Vorlesung

Objektorientiertes Programmieren in C++

Teil 3 - WS 2025/26

Detlef Wilkening
www.wilkening-online.de

8 Ein- und Ausgabe

Die Eingabe von Tastatur und die Ausgabe auf die Kommandozeile werden in C++ mit Streams gemacht.

Ein Stream (Strom) ist einfach ein Strom von Zeichen, wobei auf der einen Seite Zeichen hineingestellt werden können, und auf der anderen Seite können Zeichen herausgeholt werden. Außerdem können Streams manipuliert werden. Je nach Stream-Typ kann ein Stream auch statt Zeichen z.B. Bytes oder Objekte oder oder oder enthalten.

8.1 Ausgabe

Headerdatei: **iostream**
Standardausgabestrom: **std::ostream std::cout**

- verbunden mit stdout - typischerweise der Bildschirm

Für die Ausgabe werden die Objekte mit dem Ausgabe-Operator << einfach in den Stream hineingeschoben - die Ausgaben können dabei verkettet werden.

```
#include <iostream>
#include <string>

int main()
{
   int i = 9;
   std::cout << i;                          // Einfache Ausgabe der Variable 'i'

   char c = 'H';
   std::string s("allo Welt");
   double d = 3.14;

   std::cout << ' ' << c << s               // Verkettete Ausgaben, verteilt auf zwei Zeilen
             << " juhu\n" << d;
}
```

Ausgabe
```
9 Hallo Welt juhu
3.14
```

Hinweis – Ausgaben mit dem Ausgabe-Operator sind immer textuelle Ausgaben, d.h. Ausgaben die ein Benutzer lesen kann. Wenn Sie später eigene Ausgabe-Operatoren programmieren, sollten Sie sich an diese Konvention halten.

8.1.1 Fehlschläge

Natürlich kann eine Ausgabe auch fehlschlagen. Sehr wahrscheinlich ist dies, wenn der Stream nicht mit der Console, sondern z.B. mit einer Datei verbunden ist, da dann das Medium (Diskette, Festplatte, USB-Stick) voll sein könnte, die Verbindung unterbrochen sein könnte, oder oder oder.

Im Falle eines Fehlers geht der Ausgabe-Stream in den Status *FAIL* („fehlerhaft“) – dieser Status kann mit der Element-Funktion“fail“ abgefragt werden.

```
// Hinweis - out ist hier ein beliebiger Ausgabe-Stream,
// der z.B. mit einer Datei verbunden sein koennte

out << 42;
if (out.fail())
{
   cout << "Schreiben ist fehlgeschlagen.\n";
}
```

Hinweis - um den Zustand eines Streams wieder auf *GOOD (*„fehlerfrei“) zu setzen, muss die Element-Funktion ‘clear()’ aufgerufen werden. Achtung – solange ein Stream im Status *FAIL* ist, haben Funktionen oder Manipulationen am Stream **keine Auswirkungen**. Es ist ein typischer Anfängerfehler den Aufruf von „clear“ zu vergessen.

```
// Hinweis - output ist hier ein beliebiger Ausgabe-Stream,
// der z.B. mit einer Datei verbunden sein koennte

output << 42;
if (output.fail())
{
   output.clear();              // Ohne "clear" geht nichts mehr mit "output"
   cout << "Fehler";
}
```

8.1.2 Manipulatoren

Zusätzlich können sogenannte Manipulatoren in den Ausgabestrom geschoben werden – mit ihnen kann man z.B. die Ausgabe formatieren. Die Standard-Bibliothek stellt viele Manipulatoren zur Verfügung, von denen wir nur wenige besprechen werden.

std::boolapha	Beeinflusst die Ausgabe von Bool-Werten
std::noboolalpha	Beeinflusst die Ausgabe von Bool-Werten
std:flush	Leert den Ausgabe-Puffer
std::endl	Gibt einen Zeilenumbruch und std::flush aus

Besprechen wir erstmal die ersten beiden. In der Programmiersprache C, auf der C++ beruht, gab es keinen boolschen Daten-Typ. In C ist der Wert „0“ aller Typen mit „false“ assoziiert, und alle Werte ungleich „0“ mit „true“. Umgekehrt wird „false“ mit der „0“ und „true“ mit der „1“ repräsentiert. Diese grundsätzliche Philosophie hat C++ von C geerbt. Das spiegelt sich u.a. darin wider, dass die Ausgabe von Bool-Werten im Normallfall Nullen und Einsen ergibt.

```
#include <iostream>
```

```
using namespace std;

int main()
{
   bool bf = false;
   bool bt = true;
   cout << "false: " << bf << " - " << false << '\n';
   cout << "true:  " << bt << " - " << true << '\n';
}
```

```
Ausgabe
false: 0 - 0
true:  1 - 1
```

Die textuelle Darstellung von boolschen Werten mit „0“ oder „1“ ist für Einsteiger oft sehr verwirrend und nicht gut lesbar – d.h. kann man das mit dem Manipulator „std::boolalpha“ umstellen. Der Manipulator ändert die Ausgabe für den betroffenen Stream auf „false“ und „true“. Mit dem Manipulator „std::noboolalpha“ kann das Verhalten wieder auf „0“ und „1“ zurückgesetzt werden.

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

int main()
{
   bool bf = false;
   bool bt = true;

   cout << "false: " << bf << " - " << false << '\n';
   cout << "true:  " << bt << " - " << true << '\n';

   cout << boolalpha;

   cout << "false: " << bf << " - " << false << '\n';
   cout << "true:  " << bt << " - " << true << '\n';

   cout << noboolalpha;

   cout << "false: " << bf << " - " << false << '\n';
   cout << "true:  " << bt << " - " << true << '\n';
}
```

```
Ausgabe
false: 0 - 0
true:  1 - 1
false: false - false
true:  true - true
false: 0 - 0
true:  1 - 1
```

Ausgaben werden in allen Programmiersprachen an vielen Stellen aus Performance-Gründen gepuffert. Daher statt, dass jede Ausgabe direkt ausgeführt wird, werden die Ausgaben gesammelt und dann in einem ausgeführt. Gerade auf Netzwerken oder im Dateisystem bringt dies einen großen Performancegewinn – bei „cout“ ist dieser eher minimal. Möchten Sie, dass eine Ausgabe auf jeden Fall geschrieben ist, so müssen Sie den Ausgabe-Puffer schreiben lassen – man nennt dies „flushen“. Dazu schiebt man den Manipulator „std::flush“ in den Stream – siehe folgendes Programm. Der Manipulator „std::endl“ gibt zusätzlich vor dem Flushen noch einen Zeilenumbruch aus.

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

int main()
{
```

```
   cout << "Hallo" << flush << " Welt" << endl;
   cout << "C++" << flush << " ist" << flush << " interessant" << endl;
}
```

```
Ausgabe
Hallo Welt
C++ ist interessant
```

Achtung

- Sie sehen keinen Unterschied mit und ohne den Manipulator „flush“. Das Programm wird nur etwas langamer.
- Nutzen Sie die Manipulatoren „flush“ und gerade „endl“ nur, wenn Sie das Leeren des Ausgabe-Puffers wirklich benötigen. Ihr Programm wird nur langsamer durch diese.
- Bevorzugen Sie deshalb ‚\n‘ vor dem Manipulator „endl“ für einen Zeilenumbruch.

8.1.3 Formatierungen

Mit C++98 wurden Manipulatoren und Element-Funktionen auf den Streams zur Formatierung der Ausgabe eingeführt. Diese führen wir kurz in Kapitel 8.1.3.1 ein. Leider hat diese Art der Formulierung einige Nachteile:

- Sie ist relativ aufwändig zu schreiben (viel zu schreiben)
- Sie lässt sich nicht gut auf eigene Typen übertragen (während das mit der Ausgabe selber gut geht – siehe Kapitel 16.4)
- Diese Formatierungen lassen sich nicht nur sehr umständlich internationalisieren

Darum wurde mit C++20 eine neue Formatierungs-Bibiothek in die C++ Standard-Bibliothek übernommen, die all diese Probleme lösen soll und einfach schöner zu benutzen ist. Diese neue Format-Library wird kurz in Kapitel 8.1.3.2 vorgestellt.

8.1.3.1 C++98

Ausgabe-Streams bieten eine Menge von Möglichkeiten zur Formatierung an. Zwei wichtige Formatierungen seien hier kurz vorgestellt:

Ausgabebreite
Die Ausgabebreite kann **für die nächste Ausgabe** mit dem Manipulator **std::setw(int)** eingestellt werden - übergeben wird die Breite in Zeichen als int. Dieser Manipulator ist im Header <iomanip> definiert.

Füllzeichen
Das Default-Füllzeichen für Ausgaben, die kleiner als die eingestellte Ausgabebreite sind, ist das Leerzeichen. Das Füllzeichen kann mit der Element-Funktion „**char std::cout.fill(char)**“ gesetzt werden - die Element-Funktion gibt dabei das alte Füllzeichen zurück.

```
#include <iostream>
#include <iomanip>

int main()
{
   std::cout << 4 << '\n';
   std::cout << std::setw(2) << 5 << '\n';
   std::cout << std::setw(3) << 6 << 7 << '\n';
```

```
    char c = std::cout.fill('x');
    std::cout << std::setw(9) << "Hallo" << '\n';
    std::cout.fill('_');
    std::cout << std::setw(3) << 14 << std::setw(5) << 15 << '\n';
    std::cout.fill(c);
}
```

```
Ausgabe
4
 5
  67
xxxxHallo
_14___15
```

8.1.3.2 C++20

Die C++20 Format-Lib geht einen ganz anderen Weg. Sie besteht primär aus einer Format-Funktion „std::format“ aus dem Header „format“, die einen die Ausgabe beschreibenen Text und die Werte bekommt. Den die Ausgabe beschreibenen Text nennt man auch den Formatierungs-String. Die Funktion liefert dann den formatierten Text als „std::string“ zurück. Man kann ihn daher speichern oder direkt auf z.B. „cout“ ausgeben. Die direkte Ausgabe kann in C++23 auch direkt mit den neuen Print-Funktionen passieren – siehe Kapitel 4.4.

Das Ganze klingt komplizierter als es ist. Beginnen wir mit einem einfachen Beispiel:

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

int main()
{
    cout << format("Hallo C++20 Format-Lib\n");                // (*)

    cout << format("Zahl = {}\n", 42);                         // (**)

    int a = 11;
    int b = 12;
    cout << format("a = {} und b = {}\n", a, b);               // (***)

    string s = format("{}, {} und {}", 3.14, true, "xyz");     // (****)
    cout << s << '\n';
}
```

```
Ausgabe
Hallo C++20 Format-Lib
Zahl = 42
a = 11 und b = 12
3.14, true und xyz
```

Erklärungen:

- In Zeile (*)bekommt die Format-Funktion nur den Formatierungs-String, der dann 1:1 ausgegeben wird.
- In Zeile (**) enthält der Formatierungs-String einen Platzhalter für Argumente. Der Platzhalter besteht aus den geschweiften Klammern „{}“. Hier wird das zusätzlich übergebene Argument eingefügt.
- In der Zeile (***) enthält der Formatierungs-String zwei Platzhalter, bei denen die Argumente Variablen sind. werden an diese Stelle eingefügt –
- In Zeile (****) enthält der Formatierungs-String drei Platzhalter, und die Argumente haben unterschiedliche Typen. Ein „bool“ wird direkt als „false“ oder „true“ ausgegeben.

Außerdem wird das Ergebnis der Formatierung in einem „string“ gespeichert, und erst in der Zeile danach ausgegeben.

Ich hoffe, Sie haben die Grundidee verstanden. Der Formatierungs-String ist der gesamte Text, und an den Stellen der Platzhalter werden die weiteren Argumente eingefügt. Hierbei wird, soweit möglich, zur Compile-Zeit geprüft, ob die Platzhalter und die weiteren Argumente matchen. Ansonsten passiert die Überprüfung zur Laufzeit.

```
format("", 1, 2);          // Okay, '1' und '2' werden ignoriert
format("{}");              // Compile-Fehler - kein Argument fuer Platzhalter
format("{}, {}", 1);       // Compile-Fehler - kein Argument fuer Platzhalter
```

Die Platzhalter können Indices enthalten, die die Argumente 0-basiert indizieren. Dadurch kann man die Reihenfolge der Argumente verändern, oder Argumente in einem Text mehrfach ausgeben.

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

int main()
{
   cout << format("{0}, {1}, {3}, {2}, {1}, {0}\n", 0, 1, 22, 333);
}
```

Ausgabe

```
0, 1, 333, 22, 1, 0
```

Außerdem können in den Platzhaltern nach einem Doppelpunkt Zahlen für die Ausgabebreite angegeben werden – kombinierbar mit den Indices. Die Default-Ausrichtung für Zahlen ist dabei dann rechtsbündig – siehe Beispiel.

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

int main()
{
   cout << format("Zahlen:{:3}_{:3}|\n", 0, 1);
   cout << format("Zahlen:{1:3}_{0:3}|\n", 0, 1);
}
```

Ausgabe

```
Zahlen:  0_  1|
Zahlen:  1_  0|
```

Vor der Ausgabeweite kann man mit einem Zeichen die Ausrichtung steuern:

- „<“ – linksbündig
- „^“ – zentriert
- „>“ – rechtsbündig

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

int main()
{
   cout << format("Zahlen:{:<3}x{:^3}x{:>3}x\n", 1, 2, 3);
   cout << format("Zahlen:{2:<3}x{1:^3}x{0:>3}x\n", 1, 2, 3);
}
```

Ausgabe

```
Zahlen:1   x 2 x   3x
Zahlen:3   x 2 x   1x
```

Wird vor dem Ausrichtungszeichen ein weiteres Zeichen angebeben, so wird dieses als Füllzeichen verwendet.

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

int main()
{
   cout << format("{:*<4},{:+^4},{:X>4}\n", "ab", "cd", "ef");
   cout << format("{2:*<4},{1:+^4},{0:X>4}\n", "ab", "cd", "ef");
}
```

```
Ausgabe
ab**,+cd+,XXef
ef**,+cd+,XXab
```

Ich hoffe, Sie sehen, dass Formatierungen mit der Format-Lib von C++20 viel einfacher sind.

Hinweis: in C++23 kann man die Ausgabe alternativ mit den Print-Funktionen ausführen – siehe Kapitel 4.4. Die Print-Funktionen von C++23 unterstützten die Format-Funktionalitäten 1:1. Man muss nur nicht den Umweg über einen String gehen.

8.2 Eingabe

Headerdatei: **iostream**
Standardeingabestrom: **std::istream std::cin**

- verbunden mit stdin - typischerweise der Tastatur

Bei der Eingabe werden die Objekte mit dem Eingabe-Operator >> einfach aus den Stream heraus in die Variablen hineingeschoben. Auch die Eingaben können verkettet werden – dies ist aufgrund der notwendigen Fehlerbehandlung aber nicht empfehlenswert, s.u.

Whitespaces (Leerzeichen, Tabs, Zeilenumbrüche) trennen die Elemente der Eingabe. Ansonsten werden so viele Zeichen gelesen, wie sinnvoll verarbeitet werden können, d.h. zu dem einzulesenden Typ passen.

```
// Bitte geben Sie korrekte Werte fuer int, char und double ein - s.u.

#include <iostream>

int main()
{
   int i;
   char c;
   double d;

   std::cout << "Eingabe int, char, double: ";
   std::cin >> i;
   std::cin >> c >> d;                          // nicht empfehlenswert - s.u.

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

```
Mögliche Ein- bzw. Ausgabe
Eingabe int, char, double: 42X  3.1415
i: 42
c: X
d: 3.1415
```

Hinweis – das Einlesen von Texten bzw. ganzer Zeilen wird im Kapitel 9.1.3 über Strings behandelt.

8.2.1 Fehlschläge

Einlesen kann eigentlich immer fehlschlagen - im einfachsten Fall auf Grund einer falschen Eingabe des Benutzers. Sie sollten sich daher nach jeder Eingabe vergewissern, ob die Eingabe geklappt hat, und ansonsten angemessen reagieren.

Schlägt die Eingabe fehl, so wird der Eingabe-Stream auf *FAIL* gesetzt. Dieser Zustand kann z.B. mit der Element-Funktion ‘fail()’, die einen „bool“ Wert zurückgibt, abgefragt werden.

```
// Beispiel fuer den Fehlerzustand von Streams nach einer fehlerhaften Eingabe
// Abfrage mit der Element-Funktion fail()
//
// Bitte geben Sie Zeichen statt Zahlen ein.

#include <iostream>

int main()
{
   int i;
   std::cout << "Eingabe: ";
   std::cin >> i;
   if (std::cin.fail())
   {
      std::cout << "Fehler bei Eingabe\n";
      return 0;
   }
   std::cout << "Eingabe von: " << i << '\n';
}
```

```
Mögliche Ein- bzw. Ausgabe
Eingabe: 42
Eingabe von: 42
```

```
Mögliche Ein- bzw. Ausgabe
Eingabe: X
Fehler bei Eingabe
```

Genau genommen bedeutet der Status *FAIL*, dass **nichts** gelesen werden konnte – aus welchen Gründen auch immer.

Um den Zustand eines Eingabestreams wieder auf *GOOD* zu setzen, kann die Element-Funktion ‘clear()’ aufgerufen werden. Achtung – solange ein Stream im Status *FAIL* ist, haben Eingaben oder Manipulationen am Stream **keine Auswirkungen**.

```
cin.clear();
```

8.2.2 Stream leeren

Wenn Sie im obigen Beispiel statt einer Ziffer ein beliebiges anderes Zeichen eingeben

schlägt das Einlesen fehl, da das Zeichen nicht in den angegebenen Integer konvertiert werden kann. Das ist verständlich und offensichtlich. Nicht so offensichtlich ist, dass die falsche Eingabe dann aber immer noch im Eingabe-Stream steht, da sie nicht ausgelesen werden konnte.

Da die fehlerhaften Eingaben noch im Stream stehen, wird man im Fall einer fehlerhaften Eingabe häufig den Stream leeren müssen. Hierzu gibt es die Element-Funktion ‚ignore' die eine anzugebene Anzahl an Zeichen im Stream bis zu einem frei definierbaren Endezeichen ignoriert (d.h. diese aus dem Stream entfernt).

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

int main()
{
   int i;
   for (;;)
   {
      std::cout << "Bitte geben Sie eine Zahl ein: ";
      std::cin >> i;
      if (!std::cin.fail())
      {
         break;
      }
      std::cout << "- Fehlerhafte Eingabe - keine Zahl\n";
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');    // (*)
   }
   std::cout << "Eingabe: " << i << '\n';
}
```

```
Mögliche Ein- bzw. Ausgabe
Bitte geben Sie eine Zahl ein: X
- Fehlerhafte Eingabe - keine Zahl
Bitte geben Sie eine Zahl ein: 7
Eingabe: 7
```

Die genaue Bedeutung der Zeile (*) im Beispiel liegt zurzeit noch außerhalb unseres Wissens. Übernehmen Sie die Zeile einfach so, wenn Sie den Eingabe-Puffer leeren wollen. Hier soll uns das Wissen reichen, dass mit dieser Anweisung die „max. mögliche Anzahl an Zeichen im Stream" aus dem Stream gelöscht wird – bis zum ‚\n' inklusive. D.h. die fehlerhafte Zeile wird komplett entfernt – und das wollen wir ja.

8.2.3 Programm-Fluß und -Kontrolle

Die Benutzung von Eingabe-Streams ist nicht ganz problemlos, da das Programm beliebige Eingaben sinnvoll verarbeiten können muss, und außerdem Whitespaces auch bei Texten als Trennzeichen benutzt werden.

Außerdem verliert das Programm während der Eingabe die Kontrolle über den Programmfluß, und diese kehrt frühestens mit Eingabe von „Return" in der Console zum Programm zurück.

Nach Eingabe von „Return" versucht C++ die im Eingabe-Stream stehenden Zeichen auf die entsprechenden Eingabe-Typen zu streamen. Stehen nicht genügend Zeichen für alle

angeforderten Typen im Eingabe-Stream, so kehrt die Kontrolle über den Programm-Fluß **nicht** an das Programm zurück!

Geben Sie z.B. beim folgenden Programm, das zwei Int-Zahlen erwartet, nur eine Zahl ein, und danach „Return“. Die Kontrolle kehrt nicht zu Ihnen zurück. Genau genommen kehrte sie natürlich zu Ihnen zurück, aber da sie ja (fehlerhafter Weise) zwei Eingaben verkettet haben „cin >> i1 >> i2“, können Sie dazwischen nicht reagieren. Ein weiteres „Return“ *verpufft* auch wirkungslos, und erst die Eingabe einer Zahl mit abschliessendem „Return“ bringt die Kontrolle zu Ihnen zurück.

```
// Achtung - fehlerhaftes Programm:
// - verkettete Eingaben
// - fehlende Fehlerabfrage bei der Eingabe

int i1, i2;
cout << "Bitte Eingabe zweier Int-Zahlen: ";
cin >> i1 >> i2;
cout << "i1: " << i1 << "\ni2: " << i2 << '\n';
```

```
Mögliche Ein- bzw. Ausgabe
Bitte Eingabe zweier Int-Zahlen: 34↵
↵
67↵
i1: 34
i2: 67
```

Besonders überraschend ist der Effekt, wenn Sie einen Zeichen (d.h. Typ „char“) einlesen wollen. Diese Eingabe kann ja eigentlich keine Probleme machen, da sich ja jedes eingegebene Zeichen in einen „char“ streamen läßt, und ja auch nur ein Zeichen notwendig ist, um diese Eingabe zu erfüllen. Aber auch hier gibt es eine Tücke im Detail: leere Eingaben, d.h. nur „Returns“ lassen die Kontrolle wieder nicht zu ihnen zurückkehren.

```
char c;
cout << "Bitte Eingabe eines Zeichens: ";
cin >> c;
cout << "Eingegeben wurde '" << c << '\'';
```

```
Mögliche Ein- bzw. Ausgabe
Bitte Eingabe eines Zeichens: ↵
↵
↵↵
↵
x↵
Eingegeben wurde 'x'
```

Es gibt in ISO C++ keine Möglichkeit die Tastatur komfortabler abzufragen – hierzu müssen Sie compiler- bzw. betriebssystem-spezifische Funktionen benutzen, aber wirklich einfacher wird der Eingabe-Code dadurch auch nicht – meist hat man nur mehr Möglichkeiten.

Wie heißt es doch so schön: „Fehlertolerantes Einlesen ist einfach der pure Spaß!“.

8.2.4 *FAIL* Status selber setzen

In erster Linie wird der *FAIL* Status vom Stream selber gesetzt, wenn Eingaben

fehlgeschlagen sind. Bei komplexeren Eingaben kann es aber auch Sinn machen, selber den FAIL Status zu setzen, um dem Aufrufer das Fehlschlagen der Eingabe mitzuteilen. In diesem Fall muss die Element-Funktion ‚setstate' mit dem entsprechenden Flag aufgerufen werden.

```
std::cin.setstate(std::ios::failbit);
```

Hier ein kleines Beispiel:

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

int main()
{
   cout << "Fail-Status: " << (cin.fail() ? "true" : "false") << '\n';
   cin.setstate(ios::failbit);
   cout << "Fail-Status: " << (cin.fail() ? "true" : "false") << '\n';
   cin.clear();
   cout << "Fail-Status: " << (cin.fail() ? "true" : "false") << '\n';
}
```

Ausgabe

```
Fail-Status: false
Fail-Status: true
Fail-Status: false
```

9 Texte

Genau wie es in C++ mehrere Zeichen-Typen (char, wchar_t, char8_t, char16_t und char32_t) gibt (siehe Kapitel 5.1.1), so gibt es in C++ auch mehrere String-Typen für Texte, die auf den jeweiligen Zeichen-Typen basieren.

- std::string – char basierter String-Typ
- std::wstring – wchar_t basierter String-Typ
- std::u8string – char8_t basierter String-Typ
- std::u16string – char16_t basierter String-Typ
- std::u32string – char32_t basierter String-Typ

Da wir uns in der Vorlesung mit C++ und nicht mit Details von Zeichensätzen, Kodierungen und Internationalisierung beschäftigen wollen (alles andere würde unseren zeitlichen Rahmen sprengen) – werden wir nur den „std::string" verwenden. In der Praxis sollten Sie Unicode als Zeichensatz verwenden und daher die Unicode Zeichen-Typen (char8_t, char16_t & char32_t) und Strings (u8string, u16string, u32string).

Hinweis: seit C++17 gibt es zusätzlich noch 5 PMR Strings, bei denen man einfacher in die interne Speicherverwaltung eingreifen kann.

9.1 Strings

Für Strings (Texte) gibt es in C++ mehrere Typen (genau genommen Klassen) in der Standard-Bibliothek. Wir benutzen in der Vorlesung zur Vereinfachung nur den einfachen String-Typ:

- Typ: **std::string**
- Headerdatei: **string**

Um String-Objekte zu erzeugen, definiert die Klasse mehrere Konstruktoren – folgende drei sind die Gebräuchlichsten:

Deklaration	Beispiel	Ergebnis
string()	string s;	leerer String => ""
string(const char*)	string s("C++");	String mit der übergebener Zeichenkette => "C++"
string(string::size_type n, char c)	string s(5, 'A');	String mit „n mal das Zeichen „c“ Beispiel 5 x das A => "AAAAA"

Hier ein Beispiel-Programm, indem alle 3 Konstruktions-Arten vorkommen:

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

int main()
{
   string s("Hallo Kurs");
   cout << s << '\n';

   string s1;
   string s2("C++");
   string s3(5, 'A');

   cout << '"' << s1 << "\"\n\""
        << s2 << "\"\n\""
        << s3 << '\n';
}
```

Ausgabe

```
Hallo Kurs
""
"C++"
"AAAAA"
```

Achtung – hier nochmal der Hinweis. Zeichenkettenkonstanten sind keine Strings. Sie sind „const char Arrays“ einer Größe, die von der jeweiligen Zeichenkettenkonstante abhängt. Werden sie mit „auto“ benutzt, so wird der Typ als „const char*“ deduziert. Dies wollen wir hier nicht im Detail besprechen, da es den Rahmen der Vorlesung sprengt.

Mit C++14 wurde der Postfix „s“ eingeführt, der aus der Bibliothek kommt (daher aus dem Namensraum „std“), mit dem eine Zeichenkettenkonstante zu einem String wird.

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

int main()
{
   auto s1 = "Hallo ";         // Dies ist KEIN string
   auto s2 = "Welt"s;          // Dies ist EIN string
}
```

9.1.1 Der Typ „std::string::size_type“

Eine interessante und ganz typische C++ Lösung ist hierbei der Typ „std::string::size_type“ in dem einen Konstruktor von std::string:

```
std::string(std::string::size_type, char)
```

Dieser Parameter gibt an, wie oft das folgende Zeichen im String vorkommen soll.

Wahrscheinlich hätten Sie hier als Typ einen „int“ erwartet, oder einen anderen elementaren integralen Datentyp. Aber eine solche Definition wäre zu starr. Bedenken Sie, dass die genaue Größe der elementaren Datentypen in C++ (wie auch in C) nicht exakt festgelegt ist, sondern von der Plattform abhängt – der Standard legt hier nur Rahmenbedingungen fest.

Genauso legt der Standard nicht fest, welches die max. Länge eines Strings auf einer Plattform ist, da auch hier plattform-spezifische Größen eingehen.

Was man also braucht, wäre idealerweise ein elementarer vorzeichenloser integraler Datentyp, der groß genug ist, die max. Länge der jeweiligen String-Implementierung aufzunehmen, andererseits aber nicht größer als nötig ist (da dies Performance und Speicherplatz kosten würde).

Diese Anforderung kann vom Standard nicht allgemein auf einen konkreten elementaren Datentyp abgebildet werden – dazu können die konkreten Umsetzungen der integralen Typen und der String-Klasse viel zu unterschiedlich sein. Der Standard behilft sich hier damit einen Typ-Alias vorzugeben, den die konkrete Implementierung dann mit einem echten Typen belegen muss. Auf die Länge von Strings bezogen ist das eben: „std::string::size_type“.

Der Standard schreibt im Falle von „std::string::size_type“ nur vor:
1. Das es diesen Typ-Alias geben muss.
2. Das er groß genug ist, die max. String-Länge aufzunehmen.
3. Und das er auf einen elementaren integralen vorzeichenlosen Datentyp abgebildet werden muss.

Der Rest ist der Implementierung überlassen.

Achtung – nutzen Sie diese Typen, wenn Sie problemlosen und portablen Code schreiben wollen. Nur weil Ihr Compiler auf ihrer Plattform mit Ihrer Bibliothek hier vielleicht einen „unsigned short“ oder „unsigned int“ oder sonstwas verwendet, heißt das ja nicht, dass das auch andere Compiler so machen, oder andere Bibliotheks-Implementierungen, oder Ihre nächste Compiler-Version, oder wenn Sie auf eine 64 Bit Plattform umsteigen, oder, oder...

9.1.2 Länge

Die Länge eines String liefert die Element-Funktion „length()“ zurück – der Rückgabe-Typ ist natürlich „std::string::size_type“.


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

int main()
{
   string s("C++");
   string::size_type len = s.length();
   cout << len;                                // -> 3       (*)
}
```

Ausgabe
```
3
```

Hier haben wir übrigens ein Beispiel, bei der die Nutzung der automatischen Typ-Deduktion bei Variablen-Definition mit Hilfe von „auto“ in C++11 hilfreich ist – denn „std::string::size_type“ ist zwar kein komplexer aber doch schon recht langer Typ.

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

int main()
{
   string s;
   for (int i=0; i<4; ++i)
   {
      s += 'x';
      string::size_type len1 = s.length();                    // Mit Typ
      auto              len2 = s.length();                    // Mit "auto"
      cout << s << " -> " << len1 << " - " << len2 << '\n';
   }
}
```

Ausgabe
```
x -> 1 - 1
xx -> 2 - 2
xxx -> 3 - 3
xxxx -> 4 - 4
```

9.1.3 Ein- und Ausgabe

Strings können auf Streams ausgegeben und von Streams eingelesen werden. Achtung – bei Eingaben mit dem Eingabe-Operator >> gilt auch für Strings, dass Whitespaces die Elemente der Eingabe trennen. D.h. bei der Eingabe von z.B. „Hallo Welt“ landet nur das „Hallo“ im String und der Rest bleibt im Stream stehen, da das Leerzeichen zwischen den Wörtern die Eingabe trennt.

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

int main()
{
   string s;
   cout << "Eingabe: ";
   cin >> s;
   cout << '"' << s << '"';
}
```

Mögliche Ein- und Ausgabe:
```
Eingabe: Hallo Welt
"Hallo"
```

Wollen Sie eine komplette Eingabezeile inkl. Whitespaces in einen String einlesen, so müssen Sie die Funktion „std::getline“ aus der Standard-Bibliothek benutzen (Header „string“). Defaultmäßig liest „std::getline“ bis zum abschliessenden ‚\n’, d.h. bis zum Return vom Benutzer. Dabei wird das Ende-Zeichen (hier ‚\n’) nicht in den String geschrieben, aber trotzdem aus dem Stream entfernt. Es wird nur als Zeilen-Ende-Kennung genutzt.

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

int main()
{
   string s;
   cout << "Eingabe: ";
   getline(cin, s);
   cout << ">>" << s << "<<\n";
}
```

```
Mögliche Ein- und Ausgabe:
Eingabe: Hallo Welt
>>Hallo Welt<<
```

Hinweis - Einlesen einer ganzen Zeile als String von der Tastatur kann eigentlich nicht schief gehen. Denn was soll passieren können?

- Benutzer-Eingaben, die nicht zum Typ passen - so wie Zeichen sich nicht in einen „int“ einlesen lassen - kann es bei Strings nicht geben. In einen String passen nun mal alle Zeichen, die man auf einer Tastatur eingeben kann, hinein.
- Typische Datei-Problem wie Datei-Ende, gekappte Netzwerk-Verbindungen, entfernte Disketten, oder sowas können bei der Tastatur nicht auftauchen.
- Und wenn die Verbindund zur Tastatur unterbrochen wird, z.B. weil jemand die Tastatur entfernt? Aus Sicht der Eingabe ist auch das kein *wirklicher* Fehler. Denn die Eingabe verbleibt beim Betriebssystem bis der Benutzer „Return“ drückt. Und entweder gibt es ein „Return“, dann sollte die Eingabe auch fehlerfrei zum Programm zurückkehren, oder es gibt kein „Return“ - dann aber ist aus Sicht des Programm die Eingabe nicht beendet.

Wenn wir das so akzeptieren, dann heißt das letztlich, dass wir beim Lesen von Strings von der Tastatur keine Fehlerüberprüfung benötigen. Das gleiche gilt dann übrigens auch für das Lesen von einzelnen Zeichen - siehe Kapitel 8.2.1. Aber bedenken Sie immer: diese Argumentation gilt wirklich nur absolute Ausnahmen wie z.B. für einzelne Zeichen bzw. Strings und die Tastatur. Die allermeisten Eingaben können schief gehen.

9.1.4 String-Verkettungen

Strings können z.B. mit den Operatoren „+“ und „+=“ mit Strings und Zeichenketten-Konstanten verkettet werden. Der Operator „+=“ funktioniert auch mit Zeichen.

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

int main()
{
   string s1("Ot");
   string s2("to");
   string s3(s1 + s2);
   cout << s3 << '\n';              // -> Otto
```

```
   s3 += s1;
   cout << s3 << '\n';              // -> OttoOt
   s3 += 'x';
   cout << s3 << '\n';              // -> OttoOtx
   s3 += "-y-z";
   cout << s3 << '\n';              // -> OttoOtx-y-z
   s3 = s1 + "::" + s2;                                  // (*)
   cout << s3 << '\n';              // -> Ot::to
}
```

Ausgabe

```
Otto
OttoOt
OttoOtx
OttoOtx-y-z
Ot::to
```

Achtung – wenn Sie den „+“ Operator im Zusammenhang mit Zeichenketten-Konstanten benutzen (z.B. wie in Zeile (*)), passen Sie auf, dass mindestens einer der ersten beiden Operanden ein String ist. Ansonsten erzeugen Sie eine Adress-Addition.

9.1.5 Vergleiche

Strings können mit anderen Strings oder mit Zeichenketten-Konstanten mit den bekannten Vergleichs-Operatoren ==, !=, <, >, <=, >= verglichen werden. Diese Operatoren liefern einen „bool“ Wert (also „false“ oder „true“) zurück, und können daher z.B. in If-Ausdrücken verwandt werden.

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

int main()
{
   string s1("C++");
   string s2("APL");

   if (s1==s2)
   {
      cout << s1 << " und " << s2 << " sind gleich\n";
   }
   else
   {
      cout << s1 << " und " << s2 << " sind ungleich\n";
   }

   bool lt = s1<s2;
   cout << s1 << " ist " << (lt?"": "nicht ") << "kleiner als " << s2 << '\n';
}
```

Ausgabe

```
C++ und APL sind ungleich
C++ ist nicht kleiner als APL
```

Achtung – die Kleiner- und Größer-Relationen werden defaultmäßig nur über die Kodierung der Zeichen ausgewertet – es wird daher keine lexikalische Ordnung berücksichtigt, und damit auch z.B. keine Groß- und Kleinschreibung beachtet. Für spezielle Ordnungen auf Strings muss eine andere Lösung genommen werden.

9.1.6 Index-Zugriff auf einzelne Zeichen

Es existiert ein lesender und schreibender 0-basierter Zugriff auf einzelne Zeichen mit dem Index-Operator „[ ]“ – der Rückgabe-Typ ist „char“ bzw. „char&“. Übergeben wird der Index natürlich als Argument vom Typ „std::string::size_type“. Für konstante Strings, d.h. mit „const“ definierte String-Variablen, gibt es natürlich nur Lese-Zugriff.

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

int main()
{
   string s1("C++");
   cout << s1 << '\n';

   char c = s1[0];
   cout << c << '\n';

   s1[s1.length()-1] = 'D';
   cout << s1 << '\n';

   const string s2("Computer");
   c = s2[2];                          // nur lesender Zugriff, da "const"
   cout << c << '\n';
}
```

Ausgabe

```
C++
C
C+D
m
```

Achtung – Zugriffe ausserhalb des Strings werden nicht abgefangen, das Verhalten ist undefiniert!

9.1.7 Teil-Strings

Mit der Element-Funktion „substr“ kann ein Teil-String aus einem String herauskopiert werden. Substr kann in drei Varianten aufgerufen werden:

- substr()
  Gibt den kompletten Original-String als Kopie zurück.
- substr(string::size_type idx)
  Gibt den Teilstring ab dem Index „idx“ inkl. zurück.
- substr(string::size_type idx, string::size_type n)
  Gibt den Teilstring ab dem Index „idx“ inkl. mit max. „n“ Zeichen zurück.

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

int main()
{
   string s ("123456789");
   cout << s.substr() << '\n';
   cout << s.substr(4) << '\n';
   cout << s.substr(4, 3) << '\n';
   cout << s.substr(7, 3) << '\n';
   cout << s.substr(7, 10) << '\n';
}
```

Ausgabe

```
123456789
56789
567
89
89
```

9.1.8 Suchen und Ersetzen

In Strings kann auf vielfältige Art und Weise nach Zeichen, Zeichenketten und mehr gesucht werden – und String-Teile können durch Texte ersetzt werden.

Hier eine kleine Auswahl von Element-Funktionen zum Thema „Suchen und Ersetzen“ in „std::string“:

- string::size_type find(const char* str, string::size_type pos = 0) const
  Sucht das Vorkommen von „str“ ab der Postion „pos“ und liefert die Postion zurück. Wird „str“ nicht gefunden, so wird „string::npos“ zurückgegeben. Hierbei sind auch Start-Positionen außerhalb des Strings erlaubt – in diesem Fall ist die Rückgabe auch „string::npos“.
- string::size_type find(char c, string::size_type pos = 0) const
  Sucht das Vorkommen des Zeichens „c“ ab der Postion „pos“ und liefert die Position zurück. Wird das Zeichen „c“ nicht gefunden, so wird „string::npos“ zurückgegeben. Hierbei sind auch Start-Positionen außerhalb des Strings erlaubt – in diesem Fall ist die Rückgabe auch „string::npos“.
- string& replace(size_type pos, size_type n, const char * str)
  Ersetzt im String an der Postion „pos“ „n“ Zeichen durch „str“.
- string::size_type rfind(char c, string::size_type pos) const
  Sucht **rückwärts** das Vorkommen des Zeichens „c“ ab der Postion „pos“ und liefert die Postion zurück. Wird das Zeichen „c“ nicht gefunden, so wird „string::npos“ zurückgegeben.

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

int main()
{
   string s("Java ist toll und Java ist gut");

   for (
      string::size_type pos=s.find("Java");
      pos!=string::npos;
      pos=s.find("Java", pos))
   {
      s.replace(pos, 4, "C++");
   }
   cout << "Besser: \"" << s << "\"\n";

   for (
      string::size_type pos=s.find(' ');
      pos!=string::npos;
      pos=s.find(' ', pos))
   {
      s.replace(pos, 1, "__");
   }
   cout << "Komisch: \"" << s << "\"\n";
}
```

Ausgabe

```
Besser: "C++ ist toll und C++ ist gut"
Komisch: "C++__ist__toll__und__C++__ist__gut"
```

Es gibt noch viele weitere Varianten von „find“ und „replace“ – siehe Literatur.

Hinweis – für den Fall, dass Sie es aufwändig finden, alle Vorkommnisse in einem String über eine Schleife ersetzen zu müssen – der Standard-String bietet nicht mehr – in der Boost String Algorithmen Library findet sich ein einfacher zu nutzendes „replace_all“.

9.1.9 Löschen

Um Teile aus einem String zu löschen gibt es vier „erase“ Funktionen – hier soll eine davon als Beispiel kurz vorgestellt werden:

- string& string::erase(string::size_type pos=0, string::size_type n=string::npos)
  Löscht die ‚n‘ Zeichen ab der Position ‚pos‘ (0-basiert) aus dem String.
  Wird keine Zeichen-Anzahl ‚n‘ angegeben, wo werden alle Zeichen ab ‚pos‘ inkl. gelöscht
  Wird gar kein Parameter übergeben, so wird der komplette String gelöscht.

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

int main()
{
   string s("C++ ist nicht toll");
   s.erase(8, 6);                    // loescht 6 Zeichen ab dem neunten Zeichen inkl.
   cout << "-> '" << s << "'\n";
   s.erase(3);                       // loescht alle Zeichen ab dem vierten inkl.
   cout << "-> '" << s << "'\n";
   s.erase();                        // loescht alle Zeichen
   cout << "-> '" << s << "'\n";
}
```

```
Ausgabe
-> 'C++ ist toll'
-> 'C++'
-> ''
```

Für weitere Varianten der Funktion „erase“ siehe Literatur.

9.2 String-Views

Mit C++17 wurde ein leichtgewichtiger Text-Proxy Typ in C++ eingeführt, der String-View „std::string_view“ aus dem Header „string_view“. Auch ihn gibt es natürlich 5 x für alle Zeichen-Typen – wir nutzen ihn aber nur in der einfachen Version für „char“. Er ist ein nicht-besitzender lesender View auf einen Text.

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

int main()
{
   string_view sv{ "Ich bin ein Text" };      // String-View referenziert Text
   cout << sv << '\n';

   string s{ "Ich bin ein String" };
   sv = s;                                    // String-View referenziert String
   cout << sv << '\n';
}
```

```
Ausgabe
Ich bin ein Text
```

```
Ich bin ein String
```

Seine Eigenschaften sind:

- Er kann mit beliebigen zusammnhängenden Texten umgehen, daher z.B. für Strings, String-Views und Zeichenketten-Konstanten. Er ist sozusagen die allgemeine Lösung.
- Er ist nicht besitzend – daher er geht davon aus, dass der Source-Text weiterhin unverändert existiert. Wenn Sie den Source-Text verändern oder löschen, so sind danach Operationen auf dem String-View UB. Das macht ihn so „leichtgewichtig“ und dadurch speichersparend und performant.
- Man kann mit ihm den Text nur lesen, nicht schreiben (daher nicht verändern).
- Er hat dabei die gleiche Schnittstelle wie ein String, aber eben nur die lesenden Funktionen.

Da er den Text nicht besitzt, muss er ihn nicht kopieren. Das macht ihn so speichersparend und performant. Seine Hauptanwendung ist daher, dass man ihn als leichtgewichtigen Proxy für Texte nutzen kann, ohne dass diese kopiert werden müssen. Dies ist immer hilfreich, wenn man nur temporäre Sichten auf oder in einen bestehenden Text benötigt, z.B. als Parameter für Funktionen, oder als gleitender View über einen größeren Text.

9.3 String-Streams

Im Standard sind eine Eingabe-String-Stream- und eine Ausgabe-String-Stream-Klasse definiert:

- Eingabe-String-Stream: istringstream
- Ausgabe-String-Stream: ostringstream
- Header: sstream

Sie können wie normale Stream-Klassen verwendet werden, da sie normale Stream-Klassen sind. Wie das gelöst wird, lernen wir später im Kapitel über Vererbung.

Im Gegensatz zu den bekannten Ein- und Ausgabe-Streams, liest der Eingabe-String-Stream aus einem String statt von Tastatur, und der Ausgabe-String-Stream schreibt in einen String statt auf die Konsole.

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

int main()
{
   int i = 42;
   double d = 3.14;

   ostringstream oss;
   oss << i << ' ' << d;
   string s(oss.str());

   cout << "s: \"" << s << "\"\n";          // -> s: "42 3.14"
   cout << "len: " << s.length() << '\n';   // -> len: 7

   i = 0;
   d = 0.0;
```

```
   cout << "i: " << i << '\n';             // -> i: 0
   cout << "d: " << d << '\n';             // -> d: 0

   istringstream iss(s);                   // (*)
   iss >> i >> d;

   cout << "s: \"" << s << "\"\n";         // -> s: "42 3.14"
   cout << "i: " << i << '\n';             // -> i: 42
   cout << "d: " << d << '\n';             // -> d: 3.14
}
```

Ausgabe

```
s: "42 3.14"
len: 7
i: 0
d: 0
s: "42 3.14"
i: 42
d: 3.14
```

Ausgabe-String-Stream:

- Mit der Element-Funktion „str()“ kommt man an den Ergebnis-String der Ausgaben.
- Schreiben in einen Ausgabe-String-Stream kann *eigentlich* nicht schief gehen – solange die Ausgabe-Funktionen den Stream nicht auf „*fail*“ setzen (was sie normalerweise nicht machen), und intern genügend Speicher vorhanden ist (was über Exceptions gemeldet wird, die wir in der Vorlesung leider nicht besprechen).

Eingabe-String-Stream:

- Der Eingabe-Stream wird bei der Konstruktion direkt mit dem String initialisiert, aus dem gelesen werden soll – siehe Zeile (*) im Beispiel.
- Der String wird beim Lesen nicht verändert – der Stream erzeugt sich während der Initialisierung eine Kopie.
- Solange Sie absolut sicher sind, dass sich der String-Stream Inhalt problemlos in die entsprechenden Variablen streamen läßt (wie z.B. im Beispiel), kann die entsprechende Fehlerabfrage (siehe Kapitel 8.2.1) entfallen. Sobald aber nur die geringste Möglichkeit von Problemen besteht, müssen Sie diese natürlich auch bei String-Streams durchführen.

Im Gegensatz zur String-Konkatenierung mit dem Plus-Operator „+“ (siehe Kapitel 9.1.4), der einfach nur 2 Strings hintereinander hängt, stehen einem bei String-Streams alle Stream-Möglichkeiten für Typen und Formatierungen zur Verfügung:

- Die Erzeugung von Strings mit Streams erlaubt die Verwendung aller stream-fähigen Typen (vergleiche auch das Kapitel 9.4, indem Strings aus „int“s erzeugt werden).
- Außerdem können sämtliche Manipulatoren und Formatierungen auf Streams damit auch zur String-Erstellung genutzt werden (vergleiche Kapitel 8.1.2).

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

int main()
{
   ostringstream oss;
   oss.fill('>');
   oss << setw(1) << 1 << " - ";
```

```
    oss << setw(2) << 2 << " - ";
    oss << setw(3) << 3 << " - ";
    oss << setw(4) << 4;
    string s = oss.str();

    cout << "Erzeugter String: " << s << '\n';
}
```

Ausgabe

```
Erzeugter String:  1 - >2 - >>3 - >>>4
```

9.4 Wandlungen

Ein häufiges Problem in der Praxis sind Wandlungen von Strings in andere Datentypen, und umgekehrt.

- Problem 1 tritt z.B. häufig dann auf, wenn das Programm Kommando-Zeilen-Argumente bekommt, oder Eingaben von der Tastatur oder aus Dateien liest, und es diese dabei als String bekommt. Intern werden die Eingaben häufig als int's, double's oder sonstwas benötigt.
- Umgekehrt liegen viele Daten im Programm in entsprechenden Variablen vor, und müssen für z.B. die Ausgabe in einen String gewandelt werden. Die Ausgabe auf z.B. „std::cout" macht dies zwar automatisch, aber manchmal müssen die Ausgaben als String vorliegen (z.B. für spezielle Formatierungen, oder weil die Ausgabe-Elemente nur Strings annehmen).

Für die Wandlungen zwischen Strings und den einfachen elementaren Datentypen gibt es seit C++11 einfache schöne Unterstützungen.

- „to_string" aus dem Header „string" wandelt viele Integer-Typen und alle Fließkomma-Typen in Strings um:
  http://en.cppreference.com/w/cpp/string/basic_string/to_string
- „stoi", „stol", „stod" und viele weitere Funktionen wandeln Strings in Integer- bzw. Fließkomma-Typen um. Im Falle von Fehlern (die bei dieser Wandlung natürlich vorkommen können), werden diese mit Exceptions gemeldet. Leider werden Exceptions in der Vorlesung nicht besprochen. Außerdem unterstützen diese Funktionen noch verschiedene Basen und die Möglichkeit, die Anzahl an geparsten Zeichen zu bekommen:
  http://en.cppreference.com/w/cpp/string/basic_string/stol
  http://en.cppreference.com/w/cpp/string/basic_string/stof

Beispiel 1 mit einer Wandlung von String zu „int". Hier wird der String „s" in einen „int" gewandelt, damit mit dem Wert „42" gerechnet werden kann.

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

int main()
{
    string s("42");             // Dies ist die Zeichenkette "42" - kein Integer
                                // Zum Rechnen benoetigt man die "42" z.B. als "int"

    int n1 = stoi(s);           // Nun hat man einen "int" zum Rechnen

    int n2 = 2*n1;
    cout << "2 x " << n1 << " = " << n2 << '\n';
```

```
}
```

Ausgabe
```
2 x 42 = 84
```

Beispiel 2 mit einer Wandlung von „int“ zu String. Alle „2“en im Wert „123212321“ sollen durch 2 Underscores „__“ ersetzt werden – dies geht nur textmäßig. Daher ist vor der Bearbeitung eine Wandlung zu String notwendig.

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

int main()
{
   int n = 123212321;          // Dies ist ein "int" mit dem Wert "123212321" - kein Text
                               // Fuer Text-Operationen benoetigt man einen String

   string s = to_string(n);    // Nun hat man einen String zum Bearbeiten

   for (auto pos = s.find('2'); pos != string::npos; pos = s.find('2', pos))
   {
      s.replace(pos, 1, "_");
   }
   cout << s << '\n';
}
```

Ausgabe
```
1__3__1__3__1
```