Pessimistic Programming

Patrice Roy

Patrice.Roy@USherbrooke.ca

CeFTI, Université de Sherbrooke

Patrice.Roy@clg.qc.ca

Collège Lionel-Groulx

Who am I?

- Father of five (four girls, one boy), ages 23 to 5
- Feeds and cleans up after a varying number of animals
  - Look for Paws of Britannia with your favorite search engine
- Used to write military flight simulator code, among other things
  - CAE Electronics Ltd
- Full-time teacher since1998
  - Collège Lionel-Groulx, Université de Sherbrooke
  - Works a lot with game programmers
- Incidentally, WG21 and WG23 member (although I've been really busy recently)
  - Involved in SG14, among other study groups
  - Occasional WG21 secretary
- And so on…

Before we start

- It's often a good idea to test one's theories and approaches on more than one compiler
  - http://coliru.stacked-crooked.com/
  - http://en.cppreference.com/w/
    - Edit the examples and test them directly!
  - http://ideone.com/
  - https://wandbox.org/
  - https://tbfleming.github.io/cib/
    - A Clang online, generating WebAssembly!
  - …and many more
    - See http://h-deb.clg.qc.ca/Liens/Essayer-langages.html

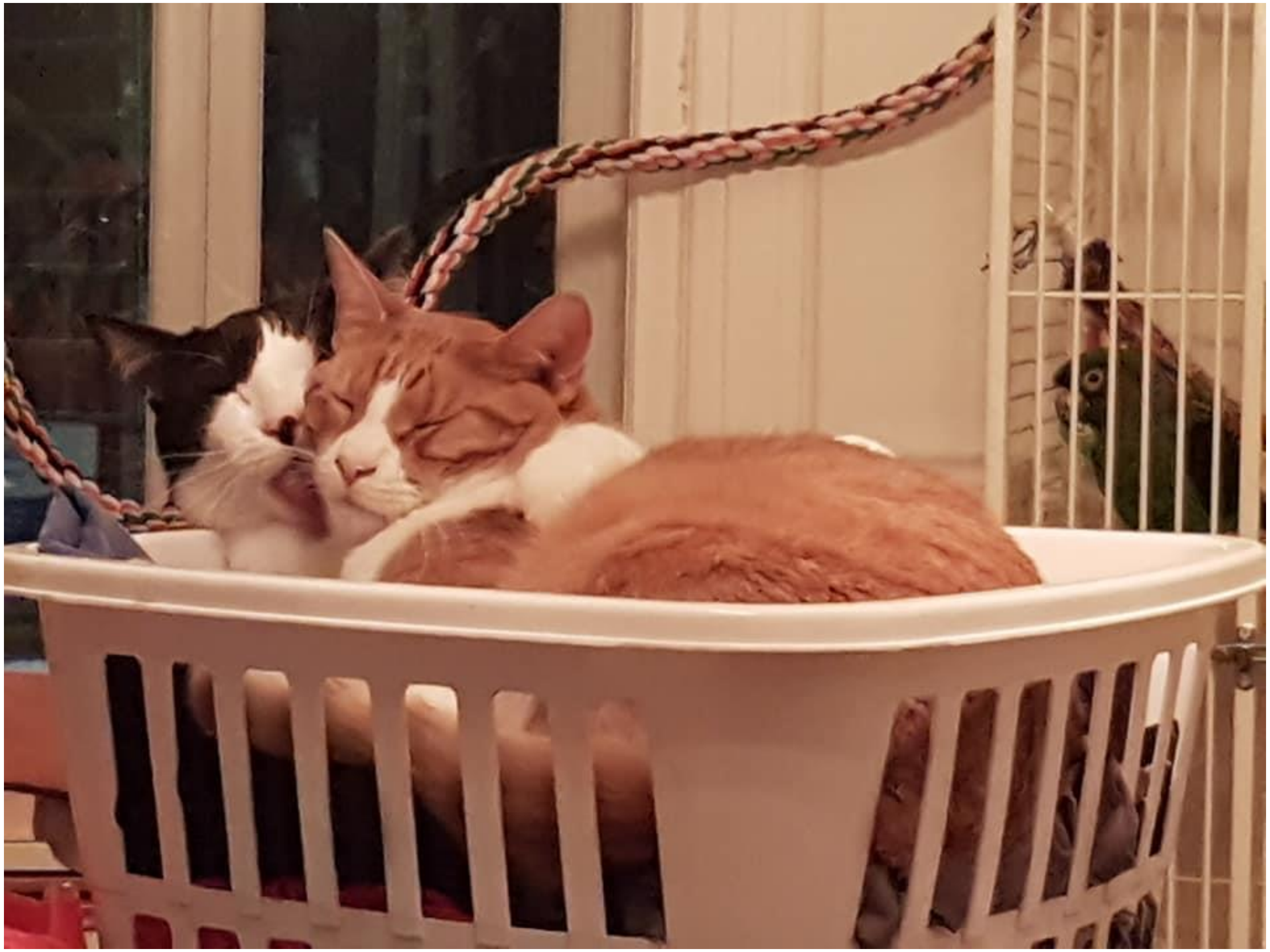
Before we start

- Compilers are only part of the equation
  - http://en.cppreference.com/w/
    - … also an amazing online help!
  - https://gcc.godbolt.org/
    - See what your compiler generates based on your sources
  - http://eel.is/c++draft/
    - The actual text of the C++ standard!
  - http://quick-bench.com/
    - For quick comparative benchmarks

Before we start

- Compilers are only part of the equation (bis.)
  - http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
    - Some recommended programming practices, meant to be instrumentable through tooling
  - https://taas.trust-in-soft.com/tsnippet/#
    - Detecting undefined behavior
  - https://cppinsights.io/
    - How your compiler rewrites your sources
  - …and many more
    - See http://h-deb.clg.qc.ca/Liens/Essayer-langages.html

Why this talk?



Why this talk?

- We often write programs in such a way as to make them run fast. We want good average speed, high throughput, and we tend to be happy when benchmarks show that our peak running speed is better than expected.

Why this talk?

- We often write programs in such a way as to make them run fast. We want good average speed, high throughput, and we tend to be happy when benchmarks show that our peak running speed is better than expected.
- However, it's sometimes useful to write programs where we want to make the worst-case scenario faster, or make it run at predictable speed, or even reduce variations in execution speed. Instead of concentrating our efforts on making the best or the average speed better, we sometimes need to make the worst case speed "less bad".

Why this talk?

- C++ is a wonderful language for such situations. C++ gives us a lot of control over what's going on, and we can use this control to our advantage.

Why this talk?

- This talk aims to:
  - Discuss techniques to make the execution speed of programs more predictable

Why this talk?

- This talk aims to:
  - Discuss techniques to make the execution speed of programs more predictable
  - Guide the compiler towards generating code where worst-case execution speed respects some constraints

Why this talk?

- This talk aims to:
  - Discuss techniques to make the execution speed of programs more predictable
  - Guide the compiler towards generating code where worst-case execution speed respects some constraints
- Target audience:
  - Intermediate C++ programmers

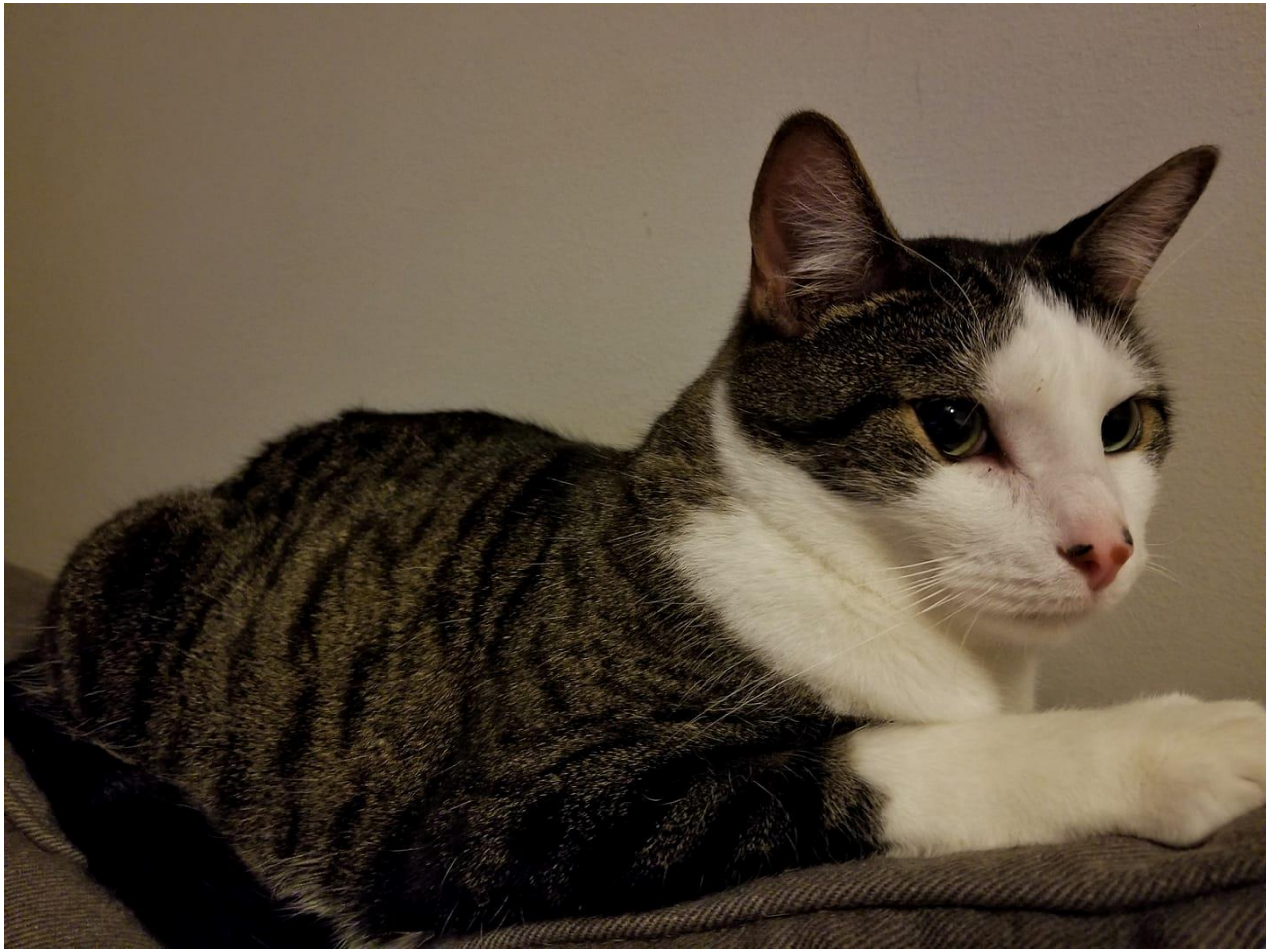
Why this talk?

- This talk aims to:
  - Discuss techniques to make the execution speed of programs more predictable
  - Guide the compiler towards generating code where worst-case execution speed respects some constraints
- Target audience:
  - Intermediate C++ programmers
  - Curious about how to address such issues, or

Why this talk?

- This talk aims to:
  - Discuss techniques to make the execution speed of programs more predictable
  - Guide the compiler towards generating code where worst-case execution speed respects some constraints
- Target audience:
  - Intermediate C++ programmers
  - Curious about how to address such issues, or
  - Wonder why it is sometimes important to be pessimistic and worry about those times when program execution takes the slow path

Why be pessimistic?



Why be pessimistic?

- All code for the examples that follow can be viewed at:
  - **https://tinyurl.com/y852gjr6**
- Note that most of the examples in the slides that follow are incomplete, although they should (hopefully) be understandable
  - The code on the shared drive is written to explain concepts (but often does nothing useful). It does compile, however

Why be pessimistic?

- Not all programmers need to worry about worst-case behavior
  - Most programmers are probably more interested in average speed or total execution time for their processes

Why be pessimistic?

- Not all programmers need to worry about worst-case behavior
  - Most programmers are probably more interested in average speed or total execution time for their processes
  - Programming in order to respect low-latency constraints (the SG14 people) is one situation where worst-case tends to be a significant source of worry

Why be pessimistic?

- Not all programmers need to worry about worst-case behavior
  - Most programmers are probably more interested in average speed or total execution time for their processes
  - Programming in order to respect low-latency constraints (the SG14 people) is one situation where worst-case tends to be a significant source of worry
  - Programs that have to be responsive, have predictable behavior at all times, iterate at a stable frequency, etc. are all use-cases for pessimistic programming

Why be pessimistic?

- Expressed otherwise:
  - One can have a 70 frames per second display, but with frames displayed at an irregular pace
    - It can be unpleasant to the eye, and seem to jitter
  - One can have a 50 frames per second display, but with images displayed every $\frac{1}{50}$ second
    - It can feel very smooth
    - The trick is to use the remaining time (if any) of every iteration in an intelligent and productive way

Why be pessimistic?

- One could say we're all pessimistic on occasion
- Take for example the case of exceptions

Why be pessimistic?

- One could say we're all pessimistic on occasion
- Take for example the case of exceptions
  - https://wandbox.org/permlink/48ZQhDSRlZssK6uK
- Output (actual numbers can vary):
  - With exceptions, nice data : (1000000,0) in 1553μs.
  - With exceptions, nasty data : (0,1000000) in 3677976μs.
  - With optional, nice data : (1000000,0) in 2031μs.
  - With optional, nasty data : (0,1000000) in 3696μs.

Why be pessimistic?

- Exceptions are actually faster in this case when nothing goes wrong
  - They're quite painful in the worst case (and this sample code is quite bad in that respect)

Why be pessimistic?

- If you care about throughput or a fast average case, you'll want to measure how frequent (or ideally, how rare) the worst case is

Why be pessimistic?

- If you care about throughput or a fast average case, you'll want to measure how frequent (or ideally, how rare) the worst case is
- But what if, when an error occurs, you have to react *fast*?

Why be pessimistic?

- One could say we're all pessimistic on occasion
- Take for example the case of containers

Why be pessimistic?

- One could say we're all pessimistic on occasion
- Take for example the case of containers
  - https://wandbox.org/permlink/Ka0Gr9hSBcsmXlwY
- The naïve, default option of `vector` underperforms here (as should be expected), even with `reserve()`
- Output (actual numbers can vary):
  - With vector : 537187µs.
  - With deque : 774µs.

Why be pessimistic?

- One could say we're all pessimistic on occasion
- Take for example the case of containers
  - `vector-vs-deque-2.cpp`
- Of course, with only insertions at the end, it's a totally different situation
- Output obtained with MSVC (my numbers with Wandbox were suspicious)
  - With vector : 1740µs.
  - With deque : 13453µs.

```
/GS /GL /analyze- /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Fd"Release\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /Gd /Oy- /Oi /MD /std:c++latest /FC /Fa"Release\" /EHsc /nologo /Fo"Release\" /Fp"Release\z17.3.pch" /diagnostics:classic
```

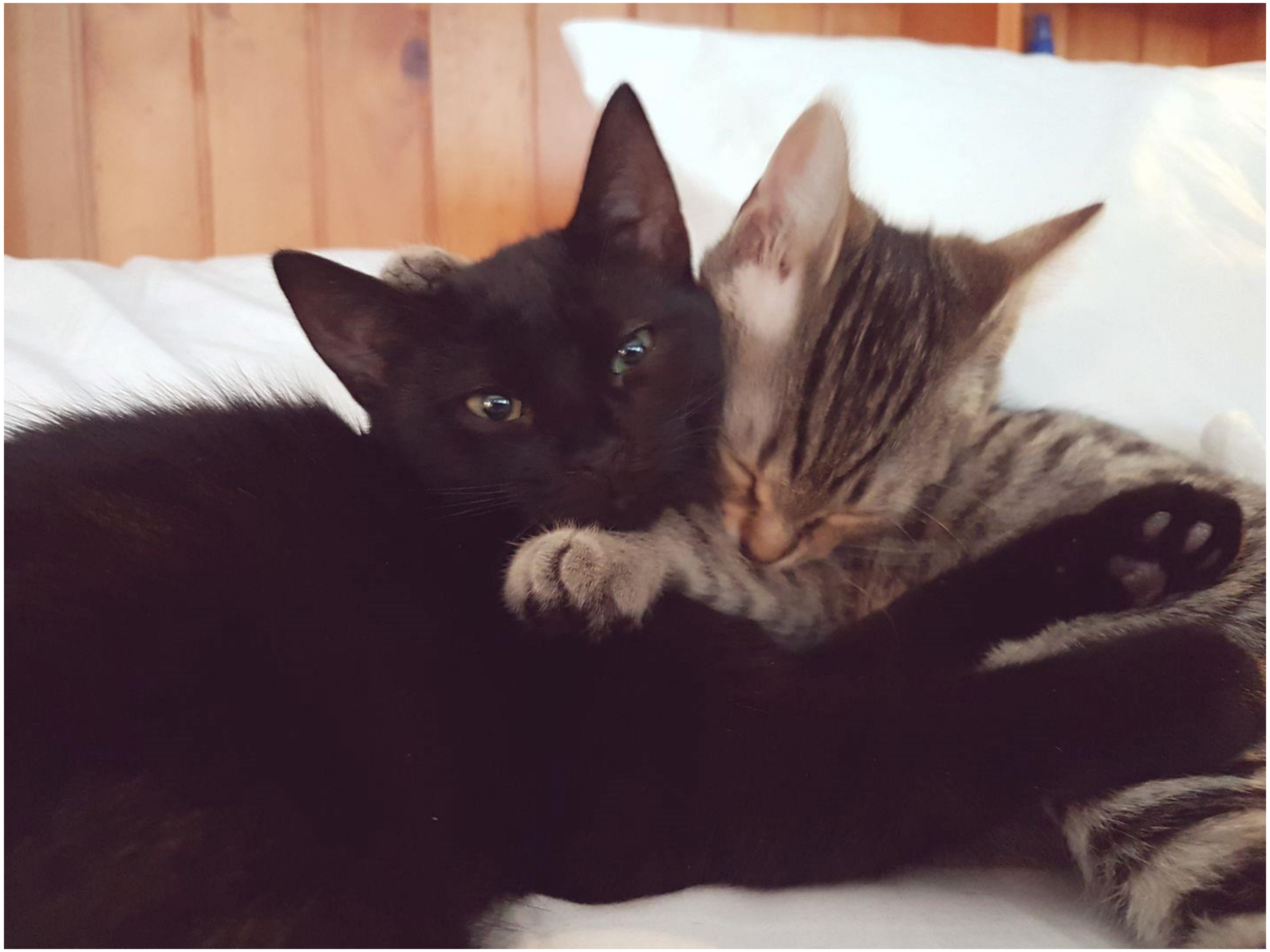
Why be pessimistic?

- So, which one should we pick: `vector` or `deque`?
  - Well, again, if inserts at the beginning are rare enough, use `vector`… on average

Why be pessimistic?

- So, which one should we pick: `vector` or `deque`?
  - Well, again, if inserts at the beginning are rare enough, use `vector`… on average
  - But what if you're concerned about the worst-case insertion time? Maybe `deque` is always *fast enough*, whereas `vector` is *typically* fast enough, and faster on average, but might *sometimes* be too slow
    - Notice the difference between *fast enough* and (on average) *fastest*

When worst-case counts



When worst-case counts

- Now, these examples seem a bit abstract, disconnected from real-word usage
  - Let's tweak them a bit
  - `driving.cpp`

When worst-case counts

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   } catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

When worst-case counts

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   } catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

Supposing a good « driver » and low collision risks, this might seem like a good implementation

When worst-case counts

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   } catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

… but what if time is of the essence when a potential collision is detected?

When average deviation counts

When average deviation counts

- Sometimes, what is required is “constant” throughput more than fastest possible throughput
  - By “constant” here, I mean average deviation that converges toward zero

When average deviation counts

```
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
    // expected to fail only in
    // extreme cases
    for(Data data; source >> data;) {
        process(data);
    }
    // expected to be extremely rare
    throw ConsumeError{};
}
```

When average deviation counts

```
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   // this construct will consume
   // data immediately if it's
   // already available
   for(Data data; source >> data;) {
      process(data);
   }
   throw ConsumeError{};
}
```

When average deviation counts

```
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
    // this construct will consume data
    // immediately if it's already available
    //
    // in that sense, it can be said to be
    // efficient
    for(Data data; source >> data;) {
        process(data);
    }
    throw ConsumeError{};
}
```

When average deviation counts

```
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   // this construct will consume data immediately
   // if it's already available
   //
   // in that sense, it can be said to be efficient
   //
   // of course, if there's no data available, it will
   // be blocked on an I/O read
   for(Data data; source >> data;) {
      process(data);
   }
   // expected to be extremely rare
   throw ConsumeError{};
}
```

When average deviation counts

```
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   // this construct will consume data immediately
   // if it's already available
   //
   // in that sense, it can be said to be efficient
   //
   // of course, if there's no data available, it will
   // be blocked on an I/O read
   for(Data data; source >> data;) {
      process(data);
   }
   // expected to be extremely rare
   throw ConsumeError{};
}
```

Such a construct can process data that arrives at a constant rate, provided that the time required for `process(data)` is bounded and that this time is less than the arrival frequency (plus some constant)

When average deviation counts

```
// same construct, slight rewrite
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
    for(;;) {
        // expected to fail only in extreme cases
        if(Data data; source >> data) {
            process(data);
        } else {
            // expected to be extremely rare
            throw ConsumeError{};
        }
    }
}
```

When average deviation counts

```
// same construct, slight rewrite
// suppose now that we have accessory tasks to perform
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   for(;;) {
      if(Data data; source >> data) {
         process(data);
         accessory_tasks();
      } else {
         throw ConsumeError{};
      }
   }
}
```

When average deviation counts

```
// same construct, slight rewrite
// suppose now that we have accessory tasks to perform
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
    for(;;) {
        if(Data data; source >> data) {
            process(data);
            accessory_tasks();
        } else {
            throw ConsumeError{};
        }
    }
}
```

At what frequency will `accessory_tasks()` be performed?

When average deviation counts

```
// same construct, slight rewrite
// suppose now that we have accessory tasks to perform
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   for(;;) {
      if(Data data; source >> data) {
         process(data);
         accessory_tasks();
      } else {
         throw ConsumeError{};
      }
   }
}
```

… in practice, we don't know, even if the time required for `process(data)` is bounded, as we depend on a blocking I/O operation

When average deviation counts

```
// same construct, slight rewrite
// suppose now that we have accessory tasks to perform
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
   for(;;) {
      if(Data data; source >> data) {
         process(data);
         accessory_tasks();
      } else {
         throw ConsumeError{};
      }
   }
}
```

… it might even never be called, e.g. if source is starving

When average deviation counts

```
// almost same construct, slight rewrite
// switching to non-blocking I/O
optional<Data> try_consume(istream&); // non-blocking
class ConsumeError{};
[[noreturn]] void processing_loop(istream &source) {
   for(;;) {
      if(auto data = try_consume(source); data) {
         process(data.value());
      } else if(!source) {
         throw ConsumeError{};
      }
      accessory_tasks();
   }
}
```

When average deviation counts

```
// almost same construct, slight rewrite
// switching to non-blocking I/O
optional<Data> try_consume(istream&); // non-blocking
class ConsumeError{};
[[noreturn]] void processing_loop(istream &source) {
   for(;;) {
      if(auto data = try_consume(source); data) {
         process(data.value());
      } else if(!source) {
         throw ConsumeError{};
      }
      accessory_tasks();
   }
}
```

With this construct, `accessory_tasks()` will be performed on every iteration

When average deviation counts

```
// almost same construct, slight rewrite
// switching to non-blocking I/O
optional<Data> try_consume(istream&); // non-blocking
class ConsumeError{};
[[noreturn]] void
processing_loop(istream &source) {
    for(;;) {
        if(auto data = try_consume(source); data) {
            process(data.value());
        } else if(!source) {
            throw ConsumeError{};
        }
        accessory_tasks();
    }
}
```

… if the time required for `process(data)` is bounded, guarantees can be offered as to the frequency at which `accessory_tasks()` will be performed

When average deviation counts

- A similar situation presents itself with the event-driven / polling duality

When average deviation counts

```
optional<Event> next_event(); // may be empty
class Registry {
   mutex m;
   vector<function<void(Event)>> to_call;
public:
   template <class F> void subscribe(F f) {
      lock_guard _{ m };
      to_call.emplace_back(f);
   }
   void callback(Event e) {
      lock_guard _{ m };
      for(auto & f : to_call) f(e);
   }
   void execute() {
      for(;;)
         if (auto e = next_event(); e)
            callback(e.value());
   }
   // ...
};
```

When average deviation counts

```
// ...
void reaction_to_event(Event);
// ...
auto reg = make_shared<Registry>();
// functions passed to reg's
// subscribe() function have to
// be thread-safe when called back
reg->subscribe(reaction_to_event);
reg->subscribe([](Event e){ /* ... */ });
thread th{ [reg] { reg->execute(); }};
th.detach();
// ...
```

When average deviation counts

```
// ...
void reaction_to_event(Event);
// ...
auto reg = make_shared<Registry>();
// functions passed to reg's
// subscribe() function have to
// be thread-safe when called back
reg->subscribe(reaction_to_event);
reg->subscribe([](Event e){ /* ... */ });
thread th{ [reg] { reg->execute(); }};
th.detach();
// ...
```

Event-driven code is generally seen as efficient

When average deviation counts

```
// ...
void reaction_to_event(Event);
// ...
auto reg = make_shared<Registry>();
// functions passed to reg's
// subscribe() function have to
// be thread-safe when called back
reg->subscribe(reaction_to_event);
reg->subscribe([](Event e){ /* ... */ });
thread th{ [reg] { reg->execute(); }};
th.detach();
// ...
```

… it reduces bandwidth requirements by only calling back if it's useful

When average deviation counts

```
// ...
void reaction_to_event(Event);
// ...
auto reg = make_shared<Registry>();
// functions passed to reg's
// subscribe() function have to
// be thread-safe when called back
reg->subscribe(reaction_to_event);
reg->subscribe([](Event e){ /* ... */ });
thread th{ [reg] { reg->execute(); }};
th.detach();
// ...
```

… however, with this approach, there can be moments when nothing occurs, and sudden bursts of intensive work

When average deviation counts

```
// ...
void reaction_to_event(Event);
// ...
auto reg = make_shared<Registry>();
// functions passed to reg's
// subscribe() function have to
// be thread-safe when called back
reg->subscribe(reaction_to_event);
reg->subscribe([](Event e){ /* ... */ });
thread th{ [reg] { reg->execute(); }};
th.detach();
// ...
```

… these bursts can impact the threads in which the called-back functions are executed and make it less responsive

When average deviation counts

- A similar situation presents itself with the event-driven / polling duality
  - Event-driven code tends to minimize resource consumption, in particular network bandwidth
  - For that reason, it’s generally seen as a better solution than polling code
  - It’s easy to confuse “it’s generally better” with “it’s always better”, and it’s dangerous to do so

When average deviation counts

```
// with polling, we have more extra work,
// but better control over bursts
vector<function<void(Event)>> to_call;
// ...
void reaction_to_event(Event);
// ... the thread-safe requirement on
// called-back functions goes away
to_call.emplace_back(reaction_to_event);
to_call.emplace_back([](Event e){
   // ...
});
// ...
```

When average deviation counts

```
// ...
deque<Event> to_process;
decltype(to_call.size()) pos {};
for(;;) {
   // ... consumption phase (could be a while instead of an if)
   if (auto e = next_event(); e) {
      to_process.push_back(e.value());
   }
   // ... processing phase
   while (enough_time_current_iteration() && !to_process.empty())
      if (pos != to_call.size()) {
         to_call[pos](to_process.front());
         ++pos;
      } else {
         to_process.pop_front();
         pos = {};
      }
   wait_for_next_cycle();
}
// ...
```

When average deviation counts

```
// ...
deque<Event> to_process;
decltype(to_call.size()) pos {};
for(;;) {
   // ... consumption phase
   if (auto e = next_event(); e) {
      to_process.push_back(e.value());
   }
   // ... processing phase
   while (enough_time_current_iteration() && !to_process.empty())
      if (pos != to_call.size()) {
         to_call[pos](to_process.front());
         ++pos;
      } else {
         to_process.pop_front();
         pos = {};
      }
   wait_for_next_cycle();
}
// ...
```

Waiting / sleeping is generally bad, but if the intent is to have a stable processing throughput, it can be reasonable (of course, it's better to perform useful work if possible)

When average deviation counts

```
// ...
deque<Event> to_process;
decltype(to_call.size()) pos {};
for(;;) {
   // ... processing phase
   while (enough_time_current_iteration() && !to_process.empty())
      if (pos != to_call.size()) {
         to_call[pos](to_process.front());
         ++pos;
      } else {
         to_process.pop_front();
         pos = {};
      }
   // ... consumption phase
   while (enough_time_current_iteration())
      if (auto e = next_event(); e)
         to_process.push_back(e.value());
   wait_for_next_cycle();
}
// ...
```

If processing has to begin at a fixed point in every iteration, then the processing phase can be put before the consumption phase. This removes the impact of time variation in the other parts of the iteration

When average deviation counts

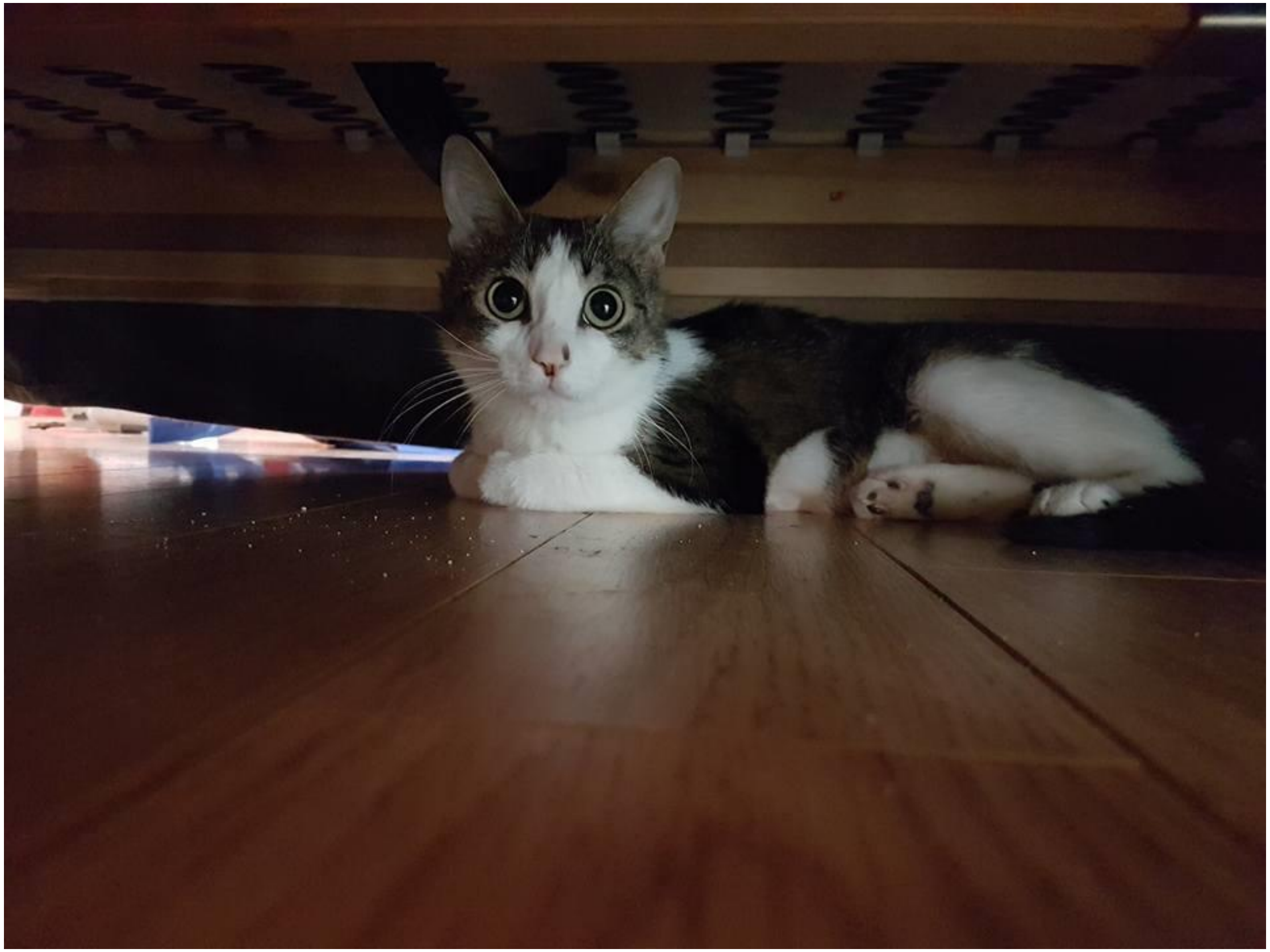
```
// ...
concurrent_queue<Event> to_process;
thread consumer{ [&]{
   for(;;)
      if(auto e = next_event(); e)
         to_process.add(e.value()); // add at the end
} };
thread processing{[&] {
   decltype(to_call.size()) pos {};
   optional<Event> e = to_process.try_extract(); // get potential front element; removes it
   for(;;) {
      while (enough_time_current_iteration())
         if (e && pos != to_call.size()) {
            to_call[pos](e.value());
            ++pos;
         } else if (pos == to_call.size()) {
            e = to_process.try_extract();
            pos = {};
         }
      wait_for_next_cycle();
   }
}};
// ...
```

If incoming events are not buffered on entry, it is also possible to split consumption and processing in distinct threads

When average deviation counts

- Polling approaches do consume more CPU cycle for the same computing results as event-driven approaches
  - On the other hand, they give the consumer more control over timing and resources
  - When avoiding bursts and reducing average deviation are important characteristics of program behavior, these aspects of a solution make a difference

Useful C++ constructs and tools for pessimistic programming



Useful C++ constructs and tools for pessimistic programming

- General rules
  - Contemporary optimizing compilers are very good
  - In most cases, just let them do their job!

Useful C++ constructs and tools for pessimistic programming

- General rules
  - Contemporary optimizing compilers are very good
  - In most cases, just let them do their job!
- Optimization is a multi-faceted problem
  - *To optimize* is always *to optimize for*…
- Sometimes, we want code to be optimized for non-obvious situations
  - The rare case when an error occurs and we have to do something *fast*
  - Situations where the average deviation has to be minimized

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - Atomics (used carefully)
  - `try_lock()` on mutexes, either as member or as non-member functions

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - Atomics (used carefully)
  - `try_lock()` on mutexes
    - `unique_lock<T>` offers `try_lock()` too
      - … and `try_lock_for()`
      - … and `try_lock_until()`
    - `lock()`, `lock_guard<T>` and `scoped_lock<Ts...>` all block until the mutex is acquired

Useful C++ constructs and tools for pessimistic programming

```
mutex m;
deque<Data> data;
// ...
thread th0{ [&] {
   vector<Data> v; // local buffer
   for(;;) {
      // the thread remains active, buffering data locally
      // if it cannot insert it in the deque
      v.emplace_back(receive_data());
      if (m.try_lock()) {
         lock_guard _{ m, adopt_lock };
         data.insert(end(data), begin(v), end(v));
         v.clear();
      }
   }
}};
// ...
```

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - `future.then()` (when it becomes available)
    - …maybe, if it suits the needs of your program
    - Reduces blocking, but not necessarily suited for pessimistic programming

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - Timed wait functions
    - e.g.: `wait_for()`, `wait_until()`
    - Lets client code determine the upper-bound on waiting time

Useful C++ constructs and tools for pessimistic programming

```
condition_variable cv;
mutex m;
bool ok = false;
// ...
thread th{ [&] {
   // while waiting for its signal, the thread
   // performs auxiliary tasks at approx. 1'000Hz
   unique_lock lck{ m };
   while (!cv.wait_for(lck, 1ms, [&ok]{ return ok; }))
      perform_auxiliary_tasks();
   perform_main_task();
}};
// ...
ok = true;
cv.notify_one();
// ...
```

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - `optional<T>` for return types
    - Exceptions tend to be optimized for the non-exceptional case, which can complicate worst-case planning
    - Raw pointer work too, but tend to be more brittle in the hands of some users

Useful C++ constructs and tools for pessimistic programming

```
template <class T>
   class concurrent_queue { // naïve
      mutex m;
      deque<T> impl;
   public:
      bool try_add(T obj) {
         if (m.try_lock()) {
            lock_guard _{ m, adopt_lock };
            impl.emplace_back(obj);
            return true;
         }
         return false;
      }
      // ...
```

Useful C++ constructs and tools for pessimistic programming

```
template <class T>
   class concurrent_queue { // naïve
      // ...
      // note: not exception-safe (can be made exception-safe
      // -- see next slide)
      optional<T> try_extract() {
         if (m.try_lock()) {
            lock_guard _{ m, adopt_lock };
            if (impl.empty()) return {};
            optional<T> res{ impl.front() };
            impl.pop_front();
            return res;
         }
         return {};
      }
   };
```

Useful C++ constructs and tools for pessimistic programming

```
template <class T>
   class concurrent_queue { // naïve
      // ...
      // note: exception-safe
      bool try_extract(T &res) {
         if (!m.try_lock()) return false;
         lock_guard _{ m, adopt_lock };
         if (impl.empty()) return false;
         res = impl.front();
         impl.pop_front();
         return true;
      }
   };
```

Useful C++ constructs and tools for pessimistic programming

- Things that don't block processing and allow for client-controlled progression
  - With C++20, the language will support attributes `[[likely]]` and `[[unlikely]]`
  - These can be used to guide optimization…
    - …but remember: compilers tend to be *very* good at this, and you will probably *pessimize* your execution instead
  - However, they are meant more to guide the optimizer toward *unintuitive* optimizations
    - Sometimes, one has context that compilers do not have
    - Sometimes, one can prefer optimizing cases that will actually slow down the program overall, e.g. because one programs for the worst case!

Useful C++ constructs and tools for pessimistic programming

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   // this supposes we want the catch to be the optimized-for path,
   // because it's when that happens that we want to be fast
   } [[likely]] catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

Useful C++ constructs and tools for pessimistic programming

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   // this supposes we want the catch to be the optimized-for path,
   // because it's when that happens that we want to be fast
   } [[likely]] catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

By definition, the `catch` clause should be rarely reached (it's meant to be exceptional!)

Useful C++ constructs and tools for pessimistic programming

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   // this supposes we want the catch to be the optimized-for path,
   // because it's when that happens that we want to be fast
   } [[likely]] catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

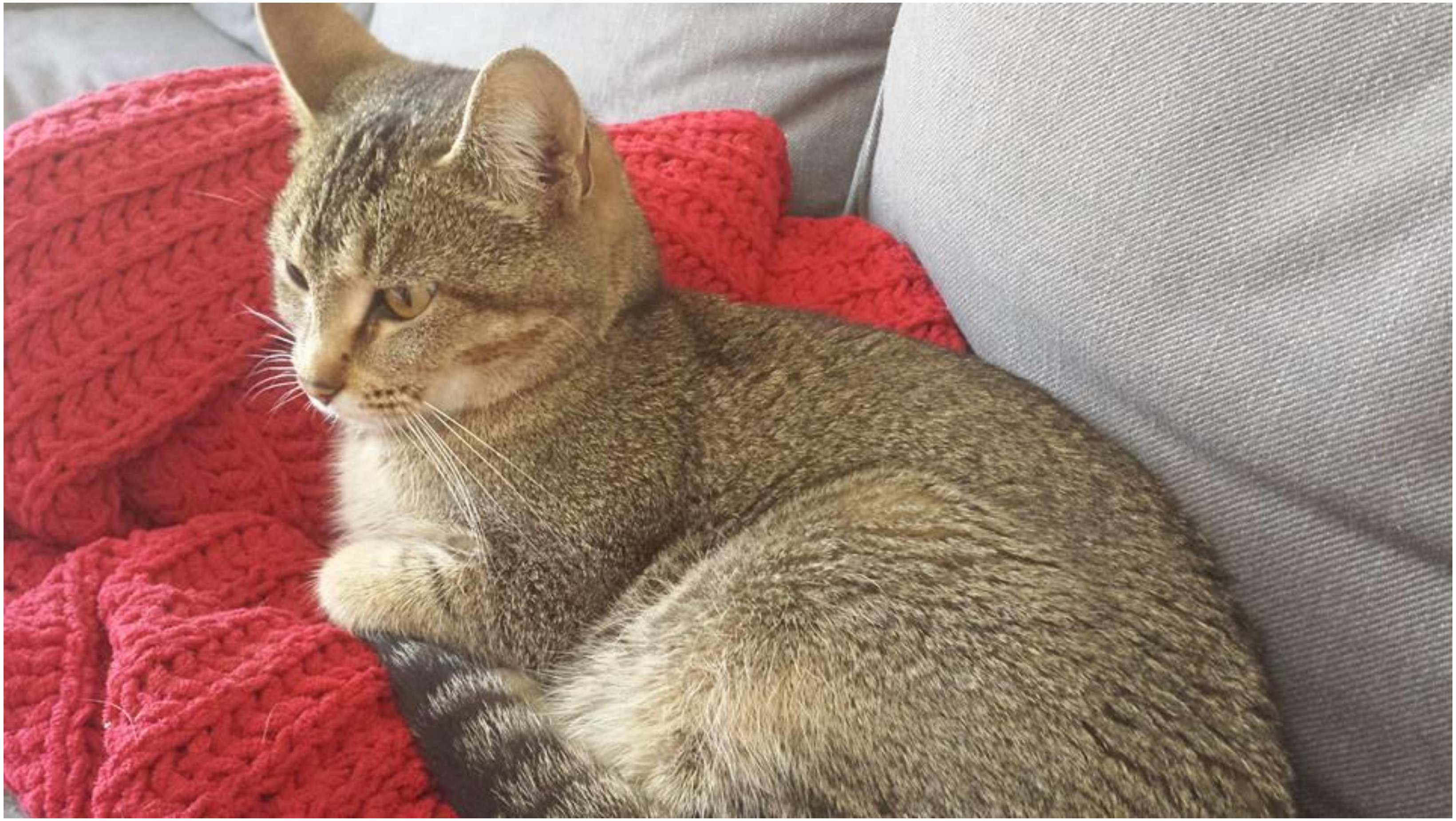
… which is why the `try` block is the optimized path. Marking the `catch` as `[[likely]]` means we will most probably be significantly slower than we would have been normally

Useful C++ constructs and tools for pessimistic programming

```
class CollisionRiskDetected{};
DrivingDirection drive(DrivingDirection current) {
   // ridiculously simplified
   if (all_clear(current)) return current;
   throw CollisionRiskDetected{};
}
// ...
auto dest = query_destination();
auto current = compute_direction(current_location(), dest);
while(current_location() != dest)
   try {
      current = drive(current); // hum
   // this supposes we want the catch to be the optimized-for path,
   // because it's when that happens that we want to be fast
   } [[likely]] catch(CollisionRiskDetected&) {
      avoid_collision();
   }
// ...
```

… that being said, if it's what one needs to to do react fast when encountering exceptional situations, then so be it

When computation exceeds budget



When computation exceeds budget

- There are cases where the upper-bound on execution time depends on input data, or when it is possible that execution time exceeds whatever time budget is available
  - Planning for artificial intelligence in games and other complex systems are typical cases for this

When computation exceeds budget

- There are cases where the upper-bound on execution time depends on input data, or when it is possible that execution time exceeds whatever time budget is available
  - Planning for artificial intelligence in games and other complex systems are typical cases for this
- In such situations, a great trick is to write smaller functions

When computation exceeds budget

- There are cases where the upper-bound on execution time depends on input data, or when it is possible that execution time exceeds whatever time budget is available
  - Planning for artificial intelligence in games and other complex systems are typical cases for this
- In such situations, a great trick is to write smaller functions
- When that's not enough, a well-known tactic is to subdivide computation in smaller steps

When computation exceeds budget

```
// old-school style (don't do this on today's computers)
bool time_slot_exceeded(); // called on occasion, to check if time remains
enum class State { Init, StepA, StepB, StepC, Done };
bool long_computation() {
   static auto state = State::Init;
   switch(state) {
      case State::Init:
         // ...
         state = State::StepA; [[fallthrough]]
      case State::StepA:
         // ...
         if (time_slot_exceeded()) return false;
         // ...
         state = State::StepB; [[fallthrough]]
      case State::StepB:
         // ...and so on
   }
   return true;
}
```

When computation exceeds budget

```
// more contemporary
bool time_slot_exceeded(); // called on occasion...
enum class State { Init, StepA, StepB, StepC, Done };
void long_computation(State &state) {
   switch(state) {
      case State::Init:
         // ...
         state = State::StepA; [[fallthrough]]
      case State::StepA:
         // ...
         if (time_slot_exceeded()) return;
         // ...
         state = State::StepB; [[fallthrough]]
      case State::StepB:
         // ...and so on
   }
}
```

When computation exceeds budget

```
// traditional transform()
template<class I, class O, class F>
  void transform(I bi, I ei, O bo, F f) {
    for(; bi != ei; ++bi, (void) ++bo)
      *bo = f(*bi);
}
```

When computation exceeds budget

```
// subdivided (segmented) version
template<
  class I, class O, class F, class Pred
>
  auto segm_transform
    (I bi, I ei, O bo, F f, Pred pred) {
    for(; bi != ei && pred();
        ++bi, (void) ++bo)
      *bo = f(*bi);
    return pair(bi, bo);
  }
```

Impact on client code (ask not what you can do for the caller…)

- Typically, client code for subdivided functions need to keep track of state
  - Where consumption of input data stopped
  - If appropriate, where production of output data stopped
  - How much time does the subdivided function have to perform its task
    - lambdas are incredibly useful for this!

Impact on client code (ask not what you can do for the caller...)

```
ProcessedData f(Data);
// ...
auto now = []{ return system_clock::now(); };
auto make_pred = [now] {
   return [now, deadline = now() + 2ms] {
      return now() < deadline;
   };
};
vector<Data> in = gather_data();
vector<ProcessedData> out;
// ...
auto p = segm_transform(
  begin(in), end(in), back_inserter(out), f, make_pred()
);
for(; p.first != end(in); p = segm_transform(
       p.first, end(in), back_inserter(out), f, make_pred()
    ))
   ;
```

Impact on client code (ask not what you can do for the caller…)

- Subdivided functions are slower than their “traditional” counterparts
  - Some overhead on every call
  - More calls for same result
    - Sometimes, like in the case of a compression algorithm divided in steps, results can be of lower quality (depending on implementation)

Impact on client code (ask not what you can do for the caller…)

- Subdivided algorithms can, on the other hand, be used in critical systems
  - Well-suited for auxiliary tasks that consume the remaining time of an iteration, after the critical tasks are done
  - Depending on available time, can make progress on tasks that would otherwise introduce unwanted delays later

Impact on client code (ask not what you can do for the caller...)

```
// ...
for(;;) {
    critical_tasks();
    // candidate for subdivision, to
    // use remaining available time,
    // provided the loop is expected
    // to iterate at a fixed rate
    accessory_tasks();
    // might be superfluous
    wait_for_next_iteration();
}
```

Impact on client code (ask not what you can do for the caller...)

```
// ... more concrete example
// long_term_plan() should ideally return something like optional<Plan>
template <class Pred> void long_term_planning(Pred pred) { /* ... */ }
// ...
const auto iter_duration =
   milliseconds{ 1000.0 / Game::frame_rate() };
for(auto cur = now(); Game::ongoing(); cur = now()) {
   // critical things for the immediate experience
   display_scene();
   prepare_next_scene();
   // important things that take time to set up
   // The argument is a continuation predicate
   long_term_planning([deadline = cur + iter_duration] {
      // one could add a 'comfort zone constant' here
      return now() < deadline;
   });
}
// ...
```

Coroutines

- You might have noticed that most of the examples provided require state management
  - Typically makes for more complex client code
  - Subdivided functions perform their tasks in smaller steps than their non-subdivided counterparts
  - Usually, client code needs to control operation granularity

Coroutines

- This sort of situation is a nice fit for coroutines
  - Resumable functions
  - State implicitly managed between calls, at least in part
- For C++, currently a TS
  - Hopeful for C++20

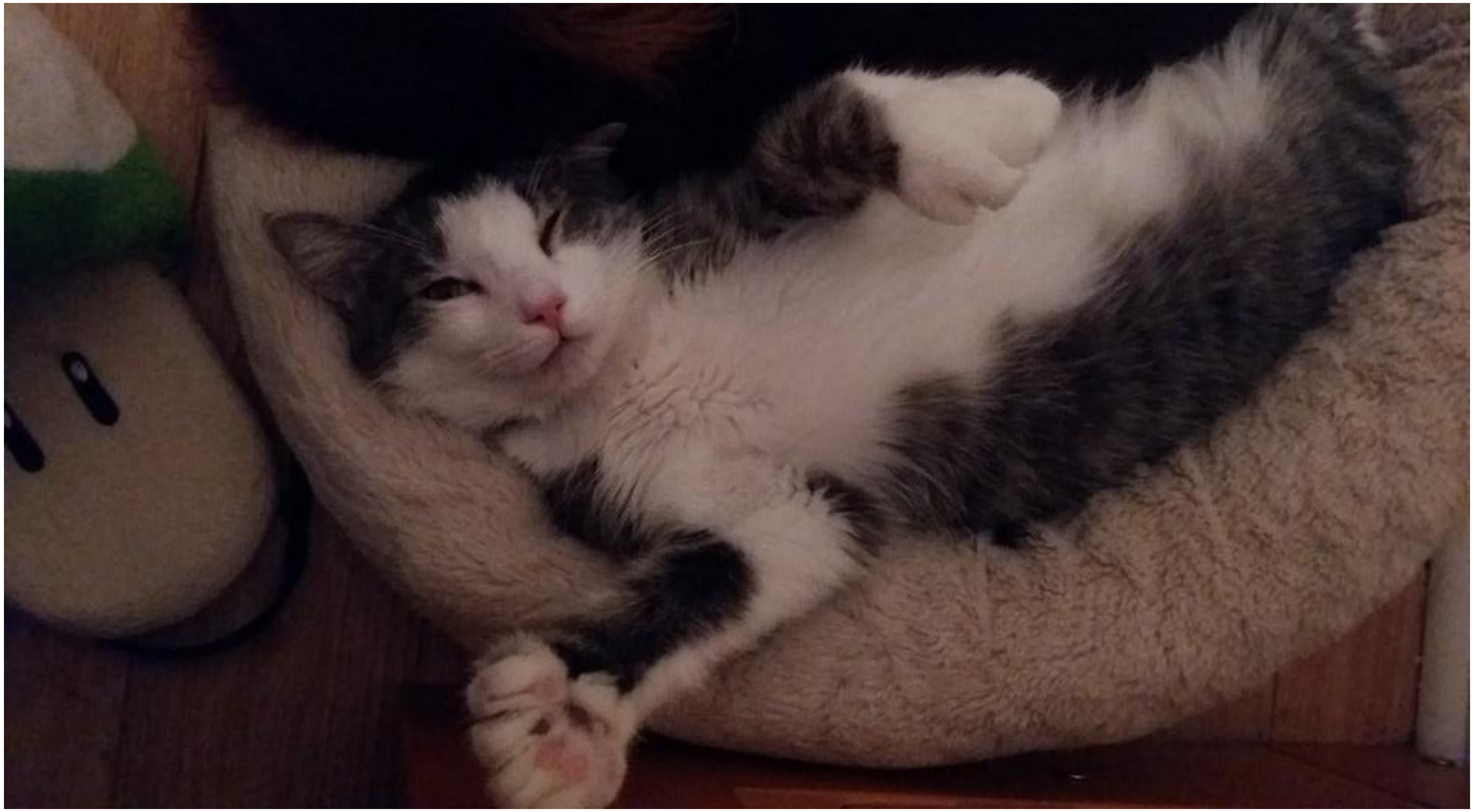
Coroutines

```
#include <experimental/generator>
#include <iostream>
#include <chrono>
#include <vector>
using namespace std;
using namespace std::chrono;
using experimental::generator;
template <class Pred>
   generator<vector<int>> even_integers(Pred pred) {
      vector<int> res;
      for (int n = 0; ; n += 2) {
         if (!pred()) {
            co_yield res;
            res.clear();
         }
         res.emplace_back(n);
      }
   }
// ...
```

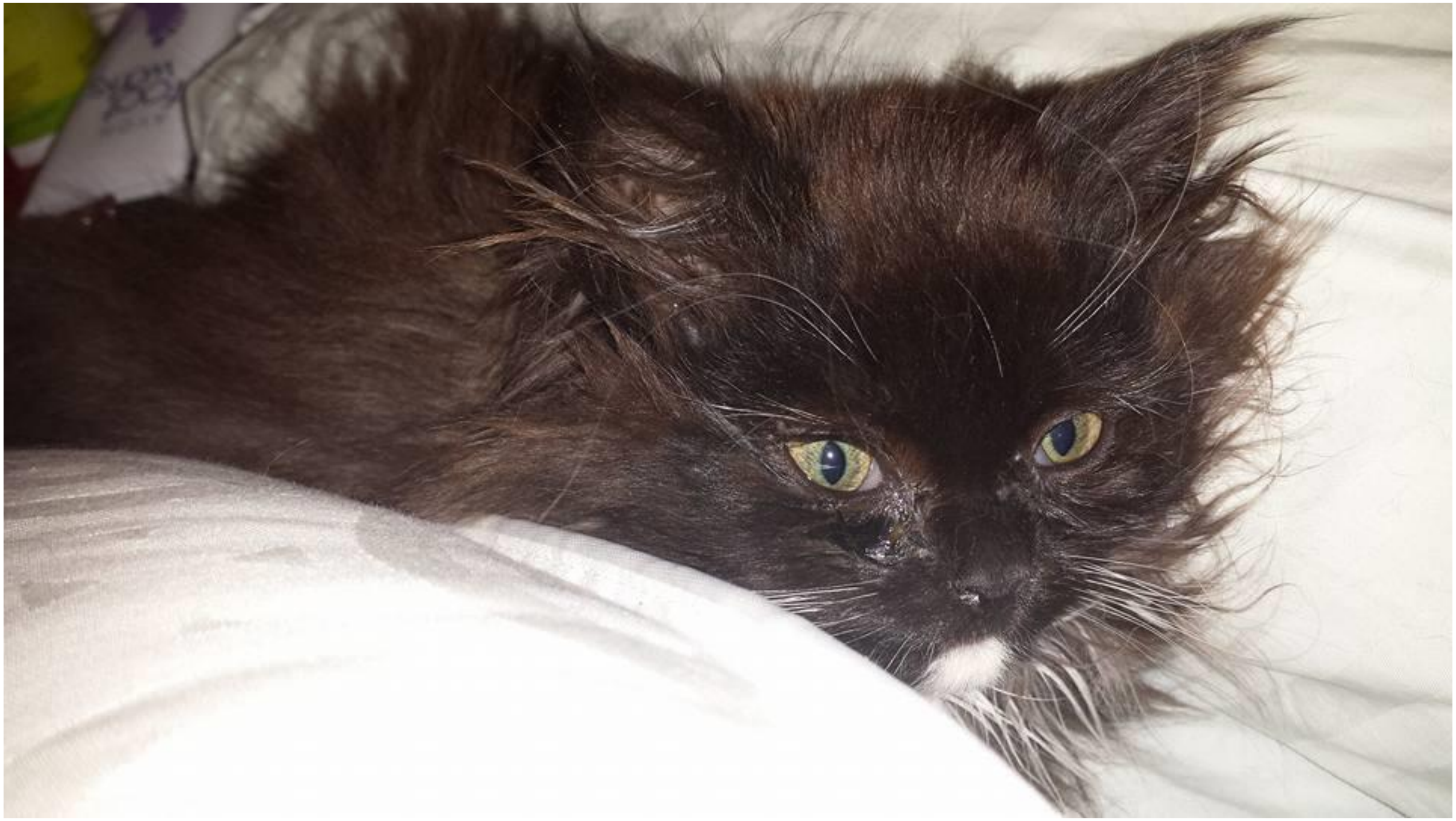
Coroutines

```
// ...
int main() {
   auto deadline = system_clock::now() + 500us;
   auto pred = [&deadline] {
     return system_clock::now() < deadline;
   }
   for (auto n : even_integers(pred)) {
      cout << "\nComputed " << n.size()
           << " even integers" << endl;
      for (auto m : n) cout << m << ' ';
      cout << endl << endl;
      deadline = system_clock::now() + 1ms;
   }
}
```

Questions?



Bonus fun: data-invariant programming



Bonus fun: data-invariant programming

- Imagine a situation where your code needs to compare two simple, C-style strings and return true only if they are identical (case-sensitive)

Bonus fun: data-invariant programming

- Imagine a situation where your code needs to compare two simple, C-style strings and return true only if they are identical (case-sensitive)

```
// possible solution
bool
compare(const char *p0, const char *p1) {
    if (!p0 || !p1)
        return !p0 && !p1;
    for(; *p0 && *p1 && *p0==*p1; ++p0, ++p1)
        ;
    return *p0 == *p1;
}
```

Bonus fun: data-invariant programming

- Imagine a situation where your code needs to compare two simple, C-style strings and return true only if they are identical (case-sensitive)

```
// possible solution
bool
compare(const char *p0, const char *p1) {
    if (!p0 || !p1)
        return !p0 && !p1;
    for(; *p0 && *p1 && *p0==*p1; ++p0, ++p1)
        ;
    return *p0 == *p1;
}
```

This simple solution works, but leaks information to a hostile observer. For some use-cases, this can be bad

Bonus fun: data-invariant programming

- Imagine a situation where your code needs to compare two simple, C-style strings and return true only if they are identical (case-sensitive)

```
// possible solution
bool
compare(const char *p0, const char *p1) {
    if (!p0 || !p1)
        return !p0 && !p1;
    for(; *p0 && *p1 && *p0==*p1; ++p0, ++p1)
        ;
    return *p0 == *p1;
}
```

… the leakage comes from the fact that we return as soon as we have a solution

Bonus fun: data-invariant programming

- Imagine a situation where your code needs to compare two simple, C-style strings and return true only if they are identical (case-sensitive)

```
// possible solution
bool
compare(const char *p0, const char *p1) {
    if (!p0 || !p1)
        return !p0 && !p1;
    for(; *p0 && *p1 && *p0==*p1; ++p0, ++p1)
        ;
    return *p0 == *p1;
}
```

… which means that, if one sequence is known (an attempt) and the other one is secret, we can infer from successive calls if we are getting closer to discovering the secret

Bonus fun: data-invariant programming

- A function is data-invariant if its execution time depends only on the length of its input
  - A proposal for standardization has been on the burner since 2014 (n4314)
    - “*One of the hardest challenges when implementing cryptographic functionality with well-defined mathematical properties is to avoid side-channel attacks, that is, security breaches exploiting physical effects dependent on secret data when performing a cryptographic operation. Such effects include variances in timing of execution, power consumption of the machine, or noise produced by voltage regulators of the CPU. C++ does not consider such effects as part of the observable behavior of the abstract machine ([intro.execution]), thereby allowing implementations to vary these properties in unspecified ways*”
  - A data-invariant function would be known as such to the compiler, which would affect the ways in which it is optimized

Bonus fun: data-invariant programming

- How could we transform this into a data-invariant equivalent?

```
bool
compare(const char *p0, const char *p1) {
    if (!p0 || !p1)
        return !p0 && !p1;
    for(; *p0 && *p1 && *p0==*p1;
            ++p0, ++p1)
        ;
    return *p0 == *p1;
}
```

Bonus fun: data-invariant programming

- One thing we need to do is ensure we always go to the end of the longest string
  - … or make sure all arguments are of fixed length, which can be seen as preprocessing
  - … or take a length (presumed valid) as argument and always compare up to that length
  - We'll make that supposition to simplify discussion

Bonus fun: data-invariant programming

```
// simplified version
// precondition: p0 && p1
// precondition: strlen(p0 <= n) && strlen(p1) <= n
// still not data-invariant
bool
compare(const char *p0, const char *p1, size_t n) {
    for(size_t i = 0; i < n; ++i) {
        if (s0[i] != s1[i]) {
            return false;
        }
    }
    return true;
}
```

Bonus fun: data-invariant programming

```
// simplified version
// precondition: p0 && p1
// precondition: strlen(p0 <= n) && strlen(p1) <= n
// still not data-invariant
bool
compare(const char *p0, const char *p1, size_t n) {
    for(size_t i = 0; i < n; ++i) {
        if (s0[i] != s1[i]) {
            return false; // early return
        }
    }
    return true;
}
```

Bonus fun: data-invariant programming

```
// simplified version
// precondition: p0 && p1
// precondition: strlen(p0 <= n) && strlen(p1) <= n
// still not data-invariant. Do you see why?
bool
compare(const char *p0, const char *p1, size_t n) {
   bool result = true;
   for(size_t i = 0; i < n; ++i) {
      if (s0[i] != s1[i]) {
         result = false;
      }
   }
   return result;
}
```

Bonus fun: data-invariant programming

```
// simplified version
// precondition: p0 && p1
// precondition: strlen(p0 <= n) && strlen(p1) <= n
// still not data-invariant. Do you see why?
bool
compare(const char *p0, const char *p1, size_t n) {
    bool result = true;
    for(size_t i = 0; i < n; ++i) {
        result = result && (s0[i] == s1[i]);
    }
    return result;
}
```

Bonus fun: data-invariant programming

```
// simplified version
// precondition: p0 && p1
// precondition: strlen(p0 <= n) && strlen(p1) <= n
// this one is data-invariant. Do you see why?
bool
compare(const char *p0, const char *p1, size_t n) {
   bool result = true;
   for(size_t i = 0; i < n; ++i) {
      result &= static_cast<int>(s0[i] == s1[i]);
   }
   return result;
}
```

Bonus fun: data-invariant programming

- Data-invariant functions strive for as little variation as possible (ideally, none) on any other factor than input length
  - It's much trickier to write than people expect
- It can be seen as a way to write function such that execution time is always the worst case
  - … and it can actually be useful!

Thanks for your participation!