C++11 BLWS (Tuesday 2)

A Mix of Useful Things

---

1. Formatting
2. Boost: I/O-State Saver
3. Boost: format
4. Boost: File System
5. C++11: Chrono
6. Boost: Date & Time
7. C++11: Random

Short breaks will be inserted as convenient.

Formatting

Compared to C-style `printf`-format strings C++ output formatting has changed the concept dramatically. One of the main reasons probably was to get more type safety as C could provide.

The big obstacle with the C++ design is its stateful stream formatting:*

- E.g. switching to a hexadecimal output format will be persistant ...
- ... until switched back to decimal - which introduces the problem:
- What if the prior output format wasn't decimal but octal?

[!] Any local change of persistant stream formatting settings should be undone afterwards, otherwise unwanted effects lurk around the corner, especially if the program logic has branches which are rarely taken and therefore not thoroughly tested.

*: Sometimes the fact that Bjarne Stroustrup had chosen to overload the shift operators for I/O is also heavily criticised, though nowadays few C++ developers seem to share this view and some even call `operator<<` the *output operation* ...

Sharing Stream Buffers

One lesser known option to avoid resetting the format flags to the original state is to work with several separate stream that share the same buffer:

```
#include <iostream>
#include <iomanip>
int main() {
    std::ostream octout{std::cout.rdbuf()}; octout << std::oct;
    std::ostream decout{std::cout.rdbuf()}; decout << std::dec;
    std::ostream hexout{std::cout.rdbuf()}; hexout << std::hex;
    std::cout << "char oct dex hex\n";
    for (char c{'a'}; c <= 'z'; ++c) {
        const int v{c};
        std::cout << "'" << c << "'  ";
        octout << std::setw(3) << v << ' ';
        decout << std::setw(3) << v << ' ';
        hexout << std::setw(3) << v << '\n';
   }
}
```

Saving and Restoring Format Flags

The direct way to restore the original state is to use the member functions to get and save the current flags, modify formatting as needed, print the value, and finally restore the saved flags:*

```
// on ostream 'os' print 'v' in hexadecimal, using uppercase for
// letters A..F, prefixed with 0x (lower case 'x'), with minimum
// field width as set before calling to this function
void print_hex(std::ostream &os, unsigned long long val) {
    const auto usewidth = (os.width()) < 2 ? : os.width() - 2;
    const auto oldflags = os.flags(std::ios::hex
                                   | std::ios::uppercase
                                   | std::ios::right);
    const auto oldfill = os.fill('0');
    os << std::setw(0) << "0x" << std::setw(usewidth) << val;
    os.setf(oldflags);
    os.fill(oldfill);
}
```

*: Note that setting the minimum field width is not persistent and will always apply to the next output, then reset to zero.

Boost: I/O-State Saver

The idea behind Boost.IO_State_Savers is to restore flags in a destructor at scope exit:*

```
void print_hex(std::ostream &os, unsigned long long val) {
    const auto usewidth = (os.width()) < 2 ? : os.width() - 2;
    boost::io::ios_flags_saver flags(os);
    boost::io::ios_fill_saver fill(os);
    os << std::setw(0) << "0x"
       << std::hex << std::uppercase << std::right
       << std::setfill('0') << std::setw(usewidth) << val;
}
```

The advantages of this approach become even more visible if control flow is not as linear as above. Especially if exceptions might occur, no `try-catch`-logic must be added.

*: Using manipulators instead of member function calls to change the formatting state is not essential here but makes the code even more compact.

Boost: format

Boost.Format is the return of C's `printf`-style output formatting into C++*

- The core format string language is much similar to C.
- Beyond this there are many extensions, e.g.
  - the order of values to format must not necessarily be the same as the order of place holders in the format string.
  - Extensions for user specified types are possible ...
  - ... but even without any type that has `operator<<` defined will work.

Type safety is guaranteed at run time, i.e. an exception will be thrown if a value to format is not compatible with the placeholder.

Boost: File System

Even with C++11 there is no portable way to access the file system to

- search through directories and sub-directories,
- determine and change file properties,
- delete, rename, link or copy files.

> Boost had tried to tackle this since a long time - with more or less success - and is currently in its 3rd major release Boost.Filesystem V3

This file system library may also - via the TR2 Path - become available with recent Standard C++ versions and is part of Microsoft Visual Studio 2013.

*: At the time of writing the final state of affairs is not quite clear.

C++11: Chrono

With C++11 a library component for "managing date and time" was introduced (beyond what was available for long because of C compatibility).

- As many similar libraries it makes a clear distinction between
  - durations and
  - time points.

> The feature that makes this library shine is its flexibility with respect to the usual trade-off between resolution, range, and space requirements of the underlying type (to store a duration or time point).

`std::chrono` - Durations

Though the duration type is fully configurable through a template*, most users will choose from

- `std::chrono::nanoseconds`
- `std::chrono::microseconds`
- `std::chrono::milliseconds`
- `std::chrono::seconds`
- `std::chrono::minutes`
- `std::chrono::hours`

for a predefined type to satisfy their needs for an appropriate resolution.

> Conversions to a *finer grained* durations will allways happen automatically, while conversions to *coarser grained* durations require a `duration_cast`.

*: E.g. a duration could well count in 5/17 microseconds if that matches the resolution of a hardware timer exactly and allows for precise calculations without any rounding errors or occasional jitter.

`std::chrono` - Clocks

A duration (type) combined with an *epoche*[*] is said to be a *clock* and represents a time point.

What clocks are supported is basically implementation defined with the following minimum requirements:

- `std::chrono::system_clock` - represents the usual "wall-clock" or "calendar date & time" of a computer system.
- `std::high_resolution_clock` - the clock with the best resolution available (but with a more or less frequent wrap-around).
- `std::chrono::steady_clock` - probably not tied to a specific calendar date and with the special guarantee that it will only advance.

> Only the last allows to realiably determine a real time span as difference of two time points returned from its static member function `now()`.

[*]: Per definition the epoche is the time point represented by the duration zero. From there a clock might reach into the past and into the futrue, usually symmetrically if an ordinary signed integral or floating point type is used.

`std::chrono` - Operations

Operators are overloaded to supprot the following mix between durations and time points:

Operand Type	Operation	Operand Type	Result Type
duration	plus or minus	duration	duration
time point	plus or minus	duration	time point
time point	minus	time point	duration
duration	multiplied with	plain number	duration
duration	divided by	plain number	duration
duration	divided by	duration	plain number
duration	modulo	duration	duration

Boost: Date & Time

Except for maintaining a similar semantic difference between durations and time points Boost.Date_time seems to have little in common with the Chrono library now part of C++11.

> While legacy code using that library will still be around for some years, on the long run it can be expected that the importance and user base of this library may decrease and code be updated to use `std::chrono`.*

*: Note that with Boost.Chrono the Chrono Library as standardized with C++11 is now available on the Boost platform too.

C++11: Random

Compared to C style pseudo random number generation with `std::rand`, C++11 has adopted a facility for generating random numbers with given distributions, but at a price:

The code to role a simple dice isn't any more as easy* as

```
int throw_dice() { return 1 + std::rand() % 6; }
```

but requires at least something along the following lines:

```
int throw_dice() {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_int_distribution<> dis(1, 6);
    return dis(gen);
}
```

*: ... and wrong or at least flawed for the following reasons: (1) Some C implementations start to repeat the "random" sequence as early as after 65534 repetitions and (2) if the range of pseudo-random numbers (starting with zero) is not evenly divisable by six the chosen way to map the numbers to 1..6 will slightly favour the lower values.