C++11 BLWS (Thursday 1)

Input and Output / Concurrency Basics

---

1. I/O-Streams (Recap)
2. The Buffer Interface
3. Boost: Iostreams
4. Boost: Serialization
5. C++11: Concurrency Basics
6. Boost: Threads Library
7. Boost: Asio

Short breaks will be inserted as convenient.

I/O-Streams (Recap)

The overall design of C++ I/O-streams is founded on seven classes further explained on the next slides.

> Actually these classes are only type definitions of some more basic templates that parametrize various aspects like the character type and traits of the streams.

For readability and as is sufficient to gain a basic understanding of the architecture, the following treats

- `std::ios`, `std::istream` ... etc. ... to `std::ostringstream`

instantiating the basic templates (`std::basic_ios` etc.) for plain characters as if they were the classes in question, not just type definitions which are similarly provided for wide characters:

- `std::wios`, `std::wistream` ... etc. ... to `std::wostringstream`

Class `std::ios`

This is the base class* of

- `std::istream` and
- `std::ostream`

centralizing a number of

- type definitions - like for the underlying character type (`ios::char_type`) or seeking in stream,
- enumerations - like `ios::fmtflags` for global formatting flags (`ios::fixed`, `ios::scientific`, ...) or the stream states, and
- common member data and functions - like for accessing and modfying formatting options (`std::ios::flags`, `std::ios::fill`, ...) or whether state changes should be notified by throwing stream exceptions.

---

*: For technical reasons a part of what is listed here is actually defined in an `std:ios` base class `std::ios_base`. To get a basic understanding for the C++ I/O-stream design this is a detail that need not be further explored.

Class `std::istream`

This class is derived from `std::ios` with the main purpose to pool the *stream extraction operations* (aka. input), e.g.:*

- read single characters (`get` with various overloads),
- lines of text into an `std::string` (`getline`),
- blocks of given length (`read`, `readsome`), or
- formatted input (`operator>>` for type dependant conversion from text)

> Whenever some function is **parametrized on a stream input source** `std::istream&` is usually the best choice for the argument type.

Choosing `std::ifstream` or `std::istringstream` instead would limit the selection of possible input sources an the callers side.

*: Not everything named below is a member - there are also global functions with an `std::istream` argument or overloads for `operator>>` with a left-hand-side operand of type `std::istream`.

Class `std::ostream`

This class is derived from `std::ios` with the main purpose to pool the *stream insertion operations* (aka. output), e.g.:

- write single characters (`get` with various overloads),*
- blocks of given length (`read`, `readsome`), or
- formatted output (`operator<<` for type dependant conversion to text)

> Whenever some function is **parametrized on a stream output source** `std::ostream&` is usually the best choice for the argument type.

Choosing `std::ofstream` or `std::ostringstream` instead would limit the selection of possible input sources an the callers side.

*: There is no `putline` counterpart to `getline` but `operator<<` is (of course) defined and fully sufficient for output of `std::string`-s.

File-Stream Classes

The file stream classes (accessible via the header `<fstream>`)

- `std::ifstream` and
- `std::ofstream`

are in turn derived from the base `std::ios`.

> Via an appropriate stream buffer an instance of any of these classes connects to a classic file as sink or source for the actual input and output.

Besides files residing as data container in some (non-volatile) storage space, the term *classic file* here includes also abractions like stream sockets.*

*: Or even more generally in Unix/Linux: everything which has an entry in the file system (`/dev/…`, `/sys/…`, `/proc/…`), i.e. serial or parallel ports, raw disks, information on processes, connected peripherals ...).

String-Stream Classes

The string stream classes (accessible via the header `<sstream>`*)

- `std::istringstream` and
- `std::ostringstream`

are in turn derived from the base `std::ios`.

> Via an appropriate stream buffer an instance of any of these classes connect to an `std::string` as sink or source for the actual input and output.

In case of `std::ostringstream` space for the character sequence stored is allocated dynamically - and fully transparent to the client.

Generally the amount of data to be stored in a string stream is only limited by the main memory available to a process and may be even more as physically installed RAM ...

*: Note that the header file name is spelled `sstream`, not `stringstream` and not `strstream` - in fact, the latter is the name of an earlier, similar interface based on character arrays instead of `std::string`s, which is deprecated since C++98.

Readable, Writeable, and Read-Writeable Streams

The previous slides concentrated on streams with a specific purpose:

- `std::ifstream` and `std::istringstream` as data source
- `std::ofstream` and `std::ostringstream` as data sink

There is also the combination, i.e.

- `std::fstream` allowing input and output to the same file
- `std::stringstream` allowing input and output to the same file `std::string`

If there is only one direction into which the data modes, the need not be used and should be second choice.

> **[!]** Otherwise you need to understand the underlying buffer management to guide your decissions if and when the buffer needs to be flushed when switching between input and output.

Seeking in Streams

At the heart of the I/O-stream model are the *seek position*, working much like an "invisble marker" just before the character that is next to be processed (on input) or overwritten (on output).

These are independantly managed for input and output also called

- *get position* accessed with `tellg` and eventually modified with `seekg`
- *put position* accessed with `tellp` and eventually modified with `seekp`

The positions are meassured in *characters*[*] the file's begin from `tellg` and `tellp` and can be set with `seekg` and `seekp` in three ways:

- relative to the begin of the file,
- relative to the current end of the file, or
- or relative to the current position.

---

[*]: The actual character type will of course be `wchar_t` for the wide streams (`wistream`, `wostream`, etc.) but it needs to be understood that these are not necessarily the "*user preceived characters*" nor need there be a 1:1 correspondence to "*code points*" of a character set as UTF-32, as both - UTF-8 (based on 8 bit units) and UTF-16 (based on 16 bit units) - use a variable length encoding scheme.

Stream States

An input or output stream can generally switch between a number of states:

Technically these are not mutually exclusive (as the term "state" might suggest) but there are just three flags that may be set independantly:

- `std::ios::eofbit` - testable with the member function `eof()`
  set when the stream has advanced over its last character*
- `std::ios::failbit` - testable with the member function `fail()` some "*soft error*" has occcured and a retry might make sense
- `std::ios::badbit` - testable with the member function `bad()` some "*hard error*" has occured (it's unlikely a retry will recover)

When none of the above flags is set, the stream is said to have *good* state - testable with the member function `good()`.

*: Testing this state makes sense mostly for data sources but in case of a data sink it might be set too if there is no more room on the device where the output is to be stored, so due to buffered output, it can rather be expected that the stream enters the fail state when the buffer is written.

Implicit Stream State Switching

For one, stream states may be switched during I/O* taking place, according to the following rules:

- When stream extraction has moved over the last character - the physical end of the file in case of an external file or the last character on an `std::string` - the `std::eofbit` flag will be set.
- When there is a problem converting the characters read while processing formatted input, the `std::ios::failbit` flag will be set.
- When there is any other severe problem the `std::badbit` flag will be set.

Once any of the above flags is set, the stream will temporarily cease to be used for nearly most any operation.

Explicit Stream State Switching

The only operations allowed while a stream is not in the good state are:

- `clear()` - clear all state flags.*
- `close()` - release connection to data sink or source

Note that clearing the state flags does what it says: the state flags will be cleared, nothing less, nothing more.

Especially the get position will not change.

If the root of the problem was a formatting error - e.g. it was tried to read an `int` while the next input character was a non-digit - that character must be skipped somehow.

> **[!]** It is a common error of novices to either forget the above or to overlook the fact that all the error flags need to be cleared **before** advancing the get position.

*: Any argument **will set(!!) the flags** named by it, so `strm.clear(std::ios::fail)` may be part of the implementation of `operator>>` for a user type to indicate a formatting error.

Stream Exceptions

Optionally an exception may be thrown when a stream sets any of its state flags*

Individually for

- each stream instance and
- each state flag

it can be chosen that an exception is thrown whenever the state bit is set.

> In addtion it causes an exception to be thrown by any stream operation that would otherwise be silently ignored (outside the *good* state).

*: The stream interface with its implicit and explicit state switching was defined in a time when C++ not yet had exceptions. For backward compatibility the default behavior is still to simply ignore I/O operations for streams that are not in the *good state*. For new software developments this might not make sense anymore and it may delay the impact of problems and therefore impede error analyis.

Stream Exceptions Example

Read `float` values from standard input and calculate sum:*

```
cin.exceptions(std::ios::failbit | std::ios::badbit);
double sum{0.0};
bool giveup{false};
do {
    try {
        double val;
        while (cin >> val)
            sum += val;
    }
    catch (std::ios_base::failure &e) {
        if (cin.bad()) {
            giveup = true;
            cerr << "giving up before eof is reached" << endl;
        }
        else if (!cin.eof()) {
            cin.clear();
            cerr << "char ignored: " << char(cin.get()) << endl;
        }
    }
} while (!cin.eof() && !giveup);
```

*: Necessary includes and namespace directives assumed.

The Buffer Interface

One of the main purposes of the stream classes is to make input and output efficient, especially if the actual source or sink is a classic file:

- Accessing the external store will usually happen in large chunks ...
- ... while the program logic can deal with the stream in whatever portions are convenient.

> Usually buffering and the logic involved is to the most part transparent to a client (program) using C++ I/O-streams.

But there is also the option to take over control by implementing new buffer classes for a specific purpose.* Applied wisely, this approach

- can lead to a hight degree of encapsulation
- and may also promote the potential for reuse.

*: Such may als be derived from `std::filebuf` or `std::stringbuf` if refined file streams or string streams are necessary.

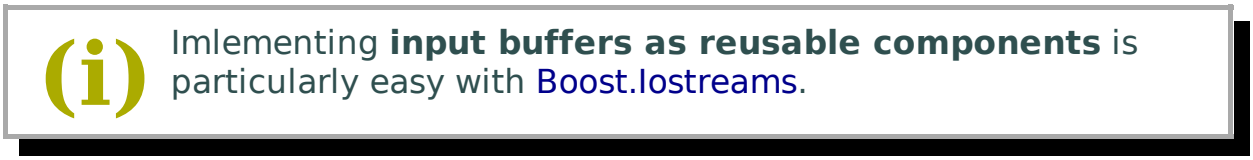
Buffers for Input Streams

The minimum requirement for implementing a buffer to be used with a special kind of input stream is to

- implement the member function `underflow`,
- which will be called whenever (more) input needs to be retrieved from the physical source.

Especially there are no other member functions necessary, though such may help to improve performance.

Actually the above also assumes the default - that is not to do any input buffering but forward each character immediately - but setting up a real input buffer is straight forward and not so much of an effort.*



*: Assuming a base class is used that takes care of the nitty-gritty details, like mandatory type definitions and default implementations for optional member functions.

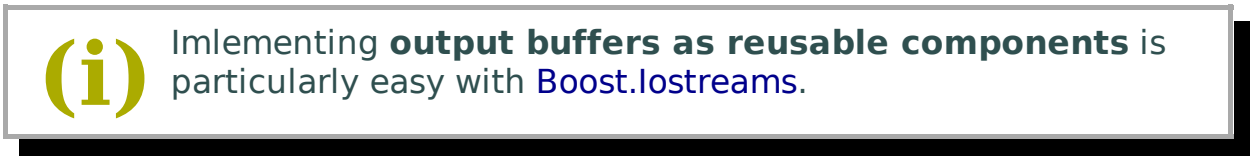
Buffers for Output Streams

The minimum requirement for implementing a buffer to be used with a special kind of output stream is to

- implement the member function `overflow`,
- which will be called whenever there is no more room to buffer the output destined for the physical sink.

Especially there are no other member functions necessary, though such may help to improve performance.

Actually the above also assumes the default - that is not to do any output buffering forward each character immediately - but setting up a real outut buffer is usually straight forward and not so much of an effort.*



*: Assuming a base class is used that takes care of the nitty-gritty details, like mandatory type definitions and default implementations for optional member functions.

Boost: Iostreams

Boost.Iostream is particularly helpful to implement

- specialized stream buffers for input and ouput as reusable components
- which may be dynamically configured depending on what is needed.

> To some degree the approach is similar to pipelining some input for or output of an application through standard programs.

The only difference is that everything happens internally, but the idea of combining small modules of which

- each *does a single, simple thing* well

is the same.*

*: If all what has to be done can be expressed with input and ouput modules only, Boost.Iostream even contains a small helper that can replace a main program feeding the input from one stream as output to the other one.

Input Preprocessing

As an example may serve an idea for compenents which help to parse Unix-style configuration files.*

There could be

- one module to strip empty lines and comments and
- another module to join continuation lines.

both of which are near to trival and hence easy to develop and test.

- An independant example were a module that recognizes and appropriately replace XML character encodings (like '&*xx*;').

In any case, from the viewpoint of the actual application, just plain text is read from an input stream.

*: Configuration file syntax in Unix was (and still is in Linux) often "ad hoc" but some common style has emerged over time: (1) empty line are not significant, (2) comments can be embedded in lines starting with a hash-sign ('#'), and (3) long lines maybe broken into parts by ending lines that continue with a backslash ('\').

Output Postprocessing

As examples may serve compenents that care for some normalization in a postprocessing step, that may be dynamically configured from a library with reusable components.

- Recognize and appropriately replace non-ASCII characters with the equivalent XML encoding (i.e. '&*xx*;').
- Format simple plain text by breaking long lines at white space if a certain line length would otherweise be exceeded.
- As before but automatically insert a line continuation convention (like ending lines to be continue with a backslash).
- Prefix each line with its line number.

In any case, from the viewpoint of the actual application, just plain text is written to an output stream.

*: As the mechanism is generic to stream buffers, such modules might also be applied in a way that the output is caught in a string, so that - with the same module - only plain text parts of a web page cound be sanitized by replacing '<' with '<' etc. but not for the actual HTML tags.

Boost: Serialization

With Boost.Serialization object instances may be safed to persinstant storage at some point to be restored later.

> This includes instances refering to each other via pointers or references: The mechanism takes care for "duplicates" if in a network of objects a particular instance is reachable via more than one path.

The obvious applications are:

- Save the last state to continue the next run of an application where the previous left off.
- Checkpoint an application to provide a roll-back facility via the snapshots taken or reconstruct history.

The degree to which this can work non-intrusive is limited, especially compared to other languages with much richer support for introspection.*

*: E.g. Java has a *reflection interface* that is close to perfect ... but on the other hand requires a lot of meta-information to be stored - even if a program never makes use of serialization.

C++11: Concurrency Basics

With C++11 support for multi-threading was made part of the

- Standard Library
- **and the Core Language**..[]

A good near to exhaustive standard reference to this part of C++11 is:

C++ Concurrency in Action

Practical Multithreading

by: Anthony Williams

ISBN-13: 978-1-9334988-77-1

*: Why multi-threading cannot be added as a library alone, i.e. the importance of the core language part, can be seen when considering long-existing features like the initialization of local `static` variables, which takes place when their definition is reached for the first time and therefore, of course, must be protected with a mutex (or similar) to avoid race conditions.

Boost: Threads Library

As C++11 threads emerged from Boost.Thread there is little difference.

Minor differences to the between C++11 and Boost may exist (depending on the version of the latter) and new features may appear and be tried in Boost first before eventually getting part of an upcomming C++ standard (like C++14).

Boost: Asio

The purpose of Boost.Asio is to support event-driven program designs by providing a framework to dispatch incoming events to previously registered event-handlers:

- The largest group of events deals with I/O, especially the arrival of data from asynchroneous sources (like sockets).
- Besides that it is also an applications sends delayed events to itself.

Event handlers are run single-threaded and hence there is no need for synchronisation as is in multi-threaded designs. For good responsiveness an event driven program design must keep event handlers small.

> Especially an event handler should **never** delay or wait for responses of an external client - rather register an handler to be started when the response arrives.