C++11 BLWS (Thursday 1)

Input and Output / Concurrency Basics

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 modifying 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 specific 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 at 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 (`put` with various overloads),*
- blocks of given length (`write`, `writesome`), or
- formatted output (`operator<<` for type specific conversion to text).

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

Choosing `std::ofstream` or `std::ostringstream` instead would limit the selection of possible output sinks at 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 abstractions like stream sockets.*

*: Or even more generally in Unix/Linux: everything that 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 can 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 discussed here 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, and
- `std::stringstream` allowing input and output to the same `std::string`.

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

> **[!]** Otherwise you need to understand the underlying buffer management to guide your decisions if 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 positions*, working much like an "invisible marker" just before the character that is next to be processed (on input) or overwritten (on output).

These are independently managed for input and output and 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 measured in *characters*[*] relative to the begin of the file by `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
- 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 perceived 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 independently:

- `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 occurred and a retry may make sense;
- `std::ios::badbit` – testable with the member function `bad()` some "*hard error*" has occurred (it's unlikely a retry will recover).

When none of the above flags is set the stream is said to be in *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 may be set too if there is no more room on the device where the output is to be stored, though 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 for an external file or the last character on an `std::string`- the `std::ios::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::ios::badbit` flag will be set.

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

Explicit Stream State Switching

The only operations carried out 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 specified **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 addition this 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 ignore I/O operations for streams that are not in the *good state*. For new software developments this makes less sense as it delays the impact of problems and therefore impedes error analysis.

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 content 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.

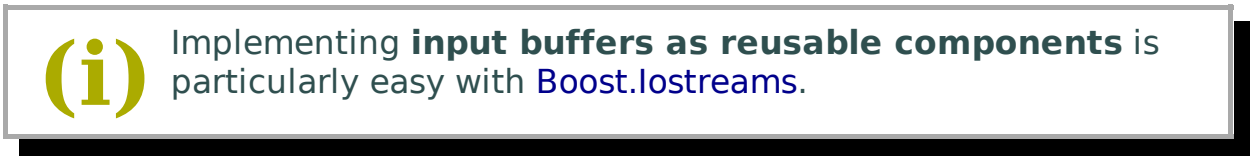
Buffers for Input Streams

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 – though 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.

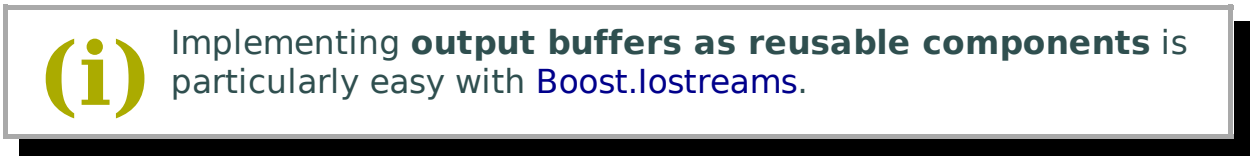
Buffers for Output Streams

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 but forward each character immediately – though setting up a real output 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 output 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 exactly the same.*

*: If all what has to be done can be expressed with input and output 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 components 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 trivial and hence easy to develop and test.

- An independent 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 components caring 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 otherwise 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 may 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 could be sanitized by replacing '<' with '<' etc. but not for the actual HTML tags.

Boost: Serialization

With Boost.Serialization object instances may be saved to persistent storage at some point to be restored later.

> This includes instances referring 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.*

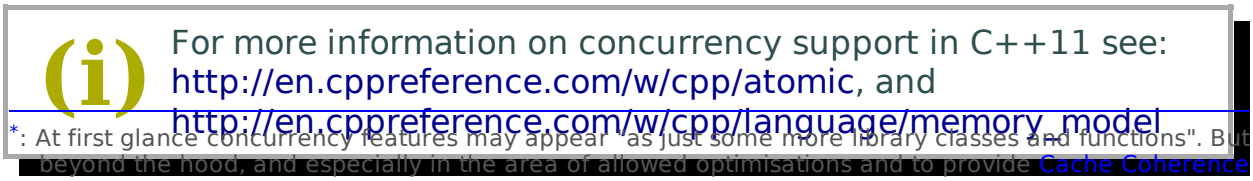
*: E.g. Java has a *reflection interface* that is close to perfect ... but on the other hand requires a lot of meta-information to be stored – even if a program never makes use of serialization.

C++11: Concurrency Basics

With C++11 support for multi-threading was introduced.*

- Parallelizing Independent Tasks
- Synchronisation with Mutexes
- One-Time Execution
- Messaging with Condition Variables
- Atomic Operations
- Direct Use of Threads
- Native Threading Model Handles
- Concurrency Recommendations

This part of the presentation were written with the intent to give an overview of the provided features only, they are by far not exhaustive!



*: At first glance concurrency features may appear "as just some more library classes and functions". But beyond the hood, and especially in the area of allowed optimisations and to provide Cache Coherence on modern multi-core CPUs, concurrency is closely intertwined with code generation issues.

Parallelizing Independent Tasks

For complex tasks that can be split into independent parts, concurrency **and scalability to multiple cores** can be easily achieved by following a simple recipe:

1. Separate the task into a number of different functions (or calls of the same function with different arguments).
2. Run each such function by handing it over to `std::async`, storing the `future` that is returned (easiest in an `auto`-typed variable).
3. Fetch (and combine) the results by calling the member function `get` for each `future`.

That way all the functions may run concurrently and the last step synchronizes by waiting for completion.

*: Technically the return value of `std::async` is an `std::future` but as this is a template and the type is usually somewhat different to spell out, most usages of `std::async` store the result in an `auto`-typed variable.

Foundation: Futures and Promises

The foundation on which parallelizing tasks is build are Futures and Promises.

These need not be fully understood to apply the API (exemplified on the next pages), but may help to understand the basic machinery:

- A future is the concrete handle which a client can use to fetch the result, presumably made available by a different thread of execution.
- A promise is a helper class which may be used in a separate thread to make a result available for a client.

(i) For more information on promises and futures see:
http://en.cppreference.com/w/cpp/thread/future
http://en.cppreference.com/w/cpp/thread/promise
http://en.cppreference.com/w/cpp/thread/packaged_task

Parallelizing Example

The following example calculates the sum of the first `N` elements in `data` by splitting the work of `std::accumulate`, into two separate function calls, that may run concurrently:*

```
// calculate sum of first N values in data
//
long long sum(const int data[], std::size_t N) {
    auto lower_part_sum = std::async(
        [=]{ return std::accumulate(&data[0], &data[N/2], 0LL); }
    );
    auto upper_part_sum = std::async(
        [=]{ return std::accumulate(&data[N/2], &data[N], 0LL); }
    );
    return lower_part_sum.get()
         + upper_part_sum.get();
}
```

---

*: Note the use of lambdas above – the actual call to `std::accumulate` will only happen when the lambda gets executed! A similar effect can be achieved as follows:

```
// preparing the callable for std::async with std::bind:
… std::async(std::bind(std::accumulate, &data[0], &data[N/2], 0LL)) …
… std::async(std::bind(std::accumulate, &data[N/2], &data[N], 0LL)) …
```

Default Launch Policy

The default behavior is that the systems decides on its own whether an asynchronously started task is run concurrently.*

Using `std::async` **without an explicit launch policy**

- **is just a hint** that it is acceptable to run a callable unit of code concurrently,
- as long as it has finished (and possibly returned a result) latest when the `get`-call returns, which has bee invoked on the `std::future`.

[!] Be sure **not** to use the default launch policy but specify concurrent execution explicitly (see next page) whenever it is essential that two callable run concurrently.

*: It is a well known effect that too many parallel threads of execution may rather degrade performance. Especially if threads are CPU bound, it makes little sense to have more threads as cores. Therefore the standard gives considerable freedom to the implementation, which might implement concurrency with `std::async` with a thread pool and turn to synchronous execution or lazy evaluation at a certain threshold.

Explicit Launch Policies

There is a second version of `std::async` which has a first argument to specify the launch strategy.

The standard defines two values:

- `std::launch::async`
  if **not set** the callable will **not** run on its own thread;*
- `std::launch::deferred` of **set** the callable will **not** be called before `get` is invoked.

*: One case in which this setting this flag may make sense is to test the program logic independent from possible problems created through race conditions or deadlocks, originating from dependencies between the "independent tasks", that had been overlooked.

Catching Exceptions

If the callable started via `std::async` throws an exception, it will appear as if it were thrown from the call to get.

Hence, if the asynchronously run task may throw, fetching the result should be done in a try block:

```
auto task1 = std::async( … ); // whatever-is-to-do (and may throw)
auto task2 = std::async( … ); // whatelse-is-to-do (and may throw)
…
try { … task1.get() … }
} catch ( … ) { // what may be thrown from whatever-is-to-do
   … // handle the case that whatever-is-to-do threw
}
try { … task2.get() … }
} catch ( … ) { // what may be thrown from whatelse-is-to-do
   … // handle the case that whatelse-is-to-do threw
}
```

Communication between Independent Tasks

First of all: If the need arises to communicate between independent tasks, this should be taken **as a strong warning** that such tasks are actually not independent.

> **[!]** If parallel tasks are not independent, further needs follow quickly with respect to synchronize access to shared data ... **with all the further intricacies following from this**.*

Nevertheless there is one common case that requires a simple form of communication between otherwise independent tasks.

- If there are several tasks working towards a common goal
- of which one fails, making the goal unattainable,
- the others should not waste CPU-time needlessly.

*: In other words: Pandora's proverbial can of worms opens quickly and widely ...

Communicate Failure between Concurrent Tasks

A basic design that communicates failure between partners working towards a common goal is outlined in the following example.*

The workers could look about so …

```
void foo ( … , bool &die) {
    …
    for ( … ) {
        if (die) return;
        …
        … // some complex
        … // algorithm
        …
        // may fail here
        if ( … ) {
            die = true;
            return;
        }
    }
}
```

… being run by that code:

```
bool die = false;
auto task1 = std::async(
    [&die]{ foo( … , die); }
);
auto task2 = std::async(
    [&die]{ foo( … , die); }
);
…
… task1.get() …
… task2.get() …
…
if (die) {
    // goal not reached
    …
}
```

*: To keep this code simple it just returns in case of problems, though it requires only a few changes if problems should be communicated to the caller via exceptions.

Synchronisation with Mutexes

The word *Mutex* abbreviates *Mutual Exclusion* and describe the basic purpose of the feature.

- Allow only one thread to enter a Critical Section, typically non-atomically executed sequence of statements
- which temporarily invalidate an Class Invariant, or
- in other ways accesses a resource not designed for shared use.

In general, mutexes have at least two operations* for

- **lock**-ing and
- **unlock**-ing,

but frequently provide additional features to make their practical use more convenient or less error prone.

*: Though the operations may not be spelled exactly *lock* and *unlock* ... (especially as mutexes are somewhat related to [Semaphores], which originally named their lock (-like) acquire operation *P* and their unlock (-like) release operation *V*.

Mutex Example (1)

The following example calculates one table from another one:*

```
template<typename In, typename Out, typename Transformation>
void worker(const In data[], std::size_t data_size,
            Out result[], std::size_t &total_progress,
            Transformation func) {
    static std::mutex critical_section;
    while (total_progress < data_size) {
        critical_section.lock();
        constexpr auto chunks = std::size_t{100};
        const auto beg = total_progress;
        const auto end = ((data_size - total_progress) > chunks)
                 ? total_progress += chunks
                 : total_progress = data_size;
        critical_section.unlock();
        std::transform(&data[beg], &data[end], &result[beg], func);
    }
}
```

*: The work is shared by any number of `worker` task run concurrently, each fetches and transforms a fixed number of values. This can be advantageous to splitting the work by calculating fixed-size regions of the table in advance, if the transformation function has a largely varying runtime depending on the argument value.

Mutex Example (2)

Assuming the transformation is to calculate square roots and there are two arrays of size `N`, say

- `data` (filled with values to transform), and
- `sqrts` to store the results

workers may be created (to be handed over to `std::async`) as follows:*

```
std::size_t processed_count = 0;
auto worker_task =
    [&]() { worker(data, N, sqrts, processed_count,
                   [](double e) { return std::sqrt(e); });
    };
```

*: Given the above, a particular nifty way to create and run workers were:

```
// assuming NCORES holds the number of cores to use by workers:
std::array<std::future<void>, NCORES> workers;
for (auto &w : workers)
    w = std::async(worker_task);
for (auto &w : workers)
    try { w.get(); } catch (...) {}
```

Mutexes and RAII

As the mutex operations **lock** and **unlock** need to come correctly paired, they make a good candidate to apply a technique called RAII.*

It works by creating a wrapper class, executing

- the acquiring operation (or *lock*-ing in this case) in its constructor, and
- the releasing operation (or *unlock*-ing) in its destructor.

Such helper classes are available in C++11 `std::lock_guard`.

The big advantage is that unlocking the mutex is guaranteed for code blocks defining a RAII-style (guard) object locally, no matter whether control flow reached its end, or by `break`, `return`, or some exception.

*: This TLA is an abbreviation Bjarne Stroustrup once coined for *Resource Acquisition is Initialisation*. In a recent interview Stroustrup revealed that he is not particularly happy with the term he once choose. But to change it would require to travel back in a time machine and suggest something more appropriate to him, as today the term RAII is in much too widespread use to be replaced by something else.

Mutex Variants

Mutexes in C++11 come in a number of flavours, which controlling their behaviour with respect to the following details:

- Whether or not some thread that already locked a mutex may lock it once more (and needs to release it as often).
- Whether or not a thread waits if it finds a mutex locked by some other thread, and in the latter case until *when* (clock-based time point) or or *how long* (duration) it waits.

For all mutex variants there are also variants of RAII-style lock guards.

(i) For more information on the different variants of mutexes and RAII-style wrappers see:
http://en.cppreference.com/w/cpp/thread/recursive_mutex
http://en.cppreference.com/w/cpp/thread/timed_mutex
http://en.cppreference.com/w/cpp/thread/recursive_timed_mutex and
http://en.cppreference.com/w/cpp/thread/unique_lock

C++14: Upgradable Locks

C++14 added the class `std::shared_lock`, supporting a frequent necessity:*

> Multiple Reader/Single Write Locking Schemes

- Any number of (reader) threads may successfully `shared_lock` that kind of mutex ...
- ... but only one single (writer) thread is allowed to actually `lock` it (unshared).

C++14 also provides a RAII-style wrapper for shared locking.

For more information on shared locking see:
http://en.cppreference.com/w/cpp/thread/shared_timed_mutex
and http://en.cppreference.com/w/cpp/thread/shared_lock

*: Note that the terms "reader" and "writer" indicates the typical use of that kind of mutex, assuming that it is sufficient for readers to obtain the lock shared, as it guarantees the invariants hold but no modifications are made, while writers will need to temporarily break invariants.

Defeating Deadlocks Caused by Mutex-Locking

As a potentially blocking mechanism mutexes are famous for creating deadlocks, i.e

- in the situation where two resources *A* and *B* are required,
- one thread acquires these in the order *first A, then B* and
- another thread acquires these in the order *first B, then A*.*

The obvious counter measure is to acquire locks always in the same order, as achievable with `std::lock` and `std::try_lock`.

> **(i)** For more information on locking several mutexes semantically atomic (i.e. without creating the potential for dead-locks) see: http://en.cppreference.com/w/cpp/thread/lock and http://en.cppreference.com/w/cpp/thread/try_lock

*: In practice, the potential for deadlocks is often not as obvious as in this example but much more intricate and close to impossible to spot, even in scrutinising code reviews. So, if (accidental) deadlocks cannot be avoided, sometimes a "self-healing" strategy is applied that works follows:
If more than one lock needs to be acquired, acquire at least all others with setting a time-out. If that hits, **release all locks** acquired so far, **delay** for some small amount of time (usually determined in a window with some slight randomness), then **try again** (and maybe in a different order).

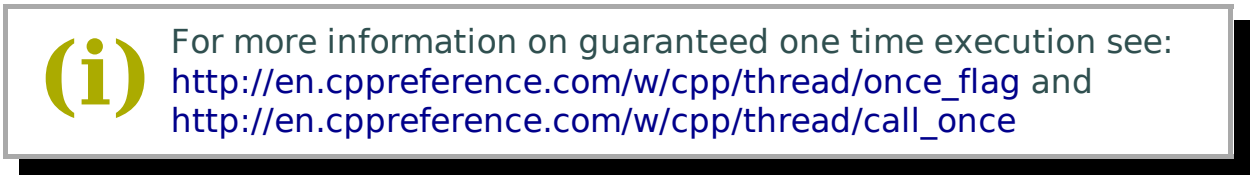
One-Time Execution

For a particular scenario that would otherwise require the use of mutex-es to avoid a Race Condition, there is pre-built solution in C++11.

Executing a piece of code exactly once can be achieved in a cookbook-style as follows:

```
// in a scope reachable from all usage points:
std::once_flag this_code_once;
…
std::call_once(this_code_once, …some callable… ); // somewhere
…
std::call_once(this_code_once, … ); // maybe somewhere else
```

For any of the callables associated with the same instance of an `std::once_flag` via `std::call_once`, without further protection via mutexes it is guaranteed that **at most one** is executed **at most once**.



One-Time Execution Example

A typical use case for guaranteed one-time execution is some initialisation, that may be expensive and is therefore delayed until a function that depends on it is called the first time.

The following fragment avoids parallel initialisations of `table`:

```
… foo( … ) {
    static std::once_flag init;
    static std::array<int, 1000> table;
    std::call_once(init, [&table]) {
        … // precalculate table when foo runs for the first time
    });
    … //
}
```

[!] Note that the above code still has a problem with respect to the initialisation it seems to guarantee …*

*: The example code shown does **not** guarantee that from several concurrently executing threads all will see a **fully** initialised table – it only guarantees sure that the callable to precalculate the content will be executed exactly once and from exactly one thread.

Local `static` Initialisation

Since C++11 supports multi-threading in the core language, initialising local `static` variables is protected to be executed at most once:

```
… foo( … ) {
   static const int z = expensive_calculation();
   … //
}
```

As since C++11 compilers are required to wrap the necessary protection around the initialisation of static locals, also this is guaranteed to work:*

```
class Singleton {
    … //
public:
    static Singleton &getInstance() {
        static Singleton instance;
        return instance;
    }
};
```

*: Non-believers should consider to copy the above code, paste it to https://gcc.godbolt.org/ (or similar) after adding a member with a runtime-dependant initialisation, and view the assembler output ...

Notifications with Condition Variables

A well-known abstraction in concurrent programming are combining mutexes with a signalling mechanism.

One main use of condition variables is to **avoid busy waiting** in producer-consumer designs,

- where consumer and producer run concurrently,
- exchanging data over some buffer data structure.

For more information on condition variables see:
http://en.cppreference.com/w/cpp/thread/condition_variable

Condition Variable Example (1)

The following *RingBuffer* class can put condition variables to good use …

```
template<typename T, std::size_t N>
class RingBuffer {
    std::array<T, N+1> buf;
    std::size_t p = 0, g = 0;
    bool empty() const { return p == g; }
    bool full() const { return (p+1) % buf.size() == g; }
public:
    void put(const T &val) {
        if (full())
            … // handle case no space is available
        buf[p++] = val; p %= buf.size();
    }
    void get(T &val) {
        if (empty())
            … // handle case no data is available
        val = buf[g++]; g %= buf.size();
    }
};
```

… exactly at the currently omitted points.

Condition Variable Example (2)

Obviously there are two conditions, that need special attention:

- The buffer may be full when `put` is called, or
- it may be empty, when `get` is called.

Therefore two condition variables* are added, furthermore a mutex to protect accessing the buffer:

```
class RingBuffer {
    … //
    … // as before
    … //
    std::condition_variable data_available;
    std::condition_variable space_available;
    std::mutex buffer_access;
public:
    … // see next page
};
```

*: As the buffer space cannot be full and empty at the same time, technically one condition variable would suffice, but for this introductory example using two different instances seems to be clearer.

Condition Variable Example (3)

There are two operations (of interest here), applicable to condition variables, **sending** and **waiting for** notifications:*

```
class RingBuffer
   … // see previous page
public:
    void put(const T &val) {
        std::unique_lock<std::mutex> lock(buffer_access);
        space_available.wait(lock, [this]{ return !full(); });
        buf[p++] = val; p %= buf.size();
        data_available.notify_one();
    }
    void get(T &val) {
        std::unique_lock<std::mutex> lock(buffer_access);
        data_available.wait(lock, [this]{ return !empty(); });
        val = buf[g++]; g %= buf.size();
        space_available.notify_one();
    }
};
```

*: Essential here is also the connection between the condition variables, the mutex protecting the `RingBuffer` invariants, and the conditions checked as part of waiting, which are detailed on the next page.

Waiting Anatomy

Waiting on a condition variable – as shown on the last page – with

```
// as part of put:
   std::unique_lock<std::mutex> lock(buffer_access);
   space_available.wait(lock, [this]{ return !full(); });


// as part of get:
   std::unique_lock<std::mutex> lock(buffer_access);
   data_available.wait(lock, [this]{ return !empty(); });
```

is equivalent to the following, **with the mutex being locked before**:

```
// as part of put:
   while (full()) { // !!full()
       buffer_access.unlock();
       // wait for notification
       buffer_access.lock();
   }
```

```
// as part of get:
   while (empty()) { // !!empty
       buffer_access.unlock();
       // wait for notification
       buffer_access.lock();
   }
```

At this point **the mutex is locked** (again) and **the condition is true.**

Spurious Wakeups

In the example code before, the loop (in the equivalent of `wait`-ing) may seem unnecessary, as the respective notification will be sent only after some action has made the condition true.

Nevertheless it makes sense and may even be necessary:

- **First of all, an implementation is allowed to give Spurious Wake-Ups, so the loop is necessary anyway.**
- If notifications are sent while nobody waits on the condition variable, it is simply discarded, therefore
  - a producer-consumer scenario is more robust if it tends to send "too many" notifications (of which some are discarded) ...
  - ... while sending "too few" could cause some thread to wait forever.

> Specifying the condition check in combination with `wait`-ing *does it right* and hence should be preferred over writing a loop explicitly.*

*: Thus avoiding some later maintainer considers the `while` "unnecessary" and replace it with `if` ...

Atomic Operation Support

With the support for atomic operations C++11 multi-threading allows to implement

- Wait-Free Algorithms,
- Lock-Free Algorithms, and
- Obstruction-Free Algorithms.

The basic concept is to provide a way to know whether a certain modification of some memory location was caused by the current thread or by another one.

For more information about atomic operation support see:
http://en.cppreference.com/w/cpp/atomic

Atomic Operations Example

The following example, demonstrating the lock-free approach with operations (instead of using mutexes) modifies a former example:

```
template<typename T1, typename T2, typename Transformation>
void worker(T1 args[], std::size_t data_size, T2 result,
            std::atomic_size_t &total_progress,
            Transformation func) {
    while (total_progress < data_size) {
        constexpr auto chunks = std::size_t{100};
        std::size_t beg = total_progress.load();
        std::size_t end;
        do {
            end = ((data_size - beg) > chunks)
                ? beg + chunks
                : data_size;
        } while (!total_progress.compare_exchange_weak(beg, end));
        std::transform(&data[beg], &data[end], &sqrts[beg], func);
    }
}
```

Be sure to understand that the loop controlled by the return value of `compare_exchange_weak` guarantees the prior calculations are (still) valid.

Memory Order (and Dependencies)

Memory order "*specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation*".*

There are a number of different models to select from,

- with the default providing "*sequentially consistent ordering*",
- being closest to what most developers would expect,
- but (possibly) not with the optimal performance for a given use case.

For more information on memory order see:
http://en.cppreference.com/w/cpp/atomic/memory_order

*: Cited from the reference further down on this page.

Atomic Operations Recommendation

- Except for the potential of deadlocks the challenges are similar to algorithms using locks:*
  - Problems may only show for a critical timing and may be reproducible in particular test environments only.
  - In general, a failed test may show the presence of errors, but even many successful tests do not guarantee their absence.

[!] Beyond trivial cases – like the one shown in the example –, implementing multi-threaded programs with atomic operations requires **substantial expertise**.

Be sure to keep the design **as simple as possible** and have it reviewed by other developers experienced in that particular fields, maybe both, colleagues and hired consultants too.

*: It may very well be the case that problematic situations depend on hardware features like the size of cache-lines, the depth of an instruction pipeline, or the way branch prediction works.

Using Class `std::thread`

In the examples before the class `std::thread` was used only indirectly (via `std::async`, building on Futures and Promises.

- There is also a class `std::thread`
- taking any runnable code as constructor argument,
- executing it in a separate thread.

For more information on the `std::thread` class see:
http://en.cppreference.com/w/cpp/thread

There are explicitly no means provided to forcefully terminate one thread from another one.

Any non-portable way* to forcefully terminate a thread risks to entail serious consequences later, as e.g. locks may not be released or awaited notifications not be sent.

*: Such as potentially existing *interrupt* or *kill* functions reachable via a Native Interface Handle.

Example for Using Class `std::thread`

In case worker **returns no value** and **throws no exception**, like in a former example (and assuming the same set-up), using class `std::thread` directly can be straight forward (left) or done in the "nifty" way (right):

```
// starting some workers
// multi-threaded ...
std::thread t1{worker_task};
std::thread t2{worker_task};
…
// ... and waiting for them
// to finish:
t1.join();
t2.join();
…
```

```
using namespace std;
constexpr auto NCORES = 4;
// starting one worker-thread
// per core ...
array<thread, NCORES> threads;
for (auto &t : threads)
    t = thread{worker_task};
// ... and wait for all to
// finish:
for (auto &t : threads)
    t.join();
```

An alternative to `join`-ing with a thread is `detach`-ing it.

[!] A program will immediately terminate if an instance of class `std::thread` referring to an active thread gets destructed.

Recommendations for Using Class `std::thread`

- In trivial cases (no exceptions, no result to fetch) using threads via instances of class `std::thread` may be considered.
- Nevertheless understand the peculiarities and know how to avoid race conditions, especially when `std::thread` objects go out of scope.*

[!] It causes the program to terminate if the callable started throws an exception or an instance of `std::thread` is still in *joinable state* when it goes out of scope and gets destructed.

*: At 1:00:47 the following video by Scott Meyers gives a good introduction to the problem and some recipes how it can be avoided: http://channel9.msdn.com/Events/GoingNative/2013/An-Effective-Cpp11-14-Sampler

Native Handles

Last and finally, most C++11 multi-threading implementation build on a threading model provided by their execution environment.

> The requirements in the standard are rather the intersection of the features provided by well-known threading models.

Usually and typically the standard does not mandate any any extensions thereof, but in some cases provides a way to "reach through" to the native thread model:

- E.g. `std::thread::native_handle` could provide ways to manipulate thread priorities, maybe including ways to specify protocols for Priority Ceiling or other means to circumvent Priority Inversion.
- Also the classes for condition variables and the various kinds of mutexes have member functions `native_handle`.
- The enumeration `std::launch` may provide more (named) values to control implementation specific details in the behavior of `std::async`.

Concurrency Recommendations

So far the presentation of C++11 concurrency support was only meant as an overview.

[!] Practically using concurrency features beyond parallelizing independent tasks requires much more knowledge and experience in this area, what this presentation can not provide.

A good and near to exhaustive coverage of the concurrency part of C++11 is:

C++ Concurrency in Action
Practical Multithreading
by Anthony Williams
ISBN-13: 978-1-9334988-77-1

Boost: Threads Library

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

Some differences 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 upcoming C++ standard.

There are many reasons why a pure library solution may have practical difficulties to properly support concurrency.

Most important is in the memory model, which needs to be clearly defined so that optimising compilers are restricted to the necessary limits – but not (too far) beyond these.*

*: A related, easy to recognize problem that demonstrates why concurrency cannot be added as a library alone but must be part of the core language part, can be seen when considering a long-existing features of C which is also part of C++: the initialization of local `static` variables. By definition this takes place when the definition is reached in the program flow for the first time and hence, in the general case must be protected with a mutex (or similar) to avoid race conditions (except for initialisation with a compile-time constant that can be loaded at program startup).

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 asynchronous sources (like sockets).
- Besides that it allows also an applications to send events to itself, maybe with a specified delay.

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.