CppNorth 2023

Moving an Existing Project to C++ 20 for Fun, Beauty... and Results!

Patrice Roy

Moving an Existing Project to C++ 20 for Fun, Beauty… and Results!

Patrice Roy, CeFTI, Université de Sherbrooke, Collège Lionel-Groulx

Patrice.Roy@USherbrooke.ca, Patrice.Roy@clg.qc.ca

July 2023

The Situation

- Suppose the following (hypothetical?) situation
  - An existing project (reduced version, because this is a one hour talk)

The Situation

- Suppose the following (hypothetical?) situation
  - An existing project (reduced version, because this is a one hour talk)
  - Disk-based I/O

The Situation

- Suppose the following (hypothetical?) situation
  - An existing project (reduced version, because this is a one hour talk)
  - Disk-based I/O
  - Parallel and concurrent processing of data

The Situation

- Suppose the following (hypothetical?) situation
  - An existing project (reduced version, because this is a one hour talk)
  - Disk-based I/O
  - Parallel and concurrent processing of data
  - Some managers (including singletons)

The Situation

- Suppose the following (hypothetical?) situation
  - An existing project (reduced version, because this is a one hour talk)
  - Disk-based I/O
  - Parallel and concurrent processing of data
  - Some managers (including singletons)
  - Some compile-time algorithm selection, etc.

The Situation

- The existing project *works*
  - It « makes money »
- …but over time, some measure of « technical debt » accumulates

The Situation

- The opportunity to migrate to a new version of the language presents itself
  - In our case, it’s a migration from C++17 to C++20
- Let’s seize the opportunity and see how we could benefit from it

~~The Situation~~ Words of Warning

- Some words of warning need to be expressed

~~The Situation~~ Words of Warning

- Some words of warning need to be expressed
  - The following project is artificial, and has been created for the sake of this talk
    - Technically, for another talk in another language, but hey, life…

~~The Situation~~ Words of Warning

- Some words of warning need to be expressed
  - The following project is artificial, and has been created for the sake of this talk
  - Our contact with C++20 will be *extremely* superficial
    - C++20 is a **gigantic** update to the C++ language
    - We will barely touch the tip of the iceberg

~~The Situation~~ Words of Warning

- Some words of warning need to be expressed
  - The following project is artificial, and has been created for the sake of this talk
  - Our contact with C++20 will be *extremely* superficial
  - Some of the new features will actually slow us down slightly
    - In some cases, it’s a matter of using the right tool for the right task
    - In other cases, it’s because the features are solidly in place but require a lot of work to fully benefit from… this too should change with time

~~The Situation~~ Words of Warning

- Some words of warning need to be expressed
  - The following project is artificial, and has been created for the sake of this talk
  - Our contact with C++20 will be *extremely* superficial
  - Some of the new features will actually slow us down slightly
  - Overall, let’s see if the experience is positive

Who am I?

- Father of five, aged 28 to 10
- I feed and take care of a varying number of animals
  - Take a look at *Paws of Britannia* with your favorite search engine
- I used to write software for industrial electrical breakers and military flight simulators
  - CAE Electronics Ltd, IREQ
- Full-time professor since 1998
  - Collège Lionel-Groulx, Université de Sherbrooke
- I work a lot with game programmers
- WG21 and WG23 member (but I missed recent WG23 meetings)
  - Involved quite a bit with SG14, the low-latency study group
  - Occasional WG21 secretary
- Etc.

Our starting point

Our starting point

- To see the whole sources (both the pre-C++20 version and the C++20 version), see https://bit.ly/43vz8vz
  - Reminder: this is an artificial project (inspired from a real one)

Our starting point

- Overall idea
  - Small system that performs analysis on mechanical pieces
    - Warning: I am *really* not a specialist of that specific application domain

Our starting point

- Overall idea
  - Small system that performs analysis on mechanical pieces
  - A system feeds pieces to our product, and our product produces diagnostics and detects defects
    - In this example, randomness will play a big role

Our starting point

- Overall idea
  - Small system that performs analysis on mechanical pieces
  - A system feeds pieces to our product, and our product produces diagnostics and detects defects
  - A summary analysis of the defective pieces is done

Our starting point

- Overall idea
  - Small system that performs analysis on mechanical pieces
  - A system feeds pieces to our product, and our product produces diagnostics and detects defects
  - A summary analysis of the defective pieces is done
  - A more in-depth analysis of the non-defective pieces follows
    - This second analysis is done concurrently, at least in part

Our starting point

- Overall idea
  - Small system that performs analysis on mechanical pieces
  - A system feeds pieces to our product, and our product produces diagnostics and detects defects
  - A summary analysis of the defective pieces is done
  - A more in-depth analysis of the non-defective pieces follows
  - A report is produced on disk at the end of the overall process

Our starting point

```
int main() {
   // - create pieces
   // - sequentially process the pieces,
   //   diagnose them and reject the
   //   defective ones
   // - analyze the cause of rejections and produce
   //   some statistics
   // - compute the average rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
    // - create pieces
    // - sequentially process the pieces,
    //   diagnose them and reject the
    //   defective ones
    // - analyze the cause of rejections and produce
    //   some statistics
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
    // - create pieces
    // - sequentially proc
    //   diagnose them and reject the
    //   defective ones
    // - analyze the cause of rejections and produce
    //   some statistics
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
class Piece { /* ... */ };
```

Our starting point

```
int main() {
    // - create pieces
    // -
    //
    //
    //
    //
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
class Piece {
public:
    using id_type = int;
    using elem_type = std::tuple<std::string, int, float>;
    // ...
};
```

Our st

```
int main
    // -
    // -
    //
    //
    //
    //
    // -
    //    b
    // - perform an in-depth analysis of valid pieces
}
```

```
class Piece {
  // ...
private:
  id_type id_;
  std::vector<elem_type> elements;
  float quality_;
  static float
    evaluate_quality(const std::vector<elem_type> &v) {
    using namespace std;
    return accumulate(v.begin(), v.end(), 0.f,
                      [](float cur, auto &&tup) {
      return max(cur, get<float>(tup));
    });
  }
  // ...
};
```

Our sta

```
int main
    // - c
    // - s
    //   c
    //   c
    //
    //
    //
    // - p
}
```

```
class Piece {
  // ...
public:
  using iterator = typename
    std::vector<elem_type>::iterator;
  using const_iterator = typename
    std::vector<elem_type>::const_iterator;
  iterator begin() { return elements.begin(); }
  iterator end() { return elements.end(); }
  const_iterator begin() const {
    return elements.begin();
  }
  const_iterator end() const {
    return elements.end();
  }
  // ...
};
```

Our sta

```
int main
  // - 
  // - 
  //
  //
  //

  //
  //
  // - 
}
```

```
class Piece {
  // ...
public:
  using iterator = typename
    std::vector<elem_type>::iterator;
  using const_iterator = typename
    std::vector<elem_type>::const_iterator;
  iterator begin() { return elements.begin(); }
  iterator end() { return elements.end(); }
  const_iterator begin() const {
    return elements.begin();
  }
  const_iterator end() const {
    return elements.end();
  }
   // ...
```

Such code will be easier to write with C++23's « deducing this », but we don't have C++23 yet

Our st

```
int m
    /
    //
    //
    //
    //
    //
    // -
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
class Piece {
  // ...
  Piece(id_type id, float quality)
    : id_{ id }, quality_{ quality } {
  }
  Piece(id_type id, const std::vector<elem_type> &elems)
    : id_{ id }, elements{ elems },
      quality_{ evaluate_quality(elems) } {
  }
  id_type id() const { return id_; }
   // ...
};
```

Our st

```
int mai
    // -
    // -
    //
    //
    /
    //
    //
    // -
}
```

```
class Piece {
  // ...
  bool operator==(const Piece &other) const noexcept {
    return id() == other.id() &&
           elements == other.elements;
  }
  bool operator!=(const Piece &other) const noexcept {
    return !(*this == other);
  }
  bool operator<(const Piece &other) const noexcept {
    return id() < other.id() ||
           id() == other.id() &&
           std::lexicographical_compare(begin(), end(),
             other.begin(), other.end());
  }
  // operators >, <= and >=
  // ...
};
```

Our starting point

```
int
    // - perform an in-depth analysis of valid pieces
}
```

```
class Piece {
    // ...
    float quality() const noexcept {
        return quality_;
    }
    friend std::ostream &operator<<(std::ostream &, const Piece &);
};
std::ostream &operator<<(std::ostream &, const Piece::elem_type &);
std::istream &operator>>(std::istream &, Piece::elem_type &);
std::istream &operator>>(std::istream &, Piece &);
float average_quality(const std::vector<Piece> &);
```

Our starting point

```
int main() {
   // - create pieces
   // - sequentially process the pieces,
   //   diagnose them and reject the
   //   defective ones
   // - analyze the cause of rejections and produce
   //   some statistics
   // - compute the average rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
    // - create pieces
    // - sequentially
    //   diagnose the
    //   defective on
    // - analyze the
    //   some statistics
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
static constexpr int npieces = 10'000'000,
                     nelems = 7;
auto v = create_pieces(npieces, nelems);
```

Our starting point

```
int main() {
   // - create p
   // - seque
   //   diagno
   //   defecti
   // - analyze
   //   some st
   // - compute
   //   both va
   // - perform
}
```

```
static constexpr int npieces = 10'000'000,
                     nelems = 7;
cout << "Creation of " << format_integral(npieces)
     << " pieces with [1,"
     << nelems << "] elements each..." << flush;
auto [v, dtc] = test([] {
   return create_pieces(npieces, nelems);
});
high_resolution_clock::duration total_time{};
cout << " completed in "
     << duration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

Our starting point

```
int main() {
   // - create
   // - seq
   //   diag
   //   defect
   // - analyz
   //   some s
   // - comput
   //   both v
   // - perfor
}
```

```
static constexpr int npieces = 10'000'000,
                     nelems = 7;
cout << "Creation of " << format_integral(npieces)
     << " pieces with [1,"
     << nelems << "] elements each..." << flush;
auto [v, dtc] = test([] {
   return create_pieces(npieces, nelems);
});
high_resolution_clock::duration total_time{};
cout << " completed in "
     << duration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

Our starting point

```
inline std::string pad_left(const std::string &s, char c,
                            std::string::size_type width) {
   return s.size() < width ?
      std::string(width - s.size(), c) + s : s;
}
template <class T>
   constexpr T absolute(T val) { /* ... */    }
template <class T>
   std::string format_integral(T val) { /* ... */ }
```

```
lush;
()
cout << "Diagnostics sequentiels..." << flush;
```

```
   // - perform
}
```

Our

```
int
```

```
inline std::string pad_left(const std::string &s, char c,
                            std::string::size_type width) {
   // ...
}
template <class T>
   constexpr T absolute(T val) {
      if constexpr (std::is_unsigned_v<T>)
         return val;
      else
         return val < 0? -val : val;
   }
template <class T>
   std::string format_integral(T val) { /* ... */ }
```

```
h;
ms\n";
cout << "Diagnostics sequentiels..." << flush;
```

```
   //
   //   both
   // - perform
}
```

```
inline std::string pad_left(const std::string &s, char c,
                            std::string::size_type width) { /* ... */ }
template <class T>
  constexpr T absolute(T val) { /* ... */ }
template <class T>
  std::string format_integral(T val) {
    using namespace std::literals;
    static_assert(std::is_integral_v<T>);
    auto sign_str = val < 0? "-"s : ""s;
    val = absolute(val);
    std::string s;
    while (val >= 1000) {
      s = "'"s + pad_left(std::to_string(val % 1000),'0',3) + s;
      val /= 1000;
    }
    return sign_str + std::to_string(val) + s;
  }
```

```
inline std::string pad_left(const std::string &s, char c,
                            std::string::size_type width) { /* ... */ }
template <class T>
  constexpr T absolute(T val) { /* ... */ }
template <class T>
  std::string format_integral(T val) {
    using namespace std::literals;
    static_assert(std::is_integral_v<T>);
    auto sign_str = val < ... : ""s;
    val = absolute(val);
    std::string s;
    while (val >= 100
      s = "'"s + pad_
      val /= 1000;
    }
    return sign_str +
  }
```

```
assert(format_integral(0) == "0"s);
assert(format_integral(12) == "12"s);
assert(format_integral(-12) == "-12"s);
assert(format_integral(12345) == "12'345"s);
assert(
   format_integral(10000000) == "10'000'000"s
);
```

Our starting point

```
int main() {
   // - create p
   // - seque
   //   diagno
   //   defecti
   // - analyze
   //   some sta
   // - compute
   //   both val
   // - perform
}
```

```
static constexpr int npieces = 10'000'000,
                     nelems = 7;
cout << "Creation of " << format_integral(npieces)
     << " pieces with [1,"
     << nelems << "] elements each..." << flush;
auto [v, dtc] = test([] {
   return create_pieces(npieces, nelems);
});
high_resolution_clock::duration total_time{};
cout << " completed in "
     << duration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

Our starting point

```
int main() {
   //
   // - perform
}
```

```
static constexpr int npieces = 10'000'000,
egral(npieces)
" << flush;
ems);
ime{};
dtc).count()
cout << "Sequential diagnostics..." << flush;
```

```
template <class F, class ... Args>
   auto test(F f, Args &&... args) {
      using namespace std;
      using namespace std::chrono;
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return pair{ res, post - pre };
   }
```

Our starting point

```
int main() {
   // - create p
   // - seque
   //   diagno
   //   defecti
   // - analyze
   //   some sta
   // - compute
   //   both va
   // - perform
}
```

```
static constexpr int npieces = 10'000'000,
                     nelems = 7;
cout << "Creation of " << format_integral(npieces)
     << " pieces with [1,"
     << nelems << "] elements each..." << flush;
auto [v, dtc] = test([] {
   return create_pieces(npieces, nelems);
});
high_resolution_clock::duration total_time{};
cout << " completed in "
     << duration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

Our starting point

```
// note : not thread-safe
template <class PRNG>
  string generate_name(PRNG &prng) {
    // note : I'm totally no good at mechanical things :)
    static const string names[]{
      "bolt", "ram", "rotating head", "nut",
      "power supply", "rim"
    };
    static uniform_int_distribution<size_t> die{ 0, size(names) - 1 };
    return names[die(prng)];
  }
string combine_name(int id, const string &name) { /* ... */ }
vector<Piece> create_pieces(size_t n, size_t m) { /* ... */ }
```

Our starting point

```
int m
    //
    //
    //
    //
    //
    //    some
    // - compute
    //   both va
    // - perform
}
```

```
template <class PRNG>
   string generate_name(PRNG &prng) { /* ... */ }
string combine_name(int id, const string &name) {
   return name + " "s + to_string(id);
}
vector<Piece> create_pieces(size_t n, size_t m) { /* ... */ }
```

```
high_resolution_clock::duration total_time{};
cout << " completed in "
     << duration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

```
template <class PRNG> string generate_name(PRNG &prng) { /* ... */ }
string combine_name(int id, const string &name) { /* ... */ }
vector<Piece> create_pieces(size_t n, size_t m) {
  vector<Piece> v;
  v.reserve(n);
  mt19937 prng{ random_device{}() };
  uniform_int_distribution<size_t> die(1, m);
  uniform_real_distribution<float> prob{ 0.975f, 1.0f };
  for (size_t i = 0; i != n; ++i)
    v.emplace_back(create_piece(Piece::id_type(i), prng, die, prob));
  return v;
}
```

```
    // - com
    //   both va
    // - perform
}
```

```
        completed in "
      uration_cast<milliseconds>(dtc).count()
     << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

```
template <class PRNG> string generate_name(PRNG &prng) { /* ... */ }
string combine_name(int id, const string &name) { /* ... */ }
vector<Piece> create_pieces(size_t n, size_t m) {
 vector<Piece> v;
   v.reserve(n);
   mt19937 prng{ random_device{}() };
   uniform_int_distribution<size_t> die(1, m);
   uniform_real_distribution<float> prob{ 0.975f, 1.0f };
   for (size_t i = 0; i != n; ++i) {
      v.emplace_back(create_piece(Piece::id_type(i), prng, die, prob));
   }
   return v;
}
```

```
    //  - com
    //    both va
    //  - perform
 }
```

```
             completed in "
              ation_cast<milliseconds>(dtc).count()
          << " ms\n";
cout << "Sequential diagnostics..." << flush;
```

```
template <class PRNG> string generate_name(PRNG &prng) { /* ... */ }
string combine_name(int id, const string &name) { /* ... */ }
vector<Piece> create_pieces(size_t n, size_t m) {
 vector<Piece> v;
   v.reserve(n);
```

```
template <class PRNG, class DISTN, class DISTR>
  Piece create_piece(size_t id, PRNG &prng, DISTN die, DISTR &prob) {
    vector<Piece::elem_type> elems;
    auto nelems = die(prng);
    for (int i = 0; i != nelems; ++i)
      elems.emplace_back(
        combine_name(i, generate_name(prng)), i, prob(prng)
      );
    return { Piece::id_type(id), elems };
  }
```

```
  }
```

Our starting point

```
int main() {
   // - create pieces
   // - sequentially process the pieces,
   //   diagnose them and reject the
   //   defective ones
   // - analyze the cause of rejections and produce
   //   some statistics
   // - compute the average rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
ofstream out{ "out.txt" };
int ndefects = 0;
vector<Piece> valids;
auto &rej_analyst = RejectionAnalyst::get();
for (auto &&p : v)
    if (auto [piece, ok, reason] = diagnose(std::move(p)); ok) {
        out << piece << '\n';
        valids.push_back(std::move(piece));
    } else {
        ++ndefects;
        out << "DEFECT DETECTED ON " << piece << "; reason: "
            << quoted(reason) << '\n';
        rej_analyst.reject(std::move(piece), std::move(reason));
    }
}
```

```
ofstream out{ "out.txt" };
int ndefects = 0;
auto [valids, dtd] = test([&] {
   vector<Piece> valids;
   auto &rej_analyst = RejectionAnalyst::get();
   for (auto &&p : v) {
      if (auto [piece, ok, reason] = diagnose(std::move(p)); ok) {
         out << piece << '\n';
         valids.push_back(std::move(piece));
      } else {
         ++ndefects;
         out << "DEFECT DETECTED ON " << piece << "; reason: "
             << quoted(reason) << '\n';
         rej_analyst.reject (std::move(piece), std::move(reason));
      }
   }
   return valids;
});
```

```
ofstream out{ "out.txt" };
int ndefects = 0;
auto [valids, dtd] = test([&] {
   vector<Piece> valids;
   auto &rej_analyst = RejectionAnalyst::get();
   for (auto &&p : v) {
      if (auto [piece, ok, reason] = diagnose(std::move(p)); ok) {
         out << piece << '\n';
         valids.push_back(std::move(piece));
      } else {
         ++ndefects;
         out << "DEFECT DETECTED ON " << piece << "; reason: "
             << quoted(reason) << '\n';
         rej_analyst.reject (std::move(piece), std::move(reason));
      }
   }
   return valids;
});
```

```
template <class T>
  struct Diagnostic {
    T obj;
    bool ok = true;
    std::string reason;
    Diagnostic(T obj) : obj{ std::move(obj) } {
    }
    Diagnostic(T obj, bool ok) : obj{ std::move(obj) }, ok{ ok } {
    }
    Diagnostic(T obj, bool ok, std::string_view reason)
      : obj{ std::move(obj) }, ok{ ok }, reason{ reason } {
      assert(!ok);
    }
  };
```

```
ofstream out{ "out.txt" };
int ndefects = 0;
auto [
   ve
   a
   f
   }
   re
});
```

```
std::string reason_to_reject(std::mt19937 &prng) { /* ... */ }
Diagnostic<Piece> diagnose(Piece p) {
  static mt19937 prng{ random_device{}() };
  // bad week at the office :)
  constexpr float defect_prob = 0.02f;
  bool ok = defect_prob > 1.0f - p.quality();
  return !ok ?
    Diagnostic<Piece>{
      std::move(p), false, reason_to_reject(prng)
    } :
    Diagnostic<Piece>{ std::move(p) };
}
```

```cpp
ofstream out{ "out.txt" };
int ndefects = 0;
auto [v
   ve
   a
   f



   }
   re
});
```

```cpp
std::string reason_to_reject(std::mt19937 &prng) { /* ... */ }
Diagnostic<Piece> diagnose(Piece p) {
  static mt19937 prng{ random_device{}() };
  // bad week at the office :)
  constexpr float defect_prob = 0.02f;
  bool ok = defect_prob > 1.0f - p.quality();
  return !ok ?
    Diagnostic<Piece>{
      std::move(p), false, reason_to_reject(prng)
    } :
    Diagnostic<Piece>{ std::move(p) };
}
```

```cpp
// note : not thread-safe
std::string reason_to_reject(std::mt19937 &prng) {
   static const string_view reason[]{
      "damaged"sv, "malformed"sv, "failed conformance tests"sv
   };
   static uniform_int_distribution die{
      std::size_t{ }, size(reason) - 1
   };
   return std::string{ reason[die(prng)] };
}
```

```cpp
o
i
a
         Diagnostic<Piece>{
            std::move(p), false, reason_to_reject(prng)
         } :
         Diagnostic<Piece>{ std::move(p) };
      }
   }
   re
});
```

```
ofstream out{ "out.txt" };
int ndefects = 0;
auto [valids, dtd] = test([&] {
   vector<Piece> valids;
   auto &rej_analyst = RejectionAnalyst::get();
   for (auto &&p : v) {
      if (auto [piece, ok, reason] = diagnose(std::move(p)); ok) {
         out << piece << '\n';
         valids.push_back(std::move(piece));
      } else {
         ++ndefects;
         out << "DEFECT DETECTED ON " << piece << "; reason: "
             << quoted(reason) << '\n';
          rej_analyst.reject(std::move(piece), std::move(reason));
      }
   }
   return valids;
});
```

```
ofstream out
int ndefect
auto [valid
    vector<P
    auto &re
    for (aut
        if (a
            ou
            va
        } els
            ++
            o
            
            }
        }
    }
    return va
});
```

```
// note : not thread-safe
class RejectionAnalyst {
  std::vector<std::pair<Piece, std::string>> rejects;
  std::map<std::string, int> reasons;
  RejectionAnalyst() = default;
public:
  RejectionAnalyst(const RejectionAnalyst&) = delete;
  RejectionAnalyst&
    operator=(const RejectionAnalyst&) = delete;
  static auto &get() noexcept {
    static RejectionAnalyst singleton;
    return singleton;
  }
  void reject(Piece p, std::string reason);
  std::string statistics() const;
  float avg_quality_of_rejects() const;
};
```

```
ofstre
int n
auto
ret
});
```

```
// note : not thread-safe
class RejectionAnalyst {
   std::vector<std::pair<Piece, std::string>> rejects;
                                                            ete;
      return              n;
   }
   void reject(Piece p, std::string reason);
   std::string statistics() const;
   float avg_quality_of_rejects() const;
};
```

```
void RejectionAnalyst::reject(Piece p, string reason) {
   reasons[rejects.emplace_back(std::move(p),
                             std::move(reason)).second]++;
}
```

Our starting point

```
int main() {
    // - create pieces
    // - sequentially process the pieces,
    //   diagnose them and reject the
    //   defective ones
    // - analyze the cause of rejections and produce
    //   some statistics
    // - compute
    //
    // -
}
```

```
out << RejectionAnalyst::get().statistics() << '\n';
```

Our starting point

```
int mai
    // -
    // -
    //
    //
    // -
    //
    // -
    //
    // -
}
```

```
string RejectionAnalyst::statistics() const {
    stringstream sstr;
    for (auto &&[s, n] : reasons) {
        sstr << "Cause : "sv << quoted(s)
             << ", nb occurrences: "sv << n << "\n"sv;
    }
    return sstr.str();
}
```

```
out << RejectionAnalyst::get().statistics() << '\n';
```

Our starting point

```
int main() {
    // - create pieces
    // - sequentially process the pieces,
    //   diagnose them and reject the
    //   defective ones
    // - analyze the cause of rejections and produce
    //   some statistics
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
    //
    //
    //
    //
    //
    //    some s        es
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
out << "Average quality of non-rejects: "
    << average_quality(valids) << '\n';
out << "Average quality of rejects: "
    << RejectionAnalyst::get().avg_quality_of_rejects() << '\n';
```

Our starting point

```
float average_quality(const std::vector<Piece> &pieces) {
    using namespace std;
    vector<float> qualities;
    qualities.reserve(pieces.size());
    transform(begin(pieces), end(pieces), back_inserter(qualities),
              [](auto &&p) { return p.quality(); });
    return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

```
i
    //    both va          ejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
float average_quality(const std::vector<Piece> &pieces) {
   using namespace std;
   vector<float> qualities;
   qualities.reserve(pieces.size());
   transform(begin(pieces), end(pieces), back_inserter(qualities),
             [](auto &&p) { return p.quality(); });
   return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

```
   //   both                    rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
float average_quality(co
   using namespace std;
   vector<float> qualit
   qualities.reserve(
   transform(begin(piece
             [](auto &&p)
   return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

```
template <class R, class It, class Cumul>
   R average(It b, It e, Cumul init) {
      return static_cast<R>(
         std::accumulate(b, e, init) /
         std::distance(b, e)
      );
}
```

```
   //   both va           rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
```

```
out << "Average quality of non-rejects: "
    << average_quality(valids) << '\n';
out << "Average quality of rejects: "
    << RejectionAnalyst::get().avg_quality_of_rejects() << '\n';
```

```
    //    so      stics
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int main() {
    //
    //
    //
    //
    //
    //    so
    // - co
    //    bo
    // - pe
}
```

```
out << "Average quality of non-rejects: "
    <<
out <<
```

```
float RejectionAnalyst::avg_quality_of_rejects() const {
   using namespace std;
   vector<Piece> pieces;
   pieces.reserve(rejects.size());
   transform(begin(rejects), end(rejects),
             back_inserter(pieces), [](auto &&d) {
      return d.first;
   });
   return average_quality(pieces);
}
```

Our starting point

This is of course the same function we saw a few slides back

```
int main() {
    //
    //
    //
    //
    //
    //    sc
    // - cc
    //    bc
    // - pe
}
```

```
out << "Average quality of non-rejects: "
    <<
out <<
```

```
float RejectionAnalyst::avg_qu   ty_of_rejects() const {
    using namespace std;
    vector<Piece> pieces;
    pieces.reserve(rejects.s  e());
    transform(begin(rejects)  end(rejects),
              back_inserte  (pieces), [](auto &&d) {
        return d.first;
    });
    return average_quality(pieces);
}
```

Our starting point

```
int main() {
   // - create pieces
   // - sequentially process the pieces,
   //   diagnose them and reject the
   //   defective ones
   // - analyze the cause of rejections and produce
   //   some statistics
   // - compute the average rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
int m
    //
    //
    //
    //
    //
    //
    // - compute the average rate of quality for
    //   both valid and rejected pieces
    // - perform an in-depth analysis of valid pieces
}
```

```
auto res = DeepAnalyst::get().analyze(valids);
out << "After deeper analysis of "
    << format_integral(res.nb_elements) << " piece elements:\n";
for (auto &&[s, rep] : res.repartition_type_element) {
   out << "\t" << rep.first << " instances of " << quoted(s)
       << ", average quality " << rep.second << '\n';
}
```

Our starting point

```
auto [res, dt] = test([valids] {
   return DeepAnalyst::get().analyze(valids);
});
out << "After deeper analysis of "
    << format_integral(res.nb_elements) << " piece elements:\n";
for (auto &&[s, rep] : res.repartition_type_element) {
   out << "\t" << rep.first << " instances of " << quoted(s)
       << ", average quality " << rep.second << '\n';
}
```

```
i
   // - comp    erage rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
auto [res, dt] = test([valids] {
   return DeepAnalyst::get().analyze(valids);
});
out << "After deeper analysis of "
    << format_integral(res.nb_elements)
for (auto &&[s, rep] : res.repartition_t
   out << "\t" << rep.first << " instanc
       << ", average quality " << rep.se
}
```

This part is a bit more complex as it involves a thread pool, distributing pieces to analyze into work units to be consumed by worked threads, synchronizing a rendez-vous point to accumulate the partial computation results…

```
i
   // - compu          erage rate of quality for
   //   both valid and rejected pieces
   // - perform an in-depth analysis of valid pieces
}
```

Our starting point

```
class DeepAnalyst::Analyst {
   vector<thread> workers;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   atomic<bool> done{ false };
   optional<function<void()>> get_work() {
      lock_guard _{ mwork };
      if (work.empty()) return { };
      auto f = work.front();
      work.pop_front();
      return f;
   }
   // ...
```

Let’s look at the (simplistic) thread pool implementation used under the covers

Our starting point

```
class DeepAnalyst::Analyst {
   vector<thread> workers;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   atomic<bool> done{ false };
   optional<function<void()>> get_work() {
      lock_guard _{ mwork };
      if (work.empty()) return { };
      auto f = work.front();
      work.pop_front();
      return f;
   }
   // ...
```

Variables **workers**, **work** and **mwork** control the production and consumption of work units, modeled here as void() functions

Our starting point

```
class DeepAnalyst::Analyst {
   vector<thread> workers;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   atomic<bool> done{ false };
   optional<function<void()>> get_work() {
      lock_guard _{ mwork };
      if (work.empty()) return { };
      auto f = work.front();
      work.pop_front();
      return f;
   }
   // ...
```

Variables **cv** and **m** let individual threads of execution wait for a signal that work units have been queued or that it's time to stop, the latter being modeled by the **done** variable

Our starting point

```
class DeepAnalyst::Analyst {
   vector<thread> workers;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   atomic<bool> done{ false };
   optional<function<void()>> get_work() {
      lock_guard _{ mwork };
      if (work.empty()) return { };
      auto f = work.front();
      work.pop_front();
      return f;
   }
   // ...
```

The **get_work()** private member function lets a thread consume a work unit and execute it. The absence of any work unit is represented by an empty (disengaged) optional

Our starting point

```
// ... public: ...
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&] {
          while(!done.load())
             if (auto w = get_work(); w) {
                (*w)();
             } else {
                unique_lock lck{ m };
                cv.wait(lck);
                lck.unlock();
                if(w = get_work(); w)
                   (*w)();
             }
      });
} // ...
```

The thread pool's constructor generates a certain number of worker threads

Our starting point

```
// ... public: ...
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&] {
         while(!done.load())
            if (auto w = get_work(); w) {
               (*w)();
            } else {
               unique_lock lck{ m };
               cv.wait(lck);
               lck.unlock();
               if(w = get_work(); w)
                  (*w)();
            }
      });
} // ...
```

Each of these threads regularly tests for a signal that it’s time to stop execution

Our starting point

```
// ... public: ...
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&] {
         while(!done.load())
            if (auto w = get_work(); w) {
               (*w)();
            } else {
               unique_lock lck{ m };
               cv.wait(lck);
               lck.unlock();
               if(w = get_work(); w)
                  (*w)();
            }
      });
} // ...
```

If a work unit is available, the thread consumes and executes it…

Our starting point

```
// ... public: ...
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&] {
         while(!done.load())
            if (auto w = get_work(); w) {
               (*w)();
            } else {
               unique_lock lck{ m };
               cv.wait(lck);
               lck.unlock();
               if(w = get_work(); w) (*w)();
            }
      });
} // ...
```

…otherwise the thread suspends itself until either a signal to conclude execution is noticed or some work unit is enqueued

Our starting point

```
  // ...
  template <class F> void add_work(F f) {
     lock_guard _{ mwork };
     work.push_back(f);
     cv.notify_one();
   }
  ~Analyst() {
     done = true;
     cv.notify_all();
     for (auto &th : workers) th.join();
  }
};
```

Enqueuing a work unit is done in a synchronized manner and sends a signal that could wake up a worker thread (if such a thread is waiting for work at that moment)

Our starting point

```
    // ...
    template <class F> void add_work(F f) {
        lock_guard _{ mwork };
        work.push_back(f);
        cv.notify_one();
    }
    ~Analyst() {
        done = true;
        cv.notify_all();
        for (auto &th : workers) th.join();
    }
};
```

At the end of the thread pool's lifetime, a stop marker (**done**) is modified and a signal is sent to all worker threads. We then wait for the gracious completion of all enqueued work units

Our starting point

```
auto
  DeepAnalyst::analyze(const vector<Piece> &v) const
    -> Result {
   // - partition the set of pieces to analyze
   // - represent each work unit of a partition
   //   of pieces by function
   // - feed the thread pool
   // - once all of the partitions have been
   //   processed, accumulate the results
}
```

Our starting point

```
auto
  DeepAnalyst::anal
    -> Result {
   // - partition t
   // - represent
   //   of pieces b
   // - feed the thread pool
   // - once all of the partitions have been
   //   processed, accumulate the results
}
```

```
struct Result {
   using size_type = std::size_t;
   size_type nb_elements;
   std::map<std::string,
            std::pair<size_type, float>>
      repartition_type_element;
};
```

Our starting point

```
auto
  DeepAnalyst::analyze(const vector<Piece> &v) const
    -> Result {
   // - partition the set of pieces to analyze
   // - represent each work unit of a partition
   //   of pieces by function
   // - feed the thread pool
   // - once all of the partitions have been
   //   processed, accumulate the results
}
```

```
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   const auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

```
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   const auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

completed counts the number of tasks that have been completed (we are done when **completed == N**). Computation results are stored in **results** and this is synchronized through **m**

Our starting point

```
auto
  DeepAnalyst::analyze(const vector<Piece> &v) const
    -> Result {
   // - partition the set of pieces to analyze
   // - represent each work unit of a partition
   //   of pieces by function
   // - feed the thread pool
   // - once all of the partitions have been
   //   processed, accumulate the results
}
```

```
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   const auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) };
         i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

Vector **v** is split in **N** blocks of approximately **n** elements and each block is processed in parallel (read-only processing, no synchronization involved)

```cpp
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

```
auto DeepAnalyst
    constexpr int
    auto n = size
    mutex m;
    atomic<int> c
    vector<Result
    for (auto [i,
        analyst->a
            // ..
            // .
            // ...
        });
        // ...
    }
    // ...
}
```

```
Result::size_type nelements = 0;
map<string, pair<Result::size_type, float>>
   repartition_type_element;
for (; b != e; ++b) {
   for (auto & elem : *b) {
      auto &[n,qual] =
        repartition_type_element[std::get<string>(elem)];
      ++n;
      qual += std::get<float>(elem);
      ++nelements;
   }
}
```

```
auto Deep
    conste
    auto n
    mutex
    atomic
    vector
    for (a
        ana

        });
        // ..
    }
    // ...
}
```

```
Result::size_type nelements = 0;
map<string, pair<Result::size_type, float>>
   repartition_type_element;
for (; b != e; ++b) {
   for (auto & elem : *b) {
      auto &[n,qual] =
        repartition_type_element[std::get<string>(elem)];
      ++n;
      qual += std::get<float>(elem);
      ++nelements;
   }
}
```

For our (quite summary) analysis, we first compute the sum of the quality of pieces, organized by piece category (note: one should make sure not to overflow, my code is not careful enough here)

```cpp
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

```
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   mutex m;
   atomic<int> complet
   vector<Result> resu
   for (auto [i, p] =
      analyst->add_wor
         // ... perfor
         // ... compute the aver                for each type of element
         // ... signal completion          the work unit
      });
      // ...
   }
   // ...
}
```

```
for (auto &[_, rep] : repartition_type_element) {
   auto &[n, qual] = rep;
   qual /= n;
}
```

```
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

```cpp
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   mutex m;
   atomic<int> completed{ 0 };
   vector<Result> results;
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ... perform the expected computations
         // ... compute the average quality for each type of element
         // ... signal completion of the work unit
      });
      // ...
   }
   // ...
}
```

```cpp
lock_guard _{ m };
results.push_back({ nelements,
                    repartition_type_element });
++completed;
```

Our starting point

```
auto
  DeepAnalyst::analyze(const vector<Piece> &v) const
    -> Result {
   // - partition the set of pieces to analyze
   // - represent each work unit of a partition
   //   of pieces by function
   // - feed the thread pool
   // - once all of the partitions have been
   //   processed, accumulate the results
}
```

Our st

```
auto
  Deep                                                                onst
    ->
   //
   //
   //
   //
}
```

```
// ...
while (completed.load() != N) // bleh
   ;
auto res =
   accumulate(next(begin(results)), end(results),
      results.front(),
      [](Result so_far, Result cur) -> Result {
        return {
          so_far.nb_elements + cur.nb_elements,
          merge(so_far.repartition_type_element,
                cur.repartition_type_element,
                [](auto &&a, auto &&b) {
                   return pair{ a.first + b.first,
                                a.second + b.second };
                }
          )
        };
   });
return res;
```

Our st

auto

Deep ... onst

```
// ...
while (completed.load() != N) // bleh
   ;
auto res =
   accumu   e(next(begin(results)), end(results),
             s.front(),
             o_far, Result cur) -> Result {

             nb_elements + cur.nb_elements,
             o_far.repartition_type_element,
              cur.repartition_type_element,
              [](auto &&a, auto &&b) {
                 return pair{ a.first + b.first,
                              a.second + b.second };
              }
          )
       };
   });
return res;
```

Busy waiting for completion of the Nth computation task (bleh, as noted in the comment)

}

O

au

}

```
// ...
while (completed.load() != N) // bleh
  ;
auto res =
   accumulate(next(begin(results)), end(results),
      results.front(),
      [](Result so_far, Result cur) -> Result {
         return {
            so_far.nb_elements + cur.nb_elements,
            merge(so_far.repartition_type_element,
                  cur.repartition_type_element,
                  [](auto &&a, auto &&b) {
                     return pair{ a.first + b.first,
                                  a.second + b.second };
                  }
            )
         };
   });
return res;
```

Accumulating the results themselves (number of elements and sum of the repartition stats). Once again, this code does not pay enough attention to overflows

```
// ...
while (completed.load() != N) // bleh
  ;
auto res =
  accumulate(next(begin(results)), end(results),
     results.front(),
     [](Result so_far, Result cur) -> Result {
       return {
         so_far.nb_elements + cur.nb_elements,
          merge(so_far.repartition_type_element,
                cur.repartition_type_element,
                [](auto &&a, auto &&b) {
                   return pair{ a.first + b.first,
                                a.second + b.second };
                }
          )
       };
  });
return res;
```

This is not std::merge(), it's a custom merge-and-transform version but it will be the same for both versions of our code

Our starting point

- We now have a complete, working C++17-based system
  - A naïve one, obviously, compared to the kind of industrial system it is meant to illustrate
- Everything works!
- We now have a golden opportunity to migrate to C++20 and, in so doing, revisit our own practices

Our starting point

- We now have a complete, working C++17-based system
    - A naïve one, obviously, compared to the kind of industrial system it is meant to illustrate
- **Everything works!**
- We now have a golden opportunity to migrate to C++20 and, in so doing, revis practices

** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in 40112 ms
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: 46024 ms

An opportunity to rethink our *praxis*…

An opportunity to rethink our *praxis*…

- We now have C++20, and with it an occasion for a fresh look at how we do things

An opportunity to rethink our *praxis*…

- We now have C++20, and with it an occasion for a fresh look at how we do things
- We will make a few adjustments here and there, to benefit more from the new possibilities we are being offered
  - Nothing exhaustive, of course!

An opportunity to rethink our *praxis*…

- We now have C++20, and with it an occasion for a fresh look at how we do things
- We will make a few adjustments here and there, to benefit more from the new possibilities we are being offered
- We will adopt a comparative, « before vs after » approach
  - The « before » part being C++17 and the « after » part being C++20

Computing the absolute value

Computing the absolute value

- We implemented the constexpr computation of the absolute value through compile-time algorithm selection based on a trait

Computing the absolute value

```
// C++17
template <class T>
   constexpr T absolute(T val) {
      if constexpr (std::is_unsigned_v<T>)
         return val;
      else
         return val < 0? -val : val;
   }
```

Computing the absolute value

- We implemented the constexpr computation of the absolute value through compile-time algorithm selection based on a trait
- This is somewhat « manual »
  - if constexpr is well-suited to decision tree-like compile-time selection processes, but that’s not really what we have here

Computing the absolute value

- We implemented the constexpr computation of the absolute value through compile-time algorithm selection based on a trait
- This is somewhat « manual »
- We could be more elegant and specialize our functions based on **concepts**

Computing the absolute value

```
// C++20
#include <concepts>
constexpr auto absolute(std::unsigned_integral auto val) {
   return val;
}
constexpr auto absolute(std::signed_integral auto val) {
   return val < 0 ? -val : val;
}
constexpr auto absolute(std::floating_point auto val) {
   return val < 0 ? -val : val;
}
```

Computing the absolute value

- We had implemented the constexpr computation of the absolute value through compile-time algorithm selection based on a trait
- We could be more elegant and specialize our functions based on **concepts**
- Concepts let us write generic code that looks like « normal » code
  - They also tend to give us much clearer error messages!

Computing the absolute value

- We had implemented the constexpr computation of the absolute value through compile-time algorithm selection based on a trait
- We could be more elegant and specialize our functions based on **concepts**
- Concepts let us write generic code that looks like « normal » code
  - They also tend to give us much clearer error messages!

Concepts will come back regularly in the code adjustments that follow. The significantly change how we approach the act of programming

Computing over ranges

Computing over ranges

- Algorithms that iterate over iterator pairs have proven themselves
  - The C++ standard library is in many ways exemplary in terms of engineering quality

Computing over ranges

- Algorithms that iterate over iterator pairs have proven themselves
- Representing sequences with iterator pairs is highly flexible

Computing over ranges

- Algorithms that iterate over iterator pairs have proven themselves
- Representing sequences with iterator pairs is highly flexible

```
bool is_rejected(const Piece&);
// I used partition() + for_each() but remove_if() would be Ok
vector<Piece*> expunge_rejects(vector<Piece*> v) {
    auto pos = partition(begin(v), end(v), [](const Piece *p) {
        return p && !is_rejected(*p);
    });
    for_each(pos, end(v), [](auto p) { delete p; });
    return { begin(v), pos };
}
```

Computing over ranges

- Algorithms that iterate over iterator pairs have proven themselves
- Representing sequences with iterator pairs is highly flexible
- In practice, of course, we often don't use that flexibility and apply algorithms over entire containers
  - The conceptual move from iterator pairs to *Ranges* eliminates the risk of using incoherent iterators
  - …and opens up new optimization opportunities!

Computing over ranges

```
// C++17
#include <numeric>
#include <iterator>
template <class R, class It, class Cumul>
   R average(It b, It e, Cumul init) {
      return static_cast<R>(
         std::accumulate(b, e, init) /
         std::distance(b, e)
      );
}
```

Computing over ranges

```
// C++20
#include <numeric>
#include <iterator>
#include <ranges>
#include <functional>
template <std::ranges::range Rg, class Cumul,
          class F = std::plus<>>
   auto accumulate(Rg &&rg, Cumul init, F f = {}) {
      for (auto &&val : rg)
         init = f(init, val);
      return init;
   }
// ...
```

Computing over ranges

```
// C++20
// ...
template <class R, std::forward_iterator It,
          class Cumul>
   R average(It b, It e, Cumul init) {
      return static_cast<R>(
         std::accumulate(b, e, init) /
         std::distance(b, e)
      );
}
// ...
```

Computing over ranges

```
// C++20
// ...
template <class R, std::ranges::range Rg>
   R average(Rg && rg, R init = {}) {
      const auto n = std::ranges::size(rg);
      return accumulate(
         std::forward<Rg>(rg), init
      ) / n;
   }
```

Computing over ranges

```
// C++17
float RejectionAnalyst::avg_quality_of_rejects() const {
    using namespace std;
    vector<Piece> pieces;
    pieces.reserve(rejects.size());
    transform(begin(rejects), end(rejects),
              back_inserter(pieces), [](auto &&d) {
        return d.first;
    });
    return average_quality(pieces);
}
```

Computing over ranges

```
float average_quality(const std::vector<Piece> &pieces) {
    using namespace std;
    vector<float> qualities;
    qualities.reserve(pieces.size());
    transform(begin(pieces), end(pieces),
              back_inserter(qualities), [](auto &&p) {
        return p.quality();
    });
    return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

```
// C
floa
    u
    v
    p
    t

    });
    return average_quality(pieces);
}
```

Computing over ranges

```
// C++20
float RejectionAnalyst::avg_quality_of_rejects() const {
   auto quality = [](const Piece &p) {
      return p.quality();
   };
   return average(
       rejects | std::views::keys |
       std::views::transform(quality),
       0.0f
   );
}
```

Computing over ranges

```
// C++20
float RejectionAnalyst::avg_qua
    auto quality = [](const Piec
        return p.quality();
    };
    return average(
        rejects | std::views::keys |
        std::views::transform(quality),
        0.0f
    );
}
```

Notice that, in addition to being shorter to write and simpler, we do not need to go through intermediate containers anymore: evaluations have become lazy and we now avoid both memory allocations and copies

Comparing pieces

Comparing pieces

- Object comparison in C++ is highly configurable
  - One can among other things implement relational operators on an individual basis
  - This can lead to redundant code, however

Comparing pieces

```
// C++17
class Piece {
   // ...
   bool operator==(const Piece &other) const noexcept {
      return id() == other.id() && elements == other.elements;
   }
   bool operator!=(const Piece &other) const noexcept { return !(*this == other); }
   bool operator<(const Piece &other) const noexcept {
      return id() < other.id() ||
             id() == other.id() &&
                std::lexicographical_compare(begin(), end(), other.begin(), other.end());
   }
   bool operator>(const Piece &other) const noexcept { return other < *this; }
   bool operator<=(const Piece &other) const noexcept { return !(other < *this); }
   bool operator>=(const Piece &other) const noexcept { return !(*this < other); }
};
```

Comparing pieces

```
// C++20 (step by step)
class Piece {
   // ...
   bool operator==(const Piece &other) const noexcept {
      return id() == other.id() && elements == other.elements;
   }
   bool operator!=(const Piece &other) const noexcept { return !(*this == other); }
   bool operator<(const Piece &other) const noexcept {
      return id() < other.id() ||
             id() == other.id() &&
                std::lexicographical_compare(begin(), end(), other.begin(), other.end());
   }
   bool operator>(const Piece &other) const noexcept { return other < *this; }
   bool operator<=(const Piece &other) const noexcept { return !(other < *this); }
   bool operator>=(const Piece &other) const noexcept { return !(*this < other); }
};
```

Comparing pieces

```
// C++20 (step by step)
class Piece {
    // ... operator!= synthesized from operator==
   bool operator==(const Piece &other) const noexcept {
      return id() == other.id() && elements == other.elements;
   }
   bool operator<(const Piece &other) const noexcept {
      return id() < other.id() ||
             id() == other.id() &&
                std::lexicographical_compare(begin(), end(), other.begin(), other.end());
   }
   bool operator>(const Piece &other) const noexcept { return other < *this; }
   bool operator<=(const Piece &other) const noexcept { return !(other < *this); }
   bool operator>=(const Piece &other) const noexcept { return !(*this < other); }
};
```

Comparing pieces

```
// C++20 (step by step)
class Piece {
   // ... operator!= synthesized from operator==
   bool operator==(const Piece &other) const noexcept {
      return id() == other.id() && elements == other.elements;
   }
   bool operator<(const Piece &other) const noexcept {
      return id() < other.id() ||
             id() == other.id() &&
                std::lexicographical_compare(begin(), end(), other.begin(), other.end());
   }
   bool operator>(const Piece &other) const noexcept { return other < *this; }
   bool operator<=(const Piece &other) const noexcept { return !(other < *this); }
   bool operator>=(const Piece &other) const noexcept { return !(*this < other); }
};
```

Comparing pieces

```
// C++20 (step by step)
class Piece {
   // ... all six relational operators synthesized
   // from a single one: operator<=>
   // (« loose » but efficient implementation!)
   auto operator<=>(const Piece &other) const noexcept {
      return std::tuple{ id(), elements }
                <=>
             std::tuple{ other.id(), other.elements };
   }
};
```

Comparing pieces

- The spaceship operator (**operator<=>**) is a three-way comparison operator
  - If well implemented, it can synthesize all relational operators at once
  - The default implementation is sometimes trivial
  - When it is not trivial, e.g.: when comparing floating point numbers or other types for which comparisons require some measure of carefulness, there are some rules to follow

Comparing pieces

- Sometimes, we have a better algorithm for **operator==** than the one we would write for the other relational operators
  - Think of vector<T>::operator== for example
- In such a case, one simply implements **operator==** and **operator<=>**
  - **operator!=** will then be synthesized from **operator==**
  - The inequality operators will be synthesized from **operator<=>**

Function interfaces

Function interfaces

- A good software engineering practice is to define function contracts in the weakest possible way that satisfies our requirements
  - This makes functions applicable to a wider array of types

Function interfaces

```
// C++17
#include <vector>
class Piece {
public:
   using id_type = int;
   using elem_type = std::tuple<std::string, int, float>;
private:
   id_type id_;
   std::vector<elem_type> elements;
   // ...
public:
    Piece(id_type id, const std::vector<elem_type> &elems)
      : id_{ id }, elements{ elems }, quality_{ evaluate_quality(elems) } {
   }
   // ...
};
```

Function interfaces

```cpp
// C++17
#include <vector>
class Piece {
public:
   using id_type = int;
   using elem_type = std::tuple<std::string, int, float>;
private:
   id_type id_;
   std::vector<elem_type> elements;
   // ...
public:
    Piece(id_type id, const std::vector<elem_type> &elems)
      : id_{ id }, elements{ elems }, quality_{ evaluate_quality(elems) } {
   }
   // ...
};
```

We require here argument **elems** to be a vector but that’s an artificial constraint, i.e.: we could have accepted arrays and it would do no harm

Function interfaces

```
// C++20
#include <span>
class Piece {
public:
   using id_type = int;
   using elem_type = std::tuple<std::string, int, float>;
private:
   id_type id_;
   std::vector<elem_type> elements; // does not change
   // ...
public:
    Piece(id_type id, std::span<const elem_type> elems)
      : id_{ id }, elements{ elems }, quality_{ evaluate_quality(elems) } {
   }
   // ...
};
```

Function interfaces

```
// C++20
#include <span>
class Piece {
public:
   using id_type = int;
   using elem_type = std::tuple<std::string, int, float>;
private:
   id_type id_;
   std::vector<elem_type> elements; // does not change
   // ...
public:
    Piece(id_type id, std::span<const elem_type> elems)
      : id_{ id }, elements{ elems }, quality_{ evaluate_quality(elems) } {
   }
   // ...
};
```

We now require argument **elems** to be contiguous-in-memory sequence of objects locally considered **const**

Function interfaces

```
// C++20
#include <span>
class Piece {
public:
   using id_type = int;
   using elem_type = std::tuple<std::string, int, float>;
private:
   id_type id_;
   std::vector<elem_type> elements; // does not change
   // ...
public:
    Piece(id_type id, std::span<const elem_type> elems)
      : id_{ id }, elements{ elems }, quality_{ evaluate_quality(elems) } {
   }
   // ...
};
```

Note that we went from a *ref-to-**const*** argument to a *pass-by-value* argument: *copying a **span** is essentially free*, and **span** behaves as a reference which means **const** now applies not to the **span** but rather to the elements to which it refers

Function interfaces

```
// C++17
#include <vector>
class Piece {
   // ...
   static float evaluate_quality(const std::vector<elem_type> &v) {
      using namespace std;
      return accumulate(v.begin(), v.end(), 0.f,
                        [](float cur, auto &&tup) {
         return max(cur, get<float>(tup));
      });
   }
   // ...
};
```

Function interfaces

```
// C++20
#include <span>
class Piece {
   // ...
   static float evaluate_quality(std::span<const elem_type> v) {
      using namespace std;
      return accumulate(v.begin(), v.end(), 0.f,
                        [](float cur, auto &&tup) {
         return max(cur, get<float>(tup));
      });
   }
   // ...
};
```

Function interfaces

```
// C++17
#include "Piece.h"
#include <vector>
class DeepAnalyst {
   // ...
   struct Result {
      // ...
   };
   // ...
   Result analyze(const std::vector<Piece>&) const;
};
```

Function interfaces

```
// C++20
#include "Piece.h"
#include <span>
class DeepAnalyst {
   // ...
   struct Result {
      // ...
   };
   // ...
   Result analyze(std::span<const Piece>) const;
};
```

Function interfaces

```
// C++17
float average_quality(const std::vector<Piece> &pieces) {
   using namespace std;
   vector<float> qualities;
   qualities.reserve(pieces.size());
   transform(begin(pieces), end(pieces),
             back_inserter(qualities),
             [](auto &&p) {
                return p.quality();
   });
   return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

Function interfaces

```
// C++20 (step by step)
float average_quality(std::span<const Piece> pieces) {
   using namespace std;
   vector<float> qualities;
   qualities.reserve(pieces.size());
   transform(begin(pieces), end(pieces),
            back_inserter(qualities),
            [](auto &&p) {
               return p.quality();
   });
   return average<float>(begin(qualities), end(qualities), 0.0f);
}
```

Function interfaces

```
// C++20 (step by step)
#include <span>
#include <ranges>
float average_quality(std::span<const Piece> pieces) {
  auto qualities =
    std::ranges::views::transform(pieces,
      [](auto &&p) {
        return p.quality();
    });
  return average(qualities, 0.0f);
}
```

Function interfaces

Here, **qualities** is not a vector; it's a view over the original data that yields a transformation of that data when observed, and the average will be computed over the transformation of those values

```
// C++20 (step by step)
#include <span>
#include <ranges>
float average_quality(std::span<const Piece> pieces) {
  auto qualities =
    std::ranges::views::transform(pieces,
      [](auto &&p) {
        return p.quality();
    });
  return average(qualities, 0.0f);
}
```

Function interfaces

- Type **span<T>** is a (begin, size) pair
  - It's very lightweight
  - Analogous to C++17's **string_view** but allows mutation of the referred objects
    - If they are non-const, of course
  - Its interface is safer than that of raw arrays

Text formatting

Text formatting

```
// C++17
// ...
template <class T>
   std::string format_integral(T val) {
      static_assert(std::is_integral_v<T>);
      auto sign_str = val < 0? "-"s : ""s;
      val = absolute(val);
      std::string s;
      while (val >= 1000) {
         s = "'"s + pad_left(std::to_string(val % 1000), '0', 3) + s;
         val /= 1000;
      }
      return sign_str + std::to_string(val) + s;
}
```

Text formatting

```
// C++17
// ...
template <class T>
  std::string format_integra
    static_assert(std::is_i
    auto sign_str = val
    val = absolute(val);
    std::string s;
    while (val >= 1000) {
      s = "'"s + pad_left(std::to_string(val % 1000), '0', 3) + s;
      val /= 1000;
    }
    return sign_str + std::to_string(val) + s;
}
```

```
assert(format_integral(0) == "0"s);
assert(format_integral(12) == "12"s);
assert(format_integral(-12) == "-12"s);
assert(format_integral(12345) == "12'345"s);
assert(
  format_integral(10000000) == "10'000'000"s
);
```

Text formatting

```
// C++20
// ...
#include <concepts>
#include <format>
std::string format_integral(std::integral auto val) {
   auto sign_str = val < 0? "-"s : ""s;
   val = absolute(val);
   std::string s;
   while (val >= 1000) {
      s = std::format("'{:0>3}", val % 1000) + s;
      val /= 1000;
   }
   return sign_str + std::to_string(val) + s;
}
```

Text formatting

```
// C++20
// ...
#include <concepts>
#include <format>
std::string format_integral(s
   auto sign_str = val <
   val = absolute(val);
   std::string s;
   while (val >= 1000) {
      s = std::format("'{:0>3}", val % 1000) + s;
      val /= 1000;
   }
   return sign_str + std::to_string(val) + s;
}
```

```
assert(format_integral(0) == "0"s);
assert(format_integral(12) == "12"s);
assert(format_integral(-12) == "-12"s);
assert(format_integral(12345) == "12'345"s);
assert(
   format_integral(10000000) == "10'000'000"s
);
```

Text formatting

```
// C++20
// ...
#include <concepts>
#include <format>
std::string format_integral(std::integral auto
   auto sign_str = val < 0? "-"s : ""s;
   val = absolute(val);
   std::string s;
   while (val >= 1000) {
      s = std::format("'{:0>3}", val % 1000) + s;
      val /= 1000;
   }
   return sign_str + std::to_string(val) + s;
}
```

format() provides us with efficient character string formatting. Here, we right-align numbers on a width of 3 and fill the left side with **'0'**

Text formatting

```
// C++20 (alternative approach)
// ...
#include <concepts>
#include <format>
template <std::integral T>
   struct formatable_integer {
      T val{};
      formatable_integer() = default;
      constexpr formatable_integer(T val) : val{ val } {
      }
      auto operator<=>(const formatable_integer &) const = default;
   };
// ...
```

Text formatting

```
// C++20 (alternative approach)
// ...
template <std::integral T> struct std::formatter<formatable_integer<T>> {
  template <class FmtCtx>
    auto format(const formatable_integer<T> &n, FmtCtx &ctx) const {
      auto val = n;
      if (val < 0) format_to(ctx.out(), "{}", '-');
      val = absolute(val.val);
      if (val >= 1000) {
        auto s = std::format("{}", formatable_integer{ val.val / 1000 });
        return format_to(ctx.out(), "{}'{:0>3}", s, val.val % 1000);
      }
      return format_to(ctx.out(), "{}", val.val);
    }
    constexpr auto parse(format_parse_context &ctx) const { return std::begin(ctx); }
  };
```

Text formatting

```
// C++20 (alter
// ...
template <std:
   template <c
      auto for
         auto
         if
         va
         i  (va
            auto s = std::format("{}", formatable_integer{ val.val / 1000 });
            return format_to(ctx.out(), "{}'{:0>3}", s, val.val % 1000);
         }
         return format_to(ctx.out(), "{}", val.val);
      }
      constexpr auto parse(format_parse_context &ctx) const { return std::begin(ctx); }
   };
```

```
assert(format("{}", formatable_integer{ 0 }) == "0"s);
assert(format("{}", formatable_integer{ 12 }) == "12"s);
assert(format("{}", formatable_integer{ -12 }) == "-12"s);
assert(format("{}", formatable_integer{ 12345 }) ==
          "12'345"s);
assert(format("{}", formatable_integer{ 10000000 }) ==
          "10'000'000"s);
```

Text formatting

```
// C++20 (alternative approach)
// ...
template <std::integral T> struct std::formatter<formatable_integer<T>> {
   template <class FmtCtx>
      auto format(const formatable_integer<T> &n, FmtCtx &ctx) const {
         auto val = n;
         if (val < 0) format_to(ctx.out(), "{}", '-');
         val = absolute(val.val);
         if (val >= 1000) {
            auto s = std::format("{}", forma
            return format_to(ctx.out(), "{}'{
         }
         return format_to(ctx.out(), "{}", va
      }
      constexpr auto parse(format_parse_context &ctx) const { return std::begin(ctx); }
   };
```

format() is highly extensible and can be adapted to our types and needs, as can be seen here. This example implements a **formatter<T>** for our **formatable_integer** type which lets us apply **format()** to that type as easily as to fundamental types

Text formatting

```
// C++17
string
  combine_name(int id, const string &name) {
    return name + " "s + to_string(id);
  }
```

Text formatting

I used a const string& argument instead of a string_view because my implementation uses operator+ on string objects

```
// C++17
string
  combine_name(int id, const string &name) {
    return name + " "s + to_string(id);
  }
```

Text formatting

```
// C++20
string
  combine_name(int id, string_view name) {
    return format("{} {}", name, id);
  }
```

Text formatting

```
// C++17
cout << "Deeper analysis of "
     << format_integral(res.nb_elements)
     << " piece elements in "
     << duration_cast<milliseconds>(dt).count()
     << " ms\n";
```

Text formatting

```
// C++20
cout << format(
  "Deeper analysis of {} piece elements in {}\n",
  formatable_integer{ res.nb_elements },
  duration_cast<milliseconds>(dt)
);
```

Text formatting

```
// C++20
cout << format(
  "Deeper analysis of {} piece elements in {}\n",
  formatable_integer{ res.nb_elements },
  duration_cast<milliseconds>(dt)
);
```

Note: with C++20 it is much simpler to serialize types from <chrono> than it used to be

Multithreading

Multithreading

```
// C++17
// include <thread>, <vector>, <deque>, <mutex>,
// <atomic>, <functional>, <condition_variable>
class DeepAnalyst::Analyst {
   vector<thread> workers;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   atomic<bool> done{ false };
   // ...
```

Multithreading

```
// C++17
// include <thread>, <vecto
// <atomic>, <functional>,
class DeepAnalyst::Analyst {
    vector<thread> workers;
    deque<function<void()>> work;
    condition_variable cv;
    mutex m, mwork;
    atomic<bool> done{ false };
    // ...
```

Type **std::thread** requires us to either **join()** or **detach()** on each thread object before reaching its destructor, which requires some measure of care. It is not accompanied by a standardized cooperative stopping mechanism, which explains our **done** variable

Multithreading

```
// C++20
// include <thread>, <vector>, <deque>, <mutex>,
// <functional>, <condition_variable>
class DeepAnalyst::Analyst {
   vector<jthread> workers;
   stop_source source;
   deque<function<void()>> work;
   condition_variable cv;
   mutex m, mwork;
   // ...
```

Multithreading

```
// C++20
// include <thread>, <vecto
// <functional>, <condition
class DeepAnalyst::Analyst {
    vector<jthread> workers;
    stop_source source;
    deque<function<void()>> work;
    condition_variable cv;
    mutex m, mwork;
    // ...
```

Type **std::jthread** (*Joinable Thread*) does an implicit **join()** when its destructor is joined, which is the typical use case. It is accompanied by a standard stop signal mechanism, a **stop_token** that emerges from a **stop_source**

Multithreading

```
// C++17 (continued)
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&] {
         while(!done.load()) {
            if (auto w = get_work(); w) {
               (*w)();
            } else {
               unique_lock lck{ m }; cv.wait(lck); lck.unlock();
               if(w = get_work(); w)
                  (*w)();
            }
         }
      });
   }
```

Multithreading

```
// C++20 (continued)
Analyst(int nb_workers) {
   for(int i = 0; i != nb_workers; ++i)
      workers.emplace_back([&, token = source.get_token()] {
         while(!token.stop_requested()) {
            if (auto w = get_work(); w) {
               (*w)();
            } else {
               unique_lock lck{ m }; cv.wait(lck); lck.unlock();
               if(w = get_work(); w)
                  (*w)();
            }
         }
      });
}
```

Multithreading

```
// C++17 (continued)
~Analyst() {
    done = true;
    cv.notify_all();
    for (auto &th : workers) th.join();
}
```

Multithreading

```
// C++20 (continued)
~Analyst() {
    source.request_stop();
    cv.notify_all();
} // implicit join()
```

Multithreading

```
// C++17 (continued)
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   atomic<int> completed{ 0 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ...
         ++completed;
      });
   }
   while (completed.load() != N) // bleh
      ;
   // ... accumulate results
}
```

Multithreading

```
// C++17 (continued)
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   atomic<int> completed{ 0 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v) }; i
      analyst->add_work([&, b = p, e = next(p
         // ...
         ++completed;
      });
   }
   while (completed.load() != N) // bleh
      ;
   // ... accumulate results
}
```

```
// in more detail
lock_guard _{ m };
results.emplace_back(...);
++completed;
```

Multithreading

```
// C++17 (continued)
auto DeepAnalyst::analyze(const vector<Piece> &v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   atomic<int> completed{ 0 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v) }; i != N
      analyst->add_work([&, b = p, e = next(p, n
         // ...
         ++completed;
      });
   }
   while (completed.load() != N) // bleh
      ;
   // ... accumulate results
}
```

What we are implementing here « manually » is a rendez-vous point for **N** threads of execution. We could be more efficient, but it works in practice

Multithreading

```
// C++20 (continued), include <latch>
auto DeepAnalyst::analyze(span <const Piece> v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   latch ltch{ N + 1 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v) }; i != N; ++i, p = next(p, n)) {
      analyst->add_work([&, b = p, e = next(p, n)]() mutable {
         // ...
         ltch.arrive_and_wait();
      });
   }
   ltch.arrive_and_wait();
   // ... accumulate results
}
```

Multithreading

```
// C++20 (continued), include <latch>
auto DeepAnalyst::analyze(span <const Piece> v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   latch ltch{ N + 1 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v)
      analyst->add_work([&, b = p,
         // ...
         ltch.arrive_and_wait();
      });
   }
   ltch.arrive_and_wait();
   // ... accumulate results
}
```

```
// in more detail (careful!)
{
   lock_guard _{ m };
   results.emplace_back(...);
}
ltch.arrive_and_wait();
```

Multithreading

```
// C++20 (continued), include <latch>
auto DeepAnalyst::analyze(span <const Piece> v) const -> Result {
   constexpr int N = 10; // arbitrary
   auto n = size(v) / N;
   latch ltch{ N + 1 };
   // ...
   for (auto [i, p] = pair{ 0, begin(v) };
      analyst->add_work([&, b = p, e = nex
         // ...
         ltch.arrive_and_wait();
      });
   }
   ltch.arrive_and_wait();
   // ... accumulate results
}
```

A **std::latch** is a single use rendez-vous point for **N** threads of execution. Here, the **N+1** is to ensure that the code that launched the threads participates in the waiting process and can take charge thereafter

Generating values

Generating values

```
// C++17
vector<Piece> create_pieces(size_t n, size_t m) {
   vector<Piece> v;
   v.reserve(n);
   mt19937 prng{ random_device{}() };
   uniform_int_distribution<size_t> die(1, m);
   uniform_real_distribution<float> prob{ 0.975f, 1.0f };
   for (size_t i = 0; i != n; ++i)
      v.emplace_back(
         create_piece(Piece::id_type(i), prng, die, prob)
      );
   return v;
}
```

Generating values

```
// C++17
vector<Piece> create_pieces(s
    vector<Piece> v;
    v.reserve(n);
    mt19937 prng{ rand
    uniform_int_distribution<s
    uniform_real_distribution<float> prob{ 0.975f, 1.0f };
    for (size_t i = 0; i != n; ++i)
        v.emplace_back(
            create_piece(Piece::id_type(i), prng, die, prob)
        );
    return v;
}
```

```
auto [v, dtc] = test([] {
    return create_pieces(npieces, nelems);
});
```

Generating values

```
// C++20
generator<Piece> create_pieces(size_t n, size_t m) {
   mt19937 prng{ random_device{}() };
   uniform_int_distribution<size_t> die(1, m);
   uniform_real_distribution<float> prob{ 0.975f, 1.0f };
   for (size_t i = 0; i != n; ++i) {
      co_yield create_piece(
         Piece::id_type(i), prng, die, prob
      );
   }
   co_return;
}
```

Generating values

```
// C++20
generator
    mt1993
    unifor
    unifor
    for (s
        co_yield cr
            Piec    ::id_type(i), prng, die, prob
        );
    }
    co_return;
}
```

```
// naïve call
auto [v, dtc] = test([] {
    vector<Piece> v;
    for (auto &&piece : create_pieces(npieces, nelems))
        v.push_back(piece);
    return v;
});
```

Generating values

```
// C++20
generator
    mt1993
    unifor
    unifor
    for (s
        co_yield cr
            Piec   ::id_type(i), p
        );
    }
    co_return;
}
```

```
// naïve call
auto [v, dtc] = test([] {
    vector<Piece> v;
    for (auto &&piece : create_pieces(npieces, nelems))
        v.push_back(piece);
    return v;
});
```

For better results, instead of placing the values in a vector, we would have done **co_await** on **create_pieces()** and continued processing asynchronously from that point on

Generating values

- Coroutines are an extremely powerful mechanism
  - Asynchronous functions that can be suspended on an execution thread and resume on another
  - Can in many cases actually replace explicit threading, mechanisms that lead to blocking or transient variables with lightweight equivalents

Generating values

- Coroutines are an extremely powerful mechanism
- Sadly, they have been adoped late in the C++20 standardization process and do not come with significant support from the standard library
  - We write code that works, but that requires from us some measure of boilerplate even for the common cases

Generating values

- Coroutines are an extremely powerful mechanism
- Sadly, they have been adoped late in the C++20 standardization process and do not come with significant support from the standard library
  - We write code that works, but that requires from us some measure of boilerplate even for the common cases

For the purpose of this presentation I wrote a small data generator that behaves like a range and lets us iterate over values: https://wandbox.org/permlink/uiBpO1ZSMHigQU9F but it’s the kind of code the standard library should have

Generating values

- Coroutines are an extreme
- Sadly, they have been ado process and do not com library
  - We write code that works, but that requires from us some measure of boilerplate even for the common cases

Note that I used coroutines here, **but the code does not make them shine: with this program they are slower than the synchronous alternative**. If the code actually did I/O and used the time during which the I/O is being performed to do something else, then it would have an opportunity to shine

… and much, *much* more

… and much, *much* more

- We could have shown so much more…
  - Immediate functions – **consteval**
  - Important refinements to **constexpr**
  - Guaranteed immediate initialization – **constinit**
  - Standard date type in <chrono>
  - **std::is_constant_evaluated()** to choose between a runtime or a **constexpr** algorithm when compiling code
  - etc.
- C++20 is gigantic
  - Larger even than C++11

Is it worth it?

Is it worth it?

- C++17

** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in 40112 ms
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: 46024 ms

Is it worth it?

- C++17

```
** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in 40112 ms
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: 46024 ms
```

- C++20

```
** C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 11329ms
Sequential diagnostics... completed in 14453ms
Deeper analysis of 39'552'676 piece elements in 642ms
Elapsed time for the entire process: 26424ms
```

C++17 is faster here...

ieces with [1,7] elements each... completed in **5346 ms**
completed in 40112 ms
4'771 piece elements in 565 ms
re process: 46024 ms

- C++20

** C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in **11329ms**
Sequential diagnostics... completed in 14453ms
Deeper analysis of 39'552'676 piece elements in 642ms
Elapsed time for the entire process: 26424ms

C++17 is faster here… but it's because I used coroutines in my C++20 code for a use-case that is not really suitable. One can suppose that the comparison would be more fair if it read the pieces from a file rather than generating them randomly at run time

ieces with [1,7] elements each... completed in **5346 ms**
completed in 40112 ms
4'771 piece elements in 565 ms
re process: 46024 ms

- C++20

** C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in **11329ms**
Sequential diagnostics... completed in 14453ms
Deeper analysis of 39'552'676 piece elements in 642ms
Elapsed time for the entire process: 26424ms

Is it worth it?

- C++17

** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in **40112 ms**
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: 46024 ms

with [1,7] elements each... completed in 11329ms
s... completed in **14453ms**
'552'676 piece elements in 642ms
entire process: 26424ms

For the actual processing of our data, on the other hand, we can see a significant advantage. This is mostly due to ranges which allowed us to avoid a number of copies and memory allocations in the C++20 version

Is it worth it?

- C++17

** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in 40112 ms
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: 46024 ms

Randomness plays a part here, but it is not significant in the overall picture

0
of 10'000'000 pieces with [1,7] elements each... completed in 11329ms
al diagnostics... completed in 14453ms
r analysis of 39'552'676 piece elements in 642ms
time for the entire process: 26424ms

Is it worth it?

- C++17

** Pre-C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 5346 ms
Sequential diagnostics... completed in 40112 ms
Deeper analysis of 39'574'771 piece elements in 565 ms
Elapsed time for the entire process: **46024** ms

- C++20

** C++20
Creation of 10'000'000 pieces with [1,7] elements each... completed in 11329ms
Sequential diagnostics... completed in 14453ms
Deeper analysis of 39'552'676 piece elements in 642ms
Elapsed time for the entire process: **26424ms**

Is it worth it?

- The biggest advantage we got from migrating to C++20 was not even execution speed…
- … it was the way code carried clarity of intent
  - Code carries a message
  - C++20 raises the level of abstraction
    - Ranges, concepts, operator<=>, format(), latch, etc.

Is it worth it?

- Is it worth it? I think it clearly does
  - Everything that makes our thoughts and practices evolve is valuable
  - I hope you will enjoy C++20…

Is it worth it?

- Is it worth it? I think it clearly does
  - Everything that makes our thoughts and practices evolve is valuable
  - I hope you will enjoy C++20…
  - Write to me if you find pearls or have particular insights to share!

Merci!

Patrice.Roy@USherbrooke.ca

Patrice.Roy@clg.qc.ca