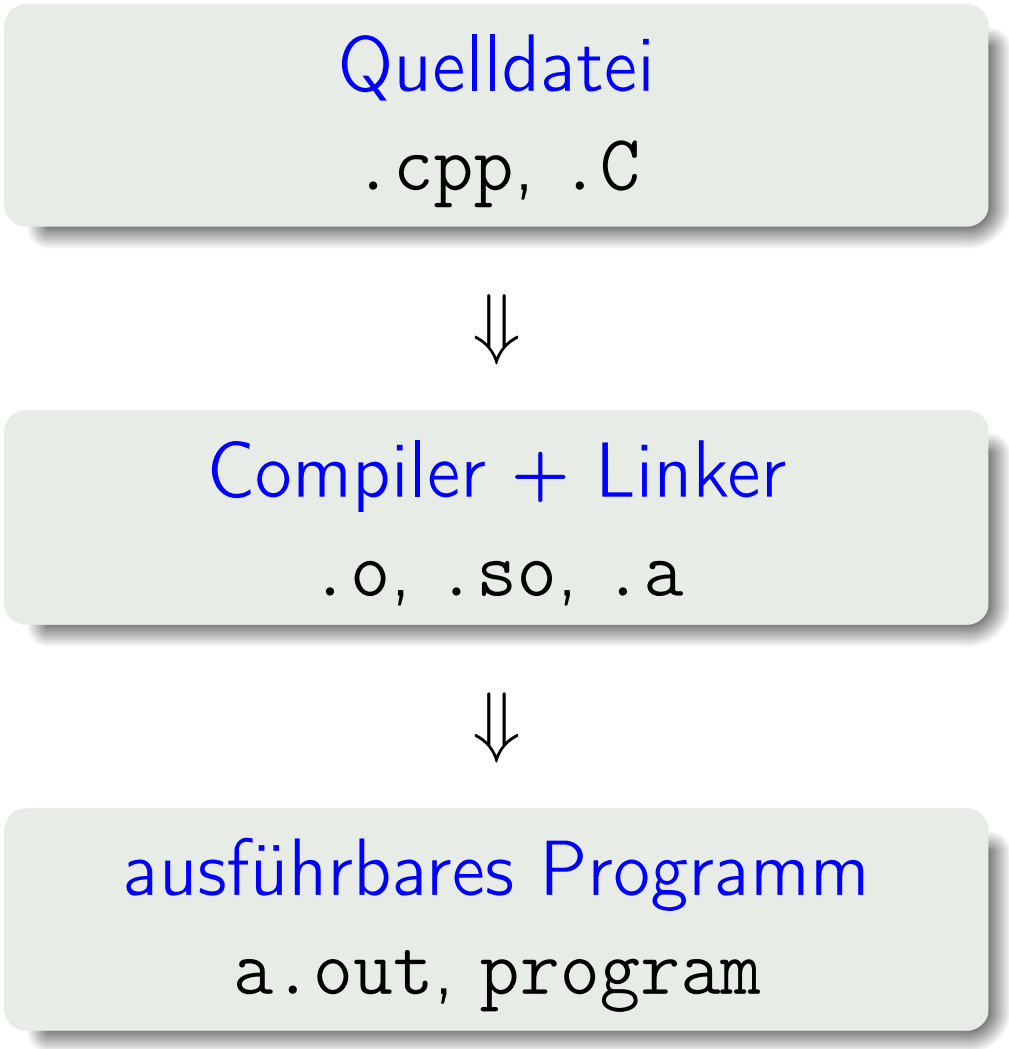
Grundstruktur eines C++-Programms

```
#include <iostream>
using namespace std ;
int main ()
{
    Hier stehen die Anweisungen fuer das Programm ;
    // Kommentar
    return 0 ;
}
```

jede Anweisung wird mit einem Semikolon `;` abgeschlossen

Wiederholung: C/C++ II

Kompilieren eines C++-Programms:



Befehl zum Kompilieren + Linken:

`g++ -o` *program program*`.cpp`

(GNU-Compiler für C++)

- Nur kompilieren, nicht linken:

  `g++ -c` *program*`.cpp`

  erstellt *programm*`.o` (Objectdatei, nicht ausführbar)
- Option `-o` *name* definiert einen Namen für die Datei, die das ausführbare Programm enthält, sonst heißt das Programm `a.out`

  Name des ausführbaren Programms beliebig

Einfaches Programm zur Bildschirmausgabe I

Beispiel: C++ Bildschirmausgabe mittels Streams

```
#include <iostream>

using namespace ::std ;

int main ()
{
    cout << endl << "Hallo, Welt!" << endl ;

    return 0 ; // alles ok

}
```

Einfaches Programm zur Bildschirmausgabe II

- `<iostream>` ... ist eine C++-Programmbibliothek (Ein-/Ausgabe)
- `main()` ... Hauptprogramm (Funktion)
- `return 0` ... gibt den Rückgabewert 0 an main (alles in Ordnung)
- der Quelltext kann frei formatiert werden, d.h. es können beliebig viele Leerzeichen und Leerzeilen eingefügt werden $\rightarrow$ z.B. Einrückungen zur optischen Gliederung
- Kommentare werden mit `//` eingeleitet - alles rechts davon wird ignoriert,
  C kennt nur `/* kommentar */` zum mehrzeiligen (Aus-)Kommentieren
- `cout` ... Bildschirmausgabe (C++)
- `<<` ... Ausgabeoperator (C++)
- Zeichenkette (String) `"Hallo, Welt!"` muss in Anführungszeichen gesetzt werden
- `endl` ... Steuerzeichen: Zeilenumbruch (C++)
- ein Block sind mehrere Anweisungen, die mit geschweiften Klammern zusammengefasst werden

Funktionen I

C/C++ ist eine prozedurale Sprache.
Die Prozeduren in C/C++ heißen *Funktionen*.

- Hauptprogramm: Funktion mit Namen: `main(){}`
- jede Funktion hat einen Rückgabetyp, z.B.: `int main (){}`
- Funktionen können Argumente übergeben bekommen, z.B.:
  `int main (int argc, char *argv[]){}`
- Funktionen müssen vor ihrem Aufrauf im Hauptprogramm *deklariert* (namentlich genannt),
  z.B. `void vertauschen(int &a, int &b) ;`
  oder über eine Headerdatei eingebunden werden:
  `#include <cmath>`
- in geschweiften Klammern `{ }`, dem Funktionskörper, steht die *Definition* der Funktion (was soll sie wie tun), z.B.:
  `int main () { return 0 ; }`

Funktionen II

Beispiel

```
#include <iostream>
using namespace std ;

float cube(float x) ;

int main()
{
  float x = 4. ;
  cout  << cube(x) << endl ;
  return 0 ;
}

float cube(float x)
{
  return x*x*x ;
}
```

Variablen

- eine Variable ist ein Stück Speicher
- C/C++ nutzt eine statische, explizite Typisierung

Wir unterscheiden nach Sichtbarkeit:

- Globale Variablen $\rightarrow$ außerhalb von Funktionen und vor `main` deklariert
- lokale Variablen $\rightarrow$ in einer Funktion oder einem Block `{ }` deklariert, nur dort sichtbar

. . . nach Datentypen $\rightarrow$ einfache Datentypen:

- `int` $\rightarrow$ Integer, Ganzzahlen, z.B. `int n = 3 ;`
- `float` $\rightarrow$ Gleitkommazahlen, z.B. `float x = 3.14, y = 1.2E-4 ;`
- `char` $\rightarrow$ Character, Zeichen, z.B. `char zeichen ;`
- `bool` $\rightarrow$ Logische Variablen, z.B. `bool btest = true ;`

Integer-Datentypen

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \mathrel{\widehat{=}} \boxed{1}\boxed{1}\boxed{0}\boxed{1} \quad \text{(binär)}$$

`int`	Compiler reserviert 32 Bit (= 4 Byte) Speicher 1 Bit Vorzeichen und $2^{31} = 2\,147\,483\,648$ Werte (inkl. 0): $\rightarrow$ Wertebereich: `int` $= -2\,147\,483\,648 \ldots + 2\,147\,483\,647$
`unsigned int`	32 Bit, kein Vorzeichenbit $\rightarrow 2^{32}$ Werte (inkl. 0) `unsigned int` $= 0 \ldots 4\,294\,967\,295$
`long`	auf 64Bit-Systemen: 64 Bit (= 8 Byte), 1 Bit Vorzeichen: $-9.2 \times 10^{18} \ldots 9.2 \times 10^{18}$
`unsigned long`	64 Bit ohne Vorzeichen: $0 \ldots 1.8 \times 10^{19}$

Gleitkomma-Datentypen

Geregelt durch Norm IEEE 754 : $x = s \cdot m \cdot b^e$
Vorzeichen s, Basis $b = 2$ und normalisierte Mantisse m, Bias:

- einfache Genauigkeit (32 Bit): Exponent 8 Bit, Mantisse 23 Bit

$$-126 \leq e \leq 127 \text{ (Basis 2)}$$

$$\rightarrow \approx 10^{-45} \ldots 10^{38}$$

Dezimalstellen: 7-8 $(= \log 2^{23+1} = 24 \log 2)$

- für 64 Bit – `double`: Exponent 11 Bit, Mantisse 52 Bit

$$-1022 \leq e \leq 1023 \text{ (Basis 2)}$$

$$\rightarrow \approx 10^{-324} \ldots 10^{308}$$

Dezimalstellen: 15-16 $(= \log 2^{52+1})$

Logische Variablen

```
bool b ;
```

ebenfalls einfacher Datentyp, kann nur zwei verschiedene Werte annehmen:

```
bool btest, bdo ;
btest = true ; // = 1
bdot = false ; // = 0
```

allerdings auch:

```
btest = 0. ; // = false
btest = -1.3E-5 // = true
```

Ausgabe mittels `cout` ergibt `0` bzw. `1`.

Zeichenvariablen

```
char buchstabe ;
```

sind als Ganzzahlen kodiert:

```
char Zeichen = 'A' ;
char Zeichen = 65 ;
```

stehen jeweils für dasselbe Zeichen (ASCII-Code)

Zuweisungen von Zeichenliteralen an Zeichenvariablen erfordern einfache Anführungszeichen ' :

```
char ja = 'Y' ;
```

Felder (Arrays) in C/C++

Statische Feldvereinbarung für ein eindimensionales Feld vom Typ `double`:

`double a[5] ;` eindimensionales Feld mit 5 Double-Typ-Elementen (z.B. für Vektoren)

Zugriff auf einzelne Element:

```
total = a[0] + a[1] + a[2] + a[3] + a[4] ;
```

Achtung:

Der Laufindex beginnt in C/C++ immer bei 0 und läuft in diesem Beispiel dann bis 4, d.h. das letzte Element ist `a[4]`

Eine beliebte Fehlerquelle in C/C++ !!!

Zweidimensionale Felder

`int a[i][j]` ... statisches zweidimensionales Feld, z.B. für Matrizen.

`i` ist der Index für die Zeilen,
`j` für die Spalten.

$$\text{z.B. } a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

In C/C++ läuft der zweite Index zuerst, im Speicher liegen die Einträge von `a[2][3]` so hintereinander:

`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[1][0]`	`a[1][1]`	`a[1][2]`
1	2	3	4	5	6

(row-major order)

Initialisieren von Arrays

Ein Array kann mithilfe von geschweiften Klammern intialisiert werden:

```
int feld[5] = {0, 1, 2, 3, 4} ;

short acker[] = {0, 1} ; // Array acker wird
                         // automatisch dimensioniert

float x[77] = {0} ; // alle Werte auf 0 setzen
```

Zeichenketten (Strings)

Es gibt in der Sprache C keine Stringvariable. Strings werden deshalb in eindimensionale Felder geschrieben:

```
char text[6] = "Hallo" ;
```

Die Stringliteralkonstante `"Hallo"` besteht aus 5 druckbaren Zeichen und wird vom Compiler automatisch mit dem Null-Zeichen `\0` abgeschlossen, d.h. das Feld muss 6 Zeichen lang sein. Man beachte dabei die doppelten Anführungszeichen.

Beispiel

```
char text[80] ;
cout << endl << "Bitte einen String eingeben:" ;
cin >> text ;
cout << "Sie haben" << text << "eingegeben\". << endl ;
```

Pointer I

Zeigervariablen - kurz: Zeiger oder Pointer - ermöglichen einen direkten Zugriff (d.h. nicht über den Namen) auf die Variable.

Deklaration eines Pointers

```
int    *pa  ; // Zeiger auf int
float  *px  ; // Zeiger auf float
int   **ppb ; // Zeiger auf Zeiger auf int
```

`*` ... heißt hier Verweisoperator und bedeutet "Inhalt von".

Pointer II

Ein Pointer ist eine Variable, die eine Adresse enthält, d.h. sie zeigt auf einen Speicherplatz.
Wie jede Variable besitzt auch eine Zeigervariable einen bestimmten Typ.
Der Werte der Speicherzelle, auf den die Zeigervariable zeigt, muss vom vereinbarten Typ sein.

Adresse	Inhalt	Variable
1000	0.5	x
1004	42	n
1008	3.141...	d
1012	...5926	
1016	H E Y !	gruss
1020	1000	px
1024	1008	pd
1028	1004	pn
1032	1016	pgruss
1036	1028	pp

Pointer III

Pointer müssen vor der Verwendung stets initialisiert werden!

Initialisierung von Pointern

```
int  *pa ; // Zeiger auf int
int b ;    // int
pa = &b ;  // Zuweisung der Adresse von b an a
```

Das Zeichen & heißt Adressoperator ("Adresse von")
(nicht zu Verwechseln mit der Referenz `int &i = b ;`).

Deklaration und Initialisierung

```
int b ;
int *pa = &b ;
```

→ Inhalt von `pa` = Adresse von `b`

Pointer IV

Mit dem Dereferenzierungsoperator `*` kann auf den Wert der Variablen `b` zugegriffen werden, man sagt, Pointer `pa` wird dereferenziert:

Derefrenzierung eines Pointers

```
int b, *pa = &b ;
*pa = 5 ;
```

Die Speicherzelle, auf die `pa` zeigt, enthält nun den Wert `5`, dies ist nun auch der Werte der Variablen `b`.

```
cout << b << endl ; // ergibt 5
cout << pa << endl ; // z.B. 0x7fff5fbff75c
```

Pointer V

Nochmal:

Pointerdeklaration:

```
float *pz, a = 2.1 ;
```

Pointerinitialisierung:

```
pz = &a ;
```

Resultat - Ausgabe:

```
cout << "Adresse der Variablen a (Inhalt von pz): "
     << pz << endl ;
cout << "Inhalt der Variablen a: "
     << *pz << endl ;
*pz = 5.2 ; // Wert von a ändern
```

Referenzen

```
int &n = m ;
m2 = n + m ;
```

- Eine Referenz ist ein neuer Name, ein Alias für eine Variable. Dadurch kann man ein und denselben Speicherplatz (Variable) unter zwei verschiedenen Namen im Programm ansprechen. Jede Modifikation der Referenz ist eine Modifikation der Variablen selbst - und umgekehrt.
- Referenzen werden mit dem &- Zeichen (Referenzoperator) deklariert und müssen sofort initialisiert werden:

```
int a ;
int &b = a ;
```

- Diese Initialisierung kann im Programm nie wieder geändert werden!

Übergabe von Variablen an Funktionen I

Struktur von Funktionen - Definition

rückgabetyp name (arg1, ...) { ... }

Bsp.: `int main (int argc, char *argv[]) { }`

- in runden Klammern stehen die Funktionsargumente, sog. formale Parameter
- beim Aufruf der Funktionen werden von den übergebenen Variablen Kopien erstellt $\rightarrow$ *Call-by-value* (Parameterübergabe per Wert)

```
setzero (float x) { x = 0. ; }
int main () {
    float y = 3. ;
    setzero (y) ;
    cout << y ; // gibt 3. aus }
```

Übergabe von Variablen an Funktionen II

Parameterübergabe per Wert

Vorteile:

- die übergebenen Variablen können nicht unabsichtlich in der Funktion verändert werden

Nachteile:

- die übergebenen Variablen können auch nicht absichtlich verändert werden
- bei jedem Funktionsaufruf müssen die Werte *kopiert* werden
  $\rightarrow$ zusätzlicher Zeit-Overhead
  (Ausnahme: Ist der Parameter ein Array, dann wird nur die *Startadresse* übergeben $\rightarrow$ Pointer)

Funktionsstruktur: Call by reference

```
void swap(int &a, int &b) ;
```

Übergabe der Argumente als Referenzen:

Die übergebenen Variablen werden in der Funktion swap geändert und behalten nun aber diesen Wert, auch nach Verlassen von swap.

```
void vertausche (int &a, int &b) {
 int t = a ;  a = b ; b = t ; }
```

Damit können wir nun beliebig viele Werte aus einer Funktion zurückgeben.

Hinweis: Mittels des Schüsselwortes `const` kann verhindert werden, dass der übergebene Parameter in der Funktion verändert wird:
`sum (int const &a, int const &b) ;`

Parameterübergabe an Funktionen mit Zeigern

Eine Funktion zum Vertauschen zweier `int`-Variablen lässt sich mittels Pointern auch so schreiben:

```
void swap(int *a, int *b) // Pointer als formale Parameter
     {
        int tmp ;
        tmp = *a ; *a = *b ; *b = tmp ;
     }
```

Aufruf in `main()`:

```
  swap (&x, &y) ;  // Adressen (!) von x und y
                   // werden übergeben
```

Übergabe von Arrays Funktionsaufruf

Im Unterschied zu (skalaren) Variablen, werden Arrays durch
`myfunc ( float x[] )`
automatisch per Adresse (Zeiger) übergeben.

Pointer und Referenzen

Pointervariablen

- speichern Adressen
- müssen dereferenziert werden
- können im Programm immer wieder anders initialisiert werden (auf eine andere Variable des korrekten Typs zeigen)

Referenzen

- sind Aliasnamen für andere Variablen,
- sie werden einfach mit ihrem Namen angesprochen (keine Dereferenzierung)
- die (notwendige!) Eingangsinitialisierung darf nie geändert werden

Struct und class - Eigene Datentypen definieren I

Neben den elementaren Datentypen gibt es noch viele weitere Datentypen, die selbst definiert werden können:

struct

```
struct complex
  {
    float re ;
    float im ;
  } ;
```

Obiges Beispiel definiert einen Datentypen `complex`, der als *Membervariablen* einen Real- und einen Imagiärteil hat.

Struct und class - Eigene Datentypen definieren II

Strukturen kann man sich als eine Sammlung von Variablen vorstellen.

struct

```
struct element
  {
     char symbol[3] ;
     unsigned short ordnungszahl ;
     float atomgewicht ;
  } ;
```

Diese Datentypen können dann wie andere auch benutzt werden:

Deklaration von struct-Objekten

```
complex z, c ;
element helium ;
```

Struct und class - Eigene Datentypen definieren III

Die so deklarierten konkreten Strukturen nennt man *Instanzen* oder *Objekte* ($\rightarrow$ Objektorientierte Programmierung) einer Klasse (Struktur).

Deklaration und Initialisierung

```
complex z = {1.1 , 2.2} ;
element neon = {"Ne", 10, 20.18} ;
```

Der Zugriff auf die *Membervariablen* erfolgt mittels des *Member-Selection-Operators* . (Punkt):

Zugriff auf Member

```
realteil = z.re ;
neon.ordnungszahl = 10 ;
```

Struct und class - Eigene Datentypen definieren IV

Man kann in der Struktur auch Funktionen (sog. Methoden) definieren:

Memberfunktionen

```
struct complex
  {
     ...
     float betrag ()
       {
          return (sqrt(re*re + im*im)) ;
       }
  } ;
  complex c = {2., 4.} ;
  cout << c.betrag() << endl ;
```

Der Aufruf der *Memberfunktion* erfolgt wieder mit dem `.`, die Funktion ist mit dem Objekt assoziiert.

Schreiben von Daten in eine Datei I

Ausgabe mittels Bibliothek `fstream`:

1. `#include <fstream>`
2. Objekt der Klasse `ofstream` anlegen:

   ```
   ofstream dateiout ;
   ```

3. Methode `open` der Klasse `ofstream`:

   ```
   dateiout.open("grafik.ps") ;
   ```

4. Einlesen der Daten: z.B.
   `dateiout << x ;`
5. Schließen mit Methode `close`:

   ```
   dateiout.close() ;
   ```

Kontrollstrukturen - for-Schleifen I

Kontrollstrukturen steuern den Programmablauf, indem sie bestimmte Anweisungen wiederholen (Schleifen) oder in verschiedene Programmabschnitte verzweigen (bedingt/unbedingt).

Wiederholte Ausführung eines Befehls/Blocks:

for-Schleifen

```
for (int k = 0 ; k < 6 ; k++ ) sum = sum + 7 ;

for (float x = 0.7 ; x < 17.2 ; x = x + 0.3)
{
    y = a * x + b ;
    cout << x << " " << y << endl;
}
```

Kontrollstrukturen - for-Schleifen II

Struktur des `for`-Schleifenkopfes:

Es gibt (bis zu) drei Argumente, jeweils mit Semikolon getrennt:

1. Intialisierung der Schleifenvariablen, ggf. Deklaration, z.B.:
   `int k = 0 ;` †
   → wird vor dem *ersten* Schleifendurchlauf ausgeführt
2. Abbruchbedingung für Schleife, i.d.R. mittels arithmetischen Vergleichs für Schleifenvariable, z.B.
   `k < 10 ;`
   wird *vor jedem* Schleifendurchlauf geprüft
3. Ausdruck: Inkrementierung/Dekrementierung der Schleifenvariable, z.B.
   `k++` oder `k--` oder `k += 3`
   wird *nach jedem* Schleifendurchlauf ausgeführt

† interessanterweise auch: `int k = 0, j = 1;`

Inkrementierungsoperatoren

```
summe += a
```
$\rightarrow$ `summe = summe + a`

```
x++
```
$\rightarrow$ `x = x + 1` (Inkrementoperator)

```
x--
```
$\rightarrow$ `x = x - 1` (Dekrementoperator)

Logische Operatoren I - Vergleiche

→ geben entweder `true` oder `false` zurück:

`a > b`	größer
`a >= b`	größer-gleich
`a == b`	gleich
`a != b`	ungleich
`a <= b`	kleiner-gleich
`a < b`	kleiner

Vorsicht!

Der exakte Vergleich == sollte wegen der begrenzten Darstellungsgenauigkeit bei Gleitkomma-Datentypen vermieden werden.

Schleifen I

Darüber hinaus gibt es auch:

while-Schleifen

```
while (x < 0.) x = x + 2. ;

do x = x + 2. ;  // do-Schleife wird mind. einmal
while (x < 0.) ; // durchlaufen
```

Befehle zur Schleifensteuerung

```
break ; // stoppt Schleifenausfuehrung
continue ; // springt zum naechsten Schleifendurchlauf
```

Schleifen II

In C/C++: keine echten Zählschleifen

→ Schleifenvariablen (Zähler, Grenzen) können auch im Schleifenkörper geändert werden

langsam, schlechte Optimierbarkeit für Compiler/Prozessor

Empfehlung: *lokale* Schleifenvariablen

→ Deklaration im Schleifenkopf
→ Sichtbarkeit nur im Schleifenkörper

Bedingte Ausführung mittels `if`:

```
if (z != 1.0) k = k + 1 ;
```

Entscheidungen/Verzweigungen

```
if (a == 0) cout << "Ergebnis" ; // Einzeiler

if (a == 0) a = x2 ; // Verzweigungen
if else (a > 1)
{
            a = x1 ;
}
else        a = x3 ;
```

Falls die Entscheidungsvariable nur diskrete Werte annimmt (z.B. `int`, `char`, kein Gleitkomma!), können Bedingungen alternativ auch mittels `switch/case` formuliert werden:

Verzweigungen II

```
switch (Ausdruck)
{
     case Wert1 : Anweisung ; break ;
     case Wert2 : Anweisung1 ;
                  Anweisung2 ; break ;
     default    : Anweisung ;
 }
```

Achtung!

Jeder case-Anweisungsblock sollte mit einem `break` abgeschlossen werden, ansonsten wird automatisch der nächste case-Anweisungsblock ausgeführt.

Beispiel: switch

```
int k ;
cout << "Bitte Zahl eingeben, 0 oder 1: " ;
cin >> k ;
switch (k)
{
 case    0 : cout << "Pessimist" << endl ; break ;
 case    1 : cout << "Optimist"  << endl ; break ;
 default   : cout << "Neutraler" << endl ;
}
```

Zusammenfassung

Häufige Fehler in C/C++:

- vergessenes Semikolon `;`
- fehlerhafte Dimensionierung/Zugriff auf Arrays
  `int m[4] ; imax = m[4] ;` → `imax = m[3] ;`
- falscher Datentyp bei Befehlen/Funktionsaufrufen
  `float x ; ... switch (x)`
  `void swap (int *i, int *j) ; ... swap(n,m) ;`
- Verwechseln des Zuweisungsoperators `=` mit dem Vergleichsoperator `==`
  `if(i = j)` → `if(i == j)`
- vergessene Funktionsklammern bei argumentlosen Funktionen
  `clear ;` → `clear();`
- uneindeutige Ausdrücke
  `a[n] = n++ ;` → `a[n] = n ;` oder `a[n+1] = n ;`