C++ FOR (Montagvormittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Falls die Übungsaufgabe nicht wegen eines verkürzten Vormittagsteils entfällt, erfolgt die Besprechung der Musterlösung(en) im direkten Anschluss an die Mittagspause.

Auffrischung einiger wichtiger Grundlagen

- Getrennte Kompilierung
- Definition/Reference-Modell
- Schreibschutz durch den Compiler

- Zeiger versus Referenzen
- RValue-Referenzen (neu in C++11)
- Defaultwerte für Argumente

- Überladen von Funktionen
- Überladen von Operatoren

- Automatische Typ-Konvertierung
- Typ-Konvertierung mittels *Cast*
- Klassenspezifische Typ-Konvertierung

Getrennte Kompilierung

Grundlagen der getrennten Kompilierung

Ein C++-Programm wird üblicherweise in eine mehr oder weniger große Zahl von Übersetzungseinheiten aufgeteilt.

- Diese stehen in Implementierungsdateien, deren Dateinamens-Suffix meist `.cpp` ist.
- Sind ein und dieselben Informationen in mehr als einer Übersetzungseinheit notwendig, gehören diese in Header-Files, deren Dateinamens-Suffix meist `.h` ist (seltener: `.hpp`).

> **(i)** Bereits bei einer kleinen Zahl von Übersetzungseinheiten sind die Abhängigkeiten zwischen diesen inklusive ihrer Header-Files oft nur noch schwer zu überblicken, so dass sich die Verwendung eines Build-Systems empfiehlt.*

*: Unter den heute verwendeten Build-Systemen hat das 1976 an den [Bell-Labs] von Stuart Feldman entwickelte Unix `make` weitaus mehr als eine nur historische Bedeutung, in der ihrer Kernsyntax sind auch moderne Derivate wie GNU make und CMake nach wie vor identisch zu ihrem Vorläufer.

Abhängigkeiten zwischen Header-Files

Nicht selten kommt es auch zu Abhängigkeiten von Header-Files untereinander, wenn z.B. in einem Header-File ein Datentyp oder eine Klasse verwendet wird, die in einem anderen Header-File definiert ist.

In solchen Fällen ist es üblich, im abhängigen Header-File den als Voraussetzung erforderlichen zweiten Header-File direkt zu inkludieren.

```
// header file: Base.h
class Base {
    // ...
};
```

```
// header file: Derived.h
#include "Base.h"
class Base : public Derived {
    // ...
};
```

Include-Guards

Da ein und derselbe Header-File oft auf verschiedenen Wegen inkludiert wird, muss die **mehrfache Verarbeitung*** ausgeschlossen werden. Dies geschieht mit sogenannten Include-Guards:

```
// header file: Base.h
#ifndef BASE_H
#define BASE_H
class Base {
    …
};
#endif
```

```
// header file: Derived.h
#ifndef DERIVED_H
#define DERIVED_H
#include "Base.h"
class Base : public Derived {
    …
};
#endif
```

*: Der Grund hierfür liegt vor allem in der One Definition Rule (ODR), welche die wiederholte Einführung von Bezeichnern stark einschränkt.

Namespaces

[!] Der Include-Guard sollte stets auch den Namen des `namespace` enthalten!

Wie folgende Beispiel zeigt, ist der pure Klassenname in einem Header-File als Include-Guard nicht unbedingt ausreichend:

```
#ifndef MINE_SOMECLASS_H
#define MINE_SOMECLASS_H
namespace Mine {
    class SomeClass {
        …
    };
}
#endif
```

```
#ifndef OTHER_SOMECLASS_H
#define OTHER_SOMECLASS_H
namespace Other {
    class SomeClass {
        …
    };
}
#endif
```

Zyklische Abhängigkeiten

Einiges Kopfzerbrechen dürfte die Fehlermeldung bereiten, welche trotz (oder wegen?) des Include Guard aus der folgenden Situation resultiert:*

```
// file: someclass.h
#ifndef SOMECLASS_H
#define SOMECLASS_H
#include "OtherClass.h"
namespace Mine {
    class SomeClass {
        // ...
        OtherClass *link;
    };
}
#endif
```

```
// file: otherclass.h
#ifndef OTHERCLASS_H
#define OTHERCLASS_H
#include "SomeClass.h"
namespace Other {
    class OtherClass {
        // ...
        SomeClass *link;
    };
}
#endif
```

*: In kompilierbarer Form finden Sie die Dateien zu diesem Beispiel unter Examples/Cyclic_Broken und Examples/Cyclic (fehlerbereinigt).

Definition/Reference-Modell

Zu jeder in einem Progamm verwendeten Variablen (egal Grundttyp oder Objekt-Instanz) muss es **genau eine** Definition geben, mit möglicherweise vielen Bezugnahmen.

- Bei der Bezugnahme aus einer anderen Übersetzungseinheit muss diese wiederum eine `extern`-Deklaration vornehmen.
- Die tatsächliche **Definition darf** ebenfalls das Wort `extern` enthalten, wenn sie mit einer Initialisierung verbunden ist.
- Sie **muss** es sogar, wenn zugleich eine `const`-Qualifizierung verwendet wird und keine`extern`-Deklaration vorausgeht.

[!] Die Verwendung von `const` auf globaler Ebene impliziert - vielleicht überraschenderweise - den Sichbarkeitsschutz gegenüber dem Linker.*

*: Einstmals wurde von Bjarne Stroustrup damit das Ziel verfolgt, eine möglichst unproblematische Umstellung von `#define`-s auf `const` zu unterstützen. Das Risiko von Name Clashes in der Link-Phase wurde damit ausgeschlossen auf Kosten von evtl. mehrfach für den selben Zweck reservierten Speicher im globalen Bereich, der bei einfachen Datentypen wiederum von optimierenden Compilern vermieden werden kann, solange kein Zugriff auf die Adresse der jeweiligen Variablen erfolgte.

Schreibschutz durch den Compiler (`const`)

Mittels `const`-Qualifizierung wird die Zuweisung eines (neuen) Werts an eine Variable verboten. Es ist nur noch die Initialisierung bei der Definition möglich.

```
const int x = 42;               // Initialisierung notwendig
extern const unsigned VERSION; // nur Bezugnahme
```

Folgendes führt nun zu einem Compile-Fehler:*

```
++x;
…
if (VERSION = 3014u) {
    // special case for version 3.14
    …
}
```

*: Abhängig von der Art der Variablen und den Möglichkeiten der Hardware ist für `const`-qualifizierte Variablen eventuell auch ein physikalischer Schreibschutz möglich.

Anwendung der const-Qualifizierung auf Zeiger

Bei Zeigern ist zu beachten, dass sich die Konstantheit auf den Zeiger selbst beziehen kann

```
const int *p;  // gleichbedeutend zu: int const *p;
*p = …;        // Compile-Fehler
```

oder auf das, was über den Zeiger erreichbar ist:

```
int *const p = …; // muss initialisiert werden
p = …;    // Compile-Fehler
++p;      // Compile-Fehler
```

Bezugnahme über Referenz

Das klassiche Beispiel ist eine Funktion `swap`, welche zwei Werte miteinander vertauscht:

```
void swap(int *p, int *q) {
    const int t = *p;
    *p = *q;
    *q = t;
}
…
int a, b;
…
if (a > b) swap(&a, &b);
…
```

```
void swap(int r, int s) {
    const int t = r;
    r = s;
    s = t;
}
…
int a, b;
…
if (a > b) swap(a, b);
…
```

Referenzen versus Zeiger

C++ Referenzen können auf zwei Arten betrachtet werden:

- Eine alternative Syntax für Zeigern, welche
  - die Dereferezierung bei der Verwendung impliziert (* automatisch vorangestellt);
  - bzw. den Adress-Operator bei der Initialisierung (& autmatisch vorangestellt).
- Oder aber einen Alias-Name für bereits (an anderer Stelle) existierenden typisierten Speicherplatz.

> Der für Zeiger und Referenzen erzeugte Maschinen-Code unterscheidet sich in der Regel nicht - unterschiedlich ist nur die Syntax bei Initialisierung und beim Zugriff auf das, was referenziert wird.*

*: Der Nachweis ist beim `g++` leicht durch die Erzeugung des Assembler-Codes möglich, wozu die Option `-S` (Großbuchstabe) anzugeben ist und das Ergebnis in einer Datei mit Suffix `.s` (Kleinbuchstabe) landet. Für noch schnellere, praktische Experimente zu Fragen der Code-Erzeugung sei auf den folgenden Online Compiler hingewiesen: http://gcc.godbolt.org/

RValue-Referenzen

Mit C+11 neu eingeführt wurde das Konzept der Rvalue-Referenzen. Sie lassen sich nur mit Audrücken intitialisieren, also *temporären Werten* auf die dann kein anderer Zugriff als über die Referenz besteht.

Nachfolgend zusammengefasst die wichtigsten Regeln:*

```
T &r = …;           // … must be modifiable T in memory
const T &cr = …;    // … must be modifiable T in memory OR
                    //       non-modifiable T in memory OR
                    //       temporary T in memory (expression)
T &&rr = …;         // … must be temporary T in memory (expression)
```

Die Hauptanwendung liegt beim Überladen von Funktionen für unterschiedliche Herkunft von Argumenten, und dort wiederum insbesondere bei Kopier-Konstruktor und -Zuweisung, denen damit Move-Varianten zur Seite gestellt werden können.

*: In dem für obige Szenarien typischen Fall von Funktionsargumenten könnte im Fall der RValue-Referenz - anders als bei klassischen `const`-Referenzen - das übergebene `T` nun modifiziert werden, allerdings nur so weit, dass der Destruktor nach wie vor seine Arbeit verreichten kann.

Defaultwerte für Argumente

Argumente können mit Default-Werten versehen werden.

- Dies muss ggf. von rechts nach links geschehen, das heißt:
- Sobald ein Argument einen Default-Wert hat, müssen die weiter rechts stehenden ebenfalls einen Default-Wert haben.

> Der Defaultwert muss beim Funktionsaufruf bekannt sein als im Header-File stehen als Bestandteil des Funktions-Prototyps.

Die Namen für die formalen Argumente sind auch hier optional:*

```
// the following function can be called with 1..3 arguments:
double foo(int &count, int minsize = 0, char separator = 'z');
…
// same as:
double foo(int &, int = 0, char = 'z');
```

*: Using names for arguments in prototypes has pro's and con's: it is of course more self-documenting but there is at least a remote chance for surprising and **extremely hard to find** name clashes with preprocessor macros. Hint: view preprocessor output (`g++ -E …`) whenever you get desparate because of an completely unexplainable syntax error in your source code.

Überladen von Funktionen

Mehrere Funktionen gleichen Namens können parallel existieren sofern sie sich in Anzahl und/oder Typ ihrer Argumente unterscheiden:

```
void foo(const char *);     /*1*/
double foo(int &, char);    /*2*/
double foo(double, double); /*3*/
```

Gemäß den tatsächlichen Argumenten entscheidet nun der Compiler, was verwendet wird:*

```
int x; double y;
…
foo("hello, world");   /*1*/
foo(42, 'z');          /*2*/
foo(y, y/2);           /*3*/
foo(x, y);             /*?*/
```

*: To figure out what happens in the last case (with the question mark in the comment) is left as an exercise to the reader … :-)

Überladen von Funktionen (cont.)

Die `const`-Qualifizierung eines Parameters macht ebenfalls einen Unterschied:

```
void foo(char *); /*4*/
char data[100];
…
foo(data); /*4*/
foo("hi"); /*1*/
```

Es ist allerdings nicht zwingend notwendig, dass immer zwei Funktionen existieren, je eine für den konstanten und den nicht-konstanten Fall:

- Ohne die Funktion für den **nicht**-konstanten Fall wird die Funktion für den konstanten Fall für alle Aufrufe verwendet.*
- Ohne die Funktion für den **konstanten** Fall wird der Aufruf mit `"hi"` zum Compile-Fehler, da String-Literale den Typ `const char *` haben.

---

*: Generell gesehen ist es kein Problem, wenn einer Funktion, die versprochen hat, über ein Zeiger- oder Referenz-Argument erreichbaren Speicherplatz nicht zu verändern, die Adresse von veränderbaren Speichreplatz bekommt - **umgekehrt ist es aber sehr wohl ein Problem!**

Überladen von Operatoren

Operatoren können überladen werden mit Funktionen, deren Name mit dem Wort `operator` beginnt.

- Die meisten Operatoren können wahlweise mit freistehenden Funktionen oder mit Member-Funktionen überladen werden.
- Einige Operatoren sind auf Member-Funktionen eingeschränkt.

Zur konsistenten Überladung ganzer Operatorgruppen kann Boost.Operators hilfreich sein.

Operator-Überladung mit freistehender Funktion

Diese sieht prinzipiell so aus:

```
MyClass operaor+(const MyClass &lhs, const MyClass &rhs) {
    … // do whatever must be done
    return …;
}
```

Der Rückgabetyp ist dabei beliebig, die `return`-Anweisung muss natürlich vom Typ her passend sein, genauer gesagt: Der Ausdruck hinter `return` muss exakt den Rückgabetyp haben oder in diesen umwandelbar sein.

Operator-Überladung mit Member-Funktion

Diese sieht prinzipiell so aus:

```
MyClass &MyClass::operator+=(const MyClass &rhs) {
    … // do whatever must be done
    return *this;
}
```

Auch hier ist der Rückgabetyp grundsätzlich frei wählbar. Gemäß den Konventionen bei den Standardtypen wird in der Regel das durch die Operation gerade veränderte Objekt selbst zurückgegeben.

Überladung von Kopier-Konstruktor und -Zuweisung

Für einige Arten von Objekten muss die Zuweisungs-Operation überladen werden, da der Default - elementweise Zuweisung - ungeignet ist.*

```
class MyClass {
    T *some_ptr;
    …
public:
    …
    // avoid compiler defaults:
    MyClass(const MyClass& rhs);
    MyClass& operator=(const MyClass& rhs);
}
```

Der klassische Indikator sind als Member enthaltene Zeiger auf Speicherplatz, welcher individuell für jedes Objekt vorhanden sein muss.*

*: In C++ books this is often referred to as Rule of Three - the third member function for which the default is not appropriate is the destructor, of course.

Überladung von Move-Konstruktor und -Zuweisung

In C++11 können die RValue-Referenzen dazu verwendet werden, Initialisierung und die Zuweisung unterschiedlich zu implementieren,* abhängig davon ob

- der dazu verwendete Ausdruck direkt ein Objekt repräsentiert, das danach unverändert weiter existieren soll, oder
- temporärer Speicherplatz für einen berechneten Ausdruck oder ein Funktionsergebnis, der ohnehin bald verworfen wird.

```
class MyClass {
public:
    …
    // copy versions (rhs lives on separately afterwards)
    MyClass(const MyClass& rhs);
    MyClass& operator=(const MyClass& rhs);
    // move versions (rhs gets destroyed afterwards)
    MyClass(MyClass&& rhs);
    MyClass& operator=(MyClass&& rhs);
}
```

*: Turning the classic C++ *Rule of Three* into the Rule of Five in C++11.

Implementierung von Move-Konstruktor und -Zuweisung

Nachdem Move-Versionen deklariert sind, müssen diese natürlich auch implementiert werden.

Wenn der Kopierkonstruktor wie folgt aussieht ...

```
MyClass::MyClass(const MyClass &rhs)
    : …, some_ptr(new T(*rhs.some_ptr)), …    // cloning ressource
{ … }
```

... könnte dieser Move-Konstruktor angemessen sein:

```
MyClass::MyClass(MyClass &rhs)
    : …, some_ptr(rhs.some_ptr), …    // taking over ressource
{ …; rhs.some_ptr = nullptr; … }      // INVALIDATING it for rhs!
```

Die Zuweisungen sind ähnlich, müssen aber zuerst `some_ptr` freigeben.*

*: The general difference between constructor and assignment is that the former gets just a piece of memory while the later finds a valid object that needs to be properly de-constructed first.

Unterscheidung Copy- und Move-Versionen

Existieren beide Fassungen (Copy und Move), ergibt sich folgendes Verhalten:

```
MyClass foo() { return …; }  // … must return something of
                             // type MyClass (or at least
                             // something convertible to it)
MyClass a;         // (expects c'tor with no argumens)
MyClass b(a);      // copy c'tor (does not alter a)
MyClass c(foo()); // move c'tor (may alter temporary)
a = c;             // copy assignment (does not alter c)
b = foo();         // move assignment (may alter temporary)
c = a + b;         // move assignment provided operator+ is
                   // defined for MyClass (may alter temporary)
```

Automatische Typ-Konvertierungen

Die wichtigsten Regeln für automatische Typkonvertierungen sind:

- Innerhalb aller Typen, die arithmetische Werte darstellen, inklusive `char` (= kleine Ganzzahlen) und `bool` (Wahrheitswerte);
- Aufzählungstyp in arithmetischen Wert;
- Zeiger als Wahrheitswert (alles ungleich `nullptr` ist `true`);
- Typisierter Zeiger in allgemeinen Zeiger (`void *`);
- Zeiger oder Referenz auf öffentlich abgeleitete Klasse in Zeiger bzw. Referenz auf Basisklasse;
- Öffentlich abgeleitete Klasse auf Basisklasse durch Slicing;
- Klassenspezifische Typ-Konvertierung sofern in den beteiligten Objekten vorgesehen.

Typ-Konvertierung mittels *Cast*

Mittels sogenannter Cast-Operationen lassen sich weitere Typ-Umwandlungen erzwingen.

> **[!]** Die C-Syntax, bei welcher man den neuen Typ in runde Klammern setzt und den umzuwandelnden Wert als Operand dahinter schreibt, sollte in C++ nicht mehr benutzt werden.

Die neue Syntax beginnt mit einem der Schlüsselworte

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

Es folgt der gewünschte Zieltyp in spitzen Klammern und der umzuwandelnde Wert in runden Klammern.

Typ-Konvertierung mit `static_cast`

Hiermit lassen sich alle Typ-Umwandlungen explizit hervorheben, welche der Comiler auch automatisch vorgenommen hätte. Häufig entfallen dann Warnungen, die der Compiler typischerweise für nicht exakt übereinstimmende arithmetische Wert gibt, die sich durch die Umwandlung veändern könnten (da im Zieltyp nicht darstellbar).

Darüberhinaus funktioniert der `static_cast` in **beide** Richtungen für alle Umwandlungen, welche als **automatische** Umwandlung nur in einer Richtung eingesetzt werden:

- Arithmetische Wert zu `enum`-s
  - automatisch nur `enum`-s in arithmetische Werte
- Generische Zeiger (`void *`) zu typisierten Zeigern
  - automatisch nur typisierte Zeiger in `void *`
- Basisklassen in abgeleitete Klassen (Downcast)
  - automatisch nur abgeleitete zu Basisklassen (Upcast)

Typ-Konvertierung mit `dynamic_cast`

Hiermit lassen sich ausschließlich Typ-Umwandlungen innerhalb von Klassenhierarchien vornehmen, wobei im Fall von Downcasts zur Laufzeit eine Überprüfung stattfindet, ggf. mit Fehleranzeige, wenn der Cast nicht möglich ist.

Die Fehleranzeige besteht

- bei Casts auf Zeigerbasis in der Rückgabe eines Nullzeigers;
- bei Casts auf Referenzbasis im Auslösen einer `std::bad_cast`-Exception.

Weiteres wird später im Rahmen der Laufzeit-Typprüfung (RTTI) behandelt.

Typ-Konvertierung mit `const_cast`

Die hiermit erzielbaren Typveränderungen beschränken sich auf das

- Hinzufügen oder
- Wegnehmen von `const` und `volatile`.

Alle anderen Unterschiede zwischen dem Zieltyp und dem Typ des umzuwandelnden Ausdrucks führen zu einem Compile-Fehler.

[!] Diese Art von Cast führt gemäß C++ ISO/ANSI Standard zu undefiniertem Verhalten, zumindest wenn auf eine mit Schreibschutz definierte Adresse nach Wegnehmen der `const`-Qualifizierung schreibend zugegriffen wird.

Das typische Fehlerbild kann vom Programmabsturz bis zu einer inkonsistenten Wertverwendung reichen (teilweise alter Wert, teilweise neuer Wert) ... oder es mag sogar so erscheinen, als würde alles funktionieren.

Typ-Konvertierung mit `reinterpret_cast`

Dieses Konstrukt wird vor allem dazu eingesetzt, Zeiger auf (bekannte) Hardware-Adressen zu setzen, wie das u.a. im Bereich der Embedded Progammierung und bei Gerätetreibern notwendig sein kann.

Darüber hinaus kann man auch mit einem `reinterpret_cast`

- wie auch mit `static_cast` generische Zeiger (`void *`) in typisierte Zeiger umwandeln, und
- anders als mit `static_cast` einen typisierten Zeiger direkt in einen anders typisierten Zeiger umwandeln.

> Die per `reinterpret_cast` gebotene Möglichkeit, quasi jedes Bitmuster im Speicher gemäß einem beliebigen Typ zu interpretieren, veranlassen Kritiker von C++ zur Aussage, die Sprache sei unsicher, da nicht vollständig typgeprüft.

Diese Kritik muss dann aber ebenso für Sprachen gelten, welche ein zur C/C++ `union` vergleichbares Konstrukt bieten, so etwa das gemeinhin eher als typsicher geltende Pascal.

Klassenspezifische Typ-Konvertierungen

Eine Klasse kann auch selbst festlegen, wie man sie aus einem anderen Typ erzeugt oder wie sie in einen anderen Typ umgewandelt wird. Bildlich kann man es sich so vorstellen:*

- Jede Klasse verfügt über eine Art *charakteristische* "*Steckverbindung*" die zunächst "nur zu sich selbst" passt.
    - Entsprechend kann man in Initialisierungen und Zuweisungen nur Objekte derselben Klasse verwenden.
- Konstruktoren mit genau einem Argument sind weitere "Eingänge",
    - die sozusagen zum charakteristischen (Ausgangs-) Steckverbinder einer anderen Klasse passen.
- Sogenannte Typ-Cast-Operatoren sind weitere "Ausgänge",
    - die sozusagen zum charakteristischen (Eingangs-) Steckverbinder einer anderen Klasse passen.

---

*: If you like that picture you may include base class conversions by assuming plugs with the same basic shape for class hierarchies, using code pins to make the output connector of a derived class fit into the input receiptable of its base class, but not vice versa.

Typumwandlungen durch Konstruktoren

Konstruktoren sind dann automatische Typumwandlungen, wenn sie

- genau ein Argument besitzen und
- **nicht** mit dem Schlüsselwort `explicit` markiert sind.

```
class MyClass {
    …
public:
    MyClass(int);  // each single argument c'tor is an
                   // automatic conversion ... except
    explicit MyClass(double);  // it is marked explicit
    …
};
```

Anwendung von Konstruktoren zur Typumwandlung

Typumwandlungen durch Konstruktoren kommen wie folgt zur Anwendung:

```
void foo(MyClass);
…
foo(33);             // OK, automatic conversion by c'tor
foo(3.3)             // NOT OK, c'tor is explict
foo(MyClass(3.3));   // OK, c'tor has been used explicitely
…
```

Typumwandlungen durch Type-Cast Operationen

Type-Cast Operationen benutzen eine spezielle Syntax, bei der **nach** dem Schlüsselwort `operator` der Zieltyp folgt:*

```
class MyClass {
    …
public:
    // this is called type-cast operator:
    operator Other() const { …; return …; }
    //                                 ^-- Other (or at least
    //                                     convertible to Other)

    // the usual explicit alternative:
    int to_int() const { …; return …; }
    //                             ^-- int (or at least
    //                                 convertible to int)
};
```

*: Dieser stellt zugleich den Ergebnistyp dar, den die `return`-Anweisung einer solchen Funktion liefern muss.

Anwendung von Type-Cast Operationen zur Typumwandlung

Die Typumwandlungen von der vorhergehenden Seite kommen wie folgt zur Anwendung:

```
void foo(Other);
void bar(int);
…
MyClass m;
foo(m);           // OK, implicit use of type-cast operator
bar(m);           // NOT OK (of course), but ...
bar(m.to_int());  // ... usual style for explicit conversion
```

Sonderfall: `explicit operator bool()`

Eine als `explicit` markierte Typumwandlung in einen Wahrheitswert stellt einen Sonderfall dar:

- Sie kommt **nicht** zur Anwendung bei Argumentübergabe, Initialisierung und Zuweisung,
- **jedoch bei bool'schen Operationen und Bedingungstests.**

Die beispiele auf der nächsten Seite setzen folgendes voraus:

```
class MyClass {
    …
public:
    explicit operator bool() {
       return …; // some bool
    }
    …
};

MyClass obj;
extern void foo(bool);
```

Beispiele: `explicit operator bool()`

Die folgenen Code-Fragmente setzten das begonnene Beispiel fort:

```
// this does NOT compile ...
foo(obj);

bool bv(obj);
bool bv(bool(obj));      //#2a
bv = obj;

if (obj == true) …
if (obj == false) …
if (obj == bv) …
// ... compare with code on
// the right for corrections
```

```
// this solves the problems:
foo(bool(obj));
foo(obj.operator bool()); //#1

bool bv((bool(obj)));     //#2b
bv = bool(obj);
if (bool(obj)) …          //#3
if (obj) …
if (!obj) …
if (bool(obj) == bv) …
// boolean operators work too:
if (obj && !bv) …
```

#1 verwendet eine etwas ungewöhnliche aber erlaubte Form des Aufrufs der Typumwandlung in `bool`.

#2a löst einen Fehler aus, der in einer syntaktische Mehrdeutigkeit begründet ist, die ein Paar zusätzlicher Klammern gemäß #2b beseitigt.

#3 ist eine zulässige aber überflüssige explizite Typumwandlung.

Typ-Sicherheit in C++

Die praktische Konsequenz aus den Risiken, welche die Konstrukte zur expliziten Typumwandlung mitbringen - **allen voran `reinterpret_cast`** - ist diese:

> **[!]** Alle Formen expliziter Typumwandlung sollten auf das absolut notwendige Minimum beschränkt werden.

Darüberhinaus werden auch klassenspezifische Typumwandlungen manchmal auf unerwartete Weise angewendet:*

> **(i)** Klassenspezifische Typumwandlung sollten generell nur dort verwendet werden, wo der stillschweigende Wechsel zwischen den beteiligten Typen als "natürlich" empfunden wird.

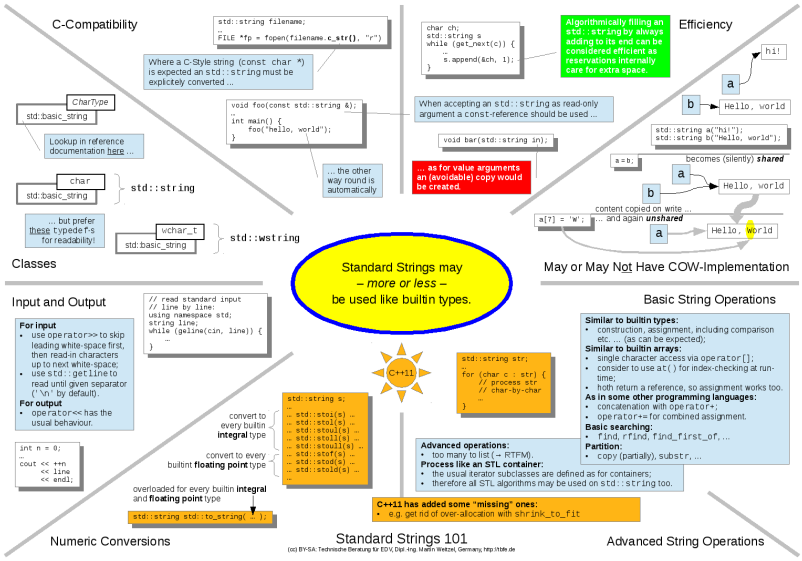
---

*: Or to put it slightly different: Experience showed there are scenarios of practical importance where a compile error would have been preferrable over the way the compiler made the code "correct" by applying a non-explicit constructor or type-cast operator.

Verwendung von Standard-Strings

- Klassen (Übersicht)
- Kompatibilität zu C
- Effizienz üblicher Implementierungen
- Optionales "Copy On Write"

- Grundlegende Operationen
- Weitere Operationen im Überblick
- Umwandlung von/in arithmetische Werte
- Ein- und Ausgabe



Klassen (Übersicht)

Die aus C++98 bekannten `std::string` und `std::wstring` sowie die von C++11 hinzugefügten Klassen `std::u16string` und `std::u32string` sind lediglich Typdefinitionen:*

```
namespace std {
    typedef basic_string<char> string;      // since C++98
    typedef basic_string<wchar_t> wstring;  // since C++98
    typedef basic_string<char16_t> wstring; // since C++11
    typedef basic_string<char32_t> wstring; // since C++11
}
```

Weitere Details sind nachzuschlagen in:

- http://en.cppreference.com/w/string/
- http://www.cplusplus.com/string/

*: The type definitions show only half the truth: other template arguments are a character traits class and an allocator (memory management policy). Both have been omitted as they do not change the essential point to make.

Kompatibilität zu C

Die Klasse `std::string` speichert die Zeichen eines Strings in einem zusammenhängenden* Speicherstück, das in der jeweils erforderlichen Mindestgröße auf dem Heap angelegt wird.

Ein Objekt der `std::string`-Klasse speichert typischerweise drei Zeiger:

- Die Adresse des ersten enthaltenen Zeichens.
- Die Addresse des letzten enthaltenen Zeichens.
- Die Addresse bis zu der Speicher alloziert ist.

Insbsondere sind die gültigen Zeichen nicht dahingehend eingeschränkt, dass ein ein Zeichen mit der ganzzahligen Wertigkeit 0 (ein "NUL-Byte") die Zeichenkette beendet - in einem `std::string` können beliebige auch als eingebetteter Inhalt auftreten.

*: The C++98 standard left it to the library implementors whether to chose contiguous or non-contiguous storage though that freedom might be removed in a future version of the standard.

`std::string` als `const char *` verwenden

Dort woe ein C-API `const char *` erwartet, muss ein `std::string` entsprechend umgewandelt übergeben werden:

```
std::string filename;
… // get file name from user (or elsewhere)
// open file for reading, using the C-API
FILE *fp = std::fopen(filename.c_str(), "r");
```

Der obige Code ist korrekt und risikolos, da auf den von `c_str()` gelieferten Zeiger (bzw. der darüber erreichbare String-Inhalt) nur kurzzeitig innerhalb von `std::fopen` zugegriffen wird.

Risiken von `c_str()`

bei der Verwendung von `c_str()` sollte klar sein, dass hierüber Zugang zu dem für den `std::string`-Inhalt auf dem Heap angelegten Speicherplatz gewährt wird:

[!] Code wie der nachfolgende ist somit hochgradig riskant: der Zugriff auf `p` zeigt nach jeder Veränderung des Inhalts von `s` möglicherweise nicht mehr auf gültigen oder auf mittlerweile für andere Zwecke allozierten Speicherplatz.

```
std::string s("see me, ");
const char *p = s.c_str();
s += std::string("feel me,");
…
s.append(" touch me, hear me");
…
```

```
char *
mamamia(std::string arg) {
    …
    return arg.data();    // as
    // of C++11 same as c_str()
}
```

*: In the example on the right hand side, storing and then derefencing the pointer returned will access deallocated heap memory once owned by `arg` with near to 100% certainty.

`const char *` als `std::string` verwenden

Die Umwandlung eines klassischen C-Strings in ein `std::string`-Objekt ist durch einen entsprechenden, als Typumwandlung wirkenden (nicht expliziten) Konstruktor stets unproblematisch.

> **(i)** Sollen Funktionen sowohl mit (unveränderbaren) `std::string`-Argumenten als auch mit klassischen String-Literalen im C-Stil aufrufbar sein, besteht die sparsamte Lösung in der Verwendung des Argumenttyps `const std::string &`.

Eine - gemessen am zu schreibenden Code - deutlich aufwändigere Lösung ist es, Überladungen für

- `const char *`,
- `std::string &` und
- evtl. `std::string &&`

bereitzustellen. Andererseits kann jede Version damit für den übergebenen Parametertyp optimiert werden.

Effizienz

Typische Implementierungen der `std::string`-Klasse nutzen die folgenden Maßnahmen zur Effizienzsteigerung:

- Allozierung von Überschuss-Speicherplatz am Ende.
- Wenn nötig propotionale Vergrößerung* der Allozierung (nicht mit konstanten Faktor).
- Insbesondere auf 64-Bit Hardware lohnend: Small String Optimization

*: This means that when the current allocation doesn't suffice any more it will be doubled (or made 1.5 or 1.8 times as large). This gives O(1) performance to algorithms that fill a very long character string by appending single characters to the end, while increasing the allocation by a fixed amount would yield $O(N^2)$ performance.

Copy On Write

Durch diese Optimierung lässt sich insbesondere Code "verbessern", welcher häufig `std::string`s als Wertargument übergibt, obwohl eine konstante Referenz eher angemessen wäre.

- Das Kopieren des eigentliche Inhalts eines `std::string` wird dabei verzögert.
- Stattdessen wird ein "Merker" gesetzt, dass das Kopieren beim ersten schreibenden Zugriff nachgeholt werden muss.
- Bis dahin teilen sich mehrere "Nutzer" den Inhat beim nur lesenden Zugriff.

Copy On Write (COW) Implementierungen, sind mittlerweile weniger häufig, insbesondere in multi-threaded Ablaufumgebungen, da dort oft die möglichen Performance-Vorteile durch gelegentlich ersparte Kopien geringer zu Buche schlagen als der Nachteil, alle Zugriffe auf den String-Inhalt durch geeignete Mechanismen koordinieren zu müssen.

Grundlegende Operationen

Soweit diese nicht ohnehin intuitiv verständlich sind, wie

- Zuweisung mit `=`,
- Vergleich mit `==`, `!=` usw.
- Verkettung mit `+` sowie
- Elementzugriff mit `[…]`

stellen sie üblicherweise keine hohe Hürde dar.

Eine Überlegung zum Programmierstil beim Element-Zugriff könnte sein, diesen in nicht performance-kritischem Code konsequent mit der Member-Funktion `at()` vorzunehmen, um undefiertes Verhalten bei Bereichsüberschreitungen zuverlässig zu vermeiden.

Weitere Operationen im Überblick

Hierfür sei an dieser Stelle auf die - beabsichtigte - Ähnlichkeit zwischen `std::string` und `std::vector<char>` verwiesen:

- Auch ein `std::string` bietet die übliche Iterator-Schnittstelle.
- Insofern sind neben den speziellen `std::string` Member-Funktionen auch alle STL-Algorithmen anwendbar.

Was jeweils zu gut verständlichen Code führt, hängt vom Einzelfall ab:

```
std::string s;
…
// process lines not only containing horizontal white-space:
if (std::getline(std::cin, s)
 && s.find_first_not_of(" \t") == std::string::npos)) …

// or alternatively:
if (std::getline(std::cin, s)
 && std::all_of(s.begin(), s.end(),
                [](char c) { return (c == ' ' || c == '\t'); })) …
```

Umwandlung von/in arithmetische Werte

Die Umwandlung zwischen Zeichenketten und arithmetische Werte in interner Darstellung (`int`, `unsigned`, `long` ... `double`) gehört zu den häufig zu lösenden Aufgaben.

Vielfach findet man hier noch sehr umständlichen, unzureichenden oder teils sogar gefährlichen Code:

```
std::string tmpfilename; // fixed part followed by sequence number
…
const char *cp = tmpfilename.c_str();
while (*cp && !std::isdigit(*cp)) ++cp; // locate number
int num = atoi(cp);                     // get value into some int
std::sprintf(cp, "%d", ++num);          // store back incremented
```

`std::string` in arithmetischen Wert umwandeln

C++11 hat zu diesem Zweck eine Reihe neuer Funktionen eingeführt:

- Das Namensschema ist `std::sto…` für *String To"
- gefolgt von einem charakteristischen Buchstaben oder einer Kombination von Buchstaben:
  - `stoi` für Umwandlung nach `int`
  - `stol` für Umwandlung nach `long`
  - `stoul` für Umwandlung nach `unsigned long`
  - `stoll` für Umwandlung nach `long long`
  - `stoull` für Umwandlung nach `unsigned long long`
  - `stof` für Umwandlung nach `float`
  - `stod` für Umwandlung nach `double`
  - `stold` für Umwandlung nach `long double`

Die Parametrisierung ist ähnlich zu den C-Funktionen `strtol`, `strtoul` und `strtof`, d.h. über ein weiteres (Zeiger-) Argument kann die Position des Zeichens ermittelt werden, bei dem die Umwandlung beendet wurde, und ein drittes Argument legt die Basis des Zahlensystems (2..36) fest.

Arithmetischen Werte in `std::string` umwandeln

Für die Umwandlung von numerischen Werten in `std::string` führte C++11 die Funktion `std::to_string` mit zahlreichen Überladungen ein.

Ihre Anwendung ist trivial und (zusammen mit der auf der vorherigen Seite eingeführten Funktion `std::stoull` aus folgendem Beispiel ersichtlich:

```
std::string tmpfilename; // fixed part followed by sequence number
…
const auto n1 = tmpfilename.find_first_of("0123456789");
assert(n1 != std::string::npos);
const auto n2 = tmpfilename.find_first_not_of("0123456789", n1+1);
std::size_t nx;
const auto num = std::stou(tmpfilename.substr(n1, n2), &nx);
assert(nx == n2-n1);
tmpfilename = tmpfilename.substr(0, n1)
            + std::to_string(num+1)
            + tmpfilename.substr(n2);
```

Ein- und Ausgabe

Die Ausgabe von Zeichenketten erfolgt üblicherweise mit dem überladenene Ausgabeoperator:

```
std::string greet{"hello, world"};
…
std::cout << greet;
```

Lesen mit `operator>>`

Der für `std::string` überladene `operator>>` liest wortweise:

```
std::string word;
while (std::cin >> word) …
```

Als Trennung zwischen den Worten gelten hier beliebig lange Leerraum-Folgen (White Spacea, üblicherweise (mindestens) die Zeichen:

- Zeilenvorschub (`'\n'`)
- Leerzeichen (`' '`) sowie
- horizontale und vertikale Tabulatoren (`'\t'` und `'\v'`).

Beim Lesen von `std::string`-s mit `operator>>` können in der Regel keine Leerzeilen erkannt (und speziell verarbeitet) werden, da alle Zeilenvorschübe im Rahmen des Überspringens von White Space stillschweigend verworfen werden.

Lesen mit `std::getline`

Eingabe können auch zeilenweise in einen `std::string` gelesen werden

```
std::string line;
… std::getline(std::cin, line) …
```

oder bis zu einem beliebigen Begrenzer:

```
std::string field;
… std::getline(std::cin, field, ':') …
```

> Sehr flexibel ist das obige Verfahren jedoch nicht, da stets nur **genau ein** Begrenzerzeichen vorgegeben werden kann.

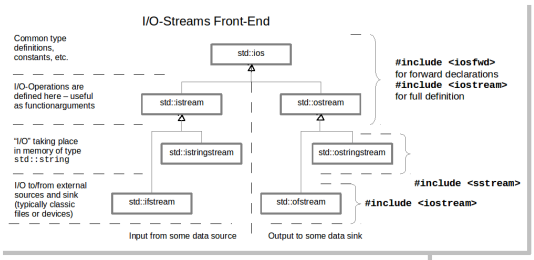
Eine Zeichenauswahl - z.B. Punkt, Komma oder Semikolon - ist nicht möglich.*

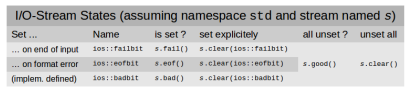
*: Though a small helper function accepting a delimiter sets shouldn't be that hard to write ...

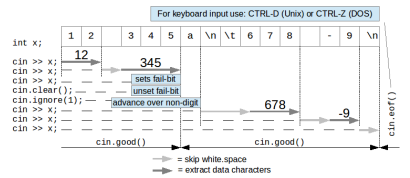
Verwendung von IO-Streams

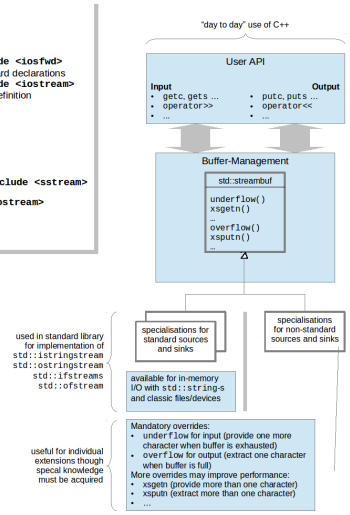
- Front-End und ...
- ... Back-End

- Zustands-Bits









I/O-Streams State-Bits

I/O-Stream Basics

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

I/O-Streams Back-End

Front-End der I/O-Streams

Das Front-End der I/O-Streams besteht aus

- der Basisklasse `std::ios`* mit einigen allgemeinen Definitionen
- davon abgeleitet die Klassen `std::istream` und `std::ostream`, welche vor allem in Form von Referenzargumenten zur Parametrisierung von I/O-Strömen als Funktionsargumente verwendet werden,
- sowie als Klassen, von denen auch Objekte angelegt werden
  - `std::ifstream`, `std::ofstream` und `std::fstream` (File-Streams) und
  - `std::istringstream`, `std::ostringstream`, und `std::stringstream` (String-Streams).

*: As with `std::string` the architecture is even more generic as the "classes" explained above are rather `typedef`s for more generic template classes, each of which is parametrized in a character type and some more aspects. This fact need not be made prominently visible if - as it is the case here - the focus is to explain the relation between the classes participating in the design.

Gemeinsame Schnittstelle

Die von einer Applikation verwendbaren Operationen zur Ein- und Ausgabe liegen teils als Member der Klassen `std::istream` und `std::ostream` vor, teils sind es globale Funktionen.

> Zur Unterstützung benutzerdefinierter Datentypen können auch weitere globale Überladungen von `operator>>` und `operator<<` existieren.

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_ios
- http://en.cppreference.com/w/cpp/io/basic_istream
- http://en.cppreference.com/w/cpp/io/basic_ostream

Filestreams

Die Verwendung erfolgt typischerweise abhängig von der gewünschten Richtung des Datenstroms:

- `std::ifstream` zum Lesen
- `std::ofstream` zum Schreiben

Darüberhinaus gibt es auch bidirektional verwendbare Filestreams:

- `std::fstream`

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_fstream
- http://en.cppreference.com/w/cpp/io/basic_ifstream
- http://en.cppreference.com/w/cpp/io/basic_ofstream

Stringstreams

Die Verwendung erfolgt typischerweise abhängig von der gewünschten Richtung des Datenstroms:

- `std::istringstream` zum Lesen
- `std::ostringstream` zum Schreiben

Darüberhinaus gibt es auch bidirektional verwendbare Stringstreams:

- `std::stringstream`

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_stringstream
- http://en.cppreference.com/w/cpp/io/basic_istringstream
- http://en.cppreference.com/w/cpp/io/basic_ostringstream

Back-End der I/O-Streams

Das Backend der I/O-Streams implementiert vor allem aus einem Mechanimus zur Datenpufferung.

> Damit kann insbesondere die Übertragung von Daten in Richtung von oder zu einem permanenten Speicher in Blockgrößen erfolgen kann, welche die lesende oder schreibende Applikation selbst nicht weiter berücksichtigen muss.

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_streambuf
- http://en.cppreference.com/w/cpp/io/basic_filebuf
- http://en.cppreference.com/w/cpp/io/basic_stringbuf

Zustands-Bits der I/O-Streams

Jeder Stream besitzt eine Reihe von Zustandsbits.

- So lange keines davon gesetzt ist, befindet sich der Stream im "good"-Zustand.
- Bei im Rahmen der Eingabe eines bestimmten Datentyps unerwarteten - also nicht zu verarbeitenden - Zeichen, wird das `std::ios::badbit`-Bit gesetzt.
- Tritt im Rahmen der Eingabe die *End-Of-File*-Bedingung ein, wird das `std::ios::eofbit` gesetzt.
- Bei anderen (vom Standard nicht näher spezifierten) Fehlerbedinungen kann auch das `std::ios::badbit` gesetzt werden.

Verhalten der I/O-Streams abhängig vom Zustand

Solange eines der genannten Bits gesetzt ist, werden von dem betreffenden Stream alle Operationen ignoriert, **ausgenommen `clear()` und `close()`**.

Dies ist besonders bei der Verarbeitung von Eingaben wichtig:

- Beim Wechsel des Zustands bleibt die aktuelle Position - also welches Zeichen als nächstes im Rahmen einer Eingabe verarbeitet wird - so, wie sie beim **Auslösen** des Zustandswechsels war.
- Bei einem Fehler im Eingabeformat, z.B. wenn ein Buchstabe erscheint, wo eine Ziffer erwartet wird, ist das nächste Zeichen immer noch der (störende) Buchstabe.
- Soll (mindestens) dieses Zeichen übersprungen werden, so muss
  - **Zuerst** der Stream in den "good"-Zustand versetzt werden, und
  - **erst danach** kann eine Operation (wie z.B. `ignore` zum Überspringen von Zeichen) ihre Wirkung entfalten.

Exceptions bei Zustanswechsel

Wahlweise kann das Verhalten eines Streams so eingestellt werden, dass

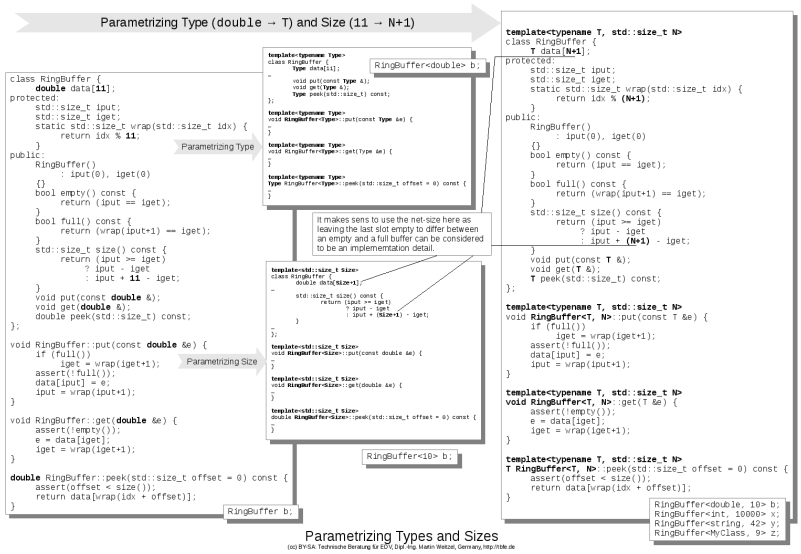
- bei jedem Zustanswechsel und
- allen (versuchten) Operationen außerhalb des "good"-Zustands

eine Exception geworfen wird:

```
// Excerpt from a hypothetical "forever running" TCP-Client
std::ifstream from_server;
… // somehow establish connection through TCP/IP-Socket
from_server.exceptions(std::ios::badbit
                     | std::ios::eofbit
                     | std::ios::failbit);
try {
    for (;;) {
        std::string command_string;
        std::getline(from_server, command_string);
        … // process command_string
    } /*notreached*/
}
catch (std::ios_base::failure &e) {
    … // socket connection closed and/or data transfer failed
}
```

Grundlegendes zu Templates

- Parametrisierte Typen und ...
- ... Compilezeit-Konstanten ...
- ... am Beispiel einer `RingBuffer`-Klasse



Parametrisierte Typen

Die ursprüngliche Absicht bei der Einführung des Template-Mechanismus in C++ war es, sich wiederholenden Code zu vermeiden, wenn es in den verschiedenen Varianten einer Klasse oder (Member) Funktion lediglich um andere Datentypen geht.

Hierzu ist

- die Funktion oder Klasse mit einer formalen Typ-Parameterliste in spitzen Klammern zu versehen,
- in welcher dem Compiler **symbolische Namen** für die parametrisierten Typen angekündigt werden.

Die symbolischen Namen können in der nachfolgenden Implementierung der Klasse oder Funktion überall dort verwendet werden, wo syntaktisch ein Datentyp stehen kann.

Bei der späteren Verwendung der Template - auch Instanziierung genannt - sind in den spitzen Klammern konkrete Typen einzutragen.

Parametrisierte Compilezeit-Konstanten

Im Rahmen einer Template-Klasse oder -Funktion können nicht nur Typen sondern auch Compilezeit-Konstanten parametrisiert werden.

Hierzu ist

- die Funktion oder Klasse mit einer formalen Wert-Parameterliste in spitzen Klammern zu versehen,
- in welcher dem Compiler **Typen und symbolische Namen** der parametrisierten Compilezeit Konstanten angekündigt werden.

Die symbolischen Namen können in der nachfolgenden in der Implementierung der Klasse oder Funktion überall dort verwendet werden, wo syntaktisch eine Compilezeit-Konstante des betreffenden Typs stehen kann.

Bei der späteren Verwendung der Template - auch Instanziierung genannt - sind in den spitzen Klammern konkrete Compilezeit-Konstanten einzutragen.

Beispiel `RingBuffer`

Die Erweiterung der zunächst für einen bestimmten Datentyp und eine bestimmte Größe implementierten `RingBuffer`-Klasse zu einer Template kann mehr oder weniger "mechanisch" durch "Suchen und Ersetzen" erfolgen:

- Das wird dadurch begünstigt, dass im ursprünglichen Code der Datentyp `double` nur dort auftritt, wo in der Template-Variante der parametrisierte Typ steht.
- Die Konstante `11` steht im ursprünglichen Code für die Anzahl der im `RingBuffer` maximal ablegbaren Elemente **Plus Eins**.*
  - In der Template-Variante erscheint es sinnvoll, als Parameter die Nettogröße (also die maximale Anzahl der Elemente) anzugeben, die der `RingBuffer` zwischenspeichern soll.
  - Dies ist leicht realisierbar, indem die Konstante `11` jeweils durch `N+1` ersetzt wird.

---

*: So that "empty" and "full" state can be easily discerned without an additional flag, the buffer never gets completely filled but a single element is always left unused, if the position into which to "put" is directly behind the position from which to "get".