Programmieren in C/C++

Literatur zu C++

- C++ kurz & gut; Kyle Loudon, Matthias Kalle Dalheimer
- C++ programmieren im Klartext; Marko Meyer
- C++ – kurzgefasst; Rainer Krienke
- Computing for Scientists / Principles of Programming with Fortran 90 and C++; R.J. Barlow, A.R. Barnett
- The C++ Programming Language; Bjarne Stroustrup

Die wichtigsten Regeln I

Regel 1

Verständlichkeit ist die höchste Tugend der Programmierung.

- *sprechende*, eindeutige Variablennamen
- Optische Strukturierung des Quelltextes, z.B. durch Einrückungen und Leerzeilen
- Kommentare!!! Am Anfang des Programms:
  - Zweck/Bedienung des Programms
  - Name des Programmierers
  - Datum, Änderungen gegenüber früheren Versionen

  Kommentare im Quelltext
  - Eindeutige, verständliche Kommentare
  - Kommentare knapp halten

Aufgabe 5.1 Regeln der Programmierung

1. Welche Variablennamen sind zu bevorzugen?

   ```
   x, frequenz, strecke, i, for, anzahl_wuerfel
   ```

2. Welche der folgenden Kommentare sind eher ungeeignet?

   ```
   // Programm euler1 berechnet e mittels Reihe
   ```

   ```
   // Autor: htodt, erstellt am 1.4.
   ```

   ```
   // Boltzmannkonstante in erg/deg
   ```

   ```
   /* Schleife ueber alle i, es wird jeweils i
      auf s aufaddiert */
   ```

   ```
   // Folgender Block ist ein Programmiertrick
   ```

Regel 2

Benutzerfreundlichkeit ist der zweitwichtigste Aspekt der Programmierung.

Eingabe – Prüfen – Rückmeldung

- *Spezifizieren der Eingabe (Prompt):*
  Was soll der Nutzer eingeben?
  Bsp.: `Eingabe des Kreisradius in cm:`
- Nutzereingaben auf Zulässigkeit *prüfen*,
  z.B. Datentyp, Wertebereich
- *Rückmeldung an Nutzer* bei Fehleingaben:
  Warum ist Eingabe falsch?
  Bsp.: `Eingegebener Radius ist negative Zahl.`

Allgemeine Programmiertipps

1. Formulieren Sie die *logische Struktur* des Problems mit eigenen Worten.
2. Zerlegen Sie das Problem in mehrere *Teilprobleme*.
3. Fügen Sie möglichst viele sinnvolle *Kommentare* in Ihr Programm ein.
4. Drucken Sie zur *Fehlersuche* den Inhalt möglichst vieler Variablen aus.
5. Speichern Sie Ihre Dateien oft ab, *sichern* Sie Ihre Dateien auf verschiedenen Medien.

Die Grundstruktur eines C++-Programms I

```
#include <iostream>

using namespace std ;

int main () {

    Hier stehen die Anweisungen fuer das Programm ;

    // Kommentar

    return 0 ;
}
```

Die Grundstruktur eines C++-Programms II

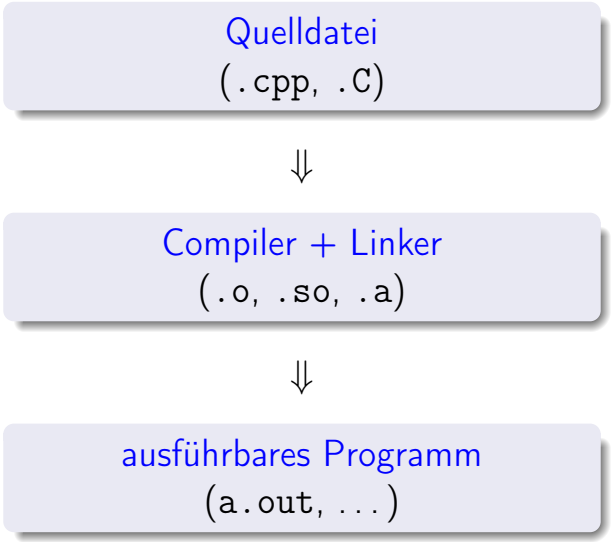
- `<iostream>` ... ist eine C++-Programmbibliothek (Ein-/Ausgabe)
- `main()` ... Hauptprogramm (Funktion)
- `return 0` ... gibt den Rückgabewert 0 an main (alles in Ordnung)
- jede Anweisung wird mit einem Semikolon `;` abgeschlossen
- der Quelltext kann frei formatiert werden, d.h. es können beliebig 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

Unterschiede C und C++

- C ist eine rein *prozedurale* (imperative) Sprache
- C++ ist eine *objektorientierte* Erweiterung von C mit derselben Syntax
- in gewissener Weise ist $C \subset C++$
  (C-Programme können von C++-Compilern übersetzt werden)
- C++ ist durch seine Sprachstrukturen (template, class) sehr viel mächtiger als C

	Interpretersprachen	Compilersprachen
Beispiele	Shell (bash, tcsh), Perl, Mathematica, Python, ...	C/C++, Fortran, Pascal, D, Go, Rust, ...
Quelltext	direkt ausführbar	wird in Maschinensprache übersetzt, z.B. 0x90 für No Operation
Laufzeit-verhalten	Interpreter läuft als Programm → volle Kontrolle über Ausführung → Fehlermeldung, Argumentprüfung	Fehlerbehandlung schwierig → Aufgabe des Programmierers, meist nur Crash
Geschwind.	meist langsam	schnell durch Optimierung

→ außerdem auch Bytecode-Compiler (JAVA, C#) für Virtuelle Maschinen, JIT-Compiler (JavaScript, Perl)


Quelldatei
(.cpp, .C)
⇓
Compiler + Linker
(.o, .so, .a)
⇓
ausführbares Programm
(a.out, ...)

Befehl zum Kompilieren + Linken:

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

(GNU-Compiler für C++)†

Aufgabe 5.2 Kompilieren

Schreiben Sie mittels eines *Editors* eine Datei `nichts.cpp`, die *nur* aus der leeren Funktion `int main(){}` besteht. Kompilieren Sie diese Datei und führen Sie das Programm in der Shell aus (`./nichts`).

†alternativ: LLVM mit clang

- 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, z.B. auch mit Endung .exe

- der GNU-Compiler für C : gcc

Die `return`-Anweisung

```
return 0 ;
```

→ sofortiges Verlassen der Funktion (hier: `main`)

→ und Rückgabe des Wertes `0` an die rufende Programmeinheit (hier: Shell).

Typ des Rückgabewerts (hier: Ganzzahl 0) muss zum Typ der Funktion passen (hier: `int main(){}`, d.h. Ganzzahl).

Exkurs: Die Anweisung `return` in `main()`

Aufgabe 5.3 `return`

Testen Sie die Bedeutung des Rückgabewertes für die Shell:

1. Fügen Sie in der `main`-Funktion von Aufgabe 5.2 eine `return`-Anweisung ein, die `0` zurückgibt. Kompilieren Sie erneut und führen Sie Ihr Programm auf der Shell wie folgt aus:
   `./program && echo Alles ok!`
2. Lassen Sie vom Programm den Wert `1` *zurückgeben*, und wiederholen Sie damit obigen Shellaufruf.
3. *Zusatz:* Wie erreicht man eine Ausgabe `echo Fehler` (Shell) im Falle eines Fehlers (`return 1 ;`)?

Einfaches Programm zur Bildschirmausgabe

Beispiel: C++ Bildschirmausgabe mittels Streams

```
#include <iostream>

using namespace std ;

int main () {

    cout << endl << "Hallo, Welt!" << endl ;

    return 0 ; // alles ok

}
```

- `cout` ... Bildschirmausgabe (C++)
- `<<` ... Ausgabestreamoperator (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

Aufgabe 5.4 Fehlermeldungen, Notwendigkeit der Deklaration

1. Schreiben Sie das vorige Beispiel in eine Datei `helloworld.cpp` und erstellen Sie das ausführbare Programm `helloworld`.
2. Kommentieren Sie die Anweisung `using namespace ...` aus. Versuchen Sie zu kompilieren. Welche Fehlermeldung erhalten Sie?
3. Ändern Sie zusätzlich die Anweisung `return` in `reeturn` und versuchen Sie erneut zu kompilieren.
4. *Zusatz:* Leiten Sie die Fehlerausgabe des Compilers in eine Datei `gpperrors.log` um.

Blöcke: Sind eine Zusammenfassung von Anweisungen in geschweiften Klammern, z.B.

```
{ a = 7 ; c = 9.3 ; }
```

Wichtig: Wo im C/C++ Programm eine Anweisung steht, darf auch ein Block stehen.

besondere Blöcke: Funktionskörper (z.B. in `int main(){ }`), Kontrollstrukturen (`for (...) { ...}`)

Variablen sind ein Stück Arbeitsspeicher des Computers:

Deklaration

```
int a ;
```

... und Zuweisung

```
a = 3 ;
```

Typ der Variablen `int` = Ganzzahl (Integer)

Name der Variablen `a`

Wert der Variablen `3`

Aufgabe 5.5 Variablen deklarieren und initialisieren

Schreiben Sie ein kurzes C++-Programm, in dem Sie eine Variable vom Typ `int` innerhalb der `main`-Funktion deklarieren (also z.B. `int a ;`). Geben Sie den Wert der noch nicht initialisierten Variable mittels `cout` aus. Weisen Sie der Variable den Wert 2147483647 zu und geben Sie den Inhalt der Variablen mittels `cout` aus. Anschließend addieren Sie `+1` zu dieser Variablen durch folgende Zuordnung:

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

und geben erneut den Wert mittels `cout` aus.

Verwenden Sie als Vorlage das Beispiel von Seite 16.

numerisch:

ganze Zahlen (*integer*)
einfach genau `int`
doppelt genau `long`

Gleitpunktzahlen (*floating point numbers*)
einfach genau `float`
doppelt genau `double`

alphanumerisch (*character*):

alle Zeichen auf der Tastatur `char`

logisch (*boolean*):

nur `true` oder `false` `bool`

Integer-Datentypen I

Ganzzahlen (Integer) werden im Speicher *exakt* dargestellt.
$\rightarrow$ binäres Zahlensystem (Basis 2), z.B.

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

Aufgabe 5.6 Zahlenbasen

Bis zu welcher Zahl kann man im Binärsystem mit 10 Fingern zählen?

In der Zuweisung `a = 3` im Quelltext stellt `3` ein Integer-Literal (auch Literal-Konstante) dar.

Der Compiler wird diese Zahl ($3 = 1 \cdot 2^0 + 1 \cdot 2^1 \mathrel{\widehat{=}} \boxed{1}\boxed{1}$) an der entsprechenden Stelle einfügen.

[1] entspricht nicht unbedingt der "Leserichtung" des Computers $\rightarrow$ "Little Endian" (kleinestes Bit zuerst), also eigentlich 1011

Integer-Datentypen II

`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: $0 \ldots 1.8 \times 10^{19}$

Aufgabe 5.7 Integer-Größen

Überprüfen Sie die angegebene Grenze (exakter Wert) für Ganzzahlen vom Typ `unsigned long` mittels einer `for`-Schleife:

1. Die `for`-Schleife in C/C++ ähnelt der in der `bash` (Syntax vergleichen!):

```
for ( int i = 2 ; i <= 10 ; i++ ) {
  ...
}
```

2. Deklarieren Sie eine Variable `longlimit` innerhalb der `main`-Funktion, der Sie vor der Schleife den Startwert 2 zuweisen. In der Schleife soll bei jedem Durchlauf ein Faktor 2 multipliziert werden.
3. Geben Sie bei jedem Schleifendurchlauf die Schleifenvariable, welche mit dem Startwert 2 initialisiert wird, z.B. `i` (s.o.), sowie `( longlimit - 1 )` mittels `cout` aus.
4. Iterieren Sie entsprechend lange, bis das Limit von `unsigned long` erreicht ist. Fällt Ihnen etwas auf? Welche Bedeutung hat `i`?
5. Wiederholen Sie den Test mit dem Datentyp `long`. Was passiert, wenn über das Limit hinaus iteriert wird?

Tabelle: Darstellung: vorzeichenloser Wert (0s), Betrag und Vorzeichen (BuV), Zweierkomplement (2'S)

Binär	0s	BuV	2'S
0000	0	0	0
0001	1	1	1
	...		
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
	...		
1111	15	-7	-1

Nachteile der Darstellung als BuV:
0 und -0; Welches Bit ist Vorzeichenbit
($\rightarrow$ konstante Stellenzahl, Auffüllen mit Nullen);

Vorteil 2'S:
negative Zahlen immer mit höchstwertigen Bit=1

$\rightarrow$ vgl. $+1 + -1$ bitweise in BuV und 2'S

Binäre Arithmetik: $1 + 1 = 2$

$$\begin{array}{rr} & 0001 \\ + & 0001 \\ \hline = & 0010 \end{array}$$

Deklaration und Sichtbarkeit von Variablen I

Deklaration der Variablen sollte am **Blockanfang** erfolgen. Ausnahmen macht man mit den Zählvariablen in Scheifen (s. Aufgabe 5.7).

`char a, b ;` ... Deklaration von a und b

`int n1 = 3 ;` ... Deklaration und Initialisierung von n1

lokale Variablen / Variablen i.A.

- sind nur in dem Block (z.B. in `int main(){...}`) bekannt, in welchem sie deklariert (vereinbart) wurden
- sind bezüglich dieses Blocks **lokal**, d.h. ihr Wert kann nur in diesem Block geändert werden.
- sind außerhalb dieses Blockes unbekannt, d.h. sie existieren dort nicht.

Globale Variablen

- werden außerhalb der Funktionen definiert, z.B. vor main()
- sind dann in allen nachfolgenden Funktionen bekannt
- besitzen *dateiweite* Sichtbarkeit
- werden erst nach Beendigung des Programms aus dem Speicher entfernt

Dringende Empfehlung

Vermeiden Sie die Verwendung von globalen Variablen. Sie erschweren die Lesbarkeit von Quelltext und können zu komplexen Fehlern führen.

Deklaration und Sichtbarkeit von Variablen III

Globale und lokale Variablen

```
int m = 0 ;       // globale Variable

void calc() {
  int k = 0;      // lokale Variable
  m = 1 ;         // ok, da globale Variable
  j++ ;           // Fehler, da j nur in main bekannt
}

int main()  {
  int j = 3 ;
  j++ ; // ok
  for (int i = 1 ; i < 10 ; i++) {
     j = m + i ; // ok, alle sichtbar
  }
  m = j - i ;     // Fehler: i nicht definiert
  return j ;
}
```

Konstanten definieren

Werte, die sich im Laufe des Programms nicht ändern, sollten als Konstanten definiert werden:

```
const int a = 5 ;
```

Konstanten müssen bei der Deklaration intialisiert werden.

Danach können sie nicht mehr geändert werden.

Wann immer *möglich*, sollte man `const` verwenden!

Gleitkommazahlen-Datentypen I

- Gleitkommazahlen sind eine approximative Darstellung reeller Zahlen.
- Gleitkommazahlen lassen sich mittels

```
float radius, pi, euler, x, y ;
double  rbb, z ;
```

deklarieren.

- Gültige Zuweisungen sind z.B.

```
x = 3.0 ;
y = 1.1E-3 ; // bedeutet 1.1 * 10^-3
```

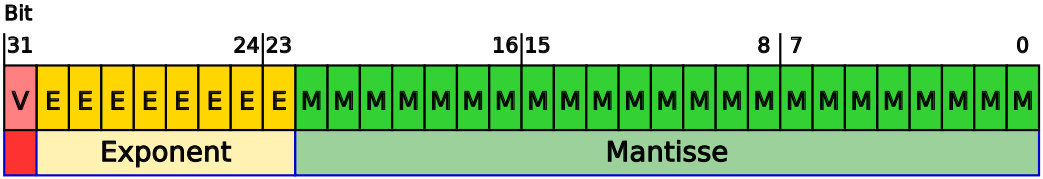
Gleitkommazahlen-Datentypen II

- Darstellung und Behandlung von Gleitkommazahlen wird z.B. durch die Norm IEEE 754 geregelt

$$x = s \cdot m \cdot b^e \tag{1}$$

dabei ist die Basis $b = 2$ (IBM Power6: auch $b = 10$)

- Für 32 Bit (Little-Endian, d.h. kleinstwertiges Bit zuerst):



Vorzeichen

(also kleinster Teil der Mantisse bei Bit 0)

- Mantisse wird durch *Normalisieren* auf die Form (Bsp.)

$$1{,}0100100 \times 2^4$$

gebracht, d.h. mit 1 vor dem Komma. Diese 1 wird nicht mitgespeichert.

Desweiteren wird auf den Exponenten noch der Bias (127 bei 32 Bit, 1023 bei 64 Bit) addiert

Beispiel Umwandlung Dezimalzahl nach IEEE-32-Bit

172.625	Basis 10
10101100.101×2^0	Basis 2
1.0101100101×2^7	Basis 2 normalisiert

addiere Bias von 127 zu Exponenten = 134

0 10000110 01011001010000000000000

Gleitkommazahlen-Datentypen IV

- daraus ergibt sich für einfache Genauigkeit (32 Bit):

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

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

Dezimalstellen: 7-8 $(= \log 2^{23+1} = 24 \log 2 \approx 24 \times 0.3)$

- analog gilt für 64 Bit – `double`:

Exponent: 11 Bit (r), Mantisse: 52 Bit

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

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

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

Viele Gleitkommazahlen lassen sich binär nicht exakt darstellen, z.B.

$$0.1 \approx 1.10011001100110011001101 \times 2^{-4} \tag{2}$$

Vergleiche: $\frac{1}{3} \approx 0.3333333\ldots$ im Dezimalsystem – das gilt für alle Brüche, deren Nenner sich nicht aus den Primfaktoren (2,5) der Basis 10 zusammensetzt, also $1/3, 1/6, 1/7, 1/9, \ldots$
Im Binärsystem nur ein Primfaktor: 2

Warnung

Keinesfalls darf man zwei Gleitkommazahlen blind auf Gleichheit prüfen, statt dessen muss ein Genauigkeitsbereich genutzt werden:
`abs( x - y ) <= eps` *statt* `x == y`

Aufgabe 5.8 Vorsicht, Gleitkomma!

Ergänzen Sie folgendes Codefragment zu einem ausführbaren Programm und lassen Sie sich yy ausgeben:

```
float x, yy ;

x = 1.043E-13 ; // (1)

x = x / 10.  ;

yy = x * x - (1.043E-14) * (1.043E-14)  ; // (2)
```

Was ist das Ergebnis? Wie kommt es zustande? Ersetzen Sie die Literalausdrücke in (1) und (2) durch Konstanten (Deklaration!) und wiederholen Sie den Test. Wie könnte man das zu erwartende Ergebnis erreichen? Worin besteht eine potenzielle Gefahr bei der Verwendung von Ausdrücken wie für yy?

Aufgabe 5.9 Vorsicht, Gleitkomma! II

Schreiben Sie ein Programm, das mittels einer Schleife für die Zahlen `x` von 1. bis 1000. in Schritten von 0.1 folgenden Inversionstest ausführt:

```
for ( int i = 1 ; i <= 10000 ; i++ ) {
   x = x + 0.1
   y = 1. / x ; // Inversion
   z = y * x ; // sollte 1. ergeben
   if ( z != 1.0) k = k + 1 ; // Test auf ‘‘fehlerhafte’’
                              // Inversion, Zählen (k)
 }
```

Denken Sie daran, `k` zu deklarieren und mit 0 zu initialisieren und lassen Sie sich `k` nach der Schleife ausgeben. Führen Sie den Test auch mit verschiedenen Gleitkommadatentypen durch. Was ist die Ursache für fehlerhafte Inversion?

Automatische Typumwandlungen

In C/C++ sind mehrere Typumwandlungen vordefiniert, die automatisch ablaufen:

```
int main () {
    int a = 3 ;
    double b ;
    b = a ;    // implizite Umwandlung  int -> double
    return 0 ;
}
```

Beim Umwandeln von Ganzzahldatentypen zu Gleitkommadatentypen werden die Nachkommastellen nicht berücksichtigt (kein Aufrunden).

Explizite Typumwandlungen (Casts) I

Desweiteren kann eine Typumwandlung auch explizit angewiesen werden (Casten):

C-Cast

```
int main () {
    int a = 3;
    double b ;
    b = (double) a ; // type cast
    return 0 ;
}
```

- klar: Integer $\leftrightarrow$ Gleitkomma
- auch: Pointer $\leftrightarrow$ Integer oder Pointer (s.u.)
- Achtung: Bei C-Casts findet zur Programmlaufzeit keine Typprüfung statt.

Explizite Typumwandlungen (Casts) II

Darüber hinaus gibt es in C++ auch gleichnamige Funktionen zur expliziten Umwandlung:

```
int i, k = 3 ;
float x  = 1.5 , y ;
i = int(x) + k ;
y = float(i) + x ;
```

Aufgabe 5.10 Ganzzahlumwandlungen

Was ist das Ergebnis für `i` und `y` im obigen Beispiel?

Explizite Typumwandlungen (Casts) III

Ein beliebter Fehler:

```
int i = 3 ;
double x, y, z;
x = 1 / i ;           // ergibt x = 0
y = 1 / double(i) ; // ergibt y = 0.333333
z = 1. / i ;          // ergibt auch 0.33333
```

Im Falle von `z = 1. / i ;` wird eine automatische Typumwandlung durchgeführt, weil `1.` - im Unterschied zu 1 - kein Integer-Literal ist.

Die Verwendung von Dezimalpunkten in Gleitkommarechnungen ist daher zu empfehlen. Gleitkommaliterale wie `1.` werden standardmäßig zu `double`-Zahlen umgewandelt. Mit dem Suffix `f`, also `1.0f` wird das Literal als `float` gespeichert.

Exkurs: Bibliotheken I

- bereits gesehen: selbst für simple Ein/Ausgabe wird die `iostream`-Bibliothek benötigt
- Idee von C/C++ im Unterschied zu vielen anderen Sprachen:
  nur sehr wenige eingebaute Befehle (z.B. `return`),
  alles andere durch entsprechende Bibliotheken
  $\Rightarrow$ hohe Flexiblität durch "Outsourcing"
- auch mathematische Funktionen werden erst durch eine entsprechende Bibliothek zugänglich

Einbinden von Bibliotheken in C++:

- Zur Compile-Zeit:
  automatischer Aufruf des C-Präprozessors (cpp) durch g++:
  Einlesen aller Anweisungen, die mit # starten, insbesondere

  ```
  #include <iostream>
  ```

  → Suche in bestimmten Standardpfaden (z.B. `/usr/include/`) nach Headerdatei, meist mit Endung `.h`, hier: `iostream`)
  → Einfügen (Inkludieren) der entsprechenden Datei
  → Übergabe an Compiler

Aufgabe 5.11 Der <iostream>-Header

In welchem Verzeichnis befindet sich die Header-Datei für die `iostream`-Bibliothek? Was enthält sie?

Aufgabe 5.12 Der C-Präprozessor

Präprozessor-Ausgabe: Rufen Sie den Präprozessor explizit:

`cpp helloworld.cpp output`

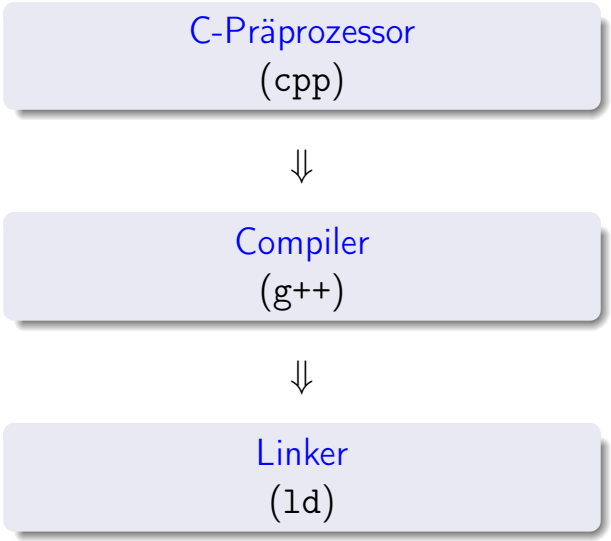
erzeugt aus dem Quelltext `helloworld.cpp` eine Ausgabedatei `output`. Sehen Sie sich deren Inhalt an.

- Zur Link-Zeit:
  Suche nach den zu den Header-Dateien gehörigen Bibliotheken, Übersetzen der in den Bibliotheken definierten Namen (Symbole) in Speicheradressen
  statisches Linken: Einfügen der benötigten Bibliothekssymbole in Programm

Aufgabe 5.13 Dynamische Bibliotheken

Der Unix-Befehl `ldd` zeigt für ein Programm die gelinkten dynamischen Bibliotheken an, z.B. `ldd helloworld`.
Vergleichen Sie, welche Bibliotheken werden für `helloworld` eingebunden und welche für `nichts`.

- Zur Laufzeit:
  dynamisches Linken: Laden des Programms und der Bibliothek in den Arbeitsspeicher
  Vorteil (gegenüber statischem Linken): Bibliothek wird nur einmal geladen, kann dann von anderen Prozesse mitbenutzt werden


C-Präprozessor
(cpp)
⇓
Compiler
(g++)
⇓
Linker
(ld)

Übersicht: Unix-Befehle für Entwickler

- `cpp`: C-Präprozessor für #-Anweisungen
- `g++`: C++-Compiler
- `ld`: Linkeditor (wird vom Compiler gerufen)
- `ldd`: zeigt Abhängigkeiten von dynamischen Bibliotheken
- `nm`: listet die *Symbole* einer Objektdatei/Bibliothek/Programms

Aufgabe 5.14 Symbole

Zu welcher Art von *Symbolen* gehört `main` im Programm `helloworld`? Wofür steht diese Art von Symbolen?

Zugriff auf mathematische Funktionen erhält man durch Einbinden der C-Mathematikbibliothek:

```
#include <cmath>
```

oder

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

Hinweis

Auf den meisten Linux-Installationen sind beide `include`-Anweisungen äquivalent. Unter MacOS X hingegen bewirkt Einbinden von `cmath` ein *Überladen* der mathematischen Funktionen, sodass z.B. `pow(3,2)` ($= 3^2$) nicht funktioniert, dafür aber `abs(-4)`.

Mathematikbibliothek II

Folgende mathematische Funktionen werden durch `math.h` bzw. `cmath` zur Verfügung gestellt:

```
cos();      sin();    tan();
asin();     atan();   acos();
cosh();     sinh();   tanh();
exp();      fabs();   abs(); // nur cmath
log();      ... natürlicher Logarithmus (Basis e)
log10();    ... dekadischer Logarithmus (Basis 10)
pow(x,y);   ... x^y †
sqrt();
```

†Es gibt absurderweise keinen Potenzoperator in C/C++. Man beachte, dass eine Multiplikation `x*x` schneller ist als `pow(x,2.0)`.

Folgende Konstanten sind u.a. vordefiniert:

`M_E`	...	e
`M_PI`	...	π
`M_PI_2`	...	$\pi/2$
`M_PI_4`	...	$\pi/4$
`M_2_PI`	...	$2/\pi$
`M_SQRT2`	...	$+\sqrt{2}$

Variablen über die Tastatur eingeben I

Einbinden der Bibliothek `iostream` ermöglicht auch die Eingabe von Variablen:

Einlesen in Variable `x`:

```
cin >> x ;
```

Variablen über die Tastatur eingeben II

Beispiel

```
#include <iostream>

using namespace ::std ;

int main () {
  int i ;
  cout << endl << "Bitte eine ganze Zahl eingeben: " ;
  cin >> i ;
  cout << endl ;
  cout <<"Sie haben " << i << " eingegeben." << endl ;
  return 0 ;
}
```

Variablen über die Tastatur eingeben III

Aufgabe 5.15 Tastatureingabe

1. Kompilieren Sie das obige Beispiel zur Tastatureingabe und führen Sie es aus.
2. Schreiben Sie ein Programm `zylinder`, welches den Benutzer auffordert, Radius und Höhe für einen Zylinder (in cm) einzugeben, und diese Größen dann auch in Variablen einliest. Das Programm sollte dann aus diesen Größen das Zylindervolumen und den Flächeninhalt der Zylinderoberfläche ausrechnen und entsprechend ausgeben.
3. *Zusatz:* Schreiben Sie ein Programm, welches eine in Sekunden eingebene Zeitdauer in das Format "hh:mm:ss" umwandelt und ausgibt.

Zeichenvariablen I

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' ;
```

Zeicheneingabe

```
char zeichen ;
int  zahl ;
cout << "Eingabe eines Zeichens: " ;
cin >> zeichen ;
cout << "Zeichen war: " << zeichen
     << " entspricht " << int(zeichen) << endl;
cout << "Eingabe einer Zahl: " ;
cin >> zahl ;
cout << "Zahl " << zahl
     << " entspricht " << char(zahl) << endl;
```

Aufgabe 5.16 Zeichen

Ergänzen Sie obiges Beispiel zu einem Programm, kompilieren Sie dieses und führen Sie es aus. Welchen (dezimalen) ASCII-Code haben }, Y und 1 ? Welches Zeichen hat den Code 97 ?

Zeichenvariablen III

Die Bibliothek `ctype.h` ermöglicht das Testen von Zeichen, z.B.

- `isdigit(eingabe)`
  liefert 1, falls `eingabe` eine Ziffer (0-9) ist, sonst 0
- `isalpha(eingabe)`
  liefert 1, falls `eingabe` ein Buchstabe (a-z, A-Z) ist, sonst 0
- `char(tolower (eingabe))` †
  Umwandlung in Kleinbuchstabe
- `char(toupper (eingabe))` †
  Umwandlung in Großbuchstabe

Strings

Zeichenketten (Strings) lassen sich in C++ mittels Arrays (Feldern) von Character-Variablen oder mithilfe der Klasse `string` speichern.

†Achtung: `ctype.h` arbeitet mit den `int`-Werten der Zeichen, daher `char( )`

Felder (Arrays) in C/C++ I

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++ !!!

Wir benötigen wieder `for`-Schleifen, s. S. 65

Aufgabe 5.17 Zugriff auf Felder

Schreiben Sie ein C++-Programm zum Einlesen von x-y-Paaren in Felder.

1. Zunächst sollten Feldern `x[100]` bzw. `y[100]` vom Typ `double` deklariert werden.
2. Der Benutzer sollte angeben müssen, wieviele n Datenpunkte er eingeben möchte.
3. Das Program soll vom Benutzer eingegebene x- und y-Werte mittels einer for-Schleife (siehe Aufgabe 5.7) in Feldern `x[100]` bzw. `y[100]` abspeichern:
   `cin >> x[i] >> y[i]`
4. Zur Überprüfung sollten die Daten nach dem Einlesen wieder paarweise, wie in einer x-y-Tabelle ausgegeben werden.
5. Das Programm soll auch den arithmetischen Mittelwert jeweils von x und y ausgeben.

Aufgabe 5.18 Lineare Regression

Die in Aufgabe 5.17 eingelesen Datenpunkte sollen weiter verarbeitet werden:

1. Für die eingegebenen Werte soll eine Ausgleichsgerade

$$y = b \cdot x + a \tag{3}$$

mittels

$$b = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{4}$$

$$a = \overline{y} - b \cdot \overline{x} \tag{5}$$

bestimmt werden. Die Koeffizienten a, b müssten am Ende ausgegeben werden.

Zweidimensionale Felder I

`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)

Aufgabe 5.19 Reihenfolge der Einträge in Arrays

Der prozessor-nahe Speicher (Cache) ist begrenzt, daher ist es wichtig, Programme so zu gestalten, dass die für eine Aufgabe in den Cache einzulesenden Daten nahe bei einander liegen.

Angenommen, Sie schreiben eine kosmologische Simulation mit 10^6 Teilchen, für jedes müssen die Koordinaten und Geschwindigkeiten (3D) in einem Array `particle[][]` abgespeichert werden. Eine Funktion loopt über alle Teilchen muss dabei jeweils auf alle Daten eines Teilchens zugreifen können.

Wie sollte das Array dimensioniert werden: `particle[6][1000000]` oder `particle[1000000][6]` ?

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) I

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.

Aufgabe 5.20

1. Was ist der Unterschied zwischen `'Y'` und `"Y"`?
2. Welche von beiden Literalen ist erlaubt: `'Ja'` oder `"Ja"`?
3. Was stimmt hier nicht: `char text[4] = "Nein" ;`?

Beispiel für C-Strings

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

int main () {
    char text[80] ;
    cout << endl << "Bitte einen String eingeben:" ;
    cin >> text ;
    cout << "Sie haben" << text << "eingegeben\".
     << endl ;
    return 0 ;
}
```

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

Wir kennen bereits for-Schleifen:

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;
}
```

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

```
sum += a  → sum = sum + a
x++       → x = x + 1      (Inkrementoperator)
x--       → x = x - 1      (Dekrementoperator)
```

Post- und Präinkrementierung

Postinkrement
```
int j = 2 ;
int a = j++ ;
```
Ergebnis: a = 2

Präinkrement
```
int j = 2 ;
int a = ++j ;
```
Ergebnis: a = 3

Vorsicht: `if (i == 0 && ++j == 1)`
→ keine Inkrementierung von `j`, falls `i` $\neq$ `0`

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.

Logische Operatoren II - Verknüpfungen

```
(a < b) || (c != a)   oder
(a < b) && (c != a)   und
   !(a < b)           nicht
```

Das Klammern ( ) bei Kombinationen von Ausdrücken ist der Eindeutigkeit halber empfehlenswert.

```
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`.

Aufgabe 5.21 Erfüllbarkeitsproblem (SAT)

Welche der folgenden aussagenlogischen Formeln ist erfüllbar (=wahr)? Durch welche Besetzung wird dies erreicht?

1. $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_2 \vee x_3 \vee x_4)$
2. $(x_1 \vee x_2) \wedge (\overline{x_2} \vee x_1) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3} \vee \overline{x_1})$

Hinweis: Sie können eine Abfolge von ineinander geschachtelten Schleifen erstellen, die die Belegung der `bools` $x_1 \ldots x_4$ ändern (Wahrheitstafel) oder sie arbeiten mit den bitweisen Verknüpfungen.
Das Zeichen $\vee$ bedeutet *oder*, $\wedge$ steht für *und*, $\overline{x}$ heißt *nicht* x.

siehe → for-Schleifen, S. 65

Aufgabe 5.22 Einfache for-Schleife

Mithilfe zweier for-Schleifen und des Befehls cout soll im Terminal ein Dreieck (z.B. rechwinklig 10×10 Zeichen) "geplottet" werden:

```
XXXXXXXXXX
XXXXXXXXX
XXXXXXXX
XXXXXXX
XXXXXX
XXXXX
XXXX
XXX
XX
X
```

Aufgabe 5.23 Das Sieb des Eratosthenes†

Alle Primzahlen unter den ersten n natürlichen Zahlen erhält man einfach mithilfe folgender Methode:

1. Man beginne mit der zwei, die nicht gestrichen wird, da sie prim ist. Man streiche alle Vielfachen von zwei.
2. Man wähle die nächstgrößere Zahl (also 3), die nicht gestrichen wurde, diese ist prim und wird nicht gestrichen. Man streiche alle Vielfachen dieser Zahl, beginnend mit ihrem Quadrat.
3. Man wiederhole Schritt 2 bis n.

Implementieren Sie ein entsprechendes Programm in C++ zur Ausgabe der Primzahlen bis n.

†Eratosthenes (275-194 v. Chr.): griech. Geograph (Erdumfang), Astronom und Mathematiker

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

break ; // stoppt Schleifenausfuehrung
```

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

Wir hatten in Aufgabe 5.9 bereits eine bedingte Ausführung mittels `if` gesehen:

```
if (z != 1.0) k = k + 1 ; // Test auf fehlerhafte Inversion
```

Entscheidungen/Verzweigungen

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

if (a == 0) a = k2 ; // Verzweigungen
else if (a > 1) {
            a = k1 ;
}
else        a = k3 ;
```

Aufgabe 5.24 Nutzerfehleingaben abfangen

Ergänzen Sie ihre Programme aus den Aufgaben 5.15 und 5.17 so, dass etwaige Fehleingaben, z.B. negativer Radius oder mehr Datenpunkte als dimensioniert, erkannt werden und der Nutzer diese nochmals neu eingeben muss.

Welche Art von Schleife ist hierfür sinnvoll?

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 ;
}
```

Aufgabe 5.25 Alternative Eingaben

Erweitern Sie Ihr Zylinder-Berechnungsprogramm aus Aufgabe 5.15 mithilfe einer `switch`-Anweisung so, dass der Nutzer Radius und Höhe in cm oder m eingeben kann.

Funktionen in C/C++ I

Wir kennen bereits Funktionen:

```
int main () { }
double pow (x,y) ;
char (i) ;
```

In C/C++ sind Funktionen die einzigen *Prozeduren* ($\rightarrow$ prozedurale Programmiersprache), auch das Hauptprogramm ist eine Funktion.

Vereinbarung von Funktionen - Deklaration

Funktionen müssen vor ihrer Verwendung, also i.d.R. vor `int main()` *deklariert* werden. Dies kann direkt:

```
double myfunc (double x, double y) ;
```

oder mittels sogenannter *Header*-Dateien und `#include` erfolgen:

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

Die Funktionsdefinition erfolgt außerhalb von `int main()`.

Struktur von Funktionen - Definition

rückgabetyp name (arg1, ...) { ... }

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

- Rückgabetyp: Jede Funktion hat *einen* Rückgabetyp, z.B. `int`. Der Rückgabewert wird mittels `return` an die rufende Funktion übergeben. Funktionen ohne Rückgabetyp müssen als `void` deklariert werden.
- Name: Funktionen werden über ihren Namen gerufen, er kann - bis auf reservierte Schlüsselwörter (z.B. `for`) - frei gewählt werden.
- (arg1, ...): Runde Klammern sind Pflicht, darin *können* Argumente (formale Parameter) zur Übergabe angegeben werden. Diese Variablen (mit diesen Namen) stehen dann auch automatisch *in* der Funktion bereit.
- { }: In geschweiften Klammern steht die Implementierung/Definition der Funktion.

Beispiel

```
double quadrat ( double x ) ; // Deklaration, vor main()

int main () {
 double y = 1.41, yy ;
 yy = quadrat (y) ; // Aufruf mit Variable
 cout << yy + quadrat (2.) ; // Aufruf mit Literal
}

// Definition nach main () :
double quadrat ( double x ){ return (x*x) ; }
```

Der Name des Übergabeparameters (hier: `x`) kann bei der Funktions- *Deklaration* auch weggelassen werden, wichtig ist die Angabe des Typs!

Vorteile von Funktionen:

- Programmteile, die mehrfach genutzt werden, müssen nur einmal geschrieben werden!
- Übersichtliche Strukturierung großer Programme.
- Variablen können *innerhalb* verschiedener Funktionen den gleichen Namen tragen ($\rightarrow$ Kapselung) - verschiedene Bearbeiter müssen sich nicht abstimmen.

Aufgabe 5.26 Bisektion - Anwendung des Zwischenwertsatzes

Schreiben Sie ein Programm, das mittels Bisektion (Intervallschachtelung) die Nullstelle x_0 der Funktion

$$f(x) = -4.905 \cdot x^2 - 2x + 71 \tag{6}$$

im Intervall $[0; 5]$ findet.
Deklarieren und definieren Sie dazu die Funktionen:

1. `double f(double x) ;`
   zur Berechnung der o.g. Polynomfunktion.
2. `double Intervall_halb (double a, double b) ;`
   zur Berechnung der Intervallmitte.

Legen Sie selbst eine Genauigkeit, z.B. 10^{-7}, für $f(x_0)$ fest, bis zu der mittels einer `while`-Schleife das Verfahren läuft.
Geben Sie bei jedem Iterationsschritt entsprechende Werte aus.

Aufgabe 5.27 Newton-Verfahren

Wiederholen Sie die Nullstellensuche von Aufgabe 5.26, diesmal unter Verwendung des Newton-Verfahrens:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7}$$

welches iterativ die Nullstelle x_{n+1} findet. Wieviel Schritte brauchen Sie (bei gleicher Genauigkeit) im Vergleich zum Bisektions-Verfahren?

Zusatz: Bei schwierigen physikalischen Problemen ist es manchmal nicht praktikabel, $f'(x)$ analytisch zu berechnen. Stattdessen greift man auf den Differenzenquotienten zurück:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{8}$$

für hinreichend kleine Δx.
Nutzen Sie das Newton-Verfahren mit Differenzenquotienten und vergleichen Sie die Konvergenzgeschwindigkeit mit dem exakten Newton-Verfahren.

Nachteil von Funktionen:

- Funktionen sind eigene Programmeinheiten im Speicher, die den Ausführungsfluss des Programms unterbrechen:
  → *Call & Return*
- viele kleine Funktionsaufrufe sind ineffizient

Lösung:

Kurze Funktionen können mit dem Schlüsselwort `inline` vor `main()` vereinbart werden:

```
inline double quadrat (double x) { return(x*x) ; }
```

Der Aufruf im Programm erfolgt dann mit `quadrat(7.2);`
→ siehe auch in diversen Header-Dateien

Vorteile von Inline-Funktionen:

- der Compiler† fügt die entsprechende Funktion beim Übersetzen ein (statt beim Linken), d.h. der Aufruf-Overhead entfällt

Vorteile gegenüber Macros (mittels `#define f(x) (x*x)`):

- Typprüfung
- Rekursion

Aufgabe 5.28 Rekursion mit Inline-Funktion

Schreiben Sie ein C++-Programm zum Berechnen der Fakultät $n!$ einer vom Benutzer einzugebenden Zahl mittels rekursiver Inline-Funktion `unsigned long fac(int n)`.
Dazu sollte die Inline-Funktion jeweils `n * fac(n-1)` zurückgeben, falls $n > 1$, sonst 1.

†Allerdings entscheidet der Compiler selbst, ob er die Ersetzung durchführt.

Rekursionen

- Rekursionen können i.d.R. durch Iterationen (Schleifen) ersetzt werden.
- Einige Programmiersprachen (z.B. die funktionale Programmiersprache Tcl) kennen keine Schleifen $\rightarrow$ stattdessen rekursiver Funktionsaufruf
- Rekursionen lassen Quellcode schlanker, eleganter aussehen.
- ABER: Rekursive Funktionsaufrufe sind langsamer, benötigen mehr Speicher (bis hin zum Stackoverflow) und lassen sich i.d.R. nicht wie Schleifen optimieren.

$\rightarrow$ **Vermeiden Sie Rekursionen.**

Funktionsstruktur: Call by value I

Bei den bisher behandelten Funktionsaufrufen werden von den Argumente nur der jeweilige *Wert* an die Funktion übergeben: Call by value.

Aufgabe 5.29 Parameterübergabe per Wert

Ergänzen Sie folgendes Code-Fragment so, dass die Funktion `summe_vertausche()` die Summe ihrer Argumente (`a, b`) berechnet und Sie außerdem vertauscht:

```
int summe_vertausche (int a, int b) ;
int main () {
    ...
     cout << a << " " << b << endl ;
     c = summe_vertausche(a, b) ;  // Funktionsaufruf
    ...
}
```

Überprüfen Sie die erfolgreiche Vertauschung *nach* dem Funktionsaufruf mittels `cout`.

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)

Lösung: Parameterübergabe per Referenz

Referenzen I

```
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!

Aufgabe 5.30 Referenzen

Schreiben Sie ein kurzes C++-Programm, mit dem Sie sich selbst die o.g. Eigenschaften von Referenzen klar machen.

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

Aufgabe 5.31 Vertauschen - Jetzt aber richtig

Verändern Sie Ihr Programm aus Aufgabe 5.29 dahingehend, dass die Funktion `summe_vertausche` zur Parameterübergabe nun Referenzen benutzt. Testen Sie den Erfolg.

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

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".

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`

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
```

Aufgabe 5.32 Pointer und Arrays

Zeiger und Arrays entsprechen einander in gewisser Weise. Betrachten Sie dazu folgende Programmzeilen:

```
int array[4] ;        // (1)
int *parray = 0 ;     // (2)
parray = array ;      // (3)
parray[3] = 1 ;       // (4)
parray = &array[2] ; // (5)
parray[1] = 6 ;       // (6)
```

1. Warum sollte ein Pointer immer mit 0 initialisiert werden, so wie in Zeile (2)?
2. Durch die verblüffende Zuweisung in Zeile (3) enthält der Pointer nun die Startadresse des Arrays. Wie müsste diese Zuweisung normalerweise erfolgen?
3. Der Pointer verhält sich plötzlich wie ein Array, d.h. er kann mit eckigen Klammern "indiziert" werden. Wie funktioniert das?
4. Was bewirken die beiden Zeilen (5) und (6)?
5. Bei welchem Arrayindex wird die Zahl 6 gespeichert?

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.

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

Starten eines Pogramms mit einem Parameter in der Kommandozeile I

Im Funktionskopf von `main` werden *diese* beiden formalen Parameter angegeben:

```
int main (int argc, char *argv[])
```

`argc` ... "argument-counter" steht für die Zahl der Parameter.
Dabei zählt der Name des Programms als erster Parameter (Eintrag 0)!

`argv` ... "argument-vector" steht für die Werte der Parameter, werden als Zeichenkette `char*` abgespeichert.

Diese Variablennamen sind nicht frei wählbar!

Starten eines Pogramms mit einem Parameter in der Kommandozeile II

Beispiel-Programm

```
int main (int argc, char *argv[]) {
    if (argc == 1) // Ein Argument übergeben!
     cout << argv[1] << " wurde eingegeben" << endl ;
    return 0 ;
}
```

Starten des Programms `beispiel` mit Parameter *weg1* oder *weg2*:

```
joule/gast>./beispiel weg1
weg1 wurde eingegeben
 oder
joule/gast>./beispiel weg2
weg2 wurde eingegeben
```

Aufgabe 5.33 Kommandozeilenparameter

Modifizieren Sie Ihr Programm aus Aufgabe 5.22 oder 5.23 so, dass es beim Aufruf eine Zahl übergeben bekommen *kann*, d.h. es sollte geprüft werden, ob ein Argument übergeben wurde.

Das Programm sollte in diesem Fall dann den Parameter mittels der Funktion `atoi()` - erfordert das Inkludieren von `<stdlib.h>` - in eine Ganzzahl umwandeln und als Seitenlänge des Dreiecks (in Zeichen) bzw. maximales n für die Primzahlensuche verwenden.

Das Programm sollte erkennen, wenn es keinen Parameter übergeben bekommt und dann einen Default-Wert benutzen.

Dynamische Feldvereinbarung I

In C++ ist es auch erlaubt, lokale Arrays (Felder) variabler Länge zu deklarieren:

```
int dim1, dim2 ;
cout << "Eingabe von n und m" ;
cin >> dim1 >> dim2 ;

int matrix[dim1][dim2] ;
         // zweidimensionales Array variabler Länge
         Anweisungen ;
```

Man beachte, dass die Größe der Matrix erst zur Laufzeit festglegt wird (dynamische Allozierung).

Allerdings kann der Compiler solche Arrays auch nicht mittels

```
int matrix [dim1][dims] = {0} ;
```

initialisieren (`for`-Schleife nötig).

Der Zugriff auf Arrays mittels Index `vec[i]` ist i.d.R. nicht die effizienteste Methode, da dafür die Typgröße (z.B. `int` = 4) erst mit dem Index `i` multipliziert werden muss, um dann auf die Startadresse des Arrays addiert zu werden.

Schneller geht es, wenn man Pointer verwendet (s. auch Aufg. 5.32):

```
int vec[N] ;
int *pvec = vec ; // Zeiger auf Startadresse

for( int i =  0 ; i < N ; i++ ) {
   *pvec = 0 ; // Setze Inhalt von vec auf 0
   pvec++ ;    // erhöhe Speicheradresse um eine Einheit
}
```

Aufgabe 5.34 Dynamische Allozierung und intialisieren

Sie können die dynamische Feldvereinbarung und das Nullen des Feldes mittels Pointer nun für Ihr Programm zur Primzahlermittlung aus Aufgabe 5.23 benutzen. D.h. erst zur Laufzeit wird bestimmt, bis zu welchem n die Überprüfung auf Primzahlen erfolgt.

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 ;
```

Die so deklarierten konkreten Strukturen nennt man *Instanzen* oder *Objekte* (→ 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 ;
```

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.

Struct und class - Eigene Datentypen definieren V

Der Zugriff auf *Member* einer Struktur/Klasse kann mittels spezieller Schlüsselwörter eingeschränkt werden ($\rightarrow$ Kapselung):

private	Zugriff nur innerhalb der Struktur/Klasse
protected	wie private, außerdem von *abgeleiteten* Klassen
public	Zugriff auch von außen möglich

Statt `struct` können Datentypen auch mit `class` definiert werden. Für Klassen sind die Member defaultmäßig `private`, während sie für Strukturen defaultmäßig `public` sind.

Die Verwendung von Klassen (inklusive Vererbung) ist ein wesentlicher Unterschied von C++ gegenüber C.

Zugriffsrechte von Klassen und Strukturen

```
class complex {
      float re, im ; //  default: private
      public :
      void set_re (float x) { re = x ;} //  notwendig,
                                        //  da re private
      void set_im (float x) { im = x ;}
} ;
```

Wenn die Membervariablen `private` sind, werden öffentliche Methoden (sog. Getter und Setter) oder *Konstruktoren* für den Zugriff benötigt.

Konstruktoren sind Methoden ohne Rückgabewert, die Instanzen von Klassen initialisieren:

Konstruktoren

```
class complex {
     float x, y ;
     public:
     complex (float x, float y) : re(x), im(y) {}
     // besser als: complex (float x, float y)
                  { re = x ; im = y ; }
}
complex c (1.1, 2.2) ;
```

Wird kein Konstruktor definiert, so erzeugt der Compiler einen Default-Konstruktor, der ohne Argumente gerufen wird:

```
complex c ;
```

Aufgabe 5.35 Komplexe Memberfunktionen

Erstellen Sie Memberfunktionen für die selbstdefinierte Struktur/Klasse `complex`:

1. `cconj` zur komplexen Konjugation
2. `print` zur Ausgabe einer komplexen Zahl im Format `(re,im)`
3. `add` zur Addition zweier komplexer Zahlen
4. `mult` zur Multiplikation zweier komplexen Zahlen

usw.

Die Bibliothek <string> stellt den Datentyp (Klasse!) string sowie Methoden für die Bearbeitung von Zeichenketten zur Verfügung:

```
#include <iosteam>
#include <string>
using namespace :: std ;
int main () {
   string s ; // leerer String
   string text = "Das ist ein String." ; // Stringliteral
   cout << text << endl ;
   string text2 ("Aha.") ; // Literal, Konstruktor
   s = text2 ; // Zuweisung eines anderen String-Objekts
   return 0 ;
}
```

Achtung! Viele C/C++-Methoden setzen nach wie vor auf char* statt string → Casten erforderlich.

Die Klasse `strings` stellt Methoden zur Bearbeitung von Strings bereit, z.B.

```
string s ;
cin >> s ;
cout << "Wortlaenge: " << s.size() << endl;
string s0 = "Du hast  gesagt." ;
s0.insert(9,s);
cout << s0 << endl ;
```

Besonders elegant ist das Verbinden von Strings mittels des (überladenen) Operators +:

```
string s1 = "Hallo" , s2 = ", Welt!" ;
cout << s1 + s2 << endl ;
```

Schreiben von Daten in eine Datei I

Die einfache Möglichkeit:

1. Ausgabe mittels `cout`
2. Ausgabeumleitung in der Shell mittels Umleitungsoperator > bzw. >>

Beispiel: `./zylinder > ergebnis.dat`

Nachteile:

- keine Benutzerinteraktion mehr, da alle Ausgabe in Datei umgeleitet wird
- Ausgabe nur in max. zwei Dateien pro Programm (`cout` und `cerr`)

Schreiben von Daten in eine Datei II

Die alternative Möglichkeit -

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() ;`

Aufgabe 5.36 Postscript-Datei selbstgemacht

Wir wollen mithilfe eines C++-Programms eine PS-Datei erzeugen, die ein n-Eck mit Umkreis darstellt. PS-Dateien sind (ASCII)-Textdateien, die - ähnlich einem bash-Skript - Befehle enthalten, die vom PS-Viewer interpretiert werden.

1. Die PS-Datei fängt wie folgt an:
   `%!PS-Adobe-1.0`
   `gsave`
2. Die PS-Datei endet mit:
   `showpage grestore`
3. Eine Linie von (x_1, y_1) nach (x_2, y_2) kann man mittels
   `x_1 y_1 moveto`
   `x_2 y_2 lineto stroke`
   zeichen, wobei `x_1` usw. Pixel-Koordinaten auf der PS-Seite sind.
4. Ein Kreis um (x, y) mit Radius r lässt sich so zeichnen:
   `x y r 0 360 arc stroke`

Aufgabe 5.36 - Fortsetzung

6. Das C++-Programm fragt den Nutzer nach der Anzahl n der Ecken.
7. Die $i$te Ecke des Polygons lässt sich z.B. wie folgt berechnen:

```
phi = ( (float) i  - 0.5 ) * 2. * M_PI / (float) n ;
  x = 0.6 + 0.5 * cos(phi) ;
  y = 0.6 + 0.5 * sin(phi) ;
```

   Dabei müssten x und y durch Multiplikation mit einem Faktor (z.B. `scale=400`) noch in Pixel umgerechnet werden.
8. *Zusatz:* Die Flexiblität des Programms wird erhöht, wenn die Zeichenfunktionen für Kreis und Linie zusammen mit einem Datentyp Punkt (mit Membervariablen x und y) in eine Klasse oder Struktur gepackt werden.

Lesen aus einer Datei

Bibliotheken inkludieren: <fstream>

```
char zeile[200] ;
ifstream dateiin ; // ifstream-Objekt erzeugen
dateiin.open("daten.dat") ; // Datei öffenen
while ( dateiin.good() ) {
      dateiin.getline(zeile,200) ; // Zeile einlesen;
                                   // Buffer (200) festlegen
      cout << zeile << endl ;      // Zeichenausgabe
}
```

- good überprüft, ob das Dateiende erreicht ist (oder ein Fehler auftrat)

Aufzählungen I

Mittels der enum-Deklaration können Aufzählungen von Konstanten erzeugt werden. Z.B.

```
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So} ;
Tag heute, morgen, gestern ; // Deklaration von Tagen
heute = Mi ; // Zuweisung
if (heute == So) cout << "freier Tag" ; // Vergleich
if (heute > Fr) cout << "Wochendende!" ; // Reihenfolge
```

Aufzählungen II

Man beachte, dass intern die Elemente der Aufzählung nummeriert sind (beginnend bei 0). Diese Nummerierung kann auch selbst gesetzt werden:

```
enum metall {Ti = 22, V, Cr, Mn} ;
enum farbe {rot = 2, gruen = 5} ;
```

Dadurch sind auch folgende Zuweisungen möglich:

```
int k = rot ;
farbe color = (farbe) 2 ; // nur durch Casten
```

Templates I

Templates ermöglichen eine universelle Definition von Strukturen. Die konkrete Realisierung mittels eines bestimmten Datentyps wird dem Compiler überlassen.

Templatefunktion

```
template <class T> // statt class kann auch typename stehen
T sqr (const T &x)
{ return x*x ; }
```

Der Compiler erkennt anhand des Schlüsselwortes `template` und den spitzen Klammern < >, dass `T` ein *Templateparameter* ist. Der Compiler wird diese Funktion erst Übersetzen, wenn beim Funktionsaufruf ein konkreter Datentyp angegeben wird, z.B.

```
double w = 3.34 ; int k = 2 ;
cout << sqr(w) << " " << sqr(k) ;
```

Darüberhinaus können Templates auch zum Erzeugen von Strukturen/Klassen genutzt werden. So ist z.B. die Klasse `complex` der Standard-C++-Bibliothek (`#include <complex>`) realisiert:

Templateklasse

```
template <class T>
class std::complex {
   T re, im ;
 public:
     ...
   T real() const return re ;
     ...
}
```

D.h. die Membervariablen `re` und `im` können beliebige Datentypen sein.

Typdefinition mittels typedef

Mittels typdef *datentyp aliasname* können Namen vereinbart werden, z.B.

```
typedef unsinged long  large;
typedef char*  pchar ;
typedef std::complex<double>  complex_d ;
```

Die so gealiasten Typennamen können dann zur Deklaration usw. benutzt werden:

```
large mmm ;
pchar Bpoint ;
complex_d z = complex_d (1.2, 3.4) ;
```

Ein wesentliche Stärke von C++ ist die Möglichkeit der Behandlung von Fehlern, sog. Ausnahmen (Exceptions) zur Laufzeit.

Exceptions werfen: try - throw - catch

```
    try {
       cin >> x ;
       if ( x < 0. ) throw "Negativer Wert!" ;
       y = g(x) ;
    }
    catch (char* info) { // Fange Exception aus try-Block ab
       cout << "Programm-Stopp, weil: " << info << endl ;
       exit (1) ;
    }
double g (double x)
{ if (x > 1000.) throw "x zu gross!" ; ... }
```

Ausnahme-Behandlung - Exceptions II

`try { ...}`

- innerhalb eines `try`-Blocks kann ein beliebige Ausnahme geworfen werden

`throw` *e* ;

- Werfen einer Ausnahme *e*
- anhand des Typs von *e* wird entschieden, welcher `catch`-Block angesprungen wird
- Ausnahmen dürfen intrinsische oder selbstdefinierte Typen sein

`catch ( typ e ) { ...}`

- nach dem `try`-Block können ein oder mehrere `catch`-Blöcke stehen
- anhand des Typs von *e* wird der erste passende `catch`-Block gewählt
- beliebige Ausnahmen lassen sich mittels `catch (...)` abfangen
- wird nach `try` kein passender `catch`-Block gefunden, so wird in den darüberliegenden Aufrufebenen gesucht
- wird gar kein passender Block gefunden, so wird `terminate` gerufen, Standard: Abbruch