C++ FOR (Montagvormittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt 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

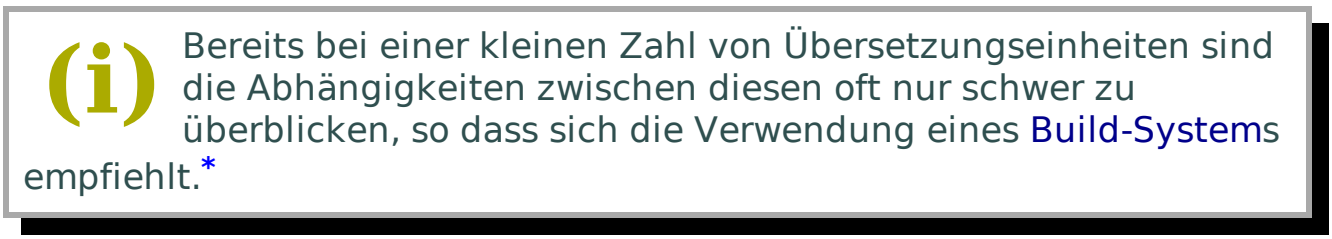
- Automatische Typ-Konvertierung
- Typ-Konvertierung mittels *Cast*
- Klassenspezifische Typ-Konvertierung

Getrennte Kompilierung (1)

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 Informationen in mehr als einer Übersetzungseinheit notwendig, stehen diese in *Header-Files*, deren Dateinamens-Suffix meist `.h` (seltener: `.hpp`) ist.



*: 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, insbesondere in Form moderner Derivate wie etwa GNU make und CMake.

Getrennte Kompilierung (2)

Abhängigkeiten zwischen Header-Files

Häufig 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 {
 // ...
};
```

Getrennte Kompilierung (3)

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
```

Getrennte Kompilierung (4)

Namespaces

[!] Wie man am folgenden Beispiel leicht erkennt, ist der pure Name einer in einem Header-File definierten Klasse als Include Guard nicht unbedingt ausreichend.

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

```
#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
```

Getrennte Kompilierung (5)

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 hier, sowie eine fehlerbereinigte Version hier.

Definition/Reference-Modell

Hiermit ist gemeint, dass es zu jeder in einem Progamm vorhandenen Variablen (Grundttyp oder Objekt-Instanz) **genau eine** Definition gibt, möglicherweise aber viele Bezugnahmen in anderen Übersetzungseinheiten.

- Im Fall der Bezugnahme auf eine Definition in einer anderen Übersetzungseinheit muss die bezugnehmende Übersetzungseinheit eine `extern`-Deklaration vornehmen.
- Eine Definition *darf* ggf. auch das Wort `extern` enthalten, sie muss es sogar, wenn zugleich eine `const`-Qualifizierung verwendet wird.

[!] Die Verwendung von `const` auf globaler Ebene beinhaltet - vielleicht überraschender Weise - zugleich den Sichbarkeitsschutz gegenüber dem Linker. Das damit seinerzeit verfolgte Ziel war eine möglichst einfache Umstellung von `#define`-s auf `const`.

Schreibschutz durch den Compiler (const)

Mittels const-Qualifizierung kann die Zuweisung an eine Variable verboten, es ist nur noch die Initialisierung im Rahmen der Vereinbarung 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, werden mit const-qualifizierte Variablen eventuell auch mit einem physikalischen Schreibschutz ausgestattet.

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 das, was darüber 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 Art von Zeigern, welche bei ihrer Verwendung automatisch dereferenziert werden (implizite *-Operation).
- Alias-Name für einen 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 zum Zugriff auf das, was darüber referenziert wird.

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 = …;           // … = modifiable T in memory
const T &cr = …;    // … = modifiable T in memory OR
                    //     non-modifiable T in memory OR
                    //     temporary T in memory (expression)
T &&rr = …;         // … = 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, oder anders ausgedrückt:
- Sobald ein Argument einen Default-Wert hat, müssen die weiter rechts stehenden ebenfalls einen Default-Wert haben.

Der Defaultwert muss beim Funktionsaufruf bekannt sein und ist insofern Bestandteil des Funktions-Prototyps, steht also ggf. bei der Deklaration im Header-File.

```
double foo(int &, char = 'z');
```

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 der Compiler nun, was verwendet wird:

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

Überladen von Funktionen (2)

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, eine für den konstanten und eine für den nicht-konstanten Fall:

- Ohne die Funktion für den **nicht**-konstanten Fall würde die Funktion für den konstanten Fall für alle Aufrufe verwendet worden.*
- Ohne die Funktion für den **konstanten** Fall wäre der Aufruf für ein String-Literal ein Compile-Fehler, da dessen Typ `const char *` ist.

---

*: Generell gesehen ist es kein Problem, wenn einer Funktion, die versprochen hat, über ein Zeiger- oder Referenz-Argument erreichbaren Speicherplatz nicht zu verändern, nun doch die Addresse von veränderbaren Speichreplatz bekommt - **umgekehrt wäre 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 aber natürlich vom Typ her passend 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 - ungegeignet ist.*

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

*: Der klassiche Indikator sind in der Klasse enthaltene Zeiger auf Speicherplatz, welcher individuell pro Objekt vorhanden sein muss.

Überladung von Move-Konstruktor und -Zuweisung

In C++11 können die RValue-Referenzen dazu verwendet werden, für Ausdrücke auf der rechten Seite das Kopieren und die Zuweisung anders zu implementieren als für den Fall Referenzen weiterhin existierender Vorlagen:

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

Überladung von Move-Konstruktor und -Zuweisung (2)

Und dann z.B. weiter so:

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

Die Zuweisungen sind ähnlich, müssen aber zunächst den Speicherplatz freigeben, auf den `some_ptr` verweist.

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 oder 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 in C++ erzwingen.

> **[!]** Die C-Syntax, bei welcher man den neuen Typ in runde Klammern setzt und den umzuwandelnden Wert dahinter, 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 beiden Richtungen für Umwandlungen, welche als automatische Umwandlung in einer der beiden Richtungen eine Einbahnstraße sind.

- Arithmetische Wert zu `enum`-s
- Generische Zeiger (`void *`) zu typisierten Zeigern
- Basisklassen in abgeleitete Klassen (Downcast)

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/oder `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, 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 (teilweise alter Wert, teilweise neuer Wert) reichen.

Typ-Konvertierung mit `reinterpret_cast`

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

Darüber hinaus kann man auch mit einem `reinterpret_cast`

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

> Die per `reinterpret_cast` gebotene Möglichkeit ein Bitmuster im Speicher gemäß einem beliebigen Typ zu interpretieren, veranlassen Kritiker von C++ immer wieder zu der Aussage, die Sprache sei extrem 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 im Unterschied zu C viel 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.

Sinngemäß kann man es sich so vorstellen:

- Eine Klasse stellt eine Art "charakteristische Steckverbindung dar, die zunächst nur zu sich selbst passt.
- Entsprechend kann man in Initialisierungen und Zuweisungen nur Objekte derselben Klasse verwenden.
- Konstruktoren mit genau einem Argument stellen zugängliche "Eingänge" dar, also den Typ der (Ausgangs-) Steckverbinder aufnehmen können, welche einer anderen Klasse entprechen.
- Sogenannte Typ-Cast-Operatoren stellen umgekehrt zusätzliche "Ausgänge" dar, die den Typ des (Eingangs-) Steckverbinders aufnehmen können, welcher einer anderen Klasse entspricht.

Typ-Konvertierungen 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
 …
};
```

Typ-Konvertierung durch Konstruktoren (2)

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
…
```

Typ-Konvertierungen durch Type-Cast Operationen

Type-Cast Operationen benutzen eine spezielle Syntax, bei welchen **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.

Typ-Konvertierungen durch Type-Cast Operationen (2)

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
```

Typ-Sicherheit in C++

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

> Alle Formen expliziter Typumwandlung sollten auf das absolut notwendige Minimum beschränkt werden.

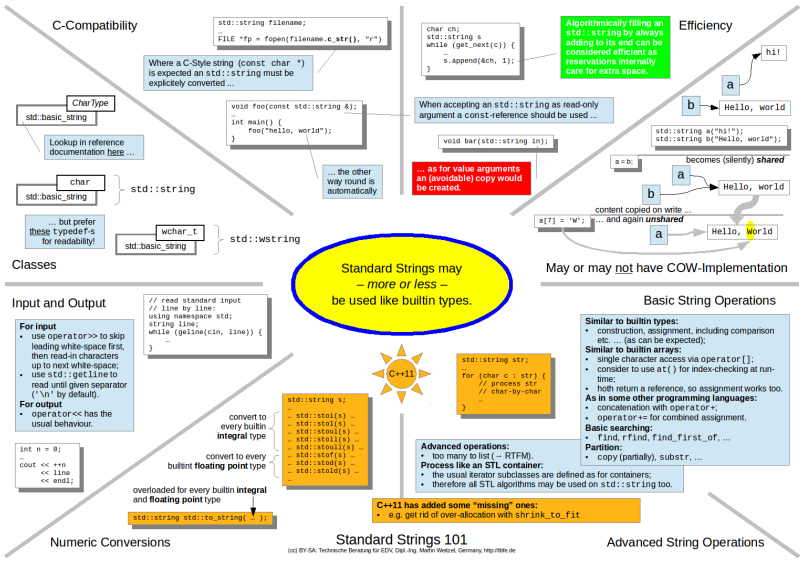
Darüberhinaus werden auch klassenspezifische Typumwandlungen manchmal in nicht erwarteter Weise angewendet:

> Klassenspezifische Typumwandlung sollten nur sparsam vorgesehen werden und ganz generell nur dort, wo der stillschweigende Wechsel zwischen den beteiligten Typen als "natürlich" empfunden wird.

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



Klassen (Übersicht)

Nachzuschlagen in:

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

Kompatibilität zu C

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Effizienz

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Effifienz

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Grundlegende Operationen

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Weitere Operationen im Überblick

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Umwandlung von/in arithmetische Werte

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Arithmetischen Werte in Standard-Strings umwandeln

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

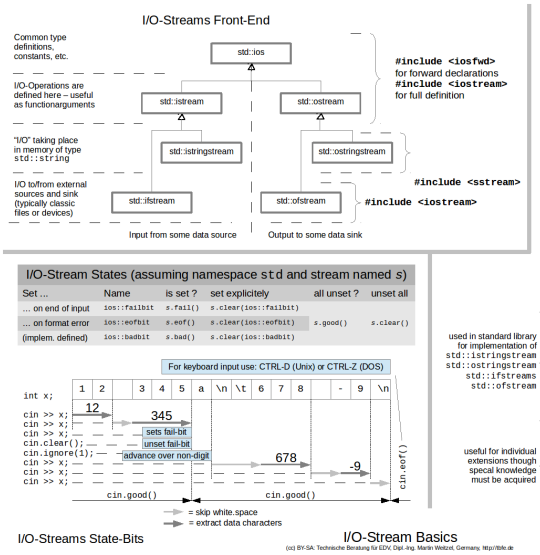
Standard-String in arithmetischen Werte umwandeln

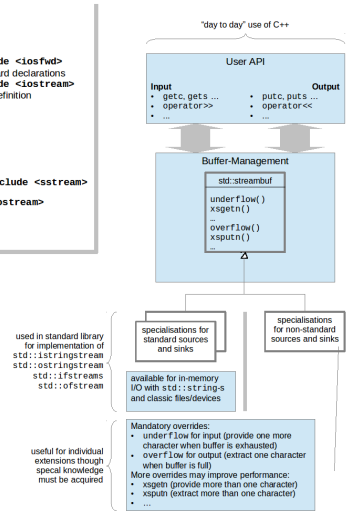
Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

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

Back-End der I/O-Streams

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Front-End der I/O-Streams

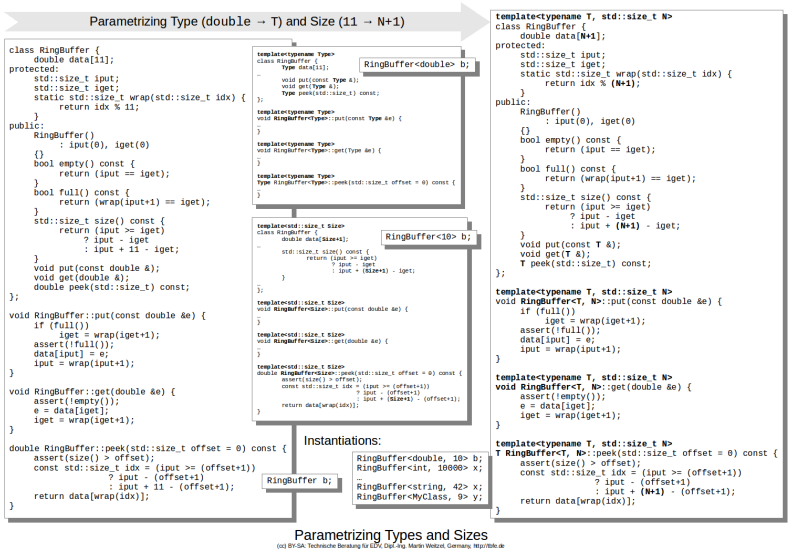
Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Zustands-Bits der I/O-Streams

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Grundlegendes zu Templates

- Template-Klasse
- Template-Funktion



Template-Klassen

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Template-Funktion

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Übung

Ziel ist die Parametrisierung einer einfachen Klasse in einem Datentyp und der Größe eines Arrays.

Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.