C++20: A survey of selected features

Kyle Knoepfel
2 March 2022

C++ standards over the years (1)

- **C++98** – First ISO standard
- **C++03** – Primarily addressed defects of C++98
- **C++11** – **Major shift in coding**
  - Move semantics
  - Automatic type deduction
  - Variadic templates
  - Compile-time computation with `constexpr`
- **C++14** – Minor improvements over C++11
  - Automatic type deduction of function return values
  - Generic lambda expressions

C++ standards over the years (2)

- **C++17** – Moderate changes to language
  - Structured bindings
  - Compile-time conditionals – `if constexpr (...) {}`
  - Guaranteed copy elision (RVO)
  - Class template argument deduction – `std::vector v{1, 2, 3}`
  - Library enhancements such as
    - `std::variant`
    - `std::optional`
    - `std::filesystem`

C++ standards over the years (2)

- **C++17** – Moderate changes to language
  - Structured bindings
  - Compile-time conditionals – `if constexpr (...) {}`
  - Guaranteed copy elision (RVO)
  - Class template argument deduction – `std::vector v{1, 2, 3}`
  - Library enhancements such as
    - `std::variant`
    - `std::optional`
    - `std::filesystem`
- **C++20 – Major shift in coding**

C++ standards over the years (2)

- **C++17** – Moderate changes to language
  - Structured bindings
  - Compile-time conditionals – `if constexpr (...) {}`
  - Guaranteed copy elision (RVO)
  - Class template argument deduction – `std::vector v{1, 2, 3}`
  - Library enhancements such as
    - `std::variant`
    - `std::optional`
    - `std::filesystem`
- **C++20 – Major shift in coding**
- **C++23** – *Feature freeze was last month; moderate changes expected*

C++20 – the big four

There are four features that receive the most attention:

Modules	Separating declarations and definitions no longer necessary
Coroutines	Interruptible functions that retain state (akin to Python generators)
Concepts	Means of expressing constraints a given type must model
Ranges	Library extension that shifts emphasis from iterators to ranges

C++20 – the big four

There are four features that receive the most attention:

Modules	Separating declarations and definitions no longer necessary
Coroutines	Interruptible functions that retain state (akin to Python generators)
Concepts	Means of expressing constraints a given type must model
Ranges	Library extension that shifts emphasis from iterators to ranges

Today I hope to give you a taste of these things and a few others:

1. Smaller features
2. Ranges
3. Expansion of constant-expression support
4. Concepts and constraints

Preliminaries

- I assume you are familiar with basic aspects of C++.
  - Including automatic type deduction and range-based `for` loops
- I have removed many uses of `const` for illustrative purposes.
- These slides include *art*-framework examples.
  - The take-home message, however, should be framework agnostic.

Preliminaries

- I assume you are familiar with basic aspects of C++.
  - Including automatic type deduction and range-based `for` loops
- I have removed many uses of `const` for illustrative purposes.
- These slides include *art*-framework examples.
  - The take-home message, however, should be framework agnostic.
- The goal is not to get through every slide–we can stop to discuss.

Part 1: Smaller features

- Mathematical constants
- Container improvements
- Range-based `for` loops with initializers
- Designated initializers

Mathematical constants (1)

How do you calculate the area of a circle–i.e. where does π come from?

```
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double r) { return M_PI * r * r; }
```

Mathematical constants (1)

How do you calculate the area of a circle–i.e. where does π come from?

```
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double r) { return M_PI * r * r; }
```

```
#include "TMath.h"  // ROOT-provided solution

double area_of_circle(double r) { return TMath::Pi() * r * r; }
```

Mathematical constants (1)

How do you calculate the area of a circle–i.e. where does π come from?

```
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double r) { return M_PI * r * r; }
```

```
#include "TMath.h"  // ROOT-provided solution

double area_of_circle(double r) { return TMath::Pi() * r * r; }
```

```
#include <numbers>  // As of C++20

double area_of_circle(double r) { return std::numbers::pi * r * r; }
```

Mathematical constants (2)

Be careful of how you use `using` *directives*:

```
#include <cmath>
#include <numbers>  // As of C++20

void MyModule::produce(art::Event& e)
{
  // Calculate normal distribution PDF
  using namespace std::numbers;
  double x = ...;
  auto pdf = inv_sqrtpi / sqrt2 * std::pow(e, -0.5 * x * x); // Uh oh.
}
```

Mathematical constants (2)

Be careful of how you use `using` *directives*:

```
#include <cmath>
#include <numbers>  // As of C++20

void MyModule::produce(art::Event& e)
{
  // Calculate normal distribution PDF
  using namespace std::numbers;
  double x = ...;
  auto pdf = inv_sqrtpi / sqrt2 * std::pow(e, -0.5 * x * x); // Uh oh.
}
```

The name `e` corresponds to `art::Event&` and not `std::numbers::e`.

Mathematical constants (2)

Introduce named function that limits the scope of the `using` declaration:

```cpp
#include <cmath>
#include <numbers>  // As of C++20

double normal_distribution_pdf(double x)
{
  using namespace std::numbers;
  return inv_sqrtpi / sqrt2 * std::pow(e, -0.5 * x * x); // Fine
}

void MyModule::produce(art::Event& e)
{
  double x = ...;
  auto pdf = normal_distribution_pdf(x);
}
```

Mathematical constants (2)

Introduce named function that limits the scope of the `using` declaration:

```cpp
#include <cmath>
#include <numbers>  // As of C++20

double normal_distribution_pdf(double x)
{
  using namespace std::numbers;
  return inv_sqrtpi / sqrt2 * std::pow(e, -0.5 * x * x); // Fine
}

void MyModule::produce(art::Event& e)
{
  double x = ...;
  auto pdf = normal_distribution_pdf(x);
}
```

Or change the `art::Event&` function parameter name to something other than `e` !

`std::string` enhancements

Current method of checking for substring at beginning or end of string (e.g.):

```cpp
// Before C++20
namespace {
  bool ends_with(std::string const& s, char const* suffix)
  {
    auto pos = s.rfind(suffix);
    return pos != std::string::npos &&
           s.compare(pos, size(s) - strlen(suffix), suffix) == 0;
  }
}

std::string s{"It was the best of times"};
bool starts_with_it = s.find("It") == 0;
bool ends_with_times = ends_with(s, "times");
```

`std::string` enhancements

Current method of checking for substring at beginning or end of string (e.g.):

```cpp
// Before C++20
namespace {
  bool ends_with(std::string const& s, char const* suffix)
  {
    auto pos = s.rfind(suffix);
    return pos != std::string::npos &&
           s.compare(pos, size(s) - strlen(suffix), suffix) == 0;
  }
}

std::string s{"It was the best of times"};
bool starts_with_it = s.find("It") == 0;
bool ends_with_times = ends_with(s, "times");
```

```cpp
// With C++20
std::string s{"It was the best of times"};
bool starts_with_it = s.starts_with("It");
bool ends_with_times = s.ends_with("times");
```

Checking for inclusion in an associative container

Current method of checking for substring at beginning or end of string (e.g.):

```cpp
namespace {
  std::set<std::string> allowed_values{"fast", "slow"};
}
```

```cpp
// Before C++20
void validate(std::string const& speed)
{
  if (allowed_values.find(speed) != cend(allowed_values)) return;
  throw art::Exception{...};
}
```

Checking for inclusion in an associative container

Current method of checking for substring at beginning or end of string (e.g.):

```
namespace {
  std::set<std::string> allowed_values{"fast", "slow"};
}
```

```
// Before C++20
void validate(std::string const& speed)
{
  if (allowed_values.find(speed) != cend(allowed_values)) return;
  throw art::Exception{...};
}
```

```
// With C++20
void validate(std::string const& speed)
{
  if (allowed_values.contains(speed)) return;
  throw art::Exception{...};
}
```

Range-based `for` loops with initializers (1)

Allows for localizing the scope of auxiliary variables:

```
void f()
{
  for (size_t i = 0; auto const& hit : make_hits()) {
    g(i, hit);
    ++i;
  }
}
```

Range-based `for` loops with initializers (2)

Avoids dangling references:

```
class Tracks {
  vector<Track> tracks_;
public:
  auto const& data() const { return tracks_; }
}

Tracks make_tracks() { ... }

void f()
{
  for (auto const& track : make_tracks().data()) { // Uh oh!
    ...
  }
}
```

Range-based `for` loops with initializers (2)

Avoids dangling references:

```
class Tracks {
  vector<Track> tracks_;
public:
  auto const& data() const { return tracks_; }
}

Tracks make_tracks() { ... }

void f()
{
  for (auto const& track : make_tracks().data()) { // Uh oh!
    ...
  }
}
```

The `make_tracks` function returns a temporary object, whose lifetime is *not* extended by the range-based for. This means that `track` **points to invalid memory**.

Range-based `for` loops with initializers (2)

The solution to this problem looks different in C++20:

```
Tracks make_tracks() { ... }

void f()
{
  auto tracks = make_tracks();
  for (auto const& track : tracks.data()) { // Okay
    ...
  }
}

void f_cpp20()
{
  for (auto tracks = make_tracks();
         auto const& track : tracks.data()) { // C++20
    ...
  }
}
```

Designated initializers

C++20 supports designated initializers, which have been supported by the C language since the C99 standard.

```
struct Coordinate { double x = 0., y{}, z{0.}; };

// Before C++20
Coordinate c1;                // (0,0,0)
Coordinate c2{};              // (0,0,0)
Coordinate c3{1, 2};          // (1,2,0)
Coordinate c4{0, 0, 3};       // (0,0,3)

// With C++20
Coordinate c5{.z=3};          // (0,0,3)
Coordinate c6{.y=2, .x=1}; // Ordering error - not allowed
```

Part 2: Ranges library

Part 2: Ranges library

We often write procedural code (the “how”) instead of declarative code (the “what”). The goal of the ranges library is to get away from this.

Part 2: Ranges library

We often write procedural code (the “how”) instead of declarative code (the “what”). The goal of the ranges library is to get away from this.

Range libraries exist (e.g. Boost, Range-v3)

Part 2: Ranges library

We often write procedural code (the “how”) instead of declarative code (the “what”). The goal of the ranges library is to get away from this.

Range libraries exist (e.g. Boost, Range-v3)

C++20 codifies the concept of a *range*, which is a collection of elements represented by `begin()` and `end()` functions.

- `begin()` and `end()` do not need to return the same type
- Element iteration is defined by the range’s iterator

Part 2: Ranges library

We often write procedural code (the “how”) instead of declarative code (the “what”). The goal of the ranges library is to get away from this.

Range libraries exist (e.g. Boost, Range-v3)

C++20 codifies the concept of a *range*, which is a collection of elements represented by `begin()` and `end()` functions.

- `begin()` and `end()` do not need to return the same type
- Element iteration is defined by the range’s iterator

The following example uses an `art::Ptr<T>`, which is a persistable pointer that, when dereferenced, lazily loads a data product into memory.

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the 'U' plane.

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (art::Ptr<Hit> const& hit_ptr : hit_ptrs) {
  if (hit_ptr->View() == geo::kU) {
    sum += hit_ptr->Integral();
  }
}
```

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the ‘U’ plane.

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (art::Ptr<Hit> const& hit_ptr : hit_ptrs) {
  if (hit_ptr->View() == geo::kU) {
    sum += hit_ptr->Integral();
  }
}
```

Step 1: Get rid of explicit `art::Ptr` dereference.

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the 'U' plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (Hit const& hit : hit_ptrs | transform(to_element)) {
  if (hit.View() == geo::kU) {
    sum += hit.Integral();
  }
}
```

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the 'U' plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (Hit const& hit : hit_ptrs | transform(to_element)) {
  if (hit.View() == geo::kU) {
    sum += hit.Integral();
  }
}
```

Step 2: Get rid of conditional statement in `for` loop.

An example

```cpp
// Calculate the zero-initialized sum of the integrals
// of all hits on the ‘U’ plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (Hit const& u_hit : hit_ptrs |
                        transform(to_element) |
                        filter(hits_on_u_plane)) {
  sum += u_hit.Integral();
}
```

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the ‘U’ plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (Hit const& u_hit : hit_ptrs |
                        transform(to_element) |
                        filter(hits_on_u_plane)) {
  sum += u_hit.Integral();
}
```

Step 3: Get rid of explicit `Integral()` call in `for` loop.

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the ‘U’ plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (double integral : hit_ptrs |
                       transform(to_element) |
                       filter(hits_on_u_plane) |
                       transform(to_integral)) {
  sum += integral;
}
```

An example

```
// Calculate the zero-initialized sum of the integrals
// of all hits on the 'U' plane.

using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double sum = 0.;
art::PtrVector<Hit> hit_ptrs = get_hits(...);
for (double integral : hit_ptrs |
                       transform(to_element) |
                       filter(hits_on_u_plane) |
                       transform(to_integral)) {
  sum += integral;
}
```

Step 4: Replace `for` loop with `accumulate` function call.

An example

```
using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

art::PtrVector<Hit> hit_ptrs = get_hits(...);
double const sum = accumulate(hit_ptrs |
                              transform(to_element) |
                              filter(hits_on_u_plane) |
                              transform(to_integral),
                              0.);
```

An example

```
using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

art::PtrVector<Hit> hit_ptrs = get_hits(...);
double const sum = accumulate(hit_ptrs |
                              transform(to_element) |
                              filter(hits_on_u_plane) |
                              transform(to_integral),
                              0.);
```

Except a range-based `accumulate` *function doesn't exist in C++20.*

An example

Best you can do:

```
using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

art::PtrVector<Hit> hit_ptrs = get_hits(...);
auto const integrals = hit_ptrs |
                       transform(to_element) |
                       filter(hits_on_u_plane) |
                       transform(to_integral);
double const sum = std::accumulate(begin(integrals), end(integrals), 0.);
```

An example

Best you can do:

```cpp
using namespace std::ranges::views;
auto to_element = [](auto& hit_ptr) -> decltype(auto) { return *hit_ptr; };
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

art::PtrVector<Hit> hit_ptrs = get_hits(...);
auto const integrals = hit_ptrs |
                       transform(to_element) |
                       filter(hits_on_u_plane) |
                       transform(to_integral);
double const sum = std::accumulate(begin(integrals), end(integrals), 0.);
```

GCC 10 and Clang 13 partially support the ranges library.

Part 3: `consteval`, `constinit`, and more `constexpr`

Part 3: `consteval`, `constinit`, and more `constexpr`

A *constant expression* is an expression that can be evaluated at compile time.

Part 3: `consteval`, `constinit`, and more `constexpr`

A *constant expression* is an expression that can be evaluated at compile time.

```
int const n = 42;  // '42' is a constant expression
std::array<double, n> numbers{}; // OK because 'n' is a constant expression
```

Part 3: `consteval`, `constinit`, and more `constexpr`

A *constant expression* is an expression that can be evaluated at compile time.

```
int const n = 42;  // '42' is a constant expression
std::array<double, n> numbers{}; // OK because 'n' is a constant expression
```

Something that is declared `constexpr` is *eligible* to be evaluated at compile-time (e.g.):

```
constexpr int tripled(int num) noexcept { return 3 * num; }
```

Part 3: `consteval`, `constinit`, and more `constexpr`

A *constant expression* is an expression that can be evaluated at compile time.

```
int const n = 42;  // '42' is a constant expression
std::array<double, n> numbers{}; // OK because 'n' is a constant expression
```

Something that is declared `constexpr` is *eligible* to be evaluated at compile-time (e.g.):

```
constexpr int tripled(int num) noexcept { return 3 * num; }
```

These are constant expressions:

```
constexpr int m = 14;
constexpr int three_m = tripled(m);
static_assert(three_m == 42);
```

These are *not* constant expressions:

```
int j = 15;
int j_tripled = tripled(j);
assert(j_tripled == 45);
```

More `constexpr` algorithms in C++20 (1)

From the `<algorithm>` header.

`all_of`
`any_of`
`none_of`
`for_each`
`for_each_n`
`find`
`find_if`
`find_if_not`
`find_end`
`find_first_of`
`adjacent_find`
`count`
`count_if`
`mismatch`
`equal`
`exchange`

`is_permutation`
`search`
`search_n`
`copy`
`copy_n`
`copy_if`
`copy_backward`
`move`
`move_backward`
`transform`
`replace`
`replace_if`
`replace_copy`
`replace_copy_if`
`fill`

`is_permutation`
`search`
`search_n`
`copy`
`copy_n`
`copy_if`
`copy_backward`
`move`
`move_backward`
`transform`
`replace`
`replace_if`
`replace_copy`
`replace_copy_if`
`fill`

`is_sorted`
`is_sorted_until`
`lower_bound`
`upper_bound`
`equal_range`
`binary_search`
`merge`
`includes`
`set_union`
`set_intersection`
`set_difference`
`set_symmetric_difference`
`is_heap`
`is_heap_until`
`lexicographical_compare`

More `constexpr` algorithms in C++20 (2)

From the `<numeric>` header.

- `accumulate`
- `reduce`
- `inner_product`
- `transform_reduce`
- `partial_sum`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `adjacent_difference`
- `iota`

Surprising `constexpr` additions

C++20 supports `constexpr` for heap-allocating operations.

Surprising `constexpr` additions

C++20 supports `constexpr` for heap-allocating operations.

```
// This is valid C++20 code
constexpr bool allowed_value(std::string const& value)
{
  using namespace std::string_literals;
  std::vector okay{"small"s, "medium"s, "large"s};
  return std::ranges::find(okay, value) != end(okay);
}

static_assert(allowed_value("medium"));
```

Surprising `constexpr` additions

C++20 supports `constexpr` for heap-allocating operations.

```
// This is valid C++20 code
constexpr bool allowed_value(std::string const& value)
{
  using namespace std::string_literals;
  std::vector okay{"small"s, "medium"s, "large"s};
  return std::ranges::find(okay, value) != end(okay);
}

static_assert(allowed_value("medium"));
```

Required `constexpr` operations

```
std::string(char const*)
std::initializer_list<std::string>{...}
std::vector<std::string>(std::initializer_list<std::string>>)
std::ranges::find
std::vector<std::string>::begin()
std::vector<std::string>::end()
std::vector<std::string>::iterator::operator!=(iterator)
```

Very surprising `constexpr` additions

- `virtual` functions
- `dynamic_cast`
- `try`/`catch` blocks
- `typeid`

Mandating compile-time evaluation

C++20 introduces the concept of the *immediate function*, a function that **must** be evaluated at compile time.

- An immediate function is denoted by the `consteval` keyword
- An immediate function is a `constexpr` function

```
constexpr auto square(double num) noexcept { return num * num; }
consteval auto cube(double num) noexcept { return square(num) * num; }
```

Mandating compile-time evaluation

C++20 introduces the concept of the *immediate function*, a function that **must** be evaluated at compile time.

- An immediate function is denoted by the `consteval` keyword
- An immediate function is a `constexpr` function

```
constexpr auto square(double num) noexcept { return num * num; }
consteval auto cube(double num) noexcept { return square(num) * num; }
```

```
auto i = 4.;
constexpr auto j = 5.;
auto i_squared = square(i);
auto j_squared = square(j);

auto i_cubed = cube(i); // Compile-time error! - 'i' is not constant expression
auto j_cubed = cube(j);
```

Mandating compile-time initialization

With C++20, you can initialize a *mutable* variable at compile-time with `constinit`.

```
struct BigWidget {
  constexpr BigWidget() {
    // Takes a long time to initialize.
  }
  void update(); // Not const-qualified
};
```

Mandating compile-time initialization

With C++20, you can initialize a *mutable* variable at compile-time with `constinit`.

```cpp
struct BigWidget {
  constexpr BigWidget() {
    // Takes a long time to initialize.
  }
  void update(); // Not const-qualified
};
```

To use it:

```cpp
constinit BigWidget widget{};  // widget is not const!

int main() {
  widget.update();
}
```

What is “run-time”?

> *"You keep using that word. I do not think it means what you think it means."*
> –Inigo Montoya to Vizzini

What is “run-time”?

> *"You keep using that word. I do not think it means what you think it means."*
> –Inigo Montoya to Vizzini

A successful execution of a C++ program requires at least two program executions:

- Program that compiles the C++ source code (*compile-time*).
- Program that runs the compiled code (*run-time*).

We tend to think of computation happening at run-time. It happens at compile-time, too!

What is “run-time”?

> *"You keep using that word. I do not think it means what you think it means."*
> –Inigo Montoya to Vizzini

A successful execution of a C++ program requires at least two program executions:

- Program that compiles the C++ source code (*compile-time*).
- Program that runs the compiled code (*run-time*).

We tend to think of computation happening at run-time. It happens at compile-time, too!

Compile-time is just the run-time of a different program.

What is “run-time”?

> *"You keep using that word. I do not think it means what you think it means."*
> –Inigo Montoya to Vizzini

A successful execution of a C++ program requires at least two program executions:

- Program that compiles the C++ source code (*compile-time*).
- Program that runs the compiled code (*run-time*).

We tend to think of computation happening at run-time. It happens at compile-time, too!

Compile-time is just the run-time of a different program.

Compiler vendors need to worry about producing efficient binaries *and* doing it efficiently. Many recent additions to C++ target the latter. So. . .

Part 4: Compile-time features

- Constraints and concepts
- Abbreviated function templates

An *art* use case: product aggregation

art is able to combine data products that correspond to the same (sub)run.

An *art* use case: product aggregation

art is able to combine data products that correspond to the same (sub)run.

```
// Arithmetic types
assert(combine(1, 1) == 2);
assert(combine(14., 12.) == 26.);

// Vectors
assert((combine(v{15, 16, 17}, v{9}) == v{15, 16, 17, 9}));
assert((combine(v{13., 9.}, v{6.}) == v{13., 9., 6.}));
```

An *art* use case: product aggregation

art is able to combine data products that correspond to the same (sub)run.

```
// Arithmetic types
assert(combine(1, 1) == 2);
assert(combine(14., 12.) == 26.);

// Vectors
assert((combine(v{15, 16, 17}, v{9}) == v{15, 16, 17, 9}));
assert((combine(v{13., 9.}, v{6.}) == v{13., 9., 6.}));
```

But what about user-defined products?

User-defined product aggregation

art does not know how users wish to combine their own products.

User-defined product aggregation

art does not know how users wish to combine their own products.

```
class MyProduct {
  int i_;
public:
  explicit MyProduct(int const i) : i_{i} {}
  auto combine(MyProduct const b) const { return MyProduct{i_ + b.i_}; }

  // To generate default Boolean comparison semantics in C++20 ...
  auto operator<=>(MyProduct const&) const = default;
};
```

User-defined product aggregation

art does not know how users wish to combine their own products.

```
class MyProduct {
  int i_;
public:
  explicit MyProduct(int const i) : i_{i} {}
  auto combine(MyProduct const b) const { return MyProduct{i_ + b.i_}; }

  // To generate default Boolean comparison semantics in C++20 ...
  auto operator<=>(MyProduct const&) const = default;
};
```

```
assert((combine(MyProduct{2}, MyProduct{3}) == MyProduct{5}));
```

User-defined product aggregation

art does not know how users wish to combine their own products.

```
class MyProduct {
  int i_;
public:
  explicit MyProduct(int const i) : i_{i} {}
  auto combine(MyProduct const b) const { return MyProduct{i_ + b.i_}; }

  // To generate default Boolean comparison semantics in C++20 ...
  auto operator<=>(MyProduct const&) const = default;
};
```

```
assert((combine(MyProduct{2}, MyProduct{3}) == MyProduct{5}));
```

art must seamlessly support each of these product-aggregation behaviors.

No templates – *reductio ad absurdum*

No templates – *reductio ad absurdum*

```
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of arithmetic types
auto combine(vector<int> a, vector<int> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}
auto combine(vector<double> a, vector<double> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
auto combine(MyProduct const a, MyProduct const b) { return a.combine(b); }
```

No templates – *reductio ad absurdum*

```
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of arithmetic types
auto combine(vector<int> a, vector<int> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}
auto combine(vector<double> a, vector<double> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
auto combine(MyProduct const a, MyProduct const b) { return a.combine(b); }
```

Would accomplish the goal, but it is not extensible.

Generalize the vector

```
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of any type
template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
auto combine(MyProduct const a, MyProduct const b) { return a.combine(b); }
```

Generalize the vector

```
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of any type
template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
auto combine(MyProduct const a, MyProduct const b) { return a.combine(b); }
```

We can generalize the implementation for user-defined types, **assuming** those types adhere to a common interface.

Generalize user-defined aggregation

```
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of any type
template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
template <typename T>
auto combine(T const a, T const b) { return a.combine(b); }
```

Generalize user-defined aggregation

```cpp
// Arithmetic types
auto combine(int const a, int const b) { return a + b; }
auto combine(double const a, double const b) { return a + b; }

// Vector of any type
template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

// User-defined types
template <typename T>
auto combine(T const a, T const b) { return a.combine(b); }
```

What about generalizing arithmetic types?

Generalizing arithmetic aggregation – attempt 1

```
template <typename T>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { // Error - Redefinition of template
  return a.combine(b);
}
```

Generalizing arithmetic aggregation – attempt 1

```cpp
template <typename T>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { // Error - Redefinition of template
  return a.combine(b);
}
```

What about using `static_assert`?

Generalizing arithmetic aggregation – attempt 2

```
template <typename T>
auto combine(T const a, T const b) {
  static_assert(std::is_arithmetic_v<T>);
  return a + b;
}

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { // Error - Still a redefinition of template
  static_assert(not std::is_arithmetic_v<T>);
  return a.combine(b);
}
```

Generalizing arithmetic aggregation – attempt 2

```
template <typename T>
auto combine(T const a, T const b) {
  static_assert(std::is_arithmetic_v<T>);
  return a + b;
}

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { // Error - Still a redefinition of template
  static_assert(not std::is_arithmetic_v<T>);
  return a.combine(b);
}
```

Static assertions cannot be used to select a template.

Generalizing arithmetic aggregation with SFINAE

Template metaprogrammers rely on *substitution failure is not an error*.

Not friendly to the average user!

Generalizing arithmetic aggregation with SFINAE

Template metaprogrammers rely on *substitution failure is not an error*.

Not friendly to the average user!

```cpp
template <typename T, typename = enable_if_t<std::is_arithmetic_v<T>>>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T, typename = enable_if_t<not std::is_arithmetic_v<T>>>
auto combine(T const a, T const b) { return a.combine(b); }
```

Generalizing arithmetic aggregation with SFINAE

Template metaprogrammers rely on *substitution failure is not an error*.

Not friendly to the average user!

```
template <typename T, typename = enable_if_t<std::is_arithmetic_v<T>>>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T, typename = enable_if_t<not std::is_arithmetic_v<T>>>
auto combine(T const a, T const b) { return a.combine(b); }
```

Have to be careful about having mutually exclusive template declarations.

Concepts

C++ concepts were introduced so that functionalities could be grouped according to the *behaviors* of a type.

Concepts

C++ concepts were introduced so that functionalities could be grouped according to the *behaviors* of a type.

Satisfaction of a concept happens *early on* during the template instantiation process. This allows for:

- Easier to understand error messages.
  - Ever tried sorting an `std::list`?
- Overload lookup rules to simplify template implementations.

Generalizing arithmetic aggregation with concepts (1)

The concept:

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

Generalizing arithmetic aggregation with concepts (1)

The concept:

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

The constraint:

```
template <typename T>
  requires Arithmetic<T>  // <- constraint
auto combine(T const a, T const b) { return a + b; }
```

Generalizing arithmetic aggregation with concepts (1)

The concept:

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

The constraint:

```
template <typename T>
  requires Arithmetic<T>  // <- constraint
auto combine(T const a, T const b) { return a + b; }
```

For the above `combine` template to be considered, the type `T` must satisfy the `Arithmetic` concept.

Generalizing arithmetic aggregation with concepts (1a)

The concept:

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

The constraint:

```
template <Arithmetic T> // <- abbreviated constraint
auto combine(T const a, T const b) { return a + b; }
```

For the above `combine` template to be considered, the type `T` must satisfy the `Arithmetic` concept.

Generalizing arithmetic aggregation with concepts (2)

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

```
template <Arithmetic T>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { return a.combine(b); }
```

Generalizing arithmetic aggregation with concepts (2)

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

```
template <Arithmetic T>
auto combine(T const a, T const b) { return a + b; }

template <typename T>
auto combine(vector<T> a, vector<T> const& b) {
  a.insert(cend(a), cbegin(b), cend(b));
  return a;
}

template <typename T>
auto combine(T const a, T const b) { return a.combine(b); }
```

User-defined `combine` template does *not* need a `requires` clause.

This solution *is* extensible and scalable.

Abbreviated function templates

Compile-time square of a number:

```
template <Arithmetic T>
constexpr auto square(T x) { return x * x; }
```

Abbreviated function templates

Compile-time square of a number:

```
template <Arithmetic T>
constexpr auto square(T x) { return x * x; }
```

This written as an *abbreviated function template*:

```
constexpr auto square(Arithmetic auto x) { return x * x; }
```

Abbreviated function templates

Compile-time square of a number:

```
template <Arithmetic T>
constexpr auto square(T x) { return x * x; }
```

This written as an *abbreviated function template*:

```
constexpr auto square(Arithmetic auto x) { return x * x; }
```

Or if you don’t care about the arithmetic constraint:

```
constexpr auto square(auto x) { return x * x; }
```

Abbreviated function templates

Compile-time square of a number:

```
template <Arithmetic T>
constexpr auto square(T x) { return x * x; }
```

This written as an *abbreviated function template*:

```
constexpr auto square(Arithmetic auto x) { return x * x; }
```

Or if you don’t care about the arithmetic constraint:

```
constexpr auto square(auto x) { return x * x; }
```

Please use your judgment–just because something can be done doesn’t mean it should be.

Other features not covered

- Format library
  - Pythonic style of formatting strings
- Calendar and timezone support (via `<chrono>` library)
  - Goodbye `getTimeOfDay`, hello `std::chrono::time_of_day`
- `std::span`
  - Non-owning view into a container
- `std::source_location`
  - Goodbye *many* macros
- etc.

References

- C++20 features
  - https://en.cppreference.com/w/cpp/20
- C++20 compiler status
  - https://en.cppreference.com/w/Template:cpp/compiler_support/20