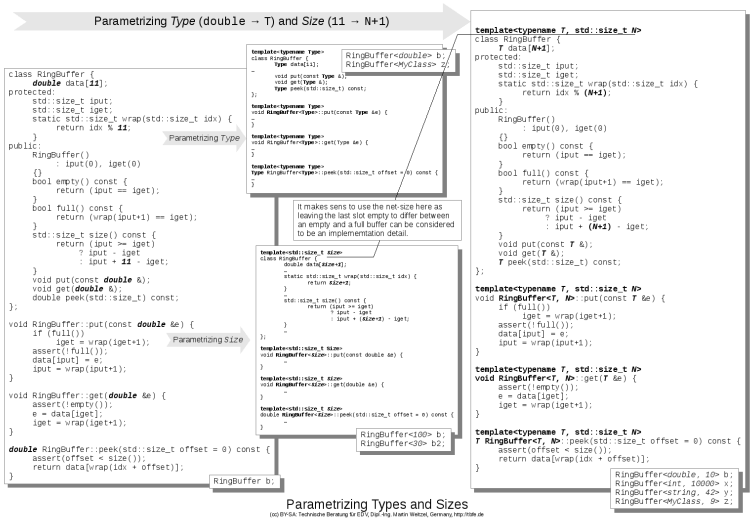
C++ FOR
(Monday Afternoon)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Template Basics

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



Parametrisierte Typen

Ursprünglich sollte mit den C++-Templates vor allem weitgehend identischer Code reduziert werden, wenn es in verschiedenen Varianten einer Klasse oder (Member) Funktion nur um andere Datentypen geht.

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

Diese Namen können in der nachfolgenden Implementierung der Klasse oder Funktion überall dort verwendet werden, wo syntaktisch ein Typ stehen kann bzw. muss.

Bei der Instanziierung der Template sind in den spitzen Klammern an der entsprechenden Stelle der Parameterliste konkrete Typen einzutragen.

*: Each name is preceeded by the keyword `class` or `typename`, which may be used interchangeably with same meaning here. (But note that the two keywords have different meanings elsewhere.)

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.

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

Bei der Instanziierung der Template sind in den spitzen Klammern an der entsprechenden Stelle der Parameterliste Compilezeit-Konstanten des betreffenden Typs einzutragen.*

* The usual automatic type converions take place as necessary – e.g. between arithmetic types.

Beispiel `RingBuffer`

Die Umwandlung der zunächst für einen bestimmten Datentyp und eine feste Größe implementierten `RingBuffer`-Klasse in eine Template kann größtenteils durch systematisches "Suchen und Ersetzen" erfolgen:

- Besonders günstig ist, dass der Datentyp `double` nur dort auftritt, wo in der Template-Variante der parametrisierte Typ stehen muss.
- 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, die Nettogröße (also die maximale Anzahl der Elemente) anzugeben, die der `RingBuffer` tatsächlich aufnehmen kann.
  - Dies ist leicht realisierbar, indem bei einem Größen-Parameter `N` die Konstante `11` jeweils durch `N+1` ersetzt wird – in Ausdrücken ggf. in Klammern zur Sicherstellung des Operator-Vorrangs.

---

*: So that the *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".

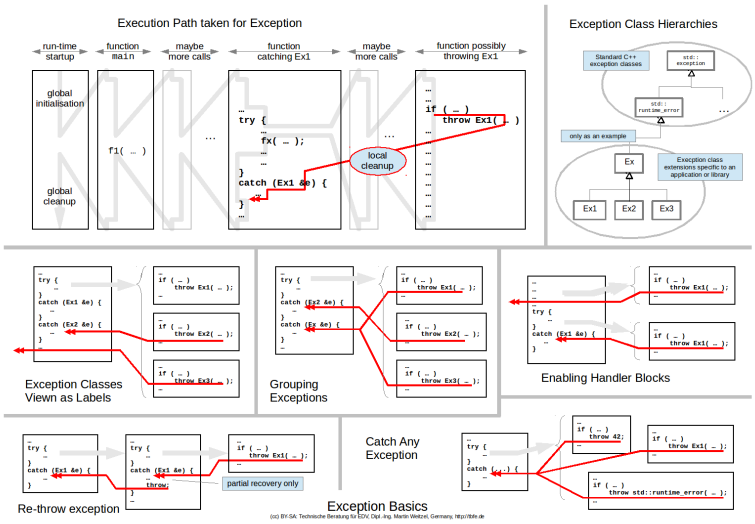
Exception Basics

- Hierarchien von Exception-Klassen

- Kontrollfluss mit und ohne Exceptions

- Exception Klassen als Label verstanden
- Gruppieren ähnlicher Exceptions
- Aktivieren der Wiedereintrittspunkte

- Unvollständig behandelte Exceptions
- Fangen aller Exceptions



Hierarchien von Exception-Klassen

Die im Rahmen von Bibliotheksfuntionen ggf. geworfenen Exceptions bilden eine Klassenhierarchie:

- An deren Spitze steht die Klasse `std::exception`.
- Davon abgeleitete Klassen sind oft als Basisklassen für eigene Exception-Klassen sinnvoll, etwa
  - `std::logic_error` oder
  - `std::runtime_error`.

Click here for Info-Graphic
ExceptionBasics
Execution Path taken for Exception

Kontrollfluss mit und ohne Exception

Solange keine Exceptions geworfen werden, folgt der Kontrollfluss den üblichen Regeln.

> Erreicht der Kontrollfluss das Ende eines `try`-Block, ohne dass eine Exception geworfen wurde, werden alle nachfolgenden `catch`-Blöcke übersprungen.

Insoweit kann das Verhalten analog zu einem Block nach `if` sehen, bei dem ebenso alle nachfolgenden `else if` und das abschließende `else` übersprungen werden, wenn die Auswertung der Bedingung `true` ergab.

Exception Klassen als Label verstanden

Sobald jedoch eine `throw`-Anweisung ausgeführt wird, stellen die den aktiven `try`-Blöcken nachfolgenden `catch`-Blöcke eine Art Label dar:

> Verzweigt wird ausgehend vom `throw` grundsätzlich im Kontrollfluss rückwärts, d.h. in Richtung auf die `main`-Funktion.

- Die Auswahl der `catch`-Blöcke erfolgt dabei
  - gemäß der dynamischen Schachtelung der Funktionsaufrufe, und
  - und innerhalb der einzelnen `catch`-Blöcke nach einem aktiven `try`-Block von oben nach unten.
- Es wird der erste `catch`-Block ausgewählt, bei dem der Typ dessen Typ zur geworfenen Exception passt.
- Gibt es keinen solchen, geht zum nächsten, gemäß der dynamischen Schachtelung folgenden `try`-Block.
- Wird dabei auch die `main`-Funktion verlassen, bricht das Programm ab.

Gruppieren ähnlicher Exceptions

Bei der Auswahl des `catch`-Blocks werden in Bezug auf die geworfene Exception prinzipiell die selben automatischen Typ-Konvertierungen berücksichtigt wie bei der Übergabe von Parametern an Funktionen.

- Insbesondere gilt das LSP, d.h. abgeleitete Klassen passen zu ihren jeweiligen Basisklassen.
- Somit lassen sich Klassenhierarchien bilden, um ähnliche Exceptions optional in einem gemeinsamen `catch`-Block fangen zu können.

Auch wenn die Klammern nach `catch` dem Konstrukt insgesamt das Aussehen einer Argumentliste geben, stellt die Verzweigung des Kontrollflusses in einen `catch`-Block **keinen Funktionsaufruf** dar.*

*: It is rather some kind of special return into the control flow of the (still) active function with the (once) active `try`-Block. But there are even more similartities to a function and a parameter specification, not only that the usual type conversions take place, but also with respect to `const` and value or reference access to the exception object thrown. Finally, two requirements imposed syntactically are that there must always be exactly one "argument" and that a `catch`-block must always be written as block, so even if it contains exactly one statement the curly braces may not be omitted.

Aktivieren der Wiedereintrittspunkte

Ein `try`-Block wird aktiv, sobald der Kontrollfluss die erste enthaltene Anweisung erreicht und deren Ausführung beginnt.

> Als Sprungziel in Betracht gezogen werden stets nur diejenigen `catch`-Blöcke, deren vorangehender `try`-Block aktiv ist.

Ein `try`-Block ist nicht mehr aktiv, wenn er explizit verlassen wird durch

- `return`*
- `break`
- `continue`

bzw. wenn die letzte enthaltene Anweisung vollständig ausgeführt wurde.

> Anschließend werden die nachfolgenden `catch`-Blöcke nicht mehr als mögliche Wiedereintrittspunkte nach einem `throw` in Betracht gezogen.

*: If a value is returned, evaluation of an expression may be part of the return. The evaluation itself takes place while technically still in the active `try`-block. But when the function has returned and its return value is only used – say in another copy c'tor outside the function that has returned – the function's `try`-block is not active any more and hence the `catch`-blocks following it are not any more possible targets.

Unvollständig behandelte Exceptions

Catch-Blöcke, welche nur eine teilweise Auflösung der Fehlersituation leisten, können die gefangene Exception erneut werfen:

```
    try {
        …
        … // code that may throw SomeException (no matter if
        … // directly, or indirectly from a function called)
        …
    }
    catch (SomeException &ex) {
        …
        … // (assuming partial recovery only)
        …
        throw;
    }
}
```

Fangen aller Exceptions

Es besteht die Möglichkeit, einen `catch`-Block für alle erdenklichen Exception "passend" zu machen:

- Hierzu sind in den runden Klammern drei Punkte (analog zu variadischen Funktionen) anzugeben.
- Ein solcher Block muss ggf. immer am Ende aller `catch`-Blöcke eines `try`-Block stehen.*

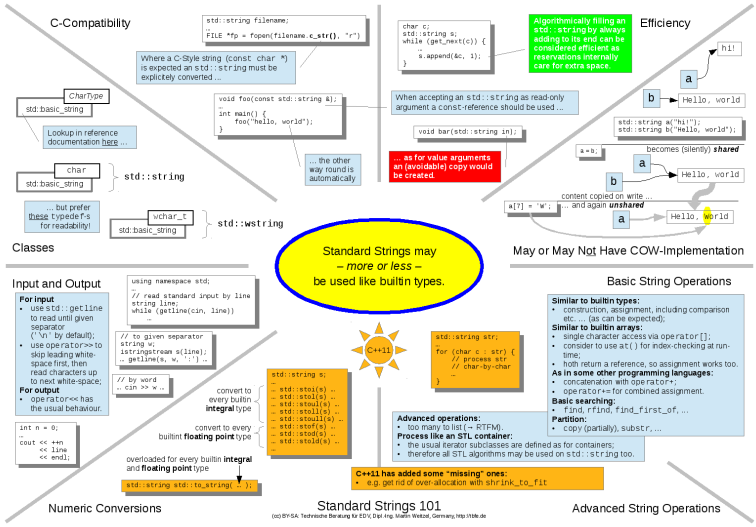
```
int main() {
    try {
        …
        …
    }
    catch (...) {
        std::cerr << "!! unhandled exception !!\n";
    }
}
```

*: This is not syntactically enforced but as "..." catches any exception and the `catch`-blocks are considered top down, it would otherwise catch anything for which a specific `catch`-block is following.

Library-Basics – Strings

- Klassen (Übersicht)
- Kompatibilität zu C
- Effizienz-Betrachtungen
- 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 ähnlich den folgenden:*

```
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 of the truth: other template arguments are a character traits class and an allocator (memory management policy). Both have been omitted in the example 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 zumindest in der erforderlichen Größe, oft aber auch mit etwas Reserve auf dem Heap angelegt wird.

Ein Objekt der `std::string`-Klasse enthält **direkt** typischerweise drei Zeiger auf bzw. in ein Stück Heap-Speicher.

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

Insbsondere sind die als `std::string` Inhalt möglichen Zeichen nicht (wie in C) dahingehend eingeschränkt, dass ein Zeichen mit der ganzzahligen Wertigkeit 0 (ein "NUL-Byte") die Zeichenkette beendet.

*: 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

Wenn ein C-API `const char *` erwartet, muss ein `std::string`, der den Dateinamen enthält, entsprechend dieser Konvention 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()` zurückgegebenen Zeiger (bzw. den darüber erreichbare String-Inhalt)

- nur innerhalb von `std::fopen` zugegriffen werden kann,
- die Variable `filename` dabei nicht verändert wird, und
- somit auch **der zugehörige Heap-Speicher derselbe bleibt**.

*: If the mode to open the file came from an `std::string` too, of course it needs to be converted similarily like that: … `std::fopen( … , openmode.c_str())` …

Risiken von `c_str()`

ei Verwendung von `c_str()` oder dem seit C++11 gleichwertigen `data()` sollte klar sein, dass darüber Zugang zu dem für den `std::string`-Inhalt auf dem Heap angelegten Speicherplatz gegeben wird.

[!] Code wie der nachfolgende ist somit hochgradig riskant: der Zugriff auf `p` zeigt nach jeder Veränderung des Inhalts von `s` auf möglicherweise nicht mehr gültigen oder 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");
…
```

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

*: In the example on the right hand side, derefencing the pointer returned from the function will access deallocated heap memory (once owned by `local`) with near to 100% certainty.

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

Die Umwandlung eines klassischen C-Strings in ein `std::string`-Objekt erfolgt stets automatisch durch einen als Typ-Konvertierung wirksamen Konstruktor.

> **(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 *`,
- `const std::string &` und
- (evtl.) `std::string &&`

bereitzustellen.*

*: On the other hand, each version could then be optimized for its argument type.

Effizienz-Betrachtungen

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

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

*: Proportional here 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 long character string by appending single characters to the end. Increasing the allocation by a constant, fixed amount would yield much worse $O(N^2)$ performance.

Optionales "Copy On Write"

Durch diese Optimierung lässt sich insbesondere Code ohne weiteren Eingriff "nachbessern", welcher häufig `std::string`-s als Wertargument übergibt, wo eine konstante Referenz angemessen wäre.

- Das Kopieren des eigentlichen Inhalts wird dabei verzögert:*
  - Stattdessen wird ein "Merker" gesetzt, und
  - das Kopieren beim ggf. beim ersten schreibenden Zugriff nachgeholt.
- So lange nur lesend zugegriffen wird, teilen sich mehrere formale Kopien den Inhalt.

> Endet die Lebenszeit einer formalen Kopie, ohne dass es jemals zu einem schreibenden Zugriff kam, wurde das Kopieren gänzlich eingespart.

*: Copy On Write (COW) Implementierungen finden sich mittlerweile weniger häufig, insbesondere in multi-threaded Ablaufumgebungen, da ihre möglichen Performance-Vorteile (durch gelegentlich ersparte Kopien) dort oftmals geringer zu Buche schlagen als ihr Nachteil, alle Zugriffe auf den String-Inhalt durch geeignete Mechanismen (z.B. Semaphoren) aufwändig 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 und damit undefiniertes Verhalten bei Bereichsüberschreitungen zuverlässig zu vermeiden.

Weitere Operationen im Überblick

An dieser Stelle sei auf die Ähnlichkeit zwischen `std::string` und `std::vector<char>` verwiesen, insbesondere gilt:

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

Was jeweils als besser verständlicher Code empfunden wird, ist mitunter auch Ansichtssache.

Das folgende Fragment liest einen `std::string s` von Standard-Eingabe und testet vor der weiteren Verarbeitung, ob ausschließlich "White-Space" enthalten ist:

```
// with std::string member function
if (std::getline(std::cin, s)
 && s.find_first_not_of(" \t") == std::string::npos)) …

// with STL algorithm (and predicate specified as lambda)
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 arithmetischen 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
…
char *cp = const_cast<char*>(tmpfilename.c_str());
while (*cp && !std::isdigit(*cp))
    ++cp;                          // locate first digit
const int num = std::atoi(cp);  // convert digit sequence to int
std::sprintf(cp, "%d", num+1);  // and store back incremented by 1
```

*: If not obvious from reviewing the code, the fragment above has the following problems:
1. `std::atoi` converts up to the first non-numeric character only (hence a missing numeric part will not be recognized – though in some cases this might be rather a feature than a bug).
2. Some else's memory might be silently overwritten if at `cp` not enough space is available for storing `num` incremented.

`std::string` in numerischen Wert umwandeln

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

- Das Namensschema ist `std::stoXX` für string `to` mit einem
- Buchstaben-Code `XX` gemäß der nachfolgenden Tabelle:

Buchstaben-Code	Umwandlung nach	basierend auf
`i`	`int`	`std::strtol`
`l`	`long`	`std::strtol`
`ll`	`long long`	`std::strtoll`
`ul`	`unsigned long`	`std::stroul`
`ull`	`unsigned long long`	`std::strtoull`
`f`	`float`	`std::strtod`
`d`	`double`	`std::strtod`
`ld`	`long double`	`std::strtold`

Siehe auch:

- http://en.cppreference.com/w/cpp/string/basic_string/stol
- http://en.cppreference.com/w/cpp/string/basic_string/stoul
- http://en.cppreference.com/w/cpp/string/basic_string/stof

Numerischen Wert in `std::string` umwandeln

Für die Umwandlung von numerischen Werten in `std::string` führte C++11 die Funktion `std::to_string` mit einer Reihe von 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 überladenenen Ausgabeoperator:*

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

*: Of course, this is not a specific operator for output but an overload to the left-shift operator (as introduced in C), when the left-hand operand is an output stream. Nevertheless, especially when C is used oustide the realm of embedded programming, some call `operator<<` now *output operator*.

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 Space), 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`

Eingaben 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 nur **genau ein** Begrenzerzeichen vorgegeben werden kann.

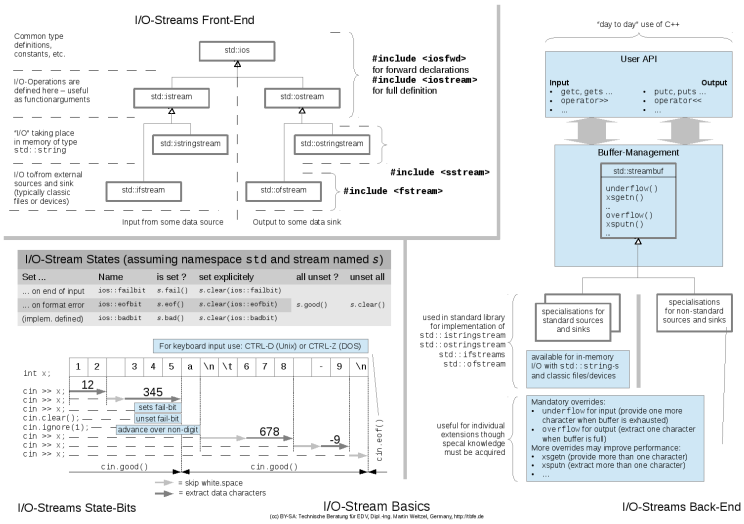
Eine Zeichenmenge – z.B. Punkt, Komma oder Semikolon – wäre oft wünschenswert, ist aber nicht möglich.*

*: While a small helper function accepting a set of delimiters shouldn't be that hard to write, this kind and much more sophisticated parsing of input patterns is possible with Regular Expressions. After having been available through Boost.Regex for a long time – regular expressions became part of the C++11 standard library (see http://en.cppreference.com/w/cpp/regex/basic_regex).

Library-Basics – IO-Streams

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

- Zustands-Bits



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 and the "classes" above are rather `typedef`-s for more generic template classes, which are parametrized not only in a character type but also in some other respects. This fact need not be made prominently visible if the focus is on explaining the relationship between the classes participating in the design – as it is the case here.

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 einen Mechanimus zur Datenpufferung.

Damit kann insbesondere

- die Übertragung von Daten in Richtung von oder zu einem permanenten Speicher in optimierten Blockgrößen erfolgen,
- welche die lesende bzw. schreibende Applikation weder kennen noch in irgend einer anderen Weise 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.

- Ist keines 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::failbit` 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 – Fehlerbedingungen kann auch das `std::ios::badbit` gesetzt werden.*

*: The usual difference between setting the *fail*- or *bad*-bit is that in the latter case there is often no (portable) way to recover, while in the former any unexpected input causing the state-switch might simply be skipped.

Verhalten der I/O-Streams abhängig vom Zustand

Solange eines der genannten Bits gesetzt ist, werden vom betreffenden Stream alle Operationen ignoriert, **ausgenommen `clear()` und `close()`**, was insbesondere bei der Verarbeitung fehlerhafter Eingaben wichtig ist:

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

Siehe auch:

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

Exceptions bei Zustandswechsel

Wahlweise kann das Verhalten eines Streams so eingestellt werden, dass

- bei jedem Zustandswechsel 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
}
```

Praktikum