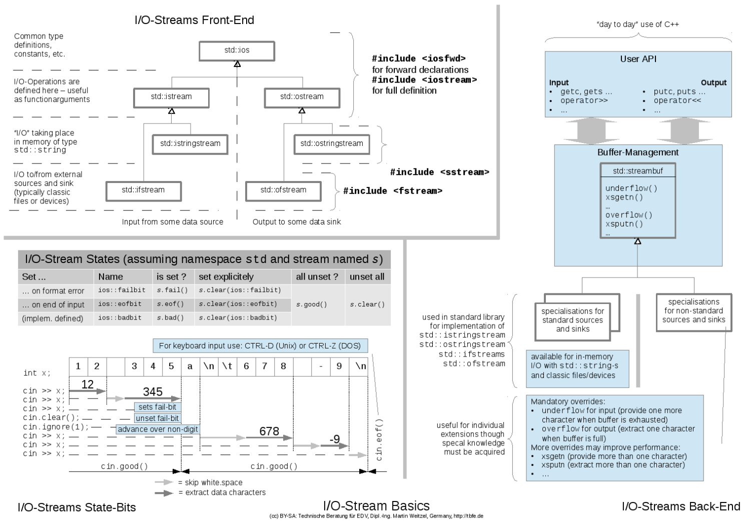
I/O-Streams

Library-Basics – I/O-Streams

- Front-End and …
- … Back-End

- I/O-State (Bits)



I/O-Stream States (assuming namespace `std` and stream named *s*)

Set …	Name	is set ?	set explicitely	all unset ?	unset all
… on format error	`ios::failbit`	`s.fail()`	`s.clear(ios::failbit)`		
… on end of input	`ios::eofbit`	`s.eof()`	`s.clear(ios::eofbit)`	`s.good()`	`s.clear()`
(implem. defined)	`ios::badbit`	`s.bad()`	`s.clear(ios::badbit)`		

I/O-Streams Front-End
Common type definitions, constants, etc.
I/O-Operations are defined here – useful as functionarguments
"I/O" taking place in memory of type std::string
I/O to/from external sources and sink (typically classic files or devices)
std::ios
std::istream
std::ostream
std::istringstream
std::ostringstream
std::ifstream
std::ofstream
#include <iosfwd>
for forward declarations
#include <iostream>
for full definition
#include <sstream>
#include <fstream>
Input from some data source
Output to some data sink
For keyboard input use: CTRL-D (Unix) or CTRL-Z (DOS)
int x;
1 2 3 4 5 a \n \t 6 7 8 - 9 \n
cin >> x;
cin >> x;
cin >> x;
cin.clear();
cin.ignore(1);
cin >> x;
cin >> x;
cin >> x;
12
345
sets fail-bit
unset fail-bit
advance over non-digit
678
-9
cin.good()
cin.good()
cin.eof()
= skip white.space
= extract data characters
I/O-Streams State-Bits
"day to day" use of C++
User API
Input
getc, gets …
operator>>
…
Output
putc, puts …
operator<<
…
Buffer-Management
std::streambuf
underflow()
xsgetn()
…
overflow()
xsputn()
…
specialisations for standard sources and sinks
specialisations for non-standard sources and sinks
used in standard library for implementation of std::istringstream std::ostringstream std::ifstreams std::ofstream
available for in-memory I/O with std::string-s and classic files/devices
useful for individual extensions though specal knowledge must be acquired
Mandatory overrides:
underflow for input (provide one more character when buffer is exhausted)
overflow for output (extract one character when buffer is full)
More overrides may improve performance:
xsgetn (provide more than one character)
xsputn (extract more than one character)
…
I/O-Stream Basics
(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de
I/O-Streams Back-End

I/O-Stream Front-End

The I/O-Stream Front-End consists of

- the base class `std::ios`* with some common defintions
- derived from it classes `std::istream` und `std::ostream` which are typically used as reference arguments to parametrise I/O-streams for functions called,
- and the following classes meant to be instantiated:
    - `std::ifstream`, `std::ofstream`, and `std::fstream` (File-Streams)
    - `std::istringstream`, `std::ostringstream`, and `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.

Common Interface

Operations for stream input and output are partially implemented as member functions of classes `std::istream` and `std::ostream`, partially as free standing functions.

More overloads of `operator>>` and `operator<<` may be added to support user defined data types.

Such overloads can only have the form of free functions (as otherwise new member functions would have to be added to `std::istream` and `std::ostream`).

(i) For more information see:
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

File Streams

The various file stream classes are used depending on the I/O direction:

- `std::ifstream` for reading
- `std::ofstream` for writing
- `std::fstream` for reading and writing*

(i) For more information see:
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

*: Note that in this case read and write positions in the stream are independent of each other and – depending on the task at hand – may or may not need explicit synchronisation.

String Streams

The various file stream classes are used depending on the I/O direction:

- `std::istringstream` for reading
- `std::ostringstream` for writing
- `std::stringstream` for reading and writing*

(i) For more information see:
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

*: Note that in this case read and write positions in the stream are independent of each other and – depending on the task at hand – may or may not need explicit synchronisation.

I/O-Stream Back-End

The main responsibility of the I/O-Stream back-end is abstraction and buffering (in case file streams).

Buffering allows to

- transmit data between internal memory and external storage in optimised block sizes,
- while the application has any freedom in which portions data is consumed or produced.

(i) For more information see:
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 Daten-Typs *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 spezifizierten – 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.

State-Dependent I/O-Stream Behaviour

As soon as an I/O-stream leaves the *good*-state further operations with that stream are ignored **except for `clear()` and `close()`**.

This has especially to be considered for a more detailed analysis of input:

- Any state change leaves the current position in the stream **prior to** the character causing the problem.
- Input not adhering to the expected format* – e.g. if a letter occurs where a digit is expected – input is processed **up to but not including** the unexpected character.

To skip over (at least) this character,

- **first** the stream must be put in the *good*-state, only
- **then** an operation like `ignore` will work.

(i) For more information see:
http://en.cppreference.com/w/cpp/io/basic_ios/clear
http://en.cppreference.com/w/cpp/io/basic_istream/ignore

*: Usually determined by the data type to be read with `operator>>`.

Exceptions on State Change

Individually for each stream and state it can be chosen that an exception is thrown for any

- **change to** that state (i.e. setting the corresponding state bit) and
- attempted operation **while in** that state.

```
// Excerpt from a hypothetical "forever running" TCP-Client
std::ifstream from_server;
… // 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
}
```