Algorithmisches Programmieren (Numerische Algorithmen mit C++)

Inhaltsverzeichnis

Literatur

- Marc Steinbach; Algorithmisches Programmieren; Skriptum, Leibniz Universität Hannover, 2015
- Jürgen Wolf; C++ von A biz Z, Galileo Computing, 2008
- http://www.cplusplus.com/

Was ist C++?

Eigenschaften:

- C++ ist eine objektorientierte höhere Programmiersprache; aber keine reine objektorientierte Sprache (da ja aus C entstanden - Vorteile und Nachteile);
- C++ umfasst die Sprache C nahezu vollständig;
- C++ wird zur Ausführung in Maschinensprache übersetzt (compiliert);
- Es können mehrere separat compilierte Programmteile kombiniert werden;
- C++ ist extrem vielseitig und teilweise sehr kompliziert (Vererbung, Objektorientierung, templates, Polymorphismus, Überladen von Funktionen);
- In C++ können komplexe Algorithmen effizient programmiert werden und bildet daher die Grundlage einiger sehr erfolgreicher Codes im Wissenschaftlichen Rechnen: deal.II, dune, Fenics, FreeFem++, ..., mit bis zu mehreren Hunderttausend!!! Zeilen Code ...
- C++ erlaubt generische Programmierung (wiederverwendbare Softwarebibliotheken mit allg. Funktionen für verschiedene Datentypen und Datenstrukturen).

Entstehung

- 1979 von Bjarne Stroustrup (dänischer Informatiker) entwickelt;
- er erweiterte die Sprache C um ein Klassenkonzept;
- C diente als Ursprung, weil diese schnellen Code erzeugt und einfach auf andere Platformen portiert werden kann;
- C war damals die am stärksten verbreitete Sprache;

$\Rightarrow$ C mit Klassen

- 1982: Umbennenung in C++
- 1998: Normierung von der ISO (ISO/IEC 14882:1998)

Stärken von C++

- Maschinennahes Programmieren;
- Erzeugung von hocheffizientem Code;
- Hohe Ausdrucksstärke und Flexibilität;
- Für große (Programmier-)Projekte geeignet (siehe obige Pakete für das wissenschaftliche Rechnen);
- Sehr weite Verbreitung (wie oben angesprochen);
- open-source (keine Organisation im Hintergrund);
- C-kompatibel, da ja aus C entstanden.

Schwächen von C++

- Da C++ auf C basiert, sind einige Teile compilerspezifisch;

$\Rightarrow$ Dies erschwert die Portierung auf verschiedene Rechnertypen, Betriebssysteme (Windows, MAC, Unix) und Compiler;

- Kaum ein compiler erfüllt die komplette Umsetzung der ISO-Norm;
- C++ gilt als relativ schwierig zu erlernen (daher dieser Kurs).

Erste Schritte (1)

Zur Entwicklung eines C++-Programms benötigen wir drei Dinge:

- Editor: Erstellung einer Textdatei;
- Compiler: Übersetzung des Programms in Maschinensprache des jeweiligen Rechners (Objektdatei);
- Linker: Erstellung einer ausführbaren Datei

Bezeichnungen:

- Quelldateien: .cc, .cpp
- Headerdateien: .h, h.pp

Bemerkung

Oft wird mit sog. IDEs (engl.: integrated development environment; Entwicklungsumgebungen) gearbeitet, die Editor, compiler und linker enthalten (z.B. Eclipse). Diese sind sehr hilfreich für fortgeschrittene Programmierer (da darüber hinaus sog. debugger enthalten sind), eignen sich aber zum Erlernen einer Programmiersprache nur bedingt, da diese den Blick auf die Funktionsweise eines Programms versperrt.

Erste Schritte (2)

Das vollständig einfachste Programm (zu implementieren in einem Texteditor, z.B. emacs):

```
int main () {}
```

und dann abgespeichert, z.B., als

```
step_nichts.cc
```

Übersetzung mittels eines compilers. Hier nehmen wir den GNU g++-compiler, welcher in einer Kommandozeile (bash, terminal, konsole) aufgerufen wird:

```
wick@leibniz-uni:> g++ step_nichts.cc
```

Wir schauen auf das erzeugte Ergebnis:

```
wick@leibniz-uni:> ls
```

und sehen u.a. im aktuellen Verzeichnis:

```
a.out
```

was die ausführbare Datei ist. Diese führen wir dann tatsächlich mittels

```
wick@leibniz-uni:> ./a.out
```

aus. In der Kommandozeile wird dann das Ergebnis erscheinen: hier nichts.

Erste Schritte (3)

- Ein alternativer (besserer!) compiler-Aufruf ist:

```
wick@leibniz-uni:> g++ -Wall -o step_nichts step_nichts.cc
```

- Hier wird die ausführbare Datei mittels -o explizit benannt.

→ Das hat den Vorteil, dass bei mehreren cc-Dateien jede ihre eigene Objektdatei besitzt und diese nicht ständig mit a.out überschrieben wird.

- Darüber hinaus schalten wir mittels -Wall alle Warnungen ein, die ggf. Extrahinweise mitausgibt.
- Ausführen können wir das Programm wie vorher in der Kommandozeile:

```
wick@leibniz-uni:> ./step_nichts
```

Bemerkung (Weitere Compiler)

Weitere compiler sind:

- *Microsoft Visual C++*
- *Borlands Free Command Line Tools*

Rechnen mit ganzen Zahlen

Beispiel 2: Dieses Programm führt einfache ganzzahlige Rechnungen aus:

```
int main() // So sieht ein Kommentar aus.
{
  int a, b, c; // Ganzzahlige Variablen mit Namen a, b, c.
  int d = 42;  // Ganzzahlige Variable mit Namen d und Wert 42.
  a = -5;      // Wertzuweisung.
  b = a + d;   // Zuweisung einer Summe ( 37).
  c = 6*a-d/3; // Zuweisung eines Terms (-44).
  int e = b/7; // Ganzz. Div. ohne Rest (  5).
  int zahl = b % 7;    // Divisionsrest (  2).k
  return 0;    // Fehlercode fuer shell (int aus 0..255; 0 = Erfolg).
}
```

- Frage: Wie sehen wir nun die Ergebnisse?
- Dazu benötigen wir einen output stream:

  `<iostream>`

  Dieser header enthält Objekte, um Daten von der Konsole einzulesen oder auszugeben.

Aufgabe

Man kompiliere obiges Programm jeweils mit und ohne -Wall. Was fällt auf?

Rechnen mit ganzen Zahlen und deren Ausgabe

Beispiel 2a: Dieses Programm führt einfache ganzzahlige Rechnungen aus und schreibt ein Ergebnis in das Terminal:

```
#include<iostream> // Einbinden von standard input/output stream

int main() // So sieht ein Kommentar aus.
{
  int a, b, c; // Ganzzahlige Variablen mit Namen a, b, c.
  int d = 42;  // Ganzzahlige Variable mit Namen d und Wert 42.
  a = -5;      // Wertzuweisung.
  b = a + d;   // Zuweisung einer Summe ( 37).
  c = 6*a-d/3; // Zuweisung eines Terms (-44).
  int e = b/7; // Ganzz. Div. ohne Rest (  5).
  int zahl = b % 7;    // Divisionsrest (  2).k

  std::cout << "Ergebnis von e: " << e << std::endl;
  return 0;    // Fehlercode fuer shell (int aus 0..255; 0 = Erfolg).
}
```

Bemerkung (Hash-Zeichen #)

Mit dem Hash-Zeichen # werden Präprozessor-Direktiven begonnen. Diese werden ohne Semikolon beendet. Die Hauptanwendungsgebiete von Präprozessor-Direktiven sine die Einbindung von Header- und Quelldateien, aber auch Definition symbolischer Konstanten und bedingte Kompilierung.

Ausgabe von Daten

Beispiel 3. Ausgabe aller berechneten Ergebnisse:

```
#include <iostream> // Benutze Ein-/Ausgabefunktionen.

int main() {
  int a, b, c; // Dieselben Operationen wie zuvor ...
  int d = 42;
  a = -5;
  b = a + d;
  c = 6*a-d/3;
  int e = b/7;
  int zahl = b % 7;
  std::cout // Schreibe auf Standardausgabe (Monitor, Datei, ...).
    << "Ergebnisse: a=" << a << ", b=" << b << ", c=" << c
    << ", d=" << d << ", e=" << e << ", zahl=" << zahl << "\n";

  // return 0; unnÃ¶tig: wird automatisch vom Compiler erzeugt.
}
```

Bemerkung

Mit `std` *greifen wir auf Funktionen der Standardbibliothek von C++ zurück. Die Standardbibliothek ist ein Beispiel für* ***generische*** *Programmierung, wo Algorithmen von den zugrunde liegenden Datenstrukturen getrennt werden.*

Bemerkung

Das Symbol `::` *ist der sog. scope resolution operator mit dem auf Funktionen innerhalb der Klasse/Bibliothek zugegriffen wird.*

Ein- und Ausgabe von Daten

Beispiel 4. Einlesen und Ausgabe von Daten:

```
#include <iostream>
#include <cmath>    // Benutze mathematische Funktionen.

int main() /* Auch C-Kommentare sind erlaubt. */
{
  std::cout << "Eingabe: a b c mit b > 0: ";
  int a, b, c;
  std::cin >> a >> b >> c;
  std::cout << "(a - 5) * b^(c + 1) = " // Potenz b hoch c + 1.
    << ((a - 5) * std::pow(b, c + 1)) << "\n";
  a -= 5; // <=> a = a - 5.
  ++c;    // <=> c = c + 1 <=> c += 1.
  int result = a * std::pow(b, c); // Dasselbe Ergebnis.
  std::cout << "(a - 5) * b^(c + 1) = " << result << "\n";
}
```

Ende Vorlesung 1 (Okt 17, 2017)

Warnings und Fehler (1)

Wir erinnern uns an das Beispiel von letzter Woche:

```
Hallo Algorithmisches Programmieren!
```

- Viele von Ihnen hatten den ein oder anderen **compiler-Fehler** oder sogenannte **warnings**.

$\Rightarrow$ Beim Programmieren macht man zwangsläufig Fehler und viele werden vom compiler erkannt. Es hilft nicht die Augen zuzudrücken, man muss sich diesen Fehlern und compiler-Warnungen frühstmöglich stellen.

- Einige Beispiele im folgenden.

Warnings und Fehler (2)

Compiliere mit

```
g++ bsp_1.cc
```

und

```
g++ -Wall bsp_1.cc
```

nochmals unser Beispiel von letzter Woche:

```
// bsp_1.cc
#include<iostream>

int main()
{
  std::cout << "Hallo Thomas. Und wie geht's?" << std::endl;
}
```

- Was sehen Sie in der Konsole?

$\Rightarrow$ Hoffentlich nichts.

Warnings und Fehler (3)

Compiliere mit

```
g++ bsp_2.cc
```

und

```
g++ -Wall bsp_1.cc
```

nochmals unser Beispiel von letzter Woche:

```
// bsp_2.cc
#include<iostream>

int main()
{
  int a;
  std::cout << "Hallo Thomas. Und wie geht's?" << std::endl;
}
```

- Was sehen Sie in der Konsole?
- Nichts mit `g++ bsp_2.cc`

Warnings und Fehler (4)

- Mit `g++ -Wall bsp_2.cc` ergibt sich nun aber:

```
01_a.cc: In function 'int main()':
01_a.cc:5:7: warning: unused variable 'a' [-Wunused-variable]
   int a;
       ^
```

- Was bedeutet das?
- Dies ist eine **warning**, also eine Warnung. Das Programm compiliert, teilt uns aber mit, dass wir unsauber programmiert haben.
- In der Funktion 'main' hat der compiler eine Schwachstelle entdeckt.
- Und zwar in Zeile 5, Spalte 7 gibt es eine Variable (hier a), die wir deklariert haben, aber nicht nutzen. Daher ist die ganze Variablendeklaration überflüssig und könnte weggelassen werden.

Warnings und Fehler (5)

Wir kompilieren nun folgendes Programm (mit oder ohne -Wall):

```
// bsp_3.cc
int main()
{
  int a;
  std::cout << "Hallo Thomas. Und wie geht's?" << std::endl;
}
```

- Was sehen wir in der Konsole?
- Mit g++ (ohne -Wall):

```
01_a.cc: In function 'int main()':
01_a.cc:6:3: error: 'cout' is not a member of 'std'
   std::cout << "Hallo Thomas. Und wie geht's?" << std::endl;
   ^
01_a.cc:6:51: error: 'endl' is not a member of 'std'
   std::cout << "Hallo Thomas. Und wie geht's?" << std::endl;
                                                   ^
```

- Hier liegen zwei Fehler vor und das Programm wird **nicht** kompiliert!
- Und zwar kann der compiler die Funktionen cout und endl nicht std zuordnen. Wir haben vergessen, die entsprechende Bibliothek via include einzubinden.

Symbole von C++ (1)

- Bezeichner: Namen von Objekten, wie beispielsweise Variablen, Funktionen, Klassen, etc.

→ bel. Buchstaben, Zahlen und Unterstrich, case sensitive:

```
HALLO, Hallo, hallo, HaLlO
```

sind vier verschiedene Bezeichner

- Schlüsselwörter: Bezeichner mit festgelegter Bedeutung in C++; können nicht anderwertig benutzt werden; beispielsweise ‘double’.
- Literale: Einzelne Zeichen ‘A’ und Zeichenketten “Hallo Thomas”. Nicht druckbare Zeichen werden mit einer Escape-Sequenz eingeleitet:

```
‘\t’  ‘\n’.
```

Symbole von C++ (2)

- Einfache Begrenzer: Semikolon/Strichpunkt (bekanntestes Zeichen überhaupt), Komma, geschweifte Klammern (Anweisungsblock):

```
int main (void)
{
  double a,b,c; // Komma: Deklaration mehrerer Variablen desselben Typs
  cout << "Hello Thomas!";
  return 0; // Rueckgabe Wert 0 des aufrufenden Prozesses
}
```

- Das Gleichheitszeichen: Trennung der Variablendeklaration oder Parameterlisten einer Funktion

Bemerkung

Ein Intialisierung ist keine noch keine Zuweisung! Beide nutzen des Gleichheitszeichen. Aber bei der Initialisierung wird das Gleichheitszeichen als Begrenzer genutzt und bei der Zuweisung als Operator. Bei der Initialisierung wird eine Variable angelegt, während eine Zuweisung in Bezug auf ein bereits existierendes Objekt ausgeführt wird.

Basisdatentypen

- Einfache vordefinierte Datentypen

```
bool // Wahrheitswert
int  // integer
short int
long int
float
double
long double
char // Zeichen 'Character'
wchar_t // Zeichen 'Character'
(void) // Typ, der nichts tut
```

Deklaration und Definition

- Deklaration: Bekanntmachung des Compilers mit einem Namen (Bezeichner) und Verknüpfung dieses Namens mit einem Typ.
- Syntax:

```
Typ name;
Typ name1, name2, name3;
```

- Mit einer Deklaration geben wir dem Compiler nur Informationen zum Typ bekannt. Es wird noch kein Maschinencode erzeugt; also insbesondere wird noch kein Speicherobjekt (Variable) angelegt.
- Definition: Anlegen des konkreten Speicherobjekts.
- Jede Definition ist automatisch eine Deklaration.

Variablen

- Stelle (Adresse) im Hauptspeicher (RAM);
- Ablegen und Zugriff eines Wertes;
- Neben Adresse auch Zuweisung eines Bezeichners;
- Syntax:

```
long mvar; // Typ: long, Bezeichner: mvar
```

- An welcher Adresse im Arbeitsspeicher Speicherplatz reserviert wird, können wir nicht beeinflussen.

Datentyp bool

- Beschreibung von Wahrheitswerten

```
bool flag_bool;
flag_bool = true; // Schalter auf wahr setzen
flag_bool = false; // Schalter auf falsch setzen
```

Datentyp char (1)

- `char` ist der grundlegende Datentyp für Zeichen;
- belegt normalerweise ein Byte an Speicher, d.h., $2^8 = 256$ Ausprägungen;
- der komplette ASCII-Zeichensatz `http://www.asciitable.com/` findet Platz
- Syntax:

```
char ch1 = 'A';
char ch2 = 'B';
```

- Alternativ ginge auch

```
char ch1 = 65; // laut ASCII-Tabelle das Zeichen 'A'
char ch2 = 66; // laut ASCII-Tabelle das Zeichen 'B'
```

Datentyp char (2)

- Neben ‘normalen’ Zeichen enthält der ASCII-Zeichensatz auch Sonderzeichen und nicht-druckbare Zeichen.
- Beispiele:

```
\n // Escape-Zeichen, Zeilenvorschub, newline
\t // Tabulator-Zeichen
\" // " wird ausgegeben
```

- Den gesamten ASCII-Zeichensatz finden wir hier: `http://www.asciitable.com/`

Bemerkung

Einzelne Zeichen werden in C++ in einfache Anführungszeichen geschlossen, während Zeichenketten (strings) von doppelten Anführungszeichen umschlossen werden.

Datentyp int (1)

- Ganze Zahlen $\mathbb{Z}$ werden mit Hilfe von `int` dargestellt;
- Natürliche Zahlen $\mathbb{N}$ werden mit Hilfe von `unsigned int` dargestellt.
- Beispiel:

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

int main (void) {
  int wert1 = 10;
  int wert2 = 21;
  cout << "Wert 1 ist: " << wert1 << endl;
  cout << "Wert 2 ist: " << wert2 << endl;
  cout << "int-Zahlenbereich ist: " << INT_MIN << " bis " << INT_MAX << endl;
  return 0;
}
```

- `short` und `long` gehören ebenfalls zur int-Familie.

Datentyp int (1)

Beispiel 10:

```
#include <iostream>
#include <limits>  // Eigenschaften arithmetischer Typen.

int main()
{
  int i_min = std::numeric_limits<int>::min(); // Minimalwert.
  int i_max = std::numeric_limits<int>::max(); // Maximalwert.
  unsigned int u_min = std::numeric_limits<unsigned int>::min();
  unsigned int u_max = std::numeric_limits<unsigned int>::max();
  int          iu_min = u_min,  iu_max = u_max; // Konversion!
  unsigned int ui_min = i_min,  ui_max = i_max;

  std::cout <<  i_min << " " <<  i_max << "\n"
            <<  u_min << " " <<  u_max << "\n"
            << iu_min << " " << iu_max << "\n"
            << ui_min << " " << ui_max << "\n";
}
```

Die Datentypen float, double, long double (1)

- Reelle Zahlen $\mathbb{R}$ werden mit Hilfe von `float` bzw. `double` dargestellt;
- Die drei Datentypen unterscheiden sich durch deren Genauigkeit.

  Vorsicht: In der Numerik und im wissenschaftlichen Rechnen rechnen wir oft mit Abbruchkriterien und Fehlerabschätzungen in sehr niedrigen Zahlenbereichen $\sim 10^{-10}$. Hier sollte man grundsätzlich mit `double` arbeiten, um die Genauigkeiten solcher Rechnungen nicht seitens C++ einzuschränken.
- Syntax:

```
float = 1.74;
double = 3.75843;
```

Die Datentypen float, double, long double (1)

Beispiel 11: Einfach und doppelt genaue Gleitkommazahlen.

```
#include <iostream>

int main()
{
  float a = 12.78, b = -3.14; // "Reelle" Variablen (Gleitkomma, floating point).
  float s = a + b, d = a - b, p = a * b, q = a / b;
  // Divisionsrest nicht sinnvoll: float r = a % b;

  std::cout.precision(10); // Ausgabe gerundet auf 10 Dezimalstellen.
  std::cout << "Einfach genaue Rechnung:\n"
    << "a = " << a << "\nb = " << b << "\ns = " << s << "\n"
    << "d = " << d << "\np = " << p << "\nq = " << q << "\n\n";

  { // Neuer Block: Namen redefinierbar.
    double a = 12.78, b = -3.14; // Doppelt genaue reelle Variablen.
    double s = a + b, d = a - b, p = a * b, q = a / b;
    std::cout << "Doppelt genaue Rechnung:\n"
      << "a = " << a << "\nb = " << b << "\ns = " << s << "\n"
      << "d = " << d << "\np = " << p << "\nq = " << q << "\n";
  }
}
```

Gemischte Arithmetik

Beispiel 13: Rechenoperationen mit gemischten Datentypen.

```
#include <iostream>

int main()
{
  int    const a = 1;
  float  const b = 1; // Konversion int -> float.
  double const c = 1; // Konversion int -> double.

  std::cout.precision(16);
  std::cout << "\nKehrwerte von (unsigned) int.\n";
  for (unsigned int k = 1U; k <= 20U; ++k) // 1U = 1u = unsigned int 1.
    std::cout << k << " " << a / k << " " << b / k << " " << c / k << "\n"
        << k << " " << 1 / k << " " << 1.0F / k << " " << 1.0 / k << "\n";
  std::cout << "\nKehrwerte von float.\n";
  for (float k = 1; k <= 20; ++k)
    std::cout << k << " " << a / k << " " << b / k << " " << c / k << "\n"
      << k << " " << 1 / k << " " << 1.0F / k << " " << 1.0 / k << "\n";
  std::cout << "\nKehrwerte von double.\n";
  for (double k = 1; k <= 20; ++k)
    std::cout << k << " " << a / k << " " << b / k << " " << c / k << "\n"
      << k << " " << 1 / k << " " << 1.0F / k << " " << 1.0 / k << "\n";
  std::cout << "\nGeometrische Folge.\n";
  for (double k = 1; 1 / k >= 0.0001; k *= 1.5) // Auch das geht ...
    std::cout << k << " " << 1 / k << "\n";
}
```

Fallunterscheidungen (1)

Beispiel 5. Dieses Beispiel führt abhängig von der Eingabe verschiedene Befehle aus:

```
#include <iostream>
#include <cmath>

int main()
{
  int a, b;
  std::cout << "Eingabe: a b fuer a^b: ";
  std::cin >> a >> b;
  if (b < 0) {
    std::cout << "Fehler: b < 0.\n";
    return 1; // Vorzeitiger Ruecksprung (mit Fehlercode).
  } else if (b == 0) { // Achtung: == ist Vergleich, = ist Zuweisung!
    std::cout << "a^b = 1.\n";
  } else { // b > 0.
    std::cout << "a^b = " << std::pow(a, b) << ".\n";
  }
}
```

Fallunterscheidungen (2)

Beispiel 6: (Beispiel 5 etwas vereinfacht)

```
#include <iostream>
#include <cmath>

int main()
{
  int a, b;
  std::cout << "Eingabe: a b fuer a^b: ";
  std::cin >> a >> b;
  if (b < 0) {
    std::cout << "Fehler: b < 0.\n";
    return 1;
  }
  // Hier ist b >= 0.
  // Den Fall b == 0 behandelt std::pow() korrekt.
  std::cout << "a^b = " << std::pow(a, b) << ".\n";
}
```

Fallunterscheidungen (3)

Beispiel 7: (weitere Variante von Beispiel 5)

```
#include <iostream>
#include <cmath>

int main()
{
  int a, b;
  std::cout << "Eingabe: a b fuer a^b: ";
  std::cin >> a >> b;
  if (b >= 0) // Nur 1 Befehl: { } unnoetig.
    std::cout << "a^b = " << std::pow(a, b) << ".\n";
  else
    std::cout << "Fehler: b < 0.\n";
}
```

Schleifen: for

Beispiel 8: Mehrmalige Ausführung eines Befehlsblocks mit verschiedenen Daten

```
#include <iostream>
#include <cmath>

int main()
{
  unsigned int n; // Nichtnegative ganzzahlige Variable n.
  std::cout << "Eingabe von n: ";
  std::cin >> n;
  std::cout << "Tabelle mit k, k^2, 2^k fuer k <= " << n << ".\n";
  for (unsigned int k = 0; k <= n; ++k) {
    unsigned int power = std::pow(2, k); // Lokale Variable.
    std::cout << k << " " << k * k << " " << power << "\n";
  }
  // Hier ist 'power' nicht definiert: nur inerhalb { ... }.
}
```

Schleifen: while

Grundgerüst: Ausführung der Anweisungen solange der Ausdruck in der while-Anweisung falsch wird. Danach wird das Programm hinter dem Block fortgesetzt.

```
while (Bedingung)
{
  // Abarbeiten der Bedingungen
}
```

Beispiel 8a:

```
# include <iostream>
using namespace std;
int main (void) {
 int var = 1;
 while (var <= 6) {
   cout << var;
   var++; // var = var + 1
 }
 return 0;
}
```

Bemerkung

Ein häufiger Fehler bei while-Schleifen ist das Update der Abbruchbedingung (hier var++), sodass es dann zu einer Endlosschleife kommt, die man nur noch gewaltsam abrechen kann. Andererseits sind manchmal Endlosschleifen erwünscht, wenn man auf ein bestimmtes Ereignis wartet oder eine Anweisung dauerhaft überwachen will.

Funktionen (1)

- Funktionen sind kleine Unterprogramme, mit denen Daten verarbeitet werden oder Teilprobleme gelöst werden.
- Funktionen dienen insbesondere zur Struktuierung eines Codes.
- Erste und wichtigste Funktion ist die main-Funktion.
- In C++ gibt es keinen standardisierten Weg, Funktionen parallel auszuführen, sodass hier immer auf externe Bibliotheken zurückgegriffen werden muss.
- Funktionen werden erst deklariert und dann definiert (wie Variablen!)
- Syntax:

```
[Spezifizierer] Rueckgabetyp Funktionsname (Parameter);
```

- Rueckgabetyp: Datentyp: int, string, etc. Falls keiner, dann `void`.
- Funktionsname: eindeutiger Name; sollte nicht ident. mit Namen der Laufzeitbibliothek sein.
- Parameter: optional. Mehrere sind möglich und durch Komma getrennt.

Funktionen (2)

Beispiel 9. Hier sind Funktionen für Quadrat und Zweierpotenz definiert:

```
#include <iostream>
#include <cmath>

unsigned int square (unsigned int a)
{
  return a * a;
}

unsigned int pow2 (unsigned int a)
{
  return std::pow(2, a);
}

int main() // Auch main() ist eine Funktion.
{
  unsigned int n;
  std::cout << "Eingabe von n: ";
  std::cin >> n;
  std::cout << "Tabelle mit k, k^2, 2^k fuer k <= " << n << ".\n";
  for (unsigned int k = 0; k <= n; ++k)
    std::cout << k << " "
      << square(k) << " " << pow2(k) << "\n";
  // Auch k ist lokal: nur innerhalb der for-Schleife definiert.
}
```

Funktionen (3)

Beispiel 14: mathematische Funktionen. Viele sind in der Standardbibliothek `std` verfügbar:

```
#include <iostream>
#include <cmath>

int main()
{
  double x;  std::cout << "\nFunktionen y = f(x) mit x = ";  std::cin >> x;
  double sin_x   = std::sin(x),  asin_y = std::asin(sin_x); // Auch cos, tan.
  double sinh_x  = std::sinh(x); // Auch cosh, tanh, und seit 2014 asinh, ...
  double exp_x   = std::exp(x),       log_y = std::log(exp_x);
  double exp10_x = std::pow(10.0, x), log10_y = std::log10(exp10_x);
  double sqrt_x  = std::sqrt(std::abs(x));

  std::cout.precision(16);
  std::cout
    << "\nsin(x)\t= " << sin_x << ",\tarcsin(y) = " << asin_y
    << "\nsinh(x) = " << sinh_x << ","
    << "\nexp(x)\t= " << exp_x << ",\tln(y)\t= " << log_y
    << "\n10^x\t= " << exp10_x << ",\tlog(y)\t= " << log10_y
    << "\n|x|^1/2 = " << sqrt_x << ",\ty^2\t= " << sqrt_x * sqrt_x
    << "\nfloor(x) = " << std::floor(x)
    << ",\tceil(x) = " << std::ceil(x) << "\n";
}
```

Funktionen (4)

Beispiel 15: Wurzelberechnung mit dem Newton-Verfahren

```
#include <iostream>
#include <limits>
#include <cmath>

double my_sqrt(double x, double eps = 1.0e-10) // Eps hat Default-Wert. {
  x = (x < 0) ? -x : +x; // Setze x = |x| (-x falls negativ, +x sonst).
  double s = x / 2;
  double e = std::max(16 * std::numeric_limits<double>::epsilon(), eps);

  for (/* leer */; std::abs(s * s - x) > e; /* leer */)
    s = (x / s + s) / 2;
  return s; // Ergebnis der Newton-Iteration für 0 = f(s) = x - s^2.
}
int main() {
  double x, eps;
  for (;; std::cin.clear()) { // Endlosschleife mit Rücksetzen von std::cin.
    std::cout << "Eingabe: x [eps] ";
    std::cin >> x;
    if (std::cin.fail()) // Keine Zahl eingegeben: Abbruch.
      break;
    std::cin >> eps;
    // Rufe my_sqrt() mit oder ohne eps auf, je nach Eingabe.
    double s = std::cin.good() ? my_sqrt(x, eps) : my_sqrt(x);
    std::cout << "s = " << s << ", x - s^2 = " << (x - s * s) << "\n";
    // Weder Zahl noch EOF (CTRL-D) eingegeben: Abbruch.
    if (std::cin.fail() && !std::cin.eof()) // Logisch 'und': &&, 'nicht': !.
      break;
  }
}
```

Funktionen (5)

Nach den einführenden Beispielen nun nochmal alles im Detail:

- Funktionsdeklaration:

```
int point ( double x_pos, double y_pos);
```

- Es ginge auch:

```
int point ( double, double );
```

  Aber dies ist kein guter Stil, da nicht klar ist, wofuer die Parameter später verwendet werden sollen.

- **Merke:** Namen von Variablen, Funktionen und Parametern sollten möglichst selbsterklärend sein!!

Funktionen (6): Aufruf und Parameterübergabe

- Aufruf einer Funktion ohne Parameter:

```
my_function ();
```

- Beispiel:

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

// Funktionsprototyp (Deklaration)
void my_function ();

int main ()
{
  // Aufruf der Funktion
  my_function ();
}

// Funktionsdefinition
void my_function ()
{
 cout << "Ich bin in der Funktion drin." << endl;
}
```

Funktionen (7): Aufruf und Parameterübergabe

- Parameterübergabe an Funktionen (call-by-value)
- Funktion mit Parameter:

```
my_function (int year);
```

- Aufgabe: Programmiere die Funktion, die dann das aktuelle Jahr ausgibt.
- Testen Sie bitte auch, was passiert, wenn kein integer value, sondern eine Gleitkommazahl übergeben wird.

Bemerkung

Neben call-by-value gibt es noch den Aufruf call-by-reference, wo nicht der Wert selbst, sondern dessen Adresse weitergegeben wird. Beide Aufrufe können verschiedene Effekte erzeugen! Später mehr.

Funktionen (8): Rückgabewerte

- Funktionen werden mächtiger, wenn diese ein Ergebnis zurückgeben;
- Die Rückgabe geschieht mittels `return`;
- Das bekannteste Beispiel ist:

```
int main () {
 // Anweisungen
 return 0;
}
```

- Aufgabe: Programmieren Sie eine Funktion, die die Maße dieses Raums übergeben bekommt und dann das Volumen berechnet und mittels `return` zurückgibt.

Funktionen (9): Lokale und globale Variablen

- Wir haben bereits gesehen, dass Funktionen Parameter übergeben bekommen können, als auch Werte zurückgeben können.
- Es können innerhalb der Funktion aber auch neue Variablen definiert werden.
- Solche Variablendeklarationen sind **lokal**.
- Diese sind lediglich innerhalb des Funktionsrumpfs gültig.
- Im Gegensatz dazu können ausserhalb von Funktionen **globale** Variablen definiert werden, die überall gültig sind (selbst in der main-Funktion)
- Eine gute Regel ist: **Variablen sollten so lokal wie möglich und so global wie nötig definiert werden**.
- Warum? übersichtlichkeit des Codes!
- Frage: Was passiert wenn zwei Variablen (eine lokale und eine globale) denselben Namen haben?

$\Rightarrow$ Die lokale Variable erhält den Zuschlag!

- Aufgabe: Testen Sie dies bitte selbst!

Funktionen (10): Standardparameter

- Die Parameter von Funktionen können mit Standardwerten besetzt werden.
- Achtung: dieses Thema ist zwar logisch aufgebaut, aber trotzdem nicht leicht!
- Beispiel:

```
void func (int var = 66)
```

- Falls die Funktion ohne Argument aufgerufen wird, dann erhält var den Wert 66:

```
func ();
```

- Die Funktion kann aber natürlich mit einem Parameter aufgerufen werden:

```
func (78);
```

- Dann hat var den Wert 78 im Funktionsrumpf.

Funktionen (11): mehrere Standardparameter

- Bei mehreren Parametern wird es trickreicher
- Die Zuordnung der Parameter erfolgt von links nach rechts
- Wenn also ganz links ein Standardparameter gesetzt wird, dann müssen alle nachfolgenden Parameter ebenfalls Standardwerte bekommen:

```
void func (int param_1 = 5, int param_2 = 78, int param_3 = 3)
```

- Der umgekehrte Fall ist aber möglich:

```
void func (int param_1, int param_2, int param_3 = 3)
```

- Hier wird der letzte Parameter mit einem Standardwert initialisiert.
- Es existieren also zwei mögliche Funktionsaufrufe:

```
func (23,45); // letzter Parameter wird auf 3 gesetzt
func (45,67,46); // letzter Parameter bekommt den Wert 46.
```

Funktionen (12): Überladen

- In C++ können Funktionen überladen werden, d.h.,
- die Funktionen haben denselben Namen
- unterscheiden sich aber hinsichtlich Parametertypen und Anzahl der Parameter
- Beispiel:

```
int func (int param);
int func (double param);
```

- Wenn die Funktionsparameter unterschiedlich sind, dann können desweiteren auch die Rückgabewerte unterschiedlich sein.
- Beispiel:

```
int func (int param);
long double func (double param);
```

- Warum funktioniert das?

⇒ C++ identifiziert Funktionen nicht anhand des Namens, sondern mittels ihrer Signatur. Die Signatur entsteht bei der Deklaration aus einer Kombination von Namen und Parameterliste.

- Der compiler findet also die richtige Funktion beim Übersetzen in Maschinensprache.

Funktionen (13): Überladen - Beispiele

Beispiele:

- Mathematische Funktionen wie

```
abs(...); log(...); pow(...); sin(...)
```

- Selbstdefinierte Funktionen:

```
float norm_squared(float a, float b)
{
  return std::sqrt(a*a + b*b);
}
double norm_squared(double a, double b)
{
  return std::sqrt(a*a + b*b);
}
double norm_squared(double a, double b, double c)
{
  return std::sqrt(a*a + b*b + c*c);
}
```

Funktionen (14): Überladen - Beispiele

Aber:

- implementiere und teste:

```
float norm_squared(float a, float b)
{
  return std::sqrt(a*a + b*b);
}


double norm_squared(float a, float b)
{
  return std::sqrt(a*a + b*b);
}
```

- Was gibt der compiler zurück?

Grundlagen

- Wir unterscheiden Operatoren anhand der Anzahl ihrer Operanden:
  - Untärer Operator
  - Binärer Operator
  - Ternärer Operator
- Neben der Anzahl wird auch die Position unterschieden:
  - Präfix - der Operator steht vor dem Operanden
  - Postfix - der Operator steht hinter dem Operanden
  - Infix - der Operator steht zwischen den Operanden
- Assoziativität: Links- und Rechtsassoziativität
- Mehrzahl der Operatoren ist linksassoziativ
- Beispiel:

```
var1 + var2 - var3 // zuerst wird var1 + var2 berechnet
var1 + (var2 - var3) // zuerst wird var2 - var3 berechnet
```

Arithmetische Operatoren

```
+ // Addition
- // Subtraktion
* // Multiplikation
/ // Division
% // Rest einer Division
```

- Es gelten die üblichen mathematischen Regeln (Punkt-vor-Strich-Regelung)
- Artithmetische Operatoren sind binäre Operatoren

```
+= // var1 += var2 entspricht var1 = var1 + var2
   // ebenso fuer die vier weiteren arith. Operatoren
```

Inkrement- und Dekrementoperator

```
++ // Inkrement (Variable um 1 erhoehen)
-- // Dekrement (Variable um 1 verringern)
```

- Hier gibt es Postfix- und Präfixschreibweisen

Logische Operatoren und Vergleiche

```
!  // Logisches Nicht
&& // Logisches Und
|| // Logisches Oder (inkl.)
<  // Kleiner
>  // Groesser
== // Gleichheit
!= // Ungleichheit
<= // Kleiner-Gleich
>= // Groesser-Gleich
```