Einführung in die Programmierung mit C++

Gundolf Haase

Graz, SS 2022, 30. Mai 2022

Inhaltsverzeichnis

Kapitel 1

Ein Schnelleinstieg

1.1 Was ist ein Programm und was nützt es *mir*?

Eigentlich kennt jeder bereits Programme, jedoch versteht man oft verschiedene Inhalte darunter.

- Parteiprogramm $\Longleftrightarrow$ Ideen
- Theaterprogramm $\Longleftrightarrow$ Ablaufplanung
- Musikpartitur $\Longleftrightarrow$ strikte Anweisungsfolge
- Windowsprogramm $\Longleftrightarrow$ interaktive Aktion mit dem Computer

Programmieren ist das Lösen von Aufgaben auf dem Computer mittels eigener Software und beinhaltet alle vier Teilaspekte in obiger Liste.

Eine typische Übungsaufgabe beinhaltet folgenden Satz:

$\vdots$
Ändern [editieren] Sie das Quellfile [source file] entsprechend der Aufgabenstellung, übersetzen [compilieren] und testen Sie das Programm.

Was (?) soll ich machen ?

Idee, z.B. math. Verfahren	Im Kopf oder auf dem Papier. (Was soll der Computer machen?)	Programmidee
$\Downarrow$		
Idee für den Computer aufbereiten	Entwurf. (Wie kann der Computer die Idee realisieren?)	Struktogramm
$\Downarrow$		
Idee in einer Programmiersprache formulieren.	Quelltext/Quellfile editieren. (Welche Ausdrücke darf ich verwenden?)	Programmcode
$\Downarrow$		
Quellfile für den Computer übersetzen	File compilieren [und linken]. (Übersetzung in Prozessorsprache)	ausführbares Programm
$\Downarrow$		
Programmcode ausführen	Programm mit verschiedenen Datensätzen testen	Programmtest

Bemerkungen:

1. Software = ausführbares Programm + Programmcode + **Ideen**
2. Der Lernprozeß beim Programmieren erfolgt typischerweise von unten nach oben in der vorangegangenen Übersicht.
3. Als Mathematiker sind vorrangig Ihre mathematischen Kenntnisse und Ideen gefragt, jedoch ist deren **eigenständig**e Umsetzung ein großer Vorteil bei vielen Arbeitsplätzen.
4. Mit **fundierten Kenntnisse**n in einer Programmiersprache fällt es recht leicht weitere Programmiersprachen zu erlernen.

Warnung : Das Programm auf dem Computer wird **genau das** ausführen, was im Programmcode beschrieben ist!

Typischer Anfängerkommentar: *Aber das habe ich doch ganz anders gemeint.*

Merke : Computer sind strohdumm! Erst die (korrekte und zuverlässige) Software nutzt die Möglichkeiten der Hardware.

Warum denn C++, es gibt doch die viel bessere Programmiersprache XYZ! Der Streit über die beste, oder die bessere Programmiersprache ist so alt wie es Programmiersprachen gibt. Folgende Gründe sprechen gegenwärtig für C++:

- C++ erlaubt sowohl **strukturiert**e, als auch **objektorientiert**e Programmierung.
- Strukturierte Programmierung ist die Basis jeder Programmierung im wissenschaftlich-technischen Bereich.
- Sie können in C++ reine C-Programme schreiben wie auch rein objektorientiert programmieren, d.h., es ist eine sehr gute Trainingsplattform.
- C++ erlaubt ein höheres **Abstraktionsniveau** beim Programmieren, d.h., ich muß mich nicht um jedes (fehleranfällige) informatische Detail kümmern. Andererseits kann ich genau dies tun, falls nötig.
- C++ ist eine Compilersprache, keine Interpretersprache, und damit können die resultierenden Programme schnell sein.
- Die Gnu[1]-Compiler für C++ sind für alle gängigen (und mehr) Betriebssysteme, insbesondere Linux, Windows, Mac-OS, **kostenlos** verfügbar. Desgleichen gibt es gute, kostenlose Programmierentwicklungsumgebungen (IDE) wie CodeBlocks[2] auf diesen.
  Die clang[3]-Compiler sind ebenfalls sehr zu empfehlen.
- Seit ca. 20 Jahren ist C++ meist unter den Top-5 im Programmiersprachenranking vertreten, siehe verschiedene Rankings wie TIOBE Index[4] oder PYPL[5]. Die Programmiertechniken der anderen Spitzensprachen lassen sich mit C++ ebenfalls realisieren.
- C++ mit seinen Bibliotheken ist sehr gut dokumentiert, siehe cplusplus.com[6], cppreference.com[7] und natürlich stackoverflow[8] für schwierigen Fälle.
- C++ wird weiterentwickelt, der neue C++11, C++14, C++17[9] Standard ist in den Compilern umgesetzt und C++20 ist in Arbeit. Wir werden die Möglichkeiten des Standards C++11 und C++-17 an passender Stelle benutzen.

[1] http://gcc.gnu.org/
[2] http://www.codeblocks.org/
[3] https://clang.llvm.org/
[4] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[5] http://pypl.github.io/PYPL.html
[6] http://www.cplusplus.com
[7] http://en.cppreference.com/w/
[8] https://stackoverflow.com/
[9] http://en.wikipedia.org/wiki/C++11

1.2 Das "Hello World" - Programm in C++

Wir beginnen mit dem einfachen "Hello World"-Programm, welches nur den String "Hello World" in einem Terminalfenster ausgeben wird. Damit läßt sich schon überprüfen, ob der Compiler und die IDE korrekt arbeiten (und Sie diese bedienen können).

Listing 1.1: Quelltext von Hello World

```
1 #include <iostream>                    // deklariert cout, endl
  using namespace std;                   // erlaubt Nutzung des Namensraumes std
3                                        //         nutze cout statt std::cout
  int main()                             // Beginn Hauptprogramm
5 {                                      // Beginn Scope
   std::cout << "Hello World" << std::endl;
7  return 0;                             // Beende das Programm
  }                                      // Ende Scope, Ende Hauptprogramm
```

HelloWorld.cpp

Der simple Code im Listing 1.1 enthält schon einige grundlegende Notwendigkeiten eines C++-Programmes:

- Kommentare bis zum Zeilenende werden mit `//` eingeleitet.
  Der C-Kommentar `/* */` kann auch in C++ verwendet werden.
- Jedes Programm benötigt eine Funktion `main()`, genannt Hauptprogramm.
  - `int main()` deklariert (ankündigen) das Hauptprogramm in Zeile 4.
  - Die geschweiften Klammern `{ }` in Zeilen 5 und 8 begrenzen den Funktionskörper der Funktion `main`.
  - In Zeile 7 wird der Ausdruck `return 0` durch den Strichpunkt `;` zu einer Anweisung im Programm. Diese spezielle Anweisung beendet das Hauptprogramm mit dem Rückgabewert 0.
- Die Ausgabe in Zeile 6 benutzt die I/O-Bibliotheken von C++.
  - `cout` ist ein Bezeichner für die Ausgabe im Terminal.
  - `<<` leitet den nachfolgenden String auf den Ausgabestrom (`cout`) um. Dies kann wiederholt in einer Anweisung geschehen.
  - `endl` ist der Bezeichner für eine neue Zeile im Ausgabestrom.
  - Die Preprocessor-Anweisung (beginnt mit `#`) in Zeile 1 inkludiert das, vom Compiler mitgelieferte, Headerfile *iostream* in den Quelltext. Erst dadurch können Bezeichner wie `cout` und `endl` der I/O-Bibliothek benutzt werden.
  - Ohne Zeile 2 müssen der Ausgabestrom etc. über die explizite Angabe des Namensraumes `std` angegeben werden, also als `std::cout` . Mit Zeile 2 wird automatisch der Namensraum `std` berücksichtigt wodurch auch `cout` identifiziert wird.

Quelltext eingeben und compilieren, Programm ausführen:

1. Quellfile editieren.
   `LINUX> geany HelloWorld.cpp`
2. Quellfile compilieren.
   `LINUX> g++ HelloWorld.cpp`
3. Programm ausführen.
   `LINUX> a.out` oder
   `LINUX> ./a.out` oder
   `WIN98> ./a.exe`

Tip zum Programmieren:
Es gibt (fast) immer mehr als eine Möglichkeit, eine Idee im Computerprogramm zu realisieren.
$\Longrightarrow$ Finden Sie Ihren eigenen Programmierstil und verbessern Sie ihn laufend.

1.3 Interne Details beim Programmieren

Der leicht geänderte Aufruf zum Compilieren
`LINUX> g++ -v HelloWorld.cpp`
erzeugt eine längere Bildschirmausgabe, welche mehrere Phasen des Compilierens anzeigt. Im Folgenden einige Tips, wie man sich diese einzelnen Phasen anschauen kann, um den Ablauf besser zu verstehen:

a) *Preprocessing*: Headerfiles (**.h*, *** und **.hpp*) werden zum Quellfile hinzugefügt (+ Makrodefinitionen, bedingte Compilierung)
`LINUX> g++ -E HelloWorld.cpp > HelloWorld.ii`
Der Zusatz `> HelloWorld.ii` lenkt die Bildschirmausgabe in das File *HelloWorld.ii*. Diese Datei *HelloWorld.ii* kann mit einem Editor angesehen werden und ist ein langes C++ Quelltextfile.

b) Übersetzen in *Assemblercode*: Hier wird ein Quelltextfile in der (prozessorspezifischen) Programmiersprache Assembler erzeugt.
`LINUX> g++ -S HelloWorld.cpp`
Das entstandene File *HelloWorld.s* kann mit dem Editor angesehen werden.

c) *Objektcode* erzeugen: Nunmehr wird ein File erzeugt, welches die direkten Steuerbefehle, d.h. Zahlen für den Prozessor beinhaltet.
`LINUX> g++ -c HelloWorld.cpp`
Das File *HelloWorld.o* kann nicht mehr im normalen Texteditor angesehen werden sondern mit
`LINUX> xxd HelloWorld.o`

d) *Linken*: Verbinden aller Objektfiles und notwendigen Bibliotheken zum ausführbaren Programm *a.out* .
`LINUX> g++ HelloWorld.o`

1.4 Bezeichnungen in der Vorlesung

- Kommandos in einer Befehlszeile unter LINUX:
`LINUX> g++ [-o myprog] file_name.cpp`
Die eckigen Klammern `[ ]` markieren optionale Teile in Kommandos, Befehlen oder Definitionen. Jeder Filename besteht aus dem frei wählbaren Basisnamen (*file_name*) und dem Suffix (*.cpp*) welcher den Filetyp kennzeichnet.

- Einige Filetypen nach dem Suffix:

Suffix	Filetyp
.cpp	C++ -Quelltextfile
.h	C++ -Headerfile
.o	Objektfile
.a	Bibliotheksfile (Library)
.exe	ausführbares Programm (unter Windows)

- Ein Angabe wie `··· < typ > ···` bedeutet, daß dieser Platzhalter durch einen Ausdruck des entsprechenden Typs ersetzt werden muß.

1.5 Integrierte Entwicklungsumgebungen

Obwohl nicht unbedingt dafür nötig, werden in der Programmierung häufig IDEs (Integrated Development Environments) benutzt, welche Editor, Compiler, Linker und Debugger - oft auch weitere Tools enthalten. In der LV benutzen wir freie Compiler[10] und -entwicklungstools[11], insbesondere die Compiler basieren auf dem GNU-Projekt und funktionieren unabhängig von Betriebssystem und Prozessortyp. Damit ist der von Ihnen geschriebene Code portabel und läuft auch auf einem Supercomputer (allerdings nutzt er diesen nicht wirklich aus, dazu sind weitere LVs nötig). Grundlage des Kurses sind die g++-Compiler ab Version 4.7.1, da diese auch den neuen C++11-Standard unterstützen. Gegebenenfalls muß der Code mit der zusätzlichen Option `-std=c++11` übersetzt werden.

Wir werden unter Windows die IDE **Code::Blocks**[12] Version 16.01 [Stand: Feb. 2016] benutzen welche auf den GNU-Compileren und -Werkzeugen basiert. Dies erlaubt die Programmierung unter Windows ohne die Portabilität zu verlieren, da diese IDE auch unter LINUX verfügbar ist. Sie können diese Software auch einfach privat installieren, siehe Download[13] (nehmen Sie `codeblocks-16.01mingw-setup.exe`) und das Manual[14]. Installieren Sie in jedem Fall **vorher** das Softwaredokumentationstool doxygen[15] (und cppcheck[16]), da es nur dann automatisch in die IDE eingebunden wird.

Da die Entwicklungsumgebung alles vereinfachen soll, ist vor dem ersten Hello-World-Programm etwas mehr Arbeit zu investieren.

1. `Code::Blocks` aufrufen:
   auf dem **Desktop** das Icon **Code::Blocks** anklicken.
2. In `Code::Blocks` ein neues *Projekt* anlegen:
   File $\longrightarrow$ **New** $\longrightarrow$ **Project**
   (a) Im sich öffnenden Fenster das Icon **Console Application** anklicken und dann auf **Go** klicken.
   (b) Bei der Auswahl der Programmiersprache **C++** anklicken und dann **Next**.
   (c) Den **Projektitel** angeben - hier bitte das Namensschema *bsp_nr* mit *nr* als Nummer der abzugebenden Übungsaufgabe einhalten.
   Den **Folder** (das Verzeichnis) auswählen in welchem das Projekt gespeichert werden soll. Darin wird dann automatisch ein Unterverzeichnis mit dem Projektnamen angelegt.
   Next klicken.
   (d) Die *Debug* und die *Release configuration* aktivieren. Auf **Finish** klicken.
   (e) Im Workspace erscheint das neue Projekt *bsp_1* welches in seinen *Sources* das File *main.cpp* enthält. Auf dieses File klicken.
   (f) Im Editor sehen Sie nun folgenden Programmtext:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

   (g) Compilieren und Linken: **Build** $\longrightarrow$ **Build**
   (h) Programm ausführen: **Build** $\longrightarrow$ **Run**
   (i) Speichern dieser Datei: **File** $\longrightarrow$ **Save**

[10] `http://gcc.gnu.org/`
[11] `http://www.gnu.org/`
[12] `http://www.codeblocks.org/`
[13] `http://www.codeblocks.org/downloads/26#windows`
[14] `http://www.codeblocks.org/user-manual`
[15] `http://www.doxygen.org`
[16] `http://cppcheck.sourceforge.net/`

1.6 Erste Schritte mit Variablen

Wir beginnen mit dem einfachen "Hello World"-Programm, welches nur den String "Hello World" in einem Terminalfenster ausgibt. Damit läßt sich schon überprüfen, ob der Compiler und die IDE korrekt arbeiten (und Sie dies bedienen können). Anschließend arbeiten wir mit ein paar einfachen Variablen.

Listing 1.2: Erweitertes "Hello World"

HelloWorld_2.cpp

```
#include <iostream>                          // deklariert  cout, endl
#include <string>                            // deklariert Klasse string
using namespace std;                         // erlaubt Nutzung des Namensraumes std
                                             //      nutze  string  statt  std::string
int main()                                   // Beginn Hauptprogramm
{                                            // Beginn Scope
 cout << "Hello World" << endl;
 int i;                                      // Deklaration Ganzzahl-Variable i
 cout << " i = ";
 cin >> i;                                   // Einlesen eines Wertes fuer i
                                             // Ausgeben des Wertes von i
 cout << endl << " i ist gleich " << i << " eine ganze Zahl" << endl;
                                             //
 float a, b;                                 // Deklaration Gleikomma-Variablen
 cout << " a b : " ;
 cin >> a >> b;
 cout << " ::  " << a << "  " << b << endl;//   'endl' - new line
                                             //
 float c = a+b;                              // Deklaration und Definition von c
 const string ss(" c :: ");                  // Deklaration und Definition von ss
 cout << ss << c << endl;
 return 0;                                   // Beende das Programm
}                                            // Ende Scope, Ende Hauptprogramm
```

Obiger Code enthält bekannte Teile aus dem Listing 1.1, wir werden kurz die neuen Zeilen erläutern:

[2] Die Deklarationen für den Datentyp (die Klasse) `std::string` werden inkludiert. In Zeile 3 wird der Namensraum für `std` freigegeben.

[8] Ein ganzzahlige Variable[17] `i` wird deklariert und darf damit in den nachfolgenden Zeilen des Gültigkeitsbereichs zwischen `{` und `}` verwendet werden, also bis zur Zeile 22. Die Variable `i` kann ganzzahlige Werte aus $[-2^{-31}, 2^{-31} - 1]$ annehmen.

[10] Die Variable `i` wird über den Terminal (Tastatureingabe) eingelesen. `cin` und `>>` sind Eingabestrom und -operator analog zur Ausgabe mit `cout` und `<<` .

[12] Die Variable `i` wird gemeinsam mit einem beschreibenden String (Zeichenkette) ausgegeben.

[14] Deklaration der Gleikommazahlen einfacher Genauigkeit (engl.: single precision[18]) `a` und `b`.

[16] Einlesen von Werten für `a` und `b`. Ausgabe der Variablenwerte in Zeile 17.

[19] Deklaration der Gleikommavariablen `c` und gleichzeitige Definition (Zuweisen eines Wertes) aus den Variablen `a` und `b`. Hierbei ist `=` der Zuweisungsoperator und `+` der Additionsoperator für Gleitkommazahlen.

[20] Deklaration und gleichzeitige Definition eines konstanten (`const`) Strings `ss`. Das Schlüsselwort legt fest, daß der `ss` zugewiesene Wert nicht mehr verändert werden kann.

[21] Gemeinsame Ausgabe des konstanten Strings und der Gleikommavariablen.

- Mit `{ }` eingeschlossene Bereiche eines Quelltextes definieren die Grenzen eines Gültigkeitsbereiches (scope) von darin definierten Variablen.

Damit haben wir eine Grundlage, um uns in die Programmierung mit C++ schrittweise einzuarbeiten.

[17] http://de.wikipedia.org/wiki/Integer_(Datentyp)#Maximaler_Wertebereich_von_Integer

[18] http://de.wikipedia.org/wiki/IEEE_754#Zahlenformate_und_andere_Festlegungen_des_IEEE-754-Standards

Kapitel 2

Variablen und Datentypen

2.1 Variablen

Jedes sinnvolle Programm bearbeitet Daten in irgendeiner Form. Die Speicherung dieser Daten erfolgt in Variablen.

Allgemeine Form der Variablenvereinbarung:

```
<typ> <bezeichner1> [, bezeichner2] ;
```

Die Variable

i) ist eine symbolische Repräsentation (Bezeichner/Name) für den Speicherplatz von Daten.

ii) wird beschrieben durch Typ und Speicherklasse.. Der Datentyp `<typ>` einer Variablen bestimmt die zulässigen Werte dieser Variablen.

iii) Die Inhalte der Variablen, d.h., die im Speicherplatz befindlichen Daten, ändern sich während des Programmablaufes.

2.1.1 Einfache Datentypen

Daten im Computer basieren auf dem binären Zahlensystem von Gottfried Wilhelm Leibniz[1] worin alle ganzen Zahlen durch die Summation von mit 0 oder 1 gewichteten Zweierpotenzen dargestellt werden können. So wird die Zahl 117 im Dezimalsystem durch

$$117_{(10)} = 01110101_{(2)} = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

binär dargestellt. Die grundlegende Informationseinheit im Computer ist das *Bit* welches genau einen der zwei Zustände 1 oder 0 annehmen kann (ja/nein, true/false). Somit werden in obiger Binärendarstellung von 117 genau 8 Bit benötigt. Diese 8 Bit stellen die kleinste Grundeinheit des Datenzugriffs bei heutigen Computern dar und werden als ein *Byte* bezeichnet.

Aufbauend auf der Speichereinheit Byte sind die in Tabelle 2.1.1 dargestellten, grundlegenden Datentypen in C++ vorhanden.

- Characterdaten speichern in einem Byte (= 8 bit) genau ein ASCII[2]- oder Sonderzeichen, d.h., die kodierten Buchstaben, Ziffern und Sonderzeichen werden durch eine ganze Zahl aus $[0, 255 = 2^8 - 1]$ (`unsigned char`) dargestellt.
- Der Speicherbedarf von ganzzahligen Datentypen (`short int`, `int`, `long int`, `long long int`) kann von Compiler und Betriebssystem (16/32/64 bit) abhängen. Es empfiehlt sich daher, die einschlägigen Compilerhinweise zu lesen bzw. mit dem `sizeof`-Operator mittels `sizeof(<typ>)` oder `sizeof(<variable>)` die tatsächliche Anzahl der benötigten Bytes zu ermitteln. Siehe dazu auch das Listing 2.1.

[1] Gottfried Wilhelm Leibniz, ⋆1.7.1646 in Leipzig, †14.11.1716 in Hannover
[2] `http://de.wikipedia.org/wiki/Ascii#ASCII-Tabelle`

Typ	Speicherbedarf in Byte	Inhalt	mögliche Werte
`char`	1	ASCII-Zeichen	'H', 'e', '\n'
`bool`	1	Booleanvariable	`false`, `true` [erst ab C90]
`signed char`	1		$[-128, 127]$; -117, 67
`unsigned char`	1		$[0, 255]$; 139, 67
`short [int]`	2		$[-2^{15}, 2^{15}-1]$; -32767
`unsigned short [int]`	2		$[0, 2^{16}-1]$; 40000
`int`	4	Ganze Zahlen	$[-2^{31}, 2^{31}-1]$
`unsigned [int]`	4	(integer)	$[0, 2^{32}-1]$
`long [int]`	4		wie `int`
`unsigned long [int]`	4		wie `unsigned int`
`long long [int]`	8		$[-2^{63}, 2^{63}-1]$
`unsigned long long [int]`	8		$[0, 2^{64}-1]$
`size_t`		implementierungsabhängig	`unsigned long long int`
`float`	4	Gleitkommazahlen	1.1, -1.56e-32
`double`	8	(floating point numbers)	1.1, -1.56e-132, 5.68e+287

Tabelle 2.1: Speicherbedarf grundlegender Datentypen [Stand 03/2019]

- Wir werden meist den Grundtyp `int` für den entsprechenden Teilbereich der ganze Zahlen und `unsigned int` für natürliche Zahlen verwenden. Die Kennzeichnung `unsigned` sowie das ungebräuchliche `signed` kann auch in Verbindung mit anderen Integertypen verwendet werden.
- Floating point numbers sind im Standard IEEE 754[3] definiert. Die Anzahl der Mantissenbits (`float`: 23; `double`: 52) bestimmt deren relative Genauigkeit, d.h., das Maschinen-ε, welches die kleinste Zahl des Datentyps darstellt für die $1+\varepsilon > 1$ noch gilt (`float`: $2^{-(23+1)}$; `double`: $2^{-(52+1)}$). Siehe dazu auch das hidden bit[4].

Listing 2.1: Abfrage der Speichergröße von Datentypen

```
#include <iostream>
using namespace std;
int main()
{
 int             ii;
 cout << "Size of           int: " << sizeof(ii) << endl;
 cout << "Size of  unsigned int: " << sizeof(unsigned int) << endl;
 return 0;
}
```

DataTypes.cpp

2.1.2 Bezeichner von Variablen

Die Variablenbezeichner müssen gewissen Regeln folgen (wiki[5]):

- Das erste Zeichen eines Bezeichners einer Variablen muss ein Buchstabe oder Unterstrich sein.
- Die folgenden Zeichen dürfen nur die Buchstaben A–Z und a–z, Ziffern und der Unterstrich sein.
- Ein Bezeichner darf kein Schlüsselwort der Sprache – zum Beispiel if, void und auto – sein.
- C/C++ unterscheidet zwischen Groß- und Kleinschreibung, d.h., `ToteHosen` und `toteHosen` sind unterschiedliche Bezeichner!

Ex210.cpp

[3] https://de.wikipedia.org/wiki/IEEE_754
[4] https://de.wikipedia.org/wiki/Gleitkommazahl#Hidden_bit
[5] http://de.wikipedia.org/wiki/C_(Programmiersprache)#Deklarationen

Gültig	Ungültig	Grund
`i`		
`j`		
`ijl`		
`i3`	`3i`	3 ist kein Buchstabe
`_3a`	`_3*a`	* ist Operatorzeichen
`Drei_mal_a`	`b-a`	- ist Operatorzeichen
`auto1`	`auto`	`auto` ist Schlüsselwort

Tabelle 2.2: Einge erlaubte und nicht erlaubte Variablenbezeichner

2.1.3 Gültigkeit von Bezeichnern

Die Bezeichner von Variablen, Konstanten etc. sind im Regelfall lokal, d.h., ein Bezeichner ist nur innerhalb seines, von `{ }` begrenzten Blockes von Codezeilen gültig, in welchem dieser Bezeichner deklariert wurde. Daher nennt man diesen Block den Gültigkeitsbereich (scope) dieser Variablen. Siehe dazu Listing 1.1 und §4.2.

2.1.4 Konstante mit Variablennamen

Wird eine Variablenvereinbarung zusätzlich mit dem Schlüsselwort `const` gekennzeichnet, so kann diese Variable nur im Vereinbarungsteil initialisiert werden und danach nie wieder, d.h., sie wirkt als eine Konstante.

Listing 2.2: Definition und Deklaration von Konstanten und Variablen

```
#include <iostream>
using namespace std;
int main()
{
 const int    N = 5;          // First and only initialization of constant N
       int i, j = 5;          // First           initialization of variable j
                              //                 unitialized variable i
 cout << "Hello World\n";
 i = j+ N;
 cout << endl << i << " " << j << " " << N << endl;
 return 0;
}
```

Ex226.cpp

Nach der Deklaration in Zeile 6 ist die Variable `i` noch nicht mit einem Wert belegt, d.h., sie hat einen undefinierten Status. In Zeile 7 würde der Wert von `i` zufällig sein, je nachdem was gerade in den reservierten 4 Byte als Bitmuster im Computerspeicher steht. Damit wäre das Ergebnis einer Berechung mit `i` nicht vorhersagbar (nicht deterministisch) und damit wertlos.

Also sollte `i` gleich bei der Deklaration auch definiert (initialisiert) werden. In C++ kann man ausnutzen, daß Deklarationsteil und Implementierungsteil gemischt werden können. Dies wird im Listing 2.3 ausgenutzt, sodaß die Variable `i` erst in Zeile 4 deklariert wird, wo dieser auch gleich ein sinnvoller Wert zugewiesen werden kann.

Listing 2.3: Variablen erst bei Gebrauch deklarieren

```
 const int    N = 5;          // First and only initialization of constant N
       int    j = 5;          // First           initialization of variable j
 cout << "Hello World\n";
 int i = j+ N;                // Better: initialize variable at declaration
```

Ex226_b.cpp

2.2 Literale Konstanten

Die meisten Programme, auch *HelloWorld.cpp*, verwenden im Programmverlauf unveränderliche Werte, sogenannte Literale und Konstanten. Literale Konstanten sind Werte ohne Variablenbezug welche in Ausdrücken explizit angegeben werden und zusätzliche Typinformationen enthalten können.

2.2.1 Integerliterale

Dezimalliterale (Basis 10):

```
100        // int;     100
512L       // long;    512
128053     // long; 128053
```

Oktalliterale (Basis 8):

```
020        // int;      16
01000L     // long;    512
0177       // int;     127
```

Hexadezimalliterale (Basis 16):

```
0x15       // int;      21
0x200      // int;     512
0x1ffffl   // long; 131071
```

2.2.2 Gleitkommaliterale

Gleitkommaliterale werden als `double` interpretiert solange dies nicht anderweitig gekennzeichnet ist.

Einige Beispiele im folgenden:

```
17631.0e-78
1E+10              //   double: 10000000000
1.                 //   double: 1
.78                //   double: 0.78
0.78
-.2e-3             //   double: -0.0002
-3.25f             //   single: -3.25
```

2.2.3 Zeichenliterale (Characterliterale)

Die Characterliterale beinhaltet das Zeichen zwischen den zwei Apostrophen ' :

```
'a', 'A', '@', '1' //  ASCII-Zeichen
' '                //    Leerzeichen
'_'                //    Unterstreichung/Underscore
'\''               //    Prime-Zeichen '
'\\'               //    Backslash-Zeichen  \
'\n'               //  neue Zeile
'\0'               //  Nullzeichen NUL
```

2.2.4 Zeichenkettenkonstanten (Stringkonstanten)

Die Zeichenkette beinhaltet die Zeichen zwischen den beiden Anführungszeichen " :

```
"Hello World\n"    //    \n is C-Stil fuer neue Zeile (C++: endl)
""                 //    leere Zeichenkette
"A"                //    String "A"
```

Jede Zeichenkette wird automatisch mit dem (Character-) Zeichen `'\0'` abgeschlossen (*"Hey, hier hört der String auf!"*). Daher ist `'A'` ungleich `"A"`, welches sich aus den Zeichen `'A'` und `'\0'` zusammensetzt und somit 2 Byte zur Speicherung benötigt.

Listing 2.4: Länge von String und Character

```
#include <iostream>
using namespace std;
int main()
{
 cout << "Hello World\n";                          // String literal
 cout << "A" << " String " << sizeof("A") << endl; // String
 cout << 'A' << " Char   " << sizeof('A') << endl; // Character
 return 0;
}
```

Ex224.cpp

2.3 Einige höhere Datentypen in C++

Die Standardbibliothek und die **S**tandard **T**emplate **L**ibrary (STL) stellen eine große Auswahl an komfortablen höheren Datenkonstrukten (genauer: *Klassen* und *Container*) zur Verfügung, welche das Programmieren vereinfachen. Die intensive Anwendung der Container für eigene Datenstrukturen (und *Klassen*) erfordert Kenntnisse in Klassenprogrammierung, Vererbung und Templates, welche wir erst später behandeln werden. Um aber schon mit diesen höheren Datentypen arbeiten zu können, erfolgt hier ein kurze Einführung in `string`, `complex`, `vector` und `valarray`. Vertiefend und weiterführend seien dazu u.a. [KPP02, KS02, SB95] empfohlen.

2.3.1 Die Klasse string

Diese Standardklasse `string` [KPP02, § 18] erlaubt eine komfortable Zeichenkettenverarbeitung

Listing 2.5: Erste Schritte mit `string`

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
 string aa{"Das ist der erste String"};
 string bb{};
 if (bb.empty())                  // Test auf leeren String
 {
   cout << "String bb ist noch leer" <<endl;
 }
 bb = "zweiter String";
 // cin >> bb;                    // direkte Eingabe eines Strings
 string cc=aa + " und " + bb;     // Strings aneinanderhaengen
 cout << cc << endl;
                                  // Laenge des Strings ausgeben
 cout << " ist " << cc.size() << " Zeichen lang." << endl;
 string dd(cc);                   // String dd hat gleiche Laenge wie aa
                                  //   und gleiche Daten
 for (unsigned int i=0; i<dd.size(); ++i)
 {
     cout << dd[i] << " ";        // Ausgabe i-tes Zeichen von dd
 }
 return 0;
}
```

demoString.cpp

Die Konvertierung von Zahlen in Strings und umgekehrt wird hier[6] erläutert und ist noch komfortabler mit `to_string` und `stof`.

2.3.2 Die Klasse complex

Die Standardklasse `complex<T>` [KPP02, p. 677] erlaubt die Verwendung komplexer Zahlen in der gewohnten Weise. Bei der Variablendeklaration muß der Datentyp der Komponenten der komplexen Zahl in Dreiecksklammern angegeben werden. Hier machen eigentlich nur `<double>` oder `<float>` einen Sinn.

demoComplex.cpp

Listing 2.6: Erste Schritte mit `complex`

```
#include <complex>                       //    complexe Zahlen
#include <iostream>
using namespace std;
int main()
{
 const complex<double> IMAG(0.0,1.0);    // Definition der imaginaeren Einheit
 complex<double> a, b(3.0,-1.0);         // a := 0 + 0 i, b := 3-i
 complex<float>  c(-0.876f, 2.765f), d(c);// c := -0.876 + 2.765 i,  c:=d
 cout << "a = " << a << "   b = " << b << "   c = " << c << endl;
 cout << "real b = " << real(b) << "   imag b = " << imag(b) << endl;
 b /= 3.0;
 a += c;
 cout << "c+b/3 = " << a+b << "   c/b = " <<  a/b << endl;
 cout << " d = " << d << endl;
 cout << "Polarkoord:  betrag= " << abs(d) << "   phi= " << arg(d) << endl;
 cout << " und wieder zurueck:  " << polar(abs(d), arg(d)) << endl;
```

[6] http://www.cplusplus.com/articles/D9j2Nwbp/

```
  cout << "sqrt(d) = " << sqrt(d) << endl;
18 return 0;
}
```

demoComplex.cpp

2.3.3 Die Klasse vector

demoVector.cpp

Die Containerklasse `vector<T>` [KS02, § 5.1] erlaubt die Verwendung von Vektoren in vielfältiger Weise, insbesondere bei dynamisch veränderlichen Vektoren. Bei der Variablendeklaration muß der Datentyp der Komponenten in Dreiecksklammern angegeben werden. Wir werden meist `<double>`, `<float>` oder `<int>` verwenden, obwohl alle (korrekt implementierten) Klassen hierfür benutzt werden dürfen. Die Klasse `vector<T>` enhält keine Vektorarithmetik.

Statischer Vektor

Nachfolgender Code demonstriert den Einsatz eines statischen Vektors, d.h., die Anzahl der Vektorelemente is a priori bekannt. Mit `resize(laenge)` kann diese Anzahl der Elemente auf einen neuen Wert geändert werden (dies ist eigentlich schon dynamisch). Die neuen Elemente müssen danach noch mit Werten initialisiert werden.

Listing 2.7: Erste Schritte mit `vector` als statischem Vektor

```
1  #include <iostream>
   #include <vector>
3  using namespace std;
   int main()
5  {
    //    int-vector der Laenge 5       uninitialisiert : [???, ???, ???, ???, ???]
7   vector<int>    aa(5);
    // double-vector der Laenge 5 mit 3.1 initialisiert : [-3.1, -3.1, -3.1, -3.1,
        -3.1]
9   vector<double> bb(5, -3.1);
    //  float-vector der Laenge 2 via  initializer list : [5, 3.1]
11  vector<float>  dd{5,3.1F};
    // double-vector der Laenge 0;                       : []
13  vector<double> cc;
    cc.resize(aa.size());   //  cc wird 5 Elemente lang : [???, ???, ???, ???, ???]
15  for (unsigned int i = 0; i < aa.size(); ++i )   // Laenge von Vektor aa: aa.size()
    {
17     aa[i] = i+1;
       cc[i] = aa[i]+bb[i];
19  }
    for (size_t i = 0; i < cc.size(); ++i )   // Ausgabe des Vektors
21  {
       cout << cc[i] << "   ";
23  }
    cout << endl;
25  return 0;
   }
```

demoVector.cpp

Dynamischer Vektor

Obiger Code behandelt Vektoren konstanter Länge. Wenn die Anzahl der Vektorelemente a priori unbekannt ist, dann kann ein vorhandener Vektor durch Benutzung der Methode `push_back(value)` um ein weiteres Element mit dem Wert *value* verlängert werden.

Listing 2.8: Erste Schritte mit `vector` als dynamischem Vektor

```
#include <iostream>
2  #include <vector>
   using namespace std;
4  int main()
   {
6   vector<int>    dd;          // int-vector der Laenge 0
    dd.push_back(-34);          // Laenge 1  Elemente  [-34]
8   dd.push_back(3);            // Laenge 2: Elemente  [-34, 3]
    dd.push_back(0);            // Laenge 3: Elemente  [-34, 3, 0]
10  cout << "Vektorlaenge: " << dd.size() << "  dd[0] : " << dd[0] << endl;
    dd.pop_back();              // Laenge 2: Elemente  [-34, 3]
12  cout << "Vektorlaenge: " << dd.size() << endl;
    return 0;
14 }
```

demoVector_dynamisch.cpp

Ausgabe eines Vektor

Leider können die Vektoren aus den vorherigen Beispielen nicht einfach mittels `cout << bb` ausgegeben werden. Ein solcher Versuch führt zu sehr kryptischen Fehlermeldungen des Compilers. Es gibt drei prinzipielle Möglichkeiten zur Ausgabe der Elemente eines Vektor `vector<int> bb` :

1. durch Ausgabe in einem for-Loop (konventionell),
2. durch Kopieren auf den Ausgabestream (benutzt Algorithmus `copy` aus der STL),
3. durch Definition einer Funktion für den Ausgabeoperator `<<` (Operatorüberladung).

Alle 3 Möglichkeiten werden in nachfolgendem Code demonstriert wobei jeweils exakt dieselbe Ausgabe erzielt wird. Der Sprung auf die nächste Ausgabezeile (`cout << endl` nach jeder Variante) wurde weggelassen.Die Methoden `begin()` und `end()` zeigen auf das erste Elemente und das hinterletzte Element (Diese beiden Methoden liefern Iteratoren zurück).

Listing 2.9: Ausgabe eines `vector`

```
#include <iostream>
#include <iterator>                                  // ostream_iterator fuer Variante 2
#include <vector>
using namespace std;
//            operator overloading fuer   operator<<
ostream& operator<<(ostream& s, const vector<double>& v)// Funktion fuer Variante 3
{
    for (unsigned int k=0; k<v.size(); ++k)  s << v[k] << "  "; //nutzt Variante 1
    return s;
}
int main()
{
 vector<double> bb(5, 3.1);  // double-vector der Laenge 5  mit 3.1 initialisiert
 for (unsigned int k=0; k<bb.size(); ++k)  cout << bb[k] << "  ";      // Variante 1
 copy(bb.begin(),bb.end(),ostream_iterator< double,char >(cout,"  "));// Variante 2
 cout << bb;                                                          // Variante 3
 return 0;
}
```

Ex234_b.cpp

2.3.4 *Die Klasse valarray

Die (nicht Standard) Klasse `valarray<T>` [KPP02, p. 687] erlaubt die Verwendung numerischer Vektoren wie sie von MatLab bekannt sind. Insbesondere sind Vektorarithmetik und mathematische Funktionen vorhanden. Bei der Variablendeklaration muß der Datentyp der Komponenten in Dreiecksklammern angegeben werden. Wir werden `<double>`, `<float>` oder `<int>` verwenden.

demoValarray.cpp

Listing 2.10: Vektorarithmetik mit `valarray`

```
// Der notwendige Speicher wird bereitgestellt
// Vektorarithmetik ist vorhanden
#include <iostream>
#include <valarray>
using namespace std;
int main()
{
 valarray<double> aa(5);        // double-vector der Laenge 5
 valarray<double> bb(3.1, 5);   // double-vector der Laenge 5 mit 3.1 initialisiert
 valarray<double> cc;           // double-vektor der Lange 0;
 cc.resize(aa.size());          //  cc wird 5 lang
 for (unsigned int i = 0; i < aa.size(); ++i )   // Laenge von Vektor aa: aa.size()
 {
    aa[i] = i+1;
 }
  cc = aa + bb;                 // Vektorarithmetik
  cc = sqrt(cc/aa) + atan(bb);
 for (unsigned int i = 0; i < cc.size(); ++i )
 {
    cout << cc[i] << "   ";
 }
 cout << endl;
 return 0;
}
```

demoValarray.cpp

Kapitel 3

Ausdrücke, Operatoren und mathematische Funktionen

- **Ausdrücke** bestehen aus Operanden und Operatoren.
- sind Variablen, Konstanten oder wieder Ausdrücke.
- **Operatoren** führen Aktionen mit Operanden aus.

3.1 Zuweisungsoperator

Der Zuweisungsoperator `<operand_A> = <operand_B>` weist dem linken Operanden, welcher eine Variable sein muß, den Wert des rechten Operanden zu.

Zum Beispiel ist im Ergebnis der Anweisungsfolge

Listing 3.1: Anweisungsfolge

```
{
 float x,y;
 x = 0;
 y = x+4;
}
```

Ex310.cpp

der Wert von `x` gleich 0 und der Wert von `y` gleich 4. Hierbei sind `x`, `y`, `0`, `x+4` Operanden, wobei letzterer gleichzeitig ein Ausdruck, bestehend aus den Operanden `x`, `4` und dem Operator `+`, ist. Sowohl `x = 0` als auch `y = x + 4` sind Ausdrücke. Erst das abschließende Semikolon `;` wandelt diese Ausdrücke in auszuführende Anweisungen!

Es können auch Mehrfachzuweisungen auftreten. Die folgenden drei Zuweisungsgruppen sind äquivalent.

Listing 3.2: Äquivalente Zuweisungen

```
{
 int   a,b,c;
 a = b = c = 123;              // 1. Moeglichkeit
 a = (b = (c = 123));          // 2. Moeglichkeit
 c = 123; b = c;  a = b;       // 3. Moeglichkeit
}
```

Ex310.cpp

3.2 Arithmetische Operatoren

3.2.1 Unäre Operatoren

Bei unären Operatoren tritt nur ein Operand auf.

Operator	Beschreibung	Beispiel
`-`	Negation	`-a`

3.2.2 Binäre Operatoren

Bei binären Operatoren treten zwei Operanden auf. Der Ergebnistyp der Operation hängt von den Operanden ab.

Operator	Beschreibung	Beispiel
`+`	Addition	`b + a`
`-`	Subtraktion	`b - a`
`*`	Multiplikation	`b * a`
`/`	Division (! bei Integer-Werten !)	`b / a`
`%`	Rest bei ganzzahliger Division	`b % a`

Die Division von Integerzahlen berechnet den ganzzahligen Anteil der Division, d.h., `8 / 3` liefert `2` als Ergebnis. Falls aber der Wert `2.666666` herauskommen soll, muß mindestens einer der Operatoren in eine Gleitkommazahl umgewandelt werden, wie im Listing 3.3 zu sehen ist.

Listing 3.3: Fallen und Typumwandlung (Casting) bei Integeroperationen

```
{
 float ij_mod, ij_div, float_ij_div;
 int   i=8, j=3;
 ij_div = i / j;                          // Achtung: Ergebnis ist 2
 ij_mod = i % j;                          // Modulu-Rechnung
                    // Casting des Nenners ==> Resultat ist 2.666666
 float_ij_div = i/(float)j;               // explizites C-casting (alt)
 float_ij_div = i/static_cast<float>(j);  // explizites C++-casting
 float_ij_div = i/(j+0.);                 // implizites Casting
}
```

Ex320.cpp

Achtung: Der Modulo einer negativen Zahl wird in C++ nicht so berechnet, wie man es seitens der Algebra erwarten würde, z.B., `-13 % 4` liefert `-1` statt `3` .

Bzgl. der Vorrangregeln für Operatoren sei auf die Literatur verwiesen, die alte Regel "*Punktrechnung geht vor Strichrechnung*" gilt auch in C/C++. Analog werden Ausdrücke in runden Klammern `( <ausdruck> )` zuerst berechnet.

Listing 3.4: Integeroperationen

```
{
 int k;
 double x = 2.1;
 k = 1;                          // k stores 1
 cout << " k_1 : " << k << endl;
 k = 9/8;                        // k stores 1, Int. Div.
 cout << " k_2 : " << k << endl;
 k = 3.14;                       // k stores 3, truncated
 cout << " k_3 : " << k << endl;
 k = -3.14;                      // k stores -3 or -4, compiler dependent
 cout << " k_4 : " << k << endl;
 k = 2.9e40;                     // undefined ee
 cout << " k_5 : " << k << endl;
 x = 9/10;                       // x stores 0
 cout << " x_1 : " << x << endl;
 x = (1.0+1)/2;                  // x stores 1.0
 cout << " x_2 : " << x << endl;
 x = 1 + 1.0/2;                  // x stores 1.5
 cout << " x_3 : " << x << endl;
 x = 0.5 + 1/2;                  // x stores 0.5
 cout << " x_4 : " << x << endl;
}
```

Ex320.cpp

3.3 Vergleichsoperatoren

Vergleichsoperatoren sind binäre Operatoren. Der Ergebniswert ist immer ein Boolean- bzw. Integerwert, wobei `false` dem Wert `0` zugeordnet ist und `true` einen Wert ungleich `0` entspricht. Zeile 6 im nächsten Listing dokumentiert die Ausgabe von Text statt 0/1 bei bei der Ausgabe solcher Booleanwerte.

Operator	Beschreibung	Beispiel
`>`	größer	`b > a`
`>=`	größer oder gleich	`b >= 3.14`
`<`	kleiner	`a < b/3`
`<=`	kleiner oder gleich	`b*a <= c`
`==`	gleich (! bei Gleitkommazahlen!)	`a == b`
`!=`	ungleich (! bei Gleitkommazahlen!)	`a != 3.14`

Listing 3.5: Vergleichsoperatoren und Ausgabe von `boolean`

```
bool bi = ( 3 <= 4 );                                   // Boolean
bool bj = ( 3 >  4 );                                   // Boolean
cout << " 3 <= 4    TRUE   = " << bi << endl;
cout << " 3 >  4    FALSE  = " << bj << endl;                       //    1/0
cout << " 3 >  4    FALSE  = " << std::boolalpha << bj << endl;  // true/false
```

Ex330.cpp

Ein **typischer Fehler** tritt beim Test auf Gleichheit auf, indem statt des Vergleichsoperators `==` der Zuweisungsoperator `=` geschrieben wird. Der Compiler akzeptiert den Quelltext, compilerabhängig werden Warnungen ausgegeben [`g++ -Wall ...`], siehe §13.5.

Listing 3.6: Typischer Fehler bei Test auf Gleichheit

```
int i;
i = 2;
cout << " i == 3  ist ein  Vergleich : " << ( i == 3) << endl;
cout << "           und belaesstt den Wert von i bei " << i << endl;
cout << " i =  3  ist eine Zuweisung : " << ( i = 3) << endl;
cout << "           und aendert den Wert von i in " << i << endl;
//
//              Use of wrong expression has side effects
i = 2;
cout << " AA: i = " << i << endl;
if ( i = 3 )                          //  E R R O R : Assignment i=3 is always true !!
  {
    cout << " BB: i = " << i << endl;
    i = 0;
  }
cout << " CC: i = " << i << endl; // i is always 0 !!
```

Ex330.cpp

Im inkorrekten Code tritt der unerwünschte Nebeneffekt auf, daß der Wert der Variablen `i` im Test geändert wird, während folgender, korrekter Code keinerlei Nebeneffekte aufweist.

3.4 Logische Operatoren

Es gibt nur einen unären logischen Operator:

Operator	Beschreibung	Beispiel
`!`	logische Negation	`! (3>4)  // TRUE`

und zwei binäre logische Operatoren:

Operator	Beschreibung	Beispiel
`&&`	logisches UND	`(3>4) && (3<=4)  // FALSE`
`||`	logisches ODER	`(3>4) || (3<=4)  // TRUE`

Die Wahrheitswertetafeln für das logische UND und das logische ODER sind aus der Algebra bekannt (ansonsten, siehe Literatur).

Listing 3.7: Verknüpfung logischer Tests

```
#include <iostream>
using namespace std;
int main()
{
 const int Ne = 5;              // one limit
       int i;
 cout << " i = " ; cin  >> i;   // Input i
                                //
 if ( 0 <= i && i <= Ne )       // Is 'i' in intervall [0,Ne] ?
  {
    cout << "i between 0 and 5" << endl;
  }
 return 0;
}
```

Ex340.cpp

3.5 Bitorientierte Operatoren

Ein Bit ist die kleinste Informationseinheit mit genau zwei möglichen Zuständen:

$$\begin{cases}\text{bit gelöscht}\\ \text{bit gesetzt}\end{cases} \equiv \begin{cases}0\\ L\end{cases} \equiv \begin{cases}0\\ 1\end{cases} \equiv \begin{cases}\texttt{false}\\ \texttt{true}\end{cases}$$

Ein Byte besteht aus 8 Bit und damit ist eine `short int` Zahl 16 Bit lang.

Als Operatoren in Bitoperationen treten normalerweise Integer-Ausdrücke auf.

3.5.1 Unäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
`~`	Binärkomplement, bitweise Negation des Operanden	`~k`

3.5.2 Binäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
`&`	bitweises UND der Operanden	`k & l`
`|`	bitweises ODER	`k | l`
`^`	bitweises exklusives ODER	`k ^ l`
`<<`	Linksverschiebung der Bits von `<op1>` um `<op2>` Stellen	`k << 2  // = k*4`
`>>`	Rechtsverschiebung der Bits von `<op1>` um `<op2>` Stellen	`k >> 2  // = k/4`

Wahrheitswertetafel:

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	L	0	L	L
L	0	0	L	L
L	L	L	L	0

Diese Operatoren seien an den folgenden Beispielen demonstriert:

Listing 3.8: Bitoperationen

```
1  int main()
   {
3    short int l = 5;                //      0..000L0L
     short int k = 6;                //      0..000LL0
5    short int   n1,n2,n3,n4,n5,n6,n7;
     n1 = ~k;       // Complement            L..LLL00L    = -7 = -6 - 1
7    n2 = k & l;    // bit-AND               0..000L00    = 4
     n3 = k | l;    // bit-OR                0..000LLL    = 7
9    n4 = k ^ l;    // bit-XOR               0..0000LL    = 3
     n5 = k << 2;   // shift  left by 2      0..0LL000    = 24 = 6 * 2^2
11   n6 = k >> 1;   // shift right by 1      0..0000LL    = 3  = 6 / 2^1
     n7 = l >> 1;   // shift right by 1      0..0000L0    = 2  = 5 / 2^1
13   return 0;
   }
```

Ex350.cpp

Die Bitoperationen sind nützlich beim Test, ob eine gerade oder ungerade Integerzahl vorliegt. Das niederwertigste Bit kann bei Integerzahlen zur Unterscheidung „gerade/ungerade Zahl“ genutzt werden (siehe auch die Bitdarstellung der Zahlen 5 und 6 im obigen Code). Wenn man daher dieses Bit mit einem gesetzten Bit über die ODER-Operation verknüpft, so bleibt das niederwertigste Bit bei ungeraden Zahlen unverändert. Dies wird im nachfolgenden Code ausgenutzt.

Listing 3.9: Test auf ungerade Zahl (Bitoperationen)

```
1  #include <iostream>
   using namespace std;
3  int main()
   {
5   const int   Maske = 1;              // 0..00000L
          int   i;
```

```
7   cout << endl << "   Eingabe einer ganzen Zahl: ";
    cin  >> i;                          //  read number
9   //      Check for odd number:
    //           Last bit remains unchanged for odd numbers
11  cout << "   " << i << " ist eine ";
    if ((i | Maske) == i)
13   {
       cout << "ungerade";
15   }
    else
17   {
       cout << "gerade";
19   }
    cout << " Zahl." << endl << endl;
21  return 0;
}
```

Ex351.cpp

3.6 Operationen mit vordefinierten Funktionen

3.6.1 Mathematische Funktionen

Im Headerfile *cmath* werden mathematische Funktionen[1] bereitgestellt. Die Nutzung des Namespaces `std` ist empfehlenswert , also `std::exp()` oder gloabl mit `using namespoace std;`.

Funktion/Konstante	Beschreibung
`sqrt(x)`	Quadratwurzel von x: $\sqrt[2]{x}$ $(x \geq 0)$
`cbrt(x)`	Kubicwurzel von x: $\sqrt[3]{x}$
`exp(x)`	e^x
`log(x)`	natürlicher Logarithmus von x: $\log_e x$ $(x > 0)$
`pow(x,y)`	Potenzieren ($x > 0$ falls y nicht ganzzahlig)
`abs(x)`	Absolutbetrag von x: $\|x\|$
`fmod(x,y)`	realzahliger Rest von x/y $(y \neq 0)$
`ceil(x)`	nächste ganze Zahl $\geq x$
`floor(x)`	nächste ganze Zahl $\leq x$
`round(x)`	gerundete Zahl
`sin(x)`, `cos(x)`, `tan(x)`	trigonometrische Funktionen
`asin(x)`, `acos(x)`	trig. Umkehrfunktionen ($x \in [-1, 1]$)
`atan(x)`	trig. Umkehrfunktion
`M_E`	Eulersche Zahl e
`M_PI`	π
`std::numbers::pi`	π (ab C++20)

Tabelle 3.1: Eine kleine Auswahl mathematischer Funktionen

Falls unter Windows die Konstante `M_PI` vom Compiler nicht erkannt wird, dann vor der Zeile `#include <cmath>` die Zeile `#undef __STRICT_ANSI__` einfügen. Alternativ kann auch die Compileroption `-U__STRICT_ANSI__` verwendet werden.

Mit dem Header `<numbers>` werden ab C++20 gängige mathematische Konstanten[2] über den Namensraum `std::numbers::` sauber eingeführt und können typsicher verwendet werden.

Listing 3.10: Konstanten in C++20

```
#include <numbers>     // pi, pi_v<T>  in C++20
{
   float alpha  = 123.4;                                    // Gradmass
   float angle1 = alpha/180.0f*std::numbers::pi_v<float>;   // Radiant
   float angle2 = alpha/180.0f*std::numbers::pi;            // Casting von double
}
```

Für die Zulässigkeit der Operationen, d.h., den Definitionsbereich der Argumente, ist der Programmierer verantwortlich. Ansonsten werden Programmabbrüche oder unsinnige Ergebnisse produziert.

[1] https://en.cppreference.com/w/cpp/header/cmath
[2] https://en.cppreference.com/w/cpp/header/numbers

Listing 3.11: Mathematische Funktionen

```
#include <cmath>                    // include defintions for math. functions
#include <iostream>
using namespace std;
int main()
{
 double x = -1;                              //   x < 0 !!
 double y = sqrt(x);                         //   Square root with wrong argument
 cout << "x = " << x << " ,  y  = " << y << endl;
 double z = std::abs(x);                     //   Absolut value
 cout << "x = " << x << " , |x| = " << z << endl;
 y = 3.0;                                    //   try 2.0 , 3.0 and 2.5
 z = pow(x,y);                               //   Power function  x^y
 cout << "(x,y) = " << x << " , " << y << " , x^y = " << z << endl;
 return 0;
}
```

Ex361.cpp

Die Funktionen aus *cmath* werden in einer speziellen mathematischen Bibliothek gespeichert, sodaß der Befehl zum Compilieren und Linken diese Bibliothek *libm.a* berücksichtigen muß, d.h.

```
LINUX> g++ Ex361.cpp [-lm]
```

3.6.2 Funktionen für die Klasse `string` (C++-Strings)

C++ bietet eine komfortablere Möglichkeit zur Zeichenkettenverarbeitung über die Klasse **`string`**, welche im Headerfile *string* deklariert ist.

Listing 3.12: Benutzung von C++-Strings

```
#include <iostream>
#include <string>                          // C++-strings and functions for C++-strings
using namespace std;
int main()
{
 string s, s1("Hello"), s2("World");
 cout << "s1 = " << s1 << endl;
 cout << "s2 = " << s2 << endl;
 bool bi = s1 < s2;                  // lex. comparison
 cout << "cmp : " << bi << endl;
 s = s1;                             // copy s1 on s
 cout << "s   : " << s << endl;
 s = s + s2;                         // Appends s2 on s
 cout << "s   : " << s << endl;
 int i = s.size();                   // length of string s
 cout << "Length of s : " << i << endl;
}
```

Ex363.cpp

3.7 Inkrement- und Dekrementoperatoren

3.7.1 Präfixnotation

```
++<lvalue>                  // <lvalue> = <lvalue> + 1
--<lvalue>                  // <lvalue> = <lvalue> - 1
```

Listing 3.13: Präfixnotation

```
{
  int i=3, j;
   ++i;                // i = 4

   j = ++i;            // i = 5, j = 5
                       // above prefix notation is equivalent to
   i = i + 1;
   j = i;
}
```

3.7.2 Postfixnotation

```
<lvalue>++                  // <lvalue> = <lvalue> + 1
<lvalue>--                  // <lvalue> = <lvalue> - 1
```

Listing 3.14: Postfixnotation

```
{
   int i=3, j;
   i++;                 // i = 4
   j = i++;             // i = 5, j = 4
                        // above postfix notation is equivalent to
   j = i;
   i = i + 1;
}
```

Prä- und Postfixnotation sollten sparsam verwendet werden. Meist benutzt man die Präfixnotation für die Indexvariablen in Zyklen (§ 4), da die entsprechende Postfixnotation eine zusätzliche Kopieroperation beinhaltet und damit aufwändiger (teurer) ist.

3.8 Zusammengesetzte Zuweisungen

Wertzuweisungen der Form

```
<lvalue> = <lvalue> <operator> <ausdruck>
```

können zu

```
<lvalue> <operator>= <ausdruck>
```

verkürzt werden.
Hierbei ist `<operator>` ∈ `{+,-,*,/,%,&,|,^,<<,>>}` aus § 3.2 und § 3.5 .

Listing 3.15: Kombination von Operatoren mit einer Zuweisung

```
{
  int   i,j,w;
  float x,y;

  i  += j              // i = i+j
  w >>= 1;             // w = w >> 1 (= w/2)
  x  *= y;             // x = x*y
}
```

3.9 Weitere nützliche Konstanten

Für systemabhängige Zahlenbereiche, Genauigkeiten usw. ist das Headerfile *limits* in C++ recht hilfreich.

Listing 3.16: Zahlbereichskonstanten in C++

```
#include <iostream>
#include <limits>                                   //    numeric_limits
using namespace std;
int main()
{
 cout << "max(double)  " << numeric_limits<double>::max()     << endl;
 cout << "min(float)   " << numeric_limits<float> ::min()     << endl;
 cout << "min(int)     " << numeric_limits<int>   ::min()     << endl;
 cout << "eps(float)   " << numeric_limits<float> ::epsilon() << endl;
 return 0;
}
```

Ex390.cpp

Die Nutzung der Templateklasse `numeric_limits` erfordert (zumindest theoretisch) Kenntnisse von Namensbereichen, Klassen und Templates. Daher ist Listing 3.16 hier nur als kurzes Kochrezept angegeben. Für weitere Methoden (Funktionen) siehe [KPP02, p.710] und [KS02, §13.5] und [Wol06, §7.3.4].

Kapitel 4

Kontrollstrukturen

4.1 Einfache Anweisung

Eine einfache Anweisung setzt sich aus einem Ausdruck und dem Semikolon als Abschluß einer Anweisung zusammen:

```
<ausdruck> ;
```

Listing 4.1: Anweisung

```
cout << "Hello World" << endl;
i = 1 ;
```

4.2 Block

Die Blocksequenz (auch Verbundanweisung, oder kurz Block) ist eine Aufeinanderfolge von Vereinbarungen und Anweisungen mittels geschweifter Klammern:

```
{
  <anweisung_1>
       ...
  <anweisung_n>
}
```

Listing 4.2: Blocksequenz

```
{       // Blockanfang (scope begin)
  int i,n;            // Vereinbarung

  i = 0;              // Anweisung
  n = i+1;            // Anweisung
}       // Blockende (scope end)
```

- In C muß der Vereinbarungsteil dem Blockanfang direkt folgen. In C++ können mehrere Vereinbarungsteile im Block existieren, sie müssen nur vor der jeweiligen Erstbenutzung der Variablennamen stehen. Dies hat den Vorteil, daß Variablen nur dort definiert (und initialisiert!!) werden müssen wo sie auch gebraucht werden.
- Der schließenden Klammer des Blockendes "}" folgt kein Semikolon.
- Ein Block kann stets anstelle einer Anweisung verwendet werden.
- Blöcke können beliebig ineinander geschachtelt werden.

- Die in einem Block vereinbarten Variablen sind nur dort sichtbar, d.h., außerhalb des Blocks ist die Variable nicht existent (Lokalität). Umgekehrt kann auf Variablen des übergeordneten Blocks zugegriffen werden.

Listing 4.3: Gültigkeitsbereich (scope) von Variablen

```
#include <iostream>
using namespace std;
int main()
{                                          // Begin outer scope
 int i{1}, j{1};                              // outer i
 {                                            // Begin inner scope
   int k;                                        // inner k
   int i;                                        // inner i
   i = k = 3;
   cout << "  inner i = " << i << endl;          // inner i is used
   cout << "i_outer j = " << j << endl;
 }                                            // End inner scope
 cout << "  outer i = " << i << endl;          // outer i is used
 cout << "  outer j = " << j << endl;
// j = i+k;                                                 // k undeclared !!
}                                          // End outer scope
```

Ex420.cpp

Im Listing 4.3 tritt die Variable `i` sowohl im inneren als auch im äußeren Block auf. Dies nennt man *shadow variable*, d.h., die innere Variable verdeckt die äußere. Damit ist der Code schwerer zu verstehen und fehleranfälliger, ergo vermeiden Sie shadow variables in Ihren Programmen. Beim Gnu-Compiler warnt die Option `-Wshadow` davor.

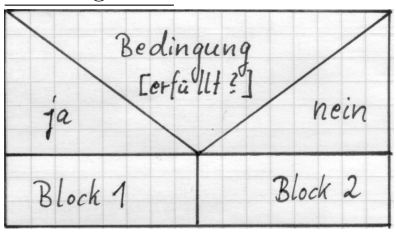
4.3 Verzweigungen

Die allgemeine Form der Verzweigungen (auch Alternative) ist

```
if ( <logischer ausdruck> )
  <anweisung_A>
else
  <anweisung_B>
```

und zählt ihrerseits wiederum als Anweisung. Der `else` -Zweig kann weggelassen werden (einfache Alternative).

Struktogramm:



Wie so oft kann ein konkretes Problem auf verschiedene Weise programmiert werden.
Beispiel: Wir betrachten dazu die Berechnung der Heaviside-Funktion

$$y(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

und stellen die folgenden vier Varianten der Implementierung vor.

a) Setzen des Standardwertes kombiniert mit einer einfache Alternative,

b) zweifache Alternative ohne Blöcke (da in jedem Zweig genau eine Anweisung steht),

c) zweifache Alternative mit Blöcken (allgemeiner),

d) mit dem Entscheidungsoperator. Treten in einer zweifachen Alternative in jedem Zweig nur je eine Wertzuweisung zur selben Variablen auf (wie in Versionen b) und c)), dann kann der Entscheidungsoperator

`<log. ausdruck> ? <ausdruck_A> : <ausdruck_B>`

Listing 4.4: Vier Varianten um Heavisidefunktion zu implementieren

```
#include <iostream>
using namespace std;
int main()
{
 double x,y;
 cout << endl << " Input Argument      : ";
 cin  >> x;
//
//                              Version a) Einseitige Alternative
//
 y = 0.0 ;                 // Setzen des Wertes fuer den else-Zweig
 if ( x >= 0.0 )
     y = 1.0 ;             // genau eine Anweisung im if-Zweig, daher keine {} noetig
 cout << " Result of version a) : " << y << endl;
//
//                              Version b) Zweiseitige Alternative
//
 if ( x >= 0.0 )
     y = 1.0 ;           // genau eine Anweisung im if-Zweig, daher keine {} noetig
 else
     y = 0.0 ;           // genau eine Anweisung im else-Zweig, daher keine {} noetig
 cout << " Result of version b) : " << y << endl;
//
//                              Version c) Zweiseitige Alternative mit Klammern
//
 if ( x >= 0.0 )
   {
     y = 1.0 ;
   }
 else
   {
     y = 0.0 ;
   }
 cout << " Result of version c) : " << y << endl;
//
//                              Version d) Entscheidungsoperator
//
 y = x >= 0 ? 1.0 : 0.0 ;
 cout << " Result of version d) : " << y << endl << endl;
 return 0;
}
```

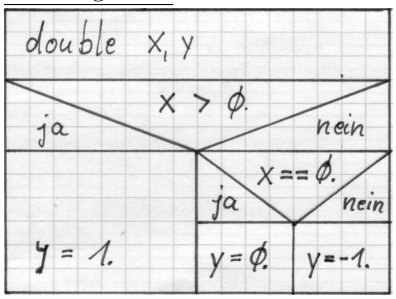
Ex431.cpp

Beispiel: Ein weiteres Beispiel ist die Berechnung der Signum-Funktion (Vorzeichenfunktion)

$$y(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

und wir stellen mehrere Varianten der Implementierung vor. Ex432.cpp

Struktogramm:



Wir betrachten folgende Implementierungsvarianten:

a) Schachtelung der Alternativen, d.h., der `else`-Zweig enthält nochmals eine Alternative.

b) Falls der `else`-Zweig nur aus einer weiteren `if-else`-Anweisung besteht, kann das `else` mit dem inneren `if` zum `elseif` kombiniert werden.

c) Die Signumfunktion kann auch als Kombination von zwei Heaviside-Funktionen ausgedrückt werden und damit als Kombination zweier Entscheidungsoperatoren implementiert werden (kurz und knapp, aber aus dem Code nur schwer zu verstehen).

Listing 4.5: Drei Varianten der Signum-Funktion

```
#include <iostream>
using namespace std;
int main()
{
 double x,y;
 cout << endl << " Input Argument       : ";
 cin  >> x;
//
//                                        Version a)
//
 if ( x > 0.0 )
   {
     y = 1.0 ;
   }
 else
   {
     if ( x == 0.0 )
       {
         y = 0.0 ;
       }
     else
       {
         y = -1.0 ;
       }
   }
 cout << " Result of version a) : " << y << endl;
//
//                                        Version b)
//
 if ( x > 0.0 )
   {
     y = 1.0 ;
   }
 else if ( x == 0.0 )        // else und nachfolgendes, inneres if zusammengezogen
   {
     y = 0.0 ;
   }
 else                        // diese else gehoert zu inneren if
   {
     y = -1.0 ;
   }
 cout << " Result of version b) : " << y << endl;
//
//                                        Version c) Entscheidungsoperator
//
 y = (x > 0 ? 1. : 0.) + (x < 0 ? -1. : 0.);
 cout << " Result of verion c) : " << y << endl << endl;
 return 0;
}
```

Ex432.cpp

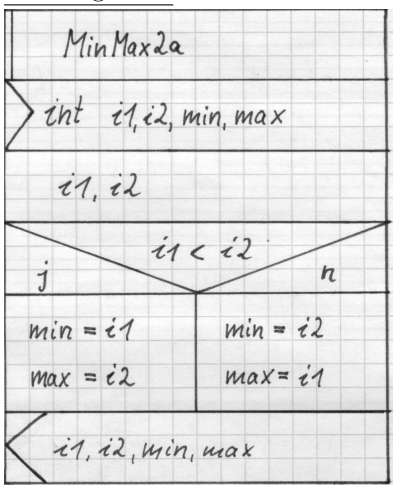
Allgemein kann eine solche Mehrwegentscheidung als

```
if ( <logischer ausdruck_1> )
  <anweisung_1>
else if ( <logischer ausdruck_2> )
  <anweisung_2>
        ...
else if ( <logischer ausdruck_(n-1)> )
  <anweisung_(n-1)>
else
  <anweisung_n>
```

geschrieben werden, wobei der `else`-Zweig wiederum optional ist.

Beispiel: Bestimmung von Minimum und Maximum zweier einzugebender Zahlen.

Struktogramm:



Listing 4.6: Drei Varianten das Minimum und das Maximum zweier Zahlen zu bestimmen

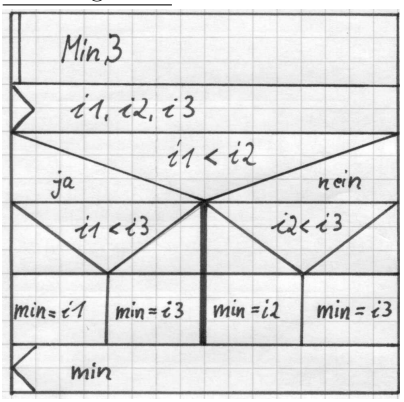
```
#include <iostream>
using namespace std;
int main()
{
 int i1, i2, imin, imax;
 cout << endl << " Input Arguments   i1 i2 : ";
 cin  >> i1 >> i2 ;
//
//                                        Version a)
//
 if ( i1 < i2 )
   {
     imin = i1 ;
     imax = i2 ;
   }
 else
   {
     imin = i2 ;
     imax = i1 ;
   }
 cout << " Min,Max  (a) : " << imin << " , " << imax << endl;
//
//                                        Version b)
//
 imin = imax = i1;
 if ( imin > i2 )
   {
     imin = i2 ;
   }
 else
   {
     imax = i2 ;
   }
 cout << " Min,Max  (b) : " << imin << " , " << imax << endl;
//
//                                        Version c), Entscheidungsoperator
//
 imin  = (i1 < i2) ? i1 : i2 ;
 imax  = (i1 > i2) ? i1 : i2 ;
 cout << " Min,Max  (c) : " << imin << " , " << imax << endl;
}
```

Ex433.cpp

Die Funktionen `min` und `max` zur Bestimmung des Minimus/Maximums zweier Zahlen sind in der STL (Standard Template Library) bereits implementiert, somit lassen sich alle 3 Varianten auch durch `imax = max(i1,i2); imin = min(i1,i2);` ausdrücken.

Beispiel: Bestimmung des Minimums dreier einzugebender Zahlen.

Struktogramm:



Listing 4.7: Varianten des Minimums dreier Zahlen

```
#include <iostream>
using namespace std;
int main()
{
 int i1,i2,i3,imin;
 cout << endl << " Input Arguments   i1 i2 i3  : ";
 cin  >> i1 >> i2 >> i3;
//
//                  Version a) geschachtelte Alternativen
//
 if ( i1 < i2 )
   {
     if ( i1 < i3 )
       {
         imin = i1;
       }
     else
       {
         imin = i3;
       }
   }
 else
   {
     if ( i2 < i3 )
       {
         imin = i2;
       }
     else
       {
         imin = i3;
       }
   }
 cout << " Min  (a) : " << imin  << endl;
//
//                  Version b), mit Enscheidungsoperator
//
 if ( i1 < i2 )
   {
     imin = ( i1 < i3 ) ? i1 : i3 ;
   }
 else
   {
     imin = ( i2 < i3 ) ? i2 : i3 ;
   }
 cout << " Min  (b) : " << imin  << endl;
//
//                  Version c) Entscheidungsoperator wird intensiv genutzt,
//                  Code wird sehr kompakt - jedoch unlesbar
//                  (WOP - Write Only  Programming)
//
 imin = (i1 < i2) ? ( ( i1 < i3 ) ? i1 : i3 ) : ( ( i2 < i3 ) ? i2 : i3 ) ;
 cout << " Min  (c) : " << imin  << endl;
 return 0;
}
```

Ex434.cpp

4.4 Der Zählzyklus (`for`-Schleife)

Beim Zählzyklus steht die Anzahl der Zyklendurchläufe **a-priori** fest, der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus. Die allgemeine Form ist

```
for (<ausdruck_1>; <ausdruck_2>; <ausdruck_3>)
  <anweisung>
```

Am besten sei der Zählzyklus an einem Beipiel erläutert.

Beispiel: Es ist die Summe der ersten 5 natürlichen Zahlen zu berechnen: $isum = \sum_{i=1}^{5} i$.

Listing 4.8: Summe der ersten 5 natürlichen Zahlen

```
#include <iostream>
using namespace std;
int main()
{
 int   i,isum,n;                          // loop index, sum, last index
 n = 5;                                   // initialize last index
//
//                                          Conventional version
//
 isum = 0;                                // initialize sum (integer !)
 for ( i = 1; i <= n; i=i+1)
   {
     isum = isum +i;
   }
 cout << endl << "Sum of first " << n << " natural numbers = " << isum << endl;
//
//                                          Sophisticated version (poor style)
//
 for ( isum = 0, i = 1; i <= n; isum += i , ++i) ;
 cout << endl << "Sum of first " << n << " natural numbers = " << isum << endl;
}
```

Ex440.cpp

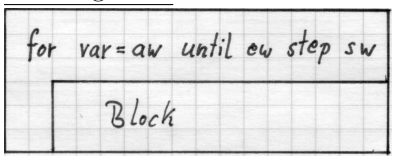
Im obigen Programmbeispiel ist `i` die Laufvariable des Zählzyklus, welche mit `i = 1` (`<ausdruck_1>`) initialisiert, mit `i = i+1` (`<ausdruck_3>`) weitergezählt und in `i <= n` (`<ausdruck_2>`) bzgl. der oberen Grenze der Schleifendurchläufe getestet wird. Im Schleifeninneren `sum = sum + i;` (`anweisung`) erfolgen die eigentlichen Berechnungsschritte des Zyklus. Die Summationsvariable `sum` muß vor dem Eintritt in den Zyklus initialisiert werden.

Eine kompakte Version dieser Summationsschleife (korrekt, aber sehr schlecht lesbar) wäre :
`for (isum = 0, int i = 1; i <= n; isum += i, ++i)`
Man unterscheidet dabei zwischen dem Abschluß einer Anweisung "`;`" und dem Trennzeichen "`,`" in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet.

Der `<ausdruck_2>` ist stets ein logischer Ausdruck (§ 3.3-3.4) und `<ausdruck_3>` ist ein arithmetischer Ausdruck zur Manipulation der Laufvariablen, z.B.

```
++i
j = j-2
j += 2
x = x+h          // float-Typ
k = 2*k          // Verdoppelung
l = l/4          // Viertelung - Vorsicht bei Integer
```

Struktogramm:



- Die Laufvariable muß eine einfache Variable aus § 2.1.1 sein, z.B., `int` oder `double`, oder ein Iterator § 6.2 (auch Pointer).
- Vorsicht bei Verwendung von Gleitkommazahlen (`float`, `double`) als Laufvariable. Dort ist der korrekte Abbruchtest wegen der internen Zahldarstellung u.U. nicht einfach zu realisieren.

Loop_Float.cpp

Beispiel: Es sei die Doppelsumme

$$\text{sum} = \sum_{k=1}^{n} \underbrace{\sum_{i=1}^{k} \frac{1}{i^2}}_{t_k} = \sum_{k=1}^{n} t_k$$

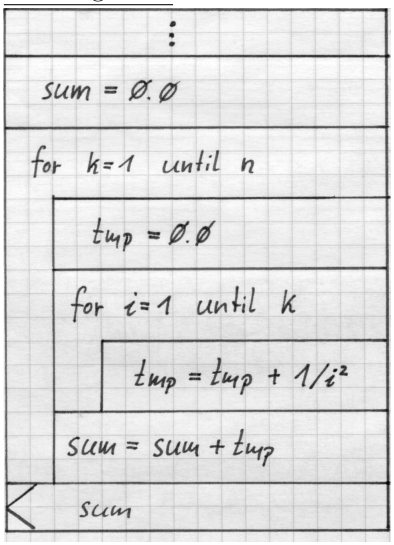
für einzugebende n zu berechnen.

Ex442fast.cpp

Hinweis: Für die innere Summe gilt $t_k = t_{k-1} + 1/k^2$ für $k = 1, \ldots, n$ mit $t_0 = 0$. Dadurch fällt diese kostspielige Summation weg wodurch der gesamte Code signifikant schneller wird [a]

[a] Andere Möglichkeit: Vertauschen der Summationen $\sum_{i=1}^{n} \sum_{k=i}^{n} \frac{1}{i^2} = \sum_{i=1}^{n} \frac{n-i+1}{i^2}$ [A. Reinhart]

Struktogramm:



Listing 4.9: Geschachtelte Zählzyklen

```
#include <iostream>
using namespace std;
int main()
{
 int    n;                    // last index
 cout << endl;
 cout << "  Input n : ";
 cin  >> n;
 double sum_k = 0.0;          // declare + initialize outer sum
 for ( int k = 1; k <= n; ++k)
  {
    double sum_i = 0.0;                  // declare + initialize inner sum
    for ( int i = 1; i <= k; ++i)                // last index depends on k !!
     {
        sum_i = sum_i + 1.0/i/i;
     }
    cout << "  Sum (" << k << ") = " << sum_i << endl;
    sum_k = sum_k + sum_i;               // sum_k grows unbounded
//    sum_k = sum_k + sum_i/n;                 // sum_k is bounded
  }
 cout << endl;
 cout << "  Double-Sum (" << n << ") = " << sum_k << endl;
 cout << endl;
 return 0;
}
```

Ex442.cpp

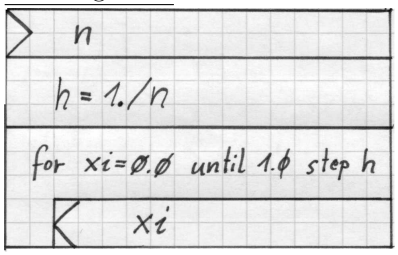
Ex443.cpp

Ex444.cpp

Weitere einfache **Beispiele** berechnen die Summe der ersten geraden natürlichen Zahlen und das Zählen eines CountDowns.

Die folgenden Beispiele verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tips zu deren Umgehung.

Struktogramm:



Loop_Float.cpp

Beispiel: Ausgabe der Stützpunkte x_i des Intervalls $[0, 1]$, welches in n gleichgroße Teilintervalle zerlegt wird, d.h.,

$$x_i = i \cdot h \quad , i = 0, \ldots, n \qquad \text{mit } h = \frac{1-0}{n}$$

Listing 4.10: Aufpassen bei Zählzyklen mit Gleikommagröße als Laufvariable

```
int main()
{
 int n;
 cin  >> n;                                  //  # subintervals
 const float  xa = 0.0e0,                    //  # start interval
              xe = 1.0e0,                    //  # end interval
       xstride = (xe-xa)/n;                  //  length subinterval
 float xi;
```

```
 int i = 0;
 for (xi = xa; xi <= xe+xstride/10; xi += xstride)
   {
    cout << " node " << i << " : " << xi << endl;
    ++i;
   }
 return 0;
}
```

Loop_Float.cpp

Da Gleitkommazahlen nur eine limitierte Anzahl gültiger Ziffern besitzen, kann es (oft) passieren, daß der letzte Knoten x_n nicht ausgegeben wird. Nur für $n = 2^k$, $k \in \mathbb{N}$, $k < 32$ kann in unserem Beispiel eine korrekte Abarbeitung des Zählzyklus garantiert werden. Auswege sind:

1. Änderung des Abbruchtests in `xi <= xe + h/2.0` , jedoch ist x_n immer noch fehlerbehaftet.
   ```
   for (xi = xa; xi <= xe + h/2.0; xi += h)
     {
      cout << xi << endl;
     }
   ```

2. Besser ist ein Zählzyklus mit einer `int`-Laufvariable wodurch die Werte $x_i = i \cdot h$ für große i genauer sind als in Variante 1.
   ```
   for (i = 0; i <= n; ++i)
     {
      xi = xa + i*h;
      cout << xi << endl;
     }
   ```

Die gemeinsame Summation kleinerer und größerer Zahlen kann ebenfalls zu Ungenauigkeiten führen. Im **Beispiel** wird die Summe $s1 := \sum_{i=1}^{n} 1/i^2$ mit der (theoretisch identischen) Summe $s2 := \sum_{i=n}^{1} 1/i^2$ für große n (65.000, 650.000) verglichen.

Listing 4.11: Auslöschung bei Summation kleiner Zahlen

```
#include <cmath>                          // M_PI
#include <iostream>
using namespace std;
int main()
{
 int n ;
 cin >> n;                                // Try  n = 65000;
//              Calculate in incrementing order
 float s1 = 0.0f;
 for (int i=1; i<=n; ++i)
   {
     s1 += 1.0f/i/i;
   }
//              Calculate in decrementing order
 float s2 = 0.0f;
 for (int i=n; i>=1; --i)
   {
     s2 += 1.0f/i/i;
//     s2 += 1.0f/(i*i);             // results in inf
//                                 since i*i may be longer than int supports
   }
 return 0;
}
```

Reihe.cpp

Das numerische Resultat in $s2$ ist genauer, da dort zuerst alle kleinen Zahlen addiert werden, welche bei $s1$ wegen der beschränkten Anzahl gültiger Ziffern keinen Beitrag zur Summation mehr liefern können. Gleichzeitig ist zu beachten, daß die Berechnung von `1.0/(i*i)` in einem Überlauf endet, da `i*i` nicht mehr in `int`-Zahlen darstellbar ist. Dagegen erfolgt die Berechnung von `1.0/i/i` vollständig im Bereich der Gleitkommazahlen.

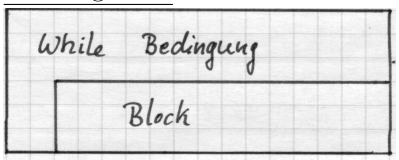
4.5 Abweisender Zyklus (`while`-Schleife)

Beim abweisenden Zyklus steht die Anzahl der Durchläufe nicht a-priori fest, der Abbruchtest erfolgt **vor** dem Durchlauf eines Zyklus.

Die allgemeine Form ist

```
while (<logischer ausdruck>)
    <anweisung>
```

Struktogramm:



Beispiel: Bestimme den aufgerundeten Binärlogarithmus (Basis 2) einer einzulesenden Zahl.

Listing 4.12: Ganzzahliger Anteil des Binärlogarithmus einer Zahl

```
#include <iostream>
using namespace std;
int main()
{
 double x;
 cin  >> x;
 const double xsave = x;            // Save to restore x
 int cnt = 0;                       // Initialize loop index !!
 while ( x > 1.0 )
   {
     x   = x/2.0 ;
     cnt = cnt + 1;
   }
 return 0;
}
```

Ex450.cpp

Bemerkung: Falls der allererste Test im abweisenden Zyklus `false` ergibt, dann wird der Anweisungsblock im Zyklusinneren nie ausgeführt (der Zyklus wird abgewiesen).

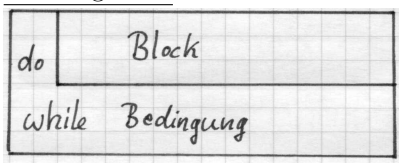
4.6 Nichtabweisender Zyklus (`do-while`-Schleife)

Beim nichtabweisenden Zyklus steht die Anzahl der Durchläufe nicht a-priori fest, der Abbruchtest erfolgt **nach** dem Durchlauf eines Zyklus. Somit durchläuft der nichtabweisende Zyklus mindestens einmal die Anweisungen im Zyklusinneren.

Die allgemeine Form ist

```
do
    <anweisung>
while (<logischer ausdruck>) ;
```

Struktogramm:



Beispiel: Es wird solange ein Zeichen von der Tastatur eingelesen, bis ein x eingegeben wird.

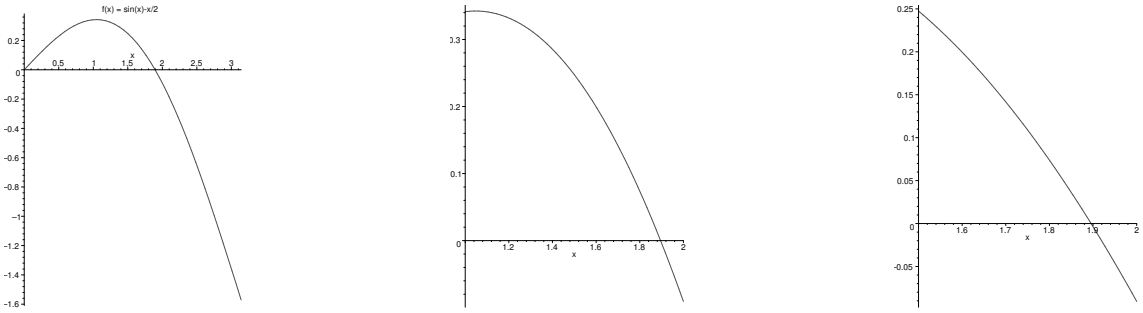
Listing 4.13: Zeicheneingabe bis zum Exit-Zeichen `x`

```
#include <iostream>
using namespace std;
int main()
{
 char ch;
 do
   {
     cout << endl << "Input command (x = exit, ...)  ";
     cin  >> ch;
   }
 while ( ch != 'x' );
 return 0;
}
```

Ex460.cpp

Betrachten wir ein etwas anspruchsvolleres **Beispiel**, und zwar soll die Lösung von $\sin(x) = x/2$ mit $x \in (0, \pi)$ bestimmt werden. Hierzu betrachtet man die äquivalente Nullstellenaufgabe: Bestimme die Nullstelle $x_0 \in (0, \pi)$ der Funktion $f(x) := \sin(x) - x/2 = 0$.

Analytisch: Kein praktikabler Lösungsweg vorhanden.
Graphisch: Die Funktion $f(x)$ wir graphisch dargestellt und das Lösungsintervall manuell verkleinert (halbiert). Diesen Prozeß setzt man so lange fort, bis x_0 genau genug, d.h., auf eine vorbestimmte Anzahl von Stellen genau, bestimmt werden kann.



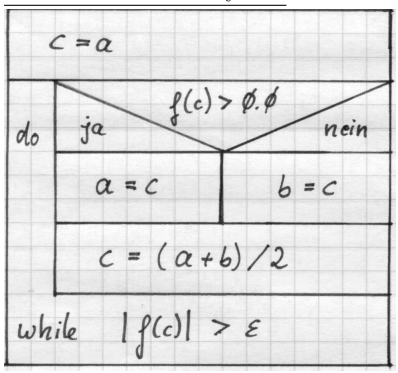
Numerisch: Obiges, graphisches Verfahren kann auf ein rein numerisches Verfahren im Computer übertragen werden (der *MAPLE*-Aufruf `fsolve(sin(x)=x/2,x=0.1..3` liefert als Näherungsergebnis $x_0 = 1.895494267$). Wir entwickeln ein Programm zur Bestimmung der Nullstelle von $f(x) := \sin(x) - x/2$ im Intervall $[a, b]$ mittels Intervallhalbierung, wobei zur Vereinfachung angenommen wird, daß $f(a) > 0$ und $f(b) < 0$ ist. Der Mittelpunkt des Intervalls sei mit $c := (a+b)/2$ bezeichnet. Dann können wir über die Lösung Folgendes aussagen: $\begin{cases} x_0 := c & \text{falls } f(c) = 0 \\ x_0 \in [c, b] & \text{falls } f(c) > 0 \\ x_0 \in [a, c] & \text{falls } f(c) < 0 \end{cases}$.

Ex462.mws

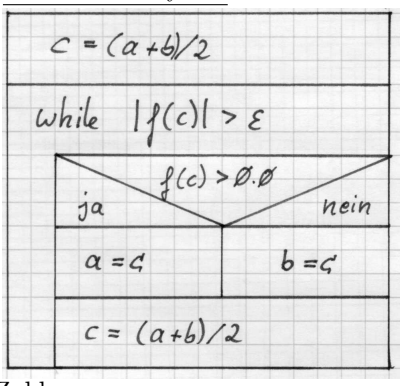
Durch Redefinition der Intervallgrenzen a und b kann die Nullstellensuche auf das kleinere (halbierte) Intervall reduziert werden. Wir demonstrieren die Umsetzung mittels eines nichtabweisenden Zyklus.

Ex462.cpp

nichtabweisender Zyklus:



abweisender Zyklus:



Wir realisieren obige Bisektion als nichtabweisenden Zyklus.

Listing 4.14: Bisektion als nichtabweisender Zyklus

```
#include <cmath>
#include <iostream>
using namespace std;
int main()
{
  const double Eps = 1e-6;
  double a,b,c,fc;
  cout << " f(a) > 0,  a : "; cin >> a;
  cout << " f(b) < 0,  b : "; cin >> b;
//              Do-While loop
  c  = a;                  // since f(a) > 0
  fc = sin(c)-c/2;
  do
   {
    if ( fc > 0.0 )
     {
       a = c;
     }
    else
     {
       b = c;
     }
    c  = (a+b)/2.0;
    fc = sin(c)-c/2;
   }
  while ( std::abs(fc) > Eps);
```

```
27 //  while ( std::abs(fc) != 0.0);        // endless!! Why ?
   return 0;
29 }
```

Ex462.cpp

Da Gleitkommazahlen nur mit limitierter Genauigkeit arbeiten, resultiert ein Abbruchtest $f(c) = 0$ meist in einem endlosen Programm. Dem ist ein Abbruchtest wie $|f(c)| < \varepsilon$ mit einer vorgegebenen Genauigkeit $0 < \varepsilon \ll 1$ vorzuziehen.

Bemerkung: Zählzyklen (`for`), welche mindestens einen Zyklus ausführen, können sowohl durch abweisende (`while`) als auch durch nichtabweisende Zyklen (`do while`) äquivalent ausgedrückt werden. Diese Äquivalenz kann bei Verwendung der Anweisungen in § 4.8 verloren gehen. Falls in einem Zählzyklus der Abbruchtest stets `false` ergibt, d.h. der Schleifenkörper wird nie ausgeführt, dann ist der entsprechende abweisende Zyklus nach wie vor äquivalent. Jedoch ist der nichtabweisende Zyklus nicht mehr äquivalent, da der dortige Schleifenkörper auch in diesem Fall einmal abgearbeitet wird.

Loops.cpp

4.7 Mehrwegauswahl (`switch`-Anweisung)

Die Mehrwegauswahl ermöglicht ein individuelles Reagieren auf spezielle Werte einer Variablen.

```
switch (<ausdruck>)
 {
   case <konst_ausdruck_1> :
        <anweisung_1>
[break;]
   ...
   case <konst_ausdruck_n> :
        <anweisung_n>
[break;]
   default:
        <anweisung_default>
 }
```

Beispiel: Ausgabe der Zahlwörter für die ganzzahlige Eingaben $\{1, 2, 3\}$.

Listing 4.15: Demonstration der Switch-Anweisung

```
#include <iostream>
2 using namespace std;
int main()
4 {
 int number;
6 //  number = 2,
 cin >> number;
8 cout << endl << "  Print names of numbers from interval [1,3]" << endl;
 switch(number)
10  {
    case 1:
12       cout << "  One = " << number << endl;
//       break;
14   case 2:
       cout << "  Two = "  << number << endl;
16     break;                          // Comment this line
   case 3:
18     cout << "  Three = "  << number << endl;
       break;
20   default:
       cout << "  Number " << number << " not in interval" << endl;
22     break;
  }
24 cout << endl;
 return 0;
26 }
```

Ex470.cpp

Obige `switch`-Anweisung könnte auch mit einer Mehrfachverzweigung (Seite 26) implementiert werden, jedoch werden in der `switch`-Anweisung die einzelnen Zweige explizit über die `break;`-Anweisung verlassen. Ohne `break;` wird zusätzlich der zum nachfolgenden Zweig gehörige Block abgearbeitet.

Es ist in C++ *nicht* möglich, *Bereiche* anzugeben, etwa der Art `case 7..11: Anweisung; break;` anstelle der korrekten C++-Vorgangsweise [Bre17, §1.8.4]

```
case 7: case 8: case 9: case 10: case 11:
   Anweisung;
   break;
```

4.8 Anweisungen zur unbedingten Steuerungsübergabe

`break` Es erfolgt der sofortige Abbruch der nächstäußeren `switch`, `while`, `do-while`, `for` Anweisung.

`continue` Abbruch des aktuellen und Start des nächsten Zyklus einer `while`, `do-while`, `for` Schleife. Ex480.cpp

`goto <marke>` Fortsetzung des Programmes an der mit
`<marke> : <anweisung>`
markierten Stelle.

Bemerkung : Bis auf `break` in der `switch`-Anweisung sollten obige Anweisungen sehr sparsam (besser gar nicht) verwendet werden, da sie dem strukturierten Programmieren zuwiderlaufen und im Extremfall einen gefürchteteten Spaghetticode erzeugen. Wenn man das strukturierte Programmieren gut beherrscht, dann kann die gezielte Verwendung von `break` und `continue` zu schnellerem Code führen.

Obige Anweisungen aus §4.8 sind im Rahmen dieser LV zur Lösung von Übungsaufgaben und im Test nicht erlaubt.

Kapitel 5

Strukturierte Datentypen

Wir werden in diesem Kapitel neue Möglichkeiten der Datenspeicherung einführen.

- **Feld (array)**, **Liste**:
  Zusammenfassung von Elementen gleichen Typs.
- **Struktur (struct)**:
  Zusammenfassung von Komponenten verschiedenen Typs.
- **Union (union)**:
  Überlagerung mehrerer Komponenten verschiedenen Typs auf dem gleichen Speicherplatz.
- **Aufzählungstyp (enum)**
  Grunddatentyp mit frei wählbarem Wertebereich.

Eine darüber hinausgehende, sehr gute Einführung in Algorithmen und Datenstrukturen ist in [PD08] zu finden.

5.1 Felder

In einem Feld/Array werden Daten (Elemente) gleichen Typs zusammengefaßt. Die klassischeVereinbarung eines statischen Feldes ist in C (geht auch in C++)

```
<typ> <bezeichner>[dimension];
```

wobei die eckigen Klammern "[" und "]" unabdingbarer Bestandteil der Vereinbarung sind. Ein eindimensionales Feld entspricht mathematisch einem Vektor. Hierbei wird zwischen *statisch*en Feldern (Länge des Feldes ist zur Compilezeit bekannt) und *dynamisch*en Feldern (Feldlänge kann erst aus Daten des Programmes bestimmt werden bzw. ändert sich während des Programmablaufes) unterschieden.

Listing 5.1: Statisches C-Array

```
1 {
  const int N=5;              // constant dimension for array
3 double x[N]= {9,7,6,5,7};
  x[0] = 1.0;                 // index starts with 0
5 x[1] = -2;
  x[2] = -x[1];
7 x[3] = x[1]+x[2];
  x[4] = x[1]*x[2];
9 // Access to  x[5]  is not permitted: out of range !!
}
```

Ex510.cpp

Die eckigen Klammern dienen im Vereinbarungsteil der Dimensionsvereinbarung `x[N]` und im Anweisungsteil dem Zugriff auf einzelne Feldelemente `x[3]` . Das Feld kann schon bei Deklaration initialisiert werden:

```
double x[N] = {9,7,6,5,7}
```

Achtung : Die Numerierung der Feldelemente beginnt mit 0. Daher darf nur auf Feldelemente x_i, $i = 0, \ldots, N-1$ zugegriffen werden. Andernfalls sind mysteriöses Programmverhalten, unerklärliche Fehlberechnungen und plötzliche Programmabstürze zu erwarten, deren Ursache nicht offensichtlich ist da sie eventuell erst in weit entfernten Programmteilen auftreten können. Diese Fehler müssen dann mit *Memory-Checkern* mühsam gesucht werden. Ein gutes und freies Programm hierfür ist `valgrind` unter Linux/Unix bzw. der *inspector* enthalten in der Intel Toolbox (freie Studentenversion).

Wir werden im Weiteren **nicht die C-Arrays benutzen, sondern die nachfolgenden C++-Vektoren**. Deren Deklaration ist unterschiedlich von oberer Deklaration, der Zugriff kann identisch erfolgen und die C++-Vektoren haben mehr Funktionalität.

5.1.1 Dynamischer C++-Vektor

Wir illustrieren den C++-Vektor `vector <T>`[1] mit dem Beispiel der Berechnung der L_2-Norm eines Vektors, d.h., $\| \underline{x} \|_{L_2} := \sqrt{\sum_{i=0}^{N-1} x_i^2}$.

Listing 5.2: Berechnung der L_2-Norm eines Vektors

```
1  #include <cmath>                          //       sqrt()
   #include <iostream>
3  #include <vector>                         //       vector<>
   using namespace std;
5  int main()
   {
7      unsigned int n;
       cout << "\n Anzahl der Vektorelemente : ";
9      cin >> n;                             // Dynamische Laenge des Vektors
       vector<double> x(n);                  // Deklariere Vektor dieser Laenge
11     for (size_t k=0; k<x.size(); ++k) {    // Belege den Vektor mit Daten
           x.at(k) = 1.0/(k+1.0);
13     }
       double norm = 0.0;
15     for (size_t k=0; k<x.size(); ++k) {    // Normberechnung
           norm += x[k]*x[k];
17     }
       norm = sqrt(norm);
19     cout << "\n Norm : " << norm << endl;
       return 0;
21 }
```

bsp511a.cpp

Ein paar Bemerkungen zu Listing 5.2:

- Zeile 10: Diese Deklaration einer Variablen vom Typ `vector<double>` enthält die Anzahl der Elemente, welche zur Compilezeit nicht feststand.
- Die Länge von Vektor `x` läßt sich mit `x.resize(n+4)` im Programmablauf ändern (auch auf einen kürzeren Vektor).
- Zeile 11: Die Länge des Vektors kann über die Methode `x.size()` abgefragt werden, d.h., diese Information ist Teil des C++-Vektors.
- Zeile 16: Der Zugriff auf Element k erfolgt über `x[k]` am schnellsten. Ein unzulässiger Zugriff `x[n]` ($n \notin [0, x.size()]$) wird jedoch nicht vom Programm abgefangen und führt zum Programmabbruch oder (schlimmer) zu nicht nachvollziehbarem Verhalten des Programmes.
- Zeile 12: C++-Vektoren erlauben den gesicherten Zugriff auf Vektorelemente über `x.at(k)`. Im Falle von $k \notin [0, x.size()]$ bricht das Programm mit einer Fehlermeldung ab.

Mit obigem Code haben wir die dynamischen Möglichkeiten von `vector` noch nicht voll ausgeschöpft. Dazu betrachten wir die gleiche Berechnung wie oben, werden aber die Anzahl der Komponenten an hand der einzugebenden Daten bestimmen.

[1] `http://www.cplusplus.com/reference/vector/vector/`

Listing 5.3: Mehr Dynamik beim Vektor

```
#include <cmath>                          //      sqrt()
#include <iostream>
#include <vector>                         //      vector<>
using namespace std;
int main()
{
    vector<double> x;             // Deklariere Vektor mit Laenge 0
    do {                          // Verlaengere den Vektor mit Eingabedaten
        double tmp;
        cout << endl << "Zahl: "; cin >> tmp;
        x.push_back(tmp);                 // Verlaengere den Vektor
    } while( x.back() >= 0.0 ); // solange kein negatives Elemement entsteht
    x.pop_back();                 // Entferne das letzte, negative Element
    double norm = 0.0;
    for (size_t k=0; k<x.size(); ++k) {     // Normberechnung
        norm += x[k]*x[k];
    }
    norm = sqrt(norm);
    cout << "\n Lange: " << x.size() << "   letztes Element : " <<  x.back();
    cout << "\n Norm : " << norm << endl;
    return 0;
}
```

bsp511b.cpp

In Listing 5.3 sind ein paar interessante Details untergebracht:

- Zeile 7: Der Vektor hat die Länge 0.
- Zeilen 8-13: Es werden solange Daten eingegeben, bis die Eingabegröße < 0 ist. Dabei wächst der Vektor in jedem Durchlauf um ein Element.
  - Zeile 11: Die Methode `push_back`[2] fügt den übergebenen Wert `tmp` als neues letztes Element an den Vektor an.
  - Zeile 12: Die Methode `back` greift auf das aktuelle, letzte Element des Vektors zu (und testet, ob dieses nichtnegativ ist).
  - Zeile 13: Die Methode `pop_back` entfernt das negative letzte Element und verkürzt den Vektor entsprechend.
- Zeile 15: Die Methode `size` liefert die Vektorlänge normalerweise als `unsigned int` zurück (kann aber auch anders definiert sein!). Wenn die Loopvariable `k` nun als `int` deklariert ist, werden beim Vergleich `k<x.size()` unterschiedliche Datentypen verglichen. Dies ist im Normalfall kein Problem, führt aber zu lästigen Compilerwarnungen. Exakterweise nimmt man daher `size_t` als Datentyp der Loopvariablen.

5.1.2 Statischer C++-Vektor

Natürlich kann man den dynamischen Vektor `vector<T>` auch als statischen Vektor benutzen. Dies ist am Anfang auch die einfachste Lösung.

Um aber die volle konzeptionelle Kompatibilität zum klassischen C-Vektor herzustellen, wurde mit dem C++11-Standard der statische Vektor `array<T,N>`[3] eingeführt bei welchem zur Compilezeit nicht nur der Datentyp `T` sondern auch die Länge `N` festgelegt sind. Dynamische Methoden sind für diesen Vektor nicht verfügbar.

Listing 5.4: Berechnung der L_2-Norm eines statischen C++-Vektors

```
#include <array>                          //      array<T,N>
#include <cmath>                          //      sqrt()
#include <iostream>
using namespace std;
int main()
{
    array<double,10> x;                   // Statischer Vektor der Lange 10
    for (size_t k=0; k<x.size(); ++k) { // Belege den Vektor mit Daten
        x.at(k) = 1.0/(k+1.0);
    }
    double norm = 0.0;
    for (size_t k=0; k<x.size(); ++k) {  // Normberechnung
```

[2] http://www.cplusplus.com/reference/vector/vector/push_back/
[3] http://www.cplusplus.com/reference/array/array/

```
        norm += x[k]*x[k];
14    }
    norm = sqrt(norm);
16    cout << "\n Lange: " << x.size() << "   letztes Element : " << x.back();
    cout << "\n Norm : " << norm << endl;
18    return 0;
}
```

bsp511c.cpp

Im Listing 5.4 hätte man in Zeile 7 genauso einen dynamischen Vektor nehmen können (`vector<double> x(10)`), bis auf das zu inkludierende Headerfile bliebe der restliche Code unverändert.

5.1.3 Beispiele zu C++-Vektoren

Als kleines **Beispiel** diene uns die Fibonacci Zahlenfolge, welche über die zweistufige Rekursion

$$f(n)_{:} = f(n-1) + f(n-2) \qquad n = 2, \ldots$$

mit den Anfangsbedingungen $f(0) = 0$, $f(1) = 1$ definiert ist. Zur Kontrolle können wir die Formel von Binet bzw. de Moivre[4] verwenden.

$$f(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Listing 5.5: Fibonacci numbers

```
#include <vector>                    // vector<T>
using namespace std;
int main()
{
    const int N = 40;                // What happens with N=50 ?? Why ??
    vector<int> x(N+1);              // !!   N + 1   !!
    x[0] = 0;                        // Calculate Fibonacci numbers
    x[1] = 1;
    for ( int i = 2; i <= N; ++i ) {
        x.at(i) = x[i - 1] + x[i - 2];
    }
    return 0;
}
```

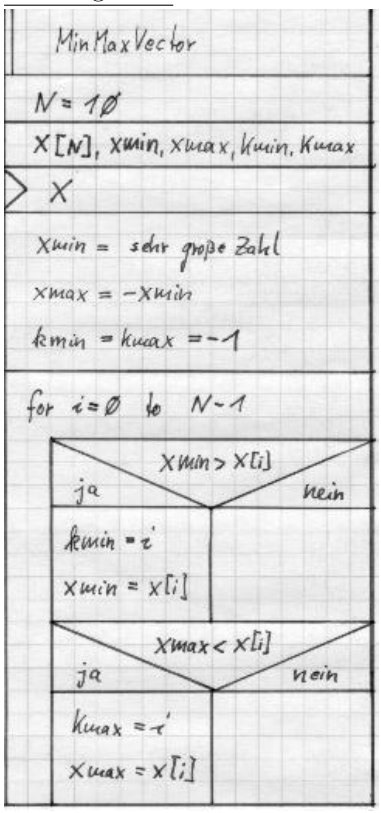
bsp_Fibo1.cpp

[4] http://de.wikipedia.org/wiki/Fibonacci-Folge#Formel_von_Moivre-Binet

Als weiteres **Beispiel** sollen Minimum und Maximum eines Vektors bestimmt und die entsprechenden Vektorelemente miteinander vertauscht werden (analog zu Pivotisierung). Dies beinhaltet die beiden Teilaufgaben:

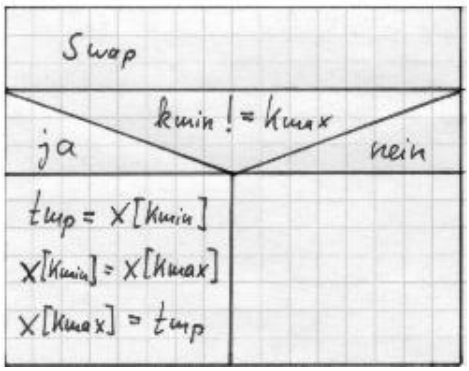
a) Bestimme Minimum und Maximum (und markiere die Positionen).

Struktogramm:



b) Vertausche Min/Max-Einträge.
Bei Vektorlänge 0 oder bei identischen Vektorelementen ist kein Vertauschen notwendig.

Struktogramm:



Beim Vertauschen führt die naheliegende, erste Idee

```
x[kmin] = x[kmax]
x[kmax] = x[kmin]
```

nicht zum Erfolg. Warum?

Listing 5.6: Bestimme Min/Max eines Vektor und vertausche die Komponenten

```
#include <limits>                       // numeric_limits
#include <vector>                       // vector<T>
using namespace std;
int main()
{
    vector<double> x(10);
//                                      Initialize x
//                                      ...
//                                      Initialize min/max values
    double xmin =  numeric_limits<double>::max(),
           xmax = -xmin;
//                                      Initialize indices storing positions for min/max
    int kmin = -1, kmax = -1;
//                                      Determine min/max
    for (size_t i = 0; i < x.size(); ++i) {
        if ( xmin > x[i] ) {
            xmin = x[i];
            kmin = i;
        }
        if ( xmax < x[i] ) {
            xmax = x[i];
            kmax = i;
        }
    }
//                                  Swap Pivot elements
//                          Do nothing for N=0 or constant vector
    if ( kmax != kmin ) {
        double tmp = x[kmin];
        x[kmin] = x[kmax];
        x[kmax] = tmp;
    }
    return 0;
}
```

bsp513.cpp

In obigen beiden Listing wurden keine dynamischen Eigenschaften des Vektors benutzt, daher hätten wir auch `array<T,N>` verwenden können.

5.1.4 Mehrdimensionale Felder in C++

Statische Matrizen

Die Elemente der bisher betrachteten 1D-Felder sind im Speicher hintereinander gespeichert (Modell des linearen Speichers), z.B, wird der Zeilenvektor

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 \end{pmatrix}$$

als

```
double x[5];
```

vereinbart und gespeichert als

x_0	x_1	x_2	x_3	x_4

wobei jede Zelle 8 Byte lang ist.

Ein zweidimensionales (statisches) Feld, z.B., eine Matrix A mit $N = 4$ Zeilen und $M = 3$ Spalten

$$A_{N\times M} := \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \\ A_{30} & A_{31} & A_{32} \end{pmatrix}$$

kann im Speicher ebenfalls nur linear gespeichert werden, d.h.,

A_{00}	A_{01}	A_{02}	A_{10}	A_{11}	A_{12}	A_{20}	A_{21}	A_{22}	A_{30}	A_{31}	A_{32}

Daraus ergeben sich zwei Möglichkeiten der 2D-Feldvereinbarung:

- Variante 1 : Als 2D-Feld.
```
double A[N][M];                 // Declaration in C
array<array<double,M>,N> A;     // Declaration in C++ (statisch)
A[3][1] = 5.0;                  // Initialize A(3,1)
```
- Variante 2 : Als 1D-Feld.
```
double A[N*M];                  // Declaration in C
array<double,M*N> A;            // Declaration in C++ (statisch)
A[3*M+1] = 5.0;                 // Initialize A(3,1)
```

Beispiel: Als Beispiel betrachten wir die Multiplikation der Matrix $A_{N\times M}$ bestehend aus $N = 4$ Zeilen und $M = 3$ Spalten mit einem Zeilenvektor $\underline{u}_M$ der Länge M. Das Ergebnis ist ein Zeilenvektor $\underline{f}_N$ der Länge N, d.h., $\underline{f}_N := A_{N\times M} \cdot \underline{u}_M$. Die Komponenten von $\underline{f} = [f_0, f_1, \ldots, f_{N-1}]^T$ berechnen sich zu

$$f_i := \sum_{j=0}^{M-1} A_{i,j} \cdot u_j \qquad \forall i = 0, \ldots, N-1 \ .$$

bsp514.cpp

In *bsp514.cpp* ist die Matrix in beiden Varianten jeweils über ein C++-`array` implementiert. Deklariert man das 2D-Feld via `array<array<T,MCOL>,NROW>` dann besteht die Chance, daß die Matrixelemente linear im Speicher abgelegt werden.

Dynamische Matrizen

Die entsprechenden Realisierungen mit dem C++-`vector` sind in *bsp514_b.cpp* verfügbar, wobei diese Implementierung bei dynamischer Matrixgröße ihre Stärken ausspielt. Bei der Realisierung der Matrix als 2D-Feld via `vector<vector<T>>` werden die Matrixelemente nicht mehr linear im Speicher abgelegt und der Zugriff erfolgt wie in Variante 1.

Listing 5.7: Dynamisches Allokieren einer Matrix

```
1 //  dynamic matrix allocation with 2D array, initialize with 0
      int nrow{101};                  // #rows
3     int mcol{49};                   // #columns
      vector<vector<float>>  C(nrow, vector<float>(mcol, 0.0f));
5     cout << C[17][33] << endl;      // access element
      cout << C.at(17).at(33) << endl; // access element
```

bsp514_b.cpp

Dynamische Tensoren

Höherdimensionale Felder können in Variante 1 analog zu Listing 5.7 deklariert und benutzt werden. In Variante 2 muß auf ein Element $B_{i,j,k}$ eines dreidimensionalen Tensors `vector<double> B(L*N*M);` mittels `B[i*M*N+j*M+k]` zugegriffen werden.

5.2 Liste

Neben dem Vektor ist die Liste ein häufig benutztes Konstrukt zum Speichern gleichartiger Argumente. Auf Listen kann nicht, wie bei Vektoren, wahlfrei über einen Index zugegriffen werden (also `x[k]` geht nicht), sondern es muß auf die Listenelemente stets in eine Richtung nacheinander zugegriffen werden.

Theoretisch sind Listen bei Sortieroperationen effizienter als Vektoren. Konkret hängt diese Aussage sehr stark vom Speicherbedarf der Elemente ab da jedes Element einer Liste intern zusätzlich 2 Pointer benötigt und somit das Sortieren von Vektoren durch den geringeren Speicherdurchsatz schneller sein kann.

Listing 5.8: Berechnung mit einer Liste

```
#include <cmath>                          //       sqrt()
#include <iostream>
#include <list>                           //       list<>
using namespace std;
int main()
{
    list<double> x;                 // Deklariere Liste der Laenge 0
    do {                            // Verlaengere die Liste mit Eingabedaten
        double tmp;
        cout << endl << "Zahl: "; cin >> tmp;
        x.push_back(tmp);                   // Verlaengere die Liste
    } while( x.back() >= 0.0 ); // solange kein negatives Elemement entsteht
    x.pop_back();               // Entferne das letzte, negative Element
    double norm = 0.0;
    for (auto pi=x.begin(); pi != x.end(); ++pi) {    // Normberechnung
        norm += *pi * *pi;
    }
    norm = sqrt(norm);
    cout << "\n Lange: " << x.size() << "   letztes Element : " <<  x.back();
    cout << "\n Norm : " << norm << endl;
    return 0;
}
```

bsp511b-list.cpp

- Zeile 7: Die Liste hat die Länge 0.
- Zeilen 8-13: Es werden solange Daten eingegeben, bis die Eingabegröße < 0 ist. Dabei wächst die Liste in jedem Durchlauf um ein Element.
  - Zeile 11: Die Methode `push_back`[5] fügt den übergebenen Wert `tmp` als neues letztes Element an die Liste an.
  - Zeile 12: Die Methode `back` greift auf das aktuelle, letzte Element der Liste zu (und testet, ob dieses nichtnegativ ist).
  - Zeile 13: Die Methode `pop_back` entfernt das negative letzte Element und verkürzt die Liste entsprechend.
- Zeile 15-17: For-Loop mit einem Iterator (§6.2) um auf die Listenelemente zuzugreifen:
  - Zeile 15: Die Zuweisung `auto pi=x.begin()` initialisiert die Laufvariable `pi` mit dem Iterator auf das erste Listenelement. Der Typ von `pi` wäre korrekterweise `liste<double>::iterator`, das Schlüsselwort `auto` erspart uns diesen Bandwurm und setzt den Typ automatisch korrekt.
  - Zeile 16: `*pi` stellt das aktuelle Listenelement dar (der Iterator muß dereferenziert werden).

[5] `http://www.cplusplus.com/reference/list/list/push_back/`

5.3 Strukturen als einfache Klassen

Die Struktur definiert einen neuen Datentyp welcher Komponenten unterschiedlichen Typs vereint. Die Typdeklaration

```
struct <struct_bezeichner>
 {
   <Datendeklaration>
 };
```

erlaubt die Deklaration von Variablen diesen Typs

```
<struct_bezeichner> <var_bezeichner>;
```

Beispiel: Wir deklarieren einen Datentyp zur Speicherung der persönlichen Daten eines Studenten.

Listing 5.9: Deklaration und Nutzung einer Struktur

```
#include <iostream>
#include <string>                          // class string
using namespace std;
struct Student {                           // new structure
    Student(): matrikel(), skz(), name(), vorname() {}
    long long int matrikel;
    int skz;
    string name, vorname;
};
int main()
{
    Student arni, robbi;                   // Variable of type Student
    cout << endl << " Vorname : ";
    cin >> arni.vorname;                   // Data input
    cout << endl << " Familienname : ";
    cin >> arni.matrikel;
    robbi = arni;                          // Copy (!! shallow copy vs. deep copy !!)
    cout << robbi.vorname << " "   << robbi.name    << ", SKZ: ";
    cout << robbi.skz     << "   " << robbi.matrikel << endl << endl;
    return 0;
```

bsp520.cpp

Die Zuweisung `robbi = arni;` kopiert den kompletten Datensatz von einer Variablen zur anderen. Dies funktioniert in Listing 5.9 automatisch, da z.B. `robbi.name = arni.name;` den erforderlichen Speicherplatz allokiert und den gesamten Inhalt des Strings kopiert. Diese *deep copy* funktioniert bei allen Standardatentypen und allen Containern der STL. **Achtung** bei Zeigern in der Struktur/Klasse. Hier muß die *deep copy* selbst implementiert werden, da für die Zeiger nur eine *shallow copy* sttattfindet, siehe dazu [Bre17, p. 238f].

Der Zugriff auf die Komponente `vorname` der Variablen `arni` (des Typs `Student`) erfolgt über `arni.vorname`

Abgespeichert werden die Daten in der Form

matrikel	skz	name	vorname

Abhängig von Compilereinstellungen bzw. -optionen können kleinere ungenutzte Speicherlücken zwischen den Komponenten im Speicher auftreten (Data Alignment für schnelleren Datenzugriff).

Die Struktur `Student` kann leicht für Studenten, welche mehrere Studienrichtungen belegen, erweitert werden.

Listing 5.10: Struktur mit dynamischem Vektor als Komponente

```
#include <iostream>
#include <string>                          // class string
#include <vector>
using namespace std;
struct Student_Mult{                       // new structure
    Student_Mult(): matrikel(0), skz(0), name(), vorname() {}
    long long int matrikel;
    vector<int> skz;                       // dynamic vector
    string name, vorname;
};
int main()
{
    cout << endl << " # skz : ";
    int n_skz;
    cin >> n_skz;
    for (int i = 0; i < n_skz; ++i) {
```

```
        cout << endl << " Studentenkennzahl : ";
18      int tmp; cin >> tmp;
        arni.skz.push_back(tmp);        // append new number to dynamic vector
20  cout << robbi.vorname << " "   << robbi.name    << "   "<< robbi.matrikel << ",
        SKZ: ";
    for (size_t i = 0; i < robbi.skz.size(); ++i) {
22      cout << robbi.skz[i] << "   ";
    }
24  return 0;
```

bsp520b.cpp

Die Struktur `Student` enthält bereits Felder als Komponenten. Andererseits können diese Datentypen wiederum zu Feldern arrangiert werden.

Listing 5.11: Dynamischem Vektor mit Strukturelementen

```
1  #include <iostream>
   #include <string>                    // class string
3  #include <vector>                    //       vector
   using namespace std;
5  struct Student {
       Student(): matrikel(0), skz(0), name(), vorname() {}
7      long long int matrikel;
       int skz;
9      string name, vorname;
   };
11 int main()
   {
13     cout << "\n How many Students : ";
       unsigned int n;
15     cin >> n;                                // input #students
       gruppe.resize(n);                      // 4 students (dynamic vector!)
17     for (size_t i = 0; i < gruppe.size(); ++i) {
           cout << endl << "Student nr. " << i << " : " << endl;
19         cout << "Familenname : ";
           cin  >> gruppe[i].name;          // [i]     no range check for index
21         cout << "Vorname     : ";
           cin  >> gruppe.at(i).vorname;    // at(i) with range check for index
23         cout << "Matrikelnummer : ";
           cin  >> gruppe[i].matrikel;
25         cout << "SKZ            : ";
           cin  >> gruppe[i].skz;
27     cout << gruppe[idx].matrikel << " , " << gruppe[idx].skz << endl;
       return 0;
```

bsp522.cpp

Strukturen können wiederum andere strukturierte Datentypen als Komponenten enthalten.

Listing 5.12: Struktur mit Strukturkomponenten

```
#include <cmath>                        // sqrt()
2  #include <iostream>
   using namespace std;
4  struct Point3D {
       double x, y, z;                      // Point consists of coordinates
6  };
   struct Line3D {
8      Point3D p1, p2;                      // Line is defined by two points
   };
10 int main()
   {
12     Line3D line;                         // one line
       cout << "\n Input Line , Point 1 (x,y,z) : ";
14     cin  >> line.p1.x >> line.p1.y >> line.p1.z;
       cout << "              Point 2 (x,y,z) : ";
16     cin  >> line.p2.x >> line.p2.y >> line.p2.z;
       return 0;
18 }
```

bsp523.cpp

In obigem Beispiel ist `line.p2` eine Variable vom Typ `Point3D`, auf deren Daten wiederum mittels des `.` Operators zugegriffen werden kann.

5.4 *Union

Alle Komponenten der Union werden auf dem gleichen Speicherbereich überlappend abgebildet. Die Typdeklaration

```
union <union_bezeichner>
 {
   <Datendeklaration>
 };
```

erlaubt die Deklaration von Variablen diesen Typs

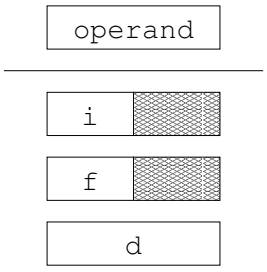
```
[union] <union_bezeichner> <var_bezeichner>;
```

Der Zugriff auf Komponenten der Union erfolgt wie bei einer Struktur.

Listing 5.13: Union

```
#include <iostream>
using namespace std;
union operand {
    int   i;
    float  f;
    double d;
};
int main()
{
    operand u;                          // Variable of union type
    cout << endl << "Size (operand) : " << sizeof(u) << " Bytes" << endl;
    u.i = 123;                          // Data input
    cout << endl << u.i << "  " << u.f << "  " << u.d << endl;
    u.f = 123;
    cout << endl << u.i << "  " << u.f << "  " << u.d << endl;
    u.d = 123;
    cout << endl << u.i << "  " << u.f << "  " << u.d << endl;
    return 0;
}
```

Ex530.cpp



Der Speicherplatzbedarf einer Union richtet sich nach der größten Komponente (hier `sizeof(double)` = 8). Die Union wird benutzt, um Speicherplatz zu sparen bzw. temporäre Felder für verschiedene Datentypen zu überlagern, sollte jedoch wegen der Fehlermöglichkeiten erfahrenen Programmierern vorbehalten bleiben (d.h. keine Verwendung im Praktikum).

5.5 *Aufzählungstyp

Der Aufzählungstyp ist ein Grundtyp mit frei wählbarem Wertebereich, dies sei an Hand der Wochentage veranschaulicht.

Listing 5.14: Enumeration-Typ für Wochentage

```
#include <iostream>
using namespace std;
enum tag {                              // day-of-the-week as enum type
    montag, dienstag, mittwoch, donnerstag, freitag, samstag, sonntag
};
int main()
{
    tag wochentag;                      // Variable of enum type
    wochentag = montag;                 // Data init
    wochentag = tag(wochentag + 1);     // next day; requires casting
    if ( wochentag == dienstag ) {
        cout << "Schlechte Laune " << wochentag << endl;
    }
    for (int i = montag; i <= sonntag; i = i + 1) {
        cout << "Tag " << i << endl;
    }
    return 0;
}
```

Ex540.cpp

5.6 *Allgemeine Typvereinbarungen

Die allgemeine Typdefinition

```
typedef <type_definition> <type_bezeichner>
```

ist die konsequente Weiterentwicklung zu frei definierbaren Typen.

Das nachfolgende Programmbeispiel illustriert die Definition der drei neuen Typen `Boolean`, `Text` und `Point3D`.

Listing 5.15: Typvereinbarungen

```
#include <cstring>
#include <iostream>
using namespace std;
                                    // new types
typedef short int Boolean;          // 2 Byte boolean (waste of memory)
typedef char      Text[100];        // Text is a C-String of length 99
typedef struct {                    // the same as struct Point3D
    double x, y, z;
} Point3D;
int main()
{
    Boolean a, b;
    Text    eintrag;
    Point3D pts[10];                // C-array of 10 points unitialized
    const Point3D  p = {1, 2, 3.45};// one point initialized
    strcpy(eintrag, "A beautiful code");//                  init
    pts[0] = p;
    a      = (p.x == pts[0].x);     // my boolean variable
    b      = !a;                    // my boolean variable
    return 0;
}
```

Ex550.cpp

Interessanterweise ist eine Variable vom Typ `Text` nunmehr stets eine Zeichenkettenvariable der (max.) Länge 99. Man beachte auch die Initialisierung der Variablen `p`. Damit kann sogar eine Konstante vom Typ `Point3d` deklariert und initialisiert werden.

C++11: Die neuere Möglichkeit der Definition eigener Typen nutzt `using`, z.B., in

Listing 5.16: C++-11 Aliases

```
                                    // new alias
using Boolean = short int;          // 2 Byte boolean (waste of memory)
using Text = char[100];             // Text is a C-String of length 99
using Point3D = struct {            // the same as struct Point3D
    double x, y, z;
};
```

Ex550_11.cpp

Kapitel 6

Referenzen und Pointer

6.1 Pointer (Zeiger)

Bislang griffen wir stets direkt auf Variablen zu, d.h., es war nicht von Interesse, wo die Daten im Speicher abgelegt sind. Ein neuer Variablentyp, der Pointer (Zeiger), speichert Adressen unter Berücksichtigung des dort abgelegten Datentyps.

6.1.1 Deklaration von Zeigern

Sei der Zeiger auf ein Objekt vom Typ `int` mit `p` bezeichnet, so ist

```
int *p;
```

dessen Deklaration, oder allgemein wird durch

```
<typ> *<bezeichner>;
```

ein Zeiger auf den Datentyp `<typ>` deklariert.
So können die folgenden Zeigervariablen deklariert werden

Listing 6.1: Pointerdeklarationen

```
1  int main()
   {
3      char            *cp;                // pointer on char
       int          x, *px;                // int-variable, pointer on int
5      float           *fp[20];            // C-array of 20 pointers on float
       float           *(fap[20]);         // pointer on a C-array of 20 float
7      Student         *ps;                // pointer on structure Student
       char           **ppc;               // pointer on pointer of char
9                                          //  ALL pointers are undefined yet !!
       px = nullptr;                       //  px = 0
11     if ( px == nullptr) {
           cout << endl << "px == nullptr : " << px << endl;
13     }
       return 0;
15 }
```

Ex610.cpp

6.1.2 Zeigeroperatoren

Der unäre **Referenzoperator** (Adressoperator)

```
&<variable>
```

bestimmt die Adresse der Variablen im Operanden.

Der unäre **Dereferenzoperator** (Zugriffsoperator)

```
*<pointer>
```

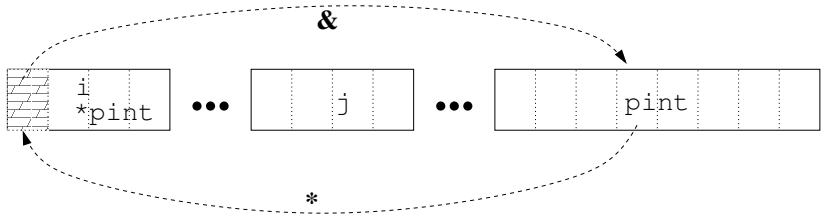
erlaubt den (indirekten) Zugriff auf die Daten auf welche der Pointer zeigt. Die Daten können wie eine Variable manipuliert werden.

Listing 6.2: Zugriff auf Variablen über Pointer

```
1 int main()
2 {
3     int i, j, *pint;
4     i     = 10;
5     pint  = &i;                    // pointer initialization
6     j     = *pint;                 // access on 'i' via pointer
7     *pint  = 0;                    // initialize 'i' by 0  via pointer
8     *pint += 2;                    // add 2 to 'i'  via pointers
9     return 0;
10 }
```

Ex620.cpp

In obigem Beispiel fungiert `*pint` als `int`-Variable und dementsprechend können auch alle dafür definierten Operationen mit ihr ausgeführt werden.



Achtung : In dem Programmfragment

```
{
  double *px;
  *px = 3.1;      // WRONG!
}
```

wird zwar Speicherplatz für den Zeiger reserviert (8 Byte), jedoch ist der Wert von `px` noch undefiniert und daher wird der Wert `3.1` in einen dafür nicht vorgesehenen Speicherbereich geschrieben $\Longrightarrow$ mysteriöse Programmabstürze und -fehler.

Es gibt eine spezielle Zeigerkonstante `nullptr` (`NULL` in C), welche auf die (hexadezimale) Speicheradresse `0x0` verweist und bzgl. welcher eine Zeigervariable getestet werden kann.

6.1.3 Zeiger und Felder - Zeigerarithmetik

Felder nutzen das Modell des linearen Speichers, d.h., ein im Index nachfolgendes Element ist auch physisch im unmittelbar nachfolgenden Speicherbereich abgelegt. Dieser Fakt erlaubt die Interpretation von Zeigervariablen als Feldbezeichner und umgekehrt.

```
{
  const int N = 10;
        int f[N], *pint;  // C-array and pointer

  pint = &f[0];           // init pointer
}
```

Feldbezeichner werden prinzipiell als Zeiger behandelt, daher ist die Programmzeile

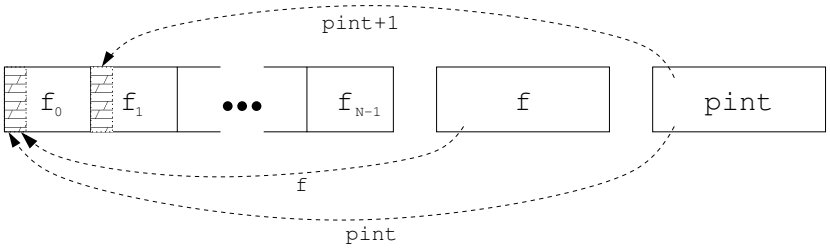
```
pint = &f[0];
```

identisch mit

```
pint = f;
```

Folgerichtig stellen daher die Ausdrücke `f[1]` , `*(f+1)` , `*(pint+1)` , `pint[1]` den identischen Zugriff auf das Feldelement f_1 dar.

Ex630.cpp

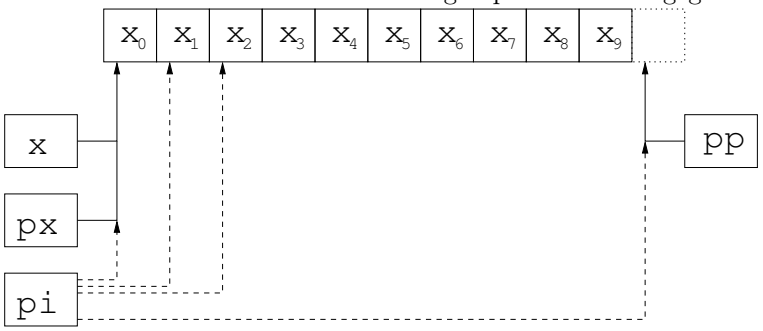


Die Adresse welche durch (`pint+1`) dargestellt wird ergibt sich zu (Adresse in `pint`) + `sizeof(int)` . Dabei bezeichnet `int` den Datentyp, auf welchen der Zeiger `pint` verweist. Der Zugriff auf andere Feldelemente f_i, $i = 0 \ldots N-1$ ist analog.

Die folgenden Operatoren sind auf Zeiger anwendbar:

- Vergleichsoperatoren: `==` , `!=` , `<` , `>` , `<=` , `>=`
- Addition `+` und Subtraktion `-`
- Inkrement `++` , Dekrement `--` und zusammengesetzte Operatoren `+=` , `-=`

Zur Demonstration betrachten wir ein **Beispiel**, in welchem ein Feld erst auf konventionelle Weise deklariert und initialisiert wird um danach mittels Zeigeroperationen ausgegeben zu werden.



Listing 6.3: Pointerarithmetik

```
#include <iostream>
using namespace std;
int main()
{
    const int N = 10;
    double    x[N], *px, *pp;
    px = &x[0];                          // address of x[0]
    pp = x;                              // address of C-array begin
    if ( px == pp) {
        cout << endl << " px and pp are identical" << endl;
    }
                                         // initialize x
    for (int i = 0; i < N; ++i ) {
        *(px + i) = (i + 1) * (i + 1);   // x[i] = ...
    }
    int i = 6;                           // check element 6
    cout << endl;                        // 4 different ways to access x[6]
    cout << x[i]    << endl;
    cout << *(x + i)  << endl;
    cout << px[i]   << endl;
    cout << *(px + i) << endl << endl;
                                        // for-loop with pointer as loop variable
    pp = x + N;                         // pointer to address after last element of x
    for ( double *pi = x; pi < pp; ++pi) {      // pointer pi as loop variable
        cout << "  " << *pi << endl;
    }
    return 0;
}
```

Ex630.cpp

In Listing 6.3 sind der Zählzyklus mit `i` als auch der Zählzyklus mit `pi` bzgl. der Zugriffe auf das C-Array `x` identisch.

6.2 Iteratoren

Das Konzept der Zeiger, welches direkt an fortlaufende Adressen im Speicher gebunden ist, wird in C++ durch das Konzept der *Iteratoren* erweitert. Dabei kann man sich die Iteratoren im einfachsten Fall von C++Feldern (`array`, `vector`) als Pointer vorstellen, jedoch spätestens bei Liste (`list`) muß man sich von dieser Simplifizierung lösen.

6.2.1 Iteratorenzugriff auf `array`

Listing 6.4: Iteratoren für C++-`array`

```
1  #include <array>                          // array<T,N>
   #include <iostream>
3  using namespace std;
   int main()
5  {
       array<double, 10>  x;                 // static array of length 10
7      double *px = &x[0];                   // address of x[0]
       double *pp = x.data();                // address of C++-array begin
9      if ( px == pp) {
           cout << endl << " px and pp are identical" << endl;
11     }
                                             // initialize x
13     for (size_t i = 0; i < x.size(); ++i ) {
           x[i] = (i + 1) * (i + 1);         // x[i] = ...
15     }
                                             // for-loop with pointer as loop variable
17     array<double, x.size()>::iterator pi; // declare iterator as loop variable
       for ( pi = x.begin(); pi != x.end(); ++pi) {
19         cout << "  " << *pi << endl;
       }
21     return 0;
   }
```

bsp631a.cpp

Das obige Listing is analog zum Listing 6.3 für ein C-array. Auf folgende Unterscheide sei hingewiesen:

- Zeile 8: Die Anfangsadresse der gespeicherten Daten ist nicht mehr `&x`, sondern `x.data()`. Der Ausdruck `&x` liefert einen Pointer vom Typ `array<double, 10>` zurück.
- Zeile 17: Die Laufvariable `pi` ist jetzt ein Iterator, speziell für `array<double, 10>`. Hätten wir ein zweites Array `array<double, 33>`, dann könnte Iterator `pi` nicht dafür verwendet werden (bei Pointern ginge dies.).

6.3 Referenzen

Referenzen sind ein Sprachmittel, um einen anderen Namen für dieselben Daten zu benutzen. In Zeile 6 des Listings 6.5 wird eine Referenz `ri` auf die Variable `i` deklariert. Danach ist ein Zugriff auf `ri` völlig äquivalent zu einem Zugriff auf `i`.

Listing 6.5: Referenz

```
#include <iostream>
2  using namespace std;
   int main()
4  {
       int   i;                          // variable 'i'
6      int &ri = i;                      // declaration reference on 'i'
       int *pi;
8      pi = &i;                          // define 'pi' as pointer on 'i';
       i = 7;
10     cout << endl << "      i = " << i
           << " ,   ri = " << ri
12         << " , *pi = " << *pi;
       ++ri;
14     cout << endl << "      i = " << i
           << " ,   ri = " << ri
16         << " , *pi = " << *pi;
       ++(*pi);
18     cout << endl << " ,    i = " << i
           << " ,   ri = " << ri
```

```
20          << " , *pi = " << *pi;
    cout << endl;
22  return 0;
}
```

bsp611.cpp

Wozu benötigt man eigenlich Referenzen?

- Um auf Teile aus einer größeren Struktur einfach zuzugreifen, *ohne diese kopieren* zu müssen.

  ```
  Student_Mult a;
  int &studium0 = a.skz[0];
  ```

- Als Kennzeichnung von Output-Parametern in der Parameterliste einer Funktion, siehe §7.3.
- Mit vorgesetztem `const` zur Kennzeichnung von reinen Input-Parametern in der Parameterliste einer Funktion, siehe §7.3. In diesem Falle werden die Daten nicht kopiert bei der Übergabe an die Funktion.

Kapitel 7

Funktionen

Zweck einer Funktion:

- Des öfteren wird ein Programmteil in anderen Programmabschnitten wieder benötigt. Um das Programm übersichtlicher und handhabbarer zu gestalten, wird dieser Programmteil einmalig als Funktion programmiert und im restlichen Programm mit seinem Funktionsnamen aufgerufen. Außerdem erleichtert dies die Wartung, da Codeänderungen nur an dieser einen Stelle durchgeführt werden müssen.
- Bereits fertiggestellte Funktionen können für andere Programme anderer Programmierer zur Verfügung gestellt werden, analog zur Benutzung von `pow(x,y)` und `strcmp(s1,s2)` in § 3.6.

7.1 Definition und Deklaration

In der allgemeinen Form der Funktions**definition** mit

```
<speicherklasse> <typ>  <funktions_name> (parameter_liste)
{
 <vereinbarungen>
 <anweisungen>
}
```

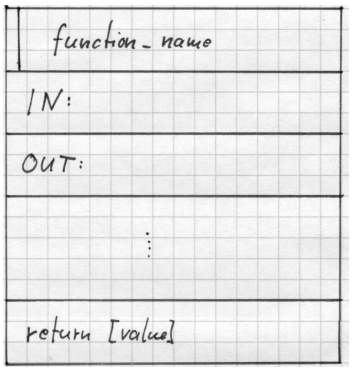
stellen Vereinbarungs- und Anweisungsteil den Funktionskörper dar und `<typ>` legt den Typ des Rückgabewertes fest. Die Kombination `<funktions_name>` und `(parameter_liste)` kennzeichnet eindeutig eine Funktion und wird daher als **Signatur** einer Funktion bezeichnet. Die Funktionsdefinition wird für jede Funktion genau einmal benötigt.

Im Unterschied dazu ist die Funktions**deklaration**

```
<speicherklasse> <typ>  <funktions_name> (parameter_liste) ;
```

in jedem Quellfile nötig welches die Funktion `<funktions_name>` aufruft.

Struktogramm:



Beispiel: Wir schreiben die Berechnung von sgn(x) von Seite 25 als Funktion.

Listing 7.1: Funktion `sgn`

```
#include <iostream>
using namespace std;
double sgn(double x);        // declare function sgn // prototype
double sgn(double x);        // 2nd identical declaration possible
// double sgn(double& x);    // not allowed in this context !!
// int sgn(double x);        // not allowed in this context  !!
double sgn(double* x);       // declaration of another, not used, sgn function
          //  |   different type of argument
//-------------------------------------------------------------------------
int main()
{
    double a, b;
    cout << endl << " Input Argument       : ";
    cin  >> a;
    b  = sgn(a);                          // function call
    cout << "  sgn(" << a << ") = " << b << endl;
    return 0;
}
//-------------------------------------------------------------------------
double sgn(double x)                      // definition of function sgn
{
    double y;                             // local variable in function
    y = (x > 0 ? 1. : 0.) + (x < 0 ? -1. : 0.);
    return y;                             // return value of function sgn
}
```

Ex710.cpp

Bemerkungen: Die Funktion `sgn()` ist durch ihre Signatur eindeutig beschrieben. Dies hat für Deklaration und Definition von Funktionen die Konsequenzen:

(i) Einige weitere (oder noch mehr) identische Funktionsdeklarationen

```
double sgn(double x);
```

sind in obigem Beispiel erlaubt.

(ii) Zusätzliche Funktionsdeklarationen mit anderen Parameterlisten sind erlaubt, z.B.:

```
double sgn(double* x);
double sgn(int x);
```

da sich die Argumente von der Ausgangsdefinition unterscheiden. Allerdings haben wir diese neuen Funktionen noch nicht definiert.

(iii) Eine zusätzliche Deklaration

```
double sgn(double& x);
```

ist nicht erlaubt, da die Signatur wie unter (i) ist. Daher kann der Compiler nicht herausfinden, ob die Funktion unter (iii) oder die Funktion unter (i) in der Anweisung

```
y = sgn(x);
```

gemeint ist.

(iv) Verschiedene Funktionen mit gleichem Namen werden anhand ihrer unterschiedlichen Parameterlisten identifiziert, siehe Pkt. (iii).

(v) Der Rückgabewert einer Funktion kann nicht zu ihrer Identifikation herangezogen werden, die Deklarationen

```
double sgn(int x);
   int sgn(int x);
```

können nicht unterschieden werden (gleiche Signatur) und daher lehnt der Compiler diesen Quelltext ab.

7.2 Parameterübergabe

Beim Programmentwurf unterscheiden wir drei Arten von Parametern einer Funktion:

INPUT Parameterdaten werden in der Funktion benutzt aber nicht verändert, d.h., sie sind innerhalb der Funktion konstant.

INOUT Parameterdaten werden in der Funktion benutzt und verändert.

OUTPUT Parameterdaten werden in der Funktion initialisiert und gegebenenfalls verändert.

Wir werden programmtechnisch nicht zwischen INOUT- und OUTPUT-Parametern unterscheiden. Es gibt generell drei Möglichkeiten der programmtechnischen Übergabe von Parametern Ex721.cpp

1. Übergabe der Daten einer Variablen (engl.: by value).
2. Übergabe der Adresse einer Variablen (engl.: by address)
3. Übergabe der Referenz auf eine Variable (engl.: by reference), wobei hierbei versteckt eine Adresse übergeben wird.

Bemerkung zu `const`:
Wenn eine Variable in der Funktion als Konstante benützt wird, dann sollte sie auch so behandelt werden, d.h., reine INPUT-Parameter sollten stets als `const` in der Parameterliste gekennzeichnet werden. Dies erhöht die Sicherheit vor einer unbeabsichtigten Datenmanipulation und erleichtert auch eine spätere Fehlersuche.

Bemerkung zur Äquivalenz von Pointern und Referenzen:
Eine Referenz `double& x` entspricht dem konstanten Pointer `double* const px` welcher nicht verändert werden kann aber über `*px` kann auf den Wert im Speicher zugegriffen werden. Dementsprechend sind in Tab. 7.1 die folgenden Einträge der Parameterliste identisch:

```
double& x        ≡  double* const px
const double& x  ≡  const double* const px
```

7.3 Rückgabewerte von Funktionen

Jede Funktion besitzt ein Funktionsergebnis vom Datentyp `<typ>` . Als Typen dürfen verwendet werden:

- einfache Datentypen (§ 2.1.1),
- Strukturen (§ 5.2), Klassen,
- Zeiger (§ 6.1),
- Referenzen (§ 6.3),

jedoch keine C-Felder und Funktionen - dafür aber Zeiger auf ein Feld bzw. eine Funktion und Referenzen auf Felder.

Der Rückgabewert (Funktionsergebnis) wird mit

```
return <ergebnis> ;
```

an das rufende Programm übergeben. Ein Spezialfall sind Funktionen der Art

```
void f(<parameter_liste>)
```

für welche kein Rückgabewert (void = leer) erwartet wird, sodaß mit

```
return ;
```

in das aufrufende Programm zurückgekehrt wird.

Listing 7.2: Funktionsergebnisse

```
using namespace std;
void spass(const int);                    // Declaration of spass()
```

Wir betrachten die Möglichkeiten der Parameterübergabe am Beispiel der `sgn` Funktion mit der Variablen `double a` .

Übergabeart	Parameterliste	Aufruf	Effekt von `x++`	Effekt von `(*x)++`	Verwendung	Empfehlung
by value	`double x`	`sgn(a)`	intern	—-	**I**NPUT	[C]
	`const double x`		nicht erlaubt	—-	**I**NPUT	C [einfache Datentypen]
by address	`double* x`	`sgn(&a)`	intern	intern/extern	IN**O**UT	C
	`const double* x`		intern	nicht erlaubt	**I**NPUT	C [komplexe Datentypen]
	`double* const x`		nicht erlaubt	intern/extern	IN**O**UT	[C]
	`const double* const x`		nicht erlaubt	nicht erlaubt	**I**NPUT	[C]
by reference	`double& x`	`sgn(a)`	intern/extern	—-	IN**O**UT	C++
	`const double& x`		nicht erlaubt	—-	**I**NPUT	C++

Tabelle 7.1: Möglichkeiten der Parameterübergabe

Die "by-reference"-Variante `double &const x` wird vom Compiler abgelehnt.

```
int main()
{
    int n;
    cout << endl << " Eingabe n : ";  cin  >> n;
    spass(n);                          // Call spass()
    return 0;
}
void spass(const int n)                 // definition of spass()
{
    cout << endl << "Jetzt schlaegt's aber " << n << endl << endl;
    return;
}
```

Ex731.cpp

Beispiele für Funktionsergebnisse:

`float f1`	`float`-Zahl
`vector<float> f2`	Vektor mit `float` als Elemente
`Student f3`	Klasse/Struktur Student
`vector<Student> f4`	Vektor mit Studenten als Elemente
`Student* f5`	Zeiger auf eine Instanz der Klasse Student
`Student& f6`	Referenz auf eine Instanz der Klasse Student entspricht `Student* const f6`
`vector<Student>& f7`	Referenz auf Vektor mit Studenten
`int* f8`	Zeiger auf `int`-Zahl
`int (*f9)[]`	Zeiger auf C-Array von `int`-Zahlen
`int (*fA)()`	Zeiger auf Funktion, welche den Ergebnistyp `int` besitzt

Bemerkungen:
Eine Funktion darf mehrere Rückgabeanweisungen `return [<ergebnis>];` besitzen, z.B., in jedem Zweig einer Alternative eine. Dies ist jedoch kein sauber strukturiertes Programmieren mehr.
$\Longrightarrow$ Jede Funktion sollte genau eine `return`-Anweisung am Ende des Funktionskörpers besitzen (Standard für das Praktikum).

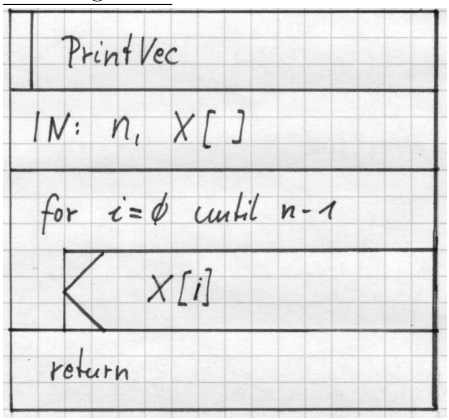
Wie in §4.8 gilt auch hier, daß ein erfahrener Programmierer, also Sie noch nicht, unter Verwendung mehrerer `return`-Anweisungen eine Codebeschleunigung erzielen kann.

7.4 Vektoren als Parameter

Statische C++-Vektoren, also `array<T,N>` können analog zu ihrer Deklaration als Funktionsparameter übergeben werden. Allerdings müssen alle Dimensionen zum Compilierungszeitpunkt bekannt sein.

Wir betrachten als erstes **Beispiel** die Ausgabe eines Vektors, d.h., Vektors $\underline{x}$ der Länge n.

Struktogramm:



Listing 7.3: Vektor als Parameter

```
#include <array>
```

```
#include <cassert>
using namespace std;
const int MCOL = 3;              // global constants (for static 2D array)
//      ----        Declarations   ------------------
/*    Prints the elements of a vector
 *
 *     @param[in]  x  vector
 */
//      ----        Definitions   ------------------
void PrintVec(const vector<double> &x)
{
    cout << endl;
    for (size_t i = 0; i < x.size(); ++i) {
        cout << "  " << x[i];
    }
    cout << endl << endl;
    return;
}
int main()
{
    vector<double> u(MCOL);
    for (size_t i=0; i<u.size(); ++i)  u[i] = i+1;
    PrintVec(u);
    return 0;
}
```

bsp740.cpp

Als nächstes betrachten wir die Ausgabe eines statischen 2D-Feldes, d.h., einer Matrix mit `MCOL` Spalten und `NROW` Zeilen. Hier **muß** die Anzahl der Spalten als globale Konstante definiert werden, da ansonsten die nachfolgende Funktion nicht compiliert werden kann.

Listing 7.4: Matrix als Parameter

```
#include <array>
#include <cassert>
#include <iostream>
#include <vector>
using namespace std;
const int MCOL = 3;              // global constants (for static 2D array)
/*    Prints the elements of a matrix stored in 2D with a globally fixed
 *    number of columns @p MCOL.
 *
 *     @param[in]  a     2D C-array
 */
void PrintMat_fix(const vector<array<double,MCOL>> &a)
//      uses the global constant MCOL |
{
    cout << endl;
    for (size_t i = 0; i < a.size(); ++i) {          // loop on rows
        cout << "row " << i << ":";
        for (int j = 0; j < MCOL; ++j) {             // loop on columns
            cout << "  " << a[i][j];
        }
        cout << endl;
    }
    cout << endl << endl;
    return;
}
int main()
{
    const int NROW = 4;                          // local constant
                                                 // static matrix as 2D-Array
    const vector<array<double,MCOL>> a{{ {{4, -1, -0.5}}, {{ -1, 4, -1}}, {{ -0.5,
        -1, 4}}, {{3, 0, -1}} }};
    assert( NROW ==  static_cast<int>(a.size()) );        // check number of rows
    PrintMat_fix(a);                             //   2D with globally fixed #columns
    return 0;
}
```

bsp740.cpp

Leider können wir die Funktion `PrintMat_fix` nur für statische 2D-Felder (Matrizen) anwenden, und dann auch nur für solche mit `NCOL=3` Spalten - schon eine Matrix `double aa[7][9]` kann mit dieser Funktion nicht mehr ausgegeben werden. Jedoch können wir das 2D-Feld als 1D-Feld der Länge `NROW*MCOL` auffassen und so die Funktion dahingehend verallgemeinern, daß beliebige statische 2D-Felder und als 2D-Felder interpretierbare dynamische 1D-Felder (wie in Version 2 auf Seite 42) übergeben werden können. Diese Funktion kann beliebigen Zeilen- und Spaltenanzahlen arbeiten.

Listing 7.5: Dynamischer Vektor als Parameter interpretiert als Matrix

```
#include <array>
#include <cassert>
```

```
#include <iostream>
#include <vector>
using namespace std;
const int MCOL = 3;              // global constants (for static 2D array)
/*    Prints the elements of a vector interpreted as matrix with
 *    @p nrows rows and @p ncol columns.
 *
 *      @param[in]  nrow  number of rows
 *      @param[in]  ncol  number of columns
 *      @param[in]  a     vector
 */
void PrintMat(const int nrow, const int ncol, const vector<double> &a)
{
    assert(nrow*ncol == static_cast<int>(a.size()));
    cout << endl;
    for (int i = 0; i < nrow; ++i) {
        cout << "Row " << i << ":";
        for (int j = 0; j < ncol; ++j) {
            cout << "  " << a[i * ncol + j] ;
        }
        cout << endl;
    }
    cout << endl << endl;
    return;
}
int main()
{
    const int NROW = 4;                          // local constant
    vector<double> b(NROW * MCOL);               // allocate dynamic matrix b as 1D-Array
    for (int i = 0; i < NROW; ++i)               // copy matrix 'a' on matrix 'b'
        for (int j = 0; j < MCOL; ++j)
            b[i * MCOL + j] = a[i][j];
    PrintMat(NROW, MCOL, b);                     //   1D interpreted as matrix
}
```

bsp740.cpp

Eine komfortable Lösung der Übergabe einer Matrix als Parameter besteht darin, daß man eine Klasse `Matrix` deklariert welche alle notwendige Funktionalität enthält §9.

7.5 Deklarationen und Headerfiles, Bibliotheken

Normalerweise setzt sich der Quelltext eines Computerprogrammes aus (wesentlich) mehr als einem Quelltextfile zusammen. Damit Funktionen, Datenstrukturen (und globale Konstanten, Variablen) und Makros aus anderen Quelltextfiles (*name.cpp*) genutzt werden können, benutzt man **Headerfiles** (*name.h*, *name.h*) welche als **öffentliche Schnittstelle** die Deklarationen für die Funktionen des Quelltextfiles *name.cpp* (= versteckter Bereich) beinhalten. Sie dazu [Wol06, §8.1].

7.5.1 Beispiel: `printvec`

Wir wollen die in §7.4 programmierten Funktionen `PrintVec` und `PrintMat` in einem anderen Quelltext (d.h., Hauptprogramm) benutzen. Zunächst kopieren wir die Definitionen der beiden Funktionen (und alles andere, was zum Compilieren benötigt wird) in das neue File *printvec.cpp*.

printvec.cpp

Listing 7.6: Implementierungsteil der Print-Funktionen

```
#include <iostream>
#include <vector>
#include <cassert>                       // assert()
using namespace std;

void PrintVec(const vector<double> &x)
{
    ...
}

void PrintMat(const int nrow, const int ncol, const vector<double> &a)
{
    assert(nrow*ncol==a.size());         // sind die Parameter kompatibel?
    ...
}
```

Die C-Funktion `assert()` erwartet eine Wahrheitswert als Inputparameter und bricht die Abarbeitung des Programmes sofort ab, falls dieser `false` ist. Da hierbei Name und Zeilennummer des Quelltextfiles angegeben werden, ist die Funktion `assert()` eine einfache Möglichkeit, um die Zulässigkeit bestimmter Größen zu überprüfen. Der Test läßt sich mit der Compileroption `-DNDEBUG`

bequem ausschalten. Die C++-Fehlerbehandlung über das *exception handling* ist komplexer, für unsere Zwecke auch noch nicht nötig und wird im Rahmen der der LV nicht behandelt.

Das File *printvec.cpp* wird nun compiliert (ohne es zu linken!)

```
LINUX> g++ -c -Wall -pedantic printvec.cpp
```

wodurch das Objektfile *printvec.o* erzeugt wird.

Das Hauptprogramm in *Ex751-old.cpp* benötigt nunmehr die Deklarationen der beiden Funktionen.

Listing 7.7: Hauptprogramm ohne Headerfile

```
#include <vector>
using namespace std;
//                    declarations of functions from printvec.cpp
void PrintVec(const vector<double> &x);
void PrintMat(const int nrow, const int ncol, const vector<double> &a);
int main()
{
    const int N = 4, M = 3;                          // local constant
    // temp. C-Arrays
    const double data[] = {4, -1, -0.5,  -1, 4, -1,  -0.5, -1, 4, 3, 0, -1 };
    // static matrix a
    const vector<double> a(data,data + sizeof(data) / sizeof(data[0]));
          vector<double> u(N);
    for (size_t k=0; k<u.size(); ++k)  u[k] = data[k];
    PrintMat(N, M, a);                    // print matrix
    PrintVec(u);                             // print vector
    return 0;
}
```

Ex751-old.cpp

Das Compilieren des Hauptfiles

```
LINUX> g++ -c -Wall -pedantic Ex751-old.cpp
```

erzeugt das Objektfile *Ex751-old.o* welches mit dem anderen Objektfile zum fertigen Programm *a.out* gelinkt werden muß

```
LINUX> g++ Ex751-old.o printvec.o
```

Sämtliches compilieren und linken läßt sich auch in einer Kommandozeile ausdrücken

```
LINUX> g++ -Wall -pedantic Ex751-old.cpp printvec.cpp
```

wobei manche Compiler im ersten Quelltextfile (hier *Ex751-old.cpp*) das Hauptprogramm `main()` erwarten.

Die Deklarationen im Hauptprogramm für die Funktionen aus *printvec.cpp* schreiben wir in das Headerfile *printvec.h*

Listing 7.8: Header der Print-Funktionen

```
#ifndef FILE_PRINTVEC                          // Header guard: begin
#define FILE_PRINTVEC
#include <vector>
using namespace std;
void PrintVec(const vector<double> &x);
void PrintMat(const int nrow, const int ncol, const vector<double> &a);
#endif                                         // Header guard: end
```

printvec.h

und wir ersetzen den Deklarationsteil im Hauptprogramm durch die Präprozessoranweisung

```
#include "printvec.h"
```

Ex751.cpp

welche den Inhalt *printvec.h* vor dem Compilieren von *Ex751.cpp* automatisch einfügt.

Listing 7.9: Hauptprogramm mit Headerfile

```
#include <vector>
using namespace std;
//                    declarations of functions from printvec.cpp
#include "printvec.h"

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

Die Anführungszeichen `"  "` um den Filenamen kennzeichnen, daß das Headerfile *printvec.h* im gleichen Verzeichnis wie das Quelltextfile *Ex751.cpp* zu finden ist.

Das Kommando

```
LINUX> g++ -Wall -pedantic Ex751.cpp printvec.cpp
```

erzeugt wiederum das Programm *a.out*.

7.5.2 Beispiel: `student`

Wir können auch selbstdefinierte Datenstrukturen, z.B. die Datenstrukturen `Student`, `Student_Mult` aus §5.3 und globale Konstanten in einem Headerfile *student.h* speichern.

Listing 7.10: Header der Strukturen und der Funktion

```
#ifndef FILE_STUDENT
#define FILE_STUDENT
#include <string>
#include <vector>
using namespace std;
//------------------------  Student  ------------------------------
struct Student {
    Student(): matrikel(0), skz(0), name(), vorname() {}
    long long int matrikel;
    int skz;
    string name, vorname;
};
//---------------------  Student_Mult  ----------------------------
struct Student_Mult {                    // new structure
    Student_Mult(): matrikel(0), skz(0), name(), vorname() {}
    long long int matrikel;
    vector<int> skz;                     // dynamic vector
    string name, vorname;
};
//      Copy Student onto Student_Mult
void Copy_Student(Student_Mult &lhs, const Student &rhs);
#endif
```

student.h

Die neue Funktion `Copy_Student` wird in *student.cpp* definiert, wobei der Funktionskörper aus *Ex643-correct.cpp* kopiert wurde.

Listing 7.11: Implementierung der Funktion welche die neuen Strukturen nutzt

```
#include "student.h"
#include <string>
#include <vector>
using namespace std;
void Copy_Student(Student_Mult &lhs, const Student &rhs)
{
    lhs.matrikel = rhs.matrikel;
    lhs.name     = rhs.name;
    lhs.vorname  = rhs.vorname;
    lhs.skz.clear();                  //  deletes all elements from vector
    lhs.skz.push_back(rhs.skz);       //  adds the skz from rhs to the (empty)
        vector
    return;
}
```

student.cpp

Da die Struktur `Student` verwendet wird, muß auch das Headerfile *student.h* in *student.cpp* eingebunden werden. Die neue Funktion `Copy_Student` kann nunmehr im Hauptprogramm *bsp752.cpp* zum Kopieren einer Struktur auf eine andere benutzt werden. Das Hauptprogramm benötigt dafür natürlich wieder das Headerfile *student.h*.

bsp752.cpp

Das Kommando

```
LINUX> g++ -std=c++11 -Wall -pedantic bsp752.cpp student.cpp
```

erzeugt schlußendlich das Programm *a.out*.

7.5.3 Eine einfache Bibliothek am Beispiel `student`

Um sich das wiederholte compilieren zusätzlicher Quelltextfiles und die damit verbundenen u.U. langen Listen von Objektfiles beim Linken zu ersparen, verwendet man Bibliotheken. Gleichzeitig haben Bibliotheken den Vorteil, daß man seine compilierten Funktionen (zusammen mit den Headerfiles) anderen in kompakter Form zur Verfügung stellen kann, ohne daß man seine Programmiergeheimnisse (geistiges Eigentum) verraten muß. Dies sei an hand des (sehr einfachen) Beispiels aus §7.5.2 demonstriert.

- Erzeugen des Objektfiles *student.o* (compilieren)
  ```
  LINUX> g++ -c student.cpp
  ```
- Erzeugen/Aktualisieren der Bibliothek *lib*stud*.a* (archivieren) aus/mit dem Objektfile *student.o*. Der Bibliotheksbezeichner stud ist frei wählbar.
  ```
  LINUX> ar r libstud.a student.o
  ```
  Die Archivierungsoptionen (hier, nur `r`) können mit dem verwendeten Compiler variieren.

bsp752.cpp

- Compilieren des Hauptprogrammes und linken mit der Bibliothek aus dem aktuellen Verzeichnis
  ```
  LINUX> g++ bsp752.cpp -L. -lstud
  ```

Die folgenden Schritte sind notwendig, um das Programm ohne Verwendung einer Bibliothek zu übersetzen und zu linken.

$$\left.\begin{array}{l} \textit{student.cpp} \xrightarrow{\texttt{g++ -c student.cpp}} \textit{student.o} \\ \textit{bsp752.cpp} \xrightarrow{\texttt{g++ -c bsp752.cpp}} \textit{bsp752.o} \end{array}\right\} \xrightarrow{\texttt{g++ bsp752.o student.o}} \textit{a.out}$$

Abkürzend ist auch möglich:

$$\textit{bsp752.cpp, student.cpp} \xrightarrow{\texttt{g++ bsp752.cpp student.cpp}} \textit{a.out}$$

Bei Verwendung der Biobliothek *lib***stud***.a* sieht der Ablauf folgendermaßen aus

$$\left.\begin{array}{l} \textit{student.cpp} \xrightarrow[\texttt{student.cpp}]{\texttt{g++ -c}} \textit{student.o} \xrightarrow[\texttt{student.o}]{\texttt{ar r lib}\mathbf{\texttt{stud}}\texttt{.a}} \textit{lib}\mathbf{stud}\textit{.a} \\ \textit{bsp752.cpp} \xrightarrow[\texttt{bsp752.cpp}]{\texttt{g++ -c}} \textit{bsp752.o} \end{array}\right\} \xrightarrow[\texttt{-L. -l}\mathbf{\texttt{stud}}]{\texttt{g++ bsp752.o}} \textit{a.out}$$

was bei bereits vorhandener Bibliothek wiederum abgekürzt werden kann:

$$\textit{bsp752.cpp, lib}\mathbf{stud}\textit{.a} \xrightarrow{\texttt{g++ bsp752.cpp -L. -l}\mathbf{\texttt{stud}}} \textit{a.out}$$

7.6 Das Hauptprogramm

Das Hauptprogramm ist eine Funktion `main` welche im gesamten Code *genau einmal* auftreten darf und von welcher ein Rückgabewert vom Typ `int` erwartet wird.. Bislang benutzen wir diese Funktion ohne Parameterliste. Das Programm wird von einer Umgebung (meist eine *Shell*) aufgerufen welche ihrerseits diesen Rückgabewert auswerten kann. Dabei wird ein Rückgabewert 0 als fehlerfreie Programmabarbeitung interpretiert.

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

Die Programmabarbeitung kann jederzeit, auch in Funktionen, mit der Anweisung `exit(<int_value>);` abgebrochen werden. Der Wert `<int_value>` ist dann der Rückgabewert des Programmes und kann zur Fehlerdiagnose herangezogen werden. Ab der Version 4.3 des Gnu–Compilers müssen zusätzliche Headerfiles für einige eingebaute Funktionen (hier für exit() und atoi()) eingebunden werden, siehe die kompakte[1] bzw. ausführliche[2] Darstellung der Änderungen.

[1] `http://www.cyrius.com/journal/gcc/gcc-4.3-include`
[2] `http://gcc.gnu.org/gcc-4.3/porting_to.html`

Das Programm in Listing 7.12 bricht bei $n < 0$ die Programmausführung in `spass()` sofort ab und liefert den Fehlercode -10 zurück.

Wie bei anderen Funktionen kann auch das Hauptprogramm mit Parametern aufgerufen werden, allerdings ist in

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

die Parameterliste (genauer, die Typen der Parameter) vorgeschrieben, wobei

- `argv[0]` den Programmnamen und
- `argv[1]` ... `argv[argc-1]` die Argumente beim Programmaufruf als Zeichenketten übergeben.
- Es gilt stets `argc`≥ 1, da der Programmname immer als `argv[0]` übergeben wird.

Listing 7.12: Hauptprogramm mit Parametern

```
#include <cstdlib>                      // needed for atoi, exit
#include <iostream>
using namespace std;
void spass(const int);                          // Declaration of spass()
int main(const int argc, const char *argv[])
{
    int n;
    cout << endl;
    cout << "This is code " << argv[0] << endl;
    if (argc > 1) {                     // at least one argument
        n = atoi(argv[1]);              // atoi : ACSII to Integer
    }
    else {                              // standard input
        cout << " Eingabe n : "; cin  >> n; cout << endl;
    }
    spass(n);                           // Call spass()
    cout << endl;
    return 0;                           // default return value is 0
}
void spass(const int n)
{
    if (n < 0) {                        // usage of exit()
        cout << "Fatal Error in spass(): n = " << n << " < 0" << endl;
        exit(-10);                      // choose an error code
    }
    cout << "Jetzt schlaegt's aber " << n << endl;
    return;
}
```

Ex760.cpp

Die Funktion `atoi(char *)` (= ASCII to int) wandelt die übergebene Zeichenkette in eine Integerzahl um und wird im Header *cstdlib* deklariert. Mittels der analogen Funktion `atod(char *)` läßt sich eine Gleitkommazahl als Parameter übergeben. Nach dem Compilieren und Linken kann das Programm `a.out` mittels

```
LINUX> ./a.out
```

bzw.

```
LINUX> ./a.out 5
```

gestartet werden. Im ersteren Fall wird der Wert von `n` von der Tastatur eingelesen, im zweiten Fall wird der Wert `5` aus der Kommandozeile übernommen und `n` zugewiesen. Eine elegante, und echte C++-Lösung, bzgl. der Übergabe von Kommandozeilenparametern kann in [Str00, pp.126] gefunden werden.

7.7 Rekursive Funktionen

Funktionen können in C/C++ rekursiv aufgerufen werden.

Beispiel: Die Potenz x^k mit $x \in \mathbb{R}$, $k \in \mathbb{N}$ kann auch als $x^k = \begin{cases} x \cdot x^{k-1} & k > 0 \\ 1 & k = 0 \end{cases}$ realisiert werden.

Listing 7.13: Rekursive Funktion `power`

```
// definition of function power
double power(const double x, const int k)
{
    double y;
    if ( k == 0) {
        y = 1.0;                        // stops recursion
    }
    else {
        y = x * power(x, k - 1);        // recursive call
    }
    return y;                           // return value of function power
}
```

Ex770.cpp

7.8 Ein größeres Beispiel: Bisektion

Im Beispiel auf Seite 32 ging es darum, die Nullstelle von $f(x) := \sin(x) - x/2$ im Intervall (a,b), mit $a = 0$ und $b = 1$ zu bestimmen. Unter der Voraussetzung $f(a) > 0 > f(b)$ kann dieses Problem (für stetige Funktionen) mittels Bisektion gelöst werden. Der Bisektionsalgorithmus besteht für jedes Intervall $[a, b]$ im wesentlichen aus den Schritten

(i) $c := (a + b)/2$

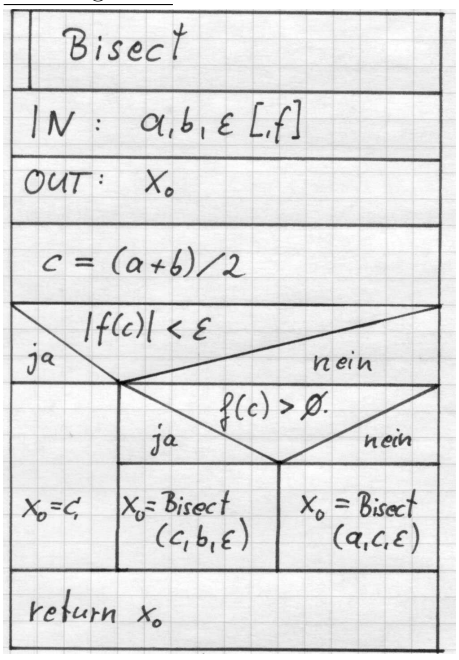
(ii) Ist $|f(c)|$ nah genug an 0 ?

(iii) In welcher Intervallhälfte muß ich weitersuchen ?

Dies ist eine klassische Rekursion, wobei Punkt (iii) die nächste Rekursion einleitet und Punkt (ii) den Abbruch der Rekursion garantieren soll. Formal können wir dies so ausdrücken:

$$x_0 := \text{Bisect}(a, b, \varepsilon) := \begin{cases} c := (a + b)/2 & \text{falls } |f(c)| < \varepsilon \\ \text{Bisect}(c, b, \varepsilon) & \text{sonst, falls } f(c) > 0 \\ \text{Bisect}(a, c, \varepsilon) & \text{sonst, falls } f(c) < 0 \end{cases}$$

Struktogramm:



Dies ergibt die Funktionsdefinition für `Bisect()` welche mit

```
x0 = Bisect(a,b,1e-6);
```

aufgerufen wird und zur Version 1 des Bisektionsprogrammes führt.

Listing 7.14: Bisektion-I

```
double Bisect1(const double a, const double b, const double eps)
{
    double x0, fc, c = (a + b) / 2;
    fc = sin(c) - 0.5 * c;
    if ( std::abs(fc) < eps ) {          // end of recursion
        x0 = c;
    }
    else if (  fc > 0.0 ) {
        x0 = Bisect1(c, b, eps);         // search in the right intervall
    }
    else {                               // i.e., fc < 0.0
        x0 = Bisect1(a, c, eps);         // search in the left intervall
    }
    return x0;                           // return the solution
}
```

Bisect1.cpp

Um das Programm etwas flexibler zu gestalten, werden wir die fix in `Bisect1()` einprogrammierte Funktion $f(x)$ durch die globale Funktion

Listing 7.15: Globale Funktion und globale Konstante

```
const double EPS = 1e-6;             // global constant

double f(const double x)             // declaration and definition of function f(x)
{
    return  sin(x) - 0.5 * x ;
}
```

ersetzen. Gleichzeitig könnten wir den Funktionsparameter `eps` durch eine globale Konstante `EPS` ersetzen, sodaß sich Version II des Codes ergibt.

Bisect2.cpp

Die Flexibilität der Bisektionsfunktion läßt sich weiter erhöhen, indem wir die auszuwertende Funktion $f(x)$ als Variable in der Parameterliste übergeben. Bei einer Funktion als Parameter müssen die Argumente wie die Deklaration für `f6` auf Seite 59 aufgebaut sein. Konkret heißt dies:

`std::function<double(double)>` ist eine Typbezeichnung für eine Funktion mit einer `double`-Variablen als Argument und `double` als Typ des Rückkehrwertes (C++11; `#include <functional>`) [Wil18, §23.3.1].

Dies erlaubt uns die Funktionsdeklaration und -definition von `Bisect3()`.

Listing 7.16: Bisektion-III mit Funktion als Parameter

```
#include <iostream>
// declaration of Bisect3
double Bisect3(const std::function<double(double)>& func,
               const double a, const double b, const double eps = 1e-6);
int main()
{
    //    ...
    return 0;
}
//      definition of Bisect3
double Bisect3(const std::function<double(double)>& func, const double a, const
    double b,
               const double eps)
{
    double x0;
    double c = (a + b) / 2;              // center of interval
    double fc = func(c);                 // function value in center
    if ( std::abs(fc) < eps ) {          // end of recursion
        x0 = c;
    }
    else if (  fc > 0.0 ) {
        x0 = Bisect3(func, c, b, eps);   // search in the right intervall
    }
    else {              // i.e., fc < 0.0
        x0 = Bisect3(func, a, c, eps);   // search in the left intervall
    }
    return x0;                           // return the solution
}
```

Bisect3.cpp

Das vierte Argument (`eps`) in der Parameterliste von `Bisect3()` ist ein **optionales Argument**, welches beim Funktionsaufruf nicht übergeben werden muß. In diesem Fall wird diesem optionalen Argument sein, in der Funktionsdeklaration festgelegter, Standardwert automatisch zugewiesen. In unserem Falle würde also der Aufruf im Hauptprogramm

```
x0 = Bisect3(f,a,b,1e-12)
```

die Rekursion bei $|f(c)| < \varepsilon := 10^{-12}$ abbrechen, während

```
x0 = Bisect3(f,a,b)
```

schon bei $|f(c)| < \varepsilon := 10^{-6}$ stoppt.

Wir könnten jetzt eine weitere Funktion

Listing 7.17: Weitere globale Funktion

```
double g(const double x)          // declaration and definition of function g(x)
{
    return -(x-1.234567)*(x+0.987654) ;
}
```

Bisect3.cpp

deklarieren und definieren, und den Bisektionsalgorithmus in Version III mit dieser aufrufen:

```
x0 = Bisect3(g,a,b,1e-12)
```

Unser Programm arbeitet zufriedenstellend für $f(x) = \sin(x) - x/2$ und liefert für die Eingabeparameter $a = 1$ und $b = 2$ die richtige Lösung $x_0 = 1.89549$, desgleichen für $a = 0$ und $b = 2$ allerdings wird hier bereits die (triviale) Lösung $x_0 = 0$ nicht gefunden, da $a = 0$ eingegeben wurde. Bei den Eingaben $a = 0$, $b = 1$ bzw. $a = -1$, $b = 0.1$ ($x_0 := 0 \in [a, b]$) bricht das Programm nach einiger Zeit mit *Segmentation fault* ab, da die Rekursion nicht abbricht und irgendwann der für Funktionsaufrufe reservierte Speicher (*Stack*) nicht mehr ausreicht.

Können wir unser Programm so absichern, daß z.B. die vorhandene Nullstelle $x_0 = 0$ sowohl in $[0, 1]$ als in $[-1, 0.1]$ gefunden wird? Welche Fälle können bzgl. der Funktionswerte $f(a)$ und $f(b)$ auftreten (vorläufige Annahme: $a < b$)?

(i) $f(a) > 0 > f(b)$ (d.h., $f(a) > 0$ und $f(b) < 0$), z.B., $a = 1$, $b = 2$
$\Longrightarrow$ Standardfall in `Bisect3()`.

(ii) $f(a) > 0$ und $f(b) > 0$, z.B., $a = 0.5$, $b = 1.5$ bzw.
$f(a) < 0$ und $f(b) < 0$, z.B., $a = -1$, $b = 0.5$
evtl. keine Nullstelle $\Longrightarrow$ Abbruch.
(Es können Nullstellen im Intervall vorhanden sein, welche wir aber mit der Bisektionsmethode nicht finden können!)

(iii) $f(a) = 0$ oder $f(b) = 0$, besser $|f(a)| < \varepsilon$ etc.
$\Longrightarrow$ a oder b sind die Nullstelle,
oder $\Longrightarrow$ sowohl a als auch b sind eine Nullstelle.

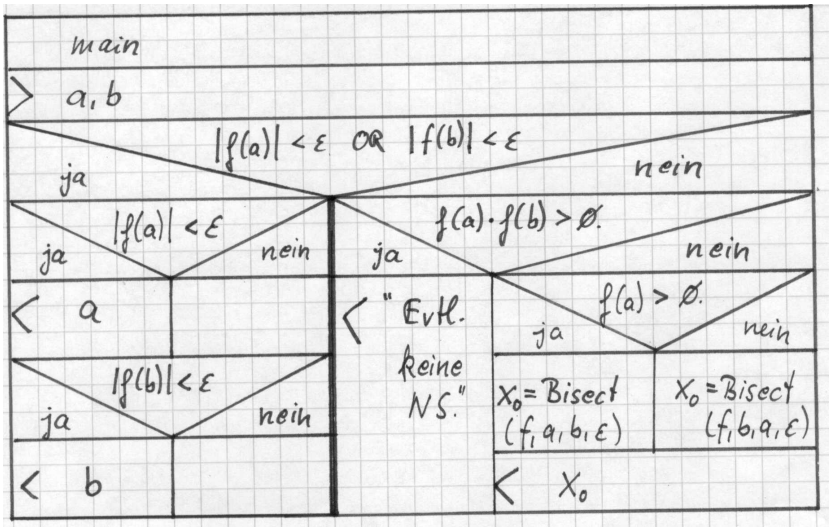
(iv) $f(a) < 0 < f(b)$, z.B. $a = -1$, $b = 0.1$
Vertausche a und b $\Longrightarrow$ Fall (i).

(v) $a = b$ $\Longrightarrow$ in (ii) und (iii) enthalten.
$b < a$ $\Longrightarrow$ führt auf (i) oder (iv).

Bisect4.cpp

Diese Fallunterscheidung führt uns zum folgenden Struktogramm und zur Version IV.

Struktogramm:



Als krönenden Abschluß definieren wir uns im Programm weitere Funktionen $h(x) = 3 - e^x$, $t(x) = 1 - x^2$, fragen den Nutzer welche math. Funktion für die Nullstellensuche benutzt werden soll und berechnen die Nullstelle(n) im gegebenen Intervall. Diese Auswahl kann leicht mit einer `switch`-Anweisung realisiert werden und führt zu Version V des Programmes.

Listing 7.18: Bisektion-V mit einer Funktionvariablen

```
#include <iostream>
double t(const double x)        // declaration and
{
    return 1 - x * x ;          //   definition of function t(x)
}
int main()
{
    std::function<double(double)> ff;            // function variable
            ff = t;
    }
        else {                                  // f(a) > 0 && f(b) < 0
    return 0;
```

Bisect5.cpp

Bemerkung: Die drei Funktionen `Bisect`[1-3]() unterscheiden sich in ihren Parameterlisten. Deshalb können alle drei Funktionen unter dem Namen `Bisect()` verwendet werden, da sich ihre Signaturen unterscheiden und somit der Compiler an Hand der Parameterliste genau weiß, welche Funktion `Bisect()` verwendet werden soll.

Bisect6.cpp

Kapitel 8

Input und Output mit Files und Terminal

Die zur Ein-/Ausgabe verwendeten Objekte `cin` und `cout` sind (in *iostream*) vordefinierte Variablen vom Klassentyp `stream`. Um von Files zu lesen bzw. auf Files zu schreiben werden nun neue Streamvariablen angelegt und zwar vom Typ `ifstream` für die Eingabe und vom Typ `ofstream` für die Ausgabe. Der Filename wird beim Anlegen der Variablen übergeben (C++ Konstruktor).

8.1 Kopieren von Files

Das folgende Programm kopiert ein Inputfile auf ein Outputfile, allerdings ohne Leerzeichen, Tabulatoren, Zeilenumbrüche.

Listing 8.1: Files ohne Leerzeichen usw. kopieren

```
#include <cassert>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string  infilename, outfilename;
    cout << " Input file: ";
    cin >> infilename;
    cout << "Output file: ";
    cin >> outfilename;
    ifstream  infile(infilename);       // Eingabefile oeffnen
    if ( !infile.is_open() ) {          // Absicherung, falls File nicht existiert
        cout << "\nFile " << infilename << " has not been found.\n\n" ;
        assert( infile.is_open() );     // exeption handling for the poor programmer
    }
    ofstream outfile(outfilename);      // Ausgabefile oeffnen
    while (infile.good()) {
        string str;
        infile >> str;
        outfile << str;
    }
    outfile.close();                    // Ausgabefile schliessen
    return 0;
}                                       // automatisches Schliessen des Eingabefiles
```

bsp811.cpp

Zeilenweises einlesen erfolgt über die Methode `getline()`.

Will man dagegen das File identisch kopieren, so muß auch zeichenweise ein- und ausgelesen werden. Hierzu werden die Methoden `get` und `put` aus den entsprechenden Streamklassen verwendet.

Listing 8.2: Identisches Kopieren von Files

```
    while (infile.good()) {
        char ch;
        infile.get(ch);                 // Zeichenweises lesen
        outfile.put(ch);                // Zeichenweises schreiben
```

```
    }
```

bsp812.cpp

8.2 Dateneingabe und -ausgabe via File

Die Dateneingabe und -ausgabe via ASCII-File und Terminal kann gemischt benutzt werden.

Listing 8.3: Dateingabe über ASCII-File und Terminal

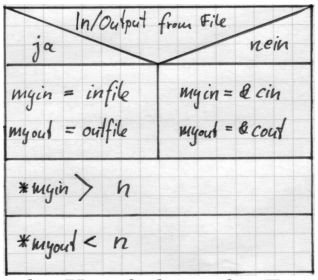
```
#include <fstream>                // needed for ifstream and ofstream
#include <iostream>
using namespace std;
int main()
{
    int n_t, n_f;
//        input file
    ifstream infile("in.txt");
    cout << "Input from terminal: ";
    cin  >> n_t;
    cout << "Input from file " << endl;
    infile >> n_f;
//        check
    cout << endl;
    cout << "Input  from terminal was " << n_t << endl;
    cout << "Output from      file was " << n_f << endl;
    cout << endl;
//        output file
    ofstream outfile("out.txt");
    cout << "This is an output to the terminal" << endl;
    outfile << "This is an output to the file" << endl;
    return 0;
}
```

FileIO_a.cpp

Mehr Details, insbesondere zum binären Daten-I/O sind im cplusplus-Tutorial[1] zu finden.

8.3 Umschalten der Ein-/Ausgabe

Manchmal ist ein problemabhängiges Umschalten zwischen File-IO und Terminal-IO wünschenswert oder nötig. Leider muß in diesem Falle mit Zeigern auf die Typen `istream` und `ostream` gearbeitet werden.



FileIO_c.cpp

FileIO_d.cpp

In den Beispielen ist eine sehr komfortable Möglichkeit des Umschaltens der Ein-/Ausgabe mittels Kommandozeilenparameter zu finden.

Listing 8.4: Flexibles Umschalten zwischen File- und Terminal-IO

```
#include <fstream>                // needed for ifstream and ofstream
#include <iostream>
using namespace std;
int main()
{
//          variables for IO streams: pointers !!
    istream  *myin;
    ostream  *myout;
//          input file
    istream *infile  = new ifstream("in.txt");
//          output file
    ostream *outfile = new ofstream("out.txt");
//      Still standard IO
//      Decide whether terminal-IO or file-IO should be used
    int tf;
    cout << "Input from terminal/file - Press 0/1 : ";
    cin  >> tf;
    bool bf = (tf == 1);
    if (bf) {                   // Remaining IO via file
```

[1] http://www.cplusplus.com/doc/tutorial/files/

```
        myin  = infile;
21      myout = outfile;
    }
23  else {                      // Remaining IO via terminal
        myin  = &cin;
25      myout = &cout;
    }
27  int n;
    (*myout) << "Input: ";
29  (*myin)  >> n;
    (*myout) << endl;           // check
31  (*myout) << "Input was " << n << endl;
    (*myout) << endl;
33  (*myout) << "This is an additional output" << endl;
    delete  outfile;            // don't forget to deallocate
35  delete  infile;
    return 0;
37 }
```

FileIO_b.cpp

8.4 Ausgabeformatierung

Die Ausgabe über Streams (<<) kann verschiedenst formatiert werden. Eine kleine Auswahl von Formatierungen sei hier angegeben, mehr dazu in der Literatur [Wol06, §7.2] zu Manipulatoren und Methoden des Input/Output–Streams.

Wir benutzen die Variablen

```
double da = 1.0/3.0,
       db = 21./2,
       dc = 1234.56789;
```

Format.cpp

- Standardausgabe:
  `cout << da << endl << db << endl << dc << endl << endl;`
- Mehr gültige Ziffern (hier 12) in der Ausgabe:
  ```
  cout.precision(12);
  cout << ...
  ```
- Fixe Anzahl (hier 6) von Nachkommastellen:
  ```
  cout.precision(6);
  cout.setf(ios::fixed, ios::floatfield);
  cout << ...
  ```
- Ausgabe mit Exponent:
  ```
  cout.setf(ios::scientific, ios::floatfield);
  cout << ...
  ```
- Rücksetzen auf Standardausgabe:
  ```
  cout.setf(ios::floatfield);
  cout << ...
  ```
- Ausrichtung (rechtsbündig) und Platzhalter (16 Zeichen) via Methode `width` oder Manipulator `setw` welcher den Header `<iomanip>` erfordert:
  ```
  cout.setf(ios::right, ios::adjustfield);       //  Ausrichtung
  cout.width(16);                                //  Platzhalter
  cout << da << endl;
  cout.width(16);   cout << db << endl;
  //  und nun Platzhalter via Manipulator
  cout << setw(16) << da << setw(16) << db << endl << endl;
  ```
  Die Nutzung von Standardmanipulatoren in der letzten Zeile ist in [Str00, §1.4.6.2, pp.679] zu finden.
- Hexadezimalausgabe von Integerzahlen:
  ```
  cout << hex;
  cout << "127 = " << 127 << endl;
  ```
- Ausgabe von Boolean:
  ```
  cout << boolalpha;
  cout << true << " " << false << endl;
  ```

8.5 Abgesicherte Eingabe

Tippfehler bei der Eingabe von Zahlen sind ärgerlich, insbesondere wenn dadurch das Programm in einer Endlosschleife der Eingabe hängenbleibt. Dies läßt sich folgendermaßen absichern, die Idee dazu ist dem Beitrag in cplusplus.com[2] entnommen.

Listing 8.5: Für Integer abgesicherte Eingabe

demoEingabe.cpp

```
#include <iostream>
#include <sstream>        // istringstream
using namespace std;
int save_input()
{
    string Text;          // string containing the number
    cin >> Text;
    int Result;           // integer number that will contain the result
    istringstream convert(Text);
//
// stringstream is used for the conversion constructed with the contents of 'Text'
// i.e.: the stream will start containing the characters of 'Text'
//
// converts the value to 'Result' using the characters in the stream
    if ( !(convert >> Result) )
        Result = 0;       // if that fails set 'Result' to 0
    return Result;
}
int main()
{
    int n = save_input();
    cout << " n : " << n << endl;
    return 0;
}
```

Obiges Beispiel kann allerdings keine Eingaben wie `3.5` statt der gewünschten `int`-Zahl `3` abfangen. Dafür wäre eine Konvertierung in `float` nötig.

[2] `http://www.cplusplus.com/articles/D9j2Nwbp/`

Kapitel 9

Erste Schritte mit Klassen

Member – Methoden; Kapselung; Kopierkonstruktor; Zuweisungsoperator

Eine Klasse ist ein Konzept, welches Daten gemeinsam mit Funktionen auf diesen Daten definiert. Daher sind diese Funktionen an ein Objekt dieser Klasse gebunden und diese Art des Programmierens wird *objektorientiert* (OOP) genannt. Ihre wahre Stärke spielen Klassen im Weiteren bei Klassenhierarchien und beim Polymorphismus in §12 aus.

Um klarer zwischen strukturierter und objektorientierten Konzepten zu unterscheiden werden folgende Begriffe benutzt:

- Daten in einer Klasse $\longrightarrow$ *Member* (auch *Eigenschaften*),
- Funktion in einer Klasse $\longrightarrow$ *Methode*
- Variable vom Typ der Klasse $\longrightarrow$ *Instanz* (auch *Objekt*) der Klasse.
- *Konstruktoren* dienen der Erstinitialisierung von Membern bei Deklaration einer Instanz.
- Der *Destruktor* gibt Ressourcen wieder frei, wenn der Gültigkeitsbereich (scope) einer Instanz endet.

Natürlich wird strukturierte Programmierung innerhalb der Methoden wieder benötigt.

9.1 Unsere Klasse `Komplex`

Wir führen weitere Begriffe an Hand der Beispielklasse `Komplex` ein, welche eine komplexe Zahl samt der entsprechenden Funktionalität speichert. Es gibt gibt bereits eine Template-Klasse `complex`[1] mit dieser Funktionalität, aber wir bezwecken hier eine einfache Einführung in Klassen. Aus Gründen des besseren Verständnisses werden die grundlegend notwendigen Methoden deklariert, obwohl diese vom Compiler auch automatisch angelegt würden.

Listing 9.1: Nutzung unserer Klasse Komplex

```
1  #include <iostream>
   #include "komplex.h"
3  using namespace std;
   int main()
5  {
       const Komplex a(3.2,-1.1); // Konstruktor Komplex(double,double)
7      Komplex b(a);              // Kopierkonstruktur wird benoetigt
       Komplex c;                 // Konstruktor  Komplex()   wird benoetigt
9      c = a+b;                   // OK: a.operator+(const Komplex&)
                                  // Zuweisungsoperator:  c.operator+(const Komplex&)
11     cout << a << endl;         // Ausgabeoperator: operator<<()  ist eine Funktion!
       cout << c << endl;
13     double dd(-3.2);
                                  // OK: a.operator+(const Komplex&)
```

[1] `http://www.cplusplus.com/reference/complex/complex/?kw=complex`

```
15      c = a + Komplex(dd,0.0);    // explizites Casting double --> Komplex
                                    //          via Konstruktor Komplex(double)
17      c = a + dd;                 // implizites Casting via Konstruktor Komplex(double)
        cout << c << endl;
19      c = dd+a;                   // Achtung: keine Methode operator+(const Komplex&)
                                    //                      fuer double verfuegbar.
21                                  // Daher Funktion operator+(double, const
                                        Komplex&) noetig
        cout << c << endl;
23      return 0;
}                                   // impliziter Destruktoraufruf
```

komplex/main.cpp

Mit unserer Klasse `Komplex` soll das Programm in Listing 9.1 funktionieren.

- **Zeilen 6-8**: Deklaration (und Initialisierung) dreier Instanzen von `Komplex`.
- **Zeilen 9,15,17**: Addition zweier Instanzen und Zuweisung des Ergebnisses an eine dritte Instanz.
- **Zeilen 11,12,18,22**: Ausgabe jeweils einer Instanz über die Standardausgabe.
- **Zeile 24**: Ende des Gültigkeitsbereichs der drei Instanzen, impliziter dreifacher Aufruf des Klassendestruktors.

Jede komplexe Zahl besteht aus Real- und Imaginärteil, welche wir als Member speichern müssen. Die Methoden der Klasse ergeben sich erstmal aus den Anforderungen des Listing. Das Projekt besteht aus dem Hauptprogramm *main.cpp*, dem Headerfile unserer Klasse *komplex.h* mit allen Deklarationen und dem Sourcefile *komplex.cpp* mit den Implementierungen der Methoden unserer Klasse. Die Deklaration der Klasse kann GUI-unterstützt in codeblocks erfolgen (File → New → Class), allerdings werden für die Methoden nur die formalen Hüllen bereitgestellt - nicht die Implementierung!

Listing 9.2: Teil des Headerfile für `Komplex`

```
#ifndef KOMPLEX_H
2 #define KOMPLEX_H
#include <iostream>
4 using namespace std;
class Komplex
6 {
public:
8     /** Default constructor */
      Komplex();
10    /** Parameter constructor
       * \param[in] re Realteil
12     * \param[in] im Imaginaerteil (default value: 0.0)
       */
14    //                          |-- Standardwert
      Komplex(double re, double im=0.0); // Parameterkonstruktor mit ein oder zwei
          Argumenten
16   /** Copy constructor
       * \param[in] org Instance to copy from
18     */
      //
20    Komplex(const Komplex& org); // Kopierkonstruktor
      /** Default destructor */
22    virtual ~Komplex();
      /** \brief Assignment operator
24     * \param[in]   rhs  Instance that is assigned to the members.
       * \return reference to the current instance.
26     */
      Komplex& operator=(const Komplex& rhs);        // Zuweisungsoperator
28    /**  Addiert die aktuelle Instanz mit einer zweiten komplexen Zahl
      * \param[in] rhs zweite komplexe Zahl
30    * \return    \p *this + \p rhs
      */
32    //                                    |--  Membervariablen dieser Instanz
          werden nicht veraendert!
      Komplex operator+(const Komplex& rhs) const;
34 protected:
private:
36    double _re; //!< Realteil
      double _im; //!< Imaginaerteil
38 };
#endif // KOMPLEX_H
```

komplex/komplex.h

- **Zeilen 1,2,39**: Die *Header Guards* sind Präprozessoranweisungen welche verhindern, daß die Deklarationen (und gegebenfalls auch Definitionen) nur einmal in jedes Quellfile eingebunden werden. Unbedingt benutzen!
- **Zeilen 5,6,38** sind der Rahmen für die Deklaration der Klasse.
- Einige Methoden sind in Zeilen 33 aufgeführt.
- Die Member (Real- und Imaginärteil) werden in Zeilen 36-37 vom Typ `double` deklariert.
- **Zeilen 7,34,35**: Für Member und Methoden werden Zugriffsrechte vergeben:
  - Mit *public* gekennzeichnte Member und Methoden sind von außerhalb der Klasse sichtbar und benutzbar.
  - *private* Member und Methoden sind nur innerhalb der Klasse sichtbar und benutzbar.
  - Sind Member und Methoden als *protected* gekennzeichnet, dann sind diese nur innerhalb der Klasse, oder von ihr abgeleiteter Klassen sichtbar und benutzbar (im Listing über einne leeren Menge).
  - Aus Gründen der *Datenkapselung* sollen Member nicht public sein. Um auf diese zugreifen zu können, müssen Getter-/Settermethoden von der Klasse bereitgestellt werden.
- **Zeilen 9, 15, 20**: Deklaration des Standardkonstruktors (leere Parameterliste), eines Parameterkonstruktors und des Kopierkonstruktors.
- **Zeile 22**: Deklaration des Destruktors.
- **Zeile 26**: Deklaration des Zuweisungsoperators `operator=` .
- **Zeile 33**: Deklaration des Additionsoperators `operator+` . Das Schlüsselwort `const` am Ende der Deklaration garantiert, daß diese Methode keine Member der aufrufenden Instanz verändert. Damit ist diese Methode für kontante Instanzen anwendbar.
- Die Klasse ist für die Verarbeitung mit doygen dokumentiert.

9.2 Konstruktoren

Konstruktoren dienen der Erstinitialisierung von Membern einer Klasse und sie haben keinen Rückgabetyp. Man unterscheidet im wesentlichen zwischen dem Standardkonstruktor mit leerer Parameterliste, den Parameterkonstruktoren und den Kopierkonstruktoren. Ob diese Konstruktoren zur Funktionalität einer Klasse dazugehören sollen oder nicht liegt im Ermessen des Designers der Klasse.

Deren Deklarationen erfolgt im Headerfile *komplex.h*, die Implemenierungen schreiben wir hier in das Sourcefile *komplex.cpp*.

9.2.1 Standardkonstruktor

Damit eine Instanz über `Komplex c;` deklariert werden kann, ist der Standardkonstruktor notwendig.

Listing 9.3: Deklaration des Standardkonstruktors im Headerfile

```
class Komplex
{
public:
    Komplex();
};
```

komplex/komplex.h

Wir implementieren die Methode im Sourcefile.

Listing 9.4: Implementierung des Standardkonstruktors im Sourcefile

```
1 #include "komplex.h"
Komplex::Komplex()
3     : _re(0.0), _im(0.0)
{
5     //ctor
}
```

komplex/komplex.cpp

Jede Methode ist ein Funktion, welche genau einer Klasse zugeordnet ist. Damit muß der Klassenname zur Identifikation mit in die Signatur der Funktion aufgenommen werden. Damit hat der Standardkonstruktor der Klasse `Komplex` die Signatur `Komplex::Komplex( )` welche im Sourcefile so angegeben werden muß.

Klassenmember sollten in der *Member Initialization List* (Zeile 3 in Listing 9.4) initialisiert werden. Für konstante Member und Basisklassen einer Klasse ist dies die einzige Möglichkeit der Initialisierung. Damit bleibt der Funktionskörper bei uns leer, was aber bei komplizierteren Klassen nicht mehr der Fall sein muß.

9.2.2 Parameterkonstruktor

Eine Instanzdeklaration `Komplex a(3.2,-1.1);` erfordert den entsprechenden Parameterkonstruktor.

Listing 9.5: Deklaration des Parameterkonstruktors im Headerfile

```
1 class Komplex
{
3 public:
    //                          |-- Standardwert
5     Komplex(double re, double im=0.0); // Parameterkonstruktor mit ein oder zwei
          Argumenten
};
```

komplex/komplex.h

Die Member werden in der Implementierung wieder über die Member Initialization List deklariert

Listing 9.6: Implementierung des Parameterkonstruktors im Sourcefile

```
1 #include "komplex.h"
Komplex::Komplex(double re, double im)
3     : _re(re), _im(im)
{
5     //ctor
}
```

komplex/komplex.cpp

Durch das optionale zweite Argument der Deklaration in Listing 9.5 kann der Konstruktor auch als `Komplex aa(-7.8)` verwendet werden, wodurch eine komplexe Zahl mit Imaginärteil gleich 0 festgelegt wird.

Dieser Konstruktor wird auch benutzt, um ein Casting von `double` zu `Komplex` durchführen wie in Zeile 17 von Listing 9.1. Dies kann notwendig werden, falls einer Funktion eine `double`-Variable übergeben wird obwohl eine `Komplex`-Variable erwartet wird. Diese implizite Nutzung des Konstruktor kann durch das Schlüsselwort `explicit` in der Deklaration verhindert werden.

Eine Klasse kann mehrere Parameterkonstruktoren haben, selbstverständlich mit unterschiedlichen Parameterlisten.

9.2.3 Kopierkonstruktor

Eine Deklaration `Komplex b(a);` erfordert den entsprechenden Kopierkonstruktor.

Listing 9.7: Deklaration des Kopierkonstruktor im Headerfile

```
1 class Komplex
{
3 public:
    Komplex(const Komplex& org); // Kopierkonstruktor
5 };
```

komplex/komplex.h

Die Member werden in der Implementierung wieder über die Member Initialization List deklariert

Listing 9.8: Implementierung des Kopierkonstruktors im Sourcefile

```
1 #include "komplex.h"
  Komplex::Komplex(const Komplex& org)
3     : _re(org._re), _im(org._im)
  {
5 }
```

komplex/komplex.cpp

Der Zugriff auf private Member (`org._re`) ist hier erlaubt, da die Methode (also unser Konstruktor) zur gleichen Klasse gehört wie die Instanz `org`.

Der Kopierkonstruktor ist ein spezieller Parameterkonstruktor einer Instanz der Klasse als einziges Element der Parameterliste. Unser Kopierkonstruktor wird auch benutzt, wenn eine Instanz der Klasse `Komplex` per value (also als Kopie!) an eine Funktion/Methode übergeben wird. Wird die Instanz dagegen per reference übergeben, dann ist keinerlei Kopieroperation notwendig!

9.3 Der Destruktor

Der Destruktor wird automatisch aufgerufen wenn das Ende des Gültigkeitsbereiches einer Instanz dieser Klasse erreicht wird (schließende Klammer }). Diese Methode hat keinen Rückgabetyp. Die Funktion des Destruktors besteht darin, die von der Instanz belegten Ressourcen ordnungsgemäß freizugeben. Es gibt im Code stets soviele Desktruktoraufrufe wie es in Summe Konstruktoraufrufe gibt. Der Destruktor kann nicht direkt aufgerufen werden.

Listing 9.9: Deklaration des Destruktors im Headerfile

```
1 class Komplex
  {
3 public:
      virtual ~Komplex();
5 };
```

komplex/komplex.h

Falls der Destruktor automatisch vom Compiler generiert wird oder, wie bei uns, einen leeren Funktionskörper enthält, dann werden die Destruktoren der Member (und der Basisklassen) aufgerufen. Das Schlüsselwort `virtual` sollte bei Klassenhierarchien zum Destruktor hinzugefügt werden um die korrekte Freigabe aller Ressourcen der Hierarchie zu garantieren.

Listing 9.10: Implementierung des Destruktors im Sourcefile

```
1 #include "komplex.h"
  Komplex::~Komplex()
3 {
      //dtor
5 }
```

komplex/komplex.cpp

Bei eigenem dynamischen Resourcenmanagement muß der Destruktor diese Ressourcen freigeben. Ansonsten wird Speicher nicht wieder freigegeben und irgendwann sind selbts 64 GB voll.

9.4 Der Zuweisungsoperator

Eine Zuweisung `c = d;` wird vom Compiler als `c.operator=(d)` umgesetzt, es wird also eine Methode `Komplex::operator=` in unserer Klasse benötigt. Dieser Zuweisungsoperator ist die erste „normale“ Methode, d.h., mit Rückgabetyp, unserer Klasse.

Listing 9.11: Deklaration des Zuweisungsoperators im Headerfile

```
  class Komplex
2 {
  public:
4     Komplex& operator=(const Komplex& rhs);        // Zuweisungsoperator
  };
```

komplex/komplex.

Die Implementierung der Zuweisung bei unserer Klasse ist nicht schwer. Der Test `if ( this!=&rhs)` verhindert die Selbstzuweisung im Falle von `c=c` , mit `this` als dem Pointer auf die aktuelle Instanz (bei uns also `c` und somit ist hier `this==&c`).

Listing 9.12: Implementierung des Zuweisungsoperators im Sourcefile

komplex/komplex.cpp

```
1 #include "komplex.h"
2 Komplex& Komplex::operator=(const Komplex& rhs)        // Assignment operator
3 {
4     if ( this!=&rhs)                // Avoids self-assignment (critical with dynnamic
          memory)
5     {
6         _re = rhs._re;
7         _im = rhs._im;
8     }
9     return *this;
10 }
```

Die dreifache Verwendung von `Komplex` in Zeile 2 mag verwirrend erscheinen, aber diese bezeichnen Rückgabetyp, Klassenname und Typ des Inputparameters. Der Rückgabetyp `Komplex&` erfordert die Rückgabe einer Referenz auf die aktuelle Instanz (wieder `c`), realsiert in Zeile 9. Damit sind Mehrfachzuweisungen wie `f = c = d;` , aber auf Funktionsaufrufe wie `print(c=d);` möglich.

Es können auch mehrere Zuweisungsoperatoren mit anderen Klassen als Parameter definiert werden. Mit jedem neuen Zuweisungsoperator erfolgt eine *Operatorüberladung*, d.h., gleicher Operator aber andere Parameterliste.

9.5 Compilergenerierte Methoden

Folgende Methoden werden vom Compiler automatisch generiert wenn diese in der Klassendeklaration nicht aufscheinen:

- Kopierkonstruktor und Movekonstruktor (mit der eigenen Klasse)
- Zuweisungsoperator und Movezuweisung (mit der eigenen Klasse)
- Destruktor

Solange man in seiner Klasse nur einfache Datentypen, Klassen oder Container der STL für die Member benutzt sind die compilergenerierten Methoden völlig ausreichend. Auch eigene Klassen können dabei als Typ von Membern fungieren solange diese die notwendigen Methoden besitzen. Es empfiehlt sich die standardmäßig generierten Methode per `=default` zu kennzeichnen, siehe *Rule of Five*[2].

Listing 9.13: Rule of Five: explizit alles default

```
class Komplex
{
public:
   // Rule of five
    Komplex(const Komplex&  org)            = default; // Copykonstruktor
    Komplex(      Komplex&& org)            = default; // Movekonstruktor
    Komplex& operator=(const Komplex&  rhs) = default; // Copy-Zuweisungsoperator
    Komplex& operator=(      Komplex&& rhs) = default; // Move-Zuweisungsoperator
    ~Komplex()                              = default; // Destruktor
}
```

Sobald Sie eine eigene dynamische Speicherverwaltung Ihrer Member verwenden wollen (`new`/`delete[]` oder `malloc`/`free`), dann müssen Sie obige Methoden selbst implementieren (deep copy vs. shallow copy)!

Will man obige (oder andere) Methoden explizit verbieten[3], dann ist der Deklaration ein `= delete;` anzuschließen. Ein generelles Verbot unseres Kopierkonstruktors wäre also im Headerfile via `Komplex(const Komplex& org) = delete;` zu erreichen.

[2] https://en.cppreference.com/w/cpp/language/rule_of_three
[3] https://en.cppreference.com/w/cpp/language/function#Deleted_functions

9.6 Zugriffsmethoden

Wenn die Member einer Klasse alle public wären, dann könnte man auf diese z.B., über `c._re` direkt zugreifen. Jede Änderung der Membernamen oder deren Speicherung würde ein Adaptieren aller Codes mit direktem Zugriff nach sich ziehen. Daher sollen **Member immer private** sein und nur über Zugriffsmethoden der Klasse erreichbar sein.

Diese, oft als Getter-/Settermethoden bezeichneten, Zugriffsmethoden werden aus Effizienzgründen normalerweise gleich als Inlinemethoden in der Klassendeklaration implementiert. Das folgende Listing zeigt die Setter- (schreibender Zugriff) und Gettermethode (lesender Zugriff) für den Realteil, die Methoden für den Imaginärteil sind analog. Die Getter-/Settermethoden können beliebig bezeichnet werden.

Listing 9.14: Deklaration und Implemenierung de Getter/Setter im Headerfile

```
class Komplex
{
public:
    //                |--  Membervariablen dieser Instanz werden nicht veraendert!
    double Get_re() const
    {
        return _re;
    }
    void Set_re(double val)
    {
        _re = val;
    }
};
```

komplex/komplex.h

Das Listing 9.15 zeigt den Einsatz der Getter-/Settermethoden. Insbesondere ist bei der Deklaration der Gettermethoden (also beim Lesen) darauf zu achten, daß diese Methoden als `const` gekennzeichnet werden. Ansonsten können im gesamten Code keinerlei konstante Instanzen der Klasse verwendet werden.

Listing 9.15: Demonstration der Getter/Setter

```
{
  Komplex d(3.2, -1);
  double dd = d.Get_re();              // get real part of 'd'
  d.Set_re(5.2);                       // new value for  real part of 'd'

  const Komplex b(-1.2, 3);            // 'b' is a constant instance
  cout << b.Get_re();                  //  ==> only constant methods allowed for 'b'
}
```

9.7 Der Additionsoperator

In Listing 9.1 benötigen wir mehrfach den Additionsoperator (+), welchen wir durch *Operatorüberladung* für unsere Klasse `Komplex` nutzbar machen können.

Die Operation `a+b` kann vom Compiler als `a.operator+(b)` interpretiert werden wodurch eine entsprechende Methode `Komplex::operator+` mit entsprechendem Argument in der Klasse notwendig wird.

Listing 9.16: Deklaration des Additionsoperators im Headerfile

```
class Komplex
{
public:
    /**  Addiert die aktuelle Instanz mit einer zweiten komplexen Zahl
    * \param[in] rhs zweite komplexe Zahl
    * \return     \p *this + \p rhs
    */
    //                                      |--  Membervariablen dieser Instanz
        werden nicht veraendert!
    Komplex operator+(const Komplex& rhs) const;
};
```

komplex/komplex.h

Die Addition zweier komplexer Zahlen erfolgt durch die Addition der entsprechenden Real- und Imaginärteile. Beide Summanden bleiben unverändert, damit ist diese Methode `const`.

Listing 9.17: Implementierung der Methode zum Addieren

```
1 #include "komplex.h"
2 Komplex Komplex::operator+(const Komplex& rhs) const
3 {
4     Komplex tmp(_re+rhs._re, _im+rhs._im);
5     return tmp;
6 }
```

komplex/komplex.cpp

Der Rückgabetyp des Additonsoperators darf keine Referenz sein, da diese auf die temporäre Instanz `tmp` verweist welche nach dem Verlassen des Scopes der Methode nicht mehr existiert. Der Funktionskörper der Methode kann auch ganz kompakt als `return Komplex(_re+rhs._re, _im+rhs._im);` geschrieben werden.

Zeile 5 im Listing 9.18 zeigt, daß der Additionsoperator unserer Klasse, im Verbund mit dem Parameterkonstruktor, auch die Addition mit einer `double`-Zahl als zweitem Summanden erlaubt.

Listing 9.18: Additionsoperator erweitert

```
1 {
2   Komplex a(3.2, -1), b;
3   double dd = -2.1;
4
5   c = a + dd;              // a.operator+( Komplex(dd,0.0) )
6   c = dd + a;              // operator+(const double, const Komplex&)
7 }
```

Dies gelingt aber nicht mehr wenn der erste Summand eine `double`-Zahl ist, da dann die hier nicht mögliche Methode `double::operator+` benötigt würde. Der Ausweg besteht in einer Funktion (keine Methode!) `operator+(const double, const Komplex&)` welche die Summanden vertauscht und dann die Methode aus unserer Klasse aufruft.

Listing 9.19: Implementierung der Funktion zum Addieren mit `double` als erstem Summanden in Sourcefile

```
1 #include "komplex.h"
2 Komplex operator+(double lhs, const Komplex& rhs)
3 {
4 //     Komplex tmp(lhs+rhs.Get_re(), rhs.Get_im());
5 //   return tmp;
6     return rhs+lhs; // Ruft Methode operator+ der Klasse Komplex: rhs.operator(lhs)
7 }
```

komplex/komplex.cpp

Als Kommentar ist eine andere Möglichkeit der Implementierung angegeben.

Bemerkung: Eine allgemeinere Lösung der Implementierung für vertauschbare Operatoren arbeitet mit der Methode `operator+=` und der Funktion `operator+(const Komplex&, const Komplex&)`.

9.8 Der Ausgabeoperator

Um eine Klasseninstanz einfach via `cout << a` auszugeben ist die Funktion (keine Methode!) des Ausgabeoperators `operator<<` erforderlich, welche hierzu für eine Instanz unserer Klasse überladen wird.

Listing 9.20: Deklaration des Ausgabeoperators im Headerfile

```
1 #include <iostream>
2 using namespace std;
3 /** Ausgabeoperator fuer die Klasse Komplex.
4  * \param[in] s   ein beliebiger Ausgabestrom
5  * \param[in] rhs  die auszugebende Instanz der Klasse Komplex
6  */
7 ostream& operator<<(ostream& s, const Komplex& rhs);
```

komplex/komplex.h

Beim Ausgabeoperator ist der Rückgabetype wie auch der erste Parameter mit `ostream&` (Outputstream) festgelegt. Da `operator<<` eine Funktion ist darf diese nicht auf direkt auf die privaten Member der Klasse zugreifen, sondern nur über die Zugriffsmethoden.

Listing 9.21: Implementierung des Ausgabeoperators im Sourcefile

```
#include <iostream>
using namespace std;
#include "komplex.h"
ostream& operator<<(ostream& s, const Komplex& rhs)
{
    s << "("<< rhs.Get_re()<< ","<<rhs.Get_im() <<")";
    return s;
}
```

komplex/komplex.cpp

Bemerkung: Der Mechanismus der Datenkapselung (nur eigene Methoden dürfen auf Member zugreifen) läßt sich über `friend`-Funktionen aufweichen. Dies wird im Rahmen dieser LV nicht empfohlen.

Kapitel 10

Templates

Der englische Begriff *template* bedeutet auf deutsch *Schablone*. Eine Schablone wird bei der Produktherstellung dazu verwendet die gleiche Form immer wieder herzustellen (oder zumindest anzureißen), gleich ob das Material nun Metall, Karton oder Holz ist.

Unser Produkt sind Funktionen und Klassen mit den Datentypen als Material.

10.1 Template-Funktionen

In irgendeinem Stadium der Implementierung stellt man plötzlich fest, daß die gleichen Funktionen für verschiedene Datentypen gebraucht werden. In nachfolgendem, einfachen Beispiel sind dies das Maximium zweier Zahlen und das Maximum eines Vektors.

Listing 10.1: Header:Funktion für jeden Datentyp einzeln.

```
{
    int    mymax (const int    a, const int    b);
    float  mymax (const float  a, const float  b);
    double mymax (const double a, const double b);

    int    mymax_elem(const vector<int>&    x);
    float  mymax_elem(const vector<float>&  x);
    double mymax_elem(const vector<double>& x);
}
```

Die Implementierungen beider Funktionsgruppen unterscheiden sich nur im jeweiligen Datentyp, die Struktur der Implementierung ist ansonsten komplett identisch.

Listing 10.2: Source:Funktion für jeden Datentyp einzeln.

```
    int    mymax (const int a, const int b)
    {
       return  a > b  ?  a  : b;
    }

    float  mymax (const float a, const float b)
    {
       return  a > b  ?  a  : b;
    }

    double mymax (const double a, const double b)
    {
       return  a > b  ?  a  : b;
    }

    float  max_elem(const vector<float>&  x)
    {
     const int=x.size();
     assert(n>0);                    // --> richtiges Exeption-handling muss rein

     float vmax = x[0];
     for (int i=1; i<n; i++)
      {
        vmax = mymax(vmax,x[i]);    // Nutze die Funktion mymax
      }
```

```
27        return vmax;
        }
29 //        usw.
        ...
```

Neben dem unguten Gefühl, sich ungeschickt anzustellen, müssen algorithmische Verbesserungen in jeder Funktion einer Funktionsgruppe implementiert werden. Die ist wiederum fehleranfällig und der Gesamtcode schwerer zu warten.

10.1.1 Implementierung eines Funktions-Templates

C++ bietet die Möglichkeit, mit Hilfe von *Templates* (engl.: Schablonen) eine parametrisierte Familie verwandter Funktionen zu definieren. Ein Funktions-Template legt die Anweisungen einer Funktion fest, wobei statt eines konkreten Typs ein Parameter verwendet wird [KPP02, p.365]. Vorteile dieser Templates sind:

- Ein Funktionen-Template muß nur einmal kodiert werden.
- Einzelne Funktionen zu einem konkreten Parameter werden anhand des Templates automatisch erzeugt.
- Typunabhängige Bestandteile (der Algorithmus) können ausgetestet werden und funktionieren dann für die anderen Parametertypen.
- Es besteht immer noch die Möglichkeit, für bestimmte Typen spezielle Lösungen anzugeben.

Einem Funktions-Template wird das Präfix

`template <class T>`

vorangestellt. Der Parameter `T` ist hierbei der Typname welcher in der nachfolgenden Definition benutzt wird. Dabei schließt das Schlüsselwort `class` auch einfache Datentypen wie `int` oder `double` ein.

Unsere Funktionsfamilien werden nunmehr mit Templates so definiert und in ein File *tfkt.tpp* geschrieben.

Listing 10.3: Source: Templatefunktion.

```
{
2 //                                  tfkt.tpp
#include <vector>
4 #include <cassert>
using namespace std;
6 //#include "tfkt.hpp"

8 template <class T>
T  mymax (const T  a,const T  b)
10  {
   return  a > b  ?  a  : b;
12  }

14 template <class T>
T  max_elem(const vector<T>&  x)
16  {
   const int=x.size();
18   assert(n>0);           // --> richtiges Exeption-handling muss rein
   T vmax = x[0];
20   for (int i=1; i<n; i++)
    {
22     vmax = mymax(vmax,x[i]);
    }
24
   return vmax;
26  }
}
```

Die Deklaration ist dann im Headerfile *tfkt.h*

Listing 10.4: Header: Templatefunktion.

```
#ifndef FILE_TFKT
#define FILE_TFKT
#include <vector>
using namespace std;

    template <class T>
    T  mymax (const T  a, const T  b);

    template <class T>
    T  max_elem(const const vector<T>&  x);

#include "tfkt.tpp"                    // Include the sources in case of templates!!
#endif
```

Da in der STL eine Funktionsfamilie `max` bereits vorhanden ist, heißt unsere Funktionsfamilie `mymax`. Ansonsten ergebens sich Zweideutigkeiten bei der Auswahl der richtigen Funktion.

Die Anwendung unserer Funktionstemplates im Hauptprogramm ist relativ einfach.

Listing 10.5: Templatefunktion anwenden.

```
#include <iostream>
#include <vector>
#include "tfkt.hpp"                    // Header mit Templates und deren Sources
using namespace std;

int main()
{
 const int N = 10;
 vector<int>    ia(N);
 vector<float>  fa(N);
 vector<double> da(N);

 for (int i=0; i<N; i++)               // Vektoren initialisieren
  { ia[i] = i;
    fa[i] = i/(i+1.0f);
    da[i] = i/(i+1.0);
  }

 int    im = max_elem(ia);             // int
 cout << "   int-Feld(max):  " << im << endl;

 float  fm = max_elem(fa);             // float
 cout << " float-Feld(max):  " << fm << endl;

 double dm = max_elem(da);             // double
 cout << "double-Feld(max):  " << dm << endl;

 return 0;
}
```

Ein Funktions-Template kann auch mit mehreren Typparametern definiert werden.

```
    template<class A, class B>
    B& func(const int n, const A mm, vector<B>& v)
    {
    ...
    }
```

10.1.2 Implizite und explizite Templateargumente

Im Hauptprogramm auf Seite 87 wurde das richtige Templateargument `T`, und damit die konkrete Funktion, anhand der Funktionsparameter bestimmt. Falls dies nicht eindeutig ist, bzw. das Templateargument nicht in der Parameterliste der Funktion vorkommt, dann muß das Templateargument beim Funktionsaufruf explizit angegeben werden. Das folgene Codefragment demonstriert implizite und explizite Templateargumente in den Zeilen 12 und 16.

Listing 10.6: Templatefunktion anwenden.

```
#include <iostream>
#include <vector>
#include "tfkt.hpp"                    // Header mit Templates und deren Sources
using namespace std;

int main()
{
```

```
  const int N = 10;
  vector<double> da(N);
  ....                                  // Vektoren initialisieren

  double dm = max_elem(da);             // double, implicit instantiation

  float  a=5.455, fm;
  double b=-3.344;
  fm = mymax<float>(a,b);               // float,  explicit instantiation

  return 0;
}
```

Weitere Hinweise zu Typanpassungen bei Funktions-Templates sind in [KPP02, p.371, p.377] zu finden. Die implizite Typanpassung kann (auch zu Debuggingzwecken) beim `g++` mittels der Option `-fno-implicit-templates` unterbunden werden.

10.1.3 Spezialisierung

So schön unsere allgemeinen Funktions-Templates sind, sie sind nicht für alle denkbaren Typen einsetzbar:

- Das Funktions-Template enthält Anweisungen, die für bestimmte Typen nicht ausgeführt werden können.
- Die allgemeine Lösung, die das Template bereitstellt, liefert kein sinnvolles Ergebnis.
- Für bestimmte Typen gibt es bessere (schnellere, speicherschonendere) Lösungen.

Für solche Fälle läßt sich z.B., für unsere Funktion `mymax` eine Spezialisierung für C++-Strings implementieren.

Folgender Code funktioniert mit unserer Templatefunktion `mymax<T>` auch ohne Spezialisierung, da `operator>` für die Klasse `string` implementiert ist und einen lexikographischen Vergleich durchführt.

Listing 10.7: Templatefunktion ohne Spezialisierung anwenden.

```
#include <iostream>
#include <vector>
#include <string>
#include "tfkt.hpp"                     // Header mit Templates und deren Sources
using namespace std;

int main()
{
  const int N = 10;
  vector<string> da(N);
  ....                                  // Vektor of strings initialisieren

  max_str = mymax(name1,name2);         // lexicographic comparison
  cout << max_str << endl;

  max_str = mymax(a[5],a[9]);           // lexicographic comparison
  cout << max_str << endl;

  max_str = max_elem(N, a);             // lexicographic comparison
  cout << max_str << endl;

 return 0;
}
```

Will man den resultierenden lexikographischen Vergleich `operator>(string, string)` in `mymax` durch einen Vergleich der Stringlängen ersetzen, dann hilft folgende Spezialisierung:

Listing 10.8: Spezialisierung einer Templatefunktion.

```
template<>                              // Einleitung der Spezialisierung
string mymax(const string &s1, const string &s2 )
{
  if ( s1.size()>s2.size() ) return s1;
  else                       return s2;
}
```

Diese Funktion überlädt, bei (genau!) passenden Parametern das entsprechende Funktions-Template von `mymax`. Das Funktions-Template `max_elem` kann nun sogar mit unserer Spezialisierung arbeiten.

10.2 Template-Klassen

10.2.1 Ein Klassen-Template für `Komplex`

Analog zu den Funktions-Templates bietet C++ die Möglichkeit, Klassen-Templates zu definieren. Die Klassen-Templates werden in Anhängigkeit von einem noch festzulegendem Typ (oder mehreren Typen) konstruiert. Diese Klassen-Templates werden häufig bei der Erstellung von Klassenbibliotheken eingesetzt, wir werden im Zusammenhang mit der STL in §11 auf solche Klassenbibliotheken zugreifen.

Einem Klassen-Template ist der Präfix `template<class T>` vorangestellt, dem die eigentliche Klassendefinition folgt.

```
template<class T>
class X
{
...                         // Definition der Klasse X<T>
};
```

Der Name des Template für Klassen ist `X<T>`. Der Parameter `T` steht wieder für einen beliebigen (auch einfachen) Datentyp. Sowohl `T` als auch `X<T>` werden in der Klassendefinition wie normale Datentypen verwendet.

Zur Demonstration leiten wir aus der Klasse `Komplex` von Seite 76 das Klassen-Template `Komplex<T>` ab indem wir den original verwandten Datentyp `double` für die Vektorelemente durch `T` und `Komplex` durch `Komplex<T>` ersetzen.

Listing 10.9: Header der Templateklasse

```
template <class T>
class Komplex
{
public:
    Komplex();
    Komplex(T re, T im=0.0); // Parameterkonstruktor mit ein oder zwei Argumenten
    virtual ~Komplex();
    T Get_re() const
    {
        return _re;
    }
    void Set_re(T val)
    {
        _re = val;
    }
    T Get_im() const
    {
        return _im;
    }
    void Set_im(T val)
    {
        _im = val;
    }
    Komplex<T>& operator+=(const Komplex<T>& rhs);
    Komplex<T> operator+(const Komplex<T>& rhs) const;
    bool operator<(const Komplex<T>& rhs) const
    {
        return _re < rhs._re || ( _re == rhs._re && _im < rhs._im );
    }
    bool operator==(const Komplex<T>& rhs) const
    {
        return _re == rhs._re && _im == rhs._im ;
    }
    bool operator>(const Komplex<T>& rhs) const
    {
        return !( *this < rhs || *this== rhs ) ;
    }
protected:
private:
    T _re; //!< Realteil
    T _im; //!< Imaginaerteil
```

```
};
#include "komplex.tpp"
```

v_8c/komplex.h

Die Deklaration der Methoden erfolgt dann analog, hier sei exemplarisch der Plus-Operator präsentiert im File.

Listing 10.10: Implementierung der Methode einer Templateklasse

```
template <class T>
Komplex<T>& Komplex<T>::operator+=(const Komplex<T>& rhs)
{
    _re += rhs._re;
    _im += rhs._im;
    return *this;    // this ist ein Pointer auf die aktuelle Instanz
}
template <class T>
Komplex<T> Komplex<T>::operator+(const Komplex<T>& rhs) const
{
    Komplex<T> tmp(*this);
//    tmp += rhs;
//    return tmp;
    return tmp+=rhs;
}
```

v_8c/komplex.tpp

Wichtig ist, daß vor jeder Methodendefinition der Präfix `template<class T>` steht.

10.2.2 Mehrere Parameter

Analog zu den Funktions-Templates können auch Klassen-Templates mit mehreren Typparametern definiert werden.

```
template<class T1, class T2>
class X
{
...                    // Definition der Klasse X<T1,T2>
};
```

Desweiteren ist ein Template mit einem festgelegtem Parameter, einem Argument, möglich [KPP02, p.397ff],

```
template<class T, int n>
class Queue {...};
```

mit welchem `n`, z.B., die Größe eines immer wieder gebrauchten, internen temporären Feldes bezeichnet. So deklariert dann

```
Queue<double, 100> db;
```

eine Instanz der Klasse `Queue<double, 100>`, welche mit `double`-Zahlen arbeitet und intern, z.B., ein temp. Feld `double tmp[100]` verwaltet.

Das Template-Argument kann mit einem Default-Parameter definiert werden,

```
template<class T, int n=255>
class Queue {...};
```

wodurch `Queue<double> db;` ein temp. Feld der Länge 255 intern verwalten würde.

Gleitkommazahlen sind nicht als Template-Argumente erlaubt, wohl aber Referenzen auf Gleitkommazahlen.

10.2.3 Umwandlung einer Klasse in eine Template-Klasse

Ein kleiner Leitfaden um aus einer Klasse (`Komplex`) eine Template-Klasse (`Komplex<T>`) zu generieren:

1. `template <class T>` vor die Klassendeklaration schreiben. (**.h*)
2. Den zu verallgemeinernden Datentyp durch den Template-Parmeter `T` ersetzen. (**.h, *.tpp*) Achtung, vielleicht wollen Sie nicht an allen Stellen diesen Datentyp ersetzen.
3. Ersetze in allen Parameterlisten, Returntypen und Variablendeklarationen den Klasstentyp `myclass` durch die Template-Klasse `myclass<T>`. (**.h, *.tpp*)
4. Vor jede Methodenimplementierung `template <class T>` schreiben. (**.tpp*) Desgleichen vor Funktionen, welche die Klasse `myclass::` als Parameter benutzen. (**.tpp*) Bei `friend`-Funktionen muß man `template <class T>` auch im Deklarationsteil vor der Funktion angeben. (**.h*)
5. In der Methodenimplementierung `myclass::` durch `myclass<T>::` ersetzen. (**.tpp*)
6. Im Headerfile das Sourcefile includieren, also `#include "myclass.tpp"`, oder gleich alles in das Headerfile schreiben (nicht empfohlen). (**.h*)

10.2.4 Template-Klasse und `friend`-Funktionen

Wenn wir den Ausgabeoperator `operator<<` der Template-Klasse `Komplex<T>` als `friend`-Funktion deklarieren wollen, dann ändert sich nichts im Implementierungsteil (außer, daß wir direkt auf die Member zugreifen können). Im Deklarationsteil unserer Klasse müssen wir für diese `friend`-Funktion einen anderen Template-Parameter (hier `S`) benutzen.

```
template <class T>
class Komplex
{
  public:
    ...
    Komplex<T> operator+(const Komplex<T>& rhs ) const;
    ...
    template <class S>              // template parameter for following deklaration
    friend ostream& operator<<(ostream& s, const Komplex<S>& rhs);
    ...
};
```

Diese Art der Deklaration funktioniert ohne Einschränkungen.

10.3 Einschränkung der Datentypen bei Templates

Die Codes in §10.1-10.2 akzeptieren vorerst jeden Templateparameter `T`. Eine Deklaration `Komplex<string> as("real","imag");` mit `T = string` ist zwar möglich und selbst die Addition würde in Listing 10.10 funktionieren, jedoch stellt man sich unter einer komplexen Zahl etwas anderes vor. Nebenbei, beim Versuch einer Multiplikation würde der Compiler mit einer recht kryptischen Fehlermeldung reagieren.

Es gibt zwei grundsätzliche Möglichkeiten die Anforderungen an den Templateparameter beim Kompilieren frühzeitig zu überprüfen:

- mit Type Traits und `static_assert` [ab C++11],
- mit Concepts [ab C++20].

Im folgenden werden diese Möglichkeiten für eine Klasse `Komplex<T>` mit der Einschränkung, daß `T` eine Gleitkommazahl sein muß demonstriert.

10.3.1 Überprüfung des Templatedatentyps mit Type Traits

Zur Laufzeit besteht die Möglichkeit Datentypen miteinander zu vergleichen bzw. diese auf bestimmte Eigenschaften zu testen. Diese `type_traits`[1] enthalten den für uns nützlichen Test auf

[1] `https://en.cppreference.com/w/cpp/header/type_traits`

eine Gleitkommazahl[2] welcher in Zeile 8 des Listing 10.11 benutzt wird. Das Testergebnis (true oder false) könnte man zur Laufzeit in allen Methoden/Funktionen auswerten.

Listing 10.11: Templateklasse (teilw.) mit Type Traits

```
1  #include <cassert>        // static_assert
2  #include <type_traits>    // std::is_floating_point<T>()
3  template <class T>
4  class Komplex
5  {
6      // check type of template at compile time
7      static_assert(std::is_floating_point<T>(),
8                    "Vector elements have to be floating point numbers.");  // C++11
9  public:
10     //
11     Komplex<T>& operator+=(const Komplex<T>& rhs);
12 private:
13     T _re; //!< Realteil
14     T _im; //!< Imaginaerteil
15 };
```

v_8c_cpp17/komplex.h

Es aber komfortabler wenn bereits der Compiler die Übersetzung des Codes abbricht sobald der Templateparameter die gewünschte Bedingung nicht erfüllt. Dazu wird im Listing `static_assert` benutzt, der zweite Parameter ist der (informative!) Fehlerstring welchen der Compiler im Fehlerfalle ausgibt.

Eine Deklaration von `Komplex<string> as("real","imag");` oder `Komplex<int> ai(1,2);` hat einen Abbruch der Compilation zur Folge, d.h., die Methoden werden natürlich auch nicht generiert. Funktionen welche den Templateparameter `T` ohne die Klasse benutzen, müßten bei Bedarf eine separate Überprüfung wie im Zeile 8 enthalten.

10.3.2 Überprüfung des Templatedatentyps mit Concepts

Seit C++20 kann man über Concepts und constraints[3] Anforderungen an den Templateparameter zu stellen. Listing 10.12 demonstriert dies für unsere Templateklasse und Listing 10.13 für die enstprechende Implementierung der Methoden und Funktionen. In jedem Falle wird ein Kompilierfehler ausgegeben, falls der Templateparameter `T` keine Gleitkommazahl ist.

Listing 10.12: Templateklasse (teilw.) unter Nutzung von Concepts

```
1  #include <concepts>                          // floating_point
2  template <typename T>
3  requires std::floating_point<T>             // C++20: concepts
4  class Komplex
5  {
6  public:
7      Komplex<T> operator+(const Komplex<T>& rhs) const;
8  private:
9      T _re; //!< Realteil
10     T _im; //!< Imaginaerteil
11 };
```

v_8c_cpp20/komplex.h

Listing 10.13: Templatemethoden unter Nutzung von Concepts

```
1  template <class T>
2  requires std::floating_point<T>             // C++20: concepts
3  Komplex<T> Komplex<T>::operator+(const Komplex<T>& rhs) const
4  {
5      Komplex<T> tmp(*this);
6      return tmp+=rhs;
7  }
```

v_8c_cpp20/komplex.tcc

Neben den in `<concepts>` vordefinierten[4] Concepts wie dem hier benutzen `floating_point`[5] kann man auch eigene Concepts erstellen, siehe dazu [Gri21b, p.10 and §4.1].

[2] https://en.cppreference.com/w/cpp/types/is_floating_point
[3] https://en.cppreference.com/w/cpp/language/constraints
[4] https://en.cppreference.com/w/cpp/concepts
[5] https://en.cppreference.com/w/cpp/concepts/floating_point

Kapitel 11

Einführung in die STL

11.1 Was ist neu?

Die **S**tandard **T**emplate **L**ibrary (STL) enthält Schablonen (engl.: templates) zur Datenspeicherung, Schablonen für oft gebräuchliche Algorithmen sowie verallgemeinerte Pointer (Iteratoren) um beide miteinander zu verbinden.

Die Standardbibliothek enhält in der STL (Standard Template Library) schon viele brauchbare Klassen und Methoden, welche man nicht immer wieder neu implementieren muß. Wir gehen nur auf einige wenige Möglichkeiten der STL ein, es sei auf [Mey98, §49], [Mey97, §6.4.1],[Yan01, §10], [KPP02, 567-711], [KS02] verwiesen. Die Klassen und Methoden der Standardbibliothek sind im Namensraum `std` untergebracht, sodaß, z.B., `cout` via den Namespace `std::cout` aufgerufen werden muß (nicht zu Verwechseln mit dem Scope-Operator) oder man gibt einzelne Komponenten des Namensraumes über `using std::cout` bzw. den gesamten Namensraum über `using namespace` frei.

Die STL basiert auf drei grundsätzlichen Konzepten: Containern, Iteratoren und Algorithmen. Die Container beinhalten (mehrere) Objekte deren Templates konkretisiert werden, und in welchen mit Hilfe von Iteratoren navigiert werden kann. In Fortführung dessen sind Algorithmen Funktionen welche auf Containern (mittels der Iteratoren) bestimmte Operationen durchführen.

Listing 11.1: Intro STL

```
#include <vector>                       //  vector<T>
#include <algorithm>                    //  find()
using namespace std;
...
{
    vector<int> v(10);                  // container
    // Init v
    ....
    vector<int>::iterator it;           // iterator
    it = find(v.begin(), v.end(), 4);   // algorithm, iterators
}
```

Das kurzes Codefragment 11.1 enthält alle drei Konzept der STL:

- *Container*[1]: unterschiedliche Datenstrukturen zur Speicherung von Elementen, z.B., `vector`, `list`, `stack`.
  Container sind eigenen Templateklassen mit eigenen Methoden (z.B., `erase`, `assign`) und zugehörigen Iteratoren.
- *Algorithmen*[2]: unabhängig vom konkreten Containers implementierte Algorithmen, z.B., `find`, `find_if`, `sort`, `unique`, `count`, `accumulate`.
  $\Longrightarrow$ Algorithmen und Container haben nichts miteinander zu tun.

[1] `http://www.cplusplus.com/reference/stl/`
[2] `http://www.cplusplus.com/reference/algorithm/`

- *Iteratoren*[3]: sind das Bindeglied zwischen Containern und Algorithmen. Ein klassischer C-Pointer `int*` kann als Iterator betrachtet und auch als solcher benutzt werden.

In obigem Code 11.1 sind `v.begin()` und `v.end()` Methoden des Containers `vector<int>` welche Iteratoren auf das erste und das hinterletzte Element zurückliefern, also den Definitionsbereich für den Algorithmus `find`. Der Algorithmus liefert wiederum einen Iterator auf das gefundenen Element zurück welcher in `it` gespeichert wird. Mit der Dereferenzierung `*it` können wir direkt auf den Containereintrag zugreifen.

Das Finden des größten Elements eines Vektors, für welches wir in Listing 10.3 auf Seite 10.3 eine eigene Templatefunktion geschrieben haben, läßt sich mittels der STL ganz einfach lösen.

Listing 11.2: Größter Vektoreintrag: STL anwenden.

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

int main()
{
 const int N = 10;
 vector<int>    ia(N);
 vector<double> da(N);
          // Vektoren initialisieren
 ...

 vector<int>::iterator pm;
 pm = max_element(ia.begin(),ia.end());
 int im = *pm;
 cout << "   int-Feld(max):  " << im << endl;

 double dm = *max_element(da.begin(),da.end());
 cout << "double-Feld(max):  " << dm << endl;

 return 0;
}
```

- Zeile 14 definiert den notwendigen Iterator.
- Zeile 15 bestimmt den Iterator des maximalen Elements des gesamten Vektors.
- Zeile 16 dereferenziert den Iterator und speichert den Wert.
- Zeile 19 kombiniert Zeilen 14-16 in einem Schritt für den entsprechenden Container.
- Der Algorithmus `max_element` benötigt den Vergleichsoperator `operator<` für die Elemente des Containers.

Einige allgemeine Hinweise zur Benutzung der STL-Algorithmen:

1. Jeder STL-Algorithmus benötigt einen, von Iteratoren begrenzten, Input-bereich eines Containers auf welchen der Algorithmus anzuwenden ist, z.B., `ia.begin(),ia.end()` in obigem Beispiel.
2. Manche STL-Algorithmen benötigen zusätzlich einen Output-bereich dessen Container **eine ausreichende Länge** haben muß, z.B., in `copy`[4].
3. Viele STL-Algorithmen haben mehrere Aufrufmöglichkeiten, einmal mit einem Standardvergleichoperator und zum zweiten mit einer spezifischen Vergleichsfunktion. Siehe dazu die Info zu `max_element`[5] und das Beispiel dort. Näheres im nächsten Abschnitt.

11.2 Wie benutze ich `max_element` ?

Wir beschränken uns in diesem Abschnitt auf den Algorithmus `max_element` um einige beachtenswerte Details hervorzuheben.

[3] `http://www.cplusplus.com/reference/iterator/`
[4] `http://www.cplusplus.com/reference/algorithm/copy/`
[5] `http://www.cplusplus.com/reference/algorithm/max_element`

11.2.1 Container mit Standarddatentypen

In Fortsetzung obigen Beispiels betrachten wir den folgenden C++11-Code[6] der Anwendung von `max_element` auf einen Container `vector<int>`.

Nutzung des Standardvergleichsoperators Da die Elemente des Containers vom Typ `int` sind, nutzt die Standardversion des Algorithmus den Vergleichsoperator `<` dieser Klasse `int` in Zeile 10 des Listings 11.3.

Listing 11.3: Algorithmus mit Standardvergleichoperator `operator<`

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  int main()
6  {
7      vector<int> v{-1,-3,4,-3,-5,-1,4};                    // C++11
8  // STL-Algorithmus mit Standardvergleich  operator<
9      vector<int>::iterator it;
10     it = max_element(v.begin(),v.end());                    // int::operator<
11     cout << " max : " << *it << endl;
12     return 0;
13 }
```

v_stl_intro/main.cpp

Die erweiterte Version des Algorithmus erfordert eine boolesche Vergleichsfunktion welche auf drei verschiedene Weisen in den nächsten Beispielen definiert wird.

Nutzung einer Vergleichsfunktion In den Zeilen 6-9 wird eine boolesche Vergleichsfunktion `fkt_abs_less` definiert, welche in Zeile 15 zum betragsmäßigen Vergleich der Elemente benutzt wird.

Listing 11.4: Algorithmus mit Vergleichsfunktion

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  //
6  bool fkt_abs_less(int a, int b)                            // Vergleichsfunktion
7  {
8      return abs(a)<abs(b);
9  }
10 int main()
11 {
12     vector<int> v{-1,-3,4,-3,-5,-1,4};                    // C++11
13 //  ...
14 // STL-Algorithmus mit Vergleichsfunktion
15     auto it2 = max_element(v.cbegin(),v.cend(), fkt_abs_less);      // function
16     cout << "|max|: " << *it2 << endl;
17     return 0;
18 }
```

v_stl_intro/main.cpp

Nutzung eines Funktors In den Zeilen 6-12 wird eine Funktorklasse (-struktur) `funktor_abs_less` deklariert, welche nur den Klammeroperator `operator()` als Methode mit booleschem Rückgabewert besitzt. Dieser Klammeroperator wird beim Vergleich im Algorithmus in Zeile 18 benutzt.

Listing 11.5: Algorithmus mit Funktor

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  //
6  struct myclass
7  {
8      bool operator() (int a, int b)  const                  // Funktor
9      {
10         return abs(a)<abs(b);
```

[6] imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Beispiele/v_stl_intro.zip

```
11      }
} funktor_abs_less;
13 int main()
{
15      vector<int> v{-1,-3,4,-3,-5,-1,4};                  // C++11
//  ...
17 // STL-Algorithmus mit Vergleichsfunktion
      auto it3 = max_element(v.cbegin(),v.cend(), funktor_abs_less); // functor
19      cout << "|max|: " << *it3 << endl;
      return 0;
21 }
```

v_stl_intro/main.cpp

Nutzung einer Lambda-Funktion Mit C++11 kann die Definition der kurzen Funktion `fkt_abs_less` aus §11.2.1 gleich in die Rufzeile des Algorithmus geschrieben werden. Diese Lambda-Funktion wird in Zeile 10 definiert und gleichzeitig angewendet. Mehr zu Lambda-Funktionen in Alex Allains sehr gutem Tutorial[7].

Listing 11.6: Algorithmus mit Lambda-Funktion

```
1 #include <iostream>
#include <algorithm>
3 #include <vector>
using namespace std;
5 int main()
{
7      vector<int> v{-1,-3,4,-3,-5,-1,4};                  // C++11
//  ...
9      auto it4 = max_element(v.cbegin(),v.cend(),                  // C++11:
          lambda-function
                        [](int a, int b) -> bool { return abs(a)<abs(b); }
11                    );
      cout << "|max|: " << *it4 << endl;
13      return 0;
}
```

v_stl_intro/main.cpp

11.2.2 Container mit Elementen der eigenen Klasse

Wenn wir statt der Standarddatentypen die eigene Klasse `komplex` aus §9.1 benutzen, dann muß unsere Klasse einige Anforderungen erfüllen. Insbesondere müssen zumindest die Vergleichsoperatoren `operator<` und `operator==` vorhanden sein. Desgleichen ist der Zuweisungsoperator `operator=` für Kopieroperationen nötig (es reicht normalerweise der vom Compiler generierte Zuweisungsoperator).

Nutzung des Standardvergleichsoperators

Listing 11.7: Algorithmus mit Standardvergleichoperator `operator<`

```
1 #include <iostream>
#include <algorithm>
3 #include <vector>
#include "komplex.h"
5 using namespace std;
int main()
7 {
      vector<Komplex> v{-1,-3,4,-3,-5,-1,4};              // C++11
9      vector<Komplex>::iterator it;
      it = max_element(v.begin(),v.end());                  // int::operator<
11      return 0;
}
```

v_stl_intro2/main.cpp

Der Standardvergleichsoperator muß für unsere Klasse deklariert und definiert sein:

Listing 11.8: Standardvergleichoperator `operator<` für `Komplex`

```
#include <iostream>
2 bool Komplex::operator<(const Komplex& rhs) const
{
4      return (_re<rhs._re) || ( _re==rhs._re && _im<rhs._im );
}
```

v_stl_intro2/komplex.cpp

[7] http://www.cprogramming.com/c++11/c++11-lambda-closures.html

Nutzung einer Vergleichsfunktion In den Zeilen 7-10 wird eine boolesche Vergleichsfunktion `fkt_abs_less` definiert welche in Zeile 14 zum betragsmäßigen Vergleich der Elemente benutzt wird.

Listing 11.9: Algorithmus mit Vergleichsfunktion

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "komplex.h"
using namespace std;
//
bool fkt_abs_less(const Komplex& a, const Komplex& b)       // Vergleichsfunktion
{
    return abs(a)<abs(b);
}
int main()
{
    vector<Komplex> v{-1,-3,4,-3,-5,-1,4};                  // C++11
    auto it2 = max_element(v.cbegin(),v.cend(), fkt_abs_less);      // function
    return 0;
}
```

v_stl_intro2/main.cpp

Hierzu benötigen wir noch die Funktion `abs` für unsere Klasse.

Listing 11.10: Inline-Funktion `abs`

```
inline double abs(const Komplex& rhs)
{
    return sqrt(rhs.Get_im()*rhs.Get_im()+rhs.Get_re()*rhs.Get_re());
}
```

v_stl_intro2/komplex.h

Natürlich hätten wird `abs` auch als Methode der Klasse `Komplex` implementieren können, dann müßte man nur die Vergleichsfunktion etwas umschreiben (`return a.abs()<b.abs();`)

Die Verwendung von Funktor bzw. Lambda-Funktion erfolgt analog zu §11.2.1.

11.3 Einige Grundaufgaben und deren Lösung mit der STL

Die Grundidee bei allen STL-Algorithmen besteht darin, daß nur Iteratoren auf Container übergeben werden. Damit besteht auch für den Algorithmus keinerlei Möglichkeit, die Länge eines Containers zu verändern. Dies kann nur über Methoden des Containers erfolgen, z.B., `erase` oder `resize`. Manche Algorithmen erfordern spezielle Eigenschaften der Iteratoren wie wahlfreien Zugriff (engl.: random access) welche nicht von allen Containeriteratoren bereitgestellt werden können. Für diese steht dann eine entsprechende Methode zur Verfügung, siehe §11.3.2.

Die folgenen Beispiele demonstrieren die Nutzung der STL für einige Grundaufgaben an Hand der Container `vector` (random access iterator) und `list` (bidirectional iterator).

Listing 11.11: Rahmencode für STL-Demonstration

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;
//  Ausgabe von vector
template <class T>
ostream& operator<<(ostream& s, const vector<T>& x)
{
    for (unsigned int i=0; i<x.size(); ++i)
    {
        s << x[i] << "  ";
    }
    return s;
}
//  Ausgabe von list
template <class T>
ostream& operator<<(ostream& s, const list<T>& x)
{
    for (auto px: x )                    // C++11: range-for
    {
        s << px << "  ";
    }
```

```
    return s;
}
int main()
{
    vector<int>  v{-1,-3,4,-3,-5,-1,4};                    // C++11
    list<int> lv{-1,-3,4,-3,-5,-1,4};                     // C++11
    return 0;
}
```

v_stl_intro3/main.cpp

Der *range-for*-Loop in Zeilen 20-23 ist die allgemeingültige Lösung um über den Gesamtbereich eines beliebigen Containers zu itererieren, d.h., dies wäre auch für die Ausgabe des vectors anwendbar. Da der `list`-Iterator nicht wahlfrei ist, kann der for-Loop aus den Zeilen 10-13 nicht auf diesen angewandt werden.

11.3.1 Kopieren von Daten

Listing 11.12: Kopieren eines Vektors

```
vector<int> b(v.size());                    // reserve enough space for b !!
copy(v.cbegin(),v.cend(),b.begin());
```

v_stl_intro3/main.cpp

Listing 11.13: Kopieren einer Liste

```
list<int> lb(lv.size());       // reserve enough space for lb !!
copy(lv.cbegin(),lv.cend(),lb.begin());
```

v_stl_intro3/main.cpp

11.3.2 Aufsteigendes Sortieren von Daten

Listing 11.14: Aufsteigendes Sortieren eines Vektors

```
sort(b.begin(), b.end());          //  operator<   is needed
```

v_stl_intro3/main.cpp

Listing 11.15: Aufsteigendes Sortieren einer Liste

```
lb.sort();                        //  operator<   is needed
```

v_stl_intro3/main.cpp

11.3.3 Mehrfache Elemente entfernen

Der Algorithmus `unique` entfernt nur diejenigen Elemente aus einem Container, welche darin **benachbart** gespeichert sind. Wenn man also aus einem Container alle mehrfachen Elemente, unabhängig von deren originaler Nachbarschaftsbeziehung, entfernen will, dann muß der Container vorher sortiert werden wie in §11.3.2. Die Länge des Containers muß anschließend mit der Methode `erase` korrigiert werden.

Listing 11.16: Entfernen mehrfacher Elemente aus einem geordneten Vektor

```
vector<int>::iterator it;
it=unique (b.begin(), b.end());          //  operator==   is needed
b.erase(it,b.end());                     //  Vektor kuerzen
b.shrink_to_fit();                       //  C++11
```

v_stl_intro3/main.cpp

Listing 11.17: Entfernen mehrfacher Elemente aus einer geordneten Liste

```
list<int>::iterator lit;
lit=unique (lb.begin(), lb.end());       //  operator==   is needed
lb.erase(lit,lb.cend());                 //  Liste kuerzen
```

v_stl_intro3/main.cpp

11.3.4 Kopieren von Elementen welche einem Kriterium nicht entsprechen

Wie beim Kopieren muß der Zielcontainer groß genug sein. In unsererm Beispiel kopiert der Algorithmus `copy_if` nur diejenigen Elemente welche nichtnegativ (Boolesche Funktion `IsNotNegative`) sind. Die Länge des Containers muß anschließend mit der Methode `erase` korrigiert werden.

Listing 11.18: Boolesche Funktion

```
1 bool IsNotNegative (const int b) { return b>0; }
```

v_stl_intro3/main.cpp

Listing 11.19: Kopieren von bestimmten Elementen aus einem Vektor

```
1     vector<int> c(b.size());
      it=copy_if (b.cbegin(),b.cend(),c.begin(), IsNotNegative);
3     c.erase(it,c.end());                    //  Vektor kuerzen
      c.shrink_to_fit();                      //  C++11
```

v_stl_intro3/main.cpp

Listing 11.20: Kopieren von bestimmten Elementen aus einer Liste

```
1     list<int> lc(lb.size());
      lit=copy_if (lb.cbegin(),lb.cend(),lc.begin(), IsNotNegative);
3     lc.erase( lit,lc.cend() );              //  Liste kuerzen
```

v_stl_intro3/main.cpp

11.3.5 Absteigendes Sortieren von Elementen

Listing 11.21: Absteigendes Sortieren eines Vektors

```
1     sort(c.begin(), c.end(),greater<int>());          // via int::operator>
```

v_stl_intro3/main.cpp

Listing 11.22: Absteigendes Sortieren einer Liste

```
1     lc.sort(greater<int>());          // via int::operator>
```

v_stl_intro3/main.cpp

Statt des Funktors `greater<int>()` (benutzt intern `int::operator>`) kann natürlich auch eine beliebige andere Boolesche Funktion benutzt werden (Listings 11.4-11.6).

11.3.6 Zählen bestimmter Elemente

Listing 11.23: Zählen bestimmter Elemente eines Vektors

```
      const int nc = count(v.cbegin(), v.cend(), -3 );
```

v_stl_intro3/main.cpp

Listing 11.24: Zählen bestimmter Elemente einer Liste

```
1     const int lnc = count(lv.cbegin(), lv.cend(), -3 );
```

v_stl_intro3/main.cpp

11.3.7 Sortieren mit zusätzlichem Permutationsvektor

Der Sortieralgorithmus der STL liefert keinen Indexvektor der Umordnung zurück. Folgende zwei Lösungen werden bei *stackoverflow*[8] vorgeschlagen.

- Statt des zu sortierenden `vector<double>` wird `vector< pair<double,int> >` sortiert, wobei der zweite Parameter mit den Indizes $0, 1, 2, \ldots$ inialisiert wird. Hinterher muß man nur noch die zweiten Parameter in einen Indexvektor extrahieren.

[8] `http://stackoverflow.com/questions/1577475/c-sorting-and-keeping-track-of-indexes`

- Meine favorisierte Lösung ist die folgende, welche eine Lambda-Funktion nutzt und statt den Datenvektor umzuordnen nur den Permutationsvektor der Umordnung zurückliefert. Auf das Minimum des Datenvektors kann anschließend mittels `v[idx[0]]` zugegriffen werden (Code[9]).

Listing 11.25: Sortieren mit Permutationsvektor

```
template <typename T>
vector<size_t> sort_indexes(const vector<T> &v)
{
  // initialize original index locations
  vector<size_t> idx(v.size());
  for (size_t i = 0; i != idx.size(); ++i) idx[i] = i;

  // sort indexes based on comparing values in v
  sort(idx.begin(), idx.end(),
       [&v](size_t i1, size_t i2) {return v[i1] < v[i2];});

  return idx;
}

//   Application
vector<double> v = {....};        // initialize v
vector<size_t>  idx =  sort_indexes(v);
```

11.3.8 Ausgabe der Elemente eines Containers

Wir nehmen als Beispielcontainer wieder `std::vector<T>`, die Codes sind leicht auf andere Container übertragbar. Abgesehen von der recht plumpen Lösung über eine Funktion `void print(vector<T> const & x)` ist das Definieren eines Ausgabeoperators die beste Lösung.

Listing 11.26: Vektorausgabe: Version 1

```
#include <iostream>
#include <vector>

template <class T>
ostream& operator<<(ostream &s, vector<T> const &x)
{
   for (const auto &pi: x)
   {
      s << pi << "  ";
   }
   return s;
}

//   Application
vector<double> v{....};        // initialize v
cout << v << endl;
```

Der range-for-Loop in obigem Listing ersetzt den Loop von `cbegin(x)` bis `cend(x)` und kann auch durch ein direktes Kopieren auf den Ausgabestrom ersetzt werden.

Listing 11.27: Vektorausgabe: Version 2

```
#include <iostream>
#include <iterator>
#include <vector>

template <class T>
ostream& operator<<(ostream &s, vector<T> const &x)
{
  copy(cbegin(x), cend(x), ostream_iterator<T>(s, "  "));
  return s;
}

...
//   Application
vector<double> v{....};        // initialize v
cout << v << endl;
...
```

[9] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Beispiele/sort_index.cpp

11.4 Allgemeine Bemerkungen zur STL

Die wichtigsten Quelle zur STL sind `cppreference.com`[10] oder `cplusplus.com`[11], da man hier schnell Details zu einzelnen Algorithmen, zu Iteratoren oder Methoden von Containern findet.

Die meisten STL-Algorithmen existieren in einer Standardvariante, in welcher ein Operator der Containerelemente benutzt wird (`operator<` in `sort`) und einer weiteren Variante, in welcher ein entsprechender Operator (oder eine Funktion) übergeben wird, meist über eine Lambda-Funktion. Genauso gibt es häufig Zwillings-Algorithmen wie `count` und `count_if`.

11.5 *Parallelität in der STL [C++17]

Ab C++-17 sind zu vielen Algorithmen auch parallele Versionen verfügbar welche als zusätzlichen ersten Parameter eine *execution policy* entgegennehmen, siehe dazu Versionen (2) und (4) von `sort`[12] sowie `std::execution::par`[13]. Dabei werden alle verfügbaren Cores der CPU(s) benutzt, dank Hyperthreading werde normalerweise doppelt so viele Threads für die Parallelisierung benutzt. Diese Anzahl der benutzen Threads ist nicht steuerbar, im Gegensatz zu einer OpenMP-Parallelisierung.
Achtung: Beim Linken muß das Flag `-ltbb` nach den Objektfiles hinzugefügt werden.
Gegebenenfalls muß die Option `-std=c++17` beim Compilieren angegeben werden.

Das Sortieren eines Containers läßt sich wie folgt beschleunigen.

Listing 11.28: Paralleles Sortieren

```
// thread_17
#include <algorithm>
#include <execution>            // execution policy
#include <vector>

...
{
    size_t const N = 1<<25;
    vector<double> v(N);
    iota(v.begin(), v.end(), 1);
    std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});

    sort(std::execution::par, v.begin(), v.end());
}
...
```

In obigem Beispiel `thread_17`[14] und einem Vector mit 33.554.432 Elementen beträgt die erzielte Beschleunigung (SpeedUp) gegenüber der sequentiellen Variante 7.5 für eine 8-core CPU (AMD Ryzen 7 3800X) bzw. 16.5 für eine 32-core CPU (AMD EPYC 7551P).
Die erzielten Beschleunigungen ändern sich mit der Anzahl der Elemente und mit deren Datentyp (Speicherbandbreite; memory bandwidth).

[10] `https://en.cppreference.com/w/`
[11] `http://www.cplusplus.com`
[12] `https://en.cppreference.com/w/cpp/algorithm/sort`
[13] `https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t`
[14] `https://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS21/thread_17.zip`

Kapitel 12

Klassenhierarchien

12.1 Ableitungen von Klassen

Dieser Abschnitt der Vorlesung orientiert sich an [Cor93, §7] und wir beginnen mit den objektorientierten Sprachunterstützungen von C++.

Wir stellen uns ein Möbelhaus mit drei Arten von Angestellten vor: den normalen Angestellten, den Verkäufern und den Managern. Von allen benötigt die Buchhaltung den Namen, die Lohnverrechnung ist jedoch unterschiedlich. So wird der Angestellte (`Worker`) auf Stundenbasis bezahlt, der Verkäufer (`salesPerson`) auf Stundenbasis und Verkaufsprovision, der Manager (`Manager`) erhält ein wöchentliches Gehalt.

Natürlich könnte man nun 3 Klassen/Strukturen konventionell programmieren und dann mit `switch`-Anweisungen immer unterscheiden, welche Variante der Lohnverrechnung nun benutzt werden soll. Dieses Konzept ist aber sehr fehleranfällig bzgl. der Erweiterbarkeit unserer Angstelltenklassen (Verkaufsmanager, Aktienindexmanager), da dann **immer alle** `switch`-Anweisungen geändert werden müssen - und irgendetwas vergißt man immer.

Der objektorientierte Ansatz stellt erstmal die Frage nach den gemeinsamen Eigenschaften unserer Angestellten und dann erst nach den Unterschieden. Die Gemeinsamkeiten bei allen sind der Name und die notwendige Lohnberechnung. Unterschiede bestehen in der Art und Weise der Lohnberechnung und der dafür notwendigen Daten. Zusätzlich könnte `salesPerson` die stundenbasierte Lohnberechnung von `Worker` nutzen.

12.1.1 Design einer Klassenhierarchie

Die für die Angestellten des Möbelhauses betrachteten gemeinsamen und zusätzlichen Eigenschaften stehen in enger Verbindung mit dem Vererbungskonzept im objektorientierten Programmieren. Dort deklariert man Basisklassen, von denen weitere Klassen, mit zusätzlichen Eigenschaften ableitet werden. In [Sch02, §8.2] ist dieses Vererbungskonzept sehr schön erläutert:

- Eine Ableitung einer Klasse repräsentiert eine **IS-A**-Relation.
- Eine Membervariable repräsentiert eine **HAS A**-Relation

Diese Relationen werden bei der Ableitung einer Klasse `B` von der Basisklasse `A` folgendermaßen ersichtlich. Klasse `B` ist eine (**IS-A**) Basisklasse `A` mit den zusätzlichen Eigenschaften (**HAS A**) des Members `_bb` und der Methode `fkt_b()`.

Listing 12.1: Ableitung einer Klasse

```
1 class A                  // Basisklasse
{
3    public:
       A(const string& name) : _ss(name)  {};
5      string GetString() const
```

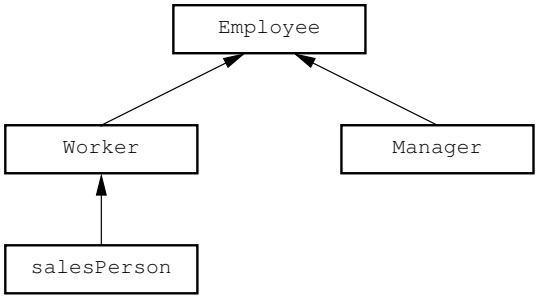
```
        {return _ss;};

    private:
        string _ss;
};

class B: public A       // abgeleitete Klasse erbt alle Methoden/Member von A
{
    public:
        B(const string& name, int b)
            : A(name), _bb(b)  {};  // Aufruf des Konstruktors der Basisklasse

        int fkt_b() const           // zusaetzliche Methode (Eigenschaft)
            {return _bb*_bb;};

    private:
        int _bb;                    // zusaetzlicher Member (Eigenschaft)
};

int main()
{
    A ia("Basis");
    B ib("bin abgeleitet",3); // String-Parameter wird an Basisklasse "durchgereicht"

    cout << ia.GetString() << endl;
    cout << ib.GetString() << endl; // abgeleitete Klasse benutzt Basisklassenmethode

//    cout << ia.fkt_b();           // fkt_b ist keine Basisklassenmethode
    cout << ib.fkt_b();

    return 0;
}
```

In Zeile 31 des Listings 12.1 ruft die Instanz `ib` der Klasse `B` die Funktion `A::GetString() const`, also eine Methode der Basisklasse. Dies ist wegen des `public`-Vererbung in Zeile 12 möglich.

In unserem Falle des Möbelhauses wären alle Angestellen erstmal Beschäftige, d.h., wir benötigen eine Basisklasse `Employee`. Dann können wir sagen:

- `Employee` **HAS-A** Name.
- `Worker` **IS-A**n `Employee` und `Worker` **HAS-A** (zusätzlich) einen Stundenlohn und eine Arbeitszeit.
- `salesPerson` **IS-A** `Worker` und `salesPerson` **HAS-A** (zusätzlich) eine Umsatzbeteiligung am erbrachten Umsatz.
- `Manager` **IS-A**n `Employee` und `Manager` **HAS-A** Wochengehalt (zusätzlich).

Dies ergibt folgende Hierarchy von Klassen. Man beachte, daß wir bis jetzt noch überhaupt keine konkrete Implementierung angesprochen haben. Vielmehr betrachten wir nur Eigenschaften der zu handhabenden Objekte. Dieses **Design** der Klassenhierarchie muß immer zu Beginn eines OO-Programmes stehen. Ein gründlich erarbeitetes Design erspart zeitraubende und fehleranfällige Umstrukturierungen der Klassenhierarchie in einem späteren Projektstadium.



12.1.2 Die Basisklasse

Unser Entwurf für die Basisklasse `Employee` sieht folgendermaßen aus:

Listing 12.2: Basisklasse `Employee`

```
#include <string>
#include <iostream>
using namespace std;
/**  Abstrakte Basisklasse eines allgemeinen Angestellten einer Verkaufsstelle
```

```
5  */
   class Employee
7  {
       public:
9          /** Parameter constructor
            @param[in]  name  Name des Angestellten
11         */
           explicit Employee(const string& name);
13         /** Default destructor */
           virtual ~Employee();
15         /**  Gibt die Daten der aktuellen Instanz aus.
              @param[in,out] s  Ausgabestrom
17         */
           virtual void print(ostream& s) const;
19         /**  Berechnet das Gehalt.
              @return  Gehalt.
21         */
           virtual float payment() const = 0;   // rein virtuell
23 //            {return 0.0f; };               // da war die Methode nur virtuell
           /**  Zugriff auf Name des Angestellten.
25            @return  Name.
           */
27         string getname() const {return _name;}
       private:
29         string _name;         //!< Name der Person
   };
```

v_9b/employee.h

Man beachte, daß die Methode für die Gehaltsausgabe, `payment()` in der Basisklasse keine sinnvolle Berechnung ausführen kann, da keinerlei Daten zur Gehaltsberechnung vorhanden sind. An dieser Stelle kann man wie in Zeile 22 erstmal eine Dummy-Methode implementieren welche 0.0 (oder −12345) zurückgibt. Das Schlüsselwort `virtual` weist darauf hin, daß diese Methode *virtuell* ist und von abgeleiteten Klassen überladen werden darf. Deklariert man, wie in Zeile 28 des Listings 12.2, die Methode `payment()` mit dem obskur erscheinenden Konstrukt `= 0` als *rein virtuelle* Methode, dann muß diese Methode in den abgeleiteten Klassen überladen werden, siehe §12.2.

12.1.3 Die abgeleiteten Klassen

Unser `Worker` **IS-A**n `Employee` and **HAS-A** Stundenlohn und Arbeitszeit. Diese Zusatzeigenschaften erfordern zusätzliche Methoden zur ihrer Handhabung und erlauben nun eine sinnvolle Methode `payment()`. Gleichzeitig *erbt* `Employee` alle Eigenschaften von `Worker`, u.a. den Namen und die Zugriffsfunktion darauf.

Listing 12.3: abgeleitete Klasse `Worker`

```
1  #include "employee.h"
   /**      Normaler Arbeiter (Packer) in einer Verkaufsstelle.
3  */
   class Worker : public Employee
5  {
       public:
7          /** Parameter constructor
            @param[in]  name  Name des Angestellten
9           @param[in]  hours Arbeitsstunden
            @param[in]  wageHours  Stundenlohn
11         */
           Worker(const string& name, float hours, float wageHours);
13         /** Default destructor */
           virtual ~Worker();
15         /**  Gibt die Daten der aktuellen Instanz aus.
              @param[in,out] s  Ausgabestrom
17         */
           void print(ostream& s) const;
19         /**  Berechnet das Gehalt.
              @return  Gehalt.
21         */
           float payment() const
23             {return _hours*_wageHours ; };
       protected:
25     private:
           float _hours; //!< Arbeitsstunden
27         float _wageHours; //!< Member Stundenlohn
   };
```

v_9b/worker.h

Die neuen Eigenschaften (**HAS-A**) sind in Zeilen 26 und 27 obigen Codefragmentes deklariert. Zeile 4 beinhaltet die Ableitung der neuen Klasse von der Basisklasse (**IS-A**). Das Schlüsselwort

`public` in Zeile 4 erlaubt den Zugriff auf Basisklassenmethoden wie `getname()` über Instanzen (Variablen) der Klasse `Worker`. Dies erlaubt dann folgendes Codefragment:

Listing 12.4: Public Methode der Basisklasse verwenden.

```
1  Worker ab("Ritchie Valens");
2  cout << ab.getname() << endl;
```

Eine Klassenableitung der Form `class Worker : private Employee` oder `class Worker : protected Employee` verbietet die Benutzung der Methode `getname()` in obiger Form. Während die Verwendung von `protected` wenigstens die Nutzung der Methode innerhalb der Klasse `Worker` erlaubt, ist selbst dies bei der Ableitung als `private` nicht möglich.

In analoger Weise leiten wir die Klasse `Manager` ab.

Listing 12.5: abgeleitete Klasse `Manager`

```
#include "employee.h"
/**      Manager einer Verkaufsstelle
*/
class manager : public Employee
{
    public:
        /** Parameter constructor
         @param[in]  name  Name des Angestellten
         @param[in]  wageWeek Wochengehalt
        */
        manager(const string& name, float wageWeek);
        /** Default destructor */
        virtual ~manager();
        /**  Gibt die Daten der aktuellen Instanz aus.
           @param[in,out] s  Ausgabestrom
        */
        void print(ostream& s) const;
        /**  Berechnet das Gehalt.
           @return  Gehalt.
        */
        float payment() const {return _wageWeek;};
    protected:
    private:
        float _wageWeek; //!< Wochengehalt
};
```

v_9b/manager.h

Die noch fehlende Klasse `salesPerson` benötigt die Klasse `Worker` als Basisklasse, da die dortige, stundenweise Lohnverrechnung auch hier wieder gebraucht wird.

Listing 12.6: abgeleitete Klasse `salesPerson`

```
#include "worker.h"
/**      Verkaeufer in einer Verkaufsstelle
*/
class salesPerson : public Worker
{
    public:
        /** Parameter constructor
         @param[in]  name  Name des Angestellten
         @param[in]  hours Arbeitsstunden
         @param[in]  wageHour  Stundenlohn
         @param[in]  commission Umsatz
         @param[in]  percent  Umsatzbeteiligung in Prozent
        */
        salesPerson(const string& name, int hours, float wageHour,
                    float commission, float percent);
        /** Default destructor */
        virtual ~salesPerson();
        /**  Gibt die Daten der aktuellen Instanz aus.
           @param[in,out] s  Ausgabestrom
        */
        void print(ostream& s) const;
        /**  Berechnet das Gehalt.
           @return  Gehalt.
        */
        float payment() const
         {return Worker::payment() + _commission*_percent;};
    protected:
    private:
        float _commission; //!< Umsatz
        float _percent; //!< Umsatzbeteiligung in Prozent
};
```

v_9b/salesperson.h

Die Implementierungen sämtlicher Methoden der vier Klassen sind in den entsprechenden Quelltextfiles und Headerfiles zu finden. Hervorzuheben ist an dieser Stelle die Methode `payment` der Klasse `salesPerson`:

v_9b.zip

Listing 12.7: Expliziter Aufruf einer Basisklassenmethode.

```
float salesPerson::payment() const
{
   return Worker::payment() + _commission*_percent;
}
```

Hier benötigen wir explizit den Scope-operator `::` um die `payment`-Methode der Basisklasse `Worker` zu benutzen. Was würde passieren, falls wir stattdessen `return payment() + comission*SalesMade;` programmieren würden?

Eine einfache Demonstration unserer neuen Klassen ist in `v_9b/main.cpp` nachzulesen, siehe auch die zugehörige Dokumentation[1].

v_9b/main.cpp

12.2 Polymorphismus

Der Grund, daß der Code `v_9b` (ohne Benutzung von `virtual`) nicht wie gewünscht die spezifischen Gehälter ausgerechnet hat liegt darin, daß die schon in der Basisklasse vorhandene Methode `payment()` (und auch `payment()`) in den abgeleiteten Klassen redefiniert wurde. Dies hat natürlich auf die Basisklasse keine Auswirkung, sodaß eine über einen Basisklassenpointer adressierte Instanz konsequenterweise immer die in der Basisklasse deklarierte Methode aufruft.

Die Alternative zur Redefinition ist ein Ersatz: Soll eine Methode der Basisklasse durch eine Methode der abgeleiteten Klasse *komplett ersetzt* werden, deklariert man diese Methode in der Basisklasse als *virtuell*. Das entsprechende Schlüsselwort ist `virtual`. Ist eine Methode einer Basisklasse virtuell, dann sind alle gleichnamigen Methoden in abgeleiteten Klassen automatisch virtuell, ohne daß das Schlüsselwort angegeben werden muß.

12.2.1 Nutzung virtueller Methoden

Mit solchen virtuellen Methoden läßt sich erreichen, daß der Code `v_9b` so funktioniert, daß immer die richtige Gehaltsberechnungsmethode in Zeile 38 aufgerufen wird. Dazu müssen wir nur die Basisklasse leicht abändern.

A4/employ.hpp

Listing 12.8: Virtuelle Methoden

```
//                      employ.hpp
...
class Employee
{
 public:
  Employee();
  Employee(const string& name);
  const string& getname() const;
  void info();                          // Namen ausgeben
  virtual void payment();       //  NEW,  virtuell
  virtual ~Employee()           //  NEW,  jetzt notwendig
                {};
 private:
  string name;
};
...
```

Im Verzeichnis *A4* kann jetzt der Code mit
`g++ -Wall -o a4_1 a4_1.cpp employ.cpp`
übersetzt und gelinkt werden.

A4/a4_1.cpp

[1] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Beispiele/v_9b/doxygen/html/class_employee.html

Listing 12.9: Polymorphie ausgenutzt

```
int main()
{
 Employee  simple("Josef Gruber");
 WageEmployee hilf("Heiko");
 hilf.setWage(20.0f); hilf.setHours(7.8f);
 SalesPerson emp("Gundolf Haase");
 emp.setWage(hilf.getWage());  emp.setHours(hilf.getHours());
 emp.setComission(0.05f); emp.setSales(10000.0f);
 Manager man("Max Planck");
 man.setSalary(1000.0f);
 //      Basisklassenzeiger
 cout << endl << "            Basisklassenzeiger" << endl;
 vector<Employee*> liste(4);            // array von Pointern auf Employee
 liste[0] = &simple;
 liste[1] = &hilf;
 liste[2] = &emp;
 liste[3] = &man;
 cout << endl << "   Nur die Namen ausgeben" << endl;
 for (size_t i=0; i<liste.size(); ++i)
  {
      liste[i]->info();
  }
//              NEU !!
 cout << endl << endl << "   Name und (spezifisches) Gehalt klappen  j e t z t" <<
     endl;
 for (size_t i=0; i<liste.size(); ++i)
  {
      liste[i]->payment();
  }
 return 0;
}
```

A4/a4_1.cpp

Die Änderung in Zeile 10 des Listings 12.8 erlaubt es, zur Laufzeit zu entscheiden, welche Methode `payment()` aufgerufen werden soll.
In unserem Code bedeutet dies für den letzten Zählzyklus in den Zeilen 25-28,

```
for (size_t i=0; i<liste.size(); ++i)
    { liste[i]->payment(); }
```

daß folgende Methoden gerufen werden:

$i = 0$ `Employee :: payment()`
$i = 1$ `Worker :: payment()`
$i = 2$ `salesPerson :: payment()`
$i = 3$ `Manager :: payment()`

Diese Möglichkeit eine Methode für eine Instanz aufzurufen ohne dessen Typ genau zu kennen, nennt man *Polymorphie*. Dieser griechische Begriff bezeichnet die Fähigkeit sich von verschiedenen Seiten zu zeigen oder verschiedene Formen anzunehmen [Cor93, p.161].

Besitzt eine Klasse mind. eine virtuelle Methode, dann muß der Destrukor ebenfalls virtuell sein, was in unserer Basisklasse `Employee` der Fall ist (Der Destruktor ist dort auch gleich definiert.). Damit man die Notwendigkeit hierfür einsieht, betrachten wir das Ende des Gültigkeitsbereiches von `liste`. Für jedes Element des Arrays von Basisklassenzeigern wird dann der Destruktor aufgerufen. Ohne einen virtuellen Destruktor in der Basisklasse würde dann stets der Destruktor der Basisklasse aufgerufen. Dies bewirkt bei abgeleiteten Klassen mit dynamisch allokiertem Speicher, daß dieser Speicher nicht freigegeben wird. Bei einem virtuellen Destruktor der Basisklasse wird wiederum erst zur Laufzeit entscheiden, welcher Destruktor aufgerufen wird sodaß ein sauberes Speichermanagement möglich ist (es wird erst der Destruktor der abgeleiteten Klasse aufgerufen welcher seinerseits den Destruktor der Basisklasse implizit aufruft).

12.2.2 Rein virtuelle Methoden

Die Methode `payment()` der Basisklasse `Employee` ist nur als Platzhalter implementiert und führt keine sinnvollen Berechnungen aus. Eigentlich benötigen wir diese Methode nur, um anzuzeigen, daß eine Methode `payment()` aus abgeleiteten Klassen verwendet werden soll. Dies ist nicht sonderlich elegant, denn der einzige Zweck von `Employee :: payment` besteht darin, *nie* aufgerufen zu werden.

Eine Alternative dazu besteht in einer Methode, die weder semantisch noch physisch existiert. Eine derartige Methode ist *rein virtuell* und sie wird definiert, indem man der Deklaration ein `=0` anhängt. Die neue Deklaration der Klasse `Employee` ist dann:

A4/employ2.hpp

Listing 12.10: Virtuelle Methoden

```
//                      employ2.hpp
class Employee
{
 public:
  Employee();
  Employee(const string& name);
  const string& getname() const;
  void info();
  virtual void payment() = 0;           // NEW, rein virtuell
  virtual ~Employee() {};
 private:
  string name;
};
```

Im Definitionsfile muß natürlich die Definion von `Employee :: payment` entfernt werden. A4/employ2.hpp

Die rein virtuelle Deklaration von `Employee :: payment` hat weitere Konsequenzen:

- Es läßt sich keine Instanz (Variable) der Klasse `Employee` mehr erzeugen. Es können jedoch nach wie vor Basisklassenpointer auf die Klasse `Employee` deklariert werden, um auf Instanzen abgeleiteter Klassen zu zeigen. Deshalb muß auch Zeile 14 des Hauptprogramms von Seite 108 in
  `liste[0] = &hilf`
  geändert und die Definition in Zeile 3 gestrichen werden. A4/a4_2.cpp
- Jetzt *muß* `payment` in den abgeleiteten Klassen definiert werden.
- `Employee` ist eine *abstrakte Klasse*, da sie eine rein virtuelle Funktion enthält und somit von ihr keine Instanzen deklariert werden können.
- Klassen, von welchen Instanzen deklariert werden können heißen *konkrete Klassen*. Somit ist, z.B., `Worker` eine konkrete Klasse.

Falls eine Klasse von einer abstrakten Basisklasse abgeleitet wird und die darin enthaltenen rein virtuellen Methoden nicht definiert sind, dann erbt die neue Klasse auch diese reine virtuellen Funktionen und wird damit selbst eine abstrakte Klasse. Bei einer normalen virtuellen Methode in der (dann konkreten) Basisklasse würde in diesem Falle einfach die Methode der Basisklasse verwendet.

12.2.3 Dynamische Bindung - Polymorphismus

In unserem Beispiel fungieren die virtuelle Methode `payment` und der virtuelle Destruktor als dynamische Methoden, im Gegensatz zu den bislang verwendeten statischen Funktionsaufrufen. Diese dynamische Bindung zum Programm wird über die Virtual Method Table (VMT) realisiert.

Eine besondere Eigenschaft der dynamischen Bindung stellt die Möglichkeit dar, das Verhalten bereits existierenden Codes nachträglich zu verändern, ohne daß die bereits existierenden Teile neu compiliert werden müssen. Bereits übersetzte Module können so ohne Veränderung des Codes oder einer Neukompilierung *nachträglich* um neue Datentypen erweitert werden. Dies wollen wir an einem Beispiel demonstrieren.

Zuerst separieren wir den Teil des Hauptprogrammes, welcher die Polymorphie ausnutzt, in eine extra Funktion `PrintListe` welche im File *liste.cpp* definiert ist.

Listing 12.11: Polymorphie in Funktion ausgenutzt

```
#include <iostream>
#include <vector>
#include "employ2.hpp"  // Nur die Basisklasse ist notwendig!!
using namespace std;
void PrintListe(const vector<Employee*>& liste )
{
 cout << endl << "   Nur die Namen ausgeben" << endl;
 for (size_t i=0; i<liste.size(); ++i)
  {
    liste[i]->info();
  }
```

```
//                NEU !!
13    cout << endl << endl << "   Name und (spezifisches) Gehalt klappt  j e t z t" <<
          endl;
    for (size_t i=0; i<liste.size(); ++i)
15  {
      liste[i]->payment();
17  }
}
```

A4/liste.cpp

Die Files *liste.cpp* und *employ2.cpp* werden compliert
`g++ -c liste.cpp employ2.cpp`
und im folgenden benutzen wir nur noch die beiden Objektfiles *liste.o* und *employ2.o*.

Von der Klasse `Manager` leiten wir eine Klasse `BoxPromoter` mit der neuen Eigenschaft der Bestechlichkeit ab. Also: `BoxPromoter` **IS-A** `Manager` und `BoxPromoter` **HAS-A** Eigenschaft der Bestechlichkeit.

Listing 12.12: BoxPromoter als abgeleitetet Klasse

```
1  //                    bestech.hpp
   // Demonstration, damit bei virtuellen Funktionen ein uebersetzter
3  // Programmteil nachtraelich veraedert werden kann.
   #include "employ2.hpp"
5  class BoxPromoter : public Manager
   {
7    public:
      BoxPromoter();
9     explicit BoxPromoter(const string& nm);                    // Konstructor
      //void setSalary(const float salary);          // --> Manager::setSalery
11    float computePay() const;                      // virtuell
      void payment() const;                          //  (auch virtual loesbar)
13    void setBestechung(const float bestechung);    // N E W
    private:
15    float bestechung_;                             // Bestechungsgeld
   };
```

A4/bestech.hpp

Die Zeile 13 ist auskommentiert, da die Klasse `BoxPromoter` die `setSalary` ihrer Basisklasse `Manager` benutzt. Das neue Hauptprogramm sieht dann recht kurz aus:

Listing 12.13: Hauptprogramm für Polymorphie

```
1  #include <iostream>
   #include <vector>
3  #include "employ2.hpp"
   #include "bestech.hpp"          // NEW
5  #include "liste.hpp"            // !! employ2.hpp wird 2-mal eingebunden!!
   using namespace std;
7  int main()
   {
9   WageEmployee hilf("Heiko");
    hilf.setWage(20.0f); hilf.setHours(7.8f);
11  SalesPerson emp("Gundolf Haase");
    emp.setWage(hilf.getWage());  emp.setHours(hilf.getHours());
13  emp.setComission(0.05f); emp.setSales(10000.0f);
    Manager man("Max Planck");
15  man.setSalary(1000.0f);
    //      Basisklassenzeiger
17  cout << endl << "            Basisklassenzeiger" << endl;
    vector<Employee*> liste(4);            // array von Pointern auf Employee
19   // NEW
    BoxPromoter boxer("Larry King");
21  boxer.setBestechung(19300.0);
    liste[0] = &boxer;
23  liste[1] = &hilf;
    liste[2] = &emp;
25  liste[3] = &man;
    PrintListe(liste);
27  return 0;
   }
```

A4/a4_3.cpp

In Zeile 31 setzen wir den Basisklassenpointer `liste[0]` auf die Instanz der neuen Klasse und in Zeile 36 wird die bereits als Objektfile vorliegende Funktion `PrintListe` angewandt. In den Zeilen 4-6 wird das Headerfile *employ2.hpp* einmal direkt und zweimal indirekt über *bestech.hpp* und *liste.hpp* eingebunden. Solange nur Deklarationen in *employ2.hpp* stehen bleibt dies folgenlos. Da wir aber z.B., den Destruktor der Klasse `Employee` auch gleich im Headerfile definiert (=implementiert) haben, würde diese Methode dreimal definiert, was eine Fehlermeldung nach sich zieht. Dies hätte zur Folge, daß beim Compilieren von *a4_3.cpp* die Klasse `Employee` dreimal deklariert

wird. Eine elegante Lösung des Problems besteht darin, den Quelltext des Headerfiles nur beim ersten Inkludieren einzubinden, was durch das *Header Guarding* erreicht wird. .

Listing 12.14: Header Guarding

```
//
#ifndef FILE_EMPLOY2
#define FILE_EMPLOY2
...                                  // Quelltext von employ2.hpp
#endif
```

Dieses Vorgehen garantiert, daß die Deklarationen und Definitionen von *employ2.hpp* genau einmal pro Compiliervorgang eingebunden werden.

Wir compilieren nun die beiden neuen Files und linken sie mit den bereits vorhandenen Objektfiles (Bibliotheken, bei größeren Projekten) zusammen.

```
g++ -o a4_3 a4_3.cpp bestech.cpp liste.o employ2.o
```

Die dynamische Bindung ermöglicht es, Bibliotheken mit Klassen und Methoden zu erstellen, die von anderen Programmierern erweitert werden können. Sie müssen dafür lediglich die Include-Dateien (**.h*, **.hpp*) und den compilierten Code (**.o*, *lib*.a*) bereitstellen, welche die Hierarchie der Klassen und die Methoden enthalten. Andere Programmierer können damit von Ihren Klassen eigene Klassen ableiten und die von ihnen deklarierten virtuellen Methoden neu definieren. Methoden, die ursprünglich nur Ihre Klassen verwendet haben, arbeiten dann auch mit den neuen Klassen [Cor93, p.164].

12.2.4 Nochmals zu Copy-Konstruktor und Zuweisungsoperator

Sie werden bislang die Copy-Konstruktoren und Zuweisungsoperatoren für die Klassen unserer Hierarchy vermißt haben - oder auch nicht. Das Fehlen derselben ist zum einen der Übersichtlichkeit geschuldet und zum anderen, daß für unsere, sehr einfachen, Klassen mit einfachen Datentypen die vom Compiler automatisch eingefügten Standardmethoden ausreichend sind.

Sobald wir aber kompliziertere Datenstrukturen haben, siehe §12.1, sind diese beiden Methoden **unbedingt notwendig** [Mey98, §16]. Ein erster, aber *falscher* Ansatz des Zuweisungsoperators für `Worker` sähe so aus:

Listing 12.15: Falscher Zuweisungsoperator

```
//                      f a l s c h e r  Zuweisungsoperator
Worker & Worker :: operator=(const Worker & orig)
{
 if ( this != &orig )
  {
    wage  = orig.wage;
    hours = orig.hours;
  }
 return *this
}
```

Der Fehler besteht darin, daß `Employee::name` nicht mit den Daten des Originals belegt wurde. Eine direkte Zuweisung ist nicht möglich, da `name` als `private` deklariert wurde und somit nur innerhalb der Basisklasse darauf zugegriffen werden kann. Die einzig saubere Lösung ist der folgende Code.

Listing 12.16: Korrekter Zuweisungsoperator

```
//                      k o r r e k t e r  Zuweisungsoperator
Worker & Worker :: operator=(const Worker & orig)
{
 if ( this != &orig )
  {
    Employee::operator=(orig);
    wage  = orig.wage;
    hours = orig.hours;
  }
 return *this
}
```

Die neue Anweisung in Zeile 6 ruft den entsprechenden Zuweisungsoperator der Basisklasse auf, in diesem Falle die Methode `this->Employee::operator=`. Zwar erwartet diese Methode ein Argument vom Typ `Employee`, aber da `Worker` von dieser Klasse abgeleitet ist, wird eine implizite Typkonvertierung durchgeführt (die Basisklasse holt sich alles Notwendige aus der abgeleiteten Klasse).

Beim Copykonstruktor muß man die entsprechenden Basisklasseninitialisierer aufrufen. In unserem Falle wäre dies.

Listing 12.17: Korrekter Kopierkonstruktor

```
1 //                    richtiger Copy-Konstruktor
Worker & Worker :: Worker(const Worker & orig)
3         : Employee(orig), wage(orig.wage), hours(orig.hours)
{}
```

12.3 Anwendung der STL auf polymorphe Klassen

12.3.1 Container mit Basisklassenpointern

v_10c.zip

Dank des Polymorphismus lassen sich Instanzen von verschiedenen Klassen einer Hierarchie in einem Container zusammenfassen. Hierzu müssen Basisklassenpointer statt der konkreten Klasse in der Definition des Containers angegeben werden, siehe Zeile 5 in Listing 12.18. Die einzelnen Elemente müssen dann Pointer auf konkrete Klassen sein und mit `new` muß deren *Speicher explizit angefordert* werden, siehe Zeilen 7-11. Dies heißt auch, daß dieser angeforderte Speicher für jeden einzelnen Pointer wieder *explizit freigegeben* werden muß (Zeilen 14-16), bevor der Gültigkeitsbereich des Container erreicht ist (Zeile 19). Anderfalls bleibt dieser Speicherbereich allokiert, ist aber nicht mehr erreichbar(orphants).

Listing 12.18: Vektor mit Basisklassenpointern

```
#include <iostream>
2 #include <vector>
#include "fahrzeug.h"
4 using namespace std;
    vector<Fahrzeug*> v;
6   //          |-- we need a pointer to an instance
    v.push_back( new Raba(3600, 4000) );
8   v.push_back( new Opel(1450) );
    v.push_back( new MAN(1200, 12000) );
10  v.push_back( new Smart(950) );
    v.push_back( new Smart(1100));
12  cout << v << endl;
//   Free those element that have been allocated by new
14  for (auto it: v)
    {
16      delete it;
    };
18  return 0;
}   // free the memory storing the pointers in vector 'v' at the end of the scope
```

v_10c/main.cpp

Die Ausgabe des Containers erfolgt mittels des Ausgabeoperators, wobei der Basisklassenpointer dereferenziert werden muß, siehe Zeile 27 in Listing 12.19.

Listing 12.19: Ausgabeoperator eines Vektor mit Basisklassenpointern

```
1 //                                          |--- parameter passing by reference
      allows polymorphism
ostream& operator<<(ostream &s, const Fahrzeug& p);
3 //                                          |--- parameter passing by pointer
      allows polymorphism
//ostream& operator<<(ostream &s, const Fahrzeug* p);
5 //                                          |--- no polymorphism if parameter
      is passed by value
//ostream& operator<<(ostream &s, const Fahrzeug p);
7 /** \brief Prints the whole vector of base class pointers
  *  \param[in,out]  s    output stream
9 *  \param[in]      v    vector of base class pointers
  *  \return         changed output stream
11 */
ostream& operator<<(ostream &s, const vector<Fahrzeug*>& v);
13 ostream& operator<<(ostream &s, const Fahrzeug& p)
```

```
{
//                    Virtual Method Table
//              |--   VMT at run time ==> method from derived class
//              |                        |--- always non-virtual method from base class
    s << p.classname() << " : " << p.Get_kg() << " kg    and "
//              |--   VMT at run time ==> method from derived class
      << p.verbrauch() << " l/100km" << endl;
    return s;
}
ostream& operator<<(ostream &s, const vector<Fahrzeug*>& v)
{
    for (auto it: v)
    {
        cout << *it;
    };
    return s;
}
```

v_10c/main.cpp

12.3.2 Sortieren eines polymorphen Containers

Wir wollen obigen Vektor von Basisklassenpointern bzgl. des Kraftstoffverbrauch sortieren.

Listing 12.20: Inkorrektes Sortieren

```
    sort(v.begin(),v.end());
    cout << v << endl;
```

Der naive Aufruf des Algorithmus `sort` in Listing 12.20 vergleicht aber nur die Pointer (also die Adressen der Instanzen) statt deren Kraftstoffverbrauch, obwohl eine Methode `Fahrzeug:operator<` vorhanden ist. Korrekterweise müßte eine Funktion `bool operator< (const Fahrzeug* a, const Fahrzeug* b)` implementiert werden für eine korrekte Sortierung, was nicht allzu flexibel ist.

Bei Anwendung von Algorithmen auf Container mit Basisklassenpointern werden wir stets die Versionen des Algorithmus mit **expliziter Angabe der Vergleichsfunktion** benutzen. Dies wird im Listing 12.21 demonstriert, wobei die Vergleichsfunktion ihrerseits die nötigen Methoden von `Fahrzeug` benutzt.

Listing 12.21: Korrektes Sortieren mit Vergleichsfunktion (aufsteigend)

```
#include <vector>
#include <algorithm>
using namespace std;

bool fuel_consumption(const Fahrzeug* a, const Fahrzeug* b)
{
    return a->verbrauch() < b->verbrauch();
}
...
int main ()
{
  ....
  sort(v.begin(),v.end(), fuel_consumption );
  ....

}
```

Natürlich kann man die separate Angabe der Vergleichfunktion einsparen, wie dies Listing 12.22 mittels eine (absteigend sortierenden) Lamda-Funktion[2] demonstriert.

Listing 12.22: Korrektes Sortieren mit Lambda-funktion (absteigend)

```
#include <vector>
#include <algorithm>
using namespace std;

...
int main ()
{
  ....
    sort(v.begin(),v.end(),
```

[2] http://stackoverflow.com/questions/5122804/sorting-with-lambda

```
10          [](const Fahrzeug* const aa, const Fahrzeug* const cc) -> bool
              {
12                return aa->verbrauch() > cc->verbrauch();
              }
14        );
    ....
16
}
```

12.3.3 Summation eines polymorphen Containers

Wenn wir den Gesamtverbrauch unseres `Fahrzeug`-Containers berechnen wollen, so können wird dies konventionell via Summationsloop erledigen.

Listing 12.23: Gesamverbrauch - konventionell berechnet

```
1 {
...
3     float sum=0.0;
      for (unsigned int i=0; i<v.size(); ++i)
5     {
          sum += v[i]->verbrauch();
7     }
...
9 }
```

Kompakter erfolgt dies durch Nutzung der Methode `accumulate` unter Nutzung einer Vorschrift `add_fuel`, welche beschreibt, wie der Wert der aktuellen Instanz zu einem Zwischenergebnis addiert werden soll.

Listing 12.24: Gesamverbrauch - via `accumulate`

```
1 #include <numeric>                    //  accumulate
  using namespace std;
3 ....
  //! @brief Adds the fuel consumption of a vehicle @p y to quantity @p x
5 //!
  //! @param[in] x  given quantity
7 //! @param[in] y  base class pointer to vehicle
  //! @return x+y.fuel
9 //!
  //!
11 float add_fuel(float x, const Fahrzeug* y)
  {
13     return x + y->verbrauch();
  }
15 ....
  {
17 ...
       float sum2 =accumulate(v.begin(), v.end(), 0.0f, add_fuel);
19 ...
  }
```

Natürlich kann man statt der Funktion `fuel_consumption` auch gleich eine entsprechende Lambda-Funktion schreiben.

12.4 Casting in der Klassenhierarchie*

Castings (Konvertierungen) waren bislang stillschweigend in unseren Programmen enthalten, wie in

Listing 12.25: Implizites Casting bei einfachen Datentypen.

```
float salesPerson::payment() const
2 {
  double  dd = 15.373;
4 int     ii = 5, kk;

6  dd = ii;
   kk = dd;
8 }
```

Das erste Casting (`int` ⟶ `double`) ist problemlos, da die Nachkommastellen der Gleitkommazahl mit Nullen gefüllt werden. Im Gegensatz dazu gehen bei dem zweiten Casting (`double` ⟶ `int`) sämtliche Nachkommastellen verloren und bei ensprechenden Optionen (`-Wall`) gibt der Compiler eine Warnung aus. Beide Castings sind versteckt im Code enthalten, daher nennt man sie *implizit*. Mit einem *explizit*en Casting (Casting-operator) der zweiten Zuweisung

```
kk = (int)dd;   // C-Casting
```

oder

```
kk = static_cast<int>(dd);   // C++-Casting
```

läßt sich die Warnung (beim `g++`) abschalten.

12.4.1 Implizites Casting

Bei Klassen gibt es eine implizite Typkonvertierung nur in Richtung abgeleitete Klasse zu Basisklasse vor dem Hintergrund, daß eine abgeleitete Klasse ein *spezieller Typ* der Basisklasse ist (also gemeinhin mehr Member/Eigenschafter beinhaltet als diese). Alle Member der Basisklasse sind automatisch in der abgeleiteten Klasse enthalten, aber nicht umgekehrt. Das folgende Beispiel Die Abbildung der Klassenhierarchie auf Seite 104 bzw. in der Dokumentation[3] veranschaulicht diesen Mechanismus des *Upcasting* und *Downcasting* gemeinsam mit nachfolgendem Beispiel.

Listing 12.26: Casting bei Klassen.

```
1  ...
2  Worker aWage;
   salesPerson  aSale("Buddy Holly");
4
   aWage = aSale;        //   erlaubtes Upcasting
6  aWage.payment();      //    (aber Provision ist futsch)
8  salesPerson aSale2;
10 //  aSale2 = aWage;    //   nicht erlaubtes Downcasting
   ...
```

In Zeile 6 sind Name, Stundenlohn und Stundenanzahl (Members) des Angestellten `aWage` als entsprechende Einträge des Verkäufers `aSale` vorhanden. Die Verkaufsprovison (spezielles Member) wird dabei nicht übernommen da diese in der Klasse Worker nicht vorkommt. Umgekehrt müßte der Compiler eine unkommentierte Zeile 10 ablehnen, denn woher soll er (implizit!!) eine vernünftige Verkaufsprovision aus den vorhandenen Daten des Angestellten ableiten (die Klasse `Worker` besitzt weniger Member als `salesPerson`)? Also, Upcasting von Instanzen ist immer möglich (wir verlieren dabei aber Eigenschaften) während ein Downcasting die Definition/Zuweisung zusätzliche Eigenschaften erfordert (spezieller Kopierkonstruktur wird benötigt). Näheres zu Konvertierungskonstruktoren und -funktionen zwischen Klassen steht in [KPP02, §9 und §19], [Sch02, §9.4.4], [SK98, §16.5].

12.4.2 Casting von Klassenpointern und -referenzen

Beim **Casting von Instanzen** in §12.4.1 entscheidet der Compiler **beim Übersetzen** des Codes, ob ein Casting möglich ist oder nicht. Daher wird dieses Casting als statisches Casting bezeichnet und wird in der expliziten Variante mittels `aWage = static_cast<Worker>(aSale);` realisiert.

Wenn wir mit (Basis-)Klassen**pointer**n (desgleichen mit **Referenzen**) arbeiten, dann kann während des Übersetzens nicht entschieden werden, ob eine Casting zulässig ist oder nicht! Dies kann erst zur Laufzeit des Programmes, bei Prüfung der hinter dem Basisklassenpointer verborgenden konkreten Instanz, entschieden werden. Demenstprechend muß ein fehlgeschlagenes Casting auch **während der Laufzeit** abgefangen werden. Daher muß in diesem Falle immer ein dynamisches Casting via `dynamic_cast<A*>(bb)` (bzw. `dynamic_cast<A&>(bb)`) erfolgen.

[3] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Beispiele/v_9b/doxygen/html/class_employee.html`

12.4.3 Dynamisches C++-Casting von Pointern

Ist eine Umwandlung eines Pointers mit `dynamic_cast` nicht möglich, dann wird dem resultierenden Pointer der Nullpointer zugewiesen auf welchen danach getestet werden kann.

Listing 12.27: Korrektes dynamisches Casting von Pointern

```
//! @brief Shows correct downcast (C++-style) from base class pointer to pointer of
    derived class.
//!
//! @param[in] aSale Instance of the derived class
//! @warning   checks for correctness (null pointer?) and stops otherwise
//!
void pointer_correct_downcast_dynamic(salesPerson  aSale)
{
    cout << "  ***  korrektes Pointer Downcasting: C++-Style (Test zur Laufzeit)"
        << endl;
    Worker       *wagePtr (& aSale);     // Pointer auf Basisklasse
    salesPerson *salePtr;                // Pointer auf SalesPerson-Klasse
    salePtr = dynamic_cast<salesPerson *>(wagePtr);  // Korrekter Versuch des
        Downcasting des Pointers
    if ( salePtr==nullptr )                          // nullptr falls Downcasting
        nicht moeglich ist.
    {
        cout << "\nDowncasting ging schief. Panic Exit.\n" << endl;
        assert(salePtr!=nullptr);
    }
    cout << salePtr->payment() << endl;
    cout << *salePtr << endl;
    return;
}
```

v_9b_cast/main.cpp

Falls das Casting in Zeile 11 des Listings 12.27 fehlschlägt, dann wird der resultierende Pointer mit einem Nullpointer belegt. In Zeilen 12-16 wird auf diesen Nullpointer getestet und gegebenfalls das Programm beendet (oder ein Exception-Handling wird gestart). Da `wagePtr` auf eine Instanz vom Typ salesPerson zeigt wird ein korrektes Casting ausgeführt.

Listing 12.28: Inkorrektes dynamisches Casting von Pointern mit Fehlerabfrage

```
//! @brief Shows incorrect downcast (C++-style) from base class pointer to pointer
    of derived class.
//!
//! @param[in] simple Instance of base class
//! @warning   checks for correctness (null pointer?) and stops
//!
void pointer_wrong_downcast_dynamic(Employee  simple)
{
    cout << "  ***  falsches Pointer Downcasting: C++-Style (Test -> Fehler kann
        bemerkt werden)" << endl;
    Employee *empPtr (&simple); // --> einfacher Employee
    salesPerson *salePtr;
    salePtr = dynamic_cast<salesPerson *>(empPtr);  // Korrekter Versuch des
        Downcasting des Pointers
    if ( salePtr==nullptr )                          // nullptr falls Downcasting
        nicht moeglich ist.
    {
        cout << "\nDowncasting ging schief. Panic Exit.\n" << endl;
        assert(salePtr!=nullptr);
    }
    cout << endl << "         jetzt kracht es" << endl;
    salePtr->setComission(0.06);                 // nunmehr formal (ohne vorherige
        Zeile) machbar, aber katastrophal da diese Methode in Employee nicht
        existiert
    cout << salePtr->payment() << endl;
    cout << *salePtr << endl;
    return;
}
```

v_9b_cast/main.cpp

In Listing 12.28 zeigt `wagePtr` auf eine Instanz vom Typ `Employee`. Damit ist aber kein Casting auf die abgeleitete Klasse möglich und der resultierende Pointer wird mit einem Nullpointer belegt. Folgerichtig wird der Code in Zeile 15 gestoppt. Casting ausgeführt.

12.4.4 Dynamisches C++-Casting von Referenzen

Ist eine Umwandlung einer Referenz mit `dynamic_cast` nicht möglich, dann wird eine Exception (`std::bad_cast`) geworfen welche vom Code abgefangen werden kann (`throw-try-catch`).

Listing 12.29: Korrektes dynamisches Casting von Referenzen

```
1  //! @brief Shows correct downcast (C++-style) from base class reference to derived
       class reference.
   //!
3  //! @param[in] simple Instance of derived class
   //! @exception <std::bad_cast>  in case downcasting was incorrect.
5  //!
   void referenz_correct_downcast_dynamic(salesPerson  aSale)
7  {
       cout << "  ***  korrektes Referenz Downcasting: C++-Style (Test -> Fehler kann
           bemerkt werden)" << endl;
9      Worker& wageRef (aSale);
       try
11     {                          // Korrekter Versuch des Downcasting der Referenz
           salesPerson& saleRef = dynamic_cast<salesPerson&>(wageRef);
13         saleRef.setComission(0.06f);
           cout << saleRef.payment() << endl;
15         cout << saleRef << endl;
       }
17     catch (std::bad_cast& bc)
       {
19         std::cerr << "bad_cast caught: " << bc.what() << '\n';
           assert(false);         // and my panic exit
21     }
       return;
23 }
```

v_9b_cast/main.cpp

Das korrekte Casting erfolgt in Zeile 12 des Listings 12.29 und der `catch`-Teil in Zeilen 17-21 wird nicht ausgeführt.

Listing 12.30: Inkorrektes dynamisches Casting von Referenzen mit Exception

```
1  //! @brief Shows incorrect downcast (C++-style) from base class reference to
       derived class reference.
   //!
3  //! @param[in] simple Instance of base class
   //! @exception <std::bad_cast>  downcasting is incorrect.
5  //!
   void referenz_wrong_downcast_dynamic(Employee  simple)
7  {
       cout << "  ***  falsches Referenz Downcasting: C++-Style (Test -> Fehler kann
           bemerkt werden)" << endl;
9      Employee& empRef (simple);        // --> einfacher Employee
       try
11     {                          // Korrekter Versuch des Downcasting der Referenz
           salesPerson& saleRef = dynamic_cast<salesPerson&>(empRef);
13         saleRef.setComission(0.06f);
           cout << saleRef.payment() << endl;
15         cout << saleRef << endl;
       }
17     catch (std::bad_cast& bc)
       {
19         std::cerr << "bad_cast caught: " << bc.what() << '\n';
           assert(false);         // and my panic exit
21     }
       return;
23 }
```

v_9b_cast/main.cpp

In Listing 12.29 wirft das inkorrekte Casting in Zeile 12 die Exception (Ausnahmebehandlung) `std::bad_cast`, welche durch das `catch` in Zeile 17 abgefangen wird, wodurch statt der Zeilen 13-15 die Fehlerbehandlung in Zeilen 19-20 ausgeführt wird.

12.4.5 Unsicheres statisches C-Casting von Klassenpointern

Das C-Casting von Pointern oder Referenzen (oder Instanzen) einer Klassenstruktur ist eines der **schlimmsten Übel** wenn Leute behaupten, daß sie C++ programmieren würden. Dafür gibt es `dynamic_cast` (und `static_cast`)!

Als Illustration dienen die beiden folgenden Codes.

Listing 12.31: Statisches C-Casting korrekt benutzt

```
1  //! @brief Shows correct downcast (C-style) from base class pointer to pointer of
       derived class.
2  //!
3  //! @param[in] aSale Instance of the derived class
4  //! @warning   no check for correctness in C-casting
5  //!
6  void pointer_correct_downcast_C(salesPerson  aSale)
7  {
8      cout << "  ***  korrektes Pointer Downcasting: C-Style (DANGER !! kein Test)"
           << endl;
9      Worker      *wagePtr (& aSale);    // Pointer auf Basisklasse
10     // Mit Pointern geht jeder Quatsch
11     salesPerson *salePtr;                  // Pointer auf SalesPerson-Klasse
12     //salePtr = wagePrt;                        // geht nicht, da kein
           implizites Downcasting moeglich ist
13     salePtr = (salesPerson *) wagePtr;  // expizite Typkonvertierung, gefaehrlich !!
14     cout << salePtr->payment() << endl; // hier klappt diese aber, da hinter
           wagePtr ein Object der Klasse salesPerson steckt
15     cout << *salePtr << endl;
16     return;
17 }
```

v_9b_cast/main.cpp

In Listing 12.31 ist das statische C-Casting in Zeile 13 korrekt, da `wagePtr` auf eine Instanz vom Typ salesPerson zeigt. Deshalb werden die Zeilen 14-15 auch die erwarteten Ausgaben liefern.

Listing 12.32: Statisches C-Casting führt zu nicht vorhersagbarem Verhalten

```
1  //! @brief Shows incorrect downcast (C-style) from base class pointer to pointer of
       derived class.
2  //!
3  //! @param[in] simple Instance of base class
4  //! @warning   No check for correctness in C-casting. Sometimes the code crashes
       (and that's good).
5  //!
6  void pointer_wrong_downcast_C(Employee  simple)
7  {
8      cout << "  ***  falsches Pointer Downcasting: C-Style (kein Test -> Code
           crashes or not)" << endl;
9      Employee *empPtr (&simple); // --> einfacher Employee
10     salesPerson *sale_Ptr;
11     sale_Ptr = (salesPerson *)empPtr;   // syntaktisch erlaubt, aber falsch da
           empPtr wirklich nur auf Basisklasse zeigt
12     cout << endl << "          jetzt kracht es" << endl;
13     sale_Ptr->setComission(0.06);              // nunmehr formal (ohne vorherige
           Zeile) machbar, aber katastrophal da diese Methode in Employee nicht
           existiert
14     cout << sale_Ptr->payment() << endl;
15     cout << *sale_Ptr << endl;
16     return;
17 }
```

v_9b_cast/main.cpp

In Listing 12.31 ist das statische C-Casting in Zeile 13 falsch, da `wagePtr` auf eine Instanz vom Typ `Employee` zeigt. Es gibt allerdings hier *keine Möglichkeit* die Korrektheit des Castings zu überprüfen. Als Konsequenz liefern die nachfolgenden Zeilen 14-15 falsche oder unsinnige Ausgaben (es gibt keine Methode `Employee::setComission`) oder das Programm bricht undefiniert und sehr schwer nachvollziehbar ab.

Merksatz: Wer gerne und lange Debugged nimmt das C-Casting, alle anderen nehmen `dynamic_cast` oder `static_cast`, und schreiben verläßliche Programme wesentlich schneller.

12.4.6 Einige Bemerkungen zum Casting

An Stelle des C-Casts `(Typ) Ausdruck` wie in `(double) idx` sollte man die vier in C++ enthaltenen Casts benutzen, welche ein spezifischeres Casting erlauben, leichter im Programmcode auffindbar sind und es dem Compiler auch erlauben, bestimmte Casts abzulehnen.

- `static_cast<Typ>(Ausdruck)` ersetzt das bekannte C-Cast, also würde `(double) idx` korrekterweise zu `static_cast<double>(idx)` .
- `const_cast<Typ>(Ausdruck)` erlaubt die Beseitigung der Konstantheit eines Objektes. Eine Anwendung dafür ist der Aufruf einer Funktion, welche nichtkonstante Objekte in der Parameterliste erwartet, diese aber nicht verändert (das Interface dieser Funktion ist schlecht designed).

```
void Print(Studenten&);
...
int main()
{
 const Studenten arni("Arni","Schwarz",89989, 787);
 Print(const_cast<Studenten>(arni));
}
```

Hat man Zugriff auf die Quellen der `Print`-Funktion, dann sollte man besser das Interface korrekt gestalten statt obiges Casting zu benutzen. Ist dies nicht möglich (wie bei Benutzung mancher alter C-Funktionen oder manchen MPI[4]-Implementierungen) dann wird das Casting unvermeidlich.

- `dynamic_cast<Typ>(Ausdruck)` dient der sicheren Umformung von Pointern und Referenzen in einer Vererbungshierarchie und zwar nach unten (abgeleitete Klasse) oder zwischen benachbarten Typen. So wäre das exakte Casting in Zeile 13 des Listings 12.31 auf Seite 118: `salePtr = dynamic_cast<salesPerson*>(wagePtr);` . Falls das Casting (zur Laufzeit!!) nicht erfolgreich ist wird ein Nullzeiger zurückgegeben bzw. bei Referenzen eine Exception geworfen.
- `reinterpret_cast<Typ>(Ausdruck)` wird für Umwandlungen benutzt deren Ergebnis fast immer implementationsabhängig ist. Meist wird es zur Umwandlung von Funktionspointern benutzt.

Obige Erläuterungen und weitere Beispiele zu diesen Casts sind in [Mey97, §1.2] zu finden. Mehr Beispiele und Bemerkungen zur Typüberprüfung der Casts, siehe [Sch02, p.246f].

[4] `https://www.open-mpi.org/`

Kapitel 13

Tips und Tricks

13.1 Präprozessorbefehle

Wir kennen bereits die Präprozessoranweisung

```
#include <cmath>
```

welche vor dem eigentlichen Compilieren den Inhalt des Files *cmath* an der entsprechenden Stelle im Quellfile einfügt. Analog können bestimmte Teile des Quelltextes beim Compilieren eingebunden oder ignoriert werden, je nach Abhängigkeit des Tests welcher mit einer Präprozessorvariablen durchgeführt wird.

preproc.cpp

Variablen des Präprozessors werden mittels

```
#define MY_DEBUG
```

definiert und wir können auch testen, ob sie definiert sind:

```
#ifdef MY_DEBUG
  cout << "Im Debug-Modus" << endl;
#endif
```

Analog kann mit

```
#ifndef MY_DEBUG
#define MY_DEBUG
#endif
```

zunächst getestet werden, ob die Variable `MY_DEBUG` bereits definiert wurde. Falls nicht, dann wird sie eben jetzt definiert. Diese Technik wird häufig benutzt um zu verhindern, daß die Deklarationen eines Headerfiles mehrfach in denselben Quelltext eingebunden werden.

studenten4.h

```
//      studenten4.hpp
#ifndef FILE_STUDENTEN
#define FILE_STUDENTEN
//      Deklarationen des Headefiles
...
#endif
```

Einer Präprozessorvariablen kann auch ein Wert zugewiesen werden

```
#define SEE_PI 5
```

welcher anschließend in Präprozessortests (oder im Programm als Konstante) benutzt werden kann:

```
#if (SEE_PI==5)
  cout << " PI = " << M_PI << endl;
#else
// leer oder Anweisungen
#endif
```

Eine häufige Anwendung besteht in der Zuweisung eines Wertes zu einer Präprozessorvariablen, falls diese noch nicht definiert wurde.

```
#ifndef M_PI
#define M_PI 3.14159
#endif
```

Desweiteren können Makros mit Parametern definiert

`#define MAX(x,y) (x>y ? x : y)`

und im Quelltext verwendet werden.

```
cout << MAX(1.456 , a) << endl;
```

Mehr über Präprozessorbefehle ist u.a. in [God98] und [Str00, $A.11] zu finden.

13.2 Zeitmessung im Programm

Zum Umfang von C++ gehören einige Funktionen, welche es erlauben die Laufzeit bestimmter Programmabschnitte (oder des gesamten Codes) zu ermitteln. Die entsprechenden Deklarationen werden im Headerfile *ctime* bereitgestellt. Die Zeitmessung mit `clock()` ist nur für sequentielle Programme hilfreich, wenn Ihr Code mehrere Cores nutzt, dann ist besser , s.u., zu verwenden.

Listing 13.1: (veraltete) Laufzeitmessung im Code mit *ctime*

```
#include <iostream>
using namespace std;
int main()
{
    double time1 = 0.0, time2 = 0.0, tstart;     // time measurment
    tstart = clock();
//      ...
    time1 += clock() - tstart;
//       Rescale to get seconds
    time1 = time1 / CLOCKS_PER_SEC;
    return 0;
}
```

Ex1121.cpp

Es können beliebig viele Zeitmessungen im Programm erfolgen (irgendwann verlangsamen diese aber ihrerseits das Programm!). Jede dieser Zeitmessungen benötigt einen Start und ein Ende, allerdings können die Zeiten verschiedener Messungen akkumuliert werden (einfach addieren).

Ex1121.cpp

Im File *Ex1121.cpp* wird der Funktionswert eines Polynoms vom Grade 20 an der Stelle x d.h., $s = \sum_{k=0}^{20} a_k \cdot x^k$, berechnet. Die 21 Koeffizienten a_k und der Wert x werden im File *input.1121* bereitgestellt. Der Funktionswert wird auf zwei, mathematisch identische, Weisen im Programm berechnet. Variante 1 benutzt die Funktion `pow`, während Variante 2 den Wert von x^k durch fortwährende Multiplikation berechnet (Implementierungsvergleich 2017[1] und 2021[2]).

Das unterschiedliche Laufzeitverhalten (Ursache !?) kann nunmehr durch Zeitmessung belegt werden und durch fortschreitende Aktivierung von Compileroptionen (`-ffast-math`[3]) zur Programmoptimierung verbessert werden, z.B.

```
LINUX>  g++ Ex1121.cpp
LINUX>  g++ -O Ex1121.cpp
LINUX>  g++ -O3 Ex1121.cpp
LINUX>  g++ -O3 -ffast-math Ex1121.cpp
```

Der Programmstart erfolgt jeweils mittels

```
LINUX>  a.out < input.1121
```

Eine bessere Zeitmessung ist über den C++-Header *chrono* möglich, da `system_clock` auch bei parallelen Prozessen die Wall Clock Time (die für den User vergangene Zeit) anzeigt

Listing 13.2: Laufzeitmessung mit *chrono*

```
#include <vector>
typedef std::chrono::system_clock Time;
typedef std::chrono::milliseconds ms;
typedef std::chrono::duration<double> dsec;
int main()
```

[1] https://baptiste-wicht.com/posts/2017/09/cpp11-performance-tip-when-to-use-std-pow.html
[2] https://stackoverflow.com/questions/2940367/what-is-more-efficient-using-pow-to-square-or-just-multiply-it-with-
[3] https://kristerw.github.io/2021/10/19/fast-math/

```
{
    auto tstart = Time::now();
//      ...
    dsec time1 = Time::now() - tstart;
    cout << "  slow (   pow) = " << s1 << "   (" << time1.count() << " sec.)\n";
    return 0;
}
```

Ex1121_mod.cpp

13.3 Profiling

Natürlich könnte man in einem Programm die Zeitmessung in jede Funktion schreiben um das Laufzeitverhalten der Funktionen und Methoden zu ermitteln. Dies ist aber nicht nötig, da viele Entwicklungsumgebungen bereits Werkzeuge zur Leistungsanalyse (performance analysis), dem **Profiling** bereitstellen. Darin wird mindestens die in den Funktionen verbrachte Zeit und die Anzahl der Funktionsaufrufe (oft graphisch) ausgegeben. Manchmal läßt sich dies bis auf einzelne Quelltextzeilen auflösen. Neben den professionellen (und kostenpflichtigen) Profiler- und Debuggingwerkzeugen sind unter LINUX/UNIX auch einfache (und kostenlose) Kommandos dafür verfügbar.

```
LINUX>  g++ -pg Jacobi.cpp matvec.cpp
LINUX>  a.out
LINUX>  gprof -b a.out > out
LINUX>  less out
```

Der Compilerschalter `-pg` bringt einige Zusatzfunktionen im Programm unter, sodaß nach dem Programmlauf das Laufzeitverhalten durch `gprof` analysiert werden kann. Der letzte Befehl (kann auch ein Editor sein) zeigt die umgeleitetete Ausgabe dieser Analyse auf dem Bildschirm an.

13.4 Debugging

Oft ist es notwendig den Programmablauf schrittweise zu verfolgen und sich gegebenenfalls Variablenwerte etc. zu Kontrollzwecken ausgeben zu lassen. Neben der stets funktionierenden, jedoch nervtötenden, Methode

```
...
cout << "AA " << variable << endl;
...
cout << "BB " << variable << endl;
...
```

sind oft professionelle Debuggingwerkzeuge verfügbar. Hier sei wiederum ein (kostenfreies) Programm unter LINUX vorgestellt.

```
LINUX>  g++ -g Ex1121.cpp
LINUX>  ddd a.out &
```

Die Handhabung der verschiedenen Debugger unterscheidet sich sehr stark. Beim ddd-Debugger kann mit `set args < input.1121` das Eingabefile angegeben werden und mit `run` wird der Testlauf gestartet, welcher an vorher gesetzten Break-Punkten angehalten wird. Dort kann dann in aller Ruhe das Programm anhand des Quellcodes schrittweise verfolgt werden.

13.5 Einige Compileroptionen

Nachfolgende Compileroptionen beziehen sich auf den GNU-Compiler g++, jedoch sind die Optionen bei anderen Compilern teilweise identisch oder ähnlich.

- Debugging und Warnungen, welche viel Debuggingarbeit ersparen:
  `g++ -g -Wall -Wextra -pedantic -Wswitch-default -Wmissing-declarations -Wfloat-equal -Wundef -Wredundant-decls -Wuninitialized -Winit-self -Wshadow -Wparentheses -Wunreachable-code`

- Zusätzliche Optionen für Klassenhierarchien nach [Mey98, Mey97]: `-Weffc++ -Woverloaded-virtual`
- Standardoptimierung: `g++ -O`
- Bessere Optimierung: `-Ofast`
- Den `assert()`-Test ausschalten: `-DNDEBUG`

13.6 Numerik in C++

Die numerische Bibliothek[4] von C++ wird kontinuierliche erweitert. Für uns von Interesse sind

- die üblichen mathematischen Funktionen[5] inkl. der Γ-Funktion;
- spezielle[6] math. Funktionen, wie der Bessel-Funktion;
- die gelisteten *Numeric Operations* der STL, wie `accumulate` oder `inner_product`;
- Bitmanipulationen;
- die Möglichkeit, über die Floating-point[7] environment auch einige Execptions bei Berechnungen abzufragen (Overflow, DivisionByZero[8], etc.);
- ein für Numerik optimierter Vektor `valarray`[9].

Daneben gibt es die *boost-Library*[10] mit einer überwältigenden Funktionalität[11]. Die Boost ist auch ein Experimentierfeld für Datenstrukuren und Algorithmen welche bei Bedarf in den C++-Standard übernommen werden.
So sind in *graph*[12] Datenstrukturen und viele Graphalgorithmen enthalten, neben weiteren mathemtischen Bibliotheken.

Für klassische lineare Algebra sei auf die Bibliotheken BLAS[13] und LAPACK[14] mit den entsprechenden C/C++-Interfaces verwiesen.

[4] `https://en.cppreference.com/w/cpp/numeric`
[5] `https://en.cppreference.com/w/cpp/numeric/math`
[6] `https://en.cppreference.com/w/cpp/numeric/special_functions`
[7] `https://en.cppreference.com/w/cpp/numeric/fenv`
[8] `https://en.cppreference.com/w/cpp/numeric/fenv/FE_exceptions`
[9] `https://en.cppreference.com/w/cpp/numeric/valarray`
[10] `https://www.boost.org/`
[11] `https://www.boost.org/doc/libs/`
[12] `https://www.boost.org/doc/libs/1_76_0/libs/graph/doc/index.html`
[13] `http://www.netlib.org/blas/`
[14] `https://www.netlib.org/lapack/`

13.7 Zufallszahlen

Zur Zufallszahlengenerierung sind neben den alten C-Funktionen in `<cstdlib>` in C++ viele Zufallszahlengeneratoren in `<random>`[15] verfügbar.

Examplarisch demonstrieren wir die Generierung von LOOPS gleichverteilter ganzzahliger (Pseudo)-Zufallszahlen aus einem Intervall [ANF, ENDE] auf drei verschiedene Arten unter Nutzung von `rand()`[16], `minstd_rand0`[17], `uniform_int_distribution`[18].

random/main.cpp

Listing 13.3: Zufallszahlen C-like

```
#include <cstdlib>     // rand()
#include <ctime>       // time()
    {   // ganzzahlige Zufallszahlen in C
        // Seed
        srand ( time(nullptr) );
        // Generierung
        for (int k=0; k<LOOPS; ++k)
        {
            int is = rand() % (ENDE-ANF+1)+ANF;   // Zufallszahl aus [ANF, ENDE]
            cout << "  " << is;
        }
        cout << endl;
    }
```

random/main.cpp

Listing 13.4: Zufallszahlen C++

```
#include <ctime>       // time()
#include <random>      // C++11 random classes: minstd_rand0::operator(),
    std::mt19937, std::uniform_int_distribution
    {   // ganzzahlige Zufallszahlen in C++, analog zu C
        // Seed
        std::minstd_rand0 gen(time(nullptr));    //Will be used to obtain a seed for
            the random number engine
        // Generierung
        for (int k=0; k<LOOPS; ++k)
        {
            int is = gen() % (ENDE-ANF+1)+ANF;   // Zufallszahl aus [ANF, ENDE]
            cout << "  " << is;
        }
        cout << endl;
    }
```

random/main.cpp

Listing 13.5: komfortable Zufallszahlen in C++

```
#include <random>      // C++11 random classes: minstd_rand0::operator(),
    std::mt19937, std::uniform_int_distribution
    {   // ganzzahlige Zufallszahlen in C++: komfortabler
        // Seed
        std::random_device rd;  //Will be used to obtain a seed for the random
            number engine
        // Generierung
        std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
        std::uniform_int_distribution<> distrib(ANF,ENDE);
        for (int k=0; k<LOOPS; ++k)
        {
            int is = distrib(gen);  // ganze Zufallszahl aus [ANF, ENDE]
            cout << "  " << is;
        }
        cout << endl;
    }
```

random/main.cpp

[15] https://en.cppreference.com/w/cpp/header/random
[16] https://en.cppreference.com/w/cpp/numeric/random/rand
[17] http://www.cplusplus.com/reference/random/linear_congruential_engine/operator()
[18] https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

Kapitel 14

Nützliche Webseiten

Die folgenden Webseiten fand ich sehr nützlich:

- Zahldarstellung und Wertebereich bei Integer[1]-Zahlen.
- Zahldarstellung und Wertebereich bei Gleitkommazahlen[2].
- Ein sehr anschauliches, deutschsprachiges Lehrbuch[3] von Arnold Willemer mit seinem Crash-Kurse:
  Von Null auf C++ auf unter 40 Seiten[4]
- Ebenfalls eine schöne, umfassende und trotzdem relativ kurze Einführung in C++[5] von Wolfgang Schröder.
- Sehr gutes Lehrbuch zur STL[6] und Übersicht der STL-Container und Algorithmen[7] von Ulrich Breymann (nichtgewerbliche Nutzung ist kostenlos), ebenso eine gute Erläuterung der STL in English[8]

Zum Nachschlagen von Funktionen/Methoden, sowie zum Beantworten von wiederkehrenden Fragen:

- Beschreibung der C++-Bibliotheken[9] in englisch.
- Weiteres Tutorial[10], welches auch C und C++11 enthält.
- Zum Nachschlagen in Deutsch[11] und Englisch[12]
- Frequently Asked Questions (FAQs)[13] in englisch.

Zur generellen (Weiter-)Entwicklung von C++

- Der neue Standard C++11 räumt mit einigen Mankos von C++ auf, siehe [Gri11, Wil12]. Diskussionsforen und Online-Tutorials dazu sind unter Solarian Programmer[14] verfügbar, auch der Wikipediaeintrag[15] ist sehr hilfreich.

[1] http://de.wikipedia.org/wiki/Integer_(Datentyp)
[2] http://de.wikipedia.org/wiki/IEEE_754
[3] http://willemer.de/informatik/cpp
[4] http://willemer.de/informatik/cpp/crash.htm
[5] http://www.cpp-tutor.de/cpp/intro/toc.htm
[6] http://www.ubreymann.de/publ.html
[7] http://www.ubreymann.de/publ.html
[8] http://www.sgi.com/tech/stl
[9] http://www.cplusplus.com/reference/
[10] http://www.cprogramming.com/tutorial.html
[11] http://www.cppreference.com/wiki/de/start
[12] http://en.cppreference.com/w/cpp
[13] http://parashift.com/c++-faq-lite/
[14] http://solarianprogrammer.com/
[15] http://en.wikipedia.org/wiki/C++11

- D - eine neue Programmiersprache mit C++–Wurzeln [iX 6/2007, p.68–72] (naja, mich überzeugt D nicht).

Ranking von Programmiersprachen:

- Monatliche Übersicht von Tiobe Software[16].

[16] `http://www.tiobe.com/tpci.htm`

Literaturverzeichnis

[Bre17] Ulrich Breymann. *Der C++ Programmierer - Aktuell zu C++17*. Hanser Fachbuchverlag, 5th edition, 2017. `http://www.cppbuch.de/`.

[Cor93] Microsoft Corp. *Richtig einsteigen in C++*. Microsoft Press, 1993.

[Fib07] Fibelkorn. *Die schwarze Kunst der Programmierung*. Semele Verlag, 2007.

[Fil18] Bartłomiej Filipek. *C++17 in detail*. bfilipek.com, 2018.

[Gal17] Jacek Galowicz. *C++17 STL Cookbook*. Packt Publishing, 2017.

[God98] Eduard Gode. *ANSI C++: kurz & gut*. O'Reilly, 1998.

[Got19] Peter Gottschling. *Forschung mit modernem C++*. Hanser Fachbuch, 2019.

[Gri11] Rainer Grimm. *C++11: Der Leitfaden für Programmierer zum neuen Standard*. Addison-Wesley, München, 2011.

[Gri21a] Rainer Grimm. *The C++ Standard Library*. leanpub.com, 2021.

[Gri21b] Rainer Grimm. *C++20: Get the Details*. leanpub.com, 2021.

[HT03] Andrew Hunt and David Thomas. *Der Pragmatische Programmierer*. Hanser Fachbuch, 2003.

[KPP02] Ulla Kirch-Prinz and Peter Prinz. *OOP mit C++*. Galileo Press, limitierte studentenausgabe edition, 2002.

[KS02] Stefan Kuhlins and Martin Schader. *Die C++ Standardbibliothek*. Springer, Berlin, Heidelberg, 3. edition, 2002.

[Lou06] Dirk Louis. *C++: Programmieren mit einfachen Beispielen. Leicht - Klar - Sofort*. Markt und Technik, 2006.

[Mey97] Scott Meyers. *Mehr effektiv C++ programmieren*. Addison-Wesley, 1997.

[Mey98] Scott Meyers. *Effektiv C++ programmieren*. Addison-Wesley, 3., aktualisierte edition, 1998.

[Mey15] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. Addison-Wesley Professional Computing Series. O'Reilly, 2015.

[OT93] Andrew Oram and Steve Talbott. *Managing Projects with make*. O'Reilly, 1993.

[PD08] Gustav Pomberger and Heinz Dobler. *Algorithmen und Datenstrukturen*. Pearson Studies, 2008.

[SB95] Gregory Satir and Doug Brown. *C++: The Core Language*. O'Reilly, 1995.

[SB16] Andreas Spillner and Ulrich Breymann. *Lean Testing für C++-Programmierer*. dpunkt.verlag, 2016.

[Sch02] Klaus Schmaranz. *Softwareentwicklung in C++*. Springer, Heidelberg, 2002.

[SK98] Martin Schader and Stefan Kuhlins. *Programmieren in C++.* Springer, Heidelberg, 5. neubearbeitete edition, 1998.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache.* Addison-Wesley, 4. aktualisierte edition, 2000.

[Str03] Thomas Strasser. *Programmieren mit Stil. Eine systematische Einführung.* dpunkt, 2003.

[Str10] Bjarne Stroustrup. *Einführung in die Programmierung mit C++.* Pearson Studium, München, 2010.

[Wil12] Torsten Will. *C++11 programmieren: 60 Techniken für guten C++11-Code.* Galileo Computing, 2012. mit CD-ROM.

[Wil18] Torsten Will. *C++11 Das umfassende Handbuch - Aktuell zu C++17.* Rheinwerk Computing, 1st edition, 2018. Bsp. zum Download; 41,10 EUR.

[Wil20a] Torsten Will. *The C++ Standard Library.* Leanpub, 3rd edition, 2020.

[Wil20b] Torsten Will. *C++20 Get the Details.* Leanpub, 2020.

[Wil20c] Torsten Will. *Concurrency with Modern C++.* Leanpub, 2020.

[Wol06] Jürgen Wolf. *C++ von A bis Z.* Galileo Press, 2006. mit CD-ROM.

[Yan01] Daoqi Yang. *C++ and object-oriented numeric computing for Scientists and Engineers.* Springer, New York, 2001.

Listings

Index