

C++ Concurrency

1

© 2023 Peter Sommerlad

C++ Concurrency Awareness

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad (🐦)

Slides:



<https://github.com/PeterSommerlad/CppConcurrencyAwareness/>

3

© 2023 Peter Sommerlad

My philosophy

Less Code

=

More Software

4

© 2023 Peter Sommerlad

Speaker notes

I borrowed this philosophy from Kevlin Henney.

© 2023 Peter Sommerlad

In this course

- Races and UB
- Typical Pitfalls
- Making code concurrency resilient
- Improve code structure and design

Not in this course

- *All* of C++
- Building C++ on the command line
- C++ build systems (cmake, scons, make)
- C++ package manager (conan, vcpkg)
- other C++ Unit Test Frameworks (Catch2, GoogleTest)
- C++98
- Other C++ IDEs (vscode, clion)

6

© 2023 Peter Sommerlad

Speaker notes

You can observe the command line used by Codeload in its Console window.

We will not look at all features of C++20 and only some of the limitations of previous C++ language standards.

© 2023 Peter Sommerlad

C++ Resources

- ISO C++ standardization
- C++ Reference
- Compiler Explorer
- C++ Core Guidelines
- Hacking C++ reference sheets

7

© 2023 Peter Sommerlad

Speaker notes

complete link texts, PDF generation via browser causes hyperlinks to vanish from slide part, but should stick in the notes part:

- <https://isocpp.org/>
- <https://en.cppreference.com/w/>
- <https://compiler-explorer.com/>
- <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- <https://hackingcpp.com/>

© 2023 Peter Sommerlad

C++ Genealogy

C++98

initial standardized version

C++03

bug-fix of C++98, no new features

C++11

major release (known as C++0x): lambdas, constexpr, threads, variadic templates

C++14

fixes and extends C++11 features: variable templates, generic lambdas

C++17

(almost) completes C++11 features: CTAD, better lambdas

C++20

new major extension: concepts, coroutines, modules, constexpr "heap"

C++23

feature-complete (2022-02), fixes/extends C++20

C++26

work has already started, first of my proposals accepted :) (2023-11)

8

© 2023 Peter Sommerlad

Speaker notes

The ISO standardization process uses a three year release cycle since 2011. However, for major releases it takes time for implementors to provide the new language features and library. Most C++ compilers do not yet have fully implemented C++20 and some implementation diverge in subtle details, because the specification is inaccurate. This is a typical chicken-egg problem: * compilers will only implement language features in production quality, when they are part of the standard * specification in the standard is only scrutinized when independent compiler/library authors implement them

C++20 modules and coroutines are not yet generally usable across compilers. Concepts are. The ranges library similarly is not "complete" for C++20 in all compilers, so I won't cover it here.

© 2023 Peter Sommerlad

Understanding Concurrency Challenges

Concurrency is hard, even world-class experts make mistakes!

10

© 2023 Peter Sommerlad

Speaker notes

This provides an oversimplified conceptual presentation of hardware evolution to give some indication why it is so hard to reason about the correctness of multi-threaded/concurrent code.

© 2023 Peter Sommerlad

simple (and wrong) thread example

```
1 #include <iostream>
2 #include <thread>
3 struct
JnsynchronizedCounter {
4     void increment() & {
5         ++value;
6     }
7     int current() const {
8         return value;
9     }
10 private:
11     int value{};
12 };

20 int main () {
21     UnsynchronizedCounter counter{};
22     auto run = [&counter]{} // !!
23     for (int i = 0; i < 100'000'000; ++i) {
24         counter.increment();
25     }
26 };
27 std::thread t1{run};
28 std::thread t2{run};
29 t1.join();
30 t2.join();
31 std::cout << "Counter " << counter.current() << "
result\n" ;
32 }
```

result is (almost always) wrong!

<https://godbolt.org/z/K5xdKKd4E>

11

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/K5xdKKd4E>

The shared resource of a counter is passed to the thread function by reference (lambda capture). This almost always indicates a problem, unless the resource is read-only (const/constexpr). Here the mutation by different threads is intentional and causes undefined behavior!

Turning on optimization might flatten the loop and accidentally cause the correct result.

© 2023 Peter Sommerlad

What is a race condition?

- You walk into town and see a nice T-shirt in shop window
- You think about it and decide to buy it
- But you first need to pick up your prescription at the pharmacy, because it closes soon
- When you return to the shop, the T-shirt is no longer on display
- You enter and ask for the T-shirt that was on display
- The shop keeper tells you, that the last one was just sold

Interleaving between the decision based on a condition and the acting up, the condition changed

12

© 2023 Peter Sommerlad

Speaker notes

Race conditions happen in real life and in computers.

Database systems can attempt to prevent race conditions with “pessimistic locking”, or detect the occurrence of a race condition with “optimistic locking”.

While the pessimistic approach limits concurrency, the later can result in user dissatisfaction through cancelled transactions.

© 2023 Peter Sommerlad

Hardware Evolution

Hardware gets faster through concurrency

- Single Processor / Single Core (1980s)
- Multi-Processor (1990s)
- Multi-Core Processor (2000s)

13

© 2023 Peter Sommerlad

Speaker notes

We will look at an oversimplified evolution to have an impression of the underlying complexity of synchronisation and also to develop a mental model, why simple sequential reasoning is not working for concurrent and parallel code.

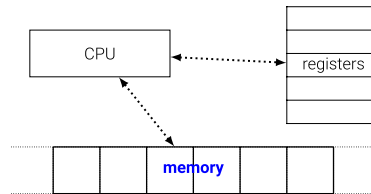
In reality, both hardware as well as the underlying software is even more complex.

Ignored here: Hardware-Pipelining, Hardware Instruction Reordering, Vectorization, speculative Execution, Compiler Optimizations, Multi-level Caches, Memory Fences, different memory models (not only sequential consistency), etc.

Even hardware people can get things wrong. For Example, nvidia found bugs in their GPU architecture after mathematical modelling it and using a theorem prover to check if desired properties actually hold. See talk: The One-Decade Task: Putting std::atomic in CUDA. - Olivier Giroux - CppCon 2019, <https://youtu.be/VogqOscJYvk?feature=shared>

© 2023 Peter Sommerlad

Classic Sequential Computing



- CPU (central processing unit)
- Registers
- Memory (RAM)

14

© 2023 Peter Sommerlad

Speaker notes

This sequential computing mental model was well working for many computers up to the 1990s.

However, it is already a simplification and can result in race conditions with multiple threads.

What can already happen that can be confusing, that when using multiple threads of computation that access the same memory location that the actual value of a computation by one thread is still in a register and not yet written to the memory location where the actual variable resides.

So updates of a variable consist of three phases:

- read value from memory into register (see)
- perform the operation on the register (decide) - possible interruption
- write value (commit)

when the computation of one thread is interrupted after the read operation but before the resulting value is written to the variable in memory, the writing might overwrite an already updated value and thus storing the wrong value.

© 2023 Peter Sommerlad

Classic Synchronisation (pre C++11/C11)

single core - real-time system

- shared **volatile** variables between interrupt handler and main loop
- manipulation/reading of shared variable under disabled interrupts
 - goal, don't miss interrupts, short disabling times
- without disabling interrupts: corrupted data possible
- “atomic” read-write memory access guarantees?

Do NOT USE volatile for Synchronization in C++

15

© 2023 Peter Sommerlad

Speaker notes

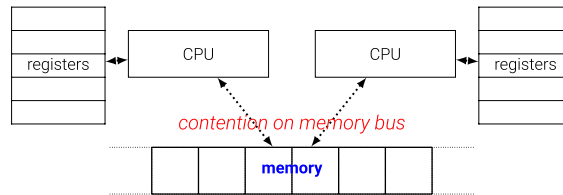
Data is usually shared between interrupt handlers and the main loop functions via **volatile** variables.

Using **volatile** for sharing data across threads

DOES NOT WORK

© 2023 Peter Sommerlad

Multi-Processor



- 2+ CPUs
- independent Register sets
- Shared Memory Bus
- Memory (RAM)
- Synchronization is expensive

16

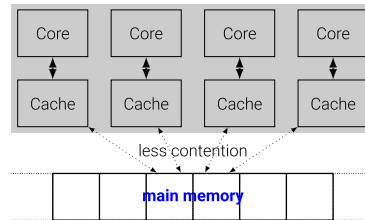
© 2023 Peter Sommerlad

Speaker notes

When multi-processors were introduced to increase processing speed, sharing the main memory lead to contention. While the hardware memory bus was able to arbitrate memory accesses of different CPUs, sharing variables between different threads that might run on different CPUs requires synchronization, which becomes more expensive, because all CPUs must be notified of a new value written. You can imagine that for synchronization, all CPUs must stop what they are doing at the moment, wait for the variable to be written and then continue , ay be re-reading the memory location of the written variable.

© 2023 Peter Sommerlad

Multi-Core with Caches



- Caches foster independent processing
- Accessing main memory expensive
- Synchronization is even more expensive

17

© 2023 Peter Sommerlad

Speaker notes

While this picture is over-simplified, multi-core processors today have per-core (and shared) cache memory. This memory allows each core to operate mostly independent of other cores. However, when a multi-threaded program runs spread over several cores, any mutable variable that is accessed by more than one thread requires synchronization (atomics or mutex).

In addition to stopping the processing in the cores, also the contents of the caches must be updated, if affected by a variable in main memory that was written.

© 2023 Peter Sommerlad

Concurrency is even harder!

Compilers and Hardware can reorder memory operations

code says:

```
a = 1  
b = 2  
c = 3  
output c  
output a  
output b
```

executed (e.g.):

```
c = 3  
a = 1  
output c  
b = 2  
output a  
output b
```

18

© 2023 Peter Sommerlad

Speaker notes

Compilers can reorder writes to memory locations of objects that are independent.

(modern) Hardware reorders loads and stores of memory locations to reduce contention and increase processor pipeline throughput, sometimes very aggressively and on multiple stages. Hardware does not take logical dependencies into account for that!

This is not C++ code and just symbolizes memory writes/stores (by assignment to a “variable”) and memory reads/loads by “output of a variable”

This slide is shown again later...

© 2023 Peter Sommerlad

Expensive Synchronisation

- when synchronisation is required it can be **expensive**
- bring all cores to a halt
- stalls cores' pipelines
- synchronize caches
- make writes visible (write through/read through)
- regardless of the mechanism (atomics, mutex)

volatile for sharing data across threads
DOES NOT WORK

19

© 2023 Peter Sommerlad

Speaker notes

On multi-core/multi-processor systems, mutexes and atomic access can be very expensive compared to regular memory accesses.

© 2023 Peter Sommerlad

Implications for Software

*Accessing a mutable variable from multiple threads is a **data race** and thus **Undefined Behavior***

- Synchronize shared data access (atomic, mutex, etc)
- Share as little mutable(=non-const) data as possible
 - **static** variables are dangerous, even local ones
- **const** data can be shared without danger
 - if not containing **mutable** members
 - Do not cast away **const**
- Pass input data to a thread by value
- Return thread results via **std::future**

20

© 2023 Peter Sommerlad

Speaker notes

Many modern C++ good practices make it easier to write multi-threaded code, e.g., "Define variables as const as possible and as local as possible!", "Pass parameters and return by value!". However, consciously designing code that runs in a multi-threaded environment still requires care.

When a lot of existing code that is used as a model to copy contains bad practices it is hard to judge, if those practices were introduced deliberately or accidental mistakes.

© 2023 Peter Sommerlad

C++ Parallelism and Concurrency

- parallelism and concurrency are hard
- even world-class experts get it sometimes wrong
- required synchronizations can make system slower
- problem must suit parallel architecture

22

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Concurrency Problems

- race conditions
 - check and acting upon result not atomic
- data race: **undefined behavior**
 - concurrent access of shared mutable data
- deadlock: circular blocking waits
- starvation: unfair wake up
- livelock: circular non-blocked waits

23

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

What is a race condition?

- You walk into town and see a nice T-shirt in shop window
- You think about it and decide to buy it
- But you first need to pick up your prescription at the pharmacy, because it closes soon
- When you return to the shop, the T-shirt is no longer on display
- You enter and ask for the T-shirt that was on display
- The shop keeper tells you, that the last one was just sold

Interleaving between the decision based on a condition and the acting up, the condition changed

24

© 2023 Peter Sommerlad

Speaker notes

Race conditions happen in real life and in computers.

Database systems can attempt to prevent race conditions with “pessimistic locking”, or detect the occurrence of a race condition with “optimistic locking”.

While the pessimistic approach limits concurrency, the later can result in user dissatisfaction through cancelled transactions.

© 2023 Peter Sommerlad

Parallelization?

- Suitable Problem?
- Architecture first!
- Minimize need for synchronisation!
- Prove that there are no
 - data races
 - deadlocks
 - other bad behaviour
- Even world-class experts make mistakes!

25

© 2023 Peter Sommerlad

Speaker notes

To parallelize a problem, one needs to map the problem to a suitable architecture first. There exist many architectural/design patterns on how to arrange computation in a parallelizable way. For example: Processing Pipelines/Pipes & Filters, Embarrassingly Parallel, Map-Reduce, ...

For real-time applications:

For example, automotive control units use special real-time multi-cores that operate in lock-step to enable bounding instructions and simplify synchronization.

General guidance: separate real-time critical parts from uncritical parts and look out for accidental priority inversion/deadline misses.

© 2023 Peter Sommerlad

Concurrency-Safe C++

What can be used safely in concurrent C++?

- named constants (**const**, **constexpr**)
 - read-only is thread-safe, even if shared
- non-static local variables
 - unless shared
- static const local variables
 - initialization is guaranteed to occur once
- value parameters
 - local variable with copy
- return by value
 - does not work across threads: **std::future**
- **std::atomic<T>** variables
 - data-race free, but race prevention requires appropriate use
- **thread_local** variables

26

© 2023 Peter Sommerlad

Speaker notes

A lot of C++ features can be dangerous when used across threads. However, a few things are safe to use unless deliberately shared across threads, e.g., by passing a reference or pointer to a local variable to a thread.

© 2023 Peter Sommerlad

Typical Concurrency Pitfalls

- unsynchronized shared mutable data
 - non-const members in concurrently accessed objects
 - mutable members cheating in const-member functions
 - non-const variables with static storage duration
 - passing non-const objects to other threads via pointers or references
- partially protected data
 - forgetting to lock mutex
 - accidental sharing of `thread_local` variables via references
- exception-unsafe explicit lock/unlock pairs
 - like other resources with explicit alloc/release function pairs
- deadlocks
 - self-deadlock while holding a mutex
 - with multiple mutexes
 - avoidable through `std::scoped_lock`
 - architecture important

27

© 2023 Peter Sommerlad

Speaker notes

Data races are undefined behavior, so do not share variables explicitly or implicitly across multiple execution agents.

If something is `const`, do not cast away `const`, or make unprotected members `mutable` to cheat.

Even experts make mistakes, so be extra careful

© 2023 Peter Sommerlad

Unsynchronized shared mutable data

- non-const members in concurrently accessed objects
- **mutable** members cheating in const-member functions
- non-const variables with static storage duration
 - also **static** locals, even when init is synchronized
- passing non-const objects to other threads via pointers or references

declare variables as **const** and as local as possible!

28

© 2023 Peter Sommerlad

Speaker notes

In general:

- minimize the need for synchronization by reducing mutable variables shared across threads.
- no global or static variables that aren't const, except for atomics
- do not share mutable objects across threads (unique_ptr's ownership model helps for heap-allocated objects)

All the above things can be found in CERN's code base (unfortunately), please refactor and correct!

© 2023 Peter Sommerlad

Partially protected data

```
private:  
mutable std::mutex mutex_;
```

```
std::scoped_lock const lock{mutex_};
```

- forgetting to lock mutex
- `std::scoped_lock{mutex_};` - locks and release immediately
- accidental sharing of `thread_local` variables via references

Synchronized access must be consciously be designed

Scoped Locking: `std::scoped_lock` `std::unique_lock`

Thread-safe Interface: public mem-funs lock, private assume locked

29

© 2023 Peter Sommerlad

Speaker notes

See for example the pattern: **Scoped Locking** and **Thread-safe Interface** in Pattern-oriented Software Architecture Volume 2

© 2023 Peter Sommerlad

Exception-unsafe Locking

explicit `lock()`/`unlock()` calls

```
class AlgResourcePool : public extends<Service, IAlgResourcePool> {  
    //...  
    std::mutex m_resource_mutex;
```

```
    StatusCode AlgResourcePool::acquireResource( std::string_view name  
) {  
    m_resource_mutex.lock();  
    m_available_resources[m_resource_indices[name]] = false; // what  
    if this throws  
    m_resource_mutex.unlock();  
    return StatusCode::SUCCESS;  
}
```

30

© 2023 Peter Sommerlad

Speaker notes

There is almost no excuse to not use Scoped Locking in C++.

Prefer `std::scoped_lock` over the legacy `std::lock_guard`, because it allows to correctly lock multiple mutexes.

In case of using a condition variable, that requires temporarily releasing the lock, use `std::unique_lock` instead.

However, some Gaudi components still use `std::mutex::lock()/unlock()` explicitly without `std::scoped_lock/std::unique_lock/std::lock_guard`

<https://gitlab.cern.ch/gaudi/Gaudi/-/blob/6babff9bec280f4e70f6d8fd45df795d5638c0bf/GaudiHive/src/AlgResourcePool.cpp#L160>

Exercise: Can you argue that the following function is safely using `.unlock()` instead of Scoped Locking?

<https://gitlab.cern.ch/gaudi/Gaudi/-/blob/6babff9bec280f4e70f6d8fd45df795d5638c0bf/GaudiHive/src/AlgResourcePool.cpp#L80>

```
if ( sc.isSuccess() ) {  
    state_type requirements = m_resource_requirements[algo_id];  
    m_resource_mutex.lock();  
    if ( requirements.is_subset_of( m_available_resources ) ) {  
        m_available_resources ^= requirements;  
    } else {  
        sc = StatusCode::FAILURE;  
        error() << "Failure to allocate resources of algorithm " << name << endlmsg;  
        // in case of not reentrant, push it back. Reentrant ones are pushed back  
        // in all cases further down  
        if ( !algo->isReentrant() ) { itQueueIAlgPtr->second->push( algo ); }  
    }  
    m_resource_mutex.unlock();  
    if ( algo->isReentrant() ) {  
        // push back reentrant algorithms immediately as it can be reused  
        itQueueIAlgPtr->second->push( algo );  
    }  
}
```

© 2023 Peter Sommerlad

Deadlocks

Thread(s) wait circularly to lock a mutex, while holding another

- self-deadlock while holding a mutex
- with multiple mutexes
 - avoidable through `std::scoped_lock` or `std::lock` algorithm
 - architecture important

Avoid self-deadlock through Thread-safe Interface Pattern

- single mutex per object
- public member functions lock
- private member functions assume lock
- no public member functions called within the class
- alternative: recursive mutex

31

© 2023 Peter Sommerlad

Speaker notes

While recursive mutexes seem like a good idea to avoid self-deadlock, they tend to be used to cover up design problems.

The pattern Thread-safe Interface separates the actual functionality into private member functions and the public member functions will always acquire a lock and call the underlying functionality in private member functions while holding a lock. Care must be taken, never to call a public member function while already holding the lock.

© 2023 Peter Sommerlad

C++ standard concurrency

- parallel algorithms
- `async` and `futures`
- `jthread` (C++20) and `thread`
- `mutex`, `scoped_lock` and `unique_lock`
- `condition_variable`
- `atomic<T>`
- `thread_local` storage class specifier
- (C++20 Coroutines (`co_await`, `co_yield`, `co_return`))

33

© 2023 Peter Sommerlad

Speaker notes

The above lists tries to be sorted by “ease of use”, but is not necessarily that one would use `async()`, when there is no parallel algorithm.

Unfortunately, the design of higher-level parallelization infrastructure for the C++ standard library was highly contentious and is still in flux. This includes the library support for C++20 Coroutines. Because of the lacking library support and also the still lacking implementation of coroutines in the major compilers, we won't look at those during this course.

© 2023 Peter Sommerlad

C++ parallel algorithms

- most of `<algorithm>`, new ones in `<numeric>`
- execution policy as additional first parameter (`std::execution::`)
 - `seq`, `par`, `par_unseq`, `unseq` (C++20)
- `par` might start new threads
- `par_unseq`, `unseq` use vectorization
- data elements and operations must be independent for parallelization
- new names in `<numeric>`, e.g.,
 - `accumulate()` becomes `reduce()`
 - `inner_product` -> `transform_reduce`
- best with `vector` and `array` of trivial types

34

© 2023 Peter Sommerlad

Speaker notes

There are restrictions on data access patterns and what types can be used for parallelization and vectorization.

Not all implementations actually parallelize the algorithms, sequential execution is conforming, even when the execution policy is not `seq`.

© 2023 Peter Sommerlad

parallel algorithm example

```
#include <vector>
#include <iostream>
#include <numeric>
#include <execution> // defines policies
int main()
{
    std::vector<double> xvalues(10007, 2.0), yvalues(10007, 2.0);
    double result = std::transform_reduce(
        std::execution::par,
        xvalues.begin(), xvalues.end(),
        yvalues.begin(), 0.0
    );
    std::cout << result << '\n';
}
```

<https://godbolt.org/z/6MMrbr5dW>

35

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/6MMrbr5dW>

Even though execution policy `par` is given, it is not parallelized on major compilers.

```
.L7:
movsd xmm0, QWORD PTR [rbx+rdx*8]
mulsd xmm0, QWORD PTR [rax+rdx*8]
add    rdx, 1
addsd  xmm1, xmm0
cmp    rdx, 10007
jne    .L7
```

Actually, with Intel Thread-building Block library activated and -O3 code will be parallelized if data is big enough on GCC.

© 2023 Peter Sommerlad

C++ simple `async()`

`std::async()` takes a function object to execute

```
#include <future>
#include <iostream>
#include <cmath>
double do_some_expensive_calculation(double input){
    return std::sqrt(input); // simulate that
}
int main(){
    auto future = std::async(std::launch::async,
                             do_some_expensive_calculation,81.0);
    std::cout << "do some other useful things\n" << std::flush;
    std::cout << "the result is= "<< future.get()<< '\n';
}
```

*`std::launch::async` = new thread
`std::launch::deferred` = run on `future.get()`*

36

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

C++ `async()` gotchas

- must obtain the future object, even if `void`
 - otherwise, `async()` is synchronous
- starting a new thread is expensive
- pass data by value to the function executed, otherwise dangling or data races can occur
 - future might be returned and scope left
 - `launch::async` starts a new thread
- `future::get()` is one-shot
(`std::shared_future`)

37

© 2023 Peter Sommerlad

Speaker notes

While `async()` was meant to be simple to use, the mixing of deferred execution with concurrent/parallel execution leads to some potentially surprising behavior. For example, the `std::launch::async` policy will start a separate thread to execute the provided function. This causes the returned future object to wait for that thread's termination, either when the value is obtained, or when the future object is destroyed. Not keeping the future returned by `async`, will cause its destructor to immediately wait for the thread started to terminate. That way it becomes a very expensive sequential call.

© 2023 Peter Sommerlad

Bad `async()` example

```
1  #include <future> // defines async and future
2  #include <iostream>
3  #include <chrono>
4  unsigned long long fibo_def(unsigned long long n){
5      if (n < 1) return 0;
6      if (n < 2) return 1;
7      auto f1 = std::async( fibo_def,n-1); // obtain future, may be start thread
8      auto f2 = std::async( fibo_def,n-2); // obtain future, may be start thread
9      return f1.get() + f2.get(); // use future's result
10 }
11 int main(int argc, char **argv){
12     if (argc < 2 || atoi(argv[1]) < 1) {
13         std::cout<< "Usage: "<< argv[0] << " number\n";
14         return 1;
15     }
16     unsigned long long n=std::strtoull(argv[1],nullptr,10);
17     using namespace std::chrono_literals;
18     using Clock=std::chrono::high_resolution_clock;
19     Clock::time_point t0 = Clock::now(); // standard millisecond timing
20     auto fibn=fibo_def(n);
21     Clock::time_point t1 = Clock::now();
22     std::cout <<"fibo_def("<<n<<") = " << fibn<< " : " << (t1 - t0)/1ms << "ms\n";
23 }
```

<https://godbolt.org/z/oEKzxWYG5>

38

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/oEKzxWYG5>

© 2023 Peter Sommerlad

Futures and underlying features

- `future<T>` - one-time ticket for a promised result
- `shared_future<T>` - multi-reader ticket
 - requires copyable T
 - a future object has `.share()` to create a `shared_future`
- `promise<T>` - shared value holder referred by `future`
- `packaged_task` - abstracts a later invocation, provides a `future`

promise with future can signal state across thread boundaries

39

© 2023 Peter Sommerlad

Speaker notes

for your own notes

© 2023 Peter Sommerlad

Guaranteeing single execution

multiple threads might try to initialize stuff, that must only be initialized once

- local **static** variables are initialized once.
- DIY can use
 - `std::call_once()` - call a function once successfully
 - `std::once_flag` - used to interlock `call_once` calls
 - throwing an exception means unsuccessful

[Example cppreference](#)

40

© 2023 Peter Sommerlad

Speaker notes

for your own notes

© 2023 Peter Sommerlad

C++20 `jthread` vs. `thread`

- both for a thread of execution running the function argument
- `jthread` joins thread in its destructor
- `thread` terminates in its destructor if not joined
- `jthread` provides “stop token” a cooperative interruption mechanism

41

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

simple (and wrong) thread example

```
1 #include <iostream>
2 #include <thread>
3 struct
JnsynchronizedCounter {
4     void increment() & {
5         ++value;
6     }
7     int current() const {
8         return value;
9     }
10 private:
11     int value{};
12 };

20 int main () {
21     UnsynchronizedCounter counter{};
22     auto run = [&counter]{} // !!
23     for (int i = 0; i < 100'000'000; ++i) {
24         counter.increment();
25     }
26 };
27 std::thread t1{run};
28 std::thread t2{run};
29 t1.join();
30 t2.join();
31 std::cout << "Counter " << counter.current() << "
result\n" ;
32 }
```

result is (almost always) wrong!

<https://godbolt.org/z/K5xdKKd4E>

42

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/K5xdKKd4E>

The shared resource of a counter is passed to the thread function by reference (lambda capture). This almost always indicates a problem, unless the resource is read-only (const/constexpr). Here the mutation by different threads is intentional and causes undefined behavior!

Turning on optimization might flatten the loop and accidentally cause the correct result.

© 2023 Peter Sommerlad

better (not good) thread example

```
1 #include <iostream>
2 #include <thread>
3 struct ConcurrentCounter {
4     void increment() {
5         m.lock();
6         ++value;
7         m.unlock(); // problematic with
8     }
9     int current() const {
10         m.lock();
11         int const current = value;
12         m.unlock(); // problematic with complex
13         return current;
14     }
15 private:
16     mutable std::mutex m{}; // will change in
17     int value{};
18 };

20 int main () {
21     ConcurrentCounter counter{};
22     auto run = [&counter]{
23         for (int i = 0; i < 100'000'000;
24             counter.increment();
25         };
26     };
27     std::thread t1{run};
28     std::thread t2{run};
29     t1.join();
30     t2.join();
31     std::cout << "Counter "
32               << counter.current() << "
33               << result\n" ;
34 }
```

result is correct now, but much slower!

<https://godbolt.org/z/h1ez43EPd>

43

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/h1ez43EPd>

Explicitly using the pair of `std::mutex` member functions `lock()` and `unlock()` can break, for example, when an exception is thrown while locked or when the control flow is complex and accidentally a branch returns without unlocking. Therefore, see next slide!

© 2023 Peter Sommerlad

better with `scoped_lock`

```
lock{m};
lock{m};

1 struct ConcurrentCounter {
2     void increment() {
3         std::scoped_lock
4             ++value;
5         } // unlocks always
6     int current() const {
7         std::scoped_lock
8             return value;
9         } // unlocks always
10 private:
11     mutable std::mutex m{};
12     int value{};
13 };

20 int main () {
21     ConcurrentCounter counter{};
22     auto run = [&counter]{
23         for (int i = 0; i < 100'000'000; ++i) {
24             counter.increment();
25         }
26     };
27     std::thread t1{run};
28     std::thread t2{run};
29     t1.join();
30     t2.join();
31     std::cout << "Counter " << counter.current() << "
32     result\n" ;
33 }
```

result is correct now, and safely locked!

<https://godbolt.org/z/zKvn9GqYs>

44

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/zKvn9GqYs>

`std::scoped_lock` always releases the lock in its destructor, regardless if a locking function returns or throws and exception.

In addition `std::scoped_lock` can lock multiple mutexes at once without causing deadlocks, when all such uses use the same set.

© 2023 Peter Sommerlad

better with `atomic<int>`

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4
5  struct AtomicCounter {
6      void increment() {
7          ++value;
8      }
9      int current() const
10         return value;
11     }
12 private:
13     std::atomic<int>
14 };

20 int main () {
21     AtomicCounter counter{};
22     auto run = [&counter]{
23         for (int i = 0; i < 100'000'000; ++i) {
24             counter.increment();
25         }
26     };
27     std::thread t1{run};
28     std::thread t2{run};
29     t1.join();
30     t2.join();
31     std::cout << "Counter " << counter.current() << "
32 }
```

For “simple” types and operations use `std::atomic<T>`

<https://godbolt.org/z/1Wzsf8qGP>

45

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/1Wzsf8qGP>

For simple values the `std::atomic<T>` wrapper can be used. It is optimized to use the most appropriate mechanism for simple types, such as `int` which also provide “specializations” for operations, such as increment (**operator++**). For user-defined types (which must provide trivially copyable, i.e., not have non-trivial members), the library can chose to use a `std::mutex` to achieve the required atomicity and usually only allows exchanging, reading and writing a value.

The specialized atomics are defined to be compatible/identical to the C11 atomics and thus provide interoperability.

Without further flags, atomic variables provide sequential consistency (`std::memory_order_seq_cst`). However, using atomic variables allows to employ more relaxed memory order flags (relaxed, acquire-release). Use of those flags might allow better performance and more interleaving on some hardware, but is less intuitive and can have surprising results. Correct application in “lock-free” data structures usually calls for a formal proof that all required properties are still correct. Very easy to make things wrong.

“relaxed” atomic for example guarantee data-race-free code (no UB in that respect), but usually do not guarantee visibility of a specific value across threads. incrementing counters can be an example.

© 2023 Peter Sommerlad

Memory-order with `atomic<int>`

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4
5  struct AtomicCounter {
6      void increment() {
7          value.fetch_add(1, relaxed);
8      }
9
10     int current() const {
11         return value.load(relaxed);
12     }
13 private:
14     constexpr static auto
15     relaxed{std::memory_order_relaxed};
16     std::atomic<int> value{};
17 };
```

```
20 int main () {
21     AtomicCounter counter{};
22     auto run = [&counter]{
23         for (int i = 0; i < 100'000'000; ++i) {
24             counter.increment();
25         }
26     };
27     std::thread t1{run};
28     run(); // at least 100'000'000 seen
29     t1.join();
30     std::cout << "Counter "
31               << counter.current() << " result\n" ;
32 }
```

Result might be incomplete on some hardware (not Intel)

<https://godbolt.org/z/v8osxT33a>

46

© 2023 Peter Sommerlad

Speaker notes

on X86 only sequential consistency is provided by the hardware. PowerPC or ARM for example, have more relaxed atomic operations and thus can deliver faster concurrency at higher risk to get things wrong.

<https://godbolt.org/z/v8osxT33a>

© 2023 Peter Sommerlad

Higher-level synchronization

Data structures require more than just mutual exclusion

- `condition_variable` - wait on a status protected by a `mutex`
 - use `unique_lock` instead of `scoped_lock`
- `condition_variable_any` - generic CV
- `notify_at_thread_exit` - `notify_all()` CVs

47

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

C++20 Higher-level synchronisation

- `latch` - one-time synchronisation barrier
- `barrier` - cyclic synchronisation barrier
- `binary_semaphore` - semaphore with two states
- `counting_semaphore` - semaphore with non-negative count
- `osyncstream` - wrap `std::ostream` for non-UB, non-intermixed output

These are still quite low-level building blocks

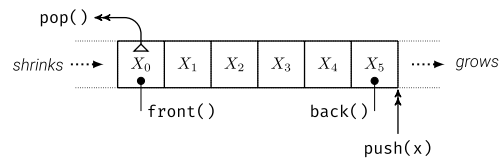
48

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Queue



pop()** needs to wait when **empty()

Producer-Consumer Queue

```
1  #include <condition_variable>
2  #include <mutex>
3  #include <queue>
4  namespace TSQ {
5  struct ThreadsafeQueue {
6      void push(T t) {
7          std::scoped_lock const
k { mx };
8          q.push(std::move(t));
9          notEmpty.notify_one();
10         }
11         T pop() {
12             std::unique_lock lk {
x };
13             notEmpty.wait(lk,
14                 [this] {return
q.empty();});
15             T t =
td::move(q.front());
16             q.pop();
17             return t;
18         }
19     };
20     bool empty() const {
21         std::scoped_lock const lk { mx };
22         return q.empty();
23     }
24     void swap(ThreadsafeQueue & other) {
25         if (this == &other) {
26             return;
27         }
28         std::scoped_lock const both { mx,
other, mx };
29         // no need to swap cv or mx
30         std::swap(q, other.q);
31     }
32     friend void swap(ThreadsafeQueue &
left,
33                     ThreadsafeQueue &
other){
34         left.swap(other);
35     }
36 private:
37     mutable MUTEX mx { };
38     std::condition_variable notEmpty { };
39     std::queue<T> q { };
40 };
41 }
```

50

© 2023 Peter Sommerlad

Speaker notes

Play with it:

<https://godbolt.org/z/nWhf487vK>

It even supports move-only types:

<https://godbolt.org/z/x816qvG1z>

© 2023 Peter Sommerlad

Using ThreadSafeQueue

```
1 int main() {
2     using namespace std::this_thread;
3     using namespace
std::chrono_literals;
4     TSQ::ThreadSafeQueue<int> queue { };
5     std::thread prod1 { [6] {
6         sleep_for(10ms); // demonstration
7         for(int j=0; j < 10; ++j) {
8             queue.push(j); yield();
9         }
10    } };
11    std::thread prod2 { [6] {
12        sleep_for(9ms); // demonstration
13        for(int i=0; i < 10; ++i) {
14            queue.push(i*11); yield();
15        }
16    } };
17    prod1.join();
18    prod2.join();
19    std::cout << "non-processed
elements:\n";
20    while (!queue.empty()) {
21        std::cout << queue.pop() << '\n';
22    }
23    }
24 }
```

Never ***synchronize*** with `sleep_for()`

51

© 2023 Peter Sommerlad

Speaker notes

Namespace `std::this_thread` provides `yield()` and `sleep_for`

Namespace `std::chrono_literals` the `ms` UDL suffix.

`std::osyncstream` wraps any output stream and guarantees 'atomic' output (non interleaving of current output with other threads). For `std::cout` even without `osyncstream` there are no data races, but potential mixed output from different threads. Other output streams (e.g. `ofstream`) that are shared across threads, such wrapping is required

The sleeping is only here to demonstrate variability in output. Never attempt to use it for solving synchronization issues.

Play with it:

<https://godbolt.org/z/nWhf487vK>

It even supports move-only types:

<https://godbolt.org/z/x816qvG1z>

© 2023 Peter Sommerlad

Synchronisation beyond mutex

Mutual exclusion (`std::mutex`) is insufficient

need to wait for other threads' work

condition variables synchronize

- `wait(lock, condition)` - only when mutex held
- internally unlocks mutex
- when notified relocks mutex and checks condition
- other threads need to notify when they fulfil condition
- when fulfilled returns with mutex locked

52

© 2023 Peter Sommerlad

Speaker notes

Working with `std::condition_variable` requires one to use `std::unique_lock` instead of `std::scoped_lock`, because the condition variable needs a means to unlock and relock the lock.

For locks that do not wrap `std::mutex` one needs to use `std::condition_variable_any`. This might be less performant, because it cannot directly use the corresponding OS synchronisation primitives employed by `std::mutex`.

© 2023 Peter Sommerlad

Single Element Queue

```
struct ThreadSafeExchange {
    void push(T const &t) {
        std::unique_lock lk { mx };
        notFull.wait(lk, [this]() {
            return not q.has_value();
        });
        q.emplace(t);
        notEmpty.notify_one();
    }
    T pop() {
        std::unique_lock lk { mx };
        notEmpty.wait(lk, [this] {
            return q.has_value();
        });
        T t = q.value();
        q.reset();
        notFull.notify_one();
        return t;
    }
};
```

```
// don't call when holding a lock!
bool empty() const {
    std::scoped_lock lk { mx };
    return not q.has_value();
}
private:
    mutable MUTEX mx { };
    std::condition_variable notEmpty { };
    std::condition_variable notFull { };
    std::optional<T> q { };
};
```

need 2 condition variables!

we use `std::optional` for a bounded queue here

<https://godbolt.org/z/KKaq35e3x>

53

© 2023 Peter Sommerlad

Speaker notes

We now need two condition variables, one marking that there is something to consume in the buffer, the other one to mark there is space in the buffer. With a larger buffer value, this means that both conditions can be met (notFull and notEmpty).

<https://godbolt.org/z/KKaq35e3x>

© 2023 Peter Sommerlad

trying - really hard

Full synchronisation can deadlock, when opposite partner is gone

```
struct ThreadSafeExchange {
    bool try_push(T const &t) {
        std::scoped_lock lk { mx };
        if (not q.has_value()){
            q.emplace(t);
            notEmpty.notify_one();
            return true;
        }
        return false;
    }
}

template <typename Rep, typename Period>
bool try_push_for(T const &t, std::chrono::duration<Rep,Period> dur) {
    std::unique_lock lk { mx };
    if (notFull.wait_for(lk, dur, [this]() {
        return not q.has_value();
    })) {
        q.emplace(t);
        notEmpty.notify_one();
        return true;
    } else {
        return false;
    }
}
```

<https://godbolt.org/z/KKaq35e3x>

54

© 2023 Peter Sommerlad

Speaker notes

Be careful to not rely on the empty() member function, because that also acquires the lock and will cause self-deadlock.

<https://godbolt.org/z/KKaq35e3x>

© 2023 Peter Sommerlad

using a starting latch

prevent threads from premature start

```
std::latch startit{3+1};  
// we have 3 threads +main  
// start other threads  
sleep_for(10ms);  
std::cout << "Go go go..."<<  
std::endl;  
startit.arrive_and_wait(); // turn on
```

```
std::jthread prod1 { [&] (std::stop_token stop)  
{  
    startit.arrive_and_wait();  
    //...  
}  
std::jthread prod2 { [&] (std::stop_token stop)  
{  
    startit.arrive_and_wait();  
    //...  
}  
std::jthread cons { [&] (std::stop_token stop)  
{  
    startit.arrive_and_wait();
```

<https://godbolt.org/z/snYa131ef>

*multi-use: cyclic **std::barrier***

<https://godbolt.org/z/4dzeh64Mj>

55

© 2023 Peter Sommerlad

Speaker notes

<https://godbolt.org/z/snYa131ef>

<https://en.cppreference.com/w/cpp/thread/latch>

In addition to the single-pass latch, one can use the cyclic **std::barrier**

<https://en.cppreference.com/w/cpp/thread/barrier>

This allows multiple synchronisation points and can execute a “completion function” that is called when the barrier opens.

© 2023 Peter Sommerlad

Lock-free data structures

Don't try - too error prone

- simple cases work, but still are error prone
- dangers:
 - busy wait
 - life-lock
 - starvation
 - ABA problem (undetectable race condition)

If you still want to do it:

Chapter 7, C++ Concurrency in Action by Anthony Williams

56

© 2023 Peter Sommerlad

Speaker notes

I have seen too many talks about lock-free data structures that contained (non-obvious) bugs.

Concurrency-correctness is hard to test

Most lock-free data structures work only well for simple cases. Some attempts fail after a long time due to counter-overflow.

Scaling might be an illusion, for example, because allocation/release of memory might synchronize as well. If you really try, measure performance!

© 2023 Peter Sommerlad

Architecture for Multicore

Introducing parallelism only with clear architecture

- Understand competing goals:
 - latency
 - throughput
 - utilization
- Minimize need for synchronization
- Know usable architectural patterns
 - and choose wisely

57

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Classic parallelism patterns

- Leader-Followers
- Half-Sync Half-Async
- Pipes and Filters
- Task Farm
- Embarrassing Parallelism

Those are a separate topic

58

© 2023 Peter Sommerlad

Speaker notes

while I could talk about these things, this would go far beyond the scope of the C++ training and also would only be fruitful after the concrete constraints are much better known.

© 2023 Peter Sommerlad

Common Architectural Features

C++ standard library lacks most

- thread pool(s)
- task (unit of work) abstraction
- task continuations
- synchronized queue(s)
- scheduling mechanism

C++26 may provide those, but still very low-level

59

© 2023 Peter Sommerlad

Speaker notes

In addition to mistakes one can make, choosing the wrong mechanism for the problem at hand, can result in less-than-required performance characteristics.

© 2023 Peter Sommerlad

Threading Summary

- programming multi-threaded code that works correctly is hard
 - even world-class experts make mistakes
- interactive debugging of multi-threaded code often hides synchronization problems
 - core dumps from deadlocks are helpful
- do not attempt multi-threading without clear architecture
 - best to employ architectural patterns or frameworks
- using parallel algorithms can be helpful
 - unfortunately not all standard libraries actually implement them parallelized
- C++20 coroutines (will) add another dimension to shoot your foot

60

© 2023 Peter Sommerlad

Speaker notes

Don't confuse coroutines with threading. Coroutines are a means to have an additional surviving local scope, where classic function assumptions about lifetime don't work. Resuming a coroutine from a different thread that suspended it, can have very interesting effects, such as releasing locks that are held by another thread.

© 2023 Peter Sommerlad

Deep-down Concurrency (optional)

62

© 2023 Peter Sommerlad

Speaker notes

Only, because some of you asked.

Better don't do this at home!

© 2023 Peter Sommerlad

Concurrency is even harder!

Compilers and Hardware can reorder memory operations

code says:

```
a = 1  
b = 2  
c = 3  
output c  
output a  
output b
```

executed (e.g.):

```
c = 3  
a = 1  
output c  
b = 2  
output a  
output b
```

63

© 2023 Peter Sommerlad

Speaker notes

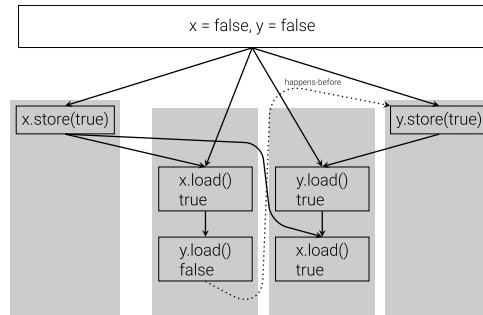
Compilers can reorder writes to memory locations of objects that are independent.

(modern) Hardware reorders loads and stores of memory locations to reduce contention and increase processor pipeline throughput, sometimes very aggressively and on multiple stages. Hardware does not take logical dependencies into account for that!

This is not C++ code and just symbolizes memory writes/stores (by assignment to a “variable”) and memory reads/loads by “output of a variable”

© 2023 Peter Sommerlad

Classic Example: sequential consistent



see <https://godbolt.org/z/zGvfnfhdh>

default memory order sequential consistency

no reordering across atomic access

64

© 2023 Peter Sommerlad

Speaker notes

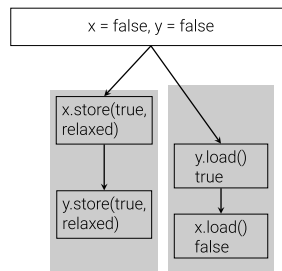
The dotted arrow shows a causal *happens-before* relationship that is guaranteed to occur before the value of `y` is set to `true`. If this happens the demo program counts `z` to be one instead of 2.

see <https://godbolt.org/z/zGvfnfhdh>

Memory order sequential consistency is the easiest to reason about. The compiler must generate code where everything that is written before an atomic access actually *happens before*, and everything that is written and observed after an atomic access actually happens after it. This relationship allows to reorder memory operations as if they happened in a consistent total order (not always the same across runs).

© 2023 Peter Sommerlad

Memory Order: Relaxed



see <https://godbolt.org/z/K8cezyY69>

memory order relaxed

reordering allowed, no visibility guarantee

65

© 2023 Peter Sommerlad

Speaker notes

There is no ordering guarantee, so even when store to y is written after the store of x, there is no *happens-before* relationship between any of the atomic operations. Therefore, reading x can return false, even if the previous read of y returned true.

see <https://godbolt.org/z/K8cezyY69>

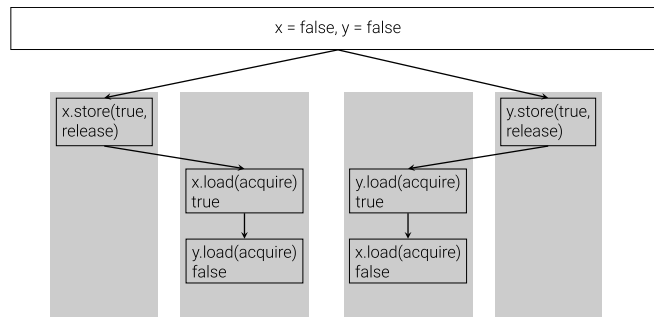
While the individual variable accesses of a relaxed atomic access will never introduce a data race and remain consistent (in contrast to non-atomic variables), there is no guarantee that a value written becomes visible on another thread, or that the order of memory accesses remains as written in code.

This is the fastest (on some processors that support it) but also very hard to reason about and does not actually synchronize. Intel X86 only supports sequential consistent atomics (afaik, at least not acquire-release).

Relaxed atomics can in theory read a value that was never written (so called “out-of-thin-air” OOTA), or can read a value that was created on a speculative branch of execution that was later invalidated (“read-from-untaken-branch” RFUB). For detailed information on the issues and possible scenarios where relaxed atomics are useful, see <https://wg21.link/p2055>.

© 2023 Peter Sommerlad

Acquire-Release



see <https://godbolt.org/z/fa5T8Edae>

memory order acquire and release

limited reordering atomic access

66

© 2023 Peter Sommerlad

Speaker notes

The arrows shows a *happens-before* relationship that is guaranteed, e.g., writing true to x must *happen-before* that value is read by a thread. However, x and y are completely independent atomic variables and thus there is no guarantee that the reads observe a consistent sequences of changes, i.e., both second read operations of the different threads could result in reading the initial value false. There is no sequential consistent reordering in that case.

see <https://godbolt.org/z/fa5T8Edae>

The reordering limitation is as follows (this description can be inaccurate):

- variable writes that are *sequenced-before* in the same thread than a store with *memory-order-release* cannot be ordered after it
- variable reads that are sequenced after in the same thread than a load with *memory-order-acquire* cannot be ordered before it
- neither can happen in the case of *memory-order_acquire-release* (`memory_order::acq_rel`)

Those guarantees can be used to provide ordering guarantees even when *memory-order-relaxed* is used, because a relaxed store cannot be moved after a release store and a relaxed load cannot be reordered before an acquire load.

see <https://godbolt.org/z/EoKxddfqY>

The `memory_order::consume` (like `memory_order::acquire` but with fewer guarantees) was intended to provide better performance than *memory-order-acquire*, which turned out to be impossible to implement. The specification might be corrected for C++26.

© 2023 Peter Sommerlad

Deepest: `std::atomic_flag`

- only atomic type that is always *lock-free*
- classic *test-and-set* operation
- useful with *acquire* and *release*
- can be used to implement *spinlocks*

```
#include <atomic>
class spinlock_mutex
{
    std::atomic_flag flag{ATOMIC_FLAG_INIT};
public:
    void lock() {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock(){
        flag.clear(std::memory_order_release);
    }
};
```

67

© 2023 Peter Sommerlad

Speaker notes

Spinlock example code adapted from [listing 5.1 ccia](#)

In practice many processors also provide other atomic types that are lock free, but the generic `std::atomic<T>` will have to use a mutex around member function calls to guarantee atomicity.

© 2023 Peter Sommerlad

Summary Deep Dive

68

© 2023 Peter Sommerlad

Speaker notes

This chapter was created after I learned that half of the audience already has seen C++ Expert training.

© 2023 Peter Sommerlad

Modernizing existing Code

Write Unit Tests (first)!

*Don't write C, preprocessor only for **#include**(-guards)*

Values over Objects.

Compile-time over Run-time

NO plain arrays or pointers

- Do you have own code to look at?

C++ Introduction C++ Advanced

70

© 2023 Peter Sommerlad

Speaker notes

Most of the modernization aspects are addressed in the other courses, but I summarize briefly here.

<https://github.com/PeterSommerlad/CPPCourseIntroduction>

<https://github.com/PeterSommerlad/CPPCourseAdvanced>

C++ 20 modules will provide even better modularization. However, the specification has some subtle holes (most of them might be fixed for C++23), and not all major compilers implement modules yet (state: end of 2023)

If you want to step deeply into code modernization I consider Michael Feathers' "Working Effectively with Legacy Code" a splendid book.

© 2023 Peter Sommerlad

Code Improvement Terminology

Refactoring: *Improving the Design of Existing Code*

Code Smell: *Indication of need for Refactoring*

Unit Test: *Exercise a piece of code to demonstrate its behavior*

TDD: *Test-driven Design - write unit tests first to learn what needs to be implemented*

71

© 2023 Peter Sommerlad

Speaker notes

While the Books of Martin Fowler with the title “Refactoring” are using Java and Javascript (2nd edition), many of the underlying concepts apply to C++ as well.

One important step during refactoring is to reduce dependencies and create smaller “units” that enable or improve the ability for automated tests.

Starting from tests using TDD (that includes writing unit tests AND refactoring) can lead to much simpler and less coupled code.

© 2023 Peter Sommerlad

C++ Code Smells 🍌

If it stinks, change it! (Kent Beck on code and diapers)

Examples:

- long function
- duplicated code
- Magic Constant
- type casts
- using built-in primitive types often
- using raw pointers in non-library code
- unused code/commented-out code

72

© 2023 Peter Sommerlad

Speaker notes

The term code smell stems from the terminology of Refactoring. My personal background story stems from Kent Beck, who explained it in relation to his baby's diapers.

It indicates a potential problem that a code restructuring/change in design is able to improve.

Not all code smells will easily go away. It requires work.

- long function: longer than 5 lines (my opinion) is a candidate, sometimes even less
- duplication: can stem from copy-paste reuse but also from mimicking existing code examples. A C++-specific aspect is the re-invention of existing standard library algorithm implementations.
- type casts: the type system helps to avoid writing ridiculous code, therefore, circumventing the type system by a cast means there is a design error or it is ridiculous code
- built-in types: for compatibility with the language C, C++ provides built-in types for integers and other numbers, including types `bool` and for characters, that happily convert among each other and that can be combined in expressions, where the compiler will provide implicit conversions (silently), This can lead to surprising results, undetected numeric overflows and undefined behavior
- pointers as provided by the language are an important expert-level building block, but their proliferation in non-library code is no longer considered reasonable in modern C++ (C requires pointers for many more things, where C++ has better alternatives)

© 2023 Peter Sommerlad

Long Function (aka Long Method) 💩

How long is a Long Function?

Indicators:

- comments within the function
 - often indicate pieces to extract
- multiple non-nested loops
- (deeply) nested if-statements

*Remedy: **Extract function refactoring***

73

© 2023 Peter Sommerlad

Speaker notes

A good function does one thing well and is named like that

Whenever you feel the need to write a comment within a function body, it is an indicator that another separate function needs to be extracted, named accordingly to what it does and called instead of inserting its code piece.

For existing code, inline comments in a function are often good indicators for sections to extract to a new function.

I consider functions longer than 5 lines a candidate, sometimes even just more than two lines is too much.

Exceptions can exist, but should not be common.

Finding good abstractions is an art/experience and often requires experimentation/practice first.

TDD can help a lot to find smaller functions that more easily combine and are simpler to test comprehensively.

© 2023 Peter Sommerlad

Magic Constants 🍌

42, 4, 8 : what does it mean?

- using numeric literals without giving them a name is problematic
- same value might be needed in multiple places
- same value can mean different things

```
mutable std::array<PrUTHitNews,8> m_hitsLayers;  
mutable std::array<PrUTHitNews,4> m_allHits;  
mutable std::array<float,4> m_normFact;  
mutable std::array<float,4> m_invNormFact;  
mutable std::array<float,4> m_bestParams;  
//...  
for( int i = 0; i<8 ; ++i){  
//...
```

74

© 2023 Peter Sommerlad

Speaker notes

From: [https://gitlab.cern.ch/lhcb/Rec/-](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.h#L112)

[/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.h#L112](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.h#L112)

Do all 4s mean the same? What about the 8?

[https://gitlab.cern.ch/lhcb/Rec/-](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L364)

[/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L364](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L364)

```
for( int i = 0 ; i<4; ++i){  
    info()<<"m_allHits["<<i<<" ] size "<<m_allHits[i].size()<<endmsg;  
}
```

[https://gitlab.cern.ch/lhcb/Rec/-](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L592)

[/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L592](https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L592)

The following code repeats the 8 of the array's dimension, what if it is changed?

```
for( int i = 0; i<8 ; ++i){  
    info()<<"NHits m_hitsLayers["<<i<<" ] = "<<m_hitsLayers.size() <<endmsg;  
    for( auto & hit : m_hitsLayers[i]){  
        printHit( hit );  
    }  
}
```

© 2023 Peter Sommerlad

Magic constants(2)

Configuration parameters in code

- hard to change
- can create incompatible executables
- can indicate lack of abstraction

```
declareProperty("MaxXSlope"           , m_maxXSlope      =  
0.350);  
declareProperty("MaxYSlope"           , m_maxYSlope      =  
0.300);  
declareProperty("centralHoleSize"     , m_centralHoleSize = 33. *  
Gaudi::Units::mm);
```

see *PrVeloUTTool.cpp*

75

© 2023 Peter Sommerlad

Speaker notes

<https://gitlab.cern.ch/lhcb/Rec/-/blob/473a2bef5e272d825dea3fc6574f189cf52b21b0/Pr/PrVeloUT/src/PrVeloUTTool.cpp#L44>

```
// Momentum determination  
declareProperty("minMomentum" , m_minMomentum = 0 * Gaudi::Units::GeV);  
declareProperty("minPT"       , m_minPT           = 0.1 * Gaudi::Units::GeV);  
declareProperty("maxPseudoChi2", m_maxPseudoChi2 = 1280.);  
// Tolerances for extrapolation  
declareProperty("MaxXSlope"    , m_maxXSlope      = 0.350);  
declareProperty("MaxYSlope"    , m_maxYSlope      = 0.300);  
declareProperty("centralHoleSize", m_centralHoleSize = 33. * Gaudi::Units::mm);  
declareProperty("YTolerance"   , m_yTol          = 0.8 * Gaudi::Units::mm);  
declareProperty("YTolSlope"    , m_yTolSlope     = 0.2);  
// Grouping tolerance  
declareProperty("HitTol1"      , m_hitTol1      = 6.0 * Gaudi::Units::mm);  
declareProperty("HitTol2"      , m_hitTol2      = 0.8 * Gaudi::Units::mm);  
declareProperty("DeltaTx1"     , m_deltaTx1   = 0.035);  
declareProperty("DeltaTx2"     , m_deltaTx2   = 0.02);  
declareProperty("IntraLayerDist", m_intraLayerDist = 15.0 * Gaudi::Units::mm);  
declareProperty("PrintVariables", m_printVariables = false);  
declareProperty("PassTracks"   , m_passTracks   = false);  
declareProperty("PassHoleSize" , m_passHoleSize = 40. * Gaudi::Units::mm);  
declareProperty("OverlapTol"   , m_overlapTol  = 0.7 * Gaudi::Units::mm);  
declareProperty("MinHighThreshold", m_minHighThres = 1);
```

© 2023 Peter Sommerlad

DIY algorithmic loop

C++ provides a rich set of generic algorithmic functions

*Writing your own loop indicates a **primitive obsession** and **code duplication***

- `<algorithm>`
- `<numeric>`
- `<iterator>`
- `<ranges>` - C++20

Treat every legacy loop as a candidate for replacement by an algorithm

76

© 2023 Peter Sommerlad

Speaker notes

Take the exercise “8 d) [algorithm trivia](#)” of my C++ introduction course to familiarize yourself with the available algorithms.

© 2023 Peter Sommerlad

Example: algorithmic loop

```
// stupid O(N^2) unique-ification..
for ( auto i = begin( m_flatUniqueAlgList ); i != end( m_flatUniqueAlgList ); ++i ) {
    auto n = next( i );
    while ( n != end( m_flatUniqueAlgList ) ) {
        if ( *n == *i )
            n = m_flatUniqueAlgList.erase( n );
        else
            ++n;
    }
}
```

AlgResourcePool.cpp

- Exercise: take a component of CERN code and look if you can find code smells and potential for refactoring, show and discuss!

77

© 2023 Peter Sommerlad

Speaker notes

<https://gitlab.cern.ch/gaudi/Gaudi/-/blob/master/GaudiHive/src/AlgResourcePool.cpp>

uses mutex without `scoped_lock/lock_guard/unique_lock` -> not exception safe!

has a very long functions

manual bad implementation of standard algorithm

A possible simpler solution using standard algorithm:

```
// stupid O(N^2) unique-ification..
for ( auto i = begin( m_flatUniqueAlgList ); i != end( m_flatUniqueAlgList ); ++i ) {
    m_flatUniqueAlgList.erase(remove(next(i),end( m_flatUniqueAlgList ),*i));
}
```

Note, the above loop works, because the `.erase()` member function will not invalidate iterator `i`, but the `end()` call in the outer loop must not be cached, the end of the container changes. But better extract the above loop first from the very long function and write unit tests for it, before actually replacing it.

One must also ask, why could there be unwanted duplicates in the container anyway.

if sorted the following would work:

```
container.erase(unique(begin(container),end(container)),end(container))
```

© 2023 Peter Sommerlad

No Pointers

- References (as parameter types)
 - for side effects, even on member functions
 - **const&** only for managers and “big” types
- return by value
 - empty **optional<T>** or exceptions to mark errors
- manage memory with **vector**, **string**, containers
 - **unique_ptr** only when other things insufficient

78




© 2023 Peter Sommerlad

Speaker notes

If you really need optional references, consider using a non-standard optional implementation (**boost::optional** or **tl::optional**) or **std::optional<std::reference_wrapper<T>>**. The latter approach is not very efficient, but may be C++26 will allow **optional<T&>**, I am working on it: [P2988](#).

© 2023 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	<code>unique_ptr<T> const</code>	must be init with <code>make_unique<T></code>
	nullable 	value	<code>optional<T></code>	to denote missing value best for return values
		heap	<code>unique_ptr<T> const</code>	T can be base class with <code>make_unique<Derived></code>
referring T 	non-null	fixed	T &	can dangle
		rebind	<code>reference_wrapper<T></code>	assignability with a reference member
	nullable 	fixed	<code>jss::object_ptr<T> const</code> <code>boost::optional<T&> const</code>	missing in std <code>std::optional</code> can not do this <code>boost::optional</code> can <code>object_ptr<T></code> by A. Williams
		rebind	<code>jss::object_ptr<T></code> <code>boost::optional<T&></code> <code>optional<reference_wrapper<T>></code>	

79

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Pointers as Array replacement

	Old/Unsafe	Modern/Better	Alternative
with explicit bound	<pre> void absarray(int a[], size_t len) { for(size_t i=0; i < len; ++i) a[i] = a[i] < 0? -a[i]:a[i]; } </pre>	<pre> template<size_t N> void absarray(int (&a)[N]) { for(size_t i=0; i < N; ++i) a[i] = a[i] < 0? -a[i]:a[i]; } </pre>	<pre> // C++ 20 or GSL void absarrayspan(std::span<int> a){ for(size_t i=0; i < a.size(); ++i) a[i] = a[i] < 0? -a[i] : a[i]; } </pre>
implicit sentinel nul/nullptr	<pre> void takecharptr(char *s){ for (; *s; ++s) *s = std::toupper(*s); } </pre>	<pre> void takestring(std::string &s){ transform(s.begin(),s.end(), s.begin(), [](char c){ return std::toupper(c);}); } </pre>	<pre> void take(std::string_view s){ // can not change } </pre>
explicit range	<pre> void absintprange(int *b, int *e){ for (;b != e; ++b){ if(*b < 0) *b = - *b; } } </pre>	<pre> void absintprange(int *b, int *e) { std::transform(b,e,b, [](auto i){ return std::abs(i);}); } </pre>	<pre> template <typename FWDITER> void absintarray(FWDITER b, FWDITER e) { std::transform(b,e,b, [](auto i){ return std::abs(i);}); } </pre>

80

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Alternatives to plain arrays

`std::vector<T>`

`std::array<T, N>`

`std::string`

`string_view` 💣

`span<T>` 💣

81

© 2023 Peter Sommerlad

Speaker notes

`std::string_view` and `std::span` are **relation types**, so they might dangle. Those types can only safely be used as parameter types to pass a range in a lightweight way (no copying of the range). However, returning them from a function or having a local variable of these types is almost always an error, because it is too easy to make the mistake to use the range view, when the underlying container is already gone. Even if it works today, a slight refactoring of the code can break it and cause undefined behavior.

© 2023 Peter Sommerlad

back from Pointer Replacement

83

© 2023 Peter Sommerlad

Speaker notes

top page for navigation purposes only

© 2023 Peter Sommerlad

Side-step `virtual`

Unbounded dynamic polymorphism only when needed

- static polymorphism: overloading and templates
- `std::variant`
- type erasure

do not overdo it!

84

© 2023 Peter Sommerlad

Speaker notes

if existing code uses `virtual` and inheritance and works well that is OK

© 2023 Peter Sommerlad

Compile-time over run-time

- mark function templates with **constexpr** (or **constexpr**)
- templates over class hierarchies
- **static_assert()** over **assert()**
- “test” your code at compile time

85

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

Warnings and Static Analysis

- compile with `-Wall -pedantic -Werror -pedantic-error`
 - `-Wextra` might give too many false positives
 - select further warning candidates
- Run (unit) tests with every build
 - employ `-fsanitize=...` for tests
- look at IDE features for code improvement
 - my past self had many implemented for Cevelop
 - fine tune against false positives
- employ (commercial) static analysis tools
 - look for upcoming new MISRA-C++ guidelines
 - [C++ Vulnerabilities](#)

86

© 2023 Peter Sommerlad

Speaker notes

Don't run static analysis tools as an afterthought. The number of messages can be overwhelming. Each tool might require fine tuning wrt potential false positives. Not every check might be appropriate in your code base, but most violation should have a good reason that goes beyond: "that's just how we implemented it".

<https://iso-iec-jtc1-sc22-wg23-cpp.github.io/wg23-tr24772-10-public/>

© 2023 Peter Sommerlad

Beware of built-in/primitive types

Code smell: **Primitive Obsession**

- Lack of appropriate domain types leads to hard to grasp code.
- Support of the type system to prevent ridiculous code is undermined by using built-in types directly for domain quantities.
- `std::string` etc are also “primitive”

87

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad

C++ problems with built-ins

C++ has specific dark corners in its type system inherited from C:

- Integral Promotion
 - shorter (unsigned) types promote to `int`
- Usual Arithmetic Conversions
 - overflow
 - silent narrowing, widening, sign change happens
- Implicit Conversions
 - might call the wrong function

Guidelines:

- Use `char` only for text characters
- Use substitute integer types
- Use strong types for domain types

88

© 2023 Peter Sommerlad

Speaker notes

One of the darkest corners of the C++ type system are the built-in types inherited from C. One aspect, for example, is that `bool` as well as all types representing characters (in different widths) are treated as integral types and implicitly convert to an integer when used in arithmetic expressions.

Special danger is rooted in the Undefined Behavior that stems from signed integer overflow. Another feature of silent wrapping in unsigned integer arithmetic can keep incorrect results undetected.

While we need to rely on built-in types, they should be encapsulated, or only used, when the domain of values clearly is sufficient.

© 2023 Peter Sommerlad

enum for wrapping integers

signed integer overflow is undefined behavior

integral promotion is a curse

- enum class types
- operator overloading
- concepts

allow to implement wrapping, non-promoting integers:

[Simple Safe Integers pssafeint.h](#)

89

© 2023 Peter Sommerlad

using enums as integers

```
// unsigned
enum class ui8: std::uint8_t{ tag_to_prevent_mixing_other_enums };
enum class ui16: std::uint16_t{ tag_to_prevent_mixing_other_enums };
enum class ui32: std::uint32_t{ tag_to_prevent_mixing_other_enums };
enum class ui64: std::uint64_t{ tag_to_prevent_mixing_other_enums };
// signed
enum class si8: std::int8_t{ tag_to_prevent_mixing_other_enums };
enum class si16: std::int16_t{ tag_to_prevent_mixing_other_enums };
enum class si32: std::int32_t{ tag_to_prevent_mixing_other_enums };
enum class si64: std::int64_t{ tag_to_prevent_mixing_other_enums };
```

90

© 2023 Peter Sommerlad

User-defined Literals (UDL)

```
inline namespace literals {
constexpr
ui16 operator""_ui16(unsigned long long val) {
    if (val <=
std::numeric_limits<std::underlying_type_t<ui16>>::max()) {
        return ui16(val);
    } else {
        throw "integral constant too large"; // trigger compile-
time error
    }
}
// etc...
```

91

© 2023 Peter Sommerlad

Using UDLs

```
using namespace pssint::literals; // required for UDLs

void ui16intExists() {
    using pssint::ui16;
    auto large=0xff00_ui16;
    //0x10000_ui16; // compile error
    //ui16{0xffffffff}; // narrowing detection
    ASSERT_EQUAL(ui16{0xff00u},large);
}
```

92

© 2023 Peter Sommerlad

Traits and Concepts

```
template<typename T>
using plain = std::remove_cvref_t<T>;
template<typename T>
concept an_enum = std::is_enum_v<plain<T>>;
// from C++23
template<an_enum T>
constexpr bool
is_scoped_enum_v = !std::is_convertible_v<T, std::underlying_type_t<T>;
template<typename T>
concept a_scoped_enum = is_scoped_enum_v<T>;
```

93

© 2023 Peter Sommerlad

Detection Idiom with Concept

```
template<typename T>
constexpr bool
is_safeint_v = false;
template<a_scoped_enum E>
constexpr bool
is_safeint_v<E> = requires {
    E{} == E::tag_to_prevent_mixing_other_enums;
} ;
template<typename E>
concept a_safeint = is_safeint_v<E>;
```

94

© 2023 Peter Sommerlad

Testing Detection Idiom

```
namespace _testing {
using namespace pssint;
static_assert(is_safeint_v<ui8>);
static_assert(is_safeint_v<ui16>);
static_assert(is_safeint_v<ui32>);
static_assert(is_safeint_v<ui64>);
static_assert(is_safeint_v<si8>);
static_assert(is_safeint_v<si16>);
static_assert(is_safeint_v<si32>);
static_assert(is_safeint_v<si64>);
enum class enum4test{};
static_assert(!is_safeint_v<enum4test>);
static_assert(!is_safeint_v<std::byte>);
}
```

95

© 2023 Peter Sommerlad

Meta Programming for Promotion

```
template<typename E>
using ULT=std::conditional_t<std::is_enum_v<plain<E>>,
    std::underlying_type_t<plain<E>>, plain<E>>;
template<typename E>
using promoted_t = // will promote keeping signedness
    std::conditional_t<
        (std::is_unsigned_v<ULT<E>> && (sizeof(ULT<E>) < sizeof(unsigned))),
        unsigned, ULT<E>>;
template<a_safeint E>
constexpr auto
to_int(E val) noexcept { // promote keeping signedness
    return static_cast<promoted_t<E>>(val);
}
```

96

© 2023 Peter Sommerlad

Testing Same-sign Promotion

```
static_assert(std::is_same_v<unsigned,decltype(to_int(1_ui8)+1)>>);
static_assert(std::is_same_v<unsigned,decltype(to_int(2_ui16)+1)>>);
static_assert(std::is_same_v<int8_t,decltype(to_int(1_si8))>>);
static_assert(std::is_same_v<int16_t,decltype(to_int(2_si16))>>);
```

97

© 2023 Peter Sommerlad

Concept for limiting integral

```
template<std::integral T>
constexpr bool
is_integer_v = std::is_same_v<uint8_t, T> ||
               std::is_same_v<uint16_t, T> ||
               std::is_same_v<uint32_t, T> ||
               std::is_same_v<uint64_t, T> ||
               std::is_same_v<int8_t, T> ||
               std::is_same_v<int16_t, T> ||
               std::is_same_v<int32_t, T> ||
               std::is_same_v<int64_t, T>;

template<typename T>
concept an_integer = is_integer_v<T>;
```

98

© 2023 Peter Sommerlad

Meta Programming for Conversion

```
template<an_integer T>
constexpr auto
from_int(T val) {
    using std::is_same_v;
    using std::conditional_t;
    struct cannot_convert_integer{};
    using result_t =
        conditional_t<is_same_v<uint8_t,T>, ui8,
        conditional_t<is_same_v<uint16_t,T>, ui16,
        conditional_t<is_same_v<uint32_t,T>, ui32,
        conditional_t<is_same_v<uint64_t,T>, ui64,
        conditional_t<is_same_v<int8_t,T>, si8,
        conditional_t<is_same_v<int16_t,T>, si16,
        conditional_t<is_same_v<int32_t,T>, si32,
        conditional_t<is_same_v<int64_t,T>, si64, cannot_convert_integer>>>>
    return static_cast<result_t>(val);
}
```

99

© 2023 Peter Sommerlad

Directed Conversion

```
template<a_safeint T0, an_integer FROM>
constexpr T0
from_int_to(FROM val) {
    using std::is_same_v;
    using std::conditional_t;
    using result_t = T0;
    if constexpr(std::is_unsigned_v<std::underlying_type_t<result_t>>){
        if (val <= std::numeric_limits<std::underlying_type_t<result_t>>::max()) {
            return static_cast<result_t>(val);
        } else {
            throw "integral constant too large";
        }
    } else {
        if (val <= std::numeric_limits<std::underlying_type_t<result_t>>::max() &&
            val >= std::numeric_limits<std::underlying_type_t<result_t>>::min()) {
            return static_cast<result_t>(val);
        } else {
            throw "integral constant out of range";
        }
    }
}
```

100

© 2023 Peter Sommerlad

Testing from_int Conversion

```
static_assert(1_ui8 == from_int(uint8_t(1)));
static_assert(42_si8 == from_int_to<si8>(42));
//static_assert(32_ui8 == from_int(' ')); // does not compile
//static_assert(1_ui8 == from_int_to<ui8>(true)); // does not compile

void checkedFromInt(){
    using namespace pssint;
    ASSERT_THROWS(from_int_to<ui8>(2400u), char const *);
}
```

101

© 2023 Peter Sommerlad

Output Operator

```
template<a_safeint E>
std::ostream& operator<<(std::ostream &out, E value){
    out << +to_int(value); // + triggers promotion and prevents
char
    return out;
}
```

concept a_safeint prevents using it for other types

```
std::ostream& operator<<(std::ostream &out, a_safeint auto value){
    out << +to_int(value); // + triggers promotion and prevents
char
    return out;
}
```

I prefer `template<a_concept T>` over `(a_concept auto`

102

© 2023 Peter Sommerlad

Arithmetic Operators

```
template<a_safeint E, a_safeint F>
constexpr auto
operator+(E l, F r) noexcept
requires same_sign<E,F> {
    using result_t=std::conditional_t<sizeof(E)>=sizeof(F),E,F>;
    return static_cast<result_t>(<
        static_cast<ULT<result_t>>(<
            to_uint(l)
            + // use unsigned op to prevent signed overflow, but wrap
            to_uint(r)
        ) );
}
```

103

© 2023 Peter Sommerlad

Wrapping Simple Safe Integers

- Free to use and download
- requires C++20
 - a C++17 backport is available
- Target audience:
 - embedded developers
 - safety critical systems developers
- best if regular integer types are prevented by static analysis

<https://github.com/PeterSommerlad/PSsimplsafeint>

104

© 2023 Peter Sommerlad

More safe Integers

I am creating more alternatives:

105

© 2023 Peter Sommerlad

Speaker notes

At the moment (Nov 2023) PSsODIN is still very fresh and PSsSATIN is not yet ready (there is a much better implementation possible).

In contrast to other safe integer libraries, these deliberately prevent mixing signed and unsigned numbers and implicit conversions, especially narrowing.

- <https://github.com/PeterSommerlad/PSsODIN>
- <https://github.com/PeterSommerlad/PSsODIN>

© 2023 Peter Sommerlad

Strong Types over built-ins

- Beware of implicit conversions and integral promotion
- Model your system with strong types

C++ Advanced Strong Types

- Consider a units library for physical dimensions

106

© 2023 Peter Sommerlad

Speaker notes

This single day course is too short to give a full introduction to all things that are relevant. But you can find some more information in the my C++ Advanced Training, where also some.

© 2023 Peter Sommerlad

Take Aways 🍟🍔

- Simplify existing code!
 - especially before changing it
- Create unit tests
 - for bugs to fix
 - for extracted functions
 - for new stuff before implementing it
- Refactor Mercilessly!
 - version control allows you to go back

107

© 2023 Peter Sommerlad

Speaker notes

For further reading: Michael Feathers - Working effectively with Legacy Code

© 2023 Peter Sommerlad

Done...

Feel free to contact me @PeterSommerlad or
peter.cpp@sommerlad.ch in case of further questions

108

© 2023 Peter Sommerlad

Speaker notes
for your own notes

© 2023 Peter Sommerlad