Modernizing your code base with C++20

High Tech Institute // Webinar // C++ fundamentals

Kris van Rens

```
Kris van(Rens<>); // High Tech Institute.
```

What's ahead?

- Overview of C++20
- Compile-time validation with concepts
- Improving algorithms using ranges

A little bit about me

C++ Fundamentals course

kris@vanrens.org

Overview of C++20

Most notable features

- **Concepts**
- **Modules**
- **Coroutines**
- **Ranges library**

- Three-way comparison ‘`operator <=>`’
- Array views with `std::span`
- Advanced text formatting library `<format>`
- New thread support primitives and `jthread`
- Calendars and time zones in `<chrono>`
- …

Designated initializers

```
1  struct SystemId {
2    Uuid        uuid;
3    std::string name;
4    std::string alias;
5    bool        verified{};
6  };
7
8  void verify(SystemId&& id);
```

Pre-C++20

```
1  verify({"465ff086-92c8-43a9-854e-f41f13bac804",
2          "Quantum vector shape interceptor",
3          "QVSI",
4          true});
```

C++20

```
1  verify({.uuid     = "465ff086-92c8-43a9-854e-f41f13bac804",
2          .name     = "Quantum vector shape interceptor",
3          .alias    = "QVSI",
4          .verified = true});
```

Omitted fields are zero-initialized.

Initialization order must be correct.

(Code @ Compiler Explorer)

Initialization in range-based `for`

Pre-C++20

```cpp
auto data = receive(); // Outside the scope of the loop..
for (auto& chunk : data.chunks) {
  if (!error_correct(chunk)) {
    log_error("Failed to correct chunk #{}", chunk.id);
  }
}
```

```cpp
std::size_t index = 0; // Outside the scope of the loop..
for (const auto& e : collection) {
  register_element(e, index);
  ++index;
}
```

C++20

```cpp
for (auto data = receive(); auto& chunk : data.chunks) {
  if (!error_correct(chunk)) {
    log_error("Failed to correct chunk #{}", chunk.id);
  }
}
```

```cpp
for (std::size_t index = 0; const auto& e : collection) {
  register_element(e, index);
  ++index;
}
```

(Code @ Compiler Explorer)

Method `contains` for STL containers

Pre-C++20

```
1  PartId lookup(std::string_view name) {
2    static CacheMap cache;
3
4    auto result = cache.find(name);
5    if (result == cache.end()) {
6      auto id = lookup_expensive(name);
7      cache[name] = id;
8      return id;
9    }
10
11   return result->second;
12 }
```

C++20

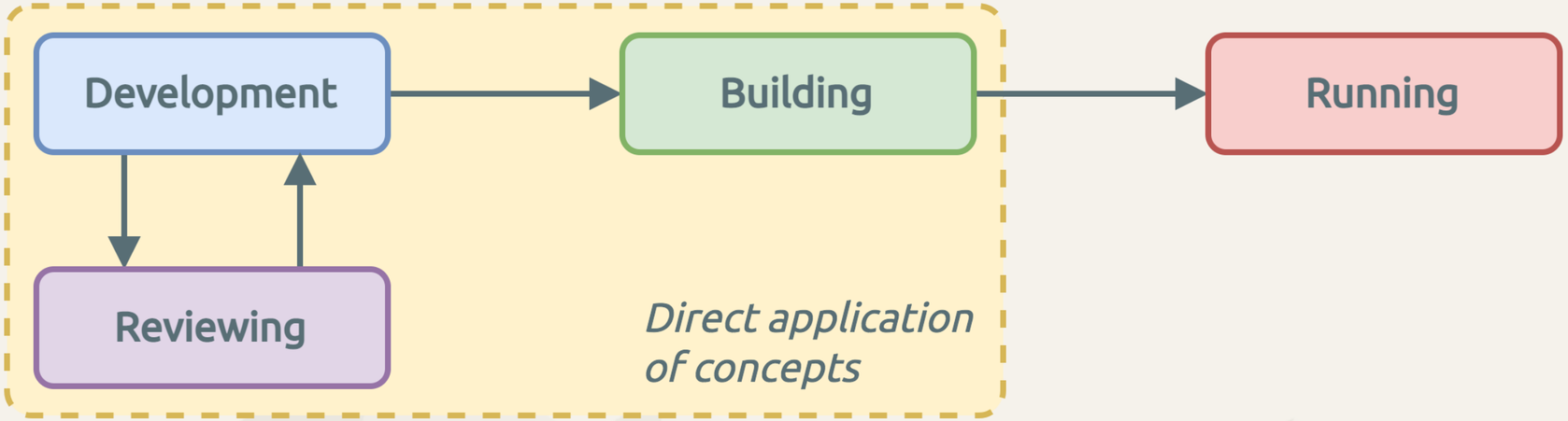
```
1  PartId lookup(std::string_view name) {
2    static CacheMap cache;
3
4    if (!cache.contains(name)) {
5      auto id = lookup_expensive(name);
6      cache[name] = id;
7      return id;
8    }
9
10   return cache[name];
11 }
```

(Code @ Compiler Explorer)

Compile-time validation with concepts

What concepts are about..

- Concepts are **named constraints**, predicates evaluated at compile-time,
- Concepts are **part of template interfaces** to improve *equational reasoning*,
- Concepts can be used to impose **syntactic and semantic constraints**.



Imposing type constraints

Use case: constrain class template type parameter.

```
SensorReading<float> r1; // OK.
SensorReading<int>   r2; // Compiler error.
```

Pre-C++20

```
1  #include <type_traits>
2
3  template<typename T>
4  class SensorReading {
5    static_assert(std::is_floating_point_v<T>,
6                  "T must be a floating point type");
7  public:
8    // ...
9  };
```

C++20

```
1  #include <concepts>
2
3  template<std::floating_point T>
4  class SensorReading {
5  public:
6    // ...
7  };
```

(Code @ Compiler Explorer)

Constraining (member-)functions

Use case: conditionally enable member function.

```
SensorReading<double> r1; // Has 'guess_snr'.
SensorReading<float>  r2; // Has no 'guess_snr'.

double snr1 = r1.guess_snr(); // OK.
double snr2 = r2.guess_snr(); // Compiler error!
```

Pre-C++20

```
1  #include <type_traits>
2
3  template<typename T>
4  class SensorReading {
5  public:
6    template<typename U = T,
7             typename = std::enable_if_t<std::is_same_v<double, U>>>
8    double guess_snr() const;
9  };
```

C++20

```
1  #include <concepts>
2
3  template<std::floating_point T>
4  class SensorReading {
5  public:
6    double guess_snr() const requires std::same_as<double, T>;
7  };
```

(Code @ Compiler Explorer)

Defining custom concepts

Example: concept to test for type equivalence.

```cpp
class Sensor {
  // ...
};

class SpecificSensor : public Sensor {
  // ...
};

class MeasurementSystem {
public:
  operator Sensor() const { // Type conversion function.
    // ...
  }
};
```

All types can be used as a ‘Sensor’

```cpp
#include <concepts>

template<typename T>
concept sensor_like = std::same_as<Sensor, T>
                    || std::derived_from<T, Sensor>
                    || std::convertible_to<T, Sensor>;

template<sensor_like S>
void process(const S& s);
```

```cpp
Sensor           s1; // std::same_as        --> true.
SpecificSensor   s2; // std::derived_from   --> true.
MeasurementSystem s3; // std::convertible_to --> true.

process(s1);
process(s2);
process(s3);
```

(Code @ Compiler Explorer)

`requires` expressions

Requires expressions can be used to **easily define powerful constraints**.

Testing for `get()`

```
1  template<typename T>
2  concept has_get = requires (T t) { t.get(); };
3  //                ~~~~~~~~~~~~~~~~~~~~~~~~~ Requires expression.
4
5  template<typename Input> requires has_get<Input>
6  void process(const Input& i) {
7    const auto value{i.get()};
8    // ...
9  }
```

```
1  class Sensor {
2  public:
3    int get() const;
4  };
5
6  class SomeType {
7  public:
8    // No 'get' function here..
9  };
```

```
1  Sensor   x1;
2  SomeType x2;
3
4  process(x1); // OK.
5  process(x2); // Compiler error.
```

(Code @ Compiler Explorer)

`requires` expressions

Example: arithmetic key/value mapping type definition.

```cpp
template<typename T>
concept key_value_store = requires (T x) {
  { x.get() };
  { x.get() } noexcept;
  { x.get() } -> std::same_as<std::pair<int, std::string>>;
  { x + x   } -> std::same_as<T>;
} && std::equality_comparable<T>;

template<key_value_store Store>
void process(const Store& s) {
  const auto& [key, value] = s.get();

  // ...
}
```

```cpp
class Mapping {
public:
  using KeyValue = std::pair<int, std::string>;

  [[nodiscard]] KeyValue get() const noexcept;

  Mapping operator+(const Mapping&);

  bool operator<=>(const Mapping&) const = default;
};
```

```cpp
Mapping m1, m2;

process(m1 + m2);
```

(Code @ Compiler Explorer)

Improving algorithms using ranges

Ranges

A “range” is an **abstraction over iterators**, enabling a *functional style*.

Classic algorithms

```
1  std::vector<int> vec{60, 4, 84, 14, 79, 51, 93, 25, 59};
2
3  std::sort(vec.begin(), vec.end());
```

We supply a pair of iterators.

C++20 ranges

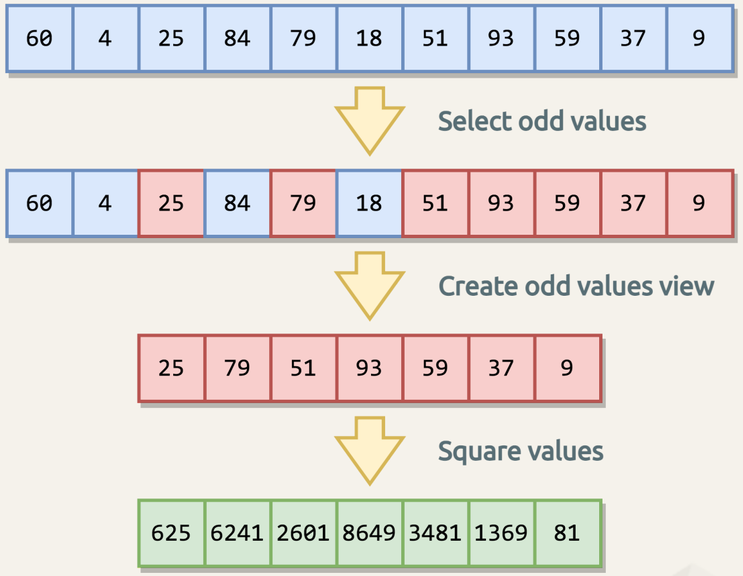
```
1  std::vector<int> vec{60, 4, 84, 14, 79, 51, 93, 25, 59};
2
3  std::ranges::sort(vec);
```

We supply a single range.

One of the main motivations for ranges is **correctness at construction**.

Range pipelines and views

A range view is a **lightweight (sub-)range representation** for ad-hoc processing.




```cpp
const std::vector<int> vec{60, 4, 84, 14, 79, 51, 93, 25, 59, 37, 9};

const auto is_odd = [](int value) -> bool { return value % 2;     };
const auto square = [](int value) -> int  { return value * value; };
const auto print  = [](int value) { std::cout << value << '\n'; };

// Shorthand names to remove noise (advice: don't apply 'using namespace').
namespace rg = std::ranges;
namespace rv = std::views;
```

Business logic:

```cpp
rg::for_each(vec | rv::filter(is_odd) | rv::transform(square), print);
```

(Code @ Compiler Explorer)

Ranges, views and projections

Range views have *reference semantics* and use **lazy evaluation**.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

const std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

```cpp
namespace rg = std::ranges;
namespace rv = std::views;

void process_log(rg::input_range auto log); // C++20 abbreviated function syntax.

const auto is_info  = [](const Event& e) { return (e.type == EventType::Info);  };
const auto is_error = [](const Event& e) { return (e.type == EventType::Error); };
```

```cpp
for (auto& e : system_log | rv::filter(is_error)) {
  std::cerr << std::format("Error event: ({}: '{}')\n", e.id, e.message);
}

process_log(system_log);                                // Feed the whole range.
process_log(system_log | rv::take_while(is_info));      // Feed up to first non-info.
process_log(system_log | rv::take(3));                  // Feed first three items.
process_log(system_log | rv::drop(2) | rv::take(3));    // Feed third to fifth item.
```

(Code @ Compiler Explorer)

Ranges, views and projections

Projections can be used in algorithms to **transform their behavior**.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

Projections for `sort` transformation:

```cpp
namespace rg = std::ranges;

rg::sort(system_log);                       // Default: take default comparison op.
rg::sort(system_log, {}, &Event::id);       // Sort using field 'id'.
rg::sort(system_log, {}, &Event::message); // Sort using field 'message'.
```

Many other algorithms take projections.

(Code @ Compiler Explorer)

The Range-v3 library

The ranges library is **fully implemented in Range-v3** and can be used today.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

const std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

```cpp
namespace rg = ranges;
namespace rv = ranges::views;   // Lazy views of ranges.
namespace ra = ranges::actions; // Eager in-place range modifications.

void process_log(rg::input_range auto log);

const auto is_error = [](const Event& e) { return (e.type == EventType::Error); };
```

Use an intermediate for modification:

```cpp
process_log(system_log | rv::take(50)
                       | rv::filter(is_error)
                       | rg::to<std::vector>() // <-- Create a mutable intermediate.
                       | ra::sort(std::less<>{}, &Event::id));
```

(Code @ Compiler Explorer)

End

Thank you!