Algorithmisches Programmieren (Numerische Algorithmen mit C++)

Marc C. Steinbach, Jan Philipp Thiele, Thomas Wick

Leibniz Universität Hannover
Institut für Angewandte Mathematik

Stand 16. Dezember 2021

Inhalt

Literatur

Bücher

- Jürgen Wolf: C++ von A bis Z, Galileo Computing, 2008
- Ulrich Kaiser, Martin Guddat: C/C++: Das umfassende Lehrbuch, Rheinwerk Verlag, 2014

Websites zum Nachlesen und Nachschlagen

- `https://en.cppreference.com` (`https://de.cppreference.com`)
- `https://www.cplusplus.com`
- `https://www.isocpp.org`

Videotutorials

- Derek Banas hat ein umfassendes zusammenhängendes C++ Tutorial:

  `https://www.youtube.com/watch?v=6y0bp-mnYU0`

  Die Reihenfolge des Videos und des Skripts stimmen zwar nicht überein, aber die einzelnen Themen sind in der Videobeschreibung mit Zeitangaben verlinkt.
- Auf dem Kanal TheCherno gibt es einzelne Videos zu verschiedenen C++-Themen, die auch in einer Playlist zusammengefasst sind:

  `https://www.youtube.com/user/TheChernoProject`

Was ist C++?

Eigenschaften

- C++ ist eine objektorientierte höhere Programmiersprache; aber keine *reine* objektorientierte Sprache (aus C entstanden: Vor- und Nachteile).
- C++ umfasst die maschinennahe Sprache C fast vollständig.
- C++ wird zur Ausführung in Maschinensprache übersetzt (*compiliert*).
- Man kann mehrere separat compilierte Programmteile kombinieren.
- C++ ist vielseitig und teilweise sehr kompliziert (Vererbung, Objektorientierung, Templates, Polymorphie, Überladen von Funktionen).
- In C++ können komplexe Algorithmen sehr effizient programmiert werden. Es bildet daher die Grundlage für einige sehr erfolgreiche Codes im Wissenschaftlichen Rechnen, Numerik und Optimierung: 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).

Was ist C++?

Geschichte: `https://www.cplusplus.com/info/history`

- 1979 von Bjarne Stroustup entwickelt als Erweiterung der Sprache C um ein Klassenkonzept ("C with classes").
- C diente als Ursprung, weil dieses schnellen Code erzeugt und einfach auf andere Plattformen portiert werden kann.
- C war damals die am weitesten verbreitete Sprache.
- 1982: Umbenennung in C++.
- 1998: erste Normierung von der ISO (ISO/IEC 14882:1998).
- Seither gibt es einige neuere Standards: C++11, C++14, C++17, C++20, demnächst C++23.

Wissenschaftliches Rechnen/Numerik/Numerische Optimierung
Zunächst dominierte Fortran. Mit wachsender Komplexität der Anwendungen zunehmende Verwendung von C und C++.

Stärken von C++

- Maschinennahes Programmieren.
- Erzeugung von hocheffizientem Code.
- Hohe Ausdrucksstärke und Flexibilität.
- Für große (Programmier-)Projekte geeignet (s.o. für Wissenschaftliches Rechnen, Numerik, Optimierung).
- Sehr weit verbreitet.
- Open Source.
- C-kompatibel, da aus C entstanden.

Schwächen von C++

- Da C++ auf C basiert, sind einige Teile compiler-spezifisch.

$\Rightarrow$ Dies erschwert die Portierung auf verschiedene Rechnertypen, Betriebssysteme (Windows, MacOS, Linux) und Compiler.

- Kaum ein compiler setzt die volle ISO-Norm um.
- C++ gilt als relativ schwierig zu erlernen (daher dieser Kurs).

Erste Schritte (1)

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

- Editor (z.B. emacs, eclipse, vim, gedit, kate, ...):
  Erstellen/Bearbeiten einer Textdatei.
- Compiler (z.B. gcc, clang):
  Übersetzen des Programms in Maschinensprache des jeweiligen Rechners (Objektdatei).
- Linker (Aufruf meist über Compiler): Erstellen einer ausführbaren Datei.

Dateiendungen:

- Quelldateien: `.cc`, `.cpp` (`.c` für C)
- Headerdateien: `.hh`, `.hpp` (`.h` für C)

Bemerkung

Oft wird mit einer IDE (Entwicklungsumgebung, engl. integrated development environment) gearbeitet, die Editor, Compiler und Linker vereint (z.B. Eclipse). IDEs sind nützlich für fortgeschrittene Programmierer (da weitere Hilfswerkzeuge, z.B. Debugger, enthalten sind). Sie eignen sich aber nur bedingt zum Erlernen einer Programmiersprache, da sie den Blick auf die Funktionsweise versperren.

Erste Schritte (2)

Beispiel 1. Öffnen Sie bitte zunächst eine neue leere Textdatei mit einem Editor Ihrer Wahl. Tippen Sie bitte darin ein:

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

und speichern sie das als `step_nichts.cc`. Dies ist das einfachste vollständige Programm. Es tut nichts, wie wir im folgenden sehen werden:

- Das Programm wird mithilfe eines Compilers in ausführbaren Code übersetzt.
- Wir benutzen den GNU-compiler g++, der wie folgt in der Kommandozeile aufgerufen wird:

  ```
  prof@luh:> g++ step_nichts.cc
  ```

- Wir lassen den Inhalt des aktuellen Ordners ausgeben mit

  ```
  prof@luh:> ls
  ```

  und sehen die neue Datei a.out.

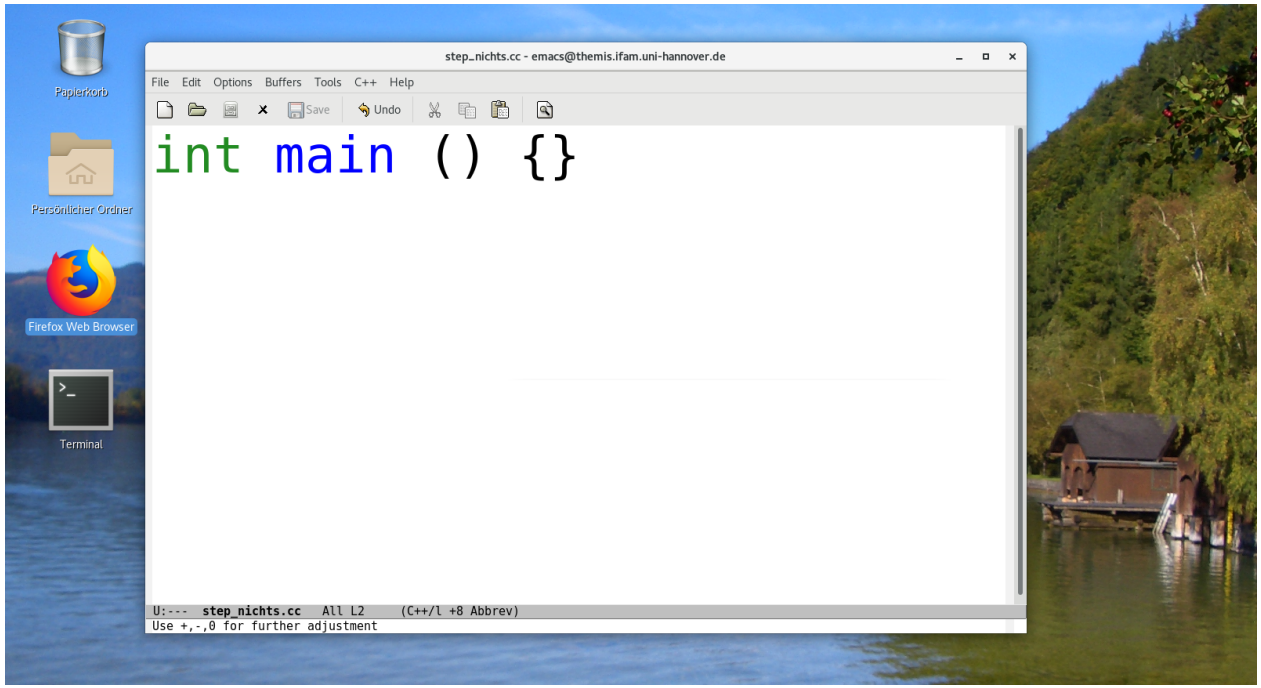
- Dies ist unsere ausführbare Datei, die wir mit

  ```
  prof@luh:> ./a.out
  ```

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

Erste Schritte (3)

- Lassen Sie uns die vorherige Folie im Detail anschauen.
- In dem Editor Ihrer Wahl, z.B. bei mir emacs, sieht das Ganze so aus:



Erste Schritte (4)

- Wofür stehen nun `int`, `main`, `()`, `{}`?
- `int` steht für den Rückgabetyp des Programms: wenn ein Programm beendet wird, gibt es einen Fehlercode an die aufrufende Stelle im Computer zurück (Terminal/Konsole/shell). Auch der Compiler gibt einen `int`-Wert zurück.
- `main` ist der Name der Hauptfunktion: diese muss immer `main` heißen.
- `()` ist ein einfaches Klammerpaar, welches Argumente an die main-Funktion übergeben könnte. Hier ist dieses Klammerpaar leer und wir geben daher keine weiteren Argumente an. Später werden wir Beispiele kennenlernen, wo solche Argumente durchaus relevant sind.
- `{}` Die geschweiften Klammern beinhalten das auszuführende Programm selbst. Diese treten in C++ sehr häufig auf und kennzeichnen sogenannte Blöcke. Zwischen den geschweiften Klammern steht also, was das Programm tun soll: hier steht nichts. Also tut das Programm nichts.

Bemerkung

Eine sehr gute Einführung und Beschreibung der ersten Schritte finden Sie auf https://www.programmierenlernen24.de/c-tutorial/

Erste Schritte (5)

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

```
prof@luh:> 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.

- Darüber hinaus schalten wir mit `-Wall` alle Warnungen ein, d.h. gegebenenfalls werden zusätzliche Hinweise mit ausgegeben.
- Ausführen können wir das Programm wie vorher in der Kommandozeile:

```
prof@luh:> ./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:

```c
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).
  return 0;     // Fehlercode für shell (int aus 0..255; 0 = Erfolg).
}
```

Aufgabe

Kompiliert obiges Programm jeweils mit und ohne Warnings. Was fällt auf?

Frage

Wie sehen wir nun die Ergebnisse?

Ausgabe von Rechenergebnissen

Beispiel 3. Dieses Programm führt einfache ganzzalige Rechnungen aus und schreibt ein Ergebnis in das Terminal:

```
#include <iostream> // Einbinden vom standard intput/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).

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

Der Übersetzungsprozess

Damit aus unserem Quellcode (`.cc`, `.hh`) eine ausführbare Datei wird, werden (oftmals automatisch im Hintergrund wohlgemerkt!) mehrere Schritte durchlaufen:

1. Der Präprozessor löscht Kommentare, arbeitet Präprozessor-Direktiven wie `#include` ab und interpretiert Makros (hier nicht behandelt).
2. Das Frontend interpretiert die spezifische Syntax und erzeugt eine allgemeine binäre Datenstruktur.
3. Die Zwischenschicht führt eine allgemeine Optimierung des Codes aus und ersetzt beispielsweise Multiplikationen mit 2 durch eine Linksschiebeanweisung.
4. Bei der Codeerzeugung wird das Ergebnis der Zwischenschicht in eine Objektdatei umgewandelt. Dabei wird das Programm auch für das Zielsystem optimiert.
5. Der Linker verbindet mehrere Objektdateien zu einer ausführbaren Datei. Dieser Schritt ermöglicht es, umfangreiche Programme in mehrere Quelldateien aufzuteilen und Bibliotheken einzubinden.

Ein- und Ausgabe von Daten

Beispiel 4. Ein einfacher „Taschenrechner“ auf der Konsole:

```
#include <iostream>

int main()
{
  std::cout << "Eingabe: a b: "; // Hinweis an den Anwender
  int a, b, c; // Variablen
  std::cin >> a >> b >> c; // Einlesen der Daten vom Terminal
  std::cout << "a+b: " << a+b << std::endl;
  std::cout << "a-b: " << a-b << std::endl;
  std::cout << "a*b: " << a*b << std::endl;
}
```

Symbole in C++ (1)

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

→ Beliebige Buchstaben, Zahlen und Unterstriche; müssen mit einem Buchstaben beginnen.

→ Bezeichner sind case sensitive:
`HALLO`, `Hallo`, `hallo` und `HaLlO`
sind vier verschiedene Bezeichner.

- Schlüsselwörter: Bezeichner mit festgelegter Bedeutung in C++;
können nicht anderweitig benutzt werden; beispielsweise `double`
- Literale: Zahlen `-42`, `3.1415`, Textzeichen `'A'`, `'?'`, Zeichenketten `"Hallo."`.
Nicht druckbare Steuerzeichen werden mit dem Escape-Zeichen \ gebildet, z.B. `'\t'` (Tabulator) und `'\n'` (Zeilenumbruch).

Bemerkung (Unterschied endline und Zeilenumbruch)

Sowohl `'\n'` als auch `std::endl` (endline) erzeugen bei der Ausgabe einen Zeilenumbruch. Endline führt aber zusätzlich dazu, dass alle bisher von cout im Arbeitsspeicher abgelegten Zeichen sofort auf der Konsole ausgegeben werden. Endline führt also letztendlich zu mehr Ausgabeoperationen, aber beim Absturz des Programms werden mit `'\n'` eventuell nicht alle Zeilen ausgegeben.

Symbole in C++ (2)

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

```
int main(void) // Hier ist void optional; heute unüblich
{
  double a,b,c; // Komma: Mehrere Variablen desselben Typs
  std::cout << "Hello!";
  return 0; // Rückgabewert 0 des aufrufenden Prozesses
}
```

- Das Gleichheitszeichen: Trennung von Deklaration und Initialisierung für Variablen oder Funktions-Parameter

Bemerkung

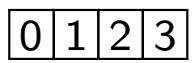
Man unterscheidet Initialisierung und Zuweisung! Beide nutzen '='. Aber bei der Initialisierung wird '=' als Begrenzer genutzt und bei der Zuweisung als Operator. Bei der Initialisierung wird eine Variable neu angelegt, während die Zuweisung den Wert einer bereits angelegten Variablen ändert.

Deklaration/Definition von Variablen

- Bekanntmachung der Variablen mit Datentyp und Namen + Anlegen im Speicher (optional initialisiert, sonst mit zufälligem Wert!)
- Mit Anlegen im Speicher ist die Variable definiert, andernfalls nur deklariert (Ausnahmefall, nicht in diesem Kurs!)
- Syntax:

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

unsigned int mvar; // Typ konkret: unsigned int, Name: mvar
```

- Ablegen und Zugriff eines Wertes.
- Stelle (Adresse) im Hauptspeicher (RAM),
  der als endliche Byte-Sequenz organisiert ist: 

- An welcher Adresse im Hauptspeicher Speicherplatz reserviert wird, können wir nicht beeinflussen. Nur Adresse 0 ist ungültig und wird garantiert nie belegt.

Häufig verwendete Basisdatentypen

```
          bool   // Wahrheitswert (logische Variable)
          int    // Integer (Ganzzahl)
unsigned int     // vorzeichenlose Ganzzahl (nichtnegativ)
          float  // Floating Point Number (Gleitkommazahl)
          double // double precision float (Gleitkommazahl)
          char   // Character (Zeichen)
          void   // leerer Typ (Sonderfall: später)
```

- Boolesche Variablen können nur die Werte `true` und `false` annehmen.
- Den arithmetischen Datentypen (Zahlen) werden wir uns in einem späteren Kapitel ausführlich widmen. Wir haben diese aber teilweise bereits verwendet.
- Zeichen belegen normalerweise nur einen Speicherplatz (Byte) und bieten damit Platz für den kompletten ASCII-Zeichensatz (`https://www.asciitable.com`).
- Zeichen werden in C++ in einfache Anführungszeichen eingeschlossen `'A'`, während Zeichenketten (Strings) von doppelten Anführungszeichen umschlossen werden `"Hallo!"`. Jeder String hat am Ende das unsichtbare Null-Zeichen `'\0'`; so muss der Compiler nicht die String-Länge abspeichern. Beispielsweise besteht `"A"` aus den beiden Zeichen `'A'` und `'\0'`.
- Der leere Datentyp wird uns im Kapitel Funktionen noch einmal begegnen.

Ende VL 1

Aufgabe

- Falls ihr den GNU-Compiler noch nicht installiert haben solltet, findet ihr auf Blatt 1 Erklärungen, wie das auf Windows und MacOS funktioniert.
- **Arbeitet die Beispiele aus dem ersten Kapitel und vom Blatt zur Vertiefung durch**.
- Es ist besonders am Anfang ganz normal, dass euch (Tipp-)Fehler unterlaufen.
- Wenn ihr nicht weiterkommt, dann schaut ebenfalls im Kapitel Warnings und Fehler nach. Dort werden ein paar der häufigen Fehler behandelt.
- Darüber hinaus sind die eingangs bereitgestellten Internet-Links sehr hilfreich.

Lokale Gültigkeitsbereiche (1)

- Anweisungen zwischen zwei geschweiften Klammern {} werden als Block bezeichnet.
- Alle in einem Block deklarierten Variablen sind lokal und nur dort gültig.
- Blöcke werden bei Kontrollstrukturen, Funktionen und Klassen verwendet.
- Wenn mehrere Variablen denselben Namen haben, dann bekommt immer die (innerste) lokale Variable den Zuschlag. Dies gilt insbesondere auch für Funktionen und Klassen.

Beispiel 5. Dieses Programm zeigt die Gültigkeit lokaler Variablen:

```
#include <iostream>
int main()
{
  int a = 5;
  { // Neuer Block
    int a = 10; // Neue Variable a, nur in diesem Block bekannt
    std::cout << a << std::endl; // Inneres a bevorzugt -> 10

    // Auf das äußere a kann man in diesem Block nicht zugreifen
  }
  std::cout << a << std::endl; // Inneres a existiert nicht mehr -> 5
}
```

Lokale Gültigkeitsbereiche (2)

- Es können auch globale Variablen definiert werden (außerhalb jeglicher Funktionen oder später Klassen). Dies sollte man aber unbedingt vermeiden!
- Mit dem Scope-Operator `::` kann auf diese globalen Variablen zugegriffen werden.

Beispiel 6. Zugriff auf globale Variablen:

```
#include <iostream>

int a = 20; // Globale Variable
int main()
{
  std::cout << a << std::endl; // Nur globales a bekannt -> 20
  int a = 5;
  std::cout << a << std::endl; // Ab hier "gilt" lokales a -> 5
  std::cout << ::a << std::endl; // Zugriff auf globales a -> 20
}
```

Namensräume (1)

- Bekanntes Beispiel: `std` (Abkürzung für "standard").
- Mit Namespaces kann man Gültigkeitsbereiche definieren und benennen.
- Innerhalb solcher Gültigkeitsbereiche können beliebige Bezeichner wie Klassen, Funktionen, Variablen, Typen oder auch innere Namespaces definiert werden.
- Insbesondere für große Projekte ist es nützlich, Klassen und Funktionen in Sinneinheiten zu sortieren.
- Zugriff auf einen Namespace erfolgt mit dem Scope-Operator `::`
- Beispiel:

```
std::cout
```

- Im Rahmen eigener Funktionen werden wir uns die Erstellung eigener Namespaces ansehen.

Namensräume (2)

Namespaces importieren:

- Mittels `using` kann ein Namespace, bzw. Teile davon importiert werden. Danach wird der Scope Operator nicht mehr benötigt.
- Syntax für einzelne Komponenten:

```
using std::cout;
```

- Daraufhin vereinfacht sich ein Ausgabebefehl zu

```
cout << "Hallo." << std::endl;
```

- Den ganzen Namespace importiert man über:

```
using namespace std;
```

- Daraufhin vereinfacht sich der Ausgabebefehl weiter zu

```
cout << "Hallo." << endl;
```

Vorsicht!

Das Importieren eines kompletten `namespace` erzeugt oft Namenskonflikte mit eigenen Funktionen und ist darum im Allgemeinen nicht zu empfehlen!

Warnungen und Fehler (1)

Wir erinnern uns an die Beispiele von letzter Woche aus der ersten Vorlesung:

- Bei vielen sind Kompilierfehler oder sogenannte Warnings aufgetreten.

$\Rightarrow$ Beim Programmieren macht man zwangsläufig irgendwann (Programmier-) Fehler.

$\rightarrow$ Insbesondere die syntaktischen Fehler werden vom Compiler erkannt. Es hilft nicht die Augen zuzudrücken, man muss sich diesen Fehlern und Compiler-Warnungen frühestmöglich stellen.

- Es folgen einige Beispiele auf den folgenden Seiten.

Warnungen und Fehler (2)

Beispiel 7. Wir betrachten den folgenden Beispielcode (`bsp_1.cc`):

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

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

Wenn wir dieses Beispiel mit

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

oder mit

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

(mit Warnings) kompilieren, sehen wir?

$\Rightarrow$ Hoffentlich nichts.

Warnungen und Fehler (3)

Beispiel 8. Erweiterung unseres Beispiels:

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

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

Aufgabe

Was seht ihr in der Konsole, wenn ihr mit oder ohne Warnings kompiliert?

Warnungen und Fehler (4)

Mit Warnings ergibt sich

```
bsp_2.cc: In Funktion »int main()«:
bsp_2.cc:6:7 Warnung: Variable »a« wird nicht
                        verwendet [-Wunused-variable]
  int a;
      ^
```

- Was bedeutet das?
- Wir erhalten eine Warnung. Das Programm kompiliert, teilt uns aber mit, dass wir unsauber programmiert haben.
- In der Funktion `main` hat der Compiler eine Schwachstelle entdeckt.
- Und zwar in Zeile 6, Spalte 7 der Datei `bsp_2.cc`. Es existiert eine Variable (hier a), die wir zwar deklariert haben, aber (bisher) nicht nutzen. Daher ist diese Codezeile überflüssig und könnte weggelassen werden.
- Wenn wir vorhaben die Variable zu nutzen und nur testweise den bisherigen Teil kompiliert haben, können wir diese Warnung natürlich ignorieren.

Warnungen und Fehler (5)

Beispiel 9. Wir wandeln unser Beispiel so ab, dass es einen häufigen Fehler produziert:

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

Wir sehen in der Konsole

```
bsp_3.cc: In Funktion »int main()«:
bsp_3.cc:5:8: Fehler »cout« ist kein Element von »std«
  std::cout << "Hallo. Und wie geht's?" << std::endl;

bsp_3.cc:5:8 Anmerkung: »std::cout« ist im Header »<iostream>«
definiert; haben Sie vergessen, »#include <iostream>«
zu schreiben?
bsp_3.cc:1:1:
+#include <iostream>
...
```

Hier liegen zwei Fehler vor und das Programm wird nicht kompiliert!
Und zwar kann der Compiler die Funktionen `cout` und `endl` nicht in `std` zuordnen.
Wir haben vergessen, die entsprechende Bibliothek via `#include` einzubinden.
Der relativ neue Compiler (GCC 8.1.0) sagt uns sogar, was wir gemeint haben könnten.

Kontrollstrukturen

In C++ existieren verschiedene Kontrollstrukturen, um den Programmablauf zu steuern.
Wir werden uns im Folgenden die drei wichtigsten anschauen:

- Fallunterscheidungen `if`, `else` prüfen, ob eine Bedingung erfüllt ist und führen entsprechend Code aus. Oder auch nicht.
- `for`-/Zählschleifen iterieren üblicherweise eine Variable über einen festgelegten Wertebereich und führen darauf basierenden Code aus.
- `while`-Schleifen führen einen Codeblock so lange aus, wie eine Bedingung erfüllt ist.

Fallunterscheidungen (1)

Beispiel 10. Dieses Programm führt abhängig von der Eingabe verschiedene Befehle aus:

```
#include <iostream>

int main()
{
  int a,b;
  std::cout << "Eingabe: a b für a/b: ";
  std::cin >> a >> b;
  if (b == 0) // Fallunterscheidung, prüfe Bedingung
  {
    std::cout << "Fehler: b = 0.\n";
    return 1; // Vorzeitiger Rücksprung (mit Fehlercode).
  }
  else // b != 0
  {
    std::cout << "a/b = " << a/b << ".\n";
  }
}
```

Fallunterscheidungen (2)

Beispiel 11. Variante des obigen Beispiels (ohne Fehlerrückgabewert 1):

```
#include <iostream>

int main()
{
 int a, b;
 std::cout << "Eingabe: a b für a/b: ";
 std::cin  >> a >> b;
 if (b == 0) // Block mit nur einem Befehl: {} unnötig
   std::cout << "Fehler: b = 0.\n";
 else
   std::cout << "a/b = " << a/b << ".\n";
}
```

Schleifen: for

Beispiel 12. Mehrmalige Ausführung eines Befehlsblocks mit verschiedenen Daten:

```
#include <iostream>
int main()
{
  // Einfachste for-Schleife (engl. for-loop)
  for (unsigned int k = 0; k <= 5; ++k) {
    std::cout << k << '\n';
  }
}
```

Bemerkung (Ausführungsblöcke)

Grundsätzlich müssen in Kontrollstrukturen Blöcke verwendet werden (geschweifte Klammern), außer wenn der Anweisungsblock lediglich eine Anweisung umfasst.
Achtung: Die lokale Variable k wird nach der for-Schleife wieder ungültig.

```
#include <iostream>
int main()
{
  // Kurze Variante (nur eine Anweisung!) ohne geschweifte Klammern
  for (unsigned int k = 0; k <= 5; ++k)
    std::cout << k << '\n';
}
```

Schleifen: for

Beispiel 13. Mehrmalige Ausführung eines Befehlsblocks mit verschiedenen Daten:

```
#include <iostream>
int main()
{
  unsigned int n; // Nichtnegative ganzzahlige Variable n.
  std::cout << "Eingabe von n: ";
  std::cin >> n;
  std::cout << "Tabelle mit k, k^2, k^3 für k <= " << n << ".\n";
  for (unsigned int k = 0; k <= n; ++k)
  {
    unsigned int k2 = k*k; // Lokale Variablen.
    unsigned int k3 = k2*k;
    std::cout << k << ' ' << k2 << ' ' << k3 << '\n';
  }
  // Erinnerung: Hier sind k2 und k3 nicht länger definiert.
}
```

Schleifen: while

Grundgerüst: Ausführung der Anweisungen so lange, bis die while-Bedingung falsch ist. Danach wird das Programm hinter dem Block fortgesetzt.

Beispiel 14. Ein einfaches Beispiel einer while-Schleife:

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

Bemerkung

Ein häufiger Fehler bei while-Schleifen ist das Weglassen des Updates der Abbruchbedingung (hier `++var`), so dass es dann zu einer Endlosschleife kommt, die man nur noch „gewaltsam“ abbrechen kann. Nur in Ausnahmefällen sind Endlosschleifen erwünscht, wenn man z.B. auf ein bestimmtes Ereignis wartet oder eine Anweisung dauerhaft überwachen will.

Zwangsweiser Abbruch eines Programms

- Falls Programm in Endlosschleife (z.B. update bei while-Schleife vergessen: probieren Sie es bitte selbst aus!)
- oder Kompilierung zu lange dauert, aufgrund eines Fehlers
- oder man gerade kompiliert hat, aber etwas im Programm vergessen hatte:
- dann ist der Notschalter in der Konsole: `Ctrl+C`, also die Taste „Steuerung“ und die Taste „C“ gleichzeitig drücken.

Ende VL 2

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 2 durch.

Grundlagen

- Wir unterscheiden Operatoren anhand der Anzahl ihrer Operanden:
  - Unärer Operator (z.B. Vorzeichen bei Variablen, oder Inkrementierungen)
  - Binärer Operator (am häufigsten anzutreffen)
  - Ternärer Operator (in C++ genau einer; oft schwer lesbar)
- Neben der Anzahl wird auch die Position unterschieden:
  - Infix: Der Operator steht zwischen den Operanden (binär)
  - Präfix: Der Operator steht vor dem Operanden (unär)
  - Postfix: Der Operator steht hinter dem Operanden (unär)
- Assoziativität: Links- und Rechtsassoziativität
- Die 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 Ganzzahldivision
```

- Es gelten die üblichen mathematischen Regeln (Punkt-vor-Strich)
- Arithmetische Operatoren sind binäre Operatoren
- Die Anwendung eines Operators auf zwei Variablen und das nachfolgende Speichern in der ersten Variablen wird so häufig benötigt, dass es dafür eine Abkürzung gibt:

```
var1 += var2 // entspricht var1 = var1 + var2
             // gilt ebenso für die vier weiteren arithm. Operatoren
```

Inkrement- und Dekrementoperator

Eine ebenso wichtige Operation ist die Addition bzw. Subtraktion von 1:

```
++i // i = i + 1; i += 1; (Inkrement: Variable um 1 erhöhen)
--i // i = i - 1; i -= 1; (Dekrement: Variable um 1 verringern)
```

- Es existieren zwei Varianten, Prä- oder Postfix, je nach Anordnung von Operator und Operand:

```
++i // Präinkrement
i++ // Postinkrement
```

- Verwendung in for-Schleifen (häufigste Anwendung) äquivalent, besser ist `++i`.
- Bei Zuweisung auf eine andere Variable etwas subtil:

Aufgabe

Testet selbst: welchen Wert hat d in

```
c = 41;
d = ++c; oder
d = c++; ?
```

Das Postinkrement sollte man nur in Fällen wie `d = c++` verwenden: wenn der „alte“ Wert benutzt wird!

Logische Operatoren und Vergleiche

```
!  // Logische Verneinung (NOT)
&& // Logisches Und (AND)
|| // Logisches Oder (OR)

<  // Kleiner
>  // Größer
== // Gleich/äquivalent
!= // Ungleich
<= // Kleinergleich
>= // Größergleich
```

Bemerkung (Ternärer Operator)

Syntax:
`(condition) ? (term_true) : (term_false);`
Kurzschreibweise von

```
if (condition)
  term_true;
else
  term_false;
```

Aber man kann das Ergebnis zuweisen: `int max = a > b ? a : b;`

Grundlagen

- Funktionen sind Unterprogramme, mit denen Daten verarbeitet werden oder Teilprobleme gelöst werden.
- Funktionen dienen insbesondere zur Strukturierung eines Codes und ermöglichen die Mehrfachnutzung häufig verwendeter Codebestandteile.
- Erste und wichtigste Funktion ist die main-Funktion.
- In C++ gibt es keinen standardisierten Weg, Funktionen parallel auszuführen, so dass hier immer auf externe Bibliotheken zurückgegriffen werden muss.
- Funktionen werden in der Regel erst deklariert und dann definiert.
- Syntax:

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

- Rückgabetyp: Datentyp: int, double etc.
  Falls die Funktion nichts zurückgeben soll, dann wird `void` verwendet.
- Funktionsname: eindeutiger Name (im jeweiligen Gültigkeitsbereich).
- Parameter: optional, weitere Informationen zum Ausführen der Funktion.
  Mehrere Parameter werden durch Kommata getrennt.

Mathematische Funktionen (1)

Beispiel 15. Viele mathematische Funktionen sind in der Standardbibliothek verfügbar (zu finden in cmath):

```
#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);       // auch cos, tan
  double asin_y  = std::asin(sin_x); // = arcsin; auch acos, atan
  double sinh_x  = std::sinh(x);      // auch cosh, tanh, asinh, ...
  double exp_x   = std::exp(x);
  double log_y   = std::log(exp_x);
  double exp10_x = std::pow(10.0, x);
  double log10_y = std::log10(exp10_x);
  double abs_x   = std::abs(x);       // |x|
  double sqrt_x  = std::sqrt(abs_x);
  ... // Fortsetzung nächste Seite
```

Mathematische Funktionen (2)

Beispiel 15. (Forts.)

```
...
std::cout.precision(16); // legt Ausgabegenauigkeit fest
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|\t= " << abs_x
  << "\n|x|^1/2 = " << sqrt_x
  // Zwei weitere Standardfunktionen:
  << "\nfloor(x) = " << std::floor(x)
  << ",\tceil(x) = " << std::ceil(x) << '\n';
 }
```

Eigene Funktionen (1)

Beispiel 16. 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); // Ab C++11 auch std::exp2(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 für k <= " << n << ".\n";
  for (unsigned int k = 0; k <= n; ++k)
    std::cout << k << ' ' << square(k) << ' ' << pow2(k) << '\n';
}
```

Eigene Funktionen (2)

Beispiel 17. Wurzelberechnung mit dem Newton-Verfahren:

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

// Eps hat Default-Wert.
double my_sqrt(double x, double eps = 1.0e-10)
{
  x = std::abs(x); // Setze x = |x|.
  double s = x / 2; // Startwert 'raten'.
  double e = 16 * std::numeric_limits<double>::epsilon();
  e = std::max(e, eps); // e größer als Maschinengenauigkeit.

  while (std::abs(s * s - x) > e)
    s = (x / s + s) / 2; // s_neu = s - f(s)/f'(s)

  return s; // Ergebnis der Newton-Iteration für 0 = f(s) = x - s^2.
}
// Fortsetzung nächste Seite ...
```

Eigene Funktionen (3)

```
// Fortsetzung von vorheriger Seite
int main()
{
 double x, eps;
 std::cout << "Eingabe: x [eps] ";
 std::cin >> x;

 if (std::cin.fail()) // Keine Zahl eingegeben: Abbruch.
  return 1;

 std::cin >> eps;

 // Rufe my_sqrt() mit oder ohne eps auf, je nach Eingabe.
 // (Hier ist der ternäre Operator sinnvoll eingesetzt.)
 double s = std::cin.good() ? my_sqrt(x, eps) : my_sqrt(x);
 std::cout << "s = " << s << ", x - s^2 = " << (x - s * s) << '\n';
}
```

Eigene Funktionen (4)

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, wofür die Parameter später verwendet werden sollen.

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

Bemerkung

Zum Beispiel sind `int a, b, c;` sehr beliebt, aber i.A. keine guten Bezeichner. Besser sind z.B. `int length_x, length_y, length_z;` um beispielsweise das Volumen eines Quaders zu berechnen.

Eigene Funktionen (5): Aufruf und Parameterübergabe

- Aufruf einer Funktion ohne Parameter:

```
my_function();
```

Beispiel 18.

```
#include <iostream>

// Funktionsprototyp (Deklaration)
void my_function();
// Ab hier darf die Funktion benutzt werden

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

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

Eigene Funktionen (6): Aufruf und Parameterübergabe

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

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

Aufgabe

Programmiert eine Funktion, die das aktuelle Jahr, das übergebene Jahr (Parameter) und deren Differenz ausgibt.
Testet außerdem, was passiert, wenn keine Ganzzahl, sondern eine Gleitkommazahl z.B. 2020.5 übergeben wird.

Bemerkung

Neben call-by-value gibt es noch den Aufruf call-by-reference, bei dem anstelle des Wertes einer Variable deren Adresse übergeben wird. Beide Aufrufe können verschiedene Effekte erzeugen! Später mehr.

Eigene Funktionen (7): Rückgabewerte

- Funktionen können weitergehend eingesetzt werden, wenn diese ein Ergebnis zurückgeben.
- Die Rückgabe geschieht mittels `return`.
- Das bekannteste Beispiel ist:

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

  Gleichzeitig kann das `return 0;` in der main-Methode aber weggelassen werden, da der Compiler das automatisch ans Ende anfügen würde.

- Soll eine Funktion bewusst keinen Rückgabewert haben, so wird ihr Rückgabetyp auf `void` (engl.: nichtig, leer) gesetzt.

Aufgabe

Programmiert eine Funktion, die die Maße eines Raums übergeben bekommt, das Volumen berechnet und dieses dann mittels `return` zurückgibt.

Eigene Funktionen (8): Rekursion

- Funktionen dürfen sich auch selbst aufrufen (außer `main`).
- Diesen Vorgang nennt man Rekursion.
- Dieses Konzept ist uns von Folgen bekannt, die oft mit einer Rekursionsvorschrift angegeben werden.
- Ein bekanntes Beispiel ist die Fakultät:

```
unsigned int factorial(unsigned int n)
{
  if (n < 2)
    return 1;
  return factorial(n-1) * n;
}
```

- Da der Aufruf einer Funktion aber Zeit kostet, sind einfache Schleifen fast immer besser!

Eigene Funktionen (9): Lokale und globale Variablen

- Wir haben bereits gesehen, dass Funktionen Parameter übergeben bekommen können und auch Werte zurückgeben können.
- Innerhalb einer Funktion können aber auch neue Variable definiert werden (siehe Einführung/Blöcke).
- Im Gegensatz dazu können außerhalb von Funktionen globale Variablen definiert werden, die überall gültig sind.
- 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?

Aufgabe

Testet dies bitte selbst!

Ende VL 3

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 3 durch.

Eigene 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, 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.

Eigene 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 par1 = 5, int par2 = 78, int par3 = 3)
```

- Der umgekehrte Fall ist aber möglich:

```
void func(int par1, int par2, int par3 = 3)
```

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

```
func(23, 45);     // par3 wird auf 3 gesetzt
func(23, 45, 46); // par3 wird auf 46 gesetzt
```

Eigene Funktionen (12): Überladen

- In C++ können Funktionen überladen werden, d.h., 'verschiedene' Funktionen haben denselben Namen.
- Die Funktionen müssen sich aber hinsichtlich Parametertypen und/oder Anzahl der Parameter unterscheiden.
- Beispiel

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

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

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

- Warum funktioniert das?

⇒ C++ identifiziert Funktionen nicht nur anhand des Namens, sondern anhand ihrer Signatur. Die Signatur besteht aus der Kombination von Namen und Parameterliste.

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

Eigene Funktionen (13): Überladen – Beispiele

Beispiele:

- Mathematische Funktionen wie

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

- Selbstdefinierte Funktionen:

```
float norm_2(float a, float b) // Ab C++11 auch std::hypot(a, b)
{
  return std::sqrt(a*a+b*b);
}
double norm_2(double a, double b) // Dito
{
  return std::sqrt(a*a+b*b);
}
double norm_2(double a, double b, double c)
// Ab C++17 auch std::hypot(a, b, c)
{
  return std::sqrt(a*a+b*b+c*c);
}
```

Eigene Funktionen (14): Überladen Beispiele

Aber:

Aufgabe

Implementiert und testet das folgende Beispiel. Was gibt der Compiler zurück?

Beispiel 19. Überladen mit gleichen Parametern:

```
float norm_2(float a, float b)
{
  return std::sqrt(a*a+b*b);
}
double norm_2(float a, float b)
{
  return std::sqrt(a*a+b*b);
}
```

Frage

Ändert eine Umbenennung der Parameter etwas an dieser Situation?

Funktions-Templates (1)

Beim Überladen haben wir gesehen, dass wir gewisse Codeabschnitte immer wieder abgetippt haben.

- Erstens ist es aufwändig, denselben Code für alle möglichen Datentypen zu schreiben.
- Zweitens unterlaufen einem beim Kopieren zwangsläufig irgendwann (mal wieder Programmier-) Fehler.
- Drittens ist die Wartung auch nicht gerade einfach, da wir denselben Code an mehreren Stellen im Code gleich halten müssen.
- Ansonsten würde unsere Funktion in irgendeiner ihrer Ausführungen etwas Unerwartetes tun.

Funktions-Templates (2)

- Einfacher geht das Ganze mit generischer Programmierung. In C++ funktioniert das mit `template`-Parametern.
- Damit erhält der Compiler eine Blaupause, anhand derer er den benötigten Code selbständig erzeugen kann.

```
template<typename T>
T norm_2(T a, T b)
{
  return std::sqrt(a*a + b*b);
}
```

Funktions-Templates (3): Instanziieren

Bei jeder Benutzung dieser Funktion mit einer neuen Kombination von Datentypen erzeugt der Kompiler automatisch den benötigten Code für die konkreten Datentypen. Dieser Schritt wird Template-Instanziierung genannt und muss eindeutig sein.

Uneindeutigkeiten können auf zwei Arten vermieden werden:

- Explizite Typenkonvertierungen
- Explizite Spezifizierung des Template-Parameters in spitzen Klammern

```
std::cout << norm_2<int>(4, 5) << std::endl;
```

Beispiel 20. Instanziierungen unserer norm_2 Funktion

```
int main()
{
  cout << norm_2(3, 4)           << '\n'; // Eindeutig, T=int
  cout << norm_2(3.14, 7.)       << '\n'; // Eindeutig, T=double
  cout << norm_2(6.1, 4)         << '\n'; // Uneindeutig: Fehler!
  cout << norm_2<double>(6.1, 4) << '\n'; // Eindeutig, T=double
  cout << norm_2(6.1, double(4)) << '\n'; // Eindeutig, T=double
  cout << norm_2<int>(6.1, 4)    << '\n'; // Warnung: 6.1 -> 6
}
```

Eigene Namensräume (1)

Beispiel 21. Definition eines eigenen namespace mit Funktionen:

```
namespace mein_Bereich
{
  int   i_wert;
  float f_wert;
  void  funktion() {
    // Anweisungen in der Funktion
  }
  void  funktion_2(unsigned int param = 42) {
    // Anweisungen in der 2. Funktion
  }
}
```

Erinnerung:

- Außerhalb des namespace (z.B. in main) Aufruf über Scope-Operator

```
int main()
  mein_Bereich::funktion_2(15);
```

- Innerhalb des namespace (z.B. in funktion_2) Aufruf auch ohne Scope-Operator möglich

```
void funktion_2(unsigned int param = 42)
  funktion();
```

Eigene Namensräume (2)

Guter Stil: Deklaration innerhalb des Namespaces; die Definition aber außerhalb.
Beispiel 22. Getrennte Deklaration und Definition bei eigenen Namespaces:

```
namespace mein_Bereich
{
  int   i_wert;
  float f_wert;
  // Deklarationen
  void funktion();
  void funktion_2(unsigned int param = 42);
}
// Definitionen
void mein_Bereich::funktion()
{
  // Anweisungen
}
void mein_Bereich::funktion_2(unsigned int param)
{
  // Anweisungen
}
```

Dies ist insbesondere bei Klassen üblich und wir werden uns diesem Thema im Kapitel Programmstrukturierung noch einmal widmen.

Motivation (1)

- Speicherplatz auf dem Computer ist – wie auch auf Taschenrechnern – beschränkt.
- Daher treten Rundungsfehler bereits bei der bloßen Darstellung einer Zahl auf.
- Zweiter Grund: der Computer kann nicht mit beliebig langen Zahlen rechnen, dabei treten also auch Rundungsfehler auf.
- Grund ist die beschränkte Datenbandbreite (z.B. 8 Bit, 16 Bit, 32 Bit, 64 Bit) der Prozessoren.
- Computer speichern Zahlen gerundet in Binärdarstellung, also zur Basis 2:

$$rd(x) = \pm \sum_{i=-n_1}^{n_2} a_i 2^i, \quad a_i \in \{0, 1\}$$

- Das Binärsystem ist auch als Dualsystem bekannt.
- Beispiel (mit $n_1 = 0$ und $n_2 = 3$):

$$0 = [0]_{10} = [0000]_2 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$
$$1 = [1]_{10} = [0001]_2 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$12 = [12]_{10} = [1100]_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

- Zu jeder Basis $b \in \mathbb{N}_{>0}$ gibt es eine entsprechende b-adische Darstellung.

Motivation (2)

- Der Namenspatron unserer Universität (Gottfried Wilhelm Leibniz) war einer der Erfinder des Dualsystems.

- Wir haben:

$$1 = [1]_{10} = [1]_2 = 1 \cdot 2^0,$$
$$2 = [2]_{10} = [10]_2 = 1 \cdot 2^1 + 0 \cdot 2^0,$$
$$4 = [4]_{10} = [100]_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^1.$$

Ganzzahlige Datentypen (1)

Typ	Bytes	Wertebereich
int	4	$\{-2147483648 \ldots 2147483647\}$
short int	2	$\{-32768 \ldots 32767\}$
long int	4 oder 8	Wie int oder long long int
long long int	8	$\{-9223372036854775808 \ldots 9223372036854775807\}$
bool	1	false, true (entspricht 0, 1)

- Erinnerung: 1 Byte = 8 Bits mit $2^8 = 256$ Ausprägungen.
- Also z.B. `short int`: 2 Byte = 16 Bits mit $2^{16} = 65536$ Ausprägungen.
- Die Überschreitung der maximalen Werte haben Sie bereits in der Übungsaufgabe zur Fakultät gesehen.

Bemerkung

- Bei den Typen `short int`, `long int`, `unsigned int` usw. ist `int` optional: Sie können auch `long` schreiben anstatt `long int`. Testen Sie selbst!
- unsigned Typen benutzen Modulo-Arithmetik und haben keinen overflow.
- Das Verhalten der signed-Typen bei overflow ist undefiniert.
- Die Speichergrößen sind typisch, aber nicht vorgeschrieben.

Ganzzahlige Datentypen (2): Binärdarstellungen

- Eine n-bit-Zahl kann wie folgt in Binärform dargestellt werden:

	Ohne Vorzeichen	Mit Vorzeichen
Bitmuster	$[b_{n-1} \dots b_1 b_0]$	$[s \vert b_{n-2} \dots b_0]$
Wert	$\sum_{i=0}^{n-1} b_i 2^i$	$s = 0 : \sum_{i=0}^{n-2} b_i 2^i$
		$s = 1 : \sum_{i=0}^{n-2} b_i 2^i - 2^{n-1}$
Wertebereich	$\{0, 1, \dots, 2^n - 1\}$	$\{-2^{n-1}, \dots, -1, 0, 1, \dots, 2^{n-1} - 1\}$

- Wie wir also sehen, haben die Zahlen im Intervall $[0, 2^{n-1} - 1]$ die selbe Binärdarstellung.
- Das ist bewusst so gewählt um die Umwandlung positiver `int` in `unsigned int` zu vereinfachen.
- Ab C++20 müssen auch $[-2^{n-1}, \dots, -1]$ (signed) und $[2^{n-1}, \dots, 2^n - 1]$ identisch sein.

Gleitkomma-Datentypen (1)

Typ	Bytes	Min	Max	Eps
float	4	1.175494e−38	3.402823e38	1.192093e−7
double	8	2.225074e−308	1.797693e308	2.220446e−16
long double	10	3.362103e−4932	1.189731e4932	1.084202e−19

- Siehe auch https://en.wikipedia.org/wiki/Floating-point_arithmetic
- Min und Max: betragsmäßig kleinste und größte Maschinenzahlen.
- Maschinengenauigkeit Eps ist die relative Genauigkeit. Für $x \in \mathbb{R} \setminus \{0\}$ und für die zugehörige Maschinendarstellung $\bar{x}$ gilt

$$\left| \frac{x - \bar{x}}{x} \right| \leq Eps.$$

- Falls $|x|$ zu groß ist, dann haben wir overflow, d.h., $\bar{x} = \pm\infty$.

Bemerkung

Die Kenntnis der Maschinengenauigkeit ist essentiell bei der Entwicklung numerischer Algorithmen, um Rundungsfehler zu analysieren und z.B. sinnvolle Abbruchgenauigkeiten für iterative Verfahren (Newton, Fixpunkt, ...) festzulegen. Hier muss entschieden werden, wann wir hinreichend nah bei der Null sind. Für hohe Genauigkeiten sollte auf jeden Fall mit `double` gearbeitet werden!

Gleitkomma-Datentypen (2)

IEEE 754: Normalisierte Gleitkommadarstellung

- n-bit-Zahl mit k bit Exponent und $1 + p$ bit Mantisse:
  ein "hidden bit" $m_0 = 1$ und p gespeicherte bits m_i
- Bitmuster: $[s|e_{k-1} \dots e_0|m_1 \dots m_p]$
- Exponent: $E = \sum_{i=0}^{k-1} e_i 2^i \in \{0, 1, \dots, E_{max} - 1, E_{max}\}, \quad E_{max} = 2^k - 1$
- Mantisse: $m = 1 + \sum_{i=1}^{p} m_i 2^{-i} \in \{1, 1 + 2^{-p}, \dots, 2 - 2^{-p}\}$
- Wert: $(-1)^s \cdot m \cdot 2^e, \quad e = E - B, \quad B = \text{Bias},$

$$rd(x) = (-1)^s \cdot [1.m_1 m_2 \dots m_p]_2 \cdot 2^{[e_{k-1} \dots e_1 e_0]_2 - B}$$

Gleitkomma-Datentypen (3)

- Sonderfälle: $E = (0 \ldots 0) = 0$: 0 und denormalisierte Zahlen.
- und $E = (1 \ldots 1) = E_{max}$: $\pm\infty$ und NaN (not a number, z.B. $0/0$).
- $Eps = 2^{-p}$, $Min = 2^{1-B}$, $Max = (2 - 2^{-p}) \cdot 2^{E_{max}-1-B} \approx 2^{E_{max}-B}$.

Typ	Bytes	$k+p$	Min	Max	Eps	E_{max}	B
float	4	23+8	2^{-126}	$(2-2^{-23})2^{127}$	2^{-23}	2^8-1	2^7-1
double	8	52+11	2^{-1022}	$(2-2^{-52})2^{1023}$	2^{-52}	$2^{11}-1$	$2^{10}-1$
long double	10	63+15	2^{-16382}	$(2-2^{-63})2^{16383}$	2^{-63}	$2^{15}-1$	$2^{14}-1$

Vorsicht!

Wenn wir zwei Gleitkommazahlen mit unterschiedlichen Exponenten addieren wollen, stellt sich das Problem, dass wir nicht einfach die Mantissen addieren können. Die Addition funktionert stattdessen so, dass die Zahl mit dem kleineren Exponenten auf den größeren Exponenten gerundet wird.

Frage

Erklären die Additionsregeln, weshalb die Berechnung der Eulerschen Konstante (Blatt 3) absteigend (also in aufsteigender Größe der Summanden) die geringere Abweichung hat?

Gleitkomma-Datentypen (4)

Bemerkungen zu IEEE (Institute of Electrical and Electronics Engineers) 754:

- Mantisse und Vorzeichen: Die Mantisse schreibt sich

$$m_0.m_1 \ldots m_p$$

  mit $m_0 = 1$ vor dem Komma.
- Für 32 und 64 bit ist die IEEE-Gleitkommadarstellung in allen Details exakt so vorgeschrieben. Also insbesondere $\pm\infty$, NaN und denormalisierte Zahlen.
- Für andere Datentypen, z.B. long double, gibt es verschiedene erlaubte Darstellungen, abhängig vom Prozessor.

Gleitkomma-Datentypen (5)

Definition (Normalisierte Gleitkommazahlen: IEEE 754-Format)

Das IEEE 754-Format beschreibt die *Gleitkommadarstellung* einer Zahl im *Binärformat*

$$x = s \cdot M \cdot 2^{e-b}$$

mit einem Vorzeichen $s \in \{+, -\}$, einer Mantisse $M \in [1, 2)$ sowie einem Exponenten $e \in \mathbb{N}$ mit Bias $b \in \mathbb{N}$. Die Gleitkommazahl wird in Binärdarstellung gespeichert:

$$s e_{\#e} \dots e_2 e_1 m_{\#m} \dots m_2 m_1,$$

mit $\#e$ Bit für den Exponenten und $\#m$ Bit für die Mantisse. Der Bias wird im Allgemeinen als $2^{\#e-1} - 1$ gewählt. Die Interpretation der Zahlen hängt vom Exponenten ab:

Normalisierte Zahl Im Fall $0 < e < 2^{\#e-1}$ ist

$$x = s \cdot 1.m_{\#m} \dots m_2 m_1 \cdot 2^{e-b}.$$

Null Im Fall $e = 0$ und $m_1 = \dots = m_{\#m} = 0$ ist $x = \pm 0$. Die Unterscheidung zwischen $+0$ und -0 entsteht beim Runden kleiner Zahlen zur Null. Im direkten Vergleich interpretiert ein Computer $+0 = -0$.

Denormalisierte Zahl Im Fall $e = 0$ und $m_i = 1$ für ein i ist $x = s \cdot 0.m_{\#m} \dots m_2 m_1 \cdot 2^{1-b}$.

Unendlich Im Fall $e = 2^{\#e} - 1$ (alle Exponenten-Bits gleich 1) und $m_1 = \dots = m_{\#m} = 0$ ist $x = \pm\infty$. Unendlich entsteht etwa beim Teilen durch 0 (mit Vorzeichen!) oder falls das Ergebnis zu groß (oder negativ zu klein) ist, um darstellbar zu sein.

NaN Im Fall $e = 2^{\#e} - 1$ und $m_i = 1$ für ein i steht der Wert für *not a number* und tritt zum Beispiel bei der Operation $0/0$ oder $\infty - \infty$ auf.

Gleitkomma-Datentypen (6)

Beispiele (aus T. Richter, T. Wick: *Einführung in die numerische Mathematik – Begriffe, Konzepte und zahlreiche Anwendungsbeispiele;* Springer, 2017): Wir gehen von vierstelliger Mantisse und vier Stellen im Exponenten aus mit Bias $2^{4-1} - 1 = 7$.

- Die Zahl $x = -96$ hat zunächst negatives Vorzeichen, also $S = 1$. Die Binärdarstellung von 96 ist

$$96_{10} = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = 1100000_2,$$

  normalisiert

$$96_{10} = 1.1000_2 \cdot 2^{6_{10}} = 1.1_2 \cdot 2^{13_{10}-7_{10}} = 1.1_2 \cdot 2^{1101_2-b}.$$

  Als Gleitkommadarstellung ergibt sich 11011000_2.
- Die Zahl $x = -384$ hat wieder negatives Vorzeichen und $S = 1$. Die Binärdarstellung von 384 ist:

$$\begin{aligned} 384_{10} &= 1 \cdot 256 + 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 \\ &= 110000000_2, \end{aligned}$$

  also normalisiert

$$384_{10} = 1.1000 \cdot 2^{8_{10}} = 1.1000 \cdot 2^{15_{10}-7_{10}} = 1.1000_2 \cdot 2^{1111_2-b}.$$

  Alle Bits im Exponenten sind 1. Der spezielle Wert $e = 1111_2$ ist jedoch zur Speicherung von NaN (not a number) vorgesehen. Die Zahl 384 ist zu groß, um in diesem Zahlenformat gespeichert zu werden. Stattdessen wird $-\infty$ oder binär 111110000_2 gespeichert.

Beispiele

- Vorsicht bei Rechnungen mit ganzen Zahlen und Gleitkommazahlen!
- Beispiel 23. int und double

```
double a = 5/2;
std::cout << a << std::endl;
```

liefert den falschen Wert 2.

- Warum? Sie geben zunächst zwei int ein und dividieren diese dann. Heraus kommt wieder ein int, der Divisionsrest wird also abgeschnitten, mit Ergebnis 2. Die anschließende Konvertierung auf double hilft dann nicht mehr, das eigentlich richtige Ergebnis zu erhalten.

Beispiele

- Beachte nun folgende Modifikation
- Beispiel 24. int und double

```
double a = 5.0/2;
```

liefert nun 2.5.

- Hier erhalten wir das richtige Ergebnis, da das Literal 5.0 den Typ double hat.
- Auch folgender Programmcode tut das Richtige:

```
double b = 5;
double c = 2;
double a = b/c;
```

- Hier werden die integer-Werte zunächst in double konvertiert, bevor wir die Division durchführen.

Ende VL 4

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 4 durch.

Zeiger (1): Adresse einer Variablen

- Schweres Thema! Trotzdem im Prinzip einfach vom logischen Standpunkt her!
- Wir fragen uns zunächst, wofür wir wirklich Zeiger benötigen, da wir mit den vorangegangenen Kapiteln bereits viele Problemstellungen berechnen (programmieren) können.
- Was sind also Zeiger (engl. pointer)?
- Zeiger greifen direkt auf die Adresse einer Variablen zu
- Zugriff mit Adressoperator &
- Beispiel 25. Zugriff auf Adresse einer Variablen:

```
#include <iostream>

int main()
{
  int var = 101;
  std::cout << "Wert von var:    " << var << std::endl;
  std::cout << "Adresse von var: " << &var << std::endl;
}
```

- Adressen unterscheiden sich abhängig vom Betriebssystem und Rechnertyp!

Zeiger (2): Deklaration / Syntax

- Was können wir nun mit der Information der Adresse tun?
- Zunächst Syntax eines Zeigers:

```
Typ *zeiger;
```

- Zeiger sind typgebunden: Zeigertyp und Datentyp des Speicherplatzes müssen identisch sein.
- Beispiel:

```
int *i_wert;
double *d_wert;
```

- Zwei Varianten der Deklaration: Sternchen bei Typ oder Sternchen bei Bezeichner:

```
int *i_wert;  // Alter C-style
int* i_wert;  // Neuer C++-style
```

- Vorsicht:

```
int* i_wert_1, i_wert_2; // Nur ein einziger Zeiger deklariert!
int *i_wert_1, *i_wert_2; // Zwei Zeiger deklariert
```

- Alter C-style etwas übersichtlicher, wenn mehrere Zeiger und ggf. Variablen deklariert werden! Guter Stil: nur eine Deklaration pro Zeile, * bei Typ.

Zeiger (3): Zusammenhang mit Adressen

- Wie bringen wir nun Adressen und Zeiger zusammen?
- Zeiger ist eine Variable, die nicht den Wert selbst speichert, sondern die Adresse im Arbeitsspeicher aufnimmt.
- Zu diesem Zweck benötigen wir den Adressoperator `&`.
- Beispiel 26. Anlegen eines Zeigers

```
int var = 101;
int *pointer_var;
pointer_var = &var;
```

- Hierbei wird die Adresse von `var` dem Pointer `pointer_var` zugewiesen.
- Test/Probe: wir geben nun die Adressen aus
- Info: Hierbei benötigen wir aber nicht `&`, da der Pointer bereits die Adresse speichert.

Zeiger (4): Beispiel

Beispiel 27. Adressen von Zeigern und Daten:

```
#include <iostream>
int main()
{
  int var = 101;
  int *pointer_var;
  pointer_var = &var;
  std::cout << "Adresse von var: " << &var;
  std::cout << "\npointer_var verweist auf Adresse: " << pointer_var;
  std::cout << "\nAdresse von pointer_var selbst: " << &pointer_var;
  std::cout << std::endl;
}
```

Aufgabe

Implementieren Sie dieses Beispiel.
Machen Sie sich klar, was die drei Ausgaben bedeuten!.

Im Detail:

- Der Wert von var wird an einer bestimmten Adresse gespeichert.
- Der Pointer zeigt auf diese Adresse.
- Der Pointer ist aber selbst eine Variable, die wiederum eine Adresse besitzt.

Zeiger (5): Dereferenzieren

- Was bringt uns das Arbeiten mit Adressen?
- In der Praxis sind wir an dem Wert selbst interessiert, nicht an dessen Adresse!

$\Rightarrow$ Auflösen (d.h. Dereferenzierung) der Adresse.

- Derefenzierung mittels des unären Indirektionsoperators $*$, sprich Auflösung der Adresse von der Zeigervariablen.

Fortführung des Beispiels:

```
#include <iostream>
int main()
{
  int var = 101;       // Normaler Basisdatentyp
  int *pointer_var;    // Anlegen eines pointers
  pointer_var = &var; // Weise Adresse von var dem pointer zu
  // Dereferenzierung: hier wird der Wert des pointers in den
  // entsprechenden Wert aufgelöst:
  int tmp = *pointer_var;
  std::cout << "Adresse von var: " << &var;
  std::cout << "\npointer_var verweist auf Adresse: " << pointer_var;
  std::cout << "\nAdresse von pointer_var selbst: " << &pointer_var;
  std::cout << "\nAdresse von tmp: " << &tmp;
  std::cout << "\nWert von tmp: " << tmp << std::endl;
}
```

Zeiger (6): Dereferenzieren

Beispiel 28. Modifikation von Daten, auf die gezeigt wird:

```
#include <iostream>
int main()
{
  int var = 101;
  int *pointer_var;
  pointer_var = &var; // Der Pointer zeigt auf die Adresse von var!

  // Nun wird's spannend.
  std::cout << "Wert von var: " << var << std::endl;

  *pointer_var = 200;
  std::cout << "Wert von var: " << var << std::endl;

  *pointer_var += 15;
  std::cout << "Wert von var: " << var << std::endl;
}
```

- Man mache sich klar, weshalb wir diese Ergebnisse so erhalten?

⇒ Zeiger zeigt auf Adresse von var. D.h., wenn wir den Zeigerwert ändern, dann ändert sich var, da wir den Zeigerwert an die Adresse von var schreiben!

Zeiger (7): Dereferenzieren

Wir erweitern das Beispiel:

```
#include <iostream>
int main()
{
  int var = 101;
  int *pointer_var;
  pointer_var = &var; // Der Pointer zeigt auf die Adresse von var!

  int tmp; // Deklaration einer weiteren Variablen
  // Dereferenziere den pointer:
  tmp = *pointer_var; // Nun hat tmp, den Wert von pointer_var,
                      // also var.

  std::cout << "var= " << var << "\ntmp= " << tmp << std::endl;
  *pointer_var = 200;
  std::cout << "var= " << var << "\ntmp= " << tmp << std::endl;
  *pointer_var += 15;
  std::cout << "var= " << var << "\ntmp= " << tmp << std::endl;
}
```

`tmp` ist kein Zeiger! Und verweist auf eine andere Adresse, daher wird immer `var` verändert, wenn sich `pointer_var` ändert, aber `tmp` wird davon nicht beeinflusst.

Zeiger (8)

- Vorsicht, wenn Zeiger auf ein nicht-gültiges Objekt zeigt.

Aufgabe

Testen Sie den nachfolgenden Code. Was passiert?

```
int *pointer_var;
*pointer_var = 101;
```

- Zeiger können auf andere Zeiger verweisen und haben noch weitere Möglichkeiten. Wir werden diese aus verschiedenen Gründen aber nicht alle besprechen.
  - Zeiger sind gewöhnungsbedürftig und dieser Kurs stellt einen Einstieg dar. Wichtig ist, dass die grundlegende Syntax bekannt ist. Die gesamte Programmiersprache kann nicht in allen Details besprochen werden. Bei Interesse schauen Sie sich bitte selbst die weiterführenden Links aus VL 1 an.
  - Hinsichtlich Ihres Mathematik-Studiums bleibt festzuhalten: letztendlich ist dies kein reines Programmieren und daher wird nicht der gesamte Sprachumfang einer Programmiersprache benötigt!

C-Arrays (1): Einführung

- Oft haben wir es mit Vektoren (numerische lineare Algebra) oder einer Folge von Zahlenwerten (Erinnerung Newton-Beispiel 17) zu tun.
- Arrays (Felder) sind im Speicher fortlaufend angeordnete Elemente gleichen Typs.
- Syntax:

```
Typ array_name[n];
```

- Der Typ gibt wie vorher den Datentyp an, z.B. int, double, etc.
- Mit n geben wir die Anzahl der Elemente an.
- Arrays mit unterschiedlichen Typen gibt es in C++ nicht.
- Beispiele:

```
int array_int[12];     // 12 int Werte
float array_float[43]; // 43 float Werte
```

- Anhand des Typs erkennt der compiler, wie viel Speicher reserviert werden muss.

C-Arrays (2): Initialisierung

Beispiel 29. Erstellen und Füllen eines Arrays:

```
int var[5];
var[0] = 17;
var[1] = 4;
var[2] = 3;
var[3] = 32;
var[4] = 1;
```

- C++ fängt bei 0 an zu zählen (im Gegensatz zu Matlab beispielsweise).
- Meist werden Arrays mit for-Schleifen bearbeitet:

```
#include <iostream>

int main() {
  int var[5];
  for (unsigned int i=0; i<5; ++i)
  {
    var[i] = (i+1) * 2;
    std::cout << "Wert: " << var[i] << std::endl;
  }
}
```

C-Arrays (3): Initialisierung

Weitere Arten der Initialisierung:
Beispiel 30. Echte Feld-Initialisierung mithilfe einer initializer-list:

```
int var[5] = {4,5,8,6,78};
```

In diesem Falle ist es möglich, die Feldlänge wegzulassen:

```
int var[] = {4,5,8,6,78};
```

Es ist ebenfalls möglich, lediglich die ersten Werte zu initialisieren und für die weiteren nur Speicherplatz zu reservieren:

```
int var[5] = {4,5}; // Nur die ersten beiden Werte werden belegt.
```

Vorsicht!

Der Inhalt der eckigen Klammern `[]`, der die Anzahl an Elementen in einem Feld bestimmt, muss ein konstanter Ausdruck (constant expression) sein. Felder sind Blöcke statischen Speichers, deren Längen zur Kompilierzeit bereits feststehen müssen (`https://www.cplusplus.com/doc/tutorial/arrays/`).
Die Benutzung von sogenanntem dynamischem Speicher in C ist möglich, aber komplexer. Wir werden uns das nach den Arrays kurz anschauen.

C-Arrays (4): Bereichsüberschreitungen

- Es ist sofort klar, welches eine der häufigsten Fehlerquellen ist: die Bereichsüberschreitung. Wir wollen Elemente hinzufügen, das Feld hat aber keinen Platz mehr:

```
#include <iostream>
int main() {
  int var[5];
  for (unsigned int i=0; i<=5; ++i)
  {
    var[i] = (i+1) * 2;
    std::cout << "Wert: " << var[i] << std::endl;
  }
}
```

- Der obige Fehler ist subtil und geschieht sehr häufig! Der „off-by-one Error“.
- C++ beginnt bei 0 zu zählen und wir wollen das Feld bis einschließlich Index 5 füllen. Das macht aber 6 Werte, obwohl das Feld nur 5 Elemente hat.

Vorsicht!

Es ist syntaktisch korrekt, den gültigen Indexbereich eines Felds zu überschreiten. Daher wird der Compiler keinen Fehler auswerfen. Wir erhalten aber kryptische Fehlermeldungen beim Ausführen unseres Programms (Laufzeitfehler).

C-Arrays (5): Länge eines Arrays ermitteln

Beispiel 31. Bestimmung der Länge eines Feldes:

```
#include <iostream>
int main() {
  int var[7];
  for (unsigned int i=0; i<7; ++i)
     var[i] = (i+1) * 2;

  std::cout << "Gesamtgröße in Bytes: " << sizeof(var);
  std::cout << "\nEinzelgröße in Bytes: " << sizeof(var[0]);
  std::cout << "\nAnzahl der Elemente:    "
            << sizeof(var) / sizeof(var[0])
            << " = " << std::size(var) // Ab C++17
            << std::endl;
}
```

Mit `sizeof()` kann man die Speichergröße beliebiger Variablen abfragen, ebenso von Datentypen: `sizeof(int)`.

C-Arrays (6): Mehrdimensionale Arrays

- Mehrdimensionale Arrays entsprechen Matrizen oder Tensoren höherer Stufe.
- Syntax:

```
int matrix[2][5];
```

- Der erste Index entspricht der Zeile, der zweite Index entspricht der Spalte (wie bei einer Matrix).
- Natürlich würde

```
int matrix[10];
```

  denselben Zweck erfüllen. Wir müssten den richtigen Elementzugriff dann aber selbst implementieren.

- Initialisierung:

```
int matrix[2][5] = { {1,2,3,4,7},      // Zeile 0
                     {65,43,46,98,43} // Zeile 1
                   };
```

- Da die Zeilen aber hintereinander im Speicher liegen, können wir auch äquivalent wie folgt initialisieren:

```
int matrix[2][5] = {1,2,3,4,7,65,43,46,98,43};
```

C-Arrays und Zeiger (7)

- Sehr enge Beziehung zwischen Arrays (Feldern) und Zeigern.
- Feld belegt während des Programmablaufs einen Speicher mit n Elementen, dessen Anfangsadresse nicht verschoben werden kann.
- D.h. ein Array besitzt einen konstanten Zeiger auf das erste Element. Genau genommen kann der Bezeichner des Feldes auch als Zeiger auf das erste Element benutzt werden (und „vergisst“ dann die Feldlänge).
- Da die Elemente eines Arrays hintereinander im Speicher liegen, kann man mit Zeigeroperationen aber auch auf alle weiteren Elemente zugreifen.
- Die folgenden Ausdrücke sind äquivalent:
  - `a[0]` und `*a` verweisen auf den Wert des ersten Elements.
  - `a[i]`, `*(a+i)` und `*(i+a)` verweisen auf den Wert des Elements `i`. Damit ist syntaktisch auch `i[a]` korrekt, aber natürlich sehr unleserlich.
- Aufgrund dieser Äquivalenzen ist auch klar, weshalb der Index bei 0 anfängt.
- Dies ist ein Haupt-Anwendungsgebiet für Zeiger: wir können die Adressen eines Arrays direkt ansteuern und die Elemente somit an der Adresse manipulieren.

C-Arrays und Zeiger (8)

Beispiel 32. Einlesen mehrerer Ganzzahlen in ein Feld:

```
#include <iostream>
int main()
{
  int const size = 5;     // Mit 'const' ist 'size' ok als ...
  int array_int[size];    // ... Feldgröße: konstanter Ausdruck.
  int *iptr = array_int; // Pointer zeigt auf erstes Feldelement
  for (unsigned int i = 0; i < size; ++i)
  {
    std::cout << "Bitte Wert eingeben: ";
    std::cin >> *iptr;
    ++iptr; // Inkrement zum nächsten Element
  }
  std::cout << "Die eingelesenen Werte lauten: \n";
  for (unsigned int i = 0; i < size; ++i)
  {
    std::cout << array_int[i] << std::endl;
  }
}
```

Bemerkung: Wir dürfen `int const` oder `const int` schreiben; die erste Variante lässt sich allerdings besser auf kompliziertere Datentypen verallgemeinern.

C-Arrays und Zeiger (9)

Beispiel 33. C-Arrays und Zeiger (https://www.cplusplus.com/doc/tutorial/pointers/):

```
#include <iostream>
int main()
{
  int numbers[5]; // 5 Einträge heißt: 0,1,2,3,4
  int* p;

  p = numbers; // Zeigt auf erstes Element
  *p = 10;

  // Vier verschiedene Techniken, um pointer zu inkrementieren.

  // Möglichkeit 1: Inkrementoperator
  ++p;
  *p = 20;

  // Möglichkeit 2: Adressoperator
  p = &numbers[2];
  *p = 30;
 ...
```

C-Arrays und Zeiger (9.1) - Fortsetzung Beispiel

Beispiel 33. (Forts.)

```
  // Möglichkeit 3: explizite Inkrementierung
  p = numbers + 3;
  *p = 40;

  // Möglichkeit 4: Inkrementiere pointer um 4 (p+4) und
  // weise dann Wert mittels Dereferenzierungsoperator zu.
  p = numbers;  // Zurücksetzen des Pointers aufs erste Element.
  *(p+4) = 50;

  // Aufgepasst: Wir müssen selbst Sorge tragen,
  // dass keine Array-Bereichsüberschreitung auftritt.
  for (int n=0; n<7; ++n)
    std::cout << numbers[n] << ", ";

  std::cout << std::endl;
}
```

C-Arrays (10): Finale Bemerkungen zu Arrays

- Nicht vergessen: höherdimensionale Arrays können immer auch als eindimensionale Arrays geschrieben werden.
- Ebenso können Arrays für double, character (char), float, etc. angelegt werden.
- Vorsicht bei C-Strings in Verwendung mit Arrays. C-String ist veraltet und aus C, wird aber dennoch häufig benutzt, da viele C-Programme in C++ umgeschrieben worden sind. Aus Kompatibilitätsgründen sind C-Strings beibehalten worden. Falls Sie es genau wissen möchten: der C-String `"Feld"` hat in C++ den Typ `char const*` und entspricht den Character-Arrays a1 und a2 in

```
char const a1[] = {'F','e','l','d','\0'};
char const a2[] = "Feld";
```

Erinnerung: die Feldlänge is 5 wegen `'\0'` am Ende!

Ende VL 5

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 5 durch.

Dynamische Speicherverwaltung (1)

Wir beginnen mit dem Anlegen von Speicher für einzelne Variablen (Felder später):

- Anlegen (Allokation) von Speicherobjekten dynamisch zur Laufzeit erfolgt mit

```
new
```

- Dynamische Objekte müssen nach ihrer Benutzung wieder gelöscht werden. Die Deallokation erfolgt mit

```
delete
```

- Beim Anlegen mit `new` fordert das System so viel Speicherplatz an, wie der Datentyp erfordert. Als Rückgabewert erhalten wir eine Adresse des reservierten Speichers. Falls nicht genügend Speicher frei ist, bekommen wir einen Laufzeitfehler (engl. run time error).
- Syntax:

```
Typ* typ_zeiger = new Typ; // Allokation
...
delete typ_zeiger; // Deallokation
```

Dynamische Speicherverwaltung (2)

Beispiel 34. Umgang mit dynamischen Zeigern:

```
#include <iostream>
int main() {
  int* iptr1 = new int;
  int* iptr2 = new int;

  // Dynamischen Speicher belegen und dann Werte tauschen (swap)
  *iptr1= 100; *iptr2 = 200;
  std::cout << "Wert von *iptr1: " << *iptr1 << '\n'
            << "Wert von *iptr2: " << *iptr2 << std::endl;
  int tmp = *iptr2;
  *iptr2 = *iptr1;
  *iptr1 = tmp;
  std::cout << "Wert von *iptr1: " << *iptr1 << '\n'
            << "Wert von *iptr2: " << *iptr2 << std::endl;
  // Dynamischen Speicher deallokieren (umgekehrte Reihenfolge!)
  delete iptr2;
  delete iptr1;
}
```

Frage

Was passiert, wenn wir uns die Werte nach dem Freigeben erneut ausgeben lassen?

Dynamische Speicherverwaltung: Felder (3)

Erinnerung (statische Feld-Allokation):

```
int const n = 1000; // Maximale Anzahl Messwerte.
double diameter[n]; // Compiler muss Wert von n kennen!
```

Nachteile:

- Programm muss bei jeder Änderung der Länge n neu übersetzt werden.
- Möglicherweise wird zu wenig, zu viel, oder völlig unnötiger Speicherplatz angelegt.

Ausweg: dynamische Allokation und Deallokation.
Beispiel 35. Benutzung von `new[]` und `delete[]` für Felder:

```
int n; std::cin >> n;                  // Variable Anzahl Messwerte
double* diameter = new double[n]; // Dynamische Feldallokation
for (int i = 0; i < n; ++i)
  std::cin >> diameter[i];             // Benutzung des Felds
...
delete[] diameter;                     // Feld-Deallokation
```

Vorsicht!

Insbesondere bei Feldern muss der Speicher immer in umgekehrter Reihenfolge zur Allokation wieder freigegeben werden! (neuester Zeiger zuerst)

Aufzählungen

- Aufzählungen (engl. enumerations) ermöglichen oft besser lesbaren (sog. sprechenden) Code, indem Sie für Folgen von Ganzzahlen Namen vergeben.
- Syntax:

```
enum bezeichner {name1, name2, ..., nameN};
```

- Gute Alternative zu `const`-Deklarationen.
- Beispiel für `const`-Deklaration:

```
int const JAN = 0;
int const FEB = 1;
...
int const DEC = 11;
```

- Einfacher als Aufzählung:

```
enum monat {JAN, FEB, MAR, APR, MAY, JUN,
            JUL, AUG, SEP, OCT, NOV, DEC};
```

- Vorsicht! Der erste Bezeichner wird mit 0 belegt.
- Alternativ lässt er sich aber auch explizit belegen:

```
enum monat {JAN = 1, FEB, MAR, APR, MAY, JUN, ..., DEC};
```

Strukturen (1)

- Mit `struct` lassen sich Werte von unterschiedlichen Typen zusammenfassen.
- Eine Struktur belegt im Prinzip nur den Speicher für die enthaltenen Daten.
- Wir werden uns später noch mit Klassen beschäftigen, die einen größeren Funktionsumfang bieten (verallgemeinerte Strukturen).

Beispiel 36. Eine Struktur für einen Punkt in 2D:

```
struct point2D
{
  char name; // 'Name' des Punktes.
  double x;  // x-Koordinate.
  double y;  // y-Koordinate.
};

point2D p1;     // Variable p1 vom Typ point2d.
p1.name = 'A'; // Zugriff auf das Element name in p1.
p1.x =  1.7;
p1.y = -3.4;

point2D p2 = {'B', -2.17, 3.1}; // Initialisierung.
// Ab C++11 {...} auch bei Zuweisung:
p1 = {'X', 3.0, 3.0};
```

Strukturen (2)

Zugriff auf Strukturmitglieder

- Zugriff erfolgt über den Punktoperator `.`
- Syntax:

```
Strukturvariable.Elementname
```

Weitere Eigenschaften von Strukturen

- Können wie Basisdatentypen als Parameter oder Rückgabewerte von Funktionen dienen.
- Strukturen können wie Basisdatentypen verglichen werden.
- Es können Felder von Strukturen gebildet werden.
- Strukturen können verschachtelt werden.
  Z.B. ein Viereck als Struktur von vier Punkten.

Motivation

- Dritte wichtige Anwendung von Zeigern!
  (Erinnerung: 1. Zugriff auf Felder und 2. dynamischer Speicher)
- Bei „normalen“ Parameterübergaben werden einfache Werte übergeben, z. B. Zahlen (`int`, `double`, ...). Hier gibt es kein Problem bezüglich des Speicherplatzes.
- Allerdings kann man auch komplexere Datenstrukturen wie Felder, Strukturen oder Klassen als Parameter übergeben.
- Beispiel: Übergabe einer 1000×1000 double-Matrix ($8000000\,\mathrm{B} = 8\,\mathrm{MB}$!), was noch eine vergleichsweise kleine Matrix ist.
- Bei der uns bekannten Übergabe (call-by-value) wird eine Kopie des überreichten Parameters angelegt und weiterverarbeitet.
- Es müssten also für jeden Aufruf einer Funktion mit der Matrix als Parameter 8 MB an zusammenhängendem Speicher angelegt und kopiert werden.
- Besser wäre es also, der Funktion mitzuteilen, wo die Matrix im Speicher liegt, um diese direkt zu bearbeiten. Das funktioniert mit call-by-pointer und call-by-reference, wobei letzteres die einfachere Variante ist.

Call-by-value

Wir wollen an einem Beispiel genau anschauen, was bei call-by-value intern passiert.

Beispiel 37. Lokale Variablenänderung in einer call-by-value Funktion:

```
#include <iostream>
// Die Angabe des Typs ohne Parameternamen genügt hier,
// da wir die Variablennamen erst zur Implementierung benötigen
void my_function_call_by_value(int, int);

int main()
{
  int iwert1 = 101, iwert2 = 45;
  my_function_call_by_value(iwert1, iwert2);

  // Gib Originalwerte nach Funktionsaufruf nochmal aus:
  std::cout << iwert1 << '\t' << iwert2 << std::endl;
}
void my_function_call_by_value (int a, int b)
{
  std::cout << a << '\t' << b << '\t' << a*b << std::endl;
  a = 0; // Ändere Variable
  b = 4; // Ändere Variable
}
```

Call-by-pointer

- Wir haben im vorherigen Beispiel gesehen, dass tatsächlich eine Kopie der Parameter übergeben wird.

→ Eine solche Kopie muss daher immer erstellt werden. Dazu muss zunächst Speicherplatz reserviert werden und dann müssen alle Daten an den neuen Platz kopiert werden.

- Was für einfache Datentypen kein Problem ist, wird also bei speicherplatzintensiven Objekten zu einem erheblichen Mehraufwand.
- Bei Übergabe mittels call-by-pointer werden wir einen Zeiger auf die Adresse unseres Objekts an die Funktion geben.

Beispiel 38. Änderung der Funktion auf call-by-pointer:

```
void my_function_call_by_pointer(int* a, int* b)
{
  std::cout << *a << '\t' << *b << '\t' << (*a) * (*b)
            << std::endl;
  *a = 0; // Ändere Variable
  *b = 4; // Ändere Variable
}
// Funktionsaufruf (mit Adress-Operator)
my_function_call_by_pointer(&iwert1, &iwert2);
```

Call-by-pointer / call-by-reference

- Da nun in der zweiten Ausgabe die geänderten Werte stehen, haben wir unsere Funktion also erfolgreich angepasst.
- Es wird nun keine Kopie mehr übergeben, da wir direkt mit der Adresse arbeiten.
- Da wir aber in der Funktion mit Zeigern arbeiten ist der Code vergleichsweise umständlich. Wir müssen bei jedem Zugriff an die Derefenzierung denken.
- Abhilfe schafft hier der call-by-reference, bei dem wir der Funktion mitteilen, dass sie direkt die Adresse (Referenz) unserer Parameter zu erwarten hat.

Beispiel 39. Änderung des Beispiels auf call-by-reference:

```
void my_function_call_by_reference(int& a, int& b)
{
  std::cout << a << '\t' << b << '\t' << a * b
            << std::endl;
  a = 0; // Ändere Variable
  b = 4; // Ändere Variable
}
// Funktionsaufruf
my_function_call_by_reference(iwert1, iwert2);
```

Konstante Referenzen

- Wie wir gesehen haben, funktioniert call-by-reference also bzgl. Aufruf und Programmierung genauso wie call-by-value.
- Eine Änderung innerhalb der Funktion hat sich jetzt aber auch auf das übergebene Objekt ausgewirkt.
- Was geschieht, wenn wir eine große Matrix übergeben wollen, diese aber innerhalb der Funktion nicht ändern wollen/dürfen, wie z. B. bei einer Funktion zur Matrix-Vektor-Multiplikation?
- Call-by-value hilft uns hier nicht weiter, da uns ggf. der Speicher ausgeht.
- Wir müssen also den call-by-reference anpassen.
- Indem wir einem Parameter zusätzlich zum Typ und Namen noch das Schlüsselwort `const` geben, teilen wir dem Compiler mit, dass dieser Parameter nicht von der Funktion verändert werden darf.

```
void my_function_call_by_reference(int const& a, int& b)
```

- Wenn wir versuchen, unseren Code mit `const` zu kompilieren erhalten wir die folgende Fehlermeldung.

```
Fehler: Zuweisung der schreibgeschützten Referenz »a«
   a = 0; // Ändere Variable
```

- Der Compiler passt also auf, dass wir keinen unbeabsichtigten Unfug treiben.

Argumente an die main-Funktion übergeben (1)

- Wir haben zusätzliche Argumente bisher immer über die Kommandozeileneingabe abgefragt.
- Wir können aber bereits beim Aufruf unseres Programms Argumente an die main-Funktion übergeben.
- Mit Argumenten sieht die main-Funktion z.B. wie folgt aus

```
int main(int argc, char** argv) {...}
```

- Hierbei ist `**` eine sogenannte doppelte Indirektion: Deklaration eines Zeigers auf einen Zeiger, der dann auf eine Variable verweist.
- `argc` (argument count) Anzahl der Argumente, die übergeben wurde.
- `argv` (argument values) Feld von C-Strings (`char`-Felder). Da jeder Eintrag des Feldes selbst wieder ein Feld ist, die doppelte Indirektion.
- `argv[0]` enthält immer den Programmnamen (Name der ausführbaren Datei), daher gilt `argc` ≥ 1 (außer auf exotischen Systemen, die das nicht können).
- Aufpassen: `argv[argc]` ist natürlich eine Bereichsüberschreitung!
- Beachten Sie: die einzelnen C-Strings sind i.A. unterschiedlich lang, `argv` ist also kein „Matrix-artiges“ zweidimensionales Feld.

Argumente an die main-Funktion übergeben (2)

Beispiel 40. Übergabe von Argumenten an die main-Funktion

```
#include <iostream>

// Statt doppelter Indirektion **
// ist auch ein Zeiger auf char-Felder möglich.
int main(int argc, char* argv[])
{
  std::cout << "Die Argumente der Kommandozeile an main:\n";
  std::cout << "Programmname: " << argv[0] << std::endl;
  for (int i = 1; i < argc; ++i)
    std::cout << argv[i] << std::endl;
}
```

Argumente an die main-Funktion übergeben (3)

- Kompilierung (der Dateiname sei mainargv.cc):

```
prof@luh:> g++ mainargv.cc
```

- In der Kommandozeile starten wir das Programm wie folgt:

```
prof@luh:> ./a.out Hallo Studierende wie geht es Ihnen ?
```

- Alternativ ein Text in einfachen Anführungszeichen '...' (ohne Kommata; aber Vorsicht mit Sonderzeichen wie z.B. &).

```
prof@luh:> ./a.out 'Hallo Studierende wie geht es Ihnen ?'
```

- Viele Codes nutzen diese Funktionalität, um Dateipfade für Ein-/Ausgabedateien zu übergeben.
- Z.B. Parameterfiles für Löser-Toleranzen, Materialparameter, usw.

Argumente an die main-Funktion übergeben (4)

Konstante Argumente (guter Stil)

Im Prinzip darf man die C-Strings `argv[i]` wie auch den Zeiger `argv` verändern. In der Regel ist das aber unsinnig und man sollte es verbieten. Hier alle Möglichkeiten:

```
char const* const argv[]; // Konstanter Zeiger auf konstante Strings
const char* const argv[];
char const* const* argv;  // +
const char* const* argv;

char const* argv[]; // Konstanter Zeiger auf veränderbare Strings
const char* argv[];
char const** argv;  // +
const char** argv;

char* const argv[]; // Veränderbarer Zeiger auf konstante Strings
char* const* argv;  // +
```

In jeder Gruppe sind die Varianten äquivalent und Sie sehen, dass durch die doppelte Indirektion alles etwas unübersichtlich wird. Ich finde die Varianten mit + und mit Kommentartext am klarsten und die mit `const` ganz zu Beginn am verwirrendsten; deshalb die Empfehlung, `const` immer dem Datentyp nachzustellen (hier `char`). Konkret sollten Sie in der Regel die dritte oder die erste Variante aus der ersten Gruppe für `main` verwenden.

Ende VL 6

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 6 durch.

Motivation

- Wie wir gesehen haben ist die Arbeit mit C-Arrays oft behäbig:
  - Bei Speicherzugriffsfehlern erhalten wir nur kryptische Laufzeitfehler
  - Die dynamische Speicherverwaltung ist aufwändig und fehleranfällig
  - Die Länge eines Felds lässt sich (bis C++14) eher kompliziert bestimmen.
  - C-Zeichenketten (char-Arrays) sind vergleichsweise kompliziert.

  $\Rightarrow$ Darum haben wir diese hier auch kaum behandelt.
- Aber: C-Arrays sind schnell und brauchen nur Speicherplatz für die Daten an sich.
- C++ ist aber objektorientiert und ermöglichst es, Klassen mit zugehörigen Funktionen (Methoden) zu schreiben. (Wie erfahren wir später ab VL 8.)
- Die Standardbibliothek bietet für viele bekannte Datenstrukturen bereits fertige Klassen (templates).
- Da diese auf einem allgemeinen Container-Konzept beruhen, existieren diverse generische Algorithmen (Sortieren, Vertauschen von Elementen, ...)
- Komplette Liste: `https://en.cppreference.com/w/cpp/container`

C++-Arrays (1)

- Seit C++11 existieren auch in C++ eigene Felder.
- Diese vereinen Vorteile von C-Arrays (Speicheraufwand, Zugriffsgeschwindigkeit) mit dem C++-Container-Konzept.
- Ebenso wie bei C-Arrays muss die Länge zur Kompilierzeit feststehen (die Länge ist ein Template-Parameter, wird also in spitzen Klammern <> angegeben).
- Der erste Template-Parameter ist der Datentyp. Wie bei Funktions-Templates werden die Template-Parameter in spitzen Klammern übergeben.

Beispiel 41. Initialisierung von C und C++-Arrays:

```cpp
#include <iostream>
#include <array>

int main()
{
  int cFeld[3] = {1,2,3};
  std::array<int,3> cppFeld = {1,2,3};
  std::array prime = {2,3,5,7}; // Ab C++17 auch ohne <Typ,N>!
}
```

Vorsicht!

Die Variable `cppFeld` ist kein Zeiger!
Diesen erhält man aber als `&cppFeld[0]` oder `cppFeld.data()`.

C++-Arrays (2)

- Neben dem bekannten Zugriff über `operator[]` existiert auch die Klassenmethode `at()`.
- Die Länge des Feldes lässt sich nun über `size()` abfragen.

Beispiel 42. Neue Funktionalitäten des C++-Arrays:

```
#include <iostream>
#include <array>
int main()
{
  std::array<int,3> a = {1,2,3};
  std::cout << a[0] << " = "
            << a.at(0) << " = "
            << a.front() << '\n'; // Zugriff auf erstes Element
  std::cout << a.back() << '\n';  // Zugriff auf letztes Element
  std::cout << a.size() << " = "
            << std::size(a) << " = "
            << sizeof(a)/sizeof(a[0]) << '\n'; // Feldgröße
  a.fill(0); // Alle Einträge werden auf 0 gesetzt.
}
```

Exceptions (1)

- C benutzt Fehlercodes, um genauer zu beschreiben, was schief läuft: `https://www.geeksforgeeks.org/error-handling-c-programs/`

→ Dies benutzen wir auch bei `return 0` in `int main()`, da das Betriebssystem ebenso arbeitet.

- In C++, wurden zusätzlich Exceptions (Ausnahmen) eingeführt. Wer bereits in Java programmiert hat, kennt dieses Konzept.
- Im Prinzip sind Exceptions Kontrollstrukturen, die den Programmablauf ändern. Sie sollten aber in der Numerik (fast) nie benutzt werden! Bei echten Fehlern ist in der Regel ein Programmabbruch besser.

Exceptions (2)

- Gelangt eine Funktion in einen fehlerhaften Zustand, so wirft (throw) sie hierzu eine passende Exception.
- Beispiel hierfür ist eine Funktion, die in eine Datei schreiben möchte, diese aber im Dateisystem nicht finden kann.
- Der Programmablauf wird an dieser Stelle abgezweigt und die Exception wird nacheinander an alle aufrufenden Funktionen weitergereicht.
- Eine dieser Funktionen muss die Exception fangen (catch) und passend reagieren, um sicherzustellen, dass das Programm immer noch in einem funktionsfähigen Zustand bleibt.
- Fängt keine der Funktionen (einschließlich main) die Exception ab, so bricht das gesamte Programm ab.
- Die Funktion `at()` wirft beispielsweise eine Exception bei Bereichsüberschreitung, `operator[]` aber nicht.

Exceptions (3)

Wir wollen die Konsolenausgabe der beiden Alternativen vergleichen und erweitern hierzu unser vorheriges Beispiel um je eine Zeile.

```
a[5] = 3;
```

```
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

Diese Ausgabe kennen Sie vermutlich schon im Zusammenhang mit C-Arrays. Sie ist nach wie vor nicht wirklich hilfreich. Der Fehler könnte weiterhin irgendwo liegen.

```
a.at(5) = 3;
```

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 5) >= _Nm (which is 3)
Abgebrochen (Speicherabzug geschrieben)
```

Wir erhalten hier die `std::out_of_range`-Exception und bekommen sogar klar gesagt, dass unser Index 5 für ein 3-elementiges Feld zu groß war.

- Bitte sehen Sie zur Ausführung auch die nächste Seite ...

Exceptions (4)

```
... // wie vorher
// Zeile von vorher
std::cout << a.front() << ' ' << a.back() << std::endl;
a[5] = 3; // neu
//a.at(5) = 3; // neu
std::cout << a.front() << ' ' << a.back() << std::endl; // neu
```

Dies liefert u.U. eine korrekte Ausgabe (es ist aber nicht klar, was intern passiert und wird im Allg. nicht funktionieren!):

```
[prof@luh ~]$ g++ -std=c++11 seite_123.cc ; ./a.out
1 = 1 = 1
3
3 = 3 = 3
0 0
0 0
```

Hier wird zwar kein Laufzeitfehler erzeugt (ansonsten würde die letzte Zeile nicht ausgegeben; siehe nächste Seite), der Wert 3 an der Stelle 5 wird aber auch nicht gesetzt.

Exceptions (5)

```
... // wie vorher
// Zeile von vorher
std::cout << a.front() << ' ' << a.back() << std::endl;
//a[5] = 3; // neu
a.at(5) = 3; // neu
std::cout << a.front() << ' ' << a.back() << std::endl; // neu
```

Die sichere Variante ist mit `a.at()` zu arbeiten:

```
[prof@luh ~]$ g++ -std=c++11 seite_123.cc ; ./a.out
1 = 1 = 1
3
3 = 3 = 3
0 0
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at
Abgebrochen (Speicherabzug geschrieben)
```

C++-Vektoren (1)

- Falls wir beim Kompilieren noch nicht wissen, wie groß unser Container sein soll, wir aber trotzdem zusammenhängenden Speicher benötigen, dann können wir mit `std::vector` arbeiten.
- Dieser „ersetzt“ und verbessert dynamisch angelegten Speicher in C, hat aber nichts mit dem mathematischen Vektor-Konzept zu tun!
- Da die Länge nicht fest ist, fällt sie als Template-Parameter weg.
- Wir können dem Vektor bei der Konstruktion/Erstellung aber trotzdem eine (erste) Länge mitteilen. Er wird bei primitiven Datentypen mit Nullen gefüllt.

```
std::vector<int> vecOfInts(5);
```

- Natürlich funktioniert auch wieder die Initialisierung mit initializer list:

```
std::vector<int> v = {7, 5, 16, 8};
```

- Der Element-Zugriff erfolgt wie bei Arrays per `operator[]` (ohne Exceptions) oder per `at()` (mit Exceptions); auch `front()` und `back()` gibt es.

C++-Vektoren (2)

- Im Gegensatz zu Arrays können bei Vektoren Elemente anfügt werden:

```
v.push_back(10);
```

- Unser vorheriger Vektor hat jetzt die Einträge $7, 5, 16, 8, 10$ (testen Sie bitte selbst!).
- Wir können im Gegenzug aber auch den letzten Eintrag mit `pop_back()` löschen.
- Mit `clear()` können wir auch den Inhalt des gesamten Vektors löschen.
- Weitere Funktionen die den Inhalt ändern können, bauen auf Iteratoren auf. Diese werden wir etwas später noch behandeln.

Frage

Ist es sinnvoll, die Länge des zugrundeliegenden dynamischen Speichers bei jedem push oder pop anzupassen? Wir wollen ja, dass alle Elemente hintereinander im Speicher liegen.

C++-Vektoren (3)

- Da wir nicht davon ausgehen können, dass der Speicher direkt hinter unserem letzten Element frei ist, wäre das Anlegen und Kopieren tatsächlich sehr aufwändig.
- Stattdessen wird, sobald neuer Speicher benötigt wird, einfach mehr reserviert als aktuell angefragt.
- Wir haben nun also die Anzahl der tatsächlich belegten Elemente, die wir wie gewohnt mit `size()` abfragen können.
- Wir haben aber zusätzlich eine Kapazität, die eventuell größer ist als `size()`. Diese können wir mit `capacity()` abfragen.
- Wenn wir z.B. beim Einlesen von Daten vorher wissen sollten, wie viel Platz wir benötigen, können wir diesen mit `reserve(n)` reservieren. Es wird dann mindestens so viel Platz reserviert, wie wir angefragt haben.
- Seit C++11 haben wir zusätzlich die Funktion `shrink_to_fit()`, die es uns ermöglicht die aktuelle Kapazität auf `size()` zu schrumpfen (z.B. wenn die Länge nach dem Einlesen von Daten endgültig feststeht).

C++-Vektoren (4)

Beispiel 43. Berechnung der Norm eines eingegebenen Vektors unbekannter Länge:

```
#include <iostream>
#include <vector>
#include <cmath>
int main()
{
  std::vector<double> vec;
  std::cout << "Berechnung der euklidischen Norm eines Vektors\n"
            << "Bitte zeilenweise die Elemente eingeben\n"
            << "Eingabe beenden mit beliebigem Zeichen, z.B. 'a'\n";
  double norm = 0, vec_entry;
  std::cin >> vec_entry;
  while (std::cin.good()) { // double eingelesen?
    vec.push_back(vec_entry);
    norm += vec_entry*vec_entry;
    std::cin >> vec_entry;
  }
  std::cout << "Der Vektor\n(" << vec.at(0);
  for (unsigned int i = 1; i < vec.size(); ++i)
    std::cout << ", " << vec.at(i);
  std::cout << ")\n hat die 2-Norm " << std::sqrt(norm) << std::endl;
}
```

Listen (1)

- Arrays und Vektoren haben gemeinsam, dass ihre Daten hintereinander im Speicher liegen müssen.
- Wenn also im Verlauf des Programms regelmäßig Daten „in der Mitte“ hinzugefügt und gelöscht werden, muss im Hintergrund immer wieder der Speicherbereich angefasst werden.
- Die `std::list` ist eine sogenannte doppelt verkettete Liste, d.h. jedes Element kennt sowohl seinen Nachfolger als auch seinen Vorgänger. Oder besser gesagt die Adressen der beiden. Wir können die Liste also in beiden Richtungen sequentiell durchlaufen und die einzelnen Elementen können an beliebigen Stellen im Speicher liegen.
- Im Gegensatz zu Arrays und Vektoren verlieren wir aber den sogenannten random access über `operator[]` bzw. `at()`.
- Dafür können wir aber mit `push_front()` und `pop_front()` auch Elemente am Anfang einfügen bzw. entfernen.

Bemerkung

Es gibt natürlich noch weitere Container (siehe Motivation), diese werden wir aber aus Zeitgründen nicht behandeln. Insbesondere `array` und `vector` sind aber für die Numerik wichtig.

Iteratoren (1)

- Wie der Name schon verrät, sind Iteratoren dazu gedacht, über die Elemente eines Containers zu iterieren.
- Iteratoren zeigen auf eine bestimmte Position in einem Container. Sie sind eine Verallgemeinerung von Zeigern.
- Es gibt eine Hierarchie von Iteratoren, geordnet nach Funktionalität.
  - Die meisten Möglichkeiten bietet der Random-Access-Iterator. Er kann, wie wir es von Feldern und Vektoren gewohnt sind, von einer Stelle an eine beliebige andere springen, z.B. mit `it+=2`.
  - Der bidirektionale Iterator kann nur zum direkten Vorgänger oder Nachfolger springen. Dementsprechend können zwei bidirektionale Iteratoren auch nur auf Gleichheit oder Ungleichheit verglichen werden.
  - Der Vorwärtsiterator ist noch weiter eingeschränkt und kann nur zum direkten Nachfolger springen, der Vorgänger ist nicht bekannt.
  - Die geringste Funktionalität bieten Input- und Ouput-Iteratoren.
    - Der Input-Iterator bietet nur lesenden Zugriff und kann den Container auch nur vorwärts durchlaufen, z.B. um die Elemente als Eingabe für eine Funktion zu verwenden.
    - Der Ouput-Iterator bietet Schreib-Zugriff und kann den Container ebenfalls nur vorwärts durchlaufen. Er ist im Prinzip das direkte Gegenstück zum Input-Iterator.
- Hierarchisch stehen Input und Output auf derselben Stufe, alle weiteren Iteratoren sind inklusiv. D.h. jeder Random-Access-Iterator ist auch ein bidirektionaler Iterator usw. Jeder Zeiger ist Random-Access-Iterator.

Iteratoren (2)

- Iteratoren ermöglichen es, generische Funktionen zu schreiben, wie beispielsweise Sortieralgorithmen, Suchalgorithmen uvm.
- Vom Prinzip ist das zu vergleichen mit dem Euklidischen Algorithmus, der für Elemente eines euklidischen Rings immer gleich funktioniert.
- In der Algorithms Library finden wir sehr viele Algorithmen die auf Iteratoren basieren: `https://en.cppreference.com/w/cpp/algorithm`
- Grundlage hierfür sind Iterator-basierte for-Schleifen, die unabhängig von dem gewählten Container funktionieren
- Erinnerung: über einen Vektor können wir im C-Stil mittels

```
for (int i = 0; i < v.size(); ++i)
  std::cout << v[i] << '\t';
```

  iterieren und z.B. die Elemente ausgeben.

- Beispiel 44. For-Schleife mit Iterator:

```
std::vector<int> v = {1,2,3};
for (std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
  std::cout << *i << '\t';
```

Iteratoren (3)

- Die drei wichtigsten Punkte sind
  - der Typ des Iterators `std::vector<int>::iterator`;
  - der Iterator `begin()`, der auf das erste Element des Vektors zeigt;
  - der Iterator `end()`, der hinter das letzte Element des Vektors zeigt (dies entspricht dem Zeiger `v+n` beim Array `float v[n]`).
- Reversible Container bieten zusätzlich `rbegin()` und `rend()`: bidirektionale Iteratoren zum Durchlaufen des Containers in umgekehrter Richtung.
- Alle Container stellen `begin()` und `end()` bereit, womit man einfacher das range-based for benutzen kann:

```
std::vector<int> v = {1,2,3};
for (int i: v)
  std::cout << i << '\t';
```

- Wie bereits erwähnt, bieten `std::list` und `std::vector` weitere generische Funktionen, die mit Iteratoren funktionieren.
- Eine wichtige Funktion ist hier `insert(pos, value)`, welche ein Element `value` vor der Position des Iterators `pos` einfügt.
- Analog entfernt `erase(pos)` das Element, auf das der Iterator `pos` zeigt.

Iteratoren (4)

Beispiel 45. Selbstgeschriebene generische Funktion, um die Reihenfolge von Daten mit bidirektionalem Iterator umzukehren:

```
#include <iostream>
#include <array>
#include <vector>
#include <list>
template<class BidirIt>
void my_reverse(BidirIt first, BidirIt last)
{
  // last zeigt hinter letztes Element
  --last; // jetzt auf letztes Element
  while (first != last)
  {
    // Tausch der Elemente *first und *last
    auto tmp = *first; // automatische Typerkennung
    *first = *last;
    *last = tmp;
    ++first; // Setze first auf erstes ungetauschtes Element
    if (first == last)
      break; // Sonderfall: ungerade Anzahl Elemente
    --last;  // Setze last auf letztes ungetauschtes Element
  }
}
```

Iteratoren (5)

Beispiel 45. (Forts.)

```
int main()
{
 std::array<int,9> arr{1,2,3,4,5,6,7,8,9};
 my_reverse(arr.begin(), arr.end());
 for (int i: arr)
   std::cout << i << ' ';
 std::cout << std::endl;

 std::vector<double> vec{1.0,2.0,3.0,4.0};
 my_reverse(vec.begin(), vec.end());
 for (double d: vec)
   std::cout << d << ' ';
 std::cout << std::endl;

 std::list<char> lis{'A','B','C','D','E'};
 my_reverse(lis.begin(), lis.end());
 for (char c: lis)
   std::cout << c << ' ';
 std::cout << std::endl;
}
```

Typdefinitionen (1)

- Unsere Bezeichner werden zum Teil immer länger und der Code dadurch tendenziell immer unleserlicher.
- Insbesondere die Breite unseres Codes wird immer größer. Eine verbreitete Konvention sind maximal 80 Zeichen pro Zeile. Warum?
  - IBM-Lochkarten hatten 80 Zeichen. Damit wurde früher z.B. Fortran programmiert.
  - Man kann auch auf kleineren Geräten (Laptop) die ganze Zeile lesen.
  - Man kann auch in „Split-View“ zwei Codes nebeneinander legen und vergleichen.
  - Man kann den Code gut lesbar und ohne seltsame Zeilenumbrüche auf Papier drucken.
- Ausweg? Typdefinitionen ermöglichen die Definition von kurzen Synonymen.

Aufgabe

Testen Sie an ihrem Rechner einmal verschiedene Zeilenlängen (80, 100, 120,..) und schauen Sie, welche Zeilen noch vollständig angezeigt werden, wenn Sie bspw. Ihren Editor auf halbe Bildschirmweite (split-view) ziehen.
Um ein bisschen Tipparbeit zu sparen, hier sind 20 Bindestriche:

```
--------------------
```

Typdefinitionen (2)

- Syntax:

```
using Aliastyp = Originaltyp; // Modernes C++
typedef Originaltyp Aliastyp; // C, altes C++
```

- Beispiel 46. Beispiele für Typdefinitionen:

```
#include <iostream>
#include <vector>
using uint_t = unsigned int;
using uint_vec_t = std::vector<unsigned int>;
uint_vec_t v; // Hier hat v Länge 0
for (uint_t i = 0; i < 10; ++i)
  v.push_back(i);
```

- Konvention: oft fügt man dem neuen Bezeichner ein `_t` an, um deutlich zu machen, dass es sich hierbei um einen Typ handelt.
- Unser Beispiel würde aber natürlich genauso gut mit `uint` und `uint_vec` funktionieren.

Funktionspointer(1)

- Wir können nicht nur Zeiger auf Datentypen übergeben.
- Adressen von Funktionen sind ebenfalls mögliche Parameter einer Funktion!
- Syntax:

```
double my_fct(double (*fct)(double), double x);
```

- Aufruf der Funktion mit

```
double val = my_fct(&fct, 1.0);
```

- Wir haben `my_fct` mitgeteilt, dass als erster Parameter eine Funktion erwartet wird, die
  1. als Rückgabetyp double besitzt;
  2. innerhalb unserer neuen Funktion den Bezeichner `fct` besitzen soll;
  3. einen Parameter mit Datentyp `double` enthält.
- Wir können also nur Funktionen übergeben, deren Signatur dazu passt.
- Code wird schnell unübersichtlich: Verwendung von `using`/`typedef`.
- Wir schauen uns dies im Folgenden anhand eines Beispiels genauer an.
- Dazu verallgemeinern wir das Newton-Verfahren zur Wurzelberechnung (Beispiel 17 aus dem Kapitel eigene Funktionen) auf beliebige reelle Funktionen.

Funktionspointer (2)

Bemerkung Newton-Verfahren

- Finde Nullstelle x^* einer Funktion $f : \mathbb{R} \to \mathbb{R}$.
- Wähle Startwert x_0 und iteriere. Wir erhalten Näherungen

$$x_1, x_2, \ldots, x_n, \ldots$$

- Die Iteration ist abgeschlossen, wenn $|f(x_n)| < tol$, z.B. $tol = 10^{-10}$. Der Funktionswert ist also hinreichend nahe bei 0 und $x^* \approx x_n$.
- Basisidee: Taylorentwicklung von $f(x_{k+1})$:

$$f(x_{k+1}) = f(x_k) + f'(x_k)(x_{k+1} - x_k) + O((x_{k+1} - x_k)^2).$$

- Einsetzen unserer Bedingung und Weglassen der Terme höherer Ordnung:

$$f(x_{k+1}) \stackrel{!}{=} 0 \approx f(x_k) + f'(x_k)(x_{k+1} - x_k).$$

- Umstellen nach x_{k+1} ergibt dann unsere Iterationsvorschrift (solange $f'(x_k) \neq 0$)

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \ldots$$

Funktionspointer (3)

- In unserem einfachen Beispiel war die Funktion $f(x) = x^2 - y$. Eine Nullstelle ist somit die Quadratwurzel von y mit $y > 0$.
- Wir benötigen i.A. also eine C++-Funktion, die den Wert der Zielfunktion und den Wert der Ableitung für ein gegebenes x berechnet.
- Hierbei sehen wir einen weiteren Nutzen von call-by-reference: wir können mehrere Werte (indirekt) zurückgeben lassen.
- Unser Funktionskopf sieht also wie folgt aus:

```
void compute_f_and_df(double x, double& f, double& df)
```

- Wir definieren mittels `using` oder `typedef` den Funktionszeigertyp:

```
using fdf_ptr = void (*)(double, double&, double&); // Variante 1
typedef void (*fdf_ptr)(double, double&, double&);  // Variante 2
```

Achtung, Klammern sind wichtig: `void *fdf_ptr(...)` wäre eine Funktion, die einen `void*` zurückgibt!

Funktionspointer (4)

Beispiel 47. Allgemeine Newtonmethode mit Funktionspointern:

```
#include <iostream>
#include <limits>
#include <cmath>
typedef void (*fdf_ptr)(double, double&, double&);
double my_newton(fdf_ptr compute, double x, double eps = 1.0e-10)
{
  double e = 16 * std::numeric_limits<double>::epsilon();
  e = std::max(e, eps);
  double f, df;
  compute(x, f, df);
  int iter = 0;
  std::cout << iter << '\t' << x << '\t' << f << '\n';

  while (std::abs(f) > e)
  {
    x = x - f / df;
    compute(x, f, df);
    std::cout << ++iter << '\t' << x << '\t' << f << '\n';
  }
  return x;
}
```

Funktionspointer (5)

- Unsere Newtonmethode für beliebige Funktionen ist somit implementiert.
- Jetzt brauchen wir nur noch eine Funktion mit Ableitung.
- Danach testen wir in der main-Methode.

Beispiel 47. (Forts.)

```
void sqrt_two(double x, double& f, double& df)
{
  f  = x * x - 2.0;
  df = 2.0 * x;
}

int main()
{
  double x = 2.0;
  double s = my_newton(&sqrt_two, x);
}
```

- Wir könnten jetzt also beliebige Funktionen implementieren und deren Nullstellen berechnen.
- Funktionspointer eignen sich also sehr gut, wenn wir allgemeine Algorithmen implementieren wollen, die auf Funktionen aufbauen.

Ende VL 7

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 7 durch.

OOP versus prozedurales Konzept

- OOP = objektorientierte Programmierung.
- Prozedurales Programmieren: Daten und Funktionen bilden keine wirkliche Einheit.

→ Häufige Probleme bei falschem Zugriff, z.B. nicht initialisierte Daten.

→ Erweiterung von Programmen aufwändig und fehleranfällig.

- OOP: Objekte bilden Einheit aus Daten und Funktionen.
- OOP: Daten = Eigenschaften, Funktion = Methode.
- Es ist selbstverständlich, dass die Vorteile des OOP am Anfang noch nicht ganz klar sind.
- Die C++-Datenstrukturen waren aber alle bereits Klassen. Viele Funktionalitäten sollten nicht neu sein.
- Wir werden im Folgenden lernen, eigene Klassen zu schreiben.

OOP-Paradigmen

- Datenabstraktion: Definition fortgeschrittener Datentypen, die die Eigenschaften und Methoden von Objekten beschreiben.
- Datenkapselung: Einzelne Elemente eines Objekts einer Klasse können von außen vor unerlaubtem Zugriff geschützt werden. Datenaustausch mit anderen Objekten geschieht über offene Schnittstellen.
- Vererbung: Neue Objekte können aus vorhandenen Objekten abgeleitet werden, die dann die Eigenschaften des Elternobjekts vererbt bekommt.
- Polymorphie: Vererbung kann auf ganze Familien gleicher Objekte angewendet werden. Abhängig vom Typ des Objekts können zur Laufzeit verschiedene Eigenschaften (Daten) verwendet werden oder andere Methoden genutzt werden.

Klassen (1)

- Eine Klasse ist ein fortgeschrittener Datentyp.
- Erinnerung: einfache oder Basisdatentypen sind `int`, `double`, `float`, `char`, ...
- Klassen sind Verallgemeinerungen von `struct`.
- Der Name Klasse stammt von klassifizieren ab.
- Klassen eignen sich zur Abbildung von Objekten der „realen“ Welt (z.B. der Mathematik, mit Matrizen und Vektoren).

Klassen (2)

- Syntax (Klassendeklaration):

```
class Point2d
{
   // Eigenschaften und Methoden eines Punkts im 2-dim Raum.
}; // Semikolon am Ende. Wichtig!
```

- Klassenname muss eindeutig sein und erzeugt einen eigenen Gültigkeitsbereich (wie ein namespace).
- Klassenname sollte selbsterkärend sein.
- Oft verwendete Konvention: Klassenname beginnt mit Großbuchstaben.

Klassen (3)

- Die Elemente der Klasse werden Member genannt.
- Member sind Eigenschaften (Daten/Variablen) oder Methoden (Funktionen).
- Syntax (Beispiel):

```
class Point2D
{
  // Deklaration verschiedener Elemente:
  double M_x;  // M_ = "uglifier" für
  double M_y;  // Klassenmember (nicht zwingend notwendig).

  void move(double x, double y); // Verschiebe Punkt.
  void move(Point2D p);          // Dito.
  void draw();
  double norm();
};
```

- Wir haben zwei Eigenschaften und vier Methoden deklariert.
- Die Eigenschaften werden auch Klassendaten genannt und sind die Variablen im lokalen Klassennamespace.
- Die Methoden sind die Funktionen, die die Klasse zur Verfügung stellt. In der Regel operieren sie auf den Klassendaten.

Klassen (4): Definition von Klassenmethoden

- Beispiel 48. Alternative 1: Deklaration mit Definition innerhalb der Klasse:

```
class Point2D
{
  void move(double x, double y) { M_x = x; M_y = y; }
};
```

- Beispiel 49. Alternative 2: Definition der Funktion außerhalb der Klassendeklaration (vgl. Kapitel eigene Namensräume):

```
class point2D
{
  void move(double x, double y);
};

void Point2D::move(double x, double y)
{
  M_x = x;
  M_y = y;
}
```

- Erinnerung: Deklaration bedeutet, die Signatur der Funktion festzulegen. Hierbei müssen die Parameter noch keine Bezeichner erhalten. Definition beinhaltet die eigentliche Funktionalität/Implementierung, wofür die Parameternamen benötigt werden. Alternative 2 ist guter Stil; Alternative 1 ist ok bei Einzeilern wie `move()`.

Klassen (5): Definition von Klassenmethoden

- Die Syntax für extern definierte Klassenfunktionen ist analog zu extern definierten Namespace-Funktionen:

```
Typ Klassenname::Funktionsname(Parameter)
{
  // Anweisungen
}
```

- Bei Variante 2 ist es wichtig, den Klassennamen mittels des Scope-Operators anzugeben. Ansonsten hätten wir lediglich eine globale Funktion definiert.
- Die Klassenmethoden stellen die Schnittstelle zur Außenwelt dar. Es ist guter Stil, nur damit die Daten der Klasse anzusprechen.

Klassen (6): Objekte definieren

- Bisher haben wir Klassen inklusive deren Variablen und Methoden deklariert und die Methoden zusätzlich definiert.
- Wie benutzen wir diese Klasse nun?

→ Definition eines Objekts. Auch als Instanz einer Klasse bezeichnet.

- Die Definition eines Klassenobjekts ist analog zu allen anderen Datentypen:

```
Klassenname Objekt;
```

- Beispiel:

```
Point2D nullpunkt;
```

- Bei dieser Definition wird nun das Objekt im Speicher angelegt.

Klassen (7.1): Beispiel (wird noch nicht funktionieren!)

Beispiel 50. Eine einfache Klasse für Punkte in einem zweidimensionalen kartesischen Koordinatensystem:

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

class Point2D
{
  double M_x;
  double M_y;

  void move(double x, double y);
  void move(Point2D p);
  void draw();
  double norm();
}; // Semikolon nicht vergessen!
...
```

Klassen (7.2): Beispiel (wird noch nicht funktionieren!)

Beispiel 50. (Forts.)

```
// Definition der Methoden.
void Point2D::move(double x, double y)
{
  M_x = x;
  M_y = y;
}

void Point2D::move(Point2D p)
{
  move(p.M_x, p.M_y); // Elementzugriff über Punktoperator
}

void Point2D::draw()
{
  std::cout << '(' << M_x << ", " << M_y << ")\n";
}

double Point2D::norm()
{
  return std::sqrt(M_x*M_x + M_y*M_y);
}
```

Klassen (7.3): Beispiel (wird noch nicht funktionieren!)

```
int main()
{
  Point2D A;
  Point2D B;

  A.move(2.3,-3.5);     // Verschiebe A nach (2.3, -3.5)
  B.move(A);            // Verschiebe B nach A
  B.draw();             // Gib B aus
  double d = A.norm(); // Abstand von A zum Ursprung
}
```

- Das Beispiel wird noch nicht laufen, da Klassen sehr sorgsam mit den Zugriffsrechten auf ihre Elemente umgehen.
- Bei der momentanen Implementierung werden implizit Standardzugriffsrechte vergeben, die sehr restriktiv sind.
- Sprich: von außerhalb kann momentan niemand auf Klassenmember zugreifen: alle sind `private`.

Klassen (8): Zugriffsrechte in der Klasse

- Es gibt drei Zugriffsrechte: `private`, `protected`, `public`.
- `protected` besprechen wir später.
- Alle Elemente, die `public` deklariert sind, können von außen angesprochen werden.
- Wir machen nun das vorherige Beispiel lauffähig:

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

class Point2D
{
private:
  double M_x;
  double M_y;

public:
  void move(double x, double y);
  void move(Point2D p);
  void draw();
  double norm();
};
```

Klassen (9): Zugriff auf die Elemente

- Im Beispiel haben wir gesehen, wie auf die Elemente der Klasse zugegriffen wird.
- Innerhalb der Klasse durch einfaches Aufrufen der Funktion. Beispiel:

```
void Point2D::move(Point2D p)
{
  move(p.M_x, p.M_y);
}
```

- Von außerhalb mittels des Punktoperators auf die `public`-Member. Beispiel:

```
int main()
{
  ...
  B.move(A);
}
```

- Wir können also Hilfsfunktionen schreiben, die nur für den internen Gebrauch sinnvoll sind und diese `private` deklarieren. Konvention: private Membernamen beginnen mit dem Uglifier `M_` (Variablen und ggf. Funktionen).

Klassen (10): Zugriffsmethoden

- Es ist guter Stil, die Daten einer Klassen als `private` zu deklarieren, um unkontrollierte Zugriffe von außen zu verhindern.
- Wenn wir dennoch den Zugiff erlauben wollen, arbeiten wir in der Regel mit Zugriffsmethoden (auch Getter und Setter genannt).
- Wahlweise kann man nur Lesezugriff erlauben oder auch Schreibzugriff.
- Mit diesen Methoden kann auch sichergestellt werden, dass Änderungen an den Variablen die Klasse nicht in einen unsinnigen Zustand bringen.
- Erinnerung: Ist der Zugriff von außen erwünscht, `public` nicht vergessen!

Klassen (11): Read-only Methoden

- Klassenmethoden, die nur lesend auf Eigenschaften zugreifen, können diesbezüglich spezifiziert werden (Schlüsselwort `const`).
- Syntax:

```
Typ Methode(Parameter) const;
```

- Fast immer wird man get-Methoden als `const` deklarieren. Beispiel:

```
double get_y() const
{
  return M_y;
}
```

- Ebenso sollten wir Methoden, die nur zur Ausgabe von Informationen gedacht sind, als `const` deklarieren, konkret `Point2D::draw()` und `Point2D::norm()`.
- Ist eine Methode als `const` deklariert, so wird der Compiler alle Änderungen an den Klassenvariablen innerhalb dieser Methode als Fehler bemängeln.
- Dazu gehört auch der Aufruf von Funktionen, die nicht als `const` deklariert sind, da der Compiler hier nicht sichergehen kann, dass nichts verändert wird!

Klassen (12): Konstruktoren

- Bisher haben wir noch nicht die Initialisierung von Variablen besprochen.
- Bis zum Aufruf der Klassenmethode `move(double x, double y)` ist noch keine Variable initialisiert: `x` und `y` haben dann zufällige Werte!
- Für unseren Punkt wäre es aber sinnvoll, die Koordinaten bei der Erzeugung des Objekts festlegen zu können.
- C++ bietet hierzu einen speziellen Mechanismus, sodass ein Objekt beim Anlegen mit wichtigen Initialisierungen ausgestattet wird.
- Dieser Mechanismus wird Konstruktor genannt.
- Konstruktoren unterscheiden sich von anderen Klassenmethoden durch drei Aspekte:
  - Der Name des Konstruktors ist identisch zu dem der Klasse.
  - Ein Konstruktor besitzt keinen Rückgabetyp (auch nicht `void`!).
  - Ein Konstruktor kann nur zur Erzeugung eines Objekts verwendet werden.

Klassen (13.1): Fortsetzung des Beispiels

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

class Point2D
{
private:
  double M_x;
  double M_y;

public:
  Point2D(double x, double y);   // Neu: Konstruktor
  void move(double x, double y);
  void move(Point2D p);
  void draw() const;
  double norm() const;
};

Point2D::Point2D(double x, double y) // Neu: Konstruktor
{
  M_x = x;
  M_y = y;
}
```

Klassen (13.2): Fortsetzung des Beispiels

```
// Methoden wie oben (nur const bei draw und norm nicht vergessen!)
...
int main()
{
//Point2D A;            // Funktioniert so 'auf einmal' nicht mehr!
  Point2D A(0,0);
  Point2D B(1,1);

  A.move(2.3,-3.5);     // Verschiebe A nach (2.3, -3.5)
  B.move(A);            // Verschiebe B nach A
  B.draw();             // Gebe B aus.
  double d = A.norm(); // Abstand von A zum Ursprung
}
```

Frage

Warum funktioniert `Point2D A;` ohne Übergabe von Parametern nicht mehr, obwohl es vorher noch ging?

Klassen (14): Konstuktoren überladen

- `Point2D A;` funktioniert nur, wenn es einen sogenannten Default-Konstruktor gibt, also einen Konstruktor ohne Parameter.
- Dieser wird nur automatisch erzeugt, wenn wir keine Konstuktoren definieren.
- Dann hat unser Objekt aber einen nicht definierten Zustand. Unser Punkt könnte also irgendwo im Raum liegen. Vor allem würde der Punkt bei jeder Ausführung irgendwo anders liegen.
- Wir haben 2 Optionen, um einen Default-Konstruktor zu schreiben:
  1. Wir überladen den Konstruktor und definieren einen weiteren Konstuktor ohne Eingabewerte:

     ```
     Point2D() { M_x = M_y = 0.0; }
     ```

  2. Wir statten unseren bestehenden Konstuktor mit Defaultparametern aus:

     ```
     Point2D(double x = 0.0, double y = 0.0) { ... }
     ```

- In beiden Fällen würden wir beim Aufruf von `Point2D C;` nun den Nullpunkt erhalten. Vorteil an Variante 2 ist, dass wir nicht zu viel doppelten Code haben.
- Es ist aber nicht immer sinnvoll, für jede Member-Variable einen Default-Wert zu haben. Ein Default-Konstuktor ist daher auch kein Muss!

Klassen (15): Destruktoren

- Gegenstück zum Konstruktor. (Es darf aber nur einen Destruktor geben!)
- Wird beim Zerstören des Objekts aufgerufen.
- Hauptsächlich zum Freigeben von dynamisch reserviertem Speicherplatz. (Aufruf von `delete` oder `delete[]`.)
- Syntax:

```
~Klassenname(); // Kein Rückgabetyp und keine Parameter!
```

- Beispiel:

```
class Point2D
{
public:
  ~Point2D(); // Deklariert in public domain der Klasse
  ...
};

// Destruktor definieren (er tut nichts: Ausgabe nur zum Testen)
Point2D::~Point2D()
{
  std::cout << "Aufruf des Destruktors." << std::endl;
}
```

Klassen (16.1): Copy-Konstruktor

- Wir haben den Default-Konstruktor kennen gelernt.
- Der Copy-Konstruktor ist eine Methode, die ein Objekt als Kopie eines bereits existierenden Objekt initialisiert.
- Syntax (immer mit call-by-reference!):

```
ClassName(ClassName const& old_obj)
{
  // Anlegen von Variablen, etc.
}
```

- Gründe zum Definieren eines Copy-Konstruktors sind:
  - Initialisierung eines neuen Objekts von einem bereits existierenden;
  - Kopieren eines Objekts, um dieses als Argument an eine Funktion zu übergeben;
  - Kopieren eines Objekts, um es von einer Funktion zurückgeben zu lassen.
- Wenn nicht explizit ein Copy-Konstruktor definiert wird, dann erzeugt ihn der Compiler implizit. Dabei werden dann die Werte aller Variablen rigoros in das neue Objekt kopiert.
- In unserem Fall wäre das kein Problem, da wir nur primitive Datentypen verwendet haben.
- Im Fall von Pointer-Variablen und dynamisch angelegtem Speicher, muss aber ein Copy-Konstruktor angelegt werden. Ansonsten hätte jede Änderung der Kopie eine Änderung des Originals zur Folge, was definitiv nicht erwünscht ist!

Klassen (16.2): Copy-Konstruktor

Beispiel 51. Copy-Konstruktor für unsere Punktklasse:

```
#include <iostream>
#include <cmath>
class Point2D
{
private:
  double M_x, M_y;
public:
  Point(double x = 0.0, double y = 0.0) { M_x = x; M_y = y; }

  // Neu: Copy-Konstruktor
  Point2d(Point2D const& p) { M_x = p.M_x; M_y = p.M_y; }

  // Weitere Methoden wie zuvor ...
};
... // Methodendefinitionen wie zuvor
int main()
{
  Point2D D(A); // Kopie von A; alternativ '... D = A', '... D{A}'
}
```

Ende VL 8

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 8 durch.

Klassen (17.1): Klassen als Membervariablen

- Wir wollen uns im Folgenden anschauen, wie man mit Klassen als Member von anderen Klassen umgeht.
- Dazu werden wir uns eine Linie anschauen, die aus zwei Punkten besteht.

Beispiel 52. Linienklasse in 2D

```
class Line2D
{
//private: (zu Beginn optional, da Standard)
  Point2D M_p1, M_p2;
public:
  Line2D(Point2D p1, Point2D p2)
  {
    M_p1.move(p1);
    M_p2.move(p2);
  }
};
```

- Problem: Konstruktor `Line2D(...)` erzeugt M_p1, M_p2 mit Default-Konstruktor, kopiert p1, p2 und ersetzt dann M_p1, M_p2 mit `move(...)`.
- Das funktioniert zwar, ist aber sehr umständlich und schlechter Stil!
- Größeres Problem: Was machen wir, wenn die Member gar keinen Default-Konstruktor haben?!

Klassen (17.2): Klassen als Membervariablen

- Man darf (und sollte!) Member-Konstruktoren explizit aufrufen:
- Syntax:

```
ClassName(...) :
  MemberConstructor1(...),
  MemberConstructor2(...), ...
{
  // Hier muss dann oft nichts mehr hin!
}
```

- Der Doppelpunkt leitet den Aufruf der Member-Konstruktoren ein.
- Der Member-Konstruktor muss mit Variablennamen und passenden Parametern aufgerufen werden.
- Die einzelnen Konstruktoren werden mit Kommata voneinander getrennt.
- Bei Trennung von Deklaration und Definition muss dieser Part zur Definition.
- Heißt für unsere Linie:

```
Line2D::Line2D(Point2D p1, Point2D p2); // Deklaration
...
Line2D::Line2D(Point2D p1, Point2D p2) : M_p1(p1), M_p2(p2) {}
```

Noch besser: `p1` und `p2` als `Point2D const&` übergeben!

Klassen (17.2): Klassen als Membervariablen

- Diese Member-Initialisierung funktioniert auch mit Basis-Datentypen:

```
Point2D::Point2D(double x, double y) : M_x(x), M_y(y) {}
```

- Die Zuweisung innerhalb der geschweiften Klammern ist weiterhin möglich, aber schlechter Stil und obendrein umständlicher:

```
Point2D::Point2D(double x,double y)
{
  M_x = x;
  M_y = y;
}
```

Klassen (18.1): dynamischer Speicher bei Membervariablen

- In C++ gibt es die wichtige Rule-of-Three. Diese soll verhindern, dass es bei Klassen mit dynamischen (C) Speicher zu einem Zugriffsfehler kommt.
- Wenn eine der drei folgenden Methoden implementiert werden muss, dann auch die anderen beiden:
  - Destruktor;
  - Copy-Konstruktor;
  - (Copy-)Zuweisungsoperator (später).
- Wir wollen uns das anhand einer Polynomklasse einmal kurz anschauen.

Beispiel 53. Polynomklasse mit dynamischem (C) Speicher:

```
class Polynom
{
private:
  int M_degree;
  double* M_coeff;
public:
  Polynom(int n);
  Polynom(Polynom const& p);
  ~Polynom();
};
...
```

Klassen (18.2): dynamischer Speicher bei Membervariablen

Beispiel 53. (Forts.)

```
Polynom::Polynom(int n)
{
  M_degree = n;
  M_coeff = new double[n+1];
  for (int i = 0; i <= n; ++i)
    M_coeff[i] = 0.0;
}
Polynom::Polynom(Polynom const& p)
{
  M_degree = p.M_degree;
  M_coeff = new double[M_degree+1];
  for (int i = 0; i <= M_degree; ++i)
    M_coeff[i] = p.M_coeff[i];
}
// Hier brauchen wir den Destruktor!
Polynom::~Polynom() { delete[] M_coeff; }

int main()
{
  Polynom p(2);
}
```

Klassen (18.3): dynamischer Speicher bei Membervariablen

- Wir wollen uns nun einmal anschauen, wie die automatisch von C++ generierten Konstruktoren und Destruktoren bei unserem Punkt und unserem Polynom aussehen würden, wenn wir diese nicht selbst implementiert hätten.
- Compiler-generierter Default-Konstruktor (nur wenn gar kein Konstuktor implementiert wurde):

```
// Ok, aber Vorsicht: Variablen sind nicht initialisiert!
Point2D() : M_x(), M_y() {}
// Nicht ok: Speicherzugriffsfehler!
Polynom() : M_degree(), M_coeff() {}
```

- Compiler-generierter Copy-Konstruktor

```
// Ok: entspricht eigener Implementierung.
Point2D(Point2D const& p) : M_x(p.M_x), M_y(p.M_y) {}
// Nicht ok: Kopie und Original haben dasselbe Koeffizientenfeld!!
Polynom(Polynom const& p) :
  M_degree(p.M_degree), M_coeff(p.M_coeff) {}
```

- Compiler-generierter Destruktor

```
~Point2D() {} // Ok: entspricht eigener Implementierung.
~Polynom() {} // Nicht ok: Speicher wird nicht freigegeben!
```

Strukturierung (1): Ein Programm organisieren

- Wir sehen, dass unsere Codes an Länge zunehmen.
- Irgendwann wird die Darstellung in einer Datei zu unübersichtlich.
- Insbesondere wird es Programmteile geben, die sozusagen allgemein gültig sind und ggf. in anderen Projekten wiederverwendbar sind.
- Hier ist es sinnvoll, diese in andere Dateien auszulagern.
- Gerade bei größeren Programmierprojekten sollten wir das Programm in mehrere Module aufteilen.
- Es vereinfacht auch die Arbeit, wenn mit mehreren Personen am selben Code gearbeitet wird.
- In der Regel werden die Deklarationen in Header-Dateien ausgelagert und die Definitionen in Source-Dateien.
- Die Header-Dateien enden mit `.hh` und die Source-Dateien wie gehabt mit `.cc`.
- Allerdings müssen die Dateien voneinander wissen!
- Im Folgenden spalten wir den Code des 2D-Punktes in drei Teile auf.

Strukturierung (2): Aufspaltung der Punktklasse

- Vorgehen: lege neues Verzeichnis an. z.B. `point_folder`.
- Lege in dieses Verzeichnis das bisherige cc-file; bei mir heißt diese Datei `point_with_default_constr.cc`.
- Diese enthält (ohne Getter und Setter) alle Änderungen bis zum Standardkonstruktor.
- Kompiliere und führe aus, sodass wir sicherstellen, dass der bisherige Code auch im neuen Verzeichnis läuft.
- Spalte den Code in drei Teile auf:

```
point2d.hh  // Hier wird alles deklariert (Headerdatei)
point2d.cc  // Hier wird alles definiert  (Quelldatei, source file)
point.cc    // Hier wird die main-Funktion implementiert
```

Strukturierung (3): point2d.hh

```
// Die drei Zeilen mit '#' sind Präprozessor-Direktiven für
// bedingte Kompilierung. Sie sind nötig, sobald die Klasse
// in mehreren anderen Dateien verwendet wird!
// Ansonsten werden dieselben Deklarationen mehrfach inkludiert.
// (Oder sogar Definitionen mehrfach inkludiert: Fehler!)
// Alternativ (neuer, nicht-Standard): '#pragma once' zu Beginn

#ifndef POINT2D_H
#define POINT2D_H 1
class Point2D
{
private:
  double M_x;
  double M_y;
public:
  Point2D(double x = 0.0, double y = 0.0);
  void move(double x, double y);
  void move(Point2D p);
  void draw() const;
  double norm() const;
}; // Semikolon nicht vergessen!
#endif
```

Strukturierung (4): point2d.cc

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

// Dieses include ist neu: es stellt die Deklarationen aus der
// Header-Datei für die Definitionen in dieser Source-Datei bereit.
#include "point2d.hh"
// Analog könnten wir weitere Klassen bekannt machen, die verwendet
// werden sollen. Z.B. kann eine Viereckklasse mit dem obigen include
// auf unseren Punkt zugreifen.

Point2D::Point2D(double x, double y) : M_x(x), M_y(y) {}

...
```

Strukturierung (5): point2d.cc (Fortsetzung)

```cpp
// Definition unserer Methoden.

void Point2D::move(double x, double y)
{
  M_x = x;
  M_y = y;
}

void Point2D::move(Point2D p) { move(p.M_x, p.M_y); }

void Point2D::draw() const
{
  std::cout << '(' << M_x << ", " << M_y << ")\n";
}

double Point2D::norm() const
{
  return std::sqrt(M_x*M_x + M_y*M_y);
}
```

Strukturierung (6): point.cc

- In point.cc muss nur das header-file bekannt gemacht werden.
- Für die Quelldatei muss intern nur die kompilierte Version zur Verfügung stehen.
- Bei richtigem Aufruf verknüpft der Compiler das aber alles korrekt.

```
#include <iostream>
#include "point2d.hh"

int main()
{
  Point2D A(1.0,1.0);
  Point2D B(2.0,3.0);
  Point2D C; // Nullpunkt
  A.move(2.3,-3.5);     // Verschiebe A nach (2.3, -3.5)
  B.move(A);            // Verschiebe B nach A
  B.draw();             // Gib B aus
  C.draw();
  double d = A.norm(); // Abstand von A zum Ursprung
  std::cout << "Distance from A to (0,0) is: " << d << std::endl;
} // main
```

Bemerkung

Weitere Infos auf `https://www.cplusplus.com/forum/articles/10627/`

Wie kompilieren wir nun?

- Bei mehreren cc-Dateien müssen alle mit-kompiliert werden!
- Syntax für das Beispiel (wird a.out liefern):

```
g++ point.cc point2d.cc
```

- Besser:

```
g++ -Wall point.cc point2d.cc -o point_example; ./point_example
```

Makefiles (1): Kompakte Kompilierung

- Wenn Sie nun viele cc-Dateien haben, müssen Sie immer wieder alles neu tippen. Natürlich können Sie im Konsolenverlauf nach den letzten Befehlen suchen. Beides ist aber umständlich.
- Zu diesem Zweck werden sogenannte Makefiles angelegt.
- Diese Datei liegt im selben Verzeichnis wie Ihre cc-Dateien und hat den schlichten Dateinamen `Makefile`.

```
prof@luh> emacs Makefile
```

Makefiles (2)

- Hierin können Sie Ihre Kompilierbefehle bündeln:

```
all:
	g++ -Wall point.cc point2d.cc -o point_example
	./point_example
```

- Der Aufruf erfolgt dann ganz einfach über

```
make
```

in der Konsole bzw.

```
make all
```

um das Make-Target ‘all’ anzusteuern.

Vorsicht!

Beim Makefile müssen Einrückungen mithilfe der Tabulator-Taste geschehen, ansonsten wirft der Befehl `make` Fehler! Wie oben zu sehen, entspricht TAB in der Regel 8 Leerzeichen.

Makefiles (3)

Vorsicht!

Wir haben nun viele Programme kompiliert und ggf. verschiedene Objektdateien. Es ist daher nicht mehr ganz klar, wann wir welche Datei kompiliert haben und welcher Datei point.o oder a.out zugeordnet sind.

- Häufiger Fehler: Wir führen Änderungen in einer Datei durch, kompilieren aber aus Versehen eine andere Datei und wundern uns, dass das Programm noch die 'alten' Ergebnisse liefert.
- Daher ist es klug, von Zeit zu Zeit alle Objektdateien zu entfernen.
- Hierzu kann das Makefile einfach erweitert werden:

```
all:
        g++ -Wall point.cc point2d.cc -o point_example
        ./point_example
clean:
        rm *.o
        @echo "Removed object files"
```

- Aufruf in der Konsole (Terminal):

```
make clean
```

Makefiles (4)

- Bisher haben wir einfach unsere Befehle in make alle 'hintereinander weg' geschrieben.
- Das funktioniert bei kleineren Projekten noch ganz gut, es kann aber vorkommen, dass make unveränderte Dateien unnötigerweise erneut kompiliert.
- Daher eine bessere (aber optionale) Erweiterung.
- Hierbei teilen wir make mit, was wir selbst über die Struktur unseres Programmes wissen.
- Vor dem Doppelpunkt steht unser Befehl (all, clean).
- Direkt dahinter können wir aber Abhängigkeiten definieren und sagen, welche Dateien für die Ausführung dieses Befehls notwendig sind.
- Zusätzlich können wir dem Compiler sagen, dass er Quelldateien 'nur' kompilieren soll, aber nicht linken, indem wir die Option '-c' mit angeben.
- g++ wird dann für die angegebene Quelldatei (.cc) eine Objektdatei (.o) gleichen Namens erstellen.
- Make wird dann nur für die Quell- und Headerdateien neue Objektdateien erstellen, die wir auch wirklich verändert haben.

Makefiles (5)

Beispiel 54. Erweitertes Makefile für unsere Punktklasse:

```
all: point_example
        ./point_example
point_example: point2d.o point.o
        g++ *.o -o point_example
point.o: point.cc point2d.hh
        g++ -c -Wall point.cc
point2d.o: point2d.cc point2d.hh
        g++ -c -Wall point2d.cc
clean:
        rm *.o
        @echo "Removed object files"
```

Mit `make` oder `make all` werden jetzt point.o bzw. point2d.o nur dann neu kompiliert, wenn point2d.hh oder point.cc bzw. point2d.cc verändert wurde. Nur dann wird auch point_example neu gelinkt.

Aufgabe

Kompilieren Sie alle Dateien mit dem erweiterten Makefile.
Warten Sie ein paar Minuten und ändern dann etwas in der main-Methode.
Kompilieren Sie erneut und schauen, welche Dateien einen älteren Zeitstempel haben (`ls -l` zeigt alle Dateien als Liste mit Zeitstempel an).

Ende VL 9

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 9 durch.

Überladen von Operatoren (1)

- Wiederholung Operatoren: z.B.

```
= // Zuweisung
+ // Addition
- // Subtraktion
* // Multiplikation
/ // Division
```

- Für weitere Operatoren siehe:
  `https://www.cplusplus.com/doc/tutorial/operators/`
- Diese sind standardmäßig für die Basisdatentypen (int, double, etc.) definiert.
- Es gibt gute Gründe, Operatoren überladen zu können.
- Beispiele: Operatoren für Matrizen, Vektoren oder auch Punkte im $\mathbb{R}^n$.
- Eine Matrix-Klasse lässt sich dann z.B. wie folgt benutzen:

```
Matrix A, B;
Matrix C = A + B; // Elementweise Addition, wie aus LA bekannt.
```

- In C++ lassen sich fast alle vorhandenen Operatoren überladen, was in vielen anderen Sprachen nicht möglich ist (z.B. Java).

Überladen von Operatoren (2)

- Wir wollen uns das direkt einmal für unsere Punktklasse anschauen.
- Es gibt grundsätzlich zwei Arten, Operatoren zu überladen:
  1. Beide Operanden werden als Parameter übergeben. Dabei ist die Operatorfunktion eine globale bzw. externe Funktion, wir benötigen also Zugriff auf `private` Member!

  $\Rightarrow$ Dies ermöglicht `friend`: eine externe Funktion darf auf alle Member zugreifen, falls sie innerhalb der Klasse als `friend` deklariert ist.

  ```
  A+B entspricht dann dem Funktionsaufruf operator+(A,B)
  ```

  2. Das aufrufende Objekt selbst ist der erste Operand, der zweite wird als Parameter übergeben. Dadurch erhalten wir eine Klassenmethode, der Zugriff auf `private` Member ist also von vornherein gestattet:

  ```
  A+B entspricht dann dem Funktionsaufruf A.operator+(B)
  ```

- Für die eigentliche Benutzung ist es egal, für welche Variante wir uns entscheiden. Es gibt aber einige Operatoren die nur mit je einer der beiden Varianten funktionieren.
- Im folgenden betrachten wir beide Fälle.

Überladen von Operatoren (3)

Beispiel 55. Überladen von Addition und Subtraktion bei der Punktklasse:

```
#include <iostream>
class Point2D
{
private:
  double M_x;
  double M_y;
public:
  Point2D(double x = 0.0, double y = 0.0);
  ... // Deklaration move, draw usw.
  // Überladen des Additions-Operators mit Variante 1:
  friend Point2D const operator+(Point2D const& a, Point2D const& b);
  // Überladen des Subtraktions-Operators mit Variante 2:
  Point2D const operator-(Point2D const& b) const;
}; ...
```

- So viel `const`! Also der Reihe nach ...
- `Point2D const` als Rückgabewert verbietet den unsinnigen Ausdruck `(A+B) = C`!
- Die Parameter a, b sollen natürlich nicht verändert werden, deswegen Übergabe als konstante Referenz.
- Aus demselben Grund ist Variante 2 `const`: dies verhindert, dass `A` bei `A-B` verändert wird!

Überladen von Operatoren (3)

Beispiel 55. (Forts.) Definition der Operatoren:

```
... // Definition move, draw, usw.
Point2D const operator+(Point2D const& a, Point2D const& b)
{
  Point2D p;
  p.M_x = a.M_x + b.M_x; // Zugriff auf die Member von a
  p.M_y = a.M_y + b.M_y; // und b, möglich dank friend!
  return p;
}
Point2D const Point2D::operator-(Point2D const& b) const
{
  Point2D p;
  p.M_x = M_x - b.M_x; // Zugriff immer möglich, da Klassenmethode.
  p.M_y = M_y - b.M_y; // Aufrufer ist erster Operand, darum direkter
  return p;            // Zugriff auf die Member M_x und M_y.
}
int main()
{
  Point2D A(1.0,1.0), B(1.0,2.0);
  Point2D C = A+B;
  Point2D D = C-B;
  C.draw(); D.draw(); // Hoffentlich D == A!
}
```

Überladen von Operatoren (4)

Im Beispiel hatten wir zur Illustration erst einen neuen Punkt erzeugt, dann Summe oder Differenz gebildet und zuletzt den Punkt zurückgegeben. Kürzer, klarer und effizienter ist alles in Einem:

```
Point2D const operator+(Point2D const& a, Point2D const& b)
{
  return Point2D{a.M_x + b.M_x, a.M_y + b.M_y};
}
```

Sogar den expliziten Konstruktoraufruf im `return`-Befehl dürfen wir weglassen. Der Compiler weiß ja aus der Signatur, was er konstruieren muss:

```
Point2D const Point2D::operator-(Point2D const& b) const
{
  return {M_x - b.M_x, M_y - b.M_y};
}
```

Wichtig: hier müssen die Konstruktorparameter in geschweiften Klammern stehen! Im `operator+` wäre dagegen auch `return Point2D(...)` erlaubt.

Überladen von Operatoren (5) Zuweisungsoperator

- Nun kommen wir zum dritten Vertreter der Rule-of-Three, dem Copy-Zuweisungsoperator: `A=B`.
- Für unseren Punkt würde die Implementierung z.B. so aussehen:

```
... public:
Point2D& operator=(Point2D const& p)
{
  M_x = p.M_x;
  M_y = p.M_y;
  return *this; // 'this' = Zeiger auf das aktuelle Objekt
}
```

- Dies ist einer der Operatoren, die nur als Methode implementiert werden können.
- Wir können auch andere (sinnvolle) Parameter wählen, z.B. `double x` für $p = (x, x)$. Dann spricht man nur noch vom Zuweisungsoperator.

Vorsicht!

Der Copy-Konstruktor und der Copy-Zuweisungsoperator sind sich zum Verwechseln ähnlich (auch bei der Implementierung), aber doch grundlegend verschieden.
`Point2D B = A` ruft den Copy-Konstruktor auf, da B noch nicht existiert.
`B = C` ruft hingegen den Zuweisungsoperator auf: B muss an dieser Stelle bereits konstruiert sein.

Überladen von Operatoren (6): Copy-Zuweisung mit dyn. Speicher

- Bei unserem Polynom haben wir zwei Optionen für die Zuweisung eines Polynoms anderen Grades.
- Wir können den Speicher für die Koeffizienten freigeben und neu allokieren (aufwändig, auch später in der Ausführung):

```
Polynom& operator=(Polynom const& p) {
  if (p.M_degree != M_degree) {
    delete[] M_coeff;
    M_degree = p.M_degree;
    M_coeff = new double[M_degree + 1];
  }
  for (int i = 0; i <= M_degree; ++i) M_coeff[i] = p.M_coeff[i];
  return *this;
}
```

- Oder wir erlauben das schlicht nicht und terminieren das Programm mit einem Fehler:

```
...
if (p.M_degree != M_degree)
  exit(EXIT_FAILURE);
...
```

- Welche Variante „besser“ ist, hängt vom Einsatzzweck der Polynomklasse ab!

Überladen von Operatoren (7)

- Wir haben jetzt einen Operator gesehen, der sich nur als Member-Operator implementieren lässt.
- Wofür brauchen wir denn nun aber die Variante als globalen/externen Operator?
- Wir könnten unseren Punkt um eine feste Zahl x (`double`) in beide Richtungen verschieben wollen, also $(a, b) + x = (a + x, b + x) = x + (a, b)$. Für die zweite Gleichung müssten wir aber einen Member-Operator von `double` schreiben.
- Da das schlicht und ergreifend nicht möglich ist, können wir stattdessen
  `friend Point2D const operator+(double a, Point2D const& b);`
  deklarieren und dann außerhalb der Klasse wie gewünscht implementieren.

Vorsicht!

Wir haben bisher nur definiert, was für `double+Point2D` passieren soll. Da die Operation im Zweifel nicht symmetrisch ist, wird C++ hier nicht einfach die Parameter vertauschen.
Stattdessen wird `A+1.0` in `A+Point2D(1.0)` umgewandelt, da für `Point2D+Point2D` eine Addition definiert ist und ein Konstruktor mit nur einem `double` Parameter existiert (ein sogenannter konvertierender Konstruktor).
Statt also wie gewünscht $(1, 1)$ zu addieren, wird $(1, 0)$ addiert ... Solche unerwünschten impliziten Konversionen verhindert man mit
`explicit Point2D(double x = 0, double y = 0).`

Überladen von Operatoren (8)

- Von unseren einfachen Zahlen kennen wir ja zusätzlich den `operator+=`, dieser ist auch ein Zuweisungsoperator und kann deshalb nur als Member implementiert werden:

```
Point2D& operator+=(Point2D const& b)
{
  M_x += b.M_x;
  M_y += b.M_y;
  return *this;
}
```

- Good Practice: Definition von Copy-Konstruktor und `operator+=`, um damit die externe Addition zu implementieren:

```
Point2D const operator+(Point2D const& p, Point2D const& q)
{
  Point2D r{p}; // Benutze Copy-Konstruktor
  r += q;       // Benutze operator+=
  return r;
  // return Point2D{p} += q; // Besser wieder alles in Einem.
}
```

- Damit ist sichergestellt, dass Addition und Addition mit Zuweisung konsistent implementiert sind, außerdem ist `friend` unnötig!

Überladen von Operatoren (9)

- Ein- und Ausgabeoperator können ebenfalls nicht als Elementfunktionen implementiert werden, weil das Stream-Objekt der erste Operand ist.
- Wir deklarieren also innerhalb unserer Klasse (<iostream> nicht vergessen!):

```
friend std::ostream& operator<<(std::ostream&, Point2D const&);
friend std::istream& operator>>(std::istream&, Point2D&);
```

- Definition der beiden Methoden außerhalb:

```
std::ostream& operator<<(std::ostream& ostr, Point2D const& p)
{
  ostr <<  '(' << p.M_x << ", " << p.M_y << ')';
  return ostr;
}
std::istream& operator>>(std::istream& istr, Point2D& p)
{
  // Lies x, y und verändere p nur im Erfolgsfall.
  double x, y;
  istr >> x >> y;
  if (istr.good())
    p.move(x, y);
  return istr;
}
```

Überladen von Operatoren (10)

Erweiterung der main-Funktion zum Testen:

```
int main()
{
  ...
  Point2D K, L;
  std::cout << "2 points that should be added to each other\n";
  std::cin >> K >> L;
  std::cout << "Result: " << K
            << " + " << L << " = "
            << K+L << std::endl;
}
```

Überladen von Operatoren (11): Richtlinien

- Der Operator sollte (mathematisch) Sinn ergeben (bspw. `operator+=` nur für Datentyp mit additiver Gruppenstruktur).
- Überladene Operatoren können entweder als Member-Methode oder als externe Funktion definiert werden.
- Nur als Member:
  - Zuweisungsoperator `=`
  - kombinierte Zuweisungsoperatoren `+=` etc.
  - Indexoperator `[]`
  - Aufrufoperator `()`
- Wird ein Objekt der Klasse zurückgegeben, sollte es `const` sein.
- Falls binäre Operatoren unterschiedlich auf linken und rechten Operanden wirken:

⇒ Implementierung als Member-Operator (z.B. soll `1.4 += p` für `Point2D p` nicht erlaubt sein).

- Falls binäre Operatoren nicht zwischen linkem und rechtem Operator unterscheiden:

⇒ Implementierung als externe Funktion (`1.4 + p` soll erlaubt sein).

! Für Symmetrie mit unterschiedlichen Parametertypen müssen beide Versionen implementiert werden.

Überladen von Operatoren (12)

Finale Bemerkungen:

- Weiterer wichtiger Operator ist die Negation `operator-()`, die bei den einfachen Datentypen das Vorzeichen ändert und unär ist.
- Hier die komplette Liste überladungsfähiger Operatoren kopiert aus `https://www.cplusplus.com/doc/tutorial/templates/`

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

- Dort sind auch noch ein paar weitere Beispiele zu finden.
- Überladene Operatoren können die spätere Benutzung unserer Klassen stark vereinfachen.

Ende VL 10

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 10 durch.

Motivation

- Vererbung fügt einer Klasse neue Funktionalität hinzu.
- Bereits existierende Klassen können in neuen Klassen verwendet werden. (Auch ohne die genaue Implementierung zu kennen.)
- Sehr effizienter und häufig eingesetzter Mechanismus in C++.
- Eine abgeleitete Klasse erbt die `public`-Methoden der Basisklasse.
- Eine Basisklasse kann selbst bereits abgeleitet sein.
- Wir kennen das analog schon von den Iteratoren: deren Hierarchie lässt sich als Vererbung interpretieren. Ein bidirektionaler Iterator erweitert die Funktionalität eines Vorwärtsiterators.
- Insbesondere kann jeder Algorithmus für einen Vorwärtsiterator auch mit einem bidirektionalen Iterator durchgeführt werden.
- Syntax:

```
class Neuer_Klassenname : Zugriffsrechte Basisklasse {...};
```

- Mit einer Basisklasse lassen sich Gemeinsamkeiten abbilden.
- Beispiel: Kreise, Dreiecke und Vierecke sind (2D-)Formen mit einer Fläche.
- Abgeleitete Klassen können selbst Basisklassen sein: Rechtecke sind Vierecke usw.
- Klassen können in C++ auch mehrere Basisklassen haben (kompliziert).

Vererbung (1)

Beispiel 56. Beispiel zur Vererbung (leicht angepasst von `https://www.cplusplus.com/doc/tutorial/inheritance/`):

```
#include <iostream>

class Polygon
{
public:
  void set_width (int w) { width  = w; }
  void set_height(int h) { height = h; }

// Protected data is visible in inheritated classes,
// but not from outside
protected:
  int width, height;
};
...
```

- Unsere Basisklasse ist somit definiert. (Konstruktor später.)
- Das Zugriffsrecht-Schlüsselwort `protected` kommt jetzt zum Einsatz.
- Gegenüber externem Zugriff verhält es sich wie `private`, aber abgeleitete Klassen dürfen auf `protected` zugreifen.

Vererbung (2)

Beispiel 56. (Forts.)

```
class Rectangle : public Polygon
{
public:
  int area() const { return width * height; }
};

class Triangle : public Polygon
{
public:
  int area() const { return width * height / 2; }
};

int main()
{
  Rectangle rect;
  Triangle trig;
  rect.set_width(4);
  rect.set_height(5);
  trig.set_width(4);
  trig.set_height(5);
  std::cout << rect.area() << '\n' << trig.area() << '\n';
}
```

Vererbung (3)

- Wir wollen unserem Beispiel noch ein Quadrat hinzufügen, dafür müssen aber Breite und Höhe identisch sein. Wir können dafür `set_width` (und `set_height`) so anpassen, dass immer beide Werte geändert werden.

```
class Square : public Rectangle
{
public:
  void set_width (int a) { width = height = a; }
  void set_height(int a) { set_width(a); }
}
```

- Die Methoden von Square dominieren also die Methoden von Rectangle, wie bei lokalen Blöcken.
- Mit Membervariablen funktioniert das genauso, ist aber fast nie sinnvoll.
- Methoden und Variablen der Basisklasse werden hier nicht überladen, sondern überlagert: sie können über `Basisklasse::Methode()` bzw. `Basisklasse::Variable` weiter benutzt werden. Ein (unsinniges!) Beispiel:

```
Square s; s.Rectangle::set_width(5);
```

In `Square` wäre es besser, die geerbten Setter `set_width()` und `set_height()` als `private` zu deklarieren und durch einen neuen Setter `set_side()` zu ersetzen, wieder mit `width = height = a`.

Vererbung (4)

- Was wird nun eigentlich alles vererbt?
- Bei einer `public` abgeleiteten Klasse wird alles vererbt bis auf:
  - die Konstrukturen und den Destruktor;
  - Zuweisungsoperatoren (operator=);
  - friend-Klassen;
  - private members.
- Bei den Konstruktoren und Destruktoren sollte man wissen, dass diese zwar nicht direkt vererbt werden, aber trotzdem durch die abgeleitete Klasse aufgerufen werden.
- Die Konstruktoren der abgeleiteten Klasse bestimmen, welche Konstruktoren der Basisklasse (explizit oder implizit) aufgerufen werden.
- Bei einer `protected` abgeleiteten Klasse funktioniert alles genauso, aber die public Members der Basisklasse werden protected.
- Bei einer `private` abgeleiteten Klasse funktioniert weiter alles genauso, aber alle Members der Basisklasse werden private.

Vererbung (5)

Beispiel 57. Beispiel (siehe https://www.cplusplus.com/doc/tutorial/inheritance/):

```
// Constructors and derived classes
#include <iostream>
using namespace std;

class Mother
{
public:
  Mother() { cout << "Mother: no parameters\n"; }
  Mother(int a) { cout << "Mother: int parameter\n"; }
};

class Daughter : public Mother
{
public:
  Daughter(int a) { cout << "Daughter: int parameter\n\n"; }
};

...
```

Vererbung (6)

```
// Fortsetzung
class Son : public Mother
{
public:
  Son(int a) : Mother(a) { cout << "Son: int parameter\n\n"; }
};

int main()
{
  Daughter kelly(0);
  Son bud(0);
}
```

Ausgabe:

```
Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter
```

Vererbung (7): Polymorphie

- Die Funktion `int area()` hatten wir in jeder abgeleiteten Klasse definiert.
- Wir wollen jetzt für ein beliebiges Polygon diese Funktion aufrufen, z.B. mit

```
Square s;
s.set_height(42);
Polygon& p = s; // Konkreter Typ des Polygons wird verdeckt!
std::cout << p.area() << std::endl;
```

- Aber: für ein allgemeines Polygon existiert keine Formel, um aus Höhe und Breite die Fläche zu bestimmen, also können wir auch nichts implementieren!
- Den Ausweg bieten sogenannte abstrakte Klassen, das sind Klassen mit mindestens einer rein virtuellen Funktion.
- Diese werden mit dem Schlüsselwort `virtual` versehen und nicht implementiert, sondern mit einem $= 0$ am Ende versehen.
- In unserem Fall würde diese Funktion wie folgt aussehen

```
class Polygon
{
public:
  ...
  virtual int area() const = 0; // const immer vor = 0
}
```

Vererbung (8): Abstrakte Klassen

- Abstrakte Klassen dienen ausschließlich als Basisklassen.
- Man kann von ihnen keine Objekte bilden. Auch nicht als Funktionsargumente oder Rückgabewerte.
- Man kann aber Referenzen und Zeiger von abstrakten Klassen bilden, auch als Parameter und Rückgabewerte. Diese müssen immer auf Objekte abgeleiteter Klassen zeigen.
- Weiter lassen sich Arrays von Zeigern auf abstrakte Klassen bilden, auch mit gemischten abgeleiteten Typen. (Wichtig!)
- Konstruktoren können nicht virtuell sein: jede abgeleitete Klasse sollte einen eigenen Konstruktor haben, der typischerweise einen Konstruktor der Basisklasse aufruft (mit ':', wie bei Klassen als Member).
- Destruktoren sollten in der Basisklasse virtuell deklariert sein (sonst kann es zu Speicherzugriffsfehlern kommen), z.B. bei

```
Polygon* b = new Triangle;
...;
delete b;
```

Bemerkung

Weitere Infos, insbesondere zu nicht rein virtuellen Methoden sind z.B. auf `https://en.cppreference.com/w/cpp/language/virtual` zu finden.

Beispiel Nullstellensuche (1)

- Wir wollen nun wieder etwas Numerik machen und Nullstellen finden.
- Neben dem Newton-Verfahren gibt es in der Numerik noch weitere Methoden um die Nullstelle einer Funktion zu finden: Intervallschachtelung, regula falsi, Sekantenverfahren, Fixpunktverfahren.
- Etwas vereinfacht haben diese Verfahren folgende Gemeinsamkeiten:
  - sie alle müssen mindestens die gegebene Funktion kennen;
  - sie sollen die Iteration bei einer gewissen Toleranz abbrechen;
  - sie brauchen eine Funktion die die eigentliche Iteration durchführt;
  - sie sollen nach einer maximalen Anzahl an Iterationen abbrechen.
- Die Grundeigenschaften einer jeden iterativen Nullstellensuche können wir also in einer Basisklasse definieren und dann die speziellen Verfahren aus dieser Basisklasse ableiten.
- Wir werden dabei insbesondere unser bereits geschriebenes Newton-Verfahren in diese Struktur einbetten. Zur Erinnerung: die gegebene Funktion lautet

$$f(x) = x^2 - y, \quad y > 0.$$

- Wir benötigen aber zunächst eine abstrakte Basisklasse, die die allgemeinen Funktionalitäten darstellt.
- Sie implementiert alle Gemeinsamkeiten, insbesondere den Grundalgorithmus, der bis auf die Iteration $x_{k+1} = \cdots$ für alle Verfahren identisch ist.

Beispiel Nullstellensuche (2)

Beispiel 58. Headerdatei der abstrakten Basisklasse zur Nullstellensuche:

```
#pragma once
#include <string>

using Fct = double (*)(double x); // f-Auswertung

class RootFindingScheme // engl. root = Nullstelle
{
public:
  RootFindingScheme(Fct f, double tol = 1e-10, int max_it = 20);
  int solve(double x0); // Durchführung der Nullstellensuche
protected:
  virtual std::string scheme_name() const = 0; // Name des Verf.
  virtual void iterate() = 0; // Iterationsvorschrift

  Fct M_fct;     // Gegebene Funktion
  double M_tol; // Geforderte Genauigkeit
  int M_max_it; // Maximale #Iterationen
  int M_it;     // Aktueller Iterationsindex
  double M_x;   // Aktuelle  Iterierte
  double M_f;   // Aktueller Funktionswert
};
```

Beispiel Nullstellensuche (3)

Beispiel 58. (Forts.) Quelldatei der abstrakten Basisklasse (Teil 1 von 2):

```
#include "RootFindingScheme.hh"
#include <iostream>
#include <limits>
#include <cmath>

RootFindingScheme::RootFindingScheme(Fct f, double tol, int max_it) :
  M_fct(f),
  M_tol{std::max(16 * std::numeric_limits<double>::epsilon(), tol)},
  M_max_it{std::max(max_it, 0)}
  // M_x, M_it, M_f werden in solve() initialisiert.
{ }

...
```

Beispiel Nullstellensuche (4)

Beispiel 58. (Forts.) Quelldatei der abstrakten Basisklasse (Teil 2 von 2):

```
...

int
RootFindingScheme::solve(double x0)
{
  std::cout << "Starting " << scheme_name() << " with tolerance "
            << M_tol << "\nIter\tx\tf(x)\n"
            << "---------------------------\n";
  M_x = x0;
  for (M_it = 0; M_it <= M_max_it; ++M_it) {
    M_f = M_fct(M_x);
    std::cout << M_it << '\t' << M_x << '\t' << M_f << std::endl;
    if (std::abs(M_f) <= M_tol)
      break; // Lösung gefunden
    iterate();
  }
  return M_it; // # durchgeführte Iterationen, in 0 ... max_it+1.
}
```

Beispiel Nullstellensuche (5)

- Jetzt wird es Zeit, unser Verfahren genauer zu spezifizieren.
- Wir wollen eine Klasse `NewtonScheme` schreiben und die Iterationsvorschrift des Newton-Verfahrens implementieren:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- Wir müssen also zusätzlich $f'(x)$ auswerten (neuer Konstruktor-Parameter).

Beispiel 58. (Forts.) Headerdatei NewtonScheme.hh:

```
#pragma once
#include "RootFindingScheme.hh"

class NewtonScheme : public RootFindingScheme
{
public:
  NewtonScheme(Fct f, Fct df, double tol = 1e-10, int max_it = 20);
protected:
  std::string scheme_name() const; // Wird überschrieben:
  void iterate();                  // deklarieren (ohne virtual)!

  Fct M_df; // Neu: df-Auswertung
};
```

Beispiel Nullstellensuche (6)

Beispiel 58. (Forts.) Quelldatei NewtonScheme.cc:

```
#include "NewtonScheme.hh"

NewtonScheme::NewtonScheme(Fct f, Fct df, double tol, int max_it) :
  RootFindingScheme(f, tol, max_it), M_df(df) {}

std::string
NewtonScheme::scheme_name() const
{
  return "Newton Scheme";
}

void
NewtonScheme::iterate()
{
  M_x -= M_f / M_df(M_x);
}
```

- Jetzt müssen wir nur noch $f(x)$ und $f'(x)$ implementieren.
- Danach können wir unser Verfahren in der main-Methode aufrufen.

Beispiel Nullstellensuche (7)

Beispiel 58. (Forts.) Haupt-Quelldatei newton.cc für konkretes Problem $x = \sqrt{2}$:

```
#include "NewtonScheme.hh"

double  f(double x) { return x * x - 2; }
double df(double x) { return 2 * x; }

int main()
{
  NewtonScheme sqrt2(f, df);
  sqrt2.solve(1.5);
  // Auch das geht:
  NewtonScheme(f, df).solve(2.5);
}
```

Beispiel Nullstellensuche (8)

Beispiel 58. (Forts.) Zu guter Letzt noch das ausführliche Makefile:

```
newton: rfs_example
	./rfs_example

rfs_example: newton.o NewtonScheme.o RootFindingScheme.o
	g++ *.o -o rfs_example

newton.o: newton.cc NewtonScheme.hh
	g++ -c -Wall newton.cc

NewtonScheme.o: NewtonScheme.cc NewtonScheme.hh RootFindingScheme.hh
	g++ -c -Wall NewtonScheme.cc

RootFindingScheme.o: RootFindingScheme.cc RootFindingScheme.hh
	g++ -c -Wall RootFindingScheme.cc

clean:
	rm *.o
	@echo "Removed object files"
```

Beispiel Nullstellensuche (9)

Zusammenfassung

- Wir haben nun gesehen, wie man Vererbung dazu verwenden kann, eine Familie von numerischen Verfahren abzubilden.
- Wir könnten jetzt auf zwei Ebenen Änderungen vornehmen.
- Auf der Ebene von NewtonScheme könnten wir weitere Verfahren implementieren (z.B. Fixpunktverfahren, regula falsi).
- Auf der globalen Ebene könnten wir `f, df` ersetzen, um Nullstellen einer völlig anderen Funktion mithilfe des fertigen Newton-Verfahrens berechnen lassen.
- Wir können also hoffentlich erahnen, wie Vererbung es ermöglicht, mit etwas mehr anfänglichem Aufwand später einmal einiges an Aufwand zu sparen.

Beispiel Matrizen (kurz)

- Ein weiteres beliebtes Beispiel aus der Numerik sind verschiedene Arten von Matrizen. (Insbesondere die Besetzungsstruktur ist hier interessant.)
- Eines der wichtigsten iterativen Verfahren zur Lösung linearer Gleichungssysteme muss lediglich Matrix-Vektor-Multiplikation durchführen: das CG-Verfahren.
- Eine abstrakte Basisklasse `AbsMat` muss also nur eine (rein virtuelle) Methode enthalten, mit der eine Matrix-Vektor Multiplikation durchgeführt werden kann. Mögliche Beispiele wären (mit existierender Vektor-Klasse `Vec`):

```
// Option 1: A*x (mit Vec als Rückgabewert: sehr teuer!)
virtual Vec AbsMat::vmult(Vec const& x) const = 0;      // 1a
virtual Vec AbsMat::operator*(Vec const& x) const = 0; // 1b
// Option 2: y += c*A*x (besser: kein teurer Rückgabewert)
virtual void
AbsMat::lin_map(double c, Vec const& x, Vec& y) const = 0;
```

- Das generische CG-Verfahren erhält nun eine AbsMat (als konstante Referenz: speicheraufwändig!) und kann mit der virtuellen Methode implementiert werden:

```
int CG(AbsMat const& A, ...)
{
...; A.lin_map(1.0, x, y); ...
}
```

Beispiel Matrizen (kurz) Forts.

- Hier drei mögliche Beispiele für die Besetzungsstruktur:
  - FullMat: die meisten Elemente sind ungleich Null; alle Elemente werden abgespeichert (vollbesetzte Matrix).
  - SparseMat: die meisten Elemente sind Null; die Nicht-Null-Einträge werden einzeln mit Positionsangabe abgespeichert (dünnbesetzte Matrix, z.B. im triplet-sparse Format).
  - BandMat: alle Nicht-Null-Elemente liegen auf der Diagonalen und einigen Nebendiagonalen; diese (Neben-)Diagonalen werden als Vektoren abgespeichert (Bandstruktur).
- Hier nutzen wir also die abstrakte Klasse, um nicht für jede abgeleitete Klasse erneut das CG-Verfahren implementieren zu müssen.
- Es gibt noch viele nützliche Anwendungen von Vererbung, dies würde aber komplett den Rahmen der Vorlesung sprengen.

Ende VL 11

Aufgabe

Zur Vertiefung des Stoffs arbeitet bitte Blatt 11 durch.

Motivation

- Dieses Kapitel dient in erster Linie dazu, etwas mehr über `template` zu lernen. Eine vollständige Einführung können wir hier nicht leisten.
- In C++ gibt es zwei Arten von Templates: Funktions-Templates und Klassen-Templates.
- Die Funktions-Templates haben wir uns ja bereits angesehen.
- Und wir haben bei den C++-Datenstrukturen schon mit einigen Klassen-Templates gearbeitet.
- Wir wollen uns aber einmal kurz ansehen, wie wir eigene Klassen mit Template-Parametern ausstatten können.
- Erinnerung: Funktions-Template für die euklidische Norm:

```
template<typename T>
T norm_2(T a, T b)
{
  return std::sqrt(a*a + b*b);
}
```

- Besser wäre es (insbesondere in Klassen), einen aussagekräftigeren Bezeichner als `T` zu verwenden, bspw. Scalar. Wir sind aber wie immer sehr frei in unserer Wahl.
- Neben `typename` und Basisdatentypen können wir auch Klassen als Template-Parameter verwenden (`class`).

Klassen-Templates (1)

Beispiel 59. Beispiel für einen Punkt in beliebiger Dimension:

```
#include <iostream>
#include <array>
#include <cmath>
using std::array;

template<int dim>
class Point
{
private:
  array<double, dim> M_coord;
public:
  Point(array<double, dim> = {0.0}); // 'Trick' um Nullpunkt zu erz.
  void draw() const;
  double norm() const;
};

template<int dim>
Point<dim>::Point(array<double, dim> a)
{
  for (int i = 0; i < dim; ++i)
    M_coord[i] = a[i];
}
```

Klassen-Templates (2)

Beispiel 59. (Forts.)

```
template<int dim>
void Point<dim>::draw() const
{
  std::cout << "( ";
  for (double d: M_coord)
    std::cout << d << ' ';
  std::cout << ")\n";
}

int main()
{
  Point<3> p_3d;
  Point<2> p_2d({1.0,2.0});
  Point<9> p_9d({3.0,1.0}); // (3,1,0,...,0)
  p_3d.draw();
  p_2d.draw();
  p_9d.draw();
}
```

- Bei der externen Definition von Klassen-Template-Methoden müssen wir die Template-Parameter „mitschleppen“ und in den Klassennamespace einsetzen.

Klassen-Templates (3): Instanziierung

Vorsicht!

Wenn wir wie üblich Deklaration und Definition des Klassen-Templates Point in Point.hh und Point.cc aufteilen und Point.cc kompilieren, weiß der Compiler nicht, welche Template-Parameterwerte später einmal benötigt werden könnten. Wir müssen dies selbst mitteilen.

- Um dieses Problem zu beheben, können wir in `Point.cc` sogenannte explizite Instanziierungen benutzen.
- Syntax:

```
template class Klassenname<T>;
```

  Hierbei müssen wir natürlich für `T` unseren Zielparameter einsetzen.
  Im Fall von Funktions-Templates müssen wir `class` durch den richtigen Rückgabetyp ersetzen und die Parameter `(...)` anfügen.
- Wenn wir unseren `Point<dim>` nun also für die Dimensionen 1–3 explizit instanziieren wollen, schreiben wir ans Ende von `Point.cc`

```
template class Point<1>;
template class Point<2>;
template class Point<3>;
```

- Diese Instanziierungen können wir auch wieder auslagern (z.B. nach `Point.inst`) und `#include "Point.inst"` ans Ende von `Point.cc` schreiben.

Klassen-Templates (4): Spezialisierung

- Wenn wir für bestimmte Template-Parameter wissen, dass die Funktion/Methode einfacher wäre als die allg. Implementierung, können wir diese spezialisieren.
- Dazu lassen wir in `template<...>` alle Parameter weg, die wir spezialisieren wollen, und setzen sie stattdessen explizit in der Definition ein:

```
template<int dim>
double Point<dim>::norm() const
{
  std::cout << "General norm called\n";
  double tmp = 0.0;
  for (double d: M_coord)
    tmp += d*d;
  return std::sqrt(tmp);
}

template<>
double Point<1>::norm() const { return std::abs(M_coord[0]); }
```

Aufgabe

In der main-Methode ausprobieren (aufrufen)!

Klassen-Templates (5): Beispiele

Im Folgenden einige Beispiele aus deal.II (`https://dealii.org`), einer C++-Bibliothek für die Finite Elemente Methode, die sehr stark auf Templates aufbaut:

- Matrizen und Vektoren mit beliebigem Werttyp: z.B. `SparseMatrix<double>`.
- Quadraturformeln für vektorwertige Funktionen der Dimension `<dim>`: z.B. `QTrapez<3>`.
- Finite Elemente mit Laplace-Ansatzfunktionen in beliebiger Raumdimension: `FE_Q<2>`.
- Löser für lineare Gleichungssysteme, die mit beliebigen Matrizen und Vektoren funktionieren: z.B. `SolverCG<Parallel::distributed::Vector<double>>`.

Wir sehen also, dass Templates, genau wie Vererbung, sehr nützlich sind um mehrfache Implementierungen zu vermeiden.
Insbesondere der Template-Parameter `int dim` in deal.II erlaubt es, mit wenigen Änderungen im Code das prinzipiell gleiche Problem in zwei oder drei Raumdimensionen zu lösen.

Ende VL 12

Aufgabe

Auf Blatt 12 findet ihr Hinweise zur Strukturierung der Klausur.

The end

```
#include <iostream>

int main()
{
  std::cout
    << "Vielen Dank für die Teilnahme an dieser Vorlesung\n"
    << "und viel Erfolg für die Zukunft!"
    << std::endl;
}
```

Anhang

Motivation

Einige der Konzepte und Sprachelemente von C++ lassen sich mit bekannten Konzepten aus der Mathematik vergleichen oder weisen zumindest eine gewisse Ähnlichkeit auf.
Diese Vergleiche sind im folgenden Kapitel nach den Kapiteln des Hauptteils sortiert zu finden.

Scopes+Namespaces

- Lokale Variablen werden auch in der Mathematik verwendet, Laufindizes bei einer Summation können auch nur hinter dem $\sum$ verwendet werden.

Kontrollstukturen (1)

- Fallunterscheidungen kommen auch in der Mathematik sehr häufig vor.
  Beispiel: Absolutwert

$$|x| = \begin{cases} x, & \text{falls} \quad x \geq 0 \\ -x, & \text{falls} \quad x < 0 \end{cases}$$

```
double abs_x; //double x bekannt!
if ( x >= 0 )
  abs_x = x;
else
  abs_x = -x;
```

- Zwei sehr einfache for-Schleifen werden regelmäßig verwendet, das Summen- und das Produkzeichen.

$$\sum_{i=1}^{n} i^2$$

```
double sum = 0; //n bekannt!
for ( int i = 1 ; i <= n ; i++)
  sum += i*i; //sum += x  entspr. sum = sum + x
```

Kontrollstukturen (2)

- Das eben genannte Beispiel kann auch mit einer while-Schleife geschrieben werden. Dann ist der Laufindex aber auch nach Ende der Schleife bekannt (Scope) und das Ganze ist ein bisschen unübersichtlicher.

```
double sum = 0;
int i = 1;
while ( i <= n )
{
  sum += i*i;
  i++
}
```

Operatoren

- Bis auf den Modulo-Operator a%b ($a \mod b$) sollten alle arithmetischen Operatoren bekannt vorkommen. Dieser gibt den Rest einer Ganzzahldivision aus, also z.B. $5/4 = 2$ Rest 1, bzw. $5 \mod 4 = 1$
  Dementsprechend ist dieser Operator auch nur für ganzzahlige Datentypen definiert. `float` und `double` fallen somit weg.

Funktionen

Wie man an der Liste der mathematischen Funktionen sehen kann weisen C++ Funktionen eine gewisse Ähnlichkeit mit dem Begriff der Funktion aus der Mathematik auf. Es gibt aber ein paar Besonderheiten und Unterschiede

- Ohne die Verwendung von speziellen Datenstrukturen (später) kann der *Wertebereich* der Funktion nur eindimensional sein, es gibt also nur einen Rückgabewert.
  $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}$ ist also ohne weiteres nicht möglich.
- Die Signatur der Funktion entspricht im Prinzip dem Namen und dem Wertebereich, also `f(int,int)` entspricht $f : \mathbb{Z} \times \mathbb{Z}$
- Die Parameternamen benötigen wir spätestens bei der Implementierung der Funktion, das kann man damit vergleichen, dass diese auch erst bei Spezifikation der Abbildung benötigt werden. Z.b. $\exp : \mathbb{R} \to \mathbb{R}$ (Signatur) $\exp : x \mapsto e^x$
- Überladung von mathematischen Funktionen fängt den Sonderfall ab, dass abhängig vom Definitionsbereich ja ein anderer Wertebereich vorliegen kann, z.B. $f : x \mapsto x^2$ mit $f : \mathbb{R} \to \mathbb{R}$ und $f : \mathbb{N} \to \mathbb{N}$
- Funktionstemplates gehen hier noch einen Schritt weiter, alle „Räume“, die die Voraussetzungen der Funktion erfüllen (verwendete Operatoren etc.) kommen als Definitions- und Wertebereich in Frage. In der Algebra ist z.B. bekannt, dass der euklidische Algorithmus auf jedem beliebigen euklidischen Ring funktioniert

Arithmetische Datentypen und Zahldarstellung

Hier wollen wir uns insbesondere anschauen, welcher Datentyp Teilraum eines bestimmten Zahlenraums ist.

- `unsigned [short,long] int` sind Teilmengen von $\mathbb{N}_0$.
  Genau genommen sind es Restklassenringe $\mathbb{Z}/n\mathbb{Z}$ mit $n = 2^m$ (m Bits), bei denen gilt

$$\circ\colon (a, b) = a \circ_{\mathbb{Z}} b \mod n, \quad \circ = \{+, -, *, /\}$$

- `[short,long] int` sind Teilmengen von $\mathbb{Z}$
- `float` und `double` sind Teilmengen von $\mathbb{R}$
  Vorsicht: Es gilt aber **nicht** (bspw. `double`)

$$\mathbb{D} = \{x \in \mathbb{R}\colon a < x < b\} \text{(überabzählbar unendlich viele Einträge!)}$$

sondern

$$\mathbb{D} \subset \mathbb{R} \text{ und } \forall x \in \mathbb{R}, a < x < b \exists y \in \mathbb{D}\colon \left|\frac{y - x}{x}\right| < \text{ Eps}$$

Daher haben wir es bei Gleitkommazahlen auch regelmäßig mit Rundungsfehlern zu tun.

Zeiger und C Datenstrukturen

- Feste und dynamische Felder können als Vektoren im entsprechenden Zahlenraum angesehen werden.
  Was aber im Vergleich zu einem Vektorraum fehlt sind die arithmetischen Operationen!
- Aufzählungen (enum) können im Prinzip als Abbildungen von einer „Wortmenge“ auf ein ganzzahliges Intervall gleicher Kardinalität angesehen werden.

C++ Datenstrukturen

- Für C++ Felder und Vektoren gilt das selbe wie für die vergleichbaren C Datenstrukturen. Was aber im Vergleich zu einem Vektorraum fehlt sind die arithmetischen Operationen!
- Schleifen über Iteratoren und insbesondere die range-based for-Schleife lassen sich mit der Summation über eine (Index-)Menge vergleichen.

$$\sum_{i \in I} v_i, \quad v \in \mathbb{R}^n,\ I = \{0, 1, \ldots, n-1\}$$

entspricht

```
double index_sum = 0; // v geg.
std::vector<double>::iterator it
for ( it = v.begin(); it != v.end() ; it++}
    index_sum += *it;
```

$$\sum_{x \in S} x, \quad S = \text{Menge reeller Zahlen}$$

entspricht

```
double set_sum = 0; //v geg
for ( double x : v )
  set_sum += x;
```

Programmstrukturierung und Makefiles

Eine gute Programmstrukturierung ist mit der guten Strukturierung eines Kapitels zu einem Satz mit kompliziertem Beweis zu vergleichen.
Man kann die nötigen Definitionen aufschreiben und dann bereits den Satz mit Aussage und Beweis aufschreiben und alle eventuell verwendeten (Hilfs-)Sätze explizit im Beweis mitbeweisen.
Das ist vergleichbar mit einer großen Datei in der die komplette Funktionalität untergebracht ist.
Alternativ kann man in Vorbereitung jeden (Hilfs-)Satz vorher aufschreiben und beweisen. Damit wird nicht nur der Hauptbeweis übersichtlicher, es wird auch einfacher z.B. in einer späteren Arbeit auf Teilresultate zu verweisen.

Eigene Templateklassen

Ähnlich wie bei den Templatefunktionen ermöglichen es Templateklassen einen allgemeineren Code zu schreiben, so können wir z.B. eine Matrix implementieren, die sowohl mit ganzen, reellen oder sogar komplexen Zahlen befüllt werden kann. Oder sogar mit einer volkommen anderen Datenstruktur die entsprechende Operatoren mitbringt, z.B. irgendwelche besonderen Ringe und Körper.

Index I