Core C++ 2021

C++ 20: The Big Four

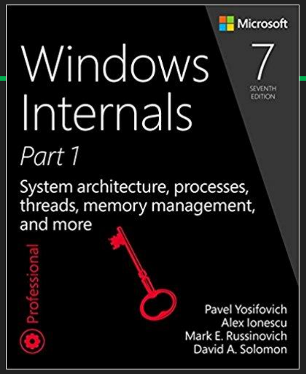
Pavel Yosifovich

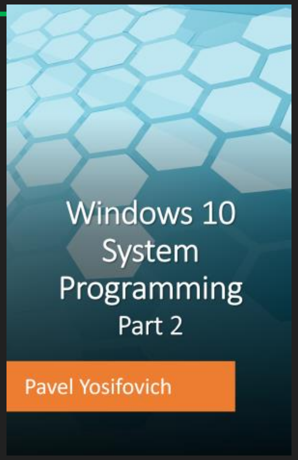
@zodiacon

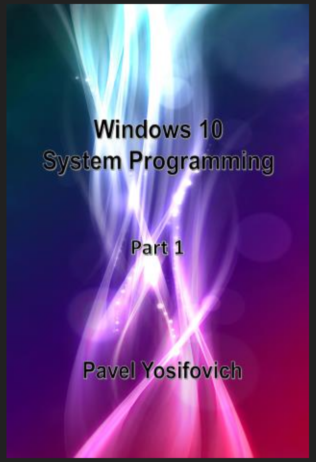
zodiacon@live.com

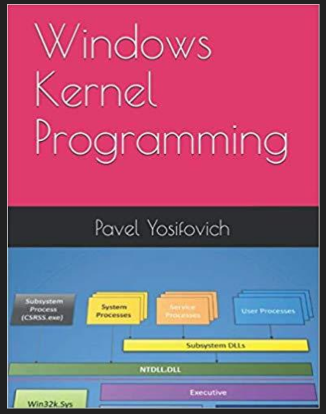
About Me

- Developer, Trainer, Author, Speaker
- Book author
  - “Windows Kernel Programming” (2019)
  - “Windows Internals 7th edition, Part 1” (co-author, 2017)
  - “Windows 10 System Programming, Part 1” (2020)
  - “Windows 10 System Programming, Part 2” (WIP)
- *Pluralsight* and *Pentester Academy* course author
- Author of several open-source tools (http://github.com/zodiacon)
- Website: http://scorpiosoftware.net









Agenda

- Overview
- Concepts
- Ranges
- Coroutines
- Modules
- Summary

Overview

- C++ evolution accelerated from C++ 11
- Getting started with C++ is challenging
  - So is making use of modern C++ features
- Guidelines are essential
- C++ 20 is a big release
- C++ is arguably the biggest programming language in terms of language features and paradigms
  - Just look at the standard

Concepts

Example: the `sort` Algorithm

- The `sort` algorithm

```
template <typename Iterator>
constexpr void sort(const Iterator first, const Iterator last);
```

- Sorting a vector

```
vector numbers{ 12, 33, 8, 55, 101, 87, 6, 16, 87, 34 };
sort(begin(numbers), end(numbers));
```

- Sorting a list

```
list numbers2{ numbers };
sort(begin(numbers2), end(numbers2));
```

(a bunch of incomprehensible error messages)

C++ Functions

- Before C++ 20, functions can be either
  - Specific

```cpp
bool IsPrime(int n) {
    auto limit = static_cast<int>(std::sqrt(n));
    for (int i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

  - Generic

```cpp
template<typename T>
bool IsPrime(T n) {
    au
    fo


    re
}
```

```cpp
bool IsPrime(auto n) {
    auto limit = static_cast<decltype(n)>(std::sqrt(n));
    for (decltype(n) i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

Testing `IsPrime`

- `cout << IsPrime(17) << endl;`
  - 1
- `cout << IsPrime((1LL << 33) – 1) << endl;`
  - 0
- `cout << IsPrime(5.9) << endl;`
  - error C2296: '%': illegal, left operand has type '_T0’ with _T0=double
  - error C2665: 'sqrt': none of the 3 overloads could convert all the argument types
- `cout << IsPrime(“hello”) << endl;`
  - error C2296: '%': illegal, left operand has type '_T0’ with T0=const char*

Concepts

- Constraints on generic types
- Makes it clear what types are expected in generic code
  - Compiler errors become comprehensible ☺

```
template<typename T>
concept Integral = std::is_integral_v<T>;
```

Testing `IsPrime` with a Concept

```
template<Integral T>
bool IsPrime(T n) {
    auto limit = static_cast<T>(std::sqrt(n));
    for (T i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

- `cout << IsPrime(5.9) << endl;`
  - error C2672: 'IsPrime': no matching overloaded function found
  - error C7602: 'IsPrime': the associated constraints are not satisfied

Functions with Concepts

```
template<Integral T>
bool IsPrime(T n) {
    auto limit = static_cast<T>(std::sqrt(n));
    for (T i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

```
bool IsPrime(Integral auto n) {
    auto limit = static_cast<decltype(n)>(std::sqrt(n));
    for (Integral auto i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

```
template<typename T>
    requires Integral<T>
bool IsPrime(T n) {
    auto limit = static_cast<T>(std::sqrt(n));
    for (T i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

Another Example

- Fully generic

```
template<typename T>
auto CountPrimes(T first, T last) {
    T count = 0;
    for (auto i = first; i <= last; i++)
        if (IsPrime(i))
            count++;
    return count;
}
```

```
auto CountPrimes(auto first, auto last) {
    decltype(last) count = 0;
    for (decltype(count) i = first; i <= last; i++)
        if (IsPrime(i))
            count++;
    return count;
}
```

- With Concepts

```
template<Integral T>
auto CountPrimes(T first, T last) {
    T count = 0;
    for (T i = first; i <= last; i++)
        if (IsPrime(i))
            count++;
    return count;
}
```

```
auto CountPrimes(Integral auto first, Integral auto last) {
    Integral auto count = 0;
    for (decltype(last) i = first; i <= last; i++)
        if (IsPrime(i))
            count++;
    return count;
}
```

More Concepts

- Many concepts defined by the standard library in the `<concepts>` header
- Functions can be overloaded on concepts
  - Compiler selects the most constrained match
- Combining concepts

```
template<typename T>
concept UnsignedIntegral = Integral<T> && !std::is_signed_v<T>;
```

```
bool IsPrime(UnsignedIntegral auto n) {
    auto limit = static_cast<decltype(n)>(std::sqrt(n));
    for (UnsignedIntegral auto i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

(Some) Concepts in the Standard Library

- Arithmetic
  - `integral`, `signed_integral`, `unsigned_integral`, `floating_point`
- Object-related
  - `is_object`, `movable`, `copyable`, `semiregular`, `regular`
  - `derived_from`, `convertible_to`
- Construction
  - `default_constructible`, `move_constructible`, `copy_constructible`, `constructible_from`
- Iterators
  - `forward_iterator`, `bidirectional_iterator`, `random_access_iterator`, …
- `sortable`, `mergeable`, `invocable`, `predicate`

The `requires` Clause

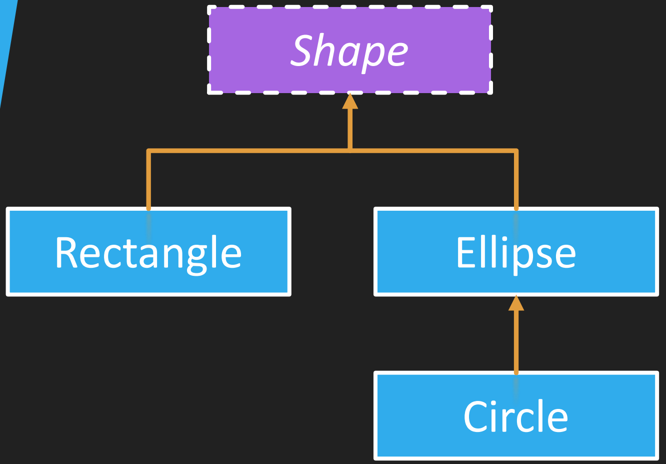
- Allows more powerful concepts definitions

```cpp
template<typename T>
concept Number = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
    { a - b } -> std::convertible_to<T>;
    { a * b } -> std::convertible_to<T>;
    { a / b } -> std::convertible_to<T>;
};
```

```cpp
template<typename T>
concept Iterator = requires(T it) {
    typename T::value_type;
    { it++ } -> std::convertible_to<T>;
    { *it } -> std::convertible_to<typename T::value_type>;
};
```

```cpp
struct MyIterator {
    using value_type = int;

    MyIterator operator++(int);
    value_type operator*() const;
    //...
};

static_assert(Iterator<MyIterator>);
```

Example: Shapes (Polymorphism)




```
struct Shape abstract {
    virtual Color GetColor() const = 0;
    virtual void Draw(DrawingContext& dc) const = 0;
    virtual ~Shape() = default;

protected:
    Shape() = default;
};

struct Rectangle : Shape {
    Rectangle(double width, double height);
                                                erride;
```

```
void DrawAll(vector<unique_ptr<Shape>> const& shapes, DrawingContext& dc) {
    for (auto& s : shapes)
        s->Draw(dc);
}

void BuildDrawing() {
    vector<unique_ptr<Shape>> shapes;
    shapes.push_back(make_unique<Circle>(6));
    shapes.push_back(make_unique<Rectangle>(4, 10));
    shapes.push_back(make_unique<Ellipse>(10, 3));

    auto& dc = DrawingContext::Get();
    DrawAll(shapes, dc);
}
```

Example: Shapes (Concepts)

```cpp
template<typename T>
concept Shape = requires(T s) {
    { s.template GetColor() } -> std::convertible_to<Color>;
    s.template Draw();
};
```

```cpp
struct Rectangle {
    Rectangle(double width, double height);
    Color GetColor() const;
    void Draw() const;
};

struct Ellipse {
    Ellipse(double radius1, double radius2);
    Color GetColor() const;

    void Draw() const;
};
```

```cpp
struct Circle : Ellipse {
    Circle(double radius) : Ellipse(radius, radius) {}
    void Draw() const;
};

struct Polygon {
    Color GetColor() const;
};

static_assert(Shape<Circle>);
static_assert(Shape<Ellipse>);
//static_assert(Shape<Polygon>);
```

Example: Shapes (Concepts) (cont.)

```cpp
void DrawShape(Shape auto const& s) {
    s.Draw();
}
```

```cpp
Circle c1(9);
DrawShape(c1);
Rectangle r1(10, 4);
DrawShape(r1);
```

```cpp
template<size_t Index = 0, Shape... Ts>
constexpr void DrawAll(std::tuple<Ts...> const& shapes) {
    if constexpr (Index == sizeof...(Ts)) {
        return;
    }
    else {
        DrawShape(std::get<Index>(shapes));
        DrawAll<Index + 1>(shapes);
    }
}
```

```cpp
std::tuple shapes{ c1, r1, Ellipse(7, 12), Circle(20), Rectangle(5, 6) };
DrawAll(shapes);

auto shapes2 = std::tuple_cat(shapes, std::make_tuple(Ellipse(100, 10)));
DrawAll(shapes2);
```

Concepts Guidelines

- Use standard library concepts whenever possible
- Use concepts as template parameters as opposed to `typename` and `requires`
- Defining concepts is mostly useful when writing generic libraries
- Defining new concepts is harder than it looks

Ranges

Ranges

- Ranges are concepts defined in the `std::ranges` namespace
  - Represent a set of objects that can be iterated over
- The standard containers are ranges
- Most algorithms are overloaded to accept ranges

```
using namespace std;
vector numbers{ 12, 33, 8, 55, 101, 87, 6, 16, 87, 34 };

// standard sort
sort(begin(numbers), end(numbers));

// sort a range
ranges::sort(numbers);
```

Views

- A view provides operations over ranges or other views
  - Adding functional style to C++
  - Never own their elements
  - Lazily evaluated
- Views can be chained (optionally with the | operator)
- Every view is a range
  - But not necessarily vice versa

Views Examples

```
using namespace std;
vector numbers{ 12, 33, 8, 55, 101, 87, 6, 16, 87, 34 };
```

```
void Display(std::string_view text, std::ranges::range auto& data) {
    cout << text << endl;
    for (const auto& value : data)
        cout << value << " ";
    cout << endl;
}
```

```
vector numbers{ 12, 33, 8, 55, 101, 88, 6, 16, 87, 34 };

Display("Original", numbers);
ranges::sort(numbers);
Display("Sorted", numbers);

auto result = numbers | views::filter([](auto n) { return n % 2 == 0; });
Display("Even numbers", result);
auto result2 = result | views::transform([](const auto& value) { return value * value; }) | views::reverse;
Display("Even numbers, squared and reversed", result2);
```

```
Original
12 33 8 55 101 88 6 16 87 34
Sorted
6 8 12 16 33 34 55 87 88 101
Even numbers
6 8 12 16 34 88
Even numbers, squared and reversed
7744 1156 256 144 64 36
```

Range Adapters

- Provide functions for creating views from ranges

Range adapter function	Range adapter type	Description
`views::all`	`all_t`	Creates a view from all elements in a range
`views::filter`	`filter_view`	Filters elements based on a predicate
`views::transform`	`transform_view`	Applies a callable to every element in the range
`views::take`	`take_view`	Creates a view with the first *n* elements
`views::take_while`	`take_while_view`	Creates a view with the initial elements of the range as long as a predicate returns `true`
`views::drop`	`drop_view`	Creates a view by dropping the first *n* elements from the range
`views::drop_while`	`drop_while_view`	Creates a view by dropping initial elements until a predicate returns `false`
`views::reverse`	`reverse_view`	Creates a view that reverses the elements in the range (iterates backwards)

Chaining Views

- View chaining can be done by making function calls or using the pipe | operator as syntactic sugar

```
vector numbers{ 12, 33, 8, 55, 101, 88, 6, 16, 87, 34 };
ranges::sort(numbers);
```

```
auto result = numbers
    | views::filter([](int n) { return n % 2 == 1; })
    | views::take_while([](int n) { return n < 100; });
Display("Chaining with operator |", result);
```

```
auto result2 = views::take_while(
    views::filter(numbers,
        [](int n) { return n % 2 == 1; }),
            [](int n) { return n < 100; });
Display("With function calls", result2);
```

More Examples

```
struct ProcessInfo {
    std::wstring ProcessName;
    uint32_t Threads;
    uint32_t Id;
    uint32_t Session;
    uint32_t Handles;
    int64_t CpuTime;
};

std::wostream& operator<<(std::wostream& out, ProcessInfo const& pi) {
    out << std::format(L"Name: {} PID: {} Threads: {} Session: {} Handles: {}",
        pi.ProcessName, pi.Id, pi.Threads, pi.Session, pi.Handles);
    return out;
}

void Display(std::wstring_view text, std::ranges::range auto& data) {
    wcout << text << endl;
    for (const auto& value : data)
        wcout << value << endl;
    wcout << endl;
}

std::vector<ProcessInfo> EnumProcesses();
```

More Examples (cont.)

```
auto processes = EnumProcesses();
Display(L"All processes", processes);

// a filtered view
auto sessionNonZero = processes | views::filter(
    [](auto const& pi) { return pi.Session > 0; });
Display(L"Session non zero processes", sessionNonZero);

// create container based on a view
auto sessionNonZeroProcesses = vector(begin(sessionNonZero), end(sessionNonZero));

// sort by process name
ranges::sort(sessionNonZeroProcesses, {}, [](auto const& p) { return p.ProcessName; });
Display(L"Session non-zero processes - Sorted", sessionNonZeroProcesses);

auto result = processes
    | views::filter([](auto const& pi) { return pi.Threads > 20; })
    | views::transform([](auto const& p) { return SimpleProcess{ p.Id, p.ProcessName, p.Threads }; });

auto sp = vector(begin(result), end(result));
ranges::sort(sp, {}, [](auto const& p) { return p.Threads; });
auto result2 = sp | views::reverse | views::take(10);
Display(L"Top processes with most threads (at least 21)", result2);
```

Range Factories

- Generate views producing values on demand

Type	Function	Description
`empty_view`	`empty`	Creates an empty view
`single_view`	`single`	Creates a view with a single element
`iota_view`	`iota`	Creates a view that generates a potentially infinite sequence of integers on demand
`basic_istream_view`	`istream_view`	Creates a view that retrieves items from operator << of an input stream

```
auto result = views::iota(1, 100)        // lazily generates numbers from 1 to 100
    | views::transform([](int n) { return n * 2; })
    | views::filter([](int n) { return n % 3 == 0; });
```

```
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96 102 108 114 120 126 132 138 144 150 156 162
168 174 180 186 192 198
```

Coroutines

Coroutines: Motivation

- Example: Calculating prime numbers

```cpp
std::vector<int> CalcPrimes(int first, int last) {
    std::vector<int> primes;

    for (int i = first; i <= last; i++)
        if (IsPrime(i))
            primes.push_back(i);
    return primes;
}
```

```cpp
for (auto n : CalcPrimes(3, 1000)) {
    std::cout << n << std::endl;
}
```

```cpp
for (auto n : CalcPrimes(3, 10000000)) {
    std::cout << n << std::endl;
     // imagine a more complex, runtime condition for exit
    if (n > 500)
        break;
}
```

- What if we want to exit the loop prematurely?

Solution: Use a Coroutine

```cpp
#include <experimental/generator>

std::experimental::generator<int> CalcPrimes2(int first, int last) {
    for (int i = first; i <= last; i++)
        if (IsPrime(i))
            co_yield i;
}
```

```cpp
for (auto n : CalcPrimes2(3, 100000000)) {
    std::cout << n << std::endl;
    // imagine a more complex, runtime condition for exit
    if (n > 500)
        break;
}
```

- Problem solved!

What is a Coroutine?

- Technically, any function having one of the following keywords in its body
  - `co_await`, `co_return`, `co_yield`
- Conceptually
  - A “resumable” function
  - Stackless (state not usually stored on the stack)
- Within a coroutine
  - Use of `return` is illegal
  - Constructors, destructors, `constexpr` and `consteval` functions, functions with variadic arguments – these cannot be coroutines

Another Generator Example

- Fibonacci series

```
template<std::unsigned_integral T = unsigned>
generator<T> Fibonacci(unsigned count) {
    if (count == 0)
        co_return;

    co_yield 1;
    co_yield 1;

    T a = 1, b = 1;
    for (unsigned i = 0; i < count - 2; i++) {
        auto c = a + b;
        co_yield c;
        a = b;
        b = c;
    }
}
```

```
template<std::unsigned_integral T = unsigned>
void TestFib(unsigned count) {
    for (auto n : Fibonacci<T>(count)) {
        if (tolower(_getch()) == 'q')
            break;
        std::cout << n << std::endl;
    }
}
```

Example: Simple Generator

```
template<typename T>
struct generator {
    using promise_type = promise<T>;

    generator(std::coroutine_handle<promise<T>> handle) : _handle(handle) {}
    ~generator() { _handle.destroy(); }

    T value() {
        return _handle.promise()._value;
    }

    std::optional<T> next() {
        _handle.resume();
        if (_handle.done())
            return std::nullopt;

        return _handle.promise()._value;
    }

private:
    std::coroutine_handle<promise<T>> _handle;
};
```

```
template<typename T>
struct promise {
    auto get_return_object() {
        return std::coroutine_handle<promise<T>>::from_promise(*this);
    }

    auto initial_suspend() {
        return std::suspend_always();
    }

    auto final_suspend() noexcept {
        return std::suspend_always();
    }

    auto yield_value(T t) {
        _value = t;
        return std::suspend_always();
    }

    void unhandled_exception() {}
    void return_void() { }

    T _value;
};
```

Example Run with Trace

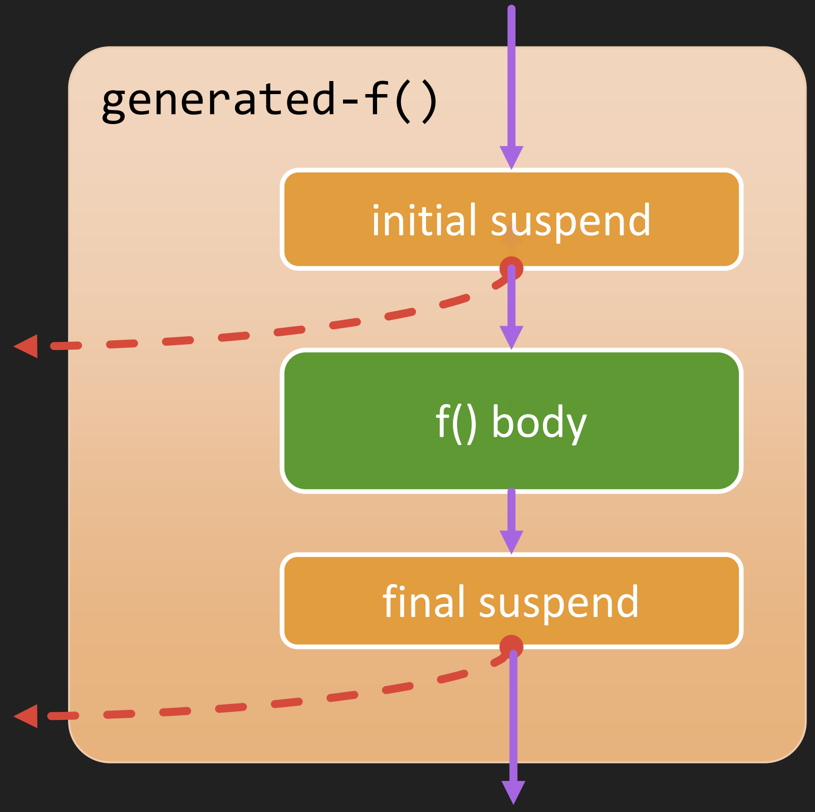
```
auto t = Fib(5);
std::optional<int> n;

while ((n = t.next()) != std::nullopt) {
    cout << *n << endl;
}
```

```
promise<int>::promise
promise<int>::get_return_object
promise<int>::initial_suspend
generator<int>::generator
generator<int>::next
promise<int>::yield_value
1
generator<int>::next
promise<int>::yield_value
1
generator<int>::next
promise<int>::yield_value
2
generator<int>::next
promise<int>::yield_value
3
generator<int>::next
promise<int>::yield_value
5
generator<int>::next
promise<int>::return_void
promise<int>::final_suspend
generator<int>::~generator
```

Coroutine Transformation

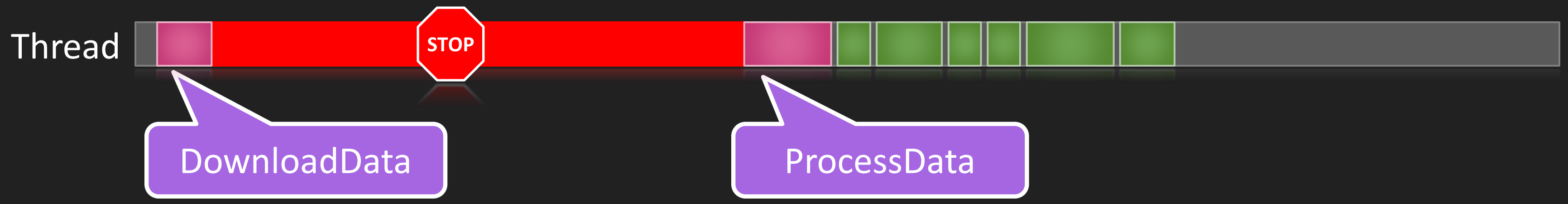
- A contract between the compiler and implementation

```
{
    Promise promise;
    co_await promise.initial_suspend();
    try {
        < function body uses co_await, co_yield, co_return >
    }
    catch(...) {
        promise.unhandled_exception();
    }
final_suspend:
    co_await promise.final_suspend();
}
```

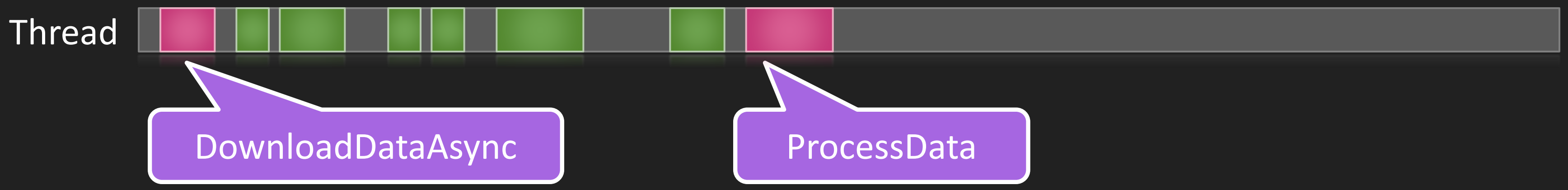


Synchronous vs. Asynchronous

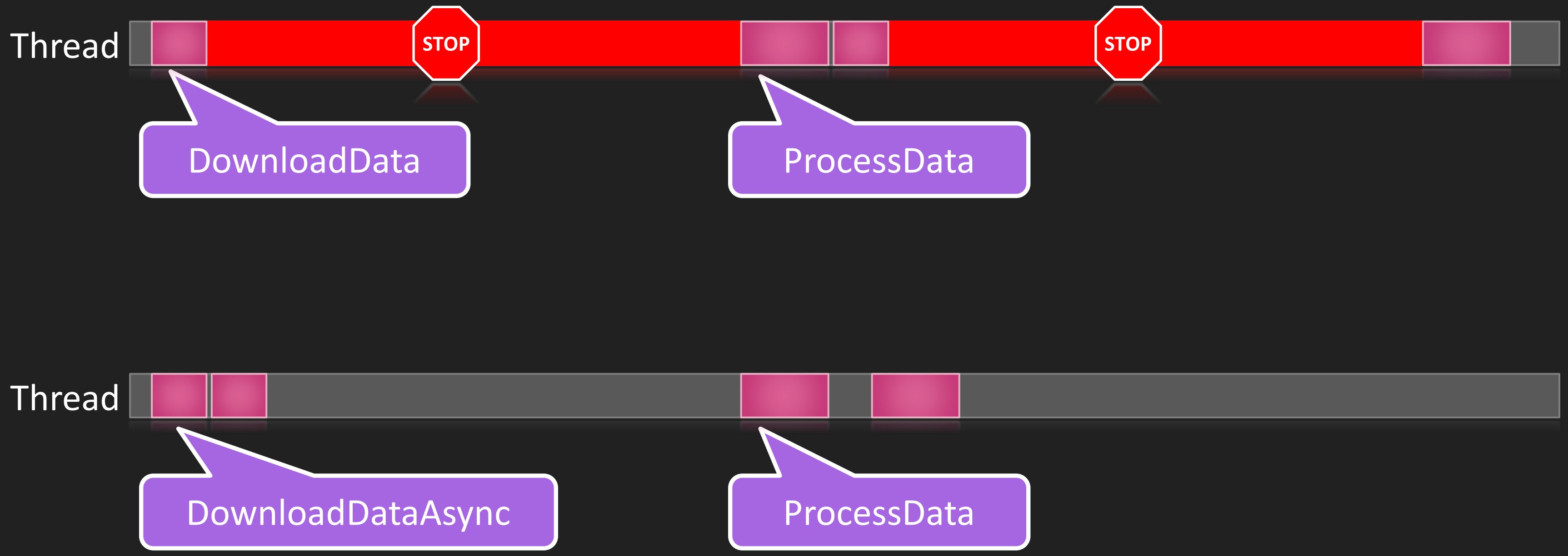
```cpp
auto data = DownloadData(...);
ProcessData(data);
```




```cpp
DownloadDataAsync(..., [](auto data) {
    ProcessData(data);
});
```



Synchronous vs. Asynchronous



Asynchrony != Threads

Asynchronous Operations

- Server scenarios
  - Key to scalability
- Client scenarios
  - Key to responsiveness
- Writing asynchronous code is hard
  - Much more complex than the “equivalent” synchronous code

Awaitables and Awaiters

- `co_await` can be called on an Awaitable
  - Searches for an awaiter
- The Awaiter is any object that satisfies an interface
  - `await_ready`, `await_suspend`, `await_resume` member functions
- If there is an `await_transform` function on the promise type, it's called to obtain an awaiter
- Otherwise, if the `co_await` operator exists on the awaitable, it's called to get an awaiter
- Otherwise, the awaitable is the awaiter

Awaitables in the Standard Library

- Awaiter member functions
  - `await_ready` – determines if the operation completed synchronously
  - `await_suspend` – schedules the continuation for execution
  - `await_resume` – returns the result of the `co_await`
- Awaiters in the standard library
  - `suspend_always` – suspends the execution always
  - `suspend_never` – never suspends
  - Convenient as a basis for custom awaiters or as return values in some cases

Example: Asynchronous Delay

```
struct DelayAwaiter : std::suspend_always {
    DelayAwaiter(chrono::milliseconds ms) : _ms(ms) {}

    void await_suspend(std::coroutine_handle<> coro) {
        thread t([coro](auto ms) {
            std::this_thread::sleep_for(ms);
            coro();
            }, _ms);
        t.detach();
    }

    chrono::milliseconds _ms;
};
```

```
struct Delay {
    Delay(chrono::milliseconds ms) : _ms(ms) {}

    auto operator co_await() {
        return DelayAwaiter(_ms);
    }

    chrono::milliseconds _ms;
};
```

```
task TestDelay() {
    for (int i = 0; i < 10; i++) {
        co_await Delay(1s);
        cout << i << endl;
    }
}
```

Example: Asynchronous I/O

```cpp
struct AsyncIoAwaiter : suspend_always {
    AsyncIoAwaiter(HANDLE hFile, DWORD offset, void* buffer, DWORD size) : _hFile(hFile), _buffer(buffer), _size(size) {
        _ov.Offset = offset;
        _ov.hEvent = ::CreateEvent(nullptr, FALSE, FALSE, nullptr);
    }

    void await_suspend(std::coroutine_handle<> handle) {
        ::ReadFile(_hFile, _buffer, _size, nullptr, &_ov);
        ::RegisterWaitForSingleObject(&_hReg, _ov.hEvent, [](auto param, auto) {
            auto coro = std::coroutine_handle<>::from_address(param);
            coro();
            }, handle.address(), INFINITE, WT_EXECUTEDEFAULT);
    };

    DWORD await_resume() noexcept {
        ::UnregisterWait(_hReg);
        DWORD bytes;
        ::GetOverlappedResult(_hFile, &_ov, &bytes, TRUE);
        ::CloseHandle(_ov.hEvent);
        return bytes;
    }

    OVERLAPPED _ov{};
    HANDLE _hFile;
    void* _buffer;
    DWORD _size;
    HANDLE _hReg;
};
```

```cpp
auto ReadFileAsync(HANDLE hFile, DWORD offset, void* buffer, DWORD size) {
    return AsyncIoAwaiter(hFile, offset, buffer, size);
}
```

```cpp
task<DWORD> TestIo(PCWSTR path) {
    auto hFile = ::CreateFile(path, GENERIC_READ, FILE_SHARE_READ, 0,
        OPEN_EXISTING, FILE_FLAG_OVERLAPPED, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        co_return 0;

    auto size = ::GetFileSize(hFile, nullptr);
    cout << "File size: " << size << endl;
    auto buffer = std::make_unique<BYTE[]>(size);
    auto bytes = co_await ReadFileAsync(hFile, 0, buffer.get(), size);
    cout << "Bytes read: " << bytes << endl;
    co_return bytes;
}
```

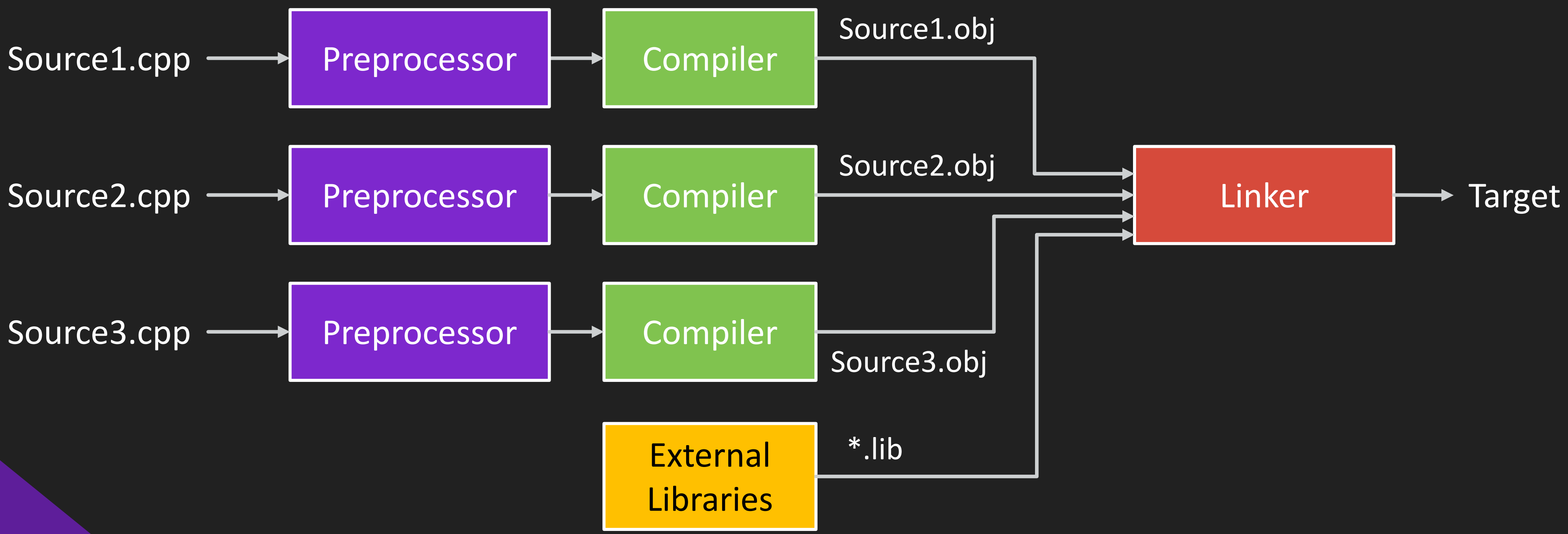
Coroutine Guidelines

- C++ 20 does not provide common coroutine implementations
- Use existing implementations unless you need something unique
  - Example: https://github.com/lewissbaker/cppcoro
- Usage of coroutines can simplify code significantly

Modules

Introduction

- The standard way of bringing in functionality from other code is to `#include` header files
- Recap: The build process



Modules Benefits

- Shorter compilation times – each module only needs to be compiled once
  - Similar to precompiled headers
- Order of imported modules is not imported
- Macros are never leak from a module
- Express logical structure rather than physical structure
- No need to separate header (interface) and implementation

A Simple Module

```cpp
// Math.ixx

export module Math;

namespace Math {

    export template<typename T>
        T Add(T a, T b) {
        return a + b;
    }

}
```

```cpp
import Math;

int main() {
    using std::cout, std::endl;

    using namespace Math;

    cout << Math::Add(10, 3) << endl;
    Point p1{ 3, 4 };
    Point p2{ 10, -2 };
    cout << Distance(p1, p2) << endl;

    return 0;
}
```

A More Complex Example

```cpp
module;

#include <cmath>

export module Math.Simple;

#define SQR(x) ((x)*(x))

namespace Math {
    export struct Point {
        double x;
        double y;
    };

    double DistanceSquared(Point const& p1, Point const& p2);

    export template<typename T>
        T Add(T a, T b) {
        return a + b;
    }

    export double Distance(Point const& p1, Point const& p2) {
        return std::sqrt(DistanceSquared(p1, p2));
    }

    double DistanceSquared(Point const& p1, Point const& p2) {
        return SQR(p1.x - p2.x) + SQR(p1.y - p2.y);
    }
}
```

Submodules and Partitions

- Modules can be imported and re-exported

```
export module Math;

export import Math.Simple;
export import Math.Trig;
```

- Modules can be further divided to partitions

```
export module Math:Simple;

export {…}
```

```
export module Math:Trig;

export {…}
```

```
export module Math;

export import :Simple;      // module name is optional
export import :Trig;
```

```
Import Math;

//…
```

Module Linkage

- C++ always had two linkage types
  - Internal linkage
    - `static` members and code in anonymous namespaces
    - Only visible in that translation unit
  - External linkage
    - Classes, functions, etc. (not `static`) accessible in other translation units
- Module linkage
  - Only accessible within the module

Importing Headers

- Header files can be imported with import
- Standard library headers are guaranteed to be imported

```
import <vector>;
import <iostream>;
import "myheader.h";
```

- Other headers have no such guarantee, and it's currently compiler dependent

Summary

- C++ 20 is a big release!
- Demos of this talk: https://github.com/zodiacon/CoreCpp21Demos
- Integrate C++ 20 features in existing code bases
  - Start small: concepts and ranges
  - Big benefit but more complex: coroutines
  - Difficult: modules

Thank you!

Q & A