Vorlesung

Objektorientiertes Programmieren in C++

Teil 6 - WS 2023/24

Detlef Wilkening
www.wilkening-online.de
© 2023

13 Weiteres aus der Standard-Bibliothek 2
13.1 File-Streams 2
13.2 Filesystem-Library 10
13.3 Exit 21
13.4 Datum, Zeit & Zeitmessungen 22
13.5 Zufallszahlen 24
13.6 Mathematische Funktionen 26
13.7 Swap-Funktion 28
13.8 Pairs und Tuple 29

13 Weiteres aus der Standard-Bibliothek

In diesem Kapitel möchte ich Ihnen einige weitere Funktionen und Klassen aus der Standard-Bibliothek vorstellen, die wir in den weiteren Kapiteln und Aufgaben benötigen um sinnvolle Beispiele und Programme schreiben zu können – z.B. der Zugriff auf Dateien oder die Generierung von Zufallszahlen.

Die einfache Nutzung dieser Funktionalitäten sollte nach diesem Kapitel kein Problem für Sie sein – im Detail verstehen werden Sie hier aber noch nicht alles. Viele der Funktionalitäten bauen z.B. auf Klassen auf, die wir erst in späteren Kapiteln kennen lernen werden.

Die hier vorgestellten Funktionalitäten sind nur ein Bruchteil der in der Standard-Bibliothek enthaltenen Features. Sie sollen keinesfalls glauben, dass Sie nach diesem Kapitel einen Überblick über die Funktionen der Standard-Bibliothek haben.

13.1 File-Streams

Analog zu den Streams für Console/Tastatur und den String-Streams gibt es auch File-Streams für Dateien.

- Headerdatei: <fstream>
- File-Ausgabe-Klasse: std::ofstream
- File-Eingabe-Klasse: std::ifstream
- Konstruktion mit dem Datei-Namen (optional mit absolutem oder relativem Pfad).

```
#include <iostream>
#include <fstream>

int main()
{
   // Der Scope um die folgenden zwei Zeilen sorgt fuer
   // das automatische Schliessen des Ausgabestroms - s.u.
   {
      std::ofstream out("temp.txt");    // Ausgabestrom oeffnen und
      out << 3.14 << 'A' << 14;         // Werte rausschieben
   }                                    // Eigentlich mit Fehlerbehandlung - s.u.

   int i;
   char c;
   double d;
   std::ifstream in("temp.txt");        // Eingabestrom oeffnen und
   in >> d >> c >> i;                   // Werte einlesen
```

```
                                           // Eigentlich mit Fehlerbehandlung - s.u.

   std::cout <<   "double: " << d
             << "\nchar:   " << c
             << "\nint:    " << i
             << '\n';
}
```

```
Ausgabe
double: 3.14
char:   A
int:    14
```

Wird ein File-Stream Objekt erzeugt, so passiert folgendes:

- ofstream – die Datei wird, wenn nicht vorhanden, angelegt und zum Schreiben geöffnet. Ist die Datei vorhanden, so wird ihr Inhalt gelöscht, und sie am Anfang zum Schreiben geöffnet.
- ifstream – ist die Datei vorhanden, so wird sie am Anfang zum Lesen geöffnet. Ist sie nicht vorhanden, so geht der Stream in den Status *FAIL*.

File-Streams verhalten sich bei analoger Nutzung analog zu normalen Streams, d.h. textbasiert. Daher werden Zahlen auch bei der Ausgabe in File-Streams als Text geschrieben. Überprüfen Sie dies bitte, in dem Sie die erzeugte Datei einmal mit einem normalen Text-Editor öffnen.

Hinweis – im Gegensatz zum z.B. C-File-Handling müssen in C++ File-Stream-Objekte „*im Normalfall*" nicht explizit geschlossen werden – damit kann dies auch nicht vergessen werden. Aber im Beispiel soll aus der gleichen Datei gelesen werden, in die direkt vorher geschrieben wurde. Das Lesen kann nur funktionieren, wenn der Output-File-Stream seinen Ausgabepuffer geleert und die Datei geschlossen hat. Darum liegt um das Schreiben ein Block, der dafür sorgt, dass beim Verlassen der Destruktor des Output-File-Stream-Objekts aufgerufen und damit die Datei implizit geschlossen wird.

Achtung – das Schreiben in und das Lesen aus Dateien ist natürlich im höchsten Maße kritisch, d.h. kann immer schief gehen. Hier sollten Sie niemals auf eine adäquate Fehlerbehandlung verzichten.

13.1.1 Datei-Ende und Fehler – Status *EOF* und *FAIL*

Im Normallfall ist das Lesen aus einer Datei das Lesen einer unbekannten Menge von z.B. Zeichen, Zahlen oder Zeilen. Das Problem ist: zu erkennen, wann das Datei-Ende erreicht wurde, und das Lesen beendet werden muss.

Aber schon früher haben wir gelernt: „Genau genommen bedeutet der Status *FAIL*, dass **nichts** gelesen werden konnte – aus welchen Gründen auch immer." Umgekehrt heißt das, wenn der Stream **nicht** im Status *FAIL* ist, dass das Einlesen geklappt hat, und die Eingabe ausgewertet werden kann.

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
   {
      ofstream out("temp.txt");
      out << "abcd";
   }

   ifstream in("temp.txt");
   for (;;)
   {
      char c;
      in >> c;
      if (in.fail()) break;
      cout << "-> " << c << '\n';
   }
}
```

```
Ausgabe
-> a
-> b
-> c
-> d
```

Mit dem Verlassen der Schleife stellt sich nur die Frage: Wieso wurde die Schleife abgebrochen? Oder anders formuliert: Was führte zum Status *FAIL*? War das Datei-Ende erreicht und alles hatte seine Ordnung, oder ist ein Problem aufgetreten (z.B. jemand hat die Diskette während des Lesens entfernt)?

Um auf das Datei-Ende abzufragen, hat ein „std::ifstream" die Element-Funktion „eof()" – für „end-of-file". Die Element-Funktion „eof()" gibt „true" zurück, wenn das Datei-Ende gelesen wurde, d.h. der Stream in den Status *EOF* gewechselt ist.

Wurde die Schleife also wegen des Datei-Endes verlassen, dann **muss** der Stream **auch** im Status *EOF* sein. Ansonsten hat es ein Problem gegeben.

13.1.1.1 Beispiel – Zeichen einlesen

Mit einer minimalen Fehler-Behandlung (eher einer Fehler-Meldung) verändert sich das Beispiel von oben zu folgendem:

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

int main()
{
   {
      ofstream out("temp.txt");
      out << "abcd";
   }

   ifstream in("temp.txt");
   for (;;)
   {
      char c;
      in >> c;
      if (in.fail()) break;
      cout << "-> " << c << '\n';
   }
   cout << (in.eof() ? "EOF" : "Fehler") << '\n';                // << neu
}
```

```
Ausgabe
-> a
-> b
-> c
-> d
EOF
```

13.1.1.2 Beispiel – Integer einlesen

Und genauso kann man z.B. eine unbestimmte Menge von Integer aus einer Datei einlesen:

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

int main()
{
   {
      ofstream out("temp.txt");
      out << "23 45 98";
   }

   ifstream in("temp.txt");
   for (;;)
   {
      int i;
      in >> i;
      if (in.fail()) break;
      cout << "-> " << i << '\n';
   }
   cout << (in.eof() ? "EOF" : "Fehler") << '\n';
}
```

```
Ausgabe
-> 23
-> 45
-> 98
EOF
```

13.1.1.3 Beispiel – Zeilenweises Einlesen

Oder auch das zeilenweise Einlesen mit „std::getline“ aus einer Datei ist überhaupt kein Problem:

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

int main()
{
   {
      ofstream out("temp.txt");
      out << "Zeile 1\nZeile 2\nZeile 3";
   }

   ifstream in("temp.txt");
   for (string s;;)
   {
      getline(in, s);
      if (in.fail()) break;
      cout << "-> " << s << '\n';
   }
   cout << (in.eof() ? "EOF" : "Fehler") << '\n';
}
```

```
Ausgabe
-> Zeile 1
-> Zeile 2
-> Zeile 3
```

```
EOF
```

13.1.1.4 Beispiel – Fehler

Wer mal einen Fehler sehen möchte, ohne gleich beim Lesen in das Datei-System einzugreifen, kann dies problemlos beim Einlesen der Integer provozieren. Man muss nur ein Nicht-Integer-Zeichen mit in die Datei schreiben:

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

int main()
{
   {
      ofstream out("temp.txt");
      out << "23 45 xx 98";          // << hier falsche Zeichen fuer Integer
   }

   ifstream in("temp.txt");
   for (;;)
   {
      int i;
      in >> i;
      if (in.fail()) break;
      cout << "-> " << i << '\n';
   }
   cout << (in.eof() ? "EOF" : "Fehler") << '\n';
}
```

```
Ausgabe
-> 23
-> 45
Fehler
```

13.1.1.5 Fehl-Benutzung *„EOF* statt *FAIL“*

Ein typischer Fehler ist die Abfrage auf *EOF* statt auf *FAIL*. Da man sich ja für das Ende der Datei interessiert, klingt *EOF* doch nach dem besseren Status. Aber dies ist falsch. *EOF* sagt nur, dass beim Lesen das Datei-Ende erreicht wurde. Damit wird **nicht** ausgesagt, dass nichts mehr eingelesen wurde.

Beispiel – in einer Datei steht „12 34“, und es werden in einer Schleife Integer aus der Datei gelesen.

1. Die Datei wird geöffnet.
   - Der interne Postions-Zeiger der Datei steht vor dem ersten Zeichen.
   - Status: *NOT-FAIL* und *NOT-EOF.*
2. Der erste Integer wird gelesen.
   - Dazu werden die Zeichen „1“ und „2“ als Integer interpretiert. Das Leerzeichen stoppt den Einlese-Vorgang, da es nicht mehr als Integer interpretiert werden kann. Der interne Postions-Zeiger der Datei steht hinter der „2“ und vor dem Leerzeichen.
   - Der Integer bekommt den Wert „12“.
   - Status: *NOT-FAIL* und *NOT-EOF.*
3. Der nächste Integer wird gelesen.
   - Dazu wird das Leerzeichen (als Whitespace) überlesen, und dann die Zeichen „3“ und „4“ als Integer interpretiert. Das Ende der Datei (eof) Leerzeichen stoppt den Einlese-

Vorgang. Der interne Postions-Zeiger der Datei steht auf dem „eof“.
- Der Integer bekommt den Wert „34“.
- Status: *NOT-FAIL* und *EOF*, denn es wurde was korrekt gelesen („34“), und das Datei-Ende wurde erreicht.

```
         1       2                3       4    eof
1.    ^
2.                   ^
3.                                             ^
```

Also:

- Nur *FAIL* gibt an, ob was gelesen werden konnte, oder nicht.
- *EOF* gibt nur an, ob zusätzlich noch das Datei-Ende erreicht wurde.

13.1.1.6 Fehl-Benutzung „Abfrage vor dem Lesen“

Egal ob korrekterweise mit „fail()“ oder fälschlicherweise mit „eof()“: es wird immer wieder gerne erst die Abfrage gemacht und dann gelesen. So in folgender Art:

```
// Fehlerhafter Code - so nicht, niemals

std::ifstream in(...);
while (!in.eof())
{
   lese was...
}
```

Das funktioniert nicht, da **ein Stream beim Lesen nicht nach vorne schaut** - das würde Performance kosten. Wenn ein Stream das nächste Zeichen beim Lesen nicht auswerten muß, dann kennt er es auch nicht, und kann es auch nicht bewerten. Und daher weiss er auch nicht, ob das Lesen fehlschlagen wird, oder ob er das Datei-Ende erreichen wird.

Wenn Sie z.B. eine Datei Zeichen für Zeichen lesen, dann liest ein Stream auch immer nur ein Zeichen. Alles Weitere wäre ineffizient.

In C++ muss man erst Lesen, dann weiss man, ob das Lesen geklappt hat (*GOOD*) oder nicht (*FAIL*). Und wenn es nicht geklappt hat, ob es am Datei-Ende (*EOF*) lag. In die Zukunft schauen können und sollen Streams nicht.

Das Problem hierbei ist, dass viele Anfänger die Fehlerbehandlung gerne vernachlässigen, und sich dann - vom Namen fehlgeleitet - auf die Element-Funktion „eof()“ stürzen. Und in manchen Situationen funktioniert dann der obige falsche Code leider - wie im folgenden Beispiel.

```
// Achtung - fehlerhafter Code!

#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
```

```
   {
      ofstream out("temp.txt");
      out << "Zeile 1\nZeile 2\nZeile 3\n";        // mit '\n' am Ende
   }

   string s;
   ifstream in("temp.txt");
   for (;;)
   {
      getline(in, s);
      if (in.eof()) break;                         // So nicht, niemals
      cout << '"' << s << "\"\n";
   }
}
```

```
Ausgabe
"Zeile 1"
"Zeile 2"
"Zeile 3"
```

Leider funktioniert das Beispiel schon nicht mehr, wenn die Datei nicht mit einem ‚\n' abgeschlossen ist.

```
// Achtung - fehlerhafter Code!

#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   {
      ofstream out("temp.txt");
      out << "Zeile 1\nZeile 2\nZeile 3";          // ohne '\n' am Ende
   }

   string s;
   ifstream in("temp.txt");
   for (;;)
   {
      getline(in, s);
      if (in.eof()) break;                         // so nicht, niemals
      cout << '"' << s << "\"\n";
   }
}
```

```
Ausgabe
"Zeile 1"
"Zeile 2"
```

Jetzt fehlt die dritte Zeile. Denn „getline" liest bis zum abschliesenden ‚\n', das in der dritten Zeile nicht vorhanden ist. Statt dessen trifft das Lesen auf das Datei-Ende, und der Stream geht natürlich in den Status *EOF*. Das dabei der Status *FAIL* nicht gesetzt ist, d.h. das Einlesen noch geklappt hat und „s" ausgewertet werden kann - das geht hier ganz unter. Ist halt fehlerhafter Code.

Hinweis – man findet solchen fehlerhaften Code in der Praxis erstaunlich häufig wieder, da er doch oft lange Zeit fehlerfrei läuft. Dies liegt daran, dass a) Datei-Operationen meistens halt doch gut gehen, b) viele Programme Dateien zeilenweise lesen, und c) viele Programme ihre Ausgaben immer mit ‚\n' abschliessen. Das ändert aber nichts daran, dass der Code falsch ist, und irgendwann zu Problemen führen wird.

Also machen Sie es richtig:

- Erst versuchen zu lesen, dann den Stream-Status auswerten.
- Hier primär *FAIL* auswerten, und dann sekundär noch *EOF*.
- Und nur Eingaben auswerten, wenn der Stream-Status *GOOD* ist.

13.1.2 Datei zum Anhängen öffnen

Wenn Sie eine Datei zum Schreiben öffnen und diese Datei existiert schon, so geht beim *„normalen“* Öffnen der alte Inhalt verloren – der alte Datei-Inhalt wird gelöscht und die Schreib-Position auf den Anfang der Datei gesetzt. Soll dies verhindert werden, so muß beim Öffnen zusätzlich das Flag „std::ios::app“ angegeben werden.

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

int main()
{
   {
      ofstream out("temp.txt");
      out << "12";
   }
   {
      ofstream out("temp.txt");               // Loescht den alten Inhalt "12"
      out << "34";
   }
   {
      int n;
      ifstream in("temp.txt");
      in >> n;
      cout << n << '\n';                      // => nur "34"
   }

   {
      ofstream out("temp.txt", ios::app);     // Oeffnet zum Anhaengen dank Flag "app"
      out << "56";
   }
   {
      int n;
      ifstream in("temp.txt");
      in >> n;
      cout << n << '\n';                      // => "3456"
   }
}
```

Ausgabe
```
34
3456
```

13.1.3 Binäre Dateien

Wollen Sie Objekte als Bitmuster abspeichern und laden, so müssen Sie den Stream mit dem Flag „std::ios::binary“ ‘binär’ öffnen, und dann auf die Element-Funktionen *„read“* und *„write“* zurückgreifen. Diese Funktionalität werden wir nicht tiefergehend besprechen – mit einem kleinen Beispiel soll das Ganze aber vorgestellt werden – wieder mal ohne Fehlerbehandlung. Achtung – für *„read“* und *„write“* werden Adressen, Zeiger und „sizeof“ benötigt, die in der Vorlesung aus Zeitmangel nicht eingeführt werden. Fassen Sie das Beispiel also bitte nur als ein Hinweis für später auf, wenn Sie diese Funktionalität mal benötigen.

```
#include <iostream>
```

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

int main()
{
   {
      ofstream out("temp.dat", ios::binary);    // Binaer oeffnen mit Flag "binary"
      int n = 1234567890;
      out.write(reinterpret_cast<const char*>(&n), sizeof(int));
   }
   {
      int n;
      ifstream in("temp.dat", ios::binary);
      in.read(reinterpret_cast<char*>(&n), sizeof(int));
      cout << n << '\n';                                   // => "1234567890"
   }
}
```

Ausgabe
```
1234567890
```

Hinweis – binäres Öffnen wird, analog zum Anhängen in Kapitel Kapitel 13.1.2, mit dem Flag „std::ios::binary“ durchgeführt, das beim Erzeugen des File-Streams angeben werden muß. Diese Flags können mit dem Bit-Oder-Operator „|“ miteinander kombiniert werden.

Öffnen Sie die erzeugte Datei ruhig mal in einem Editor – Sie werden keine textuelle Darstellung der Int-Zahl „1234567890“ in ihr finden, sondern statt dessen nur ein unverständliches Byte-Muster – so wie der „int“ eben im Speicher aussieht. Bei mir im Editor sieht es z.B. aus wie in der folgenden Abbildung:

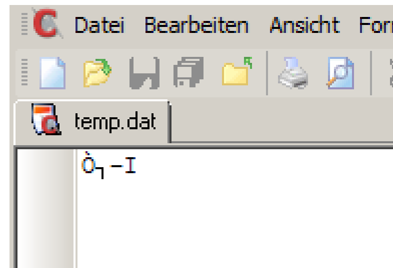

Abb. 13-1 : Datei, binär beschrieben - geöffnet in einem Editor

Für mehr Details konsultieren Sie bitte ein entsprechendes Buch oder z.B. entsprechende Seiten im Internet.

13.2 Filesystem-Library

Mit den File-Streams aus Kapitel 13.1 können wir zwar Streams an Dateien binden, und so Dateien lesen oder schreiben – aber mehr eben auch nicht. Schon bei den einfachsten Fragen und Operationen auf Datei-Ebene helfen uns die File-Streams nicht weiter:

- Existiert eine bestimmte Datei oder ein bestimmtes Verzeichnis?
- Ist ein Name eine Datei oder ein Verzeichnis?
- Wie groß ist eine Datei?
- Welche Dateien befinden sich in einem Verzeichnis?
- Kopieren, verschieben oder löschen von Dateien und/oder Verzeichnissen.

Für diese Probleme gibt es in C++ die Filesystem-Library.

Achtung – treten bei der Nutzung der Filesystem-Library Laufzeit-Probleme auf, z.B. wenn Sie versuchen die Größe einer Datei zu bestimmen, die es gar nicht gibt oder auf die Sie keine Leserechte haben – so wird die Filesystem-Library dies mit einer Exceptions melden. Nun lernen wir Exceptions leider erst später kennen. Bis dahin sollten Sie also nur Programme schreiben, die immer funktionieren ;-)

13.2.1 Test-Daten erstellen

Wenn wir beim Arbeiten mit dem Datei-System reproduzierbare und übertragbare Ergebnisse erhalten wollen, benötigen wir stabile Test-Daten. Ich habe mir darum auf meiner Platte direkt unter “C:\” eine Test-Verzeichnis-Struktur mit Test-Dateien erstellt:

```
Test-Verzeichnis-Struktur
C:\
└ 🗁 DateiSystemBeispiele
  └ 🗁 verzeichnis1
  │ └ 🗁 verzeichnis3
  │ │ └ 🗁 verzeichnis4
  │ │ │ └ 🖹 find.txt      - 2286 Byte
  │ │ │ └ 🖹 search.dat     - 1680 Byte
  │ │ │ └ 🖹 test.txt      -  270 Byte
  │ │ └ 🖹 test.txt        -   42 Byte
  │ └ 🗁 verzeichnis5
  │ │ └ 🖹 find.txt        -  683 Byte
  │ │ └ 🖹 test.txt        -  482 Byte
  │ └ 🖹 daten.dat          -  192 Byte
  │ └ 🖹 test.txt          - 1298 Byte
  └ 🗁 verzeichnis2
  │ └ 🖹 find.txt          -  250 Byte
  │ └ 🖹 test.txt          - 3078 Byte
  └ 🖹 search.dat          - 1600 Byte
  └ 🖹 test.txt            -  912 Byte
```

Diese Struktur werde ich in allen Beispielen dieses Kapitels und auch in den Aufgaben benutzen. Darum ist die Struktur auch aufwändiger als für unsere ersten Beispiele notwendig – sie muß auch den Anforderungen der kommenden Aufgaben genügen.

- Zur besseren Übersicht sind die Dateien markiert – alle „test.txt“ Dateien sind grün, alle „find.txt“ Dateien sind blau, und alle Dateien mit der Endung „.dat“ sind rot.
- Zusätzlich ist die Größe der Dateien für einige Beispiele und Aufgaben wichtig – darum habe ich sie in der Struktur mit aufgeführt.

13.2.2 Existenz, Typ und Größe für Dateien ermitteln

Zuerst wollen wir die Funktionen für die Ermittelung von Existenz, Typ und Größe in der Filesystem-Library kennen lernen. Alle Funktionen erwarten von der Deklaration her eigentlich ein “Filesystem.Path” Objekt und keinen String – können aber auch mit einem normalem String aufgerufen werden.

- `bool` **`exists`**`(const std::filesystem::path&);`
  Gibt zurück, ob das übergebene Element (Datei, Verzeichnis,…) existiert
- `bool` **`is_regular_file`**`(const std::filesystem::path&);`
  Gibt zurück, ob das übergebene Element eine normale Datei ist
  Achtung – existiert das Element nicht, so wird eine Exception geworfen
- `bool` **`is_directory`**`(const std::filesystem::path&);`
  Gibt zurück, ob das übergebene Element ein Verzeichnis ist
  Achtung – existiert das Element nicht, so wird eine Exception geworfen
- `std::uintmax_t` **`file_size`**`(const std::filesystem::path&);`
  Gibt die Größe des übergebenen Elements zurück
  Achtung – existiert die Datei nicht oder ist das Element keine Datei, so wird eine Exception geworfen

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muessen die Pfad-Namen angepasst werden.

#include <iostream>
#include <string>
#include <vector>
#include <filesystem >
using namespace std;
using namespace std::filesystem;

// Funktion gibt aus, ob das uebergebene Element existiert
void check_file_existence(const string& file)
{
   cout << "- " << file << " => " << exists(file) << '\n';
}

// Funktion gibt den Typ des uebergebenen Elements aus, wenn es existiert
void check_file_type(const string& file)
{
   cout << "- " << file << " => ";
   if (!exists(file))
   {
      cout << "---\n";
      return;
   }

   if (is_regular_file(file))
   {
      cout << "Datei\n";
   }
   else if (is_directory(file))
   {
      cout << "Verzeichnis\n";
   }
   else
   {
      cout << "unbekannter Typ\n";
   }
}

// Funktion gibt die Groesse des uebergebenen Elements aus,
// wenn es existiert und eine Datei ist
void check_file_size(const string& file)
{
   cout << "- " << file << " => ";
   if (exists(file) && is_regular_file(file))
   {
      cout << file_size(file) << " Byte\n";
   }
   else
   {
      cout << "---\n";
```

```
   }
}

int main()
{
   cout << boolalpha;

   vector<string> files;
   // Korrekter Datei-Name
   files.push_back("C:\\DateiSystemBeispiele\\search.dat");
   // Fehlerhafter Datei-Name
   files.push_back("C:\\DateiSystemBeispiele\\xy.z");
   // Korrekter Verzeichnis-Name
   files.push_back("C:\\DateiSystemBeispiele");
   // Fehlerhafter Verzeichnis-Name
   files.push_back("C:\\DateiSystemBeispiele\\kein-pfad\\search.dat");

   cout << "Datei-Existenz:\n";
   for (vector<string>::const_iterator it=files.begin(); it!=files.end(); ++it)
   {
      check_file_existence(*it);
   }

   cout << '\n';

   cout << "Datei-Typ:\n";
   for (vector<string>::const_iterator it=files.begin(); it!=files.end(); ++it)
   {
      check_file_type(*it);
   }

   cout << '\n';

   cout << "Datei-Groesse:\n";
   for (vector<string>::const_iterator it=files.begin(); it!=files.end(); ++it)
   {
      check_file_size(*it);
   }
}
```

```
Ausgabe (unter Voraussetzung der obigen Datei-Struktur)
Datei-Existenz:
- C:\DateiSystemBeispiele\search.dat => true
- C:\DateiSystemBeispiele\xy.z => false
- C:\DateiSystemBeispiele => true
- C:\DateiSystemBeispiele\kein-pfad\search.dat => false

Datei-Typ:
- C:\DateiSystemBeispiele\search.dat => Datei
- C:\DateiSystemBeispiele\xy.z => ---
- C:\DateiSystemBeispiele => Verzeichnis
- C:\DateiSystemBeispiele\kein-pfad\search.dat => ---

Datei-Groesse:
- C:\DateiSystemBeispiele\search.dat => 1600 Byte
- C:\DateiSystemBeispiele\xy.z => ---
- C:\DateiSystemBeispiele => ---
- C:\DateiSystemBeispiele\kein-pfad\search.dat => ---
```

Hinweise:

- Das obige Beispiel-Programm ist so geschrieben, daß die Filesystem-Funktionen wie z.B. „is_regular_file“ oder „file_size“ keine Exceptions werfen können. Dafür prüfe ich vor dem Aufruf von z.B. „file_size“ extra die Existenz und den Typ der Datei ab.
- Falls Sie nicht unter Windows arbeiten, müssen Sie die Pfade an die Begebenheiten Ihres Betriebssystems anpassen – also z.B. unter Linux die Backshlashes durch Shlashes ersetzen und den Root-Pfad einfach auf „/“ setzen.

13.2.3 Filesystem.Path Klasse

Unser erstes Programm mit der Filesystem-Library geht mit einer großen Hypothek ins Rennen: es ist nicht plattform-unabhängig, sondern ganz im Gegenteil so nur unter Windows lauffähig. Das Problem ist, daß die angegebenen Pfad- und Datei-Namen windows-spezifisch sind. Dieses Problem ist einer der Hintergründe des Filesystem.Path Objektes, das eigentlich an die Funktionen übergeben wird.

Filesystem.Path:
- Die Path-Klasse kapselt einen plattform-unabhängigen Datei-Namen (inkl. Pfad).
- Außerdem unterstützt die Path-Klasse die verschiedenen Zeichen-Codes Ihrer jeweiligen Plattform – Sie können es also z.B. mit ASCII aber auch mit Unicode nutzen.

Für genauere Informationen zu beiden Themen möchte ich Sie an die Filesystem.Library Dokumentation verweisen – beides sprengt den Rahmen des Tutorials.

Zusätzlich hat die Path-Klasse viele hilfreiche Element-Funktionen, um Pfad und Datei-Namen zu untersuchen, zu zerlegen und zu modifizieren. Die folgende Auflistung beschreibt ein paar dieser Funktionen. Aber Achtung – dies ist keine Referenz. Die Funktions-Signaturen und Erklärungen sind zum Teil nicht vollständig bzw. etwas vereinfacht. Für viele Zwecke ist dies ausreichend – wenn Sie aber tiefer einsteigen wollen oder müssen, dann sollten Sie die Dokumentation zur Hilfe ziehen:

- `"string" native() const;`
  Gibt den Pfad im nativen Format zurück.
  Achtung – die Rückgabe muß nicht vom Typ “std::string” sein, den wir kennen.
- `"string" string() const;`
  Gibt den Pfad als String zurück.
  Achtung – die Rückgabe muß nicht vom Typ “std::string” sein, den wir kennen.
- `"string" generic_string() const;`
  Gibt den Pfad als *generischen quasi plattform-unabhängigen* String zurück.
  Achtung – die Rückgabe muß nicht vom Typ “std::string” sein, den wir kennen.
- `bool has_root_name() const;`
  Gibt zurück, ob das Path-Objekt einen Root-Namen enthält.
  Ein Root-Name ist z.B. eine Netzwerk-Location oder ein Laufwerk-Identifier
- `path root_name() const;`
  Gibt den Root-Namen des Path-Objekts als Path-Objekt zurück.
  Existiert kein Root-Name, so wird ein leeres Path-Objekt zurückgegeben.
- `bool has_root_directory() const;`
  Gibt zurück, ob das Path-Objekt ein Root-Verzeichnis enthält.
- `path root_directory() const;`
  Gibt das Root-Verzeichnis des Path-Objekts als Path-Objekt zurück.
  Existiert kein Root-Verzeichnis, so wird ein leeres Path-Objekt zurückgegeben.
- `bool has_root_path() const;`
  Gibt zurück, ob das Path-Objekt einen Root-Pfad enthält.
- `path root_path() const;`

Gibt den Root-Pfad des Path-Objekts als Path-Objekt zurück.
Existiert kein Root-Pfad, so wird ein leeres Path-Objekt zurückgegeben.

- `bool` **`has_filename`**`() const;`
  Gibt zurück, ob das Path-Objekt einen Datei-Namen enthält.
- `path` **`filename`**`() const;`
  Gibt den Datei-Namen des Path-Objekts als Path-Objekt zurück.
  Existiert kein Datei-Name, so wird ein leeres Path-Objekt zurückgegeben.
- `bool` **`has_stem`**`() const;`
  Gibt zurück, ob das Path-Objekt einen Haupt-Datei-Namen (Stamm) enthält.
- `path` **`stem`**`() const;`
  Gibt den Haupt-Datei-Namen (Stamm) des Path-Objekts als Path-Objekt zurück.
  Existiert kein Haupt-Datei-Name, so wird ein leeres Path-Objekt zurückgegeben.
- `bool` **`has_extension`**`() const;`
  Gibt zurück, ob das Path-Objekt eine Extension enthält.
- `path` **`extension`**`() const;`
  Gibt die Extension des Path-Objekts als Path-Objekt zurück.
  Existiert keine Extension, so wird ein leeres Path-Objekt zurückgegeben.

Und ein zugehöriges Beispiel-Programm – basierend auf der Datei-Struktur aus Kapitel 13.2.1. Achtung – das Programm enthält wieder einen windows-spezifischen Pfad.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

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

int main()
{
   cout << boolalpha;

   path p("C:\\DateiSystemBeispiele\\verzeichnis1\\daten.dat");

   // "Alte" schon aus Kapitel 13.2.2 bekannte Funktionen
   cout << "Exist: " << exists(p) << '\n';
   cout << "Direc: " << is_directory(p) << '\n';
   cout << "File:  " << is_regular_file(p) << '\n';
   cout << "Size:  " << file_size(p) << '\n';

   // "Neue" Element-Funktionen der Klasse "path"
   cout << "Nativ: " << p << '\n';
   cout << "Strin: " << p.string() << '\n';
   cout << "Gener: " << p.generic_string() << '\n';
   cout << "Ro-Na: " << p.has_root_name() << " - " << p.root_name() << '\n';
   cout << "Ro-Di: " << p.has_root_directory() << " - " << p.root_directory() << '\n';
   cout << "Ro-Pa: " << p.has_root_path() << " - " << p.root_path() << '\n';
   cout << "Filen: " << p.has_filename() << " - " << p.filename() << '\n';
   cout << "Stem:  " << p.has_stem() << " - " << p.stem() << '\n';
   cout << "Exten: " << p.has_extension() << " - " << p.extension() << '\n';
}
```

```
Ausgabe (unter Voraussetzung der obigen Datei-Struktur)
Exist: true
Direc: false
File:  true
Size:  192
Nativ: "C:\DateiSystemBeispiele\verzeichnis1\daten.dat"
```

```
Strin: C:\DateiSystemBeispiele\verzeichnis1\daten.dat
Gener: C:/DateiSystemBeispiele/verzeichnis1/daten.dat
Ro-Na: true - "C:"
Ro-Di: true - "\"
Ro-Pa: true - "C:\"
Filen: true - "daten.dat"
Stem:  true - "daten"
Exten: true - ".dat"
```

Nachdem wir nun gelernt haben, wie wir Informationen zu einer einzelnen Datei oder einem Verzeichnis bekommen – ist die nächste wichtige Frage: wie kommt man an alle Dateien in einem Verzeichnis?

13.2.4 Verzeichnis auslesen

Um den Inhalt eines Verzeichnisses auszulesen, benötigt man zwei Verzeichnis-Iteratoren – dafür gibt es in der Filesystem-Library den Typ „directory_iterator“. Den Start-Iterator erhält man, indem man den Iterator mit dem Verzeichnis-Namen initialisiert – den Ende-Iterator durch eine einfache Erzeugung ohne Argumente. Der dereferenzierte Iterator liefert hierbei ein „directory_enty“ Element zurück, das uns über „path()“ den Zugriff auf ein Path-Objekt ermöglicht.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

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

int main()
{
   cout << boolalpha;

   directory_iterator it("C:\\DateiSystemBeispiele\\verzeichnis1\\verzeichnis3");// (*)
   directory_iterator eit;                                                      // (**)
   for (; it!=eit; ++it)
   {
      const path& p = it->path();                                              // (***)
      cout << "Native:    " << p << '\n';
      cout << "- Name:    " << p.filename() << '\n';
      cout << "- Datei:   " << is_regular_file(p) << '\n';
      cout << "- Verzei:  " << is_directory(p) << '\n';
      if (is_regular_file(p))
      {
         cout << "- Groesse: " << file_size(p) << '\n';
      }
      cout << '\n';
   }
}
```

```
Ausgabe (unter Voraussetzung der obigen Datei-Struktur)
Native:    "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis3\test.txt"
- Name:    "test.txt"
- Datei:   true
- Verzei:  false
- Groesse: 42

Native:    "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4"
- Name:    "verzeichnis4"
- Datei:   false
- Verzei:  true
```

Hinweise:

- Zeile (*) erzeugt den Start-Iterator durch den Verzeichnis-Namen als Argument
- Zeile (**) erzeugt den Ende-Iterator durch Angabe keiner Argumente
- Statt in fast jeder Schleifen-Zeile über den Iterator das Path-Objekt mit „it->path()" zu holen, wird es in Zeile (***) in „p" zwischen *„gecacht"*. Da es ein Objekt ist, wird es nicht kopiert sondern als Referenz *„gecacht"* – und da es nicht verändert werden soll als Const-Referenz.

13.2.5 Rekursive Datei-Suche

Ein typisches Problem, dass wir nun lösen können, ist die tiefe Suche im Dateisystem z.B. nach einer Datei mit einem bestimmten Namen. Spätestens jetzt sollten Sie sich die kleine Beispiel Verzeichnis-Struktur aus Kapitel 13.2.1 anlegen. Ich habe es leider erlebt, dass Anfänger ihr Programm zum Test auf ihr Root-Dateisystem wie z.B. „C:\" losgelassen haben, und sich dann gewundert haben dass ihr Programm ewig läuft, Fehler meldet oder scheinbar einfach nur den Rechner lahm legt. Ist doch auch kein Wunder, oder? Bei unseren heutigen Plattengrößen gibt es da ziemlich viele Verzeichnisse zu durchsuchen und Datei-Namen zu vergleichen, so das der Rechner gut belastet ist. Und mit ziemlicher Sicherheit trifft das Programm unterwegs auf Verzeichnisse, für die der aktuelle Nutzer keine Rechte hat – und schon regnet es Exceptions, und damit können wir noch gar nicht gut umgehen.

Lange Rede, kurzer Sinn – legen Sie sich eine kleine Test-Daten-Struktur auf Ihrer Platte an, z.B. die aus Kapitel 13.2.1. Suchen werden wir dann nach den drei Dateien mit dem Namen „find.txt".

Damit können wir nun loslegen. Der erste Schritt ist einfach. Wir laufen einfach über das Such-Verzeichnis und geben alles aus, was wir finden – getrennt nach Verzeichnissen und Dateien.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

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

int main()
{
   directory_iterator it("C:\\DateiSystemBeispiele");
   directory_iterator eit;
   for (; it!=eit; ++it)
   {
      const path& p = it->path();
      if (is_regular_file(p))
      {
         cout << "Datei: " << p << '\n';
      }
      else if (is_directory(p))
      {
         cout << "Verz:  " << p << '\n';
      }
   }
}
```

```
Mögliche Ausgabe (Andere Reihenfolge moeglich - die ist nicht definiert)
```

```
Datei: "C:\DateiSystemBeispiele\search.dat"
Datei: "C:\DateiSystemBeispiele\test.txt"
Verz:  "C:\DateiSystemBeispiele\verzeichnis1"
Verz:  "C:\DateiSystemBeispiele\verzeichnis2"
```

Der nächste Schritt ist minimal – wir vergleichen den gefundenen Datei-Namen mit dem Namen, den wir suchen – und damit wir was finden, suchen wir erstmal nach „test.txt“, denn im unteren Verzeichnis gibt es ja keine Datei „find.txt“. Wenn der Datei-Name stimmt, daher wir einen Treffer gelandet haben, dann geben wir den Datei-Pfad komplett aus. Und bei Verzeichnissen machen wir erstmal nix.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

#include <iostream>
#include <string>
#include <filesystem>
using namespace std;
using namespace std::filesystem;

int main()
{
   const string searchfile("test.txt");

   directory_iterator it("C:\\DateiSystemBeispiele");
   directory_iterator eit;
   for (; it!=eit; ++it)
   {
      const path& p = it->path();
      if (is_regular_file(p) && p.filename()==searchfile)
      {
         cout << "-> " << p << '\n';
      }
      else if (is_directory(p))
      {
         // Hier machen wir erstmal nichts
      }
   }
}
```

```
Ausgabe
-> "C:\DateiSystemBeispiele\test.txt"
```

Kommen wir nun zum Fall, dass das Verzeichnis selber wieder ein Verzeichnis enthält. In diesem Fall müssen wir nur für dieses Verzeichnis das Gleiche machen: alle Elemente durchlaufen, Dateien vergleichen, bei Verzeichnissen wieder das Gleiche. Das klingt nach Rekursion, wie wir sie im Funktionen Kapitel kennen gelernt haben. Wichtig – dazu müssen wir die Funktionalität in eine Funktion auslagern – sonst können wir sie nicht rekursiv aufrufen. Und mehr machen wir in diesem Schritt auch nicht – wir verschieben die Schleife in eine Funktion:

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

#include <iostream>
#include <string>
#include <filesystem>
using namespace std;
using namespace std::filesystem;

void searchFileInPath(const path& searchpath, const string& searchfile)
{
   directory_iterator it(searchpath);
   directory_iterator eit;
   for (; it!=eit; ++it)
```

```
   {
      const path& p = it->path();
      if (is_regular_file(p) && p.filename()==searchfile)
      {
         cout << "-> " << p << '\n';
      }
      else if (is_directory(p))
      {
         // Hier machen wir erstmal nichts
      }
   }
}

int main()
{
   const string searchfile("test.txt");
   path searchpath("C:\\DateiSystemBeispiele");
   searchFileInPath(searchpath, searchfile);
}
```

```
Ausgabe
-> "C:\DateiSystemBeispiele\test.txt"
```

Nachdem unsere Verzeichnis-Durchforste-Funktionalität nun in einer Funktion ist, können wir sie jetzt auch im Falle eines Verzeichnisses rekursiv aufrufen. Und außerdem ändern wir den Such-Datei-Namen in „find.txt“ um, denn diesen Namen wollten wir ja suchen. Letzlich müsen wir also nur zwei kleine Änderungen machen.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

#include <iostream>
#include <string>
#include <filesystem >
using namespace std;
using namespace std::filesystem;

void searchFileInPath(const path& searchpath, const string& searchfile)
{
   directory_iterator it(searchpath);
   directory_iterator eit;
   for (; it!=eit; ++it)
   {
      const path& p = it->path();
      if (is_regular_file(p) && p.filename()==searchfile)
      {
         cout << "-> " << p << '\n';
      }
      else if (is_directory(p))
      {
         searchFileInPath(p, searchfile);                          // << Aenderung
      }
   }
}

int main()
{
   const string searchfile("find.txt");                           // << Aenderung
   path searchpath("C:\\DateiSystemBeispiele");
   searchFileInPath(searchpath, searchfile);
}
```

```
Mögliche Ausgabe (Andere Reihenfolge moeglich - die ist nicht definiert)
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis2\find.txt"
```

Wow – alles funktioniert. So einfach und elegant ist Rekursion. Damit unser Quelltext noch etwas mehr nach richtigem Programm aussieht, integrieren wir noch ein paar Dinge:

- Ausgabe des Programm-Namens
- Ausgabe, wo gesucht wird
- Ausgabe, wonach gesucht wird
- Und eine Sicherheits-Abfrage, falls jemand „searchFileInPath“ nicht mit einem korrekten Verzeichnis als ersten Parameter aufruft.

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

#include <iostream>
#include <string>
#include <filesystem>
using namespace std;
using namespace std::filesystem;

void searchFileInPath(const path& searchpath, const string& searchfile)
{
   if (!is_directory(searchpath))                                    // << Aenderung
   {
      cout << "Fehler - " << searchpath << " ist kein Verzeichnis\n";
      return;
   }

   directory_iterator it(searchpath);
   directory_iterator eit;
   for (; it!=eit; ++it)
   {
      const path& p = it->path();
      if (is_regular_file(p) && p.filename()==searchfile)
      {
         cout << "-> " << p << '\n';
      }
      else if (is_directory(p))
      {
         searchFileInPath(p, searchfile);
      }
   }
}

int main()
{
   cout << "Rekursive Datei-Suche\n";                 // << Aenderung

   const string searchfile("find.txt");
   path searchpath("C:\\DateiSystemBeispiele");

   cout << "Suche in: " << searchpath << '\n';        // << Aenderung
   cout << "Nach: " << searchfile << '\n';

   searchFileInPath(searchpath, searchfile);
}
```

Mögliche Ausgabe (Andere Reihenfolge moeglich - die ist nicht definiert)

```
Rekursive Datei-Suche
Suche in: "C:\DateiSystemBeispiele"
Nach: find.txt
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis2\find.txt"
```

Hinweis – in den Aufgaben werden wir dieses Programm zur rekursiven Datei-Suche um weitere Fähigkeiten ergänzen.

13.2.6 Rekursive Datei-Suche noch einfacher

Im letzten Kapitel haben wir ein typisches Problem kennen gelernt, das mit Rekursion sehr

einfach zu lösen ist. Nur hätten wir das eigentlich gar nicht selber machen müssen – da dies ein ganz typisches Problem ist, liefert dieFilesystem Library schon einen Rekursiven-Verzeichnis-Iterator mit „recursive_directory_iterator“. Der übernimmt die Rekursion intern – und wir müssen eigentlich fast gar nichts mehr machen:

```
// Achtung - in dieser Form ist das Programm nur unter Windows lauffaehig
// Unter anderen Systemen muss der Pfad-Name angepasst werden.

#include <iostream>
#include <string>
#include <filesystem>
using namespace std;
using namespace std::filesystem;

void searchFileInPath(const path& searchpath, const string& searchfile)
{
   if (!is_directory(searchpath))
   {
      cout << "Fehler - " << searchpath << " ist kein Verzeichnis\n";
      return;
   }

   recursive_directory_iterator it(searchpath);                       // << Aenderung
   recursive_directory_iterator eit;                                  // << Aenderung
   for (; it!=eit; ++it)
   {
      const path& p = it->path();
      if (is_regular_file(p) && p.filename()==searchfile)
      {
         cout << "-> " << p << '\n';
      }
      // Behandlung von Verzeichnissen geloescht                          << Aenderung
   }
}

int main()
{
   cout << "Rekursive Datei-Suche\n";

   const string searchfile("find.txt");
   path searchpath("C:\\DateiSystemBeispiele");

   cout << "Suche in: " << searchpath << '\n';
   cout << "Nach: " << searchfile << '\n';

   searchFileInPath(searchpath, searchfile);
}
```

```
Mögliche Ausgabe (Andere Reihenfolge moeglich - die ist nicht definiert)
Rekursive Datei-Suche
Suche in: "C:\DateiSystemBeispiele"
Nach: find.txt
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt"
-> "C:\DateiSystemBeispiele\verzeichnis2\find.txt"
```

Trotzdem war es natürlich gut, daß wir im letzten Kapitel die rekursive Datei-Suche mal selber implementiert haben. Ein bisschen Übung und Verständnis schaden nie.

13.3 Exit

Man kann ein Programm jederzeit hart beenden, in dem man die Funktion „std::exit(int)“ aus dem Header „cstdlib“ aufruft. Hart beenden meint, daß das Programm direkt abbricht.

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

void fct()
{
   cout << "fct start\n";
   exit(1);
   cout << "fct ende\n";
}

int main()
{
   cout << "main start\n";
   fct();
   cout << "main ende\n";
}
```

```
Ausgabe
main start
fct start
```

Der an „std::exit" übergebene Int-Parameter entspricht der Int-Rückgabe der Main-Funktion. Wie in der Main-Funktion gibt man eine „0" zurück, wenn das Programm seine Funktion ordnungsgemäß ausgeführt hat, während alle anderen Werte als Ausführungs-Fehler gelten. Da ein „exit" innerhalb eines Programms meist die Reaktion auf einen Fehlerfall sein wird, wird man hier wohl meistens einen Wert ungleich „0" zurückgeben. Im den meisten Fällen sollte Ihr Programm aber nur einen Ausgang haben, und Sie sollten eine Fehlermeldung und –Behandlung mit Exceptions bevorzugen.

13.4 Datum, Zeit & Zeitmessungen

C enthält nur wenige Datum- und Zeit-Funktionalitäten, die natürlich auch in C++ verfügbar sind. In C++98 und C++03 waren dies die einzigen Datum- und Zeit-Funktionalitäten der Standard-Bibliothek. Mit C++11 wurde die Zeit-Bibliothek „Chrono" in C++ eingeführt, die neben Uhren auch das Konzept der Zeitpunkte und Zeitdauern unterstützt. In C++20 wurde Chrono dann um weitere Uhren, Zeitzonen, Kalender und Formatierungen erweitert. Leider sprengt die Chrono-Bibliothek den Rahmen der Vorlesung.

Für ein Beispiel im Kapitel 14 über Klassen benötigen wir den Zugriff auf das aktuelle Datum – darum wird dies noch in der alten C Art im nächsten Unterkapitel 13.4.1 eingeführt. Unterkapitel 13.4.2 zeigt im Gegensatz dazu ein Beispiel zur Nutzung von Chrono – hier die Messung von Zeitdauern.

13.4.1 Aktuelles Datum und Zeit in C

Um das aktuelle Datum und die aktuelle Zeit in C zu erfragen, werden mehrere Elemente aus dem Header „ctime" benötigt. Das folgende Beispiel zeigt die Benutzung dieser Elemente.

```
#include <iostream>
#include <ctime>

int main()
{
   std::time_t timer = std::time(0);
   std::tm* tblock = std::localtime(&timer);
```

```
   std::cout << "Jahr:    " << tblock->tm_year+1900 << '\n';    // +1900 muss sein
   std::cout << "Monat:   " << tblock->tm_mon+1 << '\n';        // +1 ist notwendig
   std::cout << "Tag:     " << tblock->tm_mday << '\n';
   std::cout << "Stunde:  " << tblock->tm_hour << '\n';
   std::cout << "Minute:  " << tblock->tm_min << '\n';
   std::cout << "Sekunde: " << tblock->tm_sec << '\n';
}
```

```
Mögliche Ausgabe
Jahr:    2020
Monat:   11
Tag:     29
Stunde:  20
Minute:  35
Sekunde: 18
```

Die genaue Erklärung des Beispiels möchte ich auslassen, da es zu weit geht. Letzlich geht es nur darum, dass Sie demnächst in der Lage sind das aktuelle Datum bzw. die aktuelle Zeit zu bestimmen - und das sollten Sie mit dieser Vorlage können.

Hinweis – die hier vorgestellten Elemente aus dem Header „ctime“ hat C++ von C geerbt, und stellen in aktuellem C++ eigentlich keinen adäquaten Umgang mit Datums- und Zeit-Funktionalitäten mehr dar. Nutzen Sie, wenn möglich, die Chrono Bibliothek.

13.4.2 Zeitmessung

Ab und zu benötigt man eine Zeitmessung, d.h. man will wissen, wie lange bestimmte Dinge gebraucht haben. Hierfür können wir die Chrono-Elemente aus C++11 nutzen. Mit einer Chrono-Uhr (hier „std::chrono::steady_clock“) können wir den aktuellen Zeitpunkt bestimmen (mit „now()“, und ihn in einer Zeitpunkt-Variablen (hier „start“ und „end“) abspeichern. Die Subtraktion zweier Zeitpunkte ergibt die Zeitdauer (Typ: „duration<double>“, Variable „elapsed_seconds“) in Sekunden. Durch den Typ in spitzen Klammern bei der Zeitdauer können wir bestimmen, in welcher Form wir die Zeitdauer haben wollen (hier als Fließkomma-Typ „double“).

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

int main()
{
   string s;

   cout << "Bitte druecken Sie die Return-Taste: ";
   auto start = steady_clock::now();
   getline(cin, s);
   auto end = steady_clock::now();
   duration<double> elapsed_seconds = end - start;
   cout << "Sie hatten eine Reaktionszeit von " << elapsed_seconds.count() << " s\n";

   cout << "\nUnd noch mal, vielleicht geht es diesmal schneller: ";
   start = steady_clock::now();
   getline(cin, s);
   end = steady_clock::now();
   elapsed_seconds = end - start;
   cout << "Sie hatten eine Reaktionszeit von " << elapsed_seconds.count() << " s\n";
}
```

Mögliche Ausgabe

```
Bitte druecken Sie die Return-Taste:
Sie hatten eine Reaktionszeit von 1.296 s

Und noch mal, vielleicht geht es diesmal schneller:
Sie hatten eine Reaktionszeit von 1.063 s
```

13.5 Zufallszahlen

Ab und zu benötigt man Zufallszahlen. Während in C++98 die Unterstützung von Zufallszahlen nur sehr rudimentär war, existiert seit C++11 eine große Anzahl an Zufalls-Elementen in der C++ Standard-Bibliothek. Die C++11 Zufallszahlen-Elemente sind – wie die meisten Teile der C++ Standard-Bibliothek – auf hohe Flexibilität, Erweiterbarkeit und Anpassbarkeit ausgelegt. Leider ist dadurch manchmal eine ganz einfache Benutzung aufwändiger, als man erstmal denkt.

Die C++ Zufallsszahlen basieren auf zwei Konzepten, den sogenannten Generatoren und den Distributionen.

- Generatoren erzeugen Reihen von Zufalls-Zahlen. Die C++ Standard-Bibliothek enthält dabei verschiedene Generator-Arten, die sich in der Zyklen-Länge, der Geschwindigkeit und der Verteilung der Zufalls-Zahlen unterscheiden. Der typische Generator für den Einstieg ist hierbei der „mt19937“ - ein Mersenne-Twister Generator der mit guter Performance recht gute Zufallszahlen liefert.
- Distributionen benutzen dann die Generatoren, und bilden die Zufalls-Zahlen auf ein Verteilungs-Muster ab. Mit den Distributionen kann also die eigentliche Verteilung der Zufalls-Zahlen beeinflusst werden. Einer der Tricks dabei ist, dass jede Distribution auch wieder als Generator fungieren kann – man kann also mehrere Distributionen problemlos hintereinander schalten. Eine typische Distribution ist hierbei „uniform_int_distribution“, die die Integer-Zahlen des vorgeschalteten Generators auf ein definierbares Integer-Intervall abbildet.

Folgendes Beispiel nutzt den Generator „std: mt19937“ und die Distribution „std::uniform_int_distribution“– beide aus dem Header „<random>“ – um 10 Zufallszahlen zwischen -4 und +6 (beide inkl.) auszugeben. Dazu werden zuerst der Generator „engine“ und die Distribution „dist“ erzeugt – bei der Distributions-Erzeugung werden die Grenzen angegeben. Die Zufallszahlen entstehen dann durch Aufruf der Distribution mit dem Generator „dist(engine)“.

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

int main()
{
   mt19937 engine;
   uniform_int_distribution<> dist(-4, +6);

   for (int i=0; i<10; ++i)
   {
      cout << dist(engine) << ' ';
   }
   cout << '\n';
}
```

```
Mögliche Ausgabe
2 -1 6 5 0 3 -4 2 1 0
```

Wenn Sie dieses Programm selber ausprobieren und mehrfach starten, so wird Ihnen sicher auffallen, dass das Programm bei jedem Aufruf immer die gleichen Zahlen ausgibt – dies ist sicher nicht sehr zufällig.

Das Problem ist, dass im Normalfall die Zufallszahlen in Ihrem Rechner nicht wirklich zufällig sind, sondern aus einer Rechenvorschrift entstehen – darum nennt man sie auch Pseudo-Zufallszahlen. Hierbei wird typischerweise aus einem existierenden Start-Wert ein neuer Wert berechnet, der dann die Zufallszahl ist. Dieser neue Wert wird dann gleichzeitig auch als Start-Wert für die nächsten Berechnung genommen. Ein sehr sehr schlechter und einfacher Zufallszahlen-Generator könnte also einfach die Addition um eins nutzen und würde dann folgende Zahlen liefern:

```
0 -> 1 -> 2 -> 3 -> 4 ...
```

Natürlich sind die Zufallszahlen-Generatoren der C++ Standard-Bibliothek viel besser und liefern bessere Pseudo-Zufallszahlen, aber alle basieren auf der Anforderung, dass man einen Start-Wert benötigt, und mit einem festen Start-Wert die Zahlen überhaupt nicht zufällig sind. Diesen Start-Wert nennt man „seed“, da er der Kern ist aus dem sich die Pseudo-Zufallszahlen entwickeln – und man kann ihn problemlos z.B. bei der Konstruktion setzen. Hier nochmal das gleiche Beispiel wie oben, nur dass wir den Start-Wert fest mit „42“ angeben:

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

int main()
{
   mt19937 engine(42);                         // <= 42 als Seed Konstruktor-Argument
   uniform_int_distribution<> dist(-4, +6);

   for (int i=0; i<10; ++i)
   {
      cout << dist(engine) << ' ';
   }
   cout << '\n';
}
```

```
Ausgabe
2 3 5 -1 -2 6 -1 -4 4 -2
```

Die Herausforderung ist es also, einen guten „zufälligen“ Start-Wert für unseren Generator zu bekommen. Da Problem sieht aus wie eine Schlange, die sich in den Schwanz beißt. Es gibt aber zwei Lösungen:

- Früher war die typische Lösung, den Zufallszahlen-Generator mit der Rückgabe von „std::time(0)“ (siehe Kapitel 13.4) als Seed zu füttern.
  - Aber die genaue Rückgabe von „std::time“ ist nicht definiert – meistens liefert die Funktion zwar die Anzahl an Sekunden bzw. Milli-Sekunden seit dem 1.1.1970 um 00:00 zurück – aber fest steht dies nicht. Die Idee hinter der Nutzung von „std::time“ ist daher, dass man zumindest jede Sekunde einen anderen Seed-Wert bekommt, d.h. die Zufallszahlen-Reihenfolge einen halbwegs zufälligen Start-Wert bekommt.

  - Leider hat diese Lösung mehrere Probleme, weshalb sie heute eigentlich nicht mehr empfohlen wird – in der Praxis aber noch viel eingesetzt wird.
- Statt dessen wird heutzutage empfohlen, einen anderen C++11 Zufallszahlen-Generator („std::random_device") zu nutzen, und dessen erste Zufallszahl dann als Seed für den eigentlichen Zufallszahlen-Generator zu verwenden.
  - Was im ersten Augenblick ziemlich dumm wirkt (hat „std::random_device" nicht das gleiche Problem?), ist es nicht. Denn „std::random_device" liefert im Gegensatz zu allen anderen Generatoren im Idealfall echte Zufallszahlen (mit Unterstützung der Hardware). Im schlechtesten Fall aber akzeptable echte Pseudo-Zufallszahlen.
  - Und warum nimmt man dann nicht direkt nur „std::random_device" und läßt alle anderen Pseudo-Generatoren links liegen? Leider hat „std::random_device" einige Nachteile, um seine Anforderungen erfüllen zu können – u.a. ist er meist sehr langsam und häufig liefern alle Objekte der Klasse die gleichen Zufallszahlen.

Die Lösung heute ist also, sich einen „std::random_device" Generator zu erzeugen, nur einen Wert auszulesen und diesen als Seed für seinen eigentlichen Generator zu verwenden. Das folgende Beispiel ist analog zu den oberen und macht es genauso:

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

int main()
{
   random_device rd;                               // Random-Device-Generator erzeugen
   mt19937 engine(rd());                           // Seed Argument auslesen und nutzen
   uniform_int_distribution<> dist(-4, +6);

   for (int i=0; i<10; ++i)
   {
      cout << dist(engine) << ' ';
   }
   cout << '\n';
}
```

```
Mögliche Ausgabe
5 5 -3 4 -1 1 -4 -3 -1 -4
```

Hinweis – wie die C++ Container-Library arbeitet die C++ Random-Number-Library mit abstrakten Konzepten – daher es gibt keine konkrete Klasse „generator" oder „distribution" oder ähnlich – dies sind nur abstrakte Konzepte. In der STL sind dies z.B. die Iteratoren, hier die Generatoren und die Distributionen. Der Vorteil ist, dass man problemlos eigene Generatoren und Distributionen schreiben kann, und so die Random-Number-Library problemlos anpassen und erweitern kann.

13.6 Mathematische Funktionen

Viele mathematische Funktionen finden sich im Header <cmath>, z.B. die Winkel-Funktionen für Sinus, Cosinus und Tangens, oder Funktionen für Wurzel- und Potenz-Berechungen.

Hier möchte ich nur zwei (genau genommen sechs) Funktion vorstellen, die wir schon gebraucht hätten bzw. noch einsetzen wollen: die Funktion „std::sqrt" für Wurzeln und die Funktion „std::log10" für den Zehner-Logarithmus.

13.6.1 Wurzeln mit „std::sqrt“

Die Funktion „std::sqrt“ berechnet die mathematische Wurzel aus einer Fließkomma-Zahl. Sie ist im Header „cmath“ definiert. Via Überladen gibt es die Funktion für alle drei Fließkomma-Typen – es sind also eigentlich drei Funktionen:

- float std::sqrt(float)
- double std:: sqrt(double)
- long double std:: sqrt(long double)

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

int main()
{
   cout << "sqrt(  1. ):  " << sqrt(  1.) << '\n';
   cout << "sqrt(  4.0):  " << sqrt(  4.0) << '\n';
   cout << "sqrt(  9.0):  " << sqrt(  9.0) << '\n';
   cout << "sqrt( 10.0):  " << sqrt( 10.0) << '\n';
   cout << "sqrt( 64.0):  " << sqrt( 64.0) << '\n';
   cout << "sqrt(259.21): " << sqrt(259.21) << '\n';
}
```

Ausgabe
```
sqrt(  1. ):  1
sqrt(  4.0):  2
sqrt(  9.0):  3
sqrt( 10.0):  3.16228
sqrt( 64.0):  8
sqrt(259.21): 16.1
```

13.6.2 Zehner-Logarithmus mit „std::log10“

Diese Funktion gibt es via Überladen für alle drei Fließkomma-Typen:

- float std::log10(float)
- double std::log10(double)
- long double std::log10(long double)

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

int main()
{
   cout << "log10(  1. ): " << log10(  1.) << '\n';
   cout << "log10(  1.5): " << log10(  1.5) << '\n';
   cout << "log10(  9.9): " << log10(  9.9) << '\n';
   cout << "log10( 10.0): " << log10( 10.0) << '\n';
   cout << "log10( 12.3): " << log10( 12.3) << '\n';
   cout << "log10( 98.7): " << log10( 98.7) << '\n';
   cout << "log10(123.4): " << log10(123.4) << '\n';
   cout << "log10(999.9): " << log10(999.9) << '\n';
}
```

Ausgabe
```
log10(  1. ): 0
log10(  1.5): 0.176091
log10(  9.9): 0.995635
log10( 10.0): 1
log10( 12.3): 1.08991
log10( 98.7): 1.99432
```

```
log10(123.4): 2.09132
log10(999.9): 2.99996
```

Ich möchte hier keine Mathematik treiben, und daher auch nicht auf die Frage eingehen, was der Zehner-Logarithmus genau ist. Stattdessen schauen wir uns an, wo wir ihn schon gebraucht hätten.

In einer Aufgabe wird eine schön formatierte Ausgabe gefordert. Dazu muss man wissen, wieviele Stellen die anzuzeigenden Zahlen haben, um entsprechende Einrückungen und Breiten setzen zu können. Eine mögliche Lösung ist das „Ausprobieren“ mit einer Schleife. Besser ist aber zum Beispiel die Berechnung über den Zehner-Logarithmus, da dies schneller, eleganter und daher besser wäre – denn der Zehner-Logarithmus quasi die Anzahl Stellen minus „1“ des Arguments liefert. Damit ist die „Breite“ einer Zahl nun schnell errechnet:

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

void fct(int n)
{
   int digits = static_cast<int>(log10(static_cast<double>(n))) + 1;
   cout << setw(4) << n << " hat " << digits << " Ziffern\n";
}

int main()
{
   fct(0);
   fct(9);
   fct(10);
   fct(11);
   fct(99);
   fct(100);
   fct(999);
   fct(1000);
}
```

```
Ausgabe
   0 hat 1 Ziffern
   9 hat 1 Ziffern
  10 hat 2 Ziffern
  11 hat 2 Ziffern
  99 hat 2 Ziffern
 100 hat 3 Ziffern
 999 hat 3 Ziffern
1000 hat 4 Ziffern
```

Möglicherweise werden Sie in späteren Aufgaben wieder schön formatierte Ausgaben machen wollen oder sollen – mit „std::log10“ ist das jetzt etwas einfacher geworden.

13.7 Swap-Funktion

In den Aufgaben haben Sie eine Swap-Funktion geschrieben, d.h. eine Funktion, die mit zwei Argumenten gleichen Typs aufgerufen werden kann, und die Inhalte der beiden referenzierten Variablen vertauscht. Damit wir sie nicht immer selbst schreiben müssen, stellt die Standard-Bibliothek im Header <utility> eine solche Funktion „std::swap“ zur Verfügung, die alle movebaren, kopierbaren bzw. zuweisbaren Typen unterstützt.

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

int main()
{
   int n1(4);
   int n2(8);

   cout << n1 << " : " << n2 << '\n';
   swap(n1, n2);
   cout << n1 << " : " << n2 << '\n';

   cout << '\n';

   string s1("Text 1");
   string s2("Text 2");

   cout << s1 << " : " << s2 << '\n';
   swap(s1, s2);
   cout << s1 << " : " << s2 << '\n';
}
```

Ausgabe
```
4 : 8
8 : 4
Text 1 : Text 2
Text 2 : Text 1
```

Achtung – selbst wenn die Standard-Bibliothek eine allgemeine Swap-Funktion zur Verfügung stellt – in manchen Situationen sollte man trotzdem noch eine eigene spezielle Implementierung anbieten.

13.8 Pairs und Tuple

Beim Programmieren haben wir häufiger das Problem, mal schnell zwei, drei oder mehr Elemente (Variablen) gemeinsam handeln zu müssen. Im Normallfall sollten wir dafür Klassen verwenden, die neben dem Zusammenfassen mehrerer Variablen zu einem Typ noch Konstruktoren Destruktoren Zugriffs-Bereichen, Element-Funktionen, und viele weitere Feature mitbringen. Aber manchmal sind Klassen aufwändiger als notwendig, da man einfach nur ganz lokal und kurz z.B. 2 Variablen gemeinsam handeln möchte. Für diese einfachen Situationen gibt es in C++ Pairs und Tuple.

13.8.1 Pairs

Fangen wir mit den Pairs an. Im Standard-Header <utility> gibt es die Klasse „std::pair“, die zwei Variablen zu einem Typ zusammenfasst. Die Typen der Variablen müssen in spitzen Klammern hinter dem Namen „pair“ angegeben werden (analog zu Vektoren oder anderen Containern. Die Variablen können dann über die Pair-Variable mit Punkt-Operator „first“ und „second“ angesprochen werden.

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

int main()
```

```
{
   pair<int, string> pa;
   pa.first = 4;
   pa.second = "Pairs sind manchmal sehr hilfreich";

   cout << pa.first << '\n';
   cout << pa.second << '\n';
}
```

Ausgabe

```
4
Pairs sind manchmal sehr hilfreich
```

Pair-Variablen kann man ohne bzw. mit zwei Argumenten erzeugen. Werden keine Argumente angegeben, so werden die Variablen im Pair mit ihren Default-Werten initialisiert (die dann auch bei z.B. elementaren Datentypen definiert sind. Bei Angabe von zwei Argumenten werden diese zur Initialisierung der Variablen genutzt:

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

int main()
{
   pair<int, double> pa0;                             // <-- kein Argument
   cout << pa0.first << " - " << pa0.second << '\n';

   pair<int, double> pa2(8, 3.14);                    // <-- zwei Argumente
   cout << pa2.first << " - " << pa2.second << '\n';
}
```

Ausgabe

```
0 - 0
8 - 3.14
```

Im Normallfall erzeugen wir Pair-Variablen aber nicht so explizit wie im obigen Beispiel, sondern nutzen die Funktion „std::make_pair“ aus <utility>. Sie erkennt den Typ der Parameter (vergleichbar zu z.B. „auto“ und erzeugt dann automatisch ein entsprechendes Pair-Objekt. Nutzen wir für die Typisierung der Variable dann noch „auto“, dann müssen wir die Typen nicht explizit angeben:

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

int main()
{
   auto pa1 = make_pair(1, 2);
   cout << typeid(pa1).name();
   cout << " => " << pa1.first << " - " << pa1.second << '\n';

   auto pa2 = make_pair(true, 'A');
   cout << typeid(pa2).name();
   cout << " => " << pa2.first << " - " << pa2.second << '\n';

   auto pa3 = make_pair(123L, 3.14);
   cout << typeid(pa3).name();
   cout << " => " << pa3.first << " - " << pa3.second << '\n';
}
```

Mögliche Ausgabe („mögliche“, da die Ausgabe von RTTI nicht exakt definiert ist)

```
std::pair<int,int> => 1 - 2
std::pair<bool,char> => 1 - A
std::pair<long,double> => 123 - 3.14
```

Hinweise:

- Das obige Beispiel benutzt das Feature „RTTI“, das wir nicht kennen lernen werden. Es ermöglicht uns den Typ eines Objekts zu bestimmen und dann z.B. als Text auszugeben – siehe Beispiel. Aber Achtung – die genaue Ausgabe ist nicht definiert.
- Die Funktion „make_pair“ funktioniert natürlich nur mit 2 Argumenten. Es gibt natürlich keine Variante ohne Argumente wie bei der expliziten Konstruktion oben – wie sollte der Compiler auch ohne Argumente die Typen bestimmen?

Achtung: seit C++17 kann der Compiler die Typen auch ohne die „make_pair“ Funktion bestimmen. Dies gilt auch für Tuple – in C++17 ist der Umweg über die „make_tuple“ Funktion nicht mehr notwendig.

Ein alternativer Zugriff auf die Variablen kann mit „std::get<idx>()“ geschehen, wobei „idx“ der Index in das Pair ist – also nur die Werte „0“ und „1“ annehmen darf. Während dies bei Tuples (siehe nächstes Kapitel) die normale Zugriffs-Methode ist, wird dies bei Pairs eigentlich nur bei generischem Code eingesetzt.

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

int main()
{
   auto pa = make_pair(2, 3);
   cout << get<0>(pa) << " - " << get<1>(pa) << '\n';
}
```

```
Ausgabe
2 - 3
```

Eine Anwendung von Pairs haben wir indirekt schon kennengelernt. Im Container-Kapitel wurden Maps eingeführt, die ein assoziativer Container sind (ein Container für Schlüssel/Wert-Paare) und sich bzgl. der Schlüssel wie ein Set verhalten. Der Zugriff auf den Schlüssel und den Wert mußte dort mit den Namen „first“ und „second“ erfolgen – wie beim Pair. Eine Map ist intern nämlich nichts anderes als ein Set von Pairs. Darum funktioniert die „make_pair“ Funktion auch beim Einfügen von Elementen in eine Map:

```
#include <iostream>
#include <map>
#include <typeinfo>
#include <utility>
using namespace std;

int main()
{
   map<int, int> m;
   m.insert(map<int, int>::value_type(1, 11));      // <-- Bisheriger Code - C++03
   m.insert({ 2, 22 });                              // <-- Bisheriger Code - C++11 (*)
   m.insert(make_pair(3, 33));                       // <-- "make_pair" funktioniert auch
   for (auto& pa : m)
   {
      cout << typeid(pa).name() << ": " << pa.first << " => " << pa.second << '\n';
   }
}
```

```
Mögliche Ausgabe („mögliche“, da die Ausgabe von RTTI nicht exakt definiert ist)
std::pair<int const ,int>: 1 => 11
std::pair<int const ,int>: 2 => 22
std::pair<int const ,int>: 3 => 33
```

Eine weitere nette Anwendung von Pairs (und auch Tuplen) ist die Rückgabe mehrerer Werte aus einer Funktion. In C++ gibt eine Funktion immer entweder nichts („void“) oder einen Wert zurück – mittels Pairs oder Tuplen werden daraus mehrere Werte. Im folgenden Beispiel gibt die Funktion „minmax_values“ sowohl das Minimum als auch das Maximum von 3 Integer-Zahlen zurück:

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

pair<int, int> minmax_values(int v1, int v2, int v3)
{
   int min = v1 < v2 ? v1 : v2;
   min = min < v3 ? min : v3;

   int max = v1 > v2 ? v1 : v2;
   max = max > v3 ? max : v3;

   return pair<int, int>(min, max);
}

int main()
{
   pair<int, int> mm = minmax_values(1, 3, 5);
   cout << "Minimum: " << mm.first << '\n';
   cout << "Maximum: " << mm.second << '\n';
}
```

```
Ausgabe
Minimum: 1
Maximum: 5
```

Will man die Pair-Rückgabe direkt vorhandenen Variablen zuweisen, so kann man dies mit „std::tie“ machen. Will man dabei einzelne Rückgabe-Werte ignorieren, so kann man diese mit „std::ignore“ ausblenden. Seit C++17 geht dies auch mit der Structured Binding Declartion direkt – siehe Kapitel über Variablen-Initialisierungen.

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

pair<int, int> transfer_to_pair(int x, int y)
{
   return pair<int, int>(x, y);
}

int main()
{
   int a, b;

   tie(a, b) = transfer_to_pair(11, 12);                // <-- std::tie
   cout << "Tie(a,b): " << a << " - " << b << '\n';

   tie(a, ignore) = transfer_to_pair(13, 14);           // <-- std::tie + std::ignore
   cout << "Tie(a,-): " << a << " - " << b << '\n';

   tie(ignore, b) = transfer_to_pair(15, 16);           // <-- std::tie + std::ignore
   cout << "Tie(-,b): " << a << " - " << b << '\n';

   auto [x, y] = transfer_to_pair(17, 18);              // Structured Binding Declaration
C++17
   cout << "Auto:     " << x << " - " << y << '\n';
}
```

```
Ausgabe
Tie(a,b): 11 - 12
Tie(a,-): 13 - 12
```

```
Tie(-,b): 13 - 16
Auto:     17 - 18
```

13.8.2 Tuple

So schön einfach und hilfreich Pairs auch sind – sie haben einen entscheidenen Nachteil: sie sind auf 2 Attribute beschränkt. Manchmal braucht man aber 3, 4 oder noch mehr Attribute für eine einfache Lösung. Und hier kommen Tuple ins Spiel. Sie sind quasi verallgemeinerte Pairs für eine beliebige Anzahl von Attributen. Währung Pairs von Anfang an in C++ vorhanden waren (schon im C++98 Standard), wurden Tuple in C++ erst mit C++11 in den Standard eingeführt.

Tuple (Header <tuple>) definieren Sie einfach mit den Typen in spitzen Klammern hinter dem Typ-Namen „tuple" – die Anzahl an Typen (d.h. die Anzahl an Attributen) ist frei wählbar. Wie bei Pair gibt es die Erzeugung ohne Argumente (alle Attribute werden mit dem Default-Wert initialisiert) und mit sovielen Argumenten wie das Tuple Attribute hat. Und auch Tuple haben eine zu „std::make_pair" analoge Funktion „std::make_taple". Nur der Zugriff auf die Attribute der Tuple funktioniert nur mit der Funktion „std::get<>()" – es gibt keine festen Namen:

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

int main()
{
   // Tuple verschiedener Groesse
   tuple<> tu0;                                // Groesse 0 - hat nur akademischen Wert
   tuple<int> tu1;                             // Groesse 1 - wie eine einzelne Int-Variable
   tuple<int, int> tu2;                        // Groesse 2 - koennte man auch pair<> nehmen
   tuple<int, int, int> tu3;                   // Groesse 3 - jetzt wird es interessant
   tuple<int, int, int, int> tu4;              // usw.
   tuple<int, int, int, int, int> tu5;         // ...

   // Konstruktion von Tuplen
   tuple<int, int, int> t0;
   tuple<int, int, int> t3(1, 2, 3);

   // Zugriff auf die Attribute von Tuplen nur mit "std::get<>()"
   cout << get<0>(t0) << " - " << get<1>(t0) <<  " - " << get<2>(t0) << '\n';
   cout << get<0>(t3) << " - " << get<1>(t3) <<  " - " << get<2>(t3) << '\n';

   cout << '\n';

   // Die Funktion "std::make_tuple" mit "auto" macht das Leben wiedermal leichter
   auto t = make_tuple(11, 22L, 33.3);
   cout << typeid(t).name() << '\n';
}
```

Mögliche Ausgabe („mögliche", da die Ausgabe von RTTI nicht exakt definiert ist)

```
0 - 0 - 0
1 - 2 - 3

std::tuple<int,long,double>
```

Hinweis – die Funktion „make_tuple" hat natürlich die gleiche Falle bzgl. Zeichenketten-Konstanten und C-Arrays wie „make_pair".

Und auch Tuple unterstützen wie Pairs „std::tie" und „std::ignore", oder auch die Structured-Binding-Declaration aus dem Kapitel über Variablen-Initialisierungen.

© Detlef Wilkening 2023

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

tuple<int, int, int, int, int> calc_basic_arithmetic_operations(int x, int y)
{
   return make_tuple(x+y, x-y, x*y, x/y, x%y);
}

int main()
{
   int a, s, m, d, o;

   tie(a, s, m, d, o) = calc_basic_arithmetic_operations(5, 2);
   cout << "add: " << a << '\n';
   cout << "sub: " << s << '\n';
   cout << "mul: " << m << '\n';
   cout << "div: " << d << '\n';
   cout << "mod: " << o << '\n';

   cout << '\n';

   tie(ignore, ignore, ignore, d, o) = calc_basic_arithmetic_operations(11, 4);
   cout << "11/4 = " << d << '\n';
   cout << "11%4 = " << o << '\n';

   auto [a2, s2, m2, d2, o2] = calc_basic_arithmetic_operations(11, 4);
   cout << "11/4 = " << d2 << '\n';
   cout << "11%4 = " << o2 << '\n';
}
```

Ausgabe

```
add: 7
sub: 3
mul: 10
div: 2
mod: 1

11/4 = 2
11%4 = 3
11/4 = 2
11%4 = 3
```

Wie man sieht, ist ein Tuple wirklich nur ein verallgemeinertes Pair. Ein Tuple ist genauso einfach zu nutzen wie ein Pair. Aus heutiger Sicht (C++11) benötigt man eigentlich keine Pairs mehr, sondern nutzt nur noch Tuple.