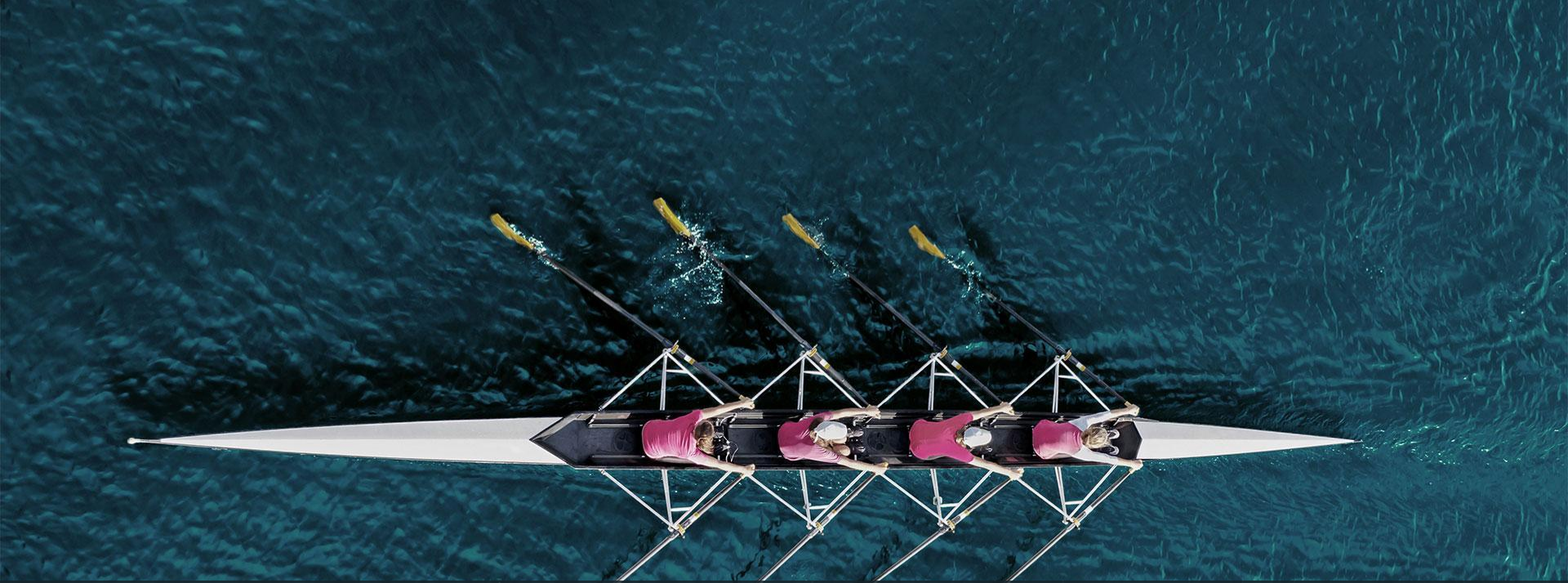


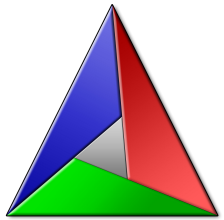
Maven

C++20 Techniques For Algorithmic Trading

Containerised Build Environment

C++20

Core libraries, exchange connectivity, trading systems, pricing engine



CMake 3.23

Modern target based builds, cross-platform, code generation, CCache + Icecream

Python 3.10

Research environment, tools, multi-language code generation, package management

Clang Tools

Clang Build Analyser, Sanitizers (Asan/Lsan/Tsan/Ubsan), Clang-Tidy, Clang-Format

Tracking Compile Time Performance

Thirdparty Libraries

And lots lots more …

Customisation Points

```
namespace std {

template<> struct hash<::maven::networking::NamedEndpoint>
{
    using result_type = size_t;
    using argument_type = ::maven::networking::NamedEndpoint;
    result_type operator()(argument_type const& value) const { return value.hash(); }
};

}
```

Customisation Points In Boost

```
namespace maven::networking {

struct Subnet;
struct SubnetEndpoint;
class NamedEndpoint;

void validate(boost::any&, std::vector<std::string> const&, Subnet*, int);
void validate(boost::any&, std::vector<std::string> const&, SubnetEndpoint*, int);
void validate(boost::any&, std::vector<std::string> const&, NamedEndpoint*, int);

}

namespace boost:asio::ip {

void validate(boost::any&, std::vector<std::string> const&, address_v4*, int);
void validate(boost::any&, std::vector<std::string> const&, address*, int);
void validate(boost::any&, std::vector<std::string> const&, tcp::endpoint*, int);
void validate(boost::any&, std::vector<std::string> const&, udp::endpoint*, int);

}
```

Customisation Points For Sequences

```
template<class T> requires (::maven::saf::IsSequence<T>)
void validate(boost::any& v, std::vector<std::string> const& s, T*, int)
{
    using namespace boost::program_options;

    if (v.empty())
        v = boost::any(T());

    T* tv = boost::any_cast<T>(&v);
    MAVEN_ASSERT(nullptr != tv);
    for (auto& ss : s)
    {
        try
        {
            boost::any a;
            std::vector const cv{ss};
            if constexpr ( requires { typename T::mapped_type; }; } )
            {
                using E = std::pair<typename T::key_type, typename T::mapped_type>;
                validate(a, cv, static_cast<E*>(nullptr), 0);
                if (!tv->insert(std::move(boost::any_cast<E&>(a))).second)
                    boost::throw_exception(::maven::saf::detail::DuplicateMapping());
            }
            else
            {
                validate(a, cv, static_cast<typename T::value_type*>(nullptr), 0);
                tv->insert(std::move(boost::any_cast<typename T::value_type&>(a)));
            }
        }
        catch (boost::bad_lexical_cast const&)
        {
            boost::throw_exception(invalid_option_value(ss));
        }
    }
}
```

Customisation Points In Maven Libraries

```
namespace maven::serialization {

template<typename K, class H, class P, class A>
struct IsSequence<absl::flat_hash_set<K, H, P, A>> : std::true_type {};

template<typename K, class H, class P, class A>
struct SequenceTraits<absl::flat_hash_set<K, H, P, A>> : detail::StdSetTraits<absl::flat_hash_set<K, H, P, A>> {};

}
```

```
template<class SerializerType, concepts::Sequence T> requires (!concepts::Blob<T>)
void serialize(SerializerType& s, T const& value, OverloadStrong)
{
    using Traits = SequenceTraits<T>;

    s.writer().beginSequence(Traits::size(value));
    for (auto const& element : value)
        s.template serialize<typename Traits::ValueType>(element);
    s.writer().endSequence();
}
```

Customisation Points In Boost Test

```
namespace boost::test_tools::tt_detail {

template<::maven::serialization::concepts::cxx20::WriteSerializable T> requires
    (!boost::has_left_shift<std::ostream, T>::value) &&
    (not maven::conversion::concepts::SpecializedToStringEnum<T>)
struct print_log_value<T>
{
    void operator()(std::ostream& os, T const& t)
    {
        ::maven::serialization::WriteSerializer<::maven::serialization::JsonStreamWriter>(os).serialize(t);
    }
};

}
```

Designated Initializers

```
template<Protocol P> requires (VersionTraits<P>::Version == 3)
struct PortEndpoints<P>
{
    BOOST_HANA_DEFINE_STRUCT(PortEndpoints,
        (boost::asio::ip::tcp::endpoint, primary),
        (boost::asio::ip::tcp::endpoint, secondary),
        (std::optional<boost::asio::ip::tcp::endpoint>, disasterRecovery));

    static PortEndpoints fromString(std::span<std::string const>& arg) noexcept(false)
    {
        auto const tokens = std::exchange(arg, arg.subspan(3));
        if (tokens.size() < 3)
            throw std::runtime_error("not enough tokens");
        return PortEndpoints{
            .primary = std::make_from_tuple<boost::asio::ip::tcp::endpoint>(
                conversion::endpointFromString(tokens[0])),
            .secondary = std::make_from_tuple<boost::asio::ip::tcp::endpoint>(
                conversion::endpointFromString(tokens[1])),
            .disasterRecovery = tokens[2].empty() ?
                std::nullopt :
                std::optional(std::make_from_tuple<boost::asio::ip::tcp::endpoint>(
                    conversion::endpointFromString(tokens[2])))};
    }
    inline static constexpr std::string_view CommandLineFormat =
        "<primary ip:port>,<secondary ip:port>,[<disaster recovery ip:port>]";
};
```

From Boost Concepts

```
template<typename T>
struct Volume
    : serialization::concepts::WriteSerializable<T>
    , boost::DefaultConstructible<T>
    , boost::CopyConstructible<T>
    , boost::Assignable<T>
    , boost::EqualityComparable<T>
    , boost::Comparable<T>
{
    BOOST_CONCEPT_USAGE(Volume)
    {
        T v2;
        v2 = v + v;
        v2 = v - v;
        v2 += v;
        v2 -= v;
        double d = conversion::numericCast<double>(v);
        int i = conversion::numericCast<int>(v);
        v2 = conversion::numericCast<T>(d);
        canonical::Volume const v3 = toCanonicalVolume(v);
        T v4 = toSpecificVolume<T>(v3);
    }

private:
    Volume();

    T const v;
};
```

To C++20 Concepts

```
namespace cxx20 {

template <typename T>
concept Volume =
    serialization::concepts::cxx20::WriteSerializable<T> &&
#ifdef __cpp_lib_concepts
    std::regular<T> &&
    std::totally_ordered<T> &&
#endif
    requires(T v, T v2, canonical::Volume v3, double d)
{
    { v + v } -> std::same_as<T>;
    { v - v } -> std::same_as<T>;
    { v2 += v } -> std::same_as<T&>;
    { v2 -= v } -> std::same_as<T&>;

    { conversion::numericCast<double>(v) } -> std::same_as<double>;
    { conversion::numericCast<int>(v) } -> std::same_as<int>;
    { conversion::numericCast<T>(d) } -> std::same_as<T>;
    { toCanonicalVolume(v) } -> std::same_as<canonical::Volume>;
    { toSpecificVolume<T>(v3) } -> std::same_as<T>;
};

}

template<cxx20::Volume> struct [[deprecated("use connectivity::concepts::cxx20::Volume")]] Volume {};
```

Non-Type Template Parameters

```
/// To be used instead of `auto` for non-type template parameters when expecting
/// string literals, since this class will construct a meta::String from them
///
/// template<auto Value> static constexpr auto using_auto = Value;
/// template<meta::Auto Value> static constexpr auto using_Auto = Value;
///
/// constexpr int i = using_auto<314>;
/// constexpr String s = using_auto<"test string">; // compiler error
///
/// constexpr int i = using_Auto<314>;
/// constexpr String s = using_Auto<"test string">; // will compile
```

```
template<class ValueType>
struct Auto
{
    ValueType value;

    template<class T>
    consteval explicit(false) Auto(T&& v) : value{std::forward<T>(v)} {}
    consteval explicit(false) Auto(ValueType v) : value{std::move(v)} {}
    Auto(Auto const&) = default;
    Auto(Auto&&) = default;
    Auto& operator=(Auto const&) = delete;
    Auto& operator=(Auto&&) = delete;

    constexpr explicit(false) operator ValueType() const { return value; }

    auto operator<=>(Auto const&) const = default;
};

template<std::size_t N>
Auto(const char(&)[N]) -> Auto<String<N-1>>;

template<class T>
Auto(T) -> Auto<T>;
```

Message Dispatch

```
template <typename Handler>
void receive(Handler&& handler, std::span<std::byte const> const data)
{
    auto messageId = Interpreter::getMessageId(data);
    auto switchCases = getSwitchCases(Interpreter::Messages);
    functional::switch_(messageId, switchCases, [&]<typename HanaMsgType>(HanaMsgType)
        {
            using MessageType = typename HanaMsgType::type;
            mInterpreter.template deserialize<MessageType>(data, handler);
        },
        [this](MessageIdType messageId)
        {
            mErrorHandler(messageId);
        }
    );
}
```

Compile Time Switch Generation

```
#define MAVEN_FUNCTIONAL_SWITCH_CASE(Z,N,_)                                                 \
    case boost::hana::value<decltype(switchCondition(boost::hana::at_c<N>(caseList)))>() :  \
        return NoReturn::invoke<Result>(                                                    \
            std::forward<CaseHandler>(caseHandler),                                         \
            switchTag(boost::hana::at_c<N>(caseList)));
#define MAVEN_FUNCTIONAL_SWITCH_OVERLOAD(Z,N,_)                                             \
template<class Result, class Condition, class CaseList, class CaseHandler, class DefaultHandler>  \
constexpr Result switch_(                                                                   \
    std::integral_constant<std::size_t, N>,                                                 \
    Condition condition,                                                                    \
    CaseList caseList,                                                                      \
    CaseHandler& caseHandler,                                                               \
    DefaultHandler& defaultHandler)                                                         \
{                                                                                           \
    switch (condition)                                                                      \
    {                                                                                       \
        BOOST_PP_REPEAT_ ## Z(N, MAVEN_FUNCTIONAL_SWITCH_CASE, nil)                         \
    }                                                                                       \
    return switchInvokeDefaultHandler<Result, Condition, DefaultHandler>(condition, defaultHandler);  \
}
BOOST_PP_REPEAT(MAVEN_FUNCTIONAL_SWITCH_MAX, MAVEN_FUNCTIONAL_SWITCH_OVERLOAD, nil)
#undef MAVEN_FUNCTIONAL_SWITCH_OVERLOAD
#undef MAVEN_FUNCTIONAL_SWITCH_CASE
```

Customisation For Compile Time Performance

```
namespace boost::hana {

// Specify hana tag via specializations; this is an optimization to reduce use of SFINAE.
// See https://www.boost.org/doc/libs/develop/libs/hana/doc/html/structboost_1_1hana_1_1tag__of.html
template<class... C>
struct tag_of<::maven::functional::SwitchCaseList<C...>> { using type = ::maven::functional::SwitchCaseListTag; };
template<class CT, CT... C>
struct tag_of<::maven::functional::SwitchCaseSequence<CT, C...>> { using type = ::maven::functional::SwitchCaseListTag; };

// Model Iterable concept via hana tag dispatch.
// See https://www.boost.org/doc/libs/develop/libs/hana/doc/html/index.html#tutorial-core-tag_dispatching
// To satisfy Iterable, we implement at, drop_front and is_empty (minimal requirement), plus length for efficiency.
template<>
struct Iterable<::maven::functional::SwitchCaseListTag> { static constexpr bool value = true; };

template<>
struct at_impl<::maven::functional::SwitchCaseListTag> {
    template<class... C>
    static constexpr auto apply(::maven::functional::SwitchCaseList<C...>, auto N) {
        return typename boost::mp11::mp_arg<N>::template fn<C...>();
    }
};
```

Constraining Switch Generation

```
template<
    concepts::SwitchCondition Condition,
    concepts::SwitchCaseList CaseList,
    class CaseHandler,
    concepts::SwitchDefaultHandlerFor<Condition> DefaultHandler = detail::SwitchDefaultHandler<Condition>>
    requires concepts::SwitchCaseListFor<CaseList, Condition, CaseHandler>
constexpr SwitchResult<Condition, CaseList, CaseHandler, DefaultHandler> switch_(
    Condition condition,
    CaseList caseList,
    CaseHandler&& caseHandler,
    DefaultHandler&& defaultHandler = DefaultHandler{})
{
    constexpr std::size_t Size = decltype(boost::hana::length(caseList))::value;
    static_assert(Size < MAVEN_FUNCTIONAL_SWITCH_MAX, "Please increase MAVEN_FUNCTIONAL_SWITCH_MAX");
    using Result = SwitchResult<Condition, CaseList, CaseHandler, DefaultHandler>;
    return detail::switch_<Result, Condition, CaseList, CaseHandler, DefaultHandler>(
        std::integral_constant<std::size_t, Size>{},
        condition,
        caseList,
        caseHandler,
        defaultHandler);
}
```

Compile Time Switch Constraints

```cpp
// nb. there is no trait for class types with implicit conversion to unique integral or enumeration type (as accepted
// by switch [stmt.switch]/2), so we need to accept all class types.
template<class T> concept SwitchCondition = std::is_integral_v<T> or std::is_enum_v<T> or std::is_class_v<T>;

template<class T, unsigned M, unsigned N>
concept SwitchCaseElementsDistinct = requires
{
    // comma operator silliness is to include M and N indices in constraint violation message
    []<auto C, auto D>() requires ("expected distinct elements at indices", M, N, C != D) {}.template operator()<
        boost::hana::value<decltype(detail::switchCondition(boost::hana::at_c<M>(std::declval<T>())))>(),
        boost::hana::value<decltype(detail::switchCondition(boost::hana::at_c<N>(std::declval<T>())))>()>();
};

template<class T, unsigned N>
concept SwitchCaseElementUnique = requires
{
    // gcc "error: duplicate case value" is useless, check ourselves in O(n^2), this is fine 😠
    []<std::size_t... J>(std::index_sequence<J...>) requires requires
    {
        ([]<std::size_t K>() requires SwitchCaseElementsDistinct<T, K, N> {}.template operator()<J>(), ...);
    } {}(std::make_index_sequence<N>());
};

template<class T>
concept SwitchCaseList = boost::hana::Iterable<boost::hana::tag_of_t<T>>::value && requires(T t)
{
    []<std::size_t... I>(std::index_sequence<I...>) requires requires
    {
        ([]<std::size_t J>() requires SwitchCaseElementUnique<T, J> {}.template operator()<I>(), ...);
    } {}(std::make_index_sequence<decltype(boost::hana::length(t))::value>());
};
```

P1306R1/P2237R0: Expansion Statements

```
template<class Result, class Condition, class CaseList, class CaseHandler, class DefaultHandler>
constexpr Result switch_(
    std::integral_constant<std::size_t, N>,
    Condition condition,
    CaseList caseList,
    CaseHandler& caseHandler,
    DefaultHandler& defaultHandler)
{
    switch (v.index()) {
    template for (constexpr int i : ints(0, N)) {
        case  boost::hana::value<decltype(switchCondition(boost::hana::at_c<N>(caseList)))>():
            return NoReturn::invoke<Result>(std::forward<CaseHandler>(caseHandler), switchTag(boost::hana::at_c<N>(caseList)));
    }
    return switchInvokeDefaultHandler<Result, Condition, DefaultHandler>(condition, defaultHandler);
}
```

Compile Time Switch Constraints

```
namespace maven::concurrency {

/// Defer task for later execution on the current thread.
/// \param f The function object to later invoke. Must implement 'void operator()()'.
template<std::invocable F>
void deferTask(F&& f)
{
    DeferredTaskScheduler::getThreadInstance().post(std::forward<F>(f));
}

/// Defer coroutine for later resumption on the current thread.
/// If the deferred task scheduler is destroyed before resumption, will throw std::system_error into the coroutine.
/// The coroutine may be destroyed while suspended, in which case there is no visible effect.
inline auto defer()
{
    struct [[nodiscard("should co_await")]] Awaiter
    {
        detail::ResumptionGuard rguard = {};
        functional::CallbackGuard cguard = {};
        bool await_ready() const { return false; }
        void await_suspend(coro::coroutine_handle<> h) { deferTask(makeGuardedCallback(rguard.suspend(h), cguard)); }
        void await_resume() { rguard.resume(); }
    };
    return Awaiter();
}

}
```

Accessing Experimental C++23 Library Features

```
// Implementation of http://wg21.link/p0792 - type-erasing callable wrapper that takes a reference to its target.
//
// Warning: the FunctionRef ctor accepts temporaries, so that it can be used as a function parameter. It is an error to
// use FunctionRef after the temporary to which it has been bound has been destroyed:
//
//     FunctionRef<void()> f{[] {}};
//     f(); // UB
//
// A FunctionRef declared as a function parameter should not be used outside that function scope; a FunctionRef
// declared within function scope (or class scope) should only be bound to an lvalue with lifetime guaranteed enclosing.
template<class Result, class... Args>
class FunctionRef<Result (Args...)>
{
public:
    using result_type = Result;

    constexpr FunctionRef() noexcept = default;
    constexpr FunctionRef(std::nullptr_t) noexcept : FunctionRef{} {}

    constexpr FunctionRef(FunctionRef const&) noexcept = default;
```

Find Out More

Come and meet us at C++OnSea
https://cpponsea.uk/sponsors/

C++ Technology Role
https://grnh.se/6d49f3131us

Maven's Blog
https://www.mavensecurities.com/blog/