# 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: Chrono
7. Boost: Date & Time
8. C++11: Random
9. Boost: 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 persistent ...
- ... until switched back to decimal – which introduces the problem:
- What if the prior output format wasn't decimal but octal?

**[!]** Any local change of persistent 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';
   }
}
```

*: This example demonstrates the principles but is not that clearly showing the advantages, mainly because lack of space (to make it fit on a single page). To get a better idea how it could improve a larger program's structure, imagine the output operations were in separate functions, called intermingled with other output operations that expect a certain formatting state.

# 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 to change the formatting state is not essential here but makes the code even more compact.

*: Boost.Scope_exit generalizes the idea to arrange beforehand that an arbitrary block of code gets executed on scope exit via the destructor of a block-local object.

# Boost: format

Boost.Format is the return of C's `printf`-style output formatting to 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. A major obstacle through all the years seems to have been uniting the classic and also modern 8-bit-char API of Unix/Linux (using UTF-8 now, which the clients – by and large – can handle "content-agnostic") with the 16-bit-char API of MS-Windows in a portable way ...

# 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).

(i) The chapter on the C++11 *Chrono Library Part* from Nicolay Josuttis Books, referenced above, has also been made available online here: http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2

# `std::chrono` – Durations

Though the duration type is fully configurable through a template*, most programs will probably choose from one of the predefined types that satisfies their needs for resolution:

- `std::chrono::nanoseconds` at least 64 bit signed
- `std::chrono::microseconds` at least 55 bit signed
- `std::chrono::milliseconds` at least 45 bit signed
- `std::chrono::seconds` at least 35 bit signed
- `std::chrono::minutes` at least 29 bit signed
- `std::chrono::hours` at least 23 bit signed

> Duration type conversions to a *finer grained* type will always happen automatically, while conversions to *coarser grained* type require an `std::chrono::duration_cast`.

A 64-bit type the minimum requirement for nanosecond resolution – with the **minimum requirement** for the other types adapted accordingly – the minimum range of a duration covers ±500 years.

*: E.g. a duration type 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.

## Duration Example (1)

For a basic use it needs only to be understood that automatic conversion happen as long as the target duration counts in finer grained units ...

```
#include <chrono>
…
std::chrono::minutes m{22};  // m.count() is 22
std::chrono::seconds s{17};  // s.count() is 17
s += m;                      // s.count() is 1337 (22*60 + 17)
s *= 100;                    // s.count() is 133700
```

... while assignments to coarser grained durations are an error ...

```
m = s;                       // does NOT compile
```

... unless an `std::chrono::duration_cast` is applied:*

```
m = std::chrono::duration_cast<std::chrono::minutes>(s);
                             // m.count() is 2228
```

## Duration Example (2)

To avoid long namespace prefixes, namespace aliases are handy:*

```
namespace sc = std::chrono;   // abbreviating std::chrono:: to sc::
…                                    // continuing from previous page
sc::hours h = sc::duration_cast<sc::hours>(m); // m.count() is 2228
//   auto h = sc::duration_cast<sc::hours>(m);
```

Finally a useful helper to turn durations into something readable:

```
std::string to_string(sc::seconds sec) {
    const auto h = sc::duration_cast<sc::hours>(sec);
    const auto m = sc::duration_cast<sc::minutes>(sec-h);
    const auto s = sc::duration_cast<sc::seconds>(sec-h-m);
    return std::to_string(h.count()) + "h"
         + std::to_string(m.count()) + "m"
         + std::to_string(s.count()) + "s";
}
…             // continuing from above
… to_string(s) … // returns "37h8m20s"
```

*: These seems especially useful to abbreviate nested `std::` namespaces – like those from the `<chrono>` – while keeping a gentle reminder the identifier following is from the standard library.

# `std::chrono` – Clocks

A duration (type) combined with an *epoch*[*] is a *clock* that represents a time point.

Which kind of 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 reliably determine a real time span as difference of two time points returned from its static member function `now()`.

[*]: Per definition the epoch of a clock is the time point represented by the duration zero. From its epoch a clock will reach into the past and into the future, usually symmetrically if an ordinary signed integral or floating point type is used.

# Clock Examples (1)

Following are the attributes of `std::chrono::system_clock` ...*

```
resolution : 1/1000000
value range: -9223372036854775808 .. 9223372036854775807
since epoch: 45+ years
          or: 16581+ days
          or: 397964+ hours
          or: 23877893+ minutes
          or: 1432673591+ seconds
or in ticks: 1432673591377941
```

---

*: ... determined on the author's system with the helper function below:

```
template<typename Clock>
 void show_clock() {
        using period = typename Clock::period;
        using limits = std::numeric_limits<typename Clock::rep>;
        std::cout << "resolution : " << period::num << '/' << period::den << '\n';
        std::cout << "value range: " << limits::min() << " .. " << limits::max() << '\n';
        const auto tse = Clock::now().time_since_epoch();
        const auto sse = sc::duration_cast<sc::seconds>(tse).count();
        std::cout << "since epoch: " << sse /60/60/24/365 << "+ years\n";
        std::cout << "          or: " << sse /60/60/24 << "+ days\n";
        std::cout << "          or: " << sse /60/60 << "+ hours\n";
        std::cout << "          or: " << sse /60 << "+ minutes\n";
        std::cout << "          or: " << sse << "+ seconds\n";
        std::cout << "or in ticks: " << tse.count() << '\n';
 }
```

## Clock Examples (2)

Clocks can also be used to determine the time passed between two time points:*

```
template<typename TestCode>
void test_timing(unsigned repeat, TestCode testrun) {
    using sc = std::chrono;
    const auto started = sc::high_resolution_clock::now();
    for (auto i = 0; i < repeat; ++i) testrun();
    const auto ended = sc::high_resolution_clock::now();
    const auto delta = ended - started;
    const auto nanosec = sc:duration_cast<sc::nanoseconds>(delta);
    const auto per_run = nanosec.count() / repeat;
    std::cout << nanosec << " ns total for " << repeat << " runs"
                            " = " << per_run << " ns per run\n";
}
…
test_timing(100*1000, // repeat 100.000 times ...
            [](){
                … // ... this some code fragment
            });
```

*: Note that – as far as shown here – real time is measured, not CPU time, but `boost::chrono` has also clocks for measuring CPU time.

# std::chrono – Operations

Operators are overloaded to support mixed 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 | modulo | plain number | duration |
| duration | divided by | duration | plain number |
| duration | modulo | duration | duration |

Combinations not listed in the table above result in compile time errors.

For operands with standard types* of different resolution the result will use the appropriate type with the finer grained resolution.

*: When non-standard types are combined the required result type will be calculated accordingly. E.g. to store the sum of a duration counting in 10/21 seconds and another one counting in 14/15 seconds, a result type counting in 1/105 seconds will be used.

# Boost: Chrono

Boost.Chrono implements the C++11 conformant Chrono classes with some additions.

Such target the area of measuring CPU-time, i.e.

- clocks to which ticks only get added when the CPU is active for the current process,
- usually making a difference between *User Time* and *System Time*.

(i) For more information on the option to measure CPU time with the additional clocks provided by Boost.Chrono see section Other Clocks in http://www.boost.org/doc/libs/release/doc/html/chrono/reference.html

# Boost: Date & Time

Boost.Date_time has little in common with C++11 Chrono, except for maintaining a similar semantic difference between durations and time points.

> 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 divisible by six the chosen way to map the numbers to 1..6 will slightly favour the lower values.

# C++11: Random Number Generators

All random number engines generate the next value by applying the function call operator (without arguments).*

Some random number generators may be used without distributions if a uniform distribution over the range of generated values is required, e.g.:

- `std::mt19937` – uniformly distributed values over a 32 bit range
- `std::mt19937_64` – uniformly distributed values over a 64 bit range

The C++11 standard also defines an `std::random_device` which may either be mapped to a non-deterministic random source or (if not available) to one of the existing (pseudo-random) sources.

For more informations on the available random number generators see subsection Random Number Engines in: http://en.cppreference.com/w/cpp/numeric/random

*: The prefix `mt` abbreviates the algorithm (Mersenne Twister) and `19937` is the period, after which the generated numbers start to repeat identically: which is $2^{19937}$. Or in other words: if such a generator existed since the big bang and random numbers were extracted with a GHz clock since, until today it would have generated only numbers from a tiny weeny fraction of its non-repeating range.

# C++11: Random Number Distributions

All distributions produce the next value by applying the function call operator with a generator (engine) as parameter.

There are many distributions available – separately from the generator – of which the *Uniform Distribution* is probably the one most often used. It is available as:*

- `std::uniform_int_distribution`
  generating integral values in a given range
- `std::uniform_real_distribution`
  generating floating point values in a given range
- `uniform_canonical`
  generating values between 0..1 with a given precision

For more information on the above and other distributions see subsection Random Number Distributions in:
http://en.cppreference.com/w/cpp/numeric/random

*: Note that contrary to the otherwise asymmetric limits, as commonly used in C and the C++-STL, the limits specified for distributions are inclusive and symmetric.

# Boost: Random

Boost.Random implements the C++11 conformant Random classes with some additions.