jbi training

C++ 20 and 23

Programming C++ 20 and C++ 23

Contents Slide

Programming C++20

About This Course

- This course overviews some recent C++ 20 features
- In the development of the C++ language there have been some significant standard releases and some less significant
- The major standard releases (since 1998) are:
  - C++ 11 (R value reference and move)
  - C++ 20

IDE Support

- This course overviews some recent C++ features
  - Visual Studio and GNU History:
    - Visual Studio 2008 – C++98
    - Visual Studio 2010 – C++0x (partial)
    - Visual Studio 2012/2013 – C++11 (partial)
    - Visual Studio 2015 – C++14 (partial)
    - Visual Studio 2017 – C++17 (partial)
    - Visual Studio 2019 – C++20 (partial)
    - Visual Studio 2022 – C++23 (partial?)
    - GNU C++ (depends on version!)
  - The support also depends upon the update

Compiler Support

- The detailed support for C++ is a moving target, see the page on 'cppreference'

https://en.cppreference.com/w/cpp/compiler_support

  - Within Visual Studio use the project properties language options to select C++20
    - For some features need to choose latest!
  - For GCC compilers using options:

-std=c++20 -fcoroutines — Additional flag for coroutines

Feature Macros

- Many defines which can be checked to determine if the feature is supported:

```
#include <version>
int main()
{
 #ifdef __cpp_lib_barrier
 std::cout <<  "Barrier supported" << std::endl;
 #endif
  #ifdef __cpp_lib_bind_front
 std::cout <<  "Bind front supported" << std::endl;
 #endif
 #ifdef __cpp_using_enum
 std::cout <<  "Using enum supported" << std::endl;
 #endif
 …
```

If project property set for language C++20

```
Barrier supported
Bind front supported
Using enum supported
…
```


C++17 Features

- The C++17 standard include a number of features considered useful:
  - Nested Namespaces
  - noexcept part of type system
  - Template Argument Deduction
  - Initialisation within if and switch!
  - Constexpr If
  - Parallel algorithms

C++17 Functions and Types

- The C++17 include a number of useful functions and types:
  - Structured Bindings
  - std::invoke, std::apply and std::make_from_tuple
  - Inline Static Variables
  - Copy Elision and RVO
  - Fold Expressions
  - std::string_view
  - std::optional, std::variant
  - std::byte
  - File System

C++20 Benefits

- The benefits of moving to C++ 20 are many and fall into a number categories:
  - Efficiency
    - Constexpr, consteval and constinit
    - Likely and Unlikely attributes
  - Compile time diagnostics
    - Concepts
  - Code simplification
    - Abbreviated template syntax
    - Lambdas
    - Operator definition (spaceship)
    - Conditional explicit

Template Features

- This section is an overview of the new template related features:
  - Abbreviated Template Syntax
  - Lambda Template Syntax
  - Template Parameters
  - Class Non-type Template Parameters
  - Non-type Parameter Pointer to Function
  - Variadic Templates

Abbreviated Templates Syntax

- C++20 supports a simplified template syntax
- Old style template function:

```
template<typename T>
T old_doubler(T t) { return t + t; }
```

- Simplified variation:

```
auto doubler(auto n) { return n + n; }
```

- Can also use concepts:

```
auto doubler_numeric(Numeric auto n) { return n + n; }
```

Multiple Template Arguments

- C++20 supports a simplified template syntax
- Old style template function:

```
template<typename T>
T my_max(T x, T y) { return x < y ? y : x; }
```

- What if the calling types are different? How do we define the return type?

```
template<typename T, typename U>
auto my_max(T x, U y) -> decltype(x+y)
                    { return x < y ? y : x; }
```

Expression to allow deduction of type?

- Simplified:

```
auto amy_max(auto x, auto y) { return x < y ? y : x;}
```


Lambda Template Syntax

- A generic lambda does not easily convey type information (use of auto):

```
auto nested_type = [](auto t)
{
  return typeid(decltype(t)::value_type).name();
};
std::cout << nested_type(std::vector{1, 2}) << std::endl;
```

Requires use of decltype to deduce type

```
auto nested_type = []<typename T>(T t)
{
  return typeid(T::value_type).name();
};
std::cout << nested_type(std::vector{1, 2}) << std::endl;
```

Type parameter, can also use concepts

Template Parameters

- C++17 introduced the ability to have non-type template parameter deduction
  - Non-type parameters essentially restricted to integral types
- C++ 20 extends non-type parameters to floating point types and class types
  - Class types should be structural types (simple!)
    - Class and base classes have public members
      - Members are structural types!

C++17 Template Non-type Deduction

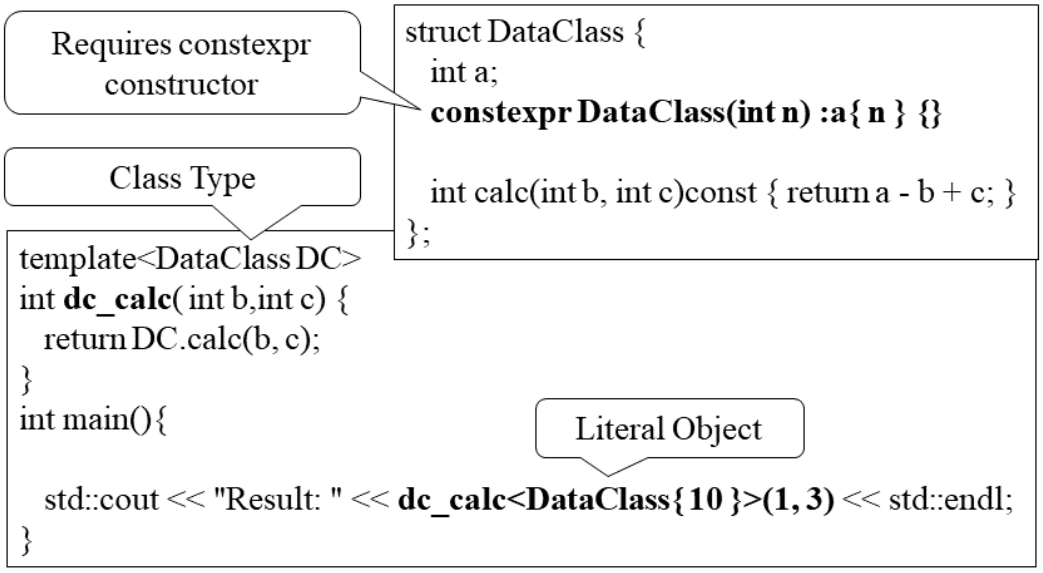
```
int main()
{
    SomeData<int,7u> sd;
    auto a = sd.getVal(4);
    std::cout << a << std::endl;
}
```

```
#include <iostream>
#include <numeric>
template <typename T, auto N>
class SomeData{
    T data[N];
public:
   SomeData()
   {
      std::iota(std::begin(data),std::end(data),1);
   }
   auto getVal(decltype(N) i) const { return data[i]; }
};
```

Deduce non-type parameter

Deduce parameter

Class Non-type Template Parameter



Lambda as Non-type Parameter

- Non-type parameter supports pointer to function
  - This can now accept a lambda expression

```
using FunPtr = double(double);

template<FunPtr F >
class Command {
public:
  double execute(double d) { return F(d); }
};
```

```
Command<[](double d)->double {return d * 7; }>
```

Constexpr Lambdas

- Non-type parameter supports pointer to function
  - This can now accept a lambda expression

```
using FunPtr = double(double);

template<FunPtr F >
class Command {
public:
  constexpr double execute(double d) { return F(d); }
};
```

```
Command<[](double d) constexpr ->double {return d * 7; }>
```


Variadic Templates

- C++20 supports capture of variadic arguments within a lambda capture:

```
template<typename... Args>          Variadic type pack

auto lambda_fun_outer = [](Args&&... args) {     Variadic argument pack

                                    Capture and forwarding of argument pack

        return ([... args = std::forward<Args>(args)]() {return (args + ...); })();
};
```

Concepts

- This section gives and overview of motivation and using concepts:
  - Concepts Introduction
  - Named Requirements
  - Named Requirements and Type Traits
  - Template Problems
  - Templates and SFINAE
  - Templates and concepts

Concepts Introduction

- This section gives an introduction to the motivation for the use of concepts
- The idea of Named Requirements will be introduced before considering some of the issues with working with templates
- Existing solutions will be considered using pre C++20 features

Named Requirements

- Named requirements have been used within the C++ Standard for many years
  - They define requirements on types for template instantiation
  - Not meeting these requirements can result in complex compiler messages
  - Some of these requirements can be checked for using type traits
- It has been largely down to the developer to meet these requirements

Named Requirements

- The expected behaviour of the Standard Library is largely defined in terms of ‘Named Requirements’
- There are now many named requirements, which fall into a number of categories:
  - Basic requirements
  - Container
  - Iterator
  - Concurrency

Basic Named Requirements

- These 'Named Requirements' provide for common features:

Requirement	Description (fairly self explanatory!)
DefaultConstructible	Object can be default constructed
CopyConstructible	lvalue constructable
MoveConstructible	rvalue constructable
CopyAssignable	Assignable from lvalue
MoveConstructible	Assignable from rvalue
Destructible	Object can be destroyed

Named Requirements Checking

- Named requirements are capable of being checked through the use of type traits, e.g. DefaultConstructible:
  - The three type traits below express a number of variations on default constructable:

Type trait (C++11)	Addition in C++17
is_default_constructible	is_default_constructible_v
is_trivially_default_constructible	is_trivially_default_constructible_v
is_nothrow_default_constructible	is_nothrow_default_constructible_v

type traits

- Type traits were introduced with C++11
  - There are a wide range type traits
  - Additional type traits add with C++17 to simplify usage, e.g.

```
template<typename T>
        inline constexpr bool is_default_constructible_v =
                is_default_constructible<T>::value;
```

  - These are now being used to formalize 'Named Requirements' within the C++20 standard

type_traits

- Type_traits allow determining characteristics for types/functions etc.

Type traits	**value**
is_abstract	True if abstract class
is_array	True if array
is_pod	True if plain old data (trivial and standard layout)
is_assignable	True if assignment possible between types
is_base_of	True if base of another type
is_class	True if class (or struct)
is_function	True if type is function

type_traits (trivial)

- Type_traits allow determining characteristics for types/functions etc.

Type traits
is_trivial
is_trivially_copyable
is_trivially_assignable
is_trivially_copy_assignable

- Each characteristic has a number of requirements
  - Largely regarding only implicitly supplied constructors; assignment operators and destructor
  - Also types not containing virtual member functions

Named Requirements - Categories

- There are now many named requirements:

Category	Examples
Basic	DefaultConstructable, CopyConstructable, CopyAssignable
Type	TriviallyCopyable, PODType
Container	Container, SequencialContainer, AssociativeContainer
ContainerElement	DefaultInsertable, CopyInsertable
Iterator	LegacyIterator, LegacyBidirectionalIterator
Stream I/O	FormattedInputFunction, FormattedOutputFunction
Concurrency	Lockable, SharedLockable, Mutex
Library	EqualityComparable, Allocator, Swappable
Other	NumericType, LiteralType, BitmaskType
Random Number	SeedSequence, RandomNumberEngine

Name Requirements and Type Traits

- Some of the Named Requirements are associated with type traits
  - There are many predefined type traits within the header file ‘type_traits’
- These type traits can be used in conjunction with ‘static_assert’ to give a compiler message:

```
static_assert(std::is_constructible<A>::value,
              "A not construcible!");
```

C++ 11

```
static_assert(std::is_constructible_v<A>,
              "A not construcible!");
```

C++ 17

Multiple Type Traits

- Multiple type traits can be applied using conjuction or disjunction
  - Logical ‘and’:

```
std::conjunction_v<...>
```

  - Logical ‘or’:

```
std::disjunction_v<...>
```

- These type traits can be useful, especially within variadic templates

Template Problems

- A simple template function could be instantiated with a wide range of types
  - Some types may not be appropriate!

Template instantiated for any type with a '<' operator

Works for 'int', but would not work as expected for const char *

```
template<typename T> inline T my_max(T x, T y)
{
  return x < y ? y : x;
}
int main()
{
  int a = 65, b = 82, result;

  result = my_max(a, b);
  …
}
```


Use of static_assert

- Static assert can be used cause compiler error of unexpected type:

```
template<typename T> inline T my_max(T x, T y)
{
  static_assert(std::is_integral_v<T> ||
                std::is_floating_point_v<T>, "Not a number!");
  return x < y ? y : x;
}

int main()
{
    const char* pch = "Hello", *pch2 = "Greetings", *result;

    result = my_max(pch, pch2);
}
```

Multiple type traits

Compiler error due to attempted use of const char *

Templates and SFINAE

- A different alternative is to use SFINAE to prevent instantiation!

```
template<typename T> inline std::enable_if_t<
        std::is_integral_v<T> ||
        std::is_floating_point_v<T>,T>
        my_max(T x, T y)
{
  return x < y ? y : x;
}
```

Multiple type traits

- Alternative versions can be provided again using ‘enable_if’

Templates and SFINAE continued…

- Alternative versions of the template function can be provided where the types statisfy different traits!

```
template<typename T> inline std::enable_if_t<
        std::is_convertible_v<T,const char*>, T>
        my_max(T x, T y)
{
  return std::string(x) < std::string(y) ? y : x;
}
```

Templates and SFINAE continued…

- The previous slide showed the use of 'enable_if' in the return type, but it could appear in the template parameters:

```
template<typename T, typename =
        std::enable_if_t<std::is_convertible_v<T, const char*>>>
inline T my_max(T x, T y)
{
return std::string(x) < std::string(y) ? y : x;
}
```

- However, this approach is not ideal!

Templates and Concepts

- The C++ 20 alternative is to use concepts
  - A max template function just for floating point types:

Concept to required type

```
template<std::floating_point T>
inline T my_max(T x, T y)
{
  return x < y ? y : x;
}
```

Constraints and concepts

- This section gives an overview of Constraints and concepts:
  - Constraints using requires
  - Concepts
  - Predefined Concepts
  - Defining concepts
  - Use of Concepts
  - Concepts and requires
  - Defining Concepts

Constraints using requires

- Constraints can be applied to template parameters using requires expressions:

```
template<typename T>
requires(std::is_integral_v<T> || std::is_floating_point_v<T>)
inline T a_max(T x, T y) {
  return x < y ? y : x;
}
```

Requires expression

Disjunction or conjunction (&&) can be used between type traits

Concepts

- C++ templates provide a very flexible means of implementing generic programming
  - Instantiation of templates, with inappropriate type, can result in all manner of errors!
- The use of ‘concepts’ can help with the definition of templates in order to restrict instantiation to viable options

Predefined Concepts

- There are many predefined concepts within the ‘concept’ header file:

Concept categories	**Examples**
Core language	same_as, integral, floating_point, destructible
Comparison	equality_comparable, totally_ordered
Object	movable, copyable, regular
Callable	invocable, regular_invocable, predicate,
Iterator	input_iterator, output_iterator, random_access_iterator
Algorithm	indirect_unary_predicate, indirect_binary_predicate
Ranges	range, view, input_range, random_access_range

Useful Concepts

- These are a few concepts in more details:

Concept	Description
std::same_as<T>	Satisfied if the type is the same as T
std::convertible_to<T>	Satisfied if the type can be converted to T
std::invocable<F, …>	Satisfied if the type is callable with the arguments
std::regular_invocable<F,…>	Std::invocable, plus, does not change function or arguments
std::predicate<F, …>	Satisfied if the type is regular_invocable and boolean-testable
std::integral	Satisfied if the type is integral
std::floating_point	Satisfied if the type is floating point

std::invocable<>

- Require to put an explicit constraint on callable passed to template:

```
template <typename Func, typename T>
requires std::invocable<Func, T, T>  // constraint
T operate(Func func,T a, T b) {
  return func(a,b);
}
```

```
int main() {
  auto mult = [](int x, int y) {return x*y;};

  auto result = operate(mult, 5, 7);

  std::cout << result << std::endl;
  return 0;
}
```

Defining concepts

- Concepts can be defined using template syntax:

```
template<typename T>
concept Integral = std::is_integral<T>::value;
template<typename T>
concept Floating = std::is_floating_point<T>::value;
```

Concept defined in terms of type traits

Use of concept to provide constraint

```
template<Integral T>void do_output(T t) {
        std::cout << "Integral: " << t << std::endl;
}
template<Floating T>void do_output(T t) {
        std::cout << "Floating: " << t << std::endl;
}
```


Use of Concepts

- The previous slide used concepts in place of typename within template definitions
- Another way of using a concept is as a requirement using requires:

```
template<typename T>void do_output2(T t)requires Integral<T> {
        std::cout << t << std::endl;
}
template<typename T>void do_output2(T t)requires Floating<T> {
        std::cout << t << std::endl;
}
```

Composite Concepts

- A composite concept can be defined using multiple type_traits or other concepts:

```
template<typename T>
concept Numeric = std::is_integral_v<T>
                  || std::is_floating_point_v<T>;
```

Disjunction

```
template<Numeric T>
inline T square(T n) { return n * n; }
int main()
{
  int a = 2;

  auto result = square(a);
  …
}
```

Concept restricts type to be used with 'square' function

Alternatively:

```
template<typename T>
concept Numeric = Integral<T>
                  || Floating<T>;
```


Concepts and Requires

- An alternative way of using constraints is by using requires:

```
template<typename T>
inline T square(T n) requires Numeric<T> { return n * n; }
```

Applying concept through requires

```
int main()
{
  int a = 2;

  auto result = square(a);

  …
}
```

Defining concepts

- Concepts can separate out requirements, e.g. incrementing:

```
template<typename It >
concept MyIncrementor = requires(It it) {
        {++it} -> std::same_as<It&>;
        {it++} -> std::same_as<It>;
};
```

Support both prefix and postfix incrementation, with return specification

- This concept can be used in its own right or combined with others to form more complex concepts

Defining concepts continued…

- Concepts can separate out requirements, e.g. dereferencing:

```
template<typename It >
concept MyDerefer = requires(It it) {

        {*it}->std::convertible_to<typename It::reference>;
};
```

Requires dereference operator *

Concept for return type

Compose concepts

- The two previous concepts of MyIncrementor and MyDerefer:

```
template<typename It>
concept MyIterator = MyIncrementor<It> && MyDerefer<It>;
```

```
template<MyIterator It>
void use_iterators(It begin, It end) {
  for (auto it = begin; it != end; ++it) {
    std::cout << *it << ", ";
  }
  std::cout << std::endl;
}
```


Full Iterator!

- A more complete implementation of an iterator concept would be of the form:

```
template<typename It >
concept MyIterator2 = requires(It it1, It it2) {
  {++it1} -> std::same_as<It&>;
  {it1++} -> std::same_as<It>;

  {*it1}->std::convertible_to<typename It::reference>;
  {it1 == it2} -> std::convertible_to<bool>;
  {it1 != it2} -> std::convertible_to<bool>;
};
```

Multiple variables

Increment

Dereference

Comparison

Interface like concept

- Concepts can be used a little like interfaces (which are available in many languages):

```
template<typename T>
concept Stringable = requires(T t) {
  { t.to_string()} -> std::same_as<std::string>;
};
```

Requirement for member function

Requirement for return type

noexcept and const

- The previous slide considered the ability to require a member function signature
- It can be useful to add additional requirements for noexcept or const:

```
template<typename T>
concept Stringable = requires(T t) {
  {std::as_const(t).to_string() } -> std::same_as<std::string>;
} && noexcept(std::declval<const T>().to_string());
```

noexcept operator

Requirement a const object

Implementing class

- A class satisfying a ‘constraint’ is defined completely independently of the ‘concepts’:

```
class PassengerDetails {
  std::string _name;
  int _weight;
public:
  PassengerDetails(const std ::string&  name, const int& weight):
          _name(name), _weight(weight){}

  std::string to_string() const { return "Name: " + _name +
           ", baggage weight: " + std::to_string(_weight); }
}
```


Using Implementing Class

- Implement template function using concept:

```
template<Stringable T>
void do_output(T t) {
  std::cout << t.to_string() << std::endl;
}
```

Tooling within some development environments show member functions available

- Calling function using an object of a class meeting requirement:

```
{
  PassengerDetails pd{ "Fred", 43 };

  do_output(pd);
}
```

Concept Operator Requirements

- Operators can be represented using concepts:

```
template<typename T>

concept Addable = requires(T n1, T n2) {
  { n1 += n2} -> std::convertible_to<T&>;
  { n1 + n2} -> std::convertible_to<T>;
};

template<Addable A>
void calc(A a1, A a2) {
  a1 += 7;

  auto result = a1 + a2;
}
```

Multiple variables

Requirement for operators

Use of concept

Implementing Operators

- The Adder concept defined satisfied by many existing types, i.e. int, std::string or a custom type:

```
class Number {
  int _val;
public:
  Number(int val):_val(val){}
  Number& operator+=(const Number& n) { _val += n._val;  return *this; }
};
inline Number operator+(const Number& lhs, const Number& rhs) {
  Number temp(lhs);
  return temp += rhs;
}
```

Span

- This section gives an overview of spans:
  - Span Introduction
  - Span Member Functions
  - Span Example
  - Subspan Example

Span Introduction

- A span represents a view onto a contiguous sequence (does not own the data)
  - Can be initialised with arrays, many sequential containers and ranges
  - Changes may be made through the span
  - Subspans can be created
  - Useful for passing arrays to functions

Span Member Functions

- There are many member functions which allow a span to be used like a container:

Member function	Description
begin, rbegin	Returns beginning or reverse beginning iterator, respectively
end, rend	Returns end or reverse end iterator, respectively
front, back	Returns values from front or back, respectively
operator[]	Index into the span (may throw out_of_range exception)
data	Returns pointer to start of contiguous data
size	Size of the span
first, last	Return spans of given size, starting at the beginning or end
subspan	Returns a span with given start point and count

Span Example

- Using span as a parameter:

```
inline int mysum(std::span<int> data) {
   int result{};
   for (size_t i = 0; i < data.size(); i++)
   {
      result += data[i];
   }
   return result;
}
int main(){
  int anarray[]{1,2,3,4,5,6,7,8,9};

  int result = mysum( anarray);
   std::cout << result << std::endl;
}
```

Take a span as a parameter

Size can be obtained from span

Index into span

'C' style array

Pass array to be viewed through a span

Subspan Example

- Subspans can be creates from a span:

```
int anarray[]{ 1,2,3,4,5,6,7,8,9 };

std::span<int> sp{ anarray };

auto first_part{ sp.subspan( 0, 4) };
auto second_part{ sp.subspan( 4, 5) };

int result_from_parts = mysum(first_part) + mysum(second_part);

int result = mysum(sp);

std::cout << result << " = " << result_from_parts << std::endl;
```

Subspan created from start and count

sum of the separate subspans

Sum from span

Same results

Ranges and Views

- This section gives an overview of Ranges and Views:
  - Ranges and Views Introduction
  - Ranges
  - Algorithms – Ranges
  - Range Factories
  - Pipelines
  - Range Adapters

Ranges and Views Introduction

- A new set of features which have been added to the standard library are ranges and views
  - Make use of a number of concepts
- The need for ranges is partly motivated by a problem with working with iterators
  - Algorithms typically require both begin and end iterator to be specified (of the same type)
  - No compiler check that begin and end refer to the same container! Thus, possible runtime error!

Ranges

- Ranges are generalization to the iterator and algorithm library
- Ranges are an abstraction for:
  - Iterators - [begin, end) – Half open interval
  - Counted sequence – [start, size)
  - Conditional – [start, predicate)
  - Unbounded – [start, …)

Range Abstractions

- Examples of abstractions:

Abstraction	Adapter
Counted	counted
Conditional	take_while
Unbounded	iota_view

Ranges

- The std::ranges namespace has many functions, including algorithms:

Function	Description
begin, cbegin, rbegin, crbegin	Returns the various begin iterator
end, cend, rend, crend	Returns the various end iterator (sentinel!)
size, ssize	Returns unsigned and signed size
empty	Returns Boolean indicating if range is empty
data, cdata	Returns pointer to beginning of ‘contiguous’ range

Constrained Algorithms

- Constrained algorithms are overloads of algorithms which take a ‘range’ instead of the traditional pair of iterators, i.e. single range or iterator sentinel pair
  - The ‘range’ concept provides for the begin and end iterator
  - The object implementing ‘range’ can simply be passed as one parameter

Algorithms - Ranges

- Including <algorithm> header file now provides algorithms which support the use of ranges:

```
#include <algorithm>
int main()
{
  std::vector<int> data(10);

  std::fill(data.begin(), data.end(), 6);

  std::ranges::fill(data, 4);
}
```

Traditional algorithm using iterator

New Algorithm using Ranges

- Equivalent numeric algorithms do not currently support Ranges

Algorithms - Ranges

- The previous slide showed the use of the std::ranges::fill with a range, however there is an overload which uses an iterator and 'sentinel'
  - The idea of a 'Sentinel' is an end only iterator!

```
#include <algorithm>
int main()
{
  std::vector<int> data(10);

  std::ranges::fill(data.begin(), data.end(), 6);
}
```

Works with conventional iterator

Algorithms - Projections

- Another feature of the new ranges algorithms is the optional use of projections
- Consider the find algorithm:

```
{
  std::vector data{ 2,4,6,8 };

  auto found = std::ranges::find(data, 4);

  std::cout << *found << std::endl;
}
```

The direct find works for a simple value

- What if the container stored objects and comparison was required with a member?

Algorithms - Projections

- Consider the struct:

```
struct Info {
  int val;
  Info(int val):val(val){}
};
```

```
{
 std::vector<Info> data{ 2,4,6,8 };

 auto result = std::ranges::find(data, 6, &T::val);

 std::cout << result->val << std::endl;
}
```

Projection for data member

- The ability to 'invoke' using the address of a data member is possible using the 'invoke' function (C++ 17)

Sentinel

- The problem which motivated the introduction of the concept of a sentinel was that traditional iteration requires iterators of the same type and of a known range!
  - Iterators work well with containers
- Sentinels allow a generalization of the iterator
  - Allows specialised collections or ranges can be define

Custom Sentinel Example

- Support a sequence was required which had an unusual ‘end’ requirement
  - Array of values, where -1 denotes the ‘end’
- A traditional iterator could not detect this
  - Therefore define a sentinel

Illustrates minimal implementation

```
struct ArraySentinel {};

template<typename T>
constexpr bool operator==( T ptr, ArraySentinel) noexcept {
       return *ptr == -1;  // Sentinel condition
}
```


Example Sentinel Usage

- Sentinel used like an ‘end’ iterator:

Sentinel value

```
std::array<int, 6> numbers = { 1, 2, 3, 4, 5, -1 };

auto sentinel = ArraySentinel{}; // Create an instance

for (auto it = std::ranges::begin(numbers);
             it != sentinel; ++it) {
  std::cout << *it << " ";
}
std::cout << std::endl;
std::ranges::copy(std::ranges::cbegin(numbers), sentinel,
             std::ostream_iterator<int>(std::cout, ", "));
std::cout << std::endl;
```

!= works!

Used in ranges algorithm

Ranges Concepts

- Many concepts are defined for working with ranges (below are a number of these concepts):

Concept	Description
range	Requires a begin and end member function
sized_range	Known sized range
view	View is a range with copy, move and assignment
input_range, output_range, etc.	Various ranges corresponding to the various iterator categories
viewable_range	Range can be converted to a view

An Aside – 'iota'

- The iota function is used to create an increasing sequence
- The numeric algorithm may be used as:

```
#include <numeric>
```

```
std::vector<int> data(10);
std::iota(data.begin(), data.end(), 1);
```

Populates the vector with the values 1 through to 10

- Named after the APL function 'ι' (index generator)
- The term 'iota' now appears in various other places involved with generation of sequences

iota views

- Variations on use of ‘iota’

```
for (int i : std::ranges::iota_view{ 0, 10 }) {
  std::cout << i << ' ';
}
std::cout << std::endl;

for (int i : std::views::iota(0, 10)) {
  std::cout << i << ' ';
}
std::cout << std::endl;

std::vector<int> data{7, 6, 5, 4, 3};
for (auto it : std::views::iota(data.begin(), data.end())) {
  std::cout << *it << ' ';
}
std::cout << std::endl;
```

Create a view over 0 through to 9

view over 0 through to 9

Increment iterator from beginning to !end

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
7 6 5 4 3

Range Factories

- Various range factories are available:

Range factory	Description
ranges::empty_view views::empty	View contains no elements
ranges::single_view views::single	View with a single element
ranges::iota_view views::iota	View with increasing elements
ranges::basic_istream_view views::istream	Elements input from ‘>>’ operator

istream_view from keyboard

- Input from many source can be obtained using ‘istream_view’:

```
auto results = std::ranges::istream_view<std::string>(std::cin);

std::ostream_iterator<std::string> osi(std::cout, " ");

std::ranges::copy(results, osi);
```

- Keyboard input will continue until ‘Ctrl Z’ on Windows and ‘Ctrl D’ on Linux

istream_view from std::string

- Values can be read from a string:

```
std::istringstream msg{ "12.3 6.3 7.3" };
auto results = std::ranges::istream_view<double>(msg);

std::ostream_iterator<double> osi(std::cout, " - ");

std::ranges::copy(results, osi);
```

- Output: **12.3 - 6.3 - 7.3 -**

Dangling

- Care needs to be taken when using ranges, to avoid ‘dangling’:

```
int main()
{
  auto found = std::ranges::find(get_data(), 3);

  std::cout << *found << std::endl;
}
```

Found an iterator?

get_data() returns a temporary

*found causes compilation error due to found not having an operator *

Dangling Solution

- To solve the previous problem create an object with appropriate lifetime:

```
auto get_data() { return std::vector{ 1,2,3,4,5 }; }

int main()
{
  auto data = get_data();
  auto found = std::ranges::find( data, 3);

   std::cout << *found << std::endl;
}
```

Object with larger scope

Dereferencing now safe as ‘data’ in scope

Dangling

- Some types resolve the problem of dangling, by having borrowed ranges, such as std::string_view and std::span:

```
using std::string_literals::operator"" s;
{
  auto str{ "Greetings"s };

  auto found = std::ranges::find(std::string_view{str},'t');

  std::cout << *found << std::endl;
}
```

Temporary string_view!

This is valid as the underlying 'string' still exists!

Pipelines

- Ranges and Views can be used to create pipelines
- The use of multiple traditional algorithms will result in iterating over possible the same container(s) multiple times
- The use of Pipelines allows a single iteration whilst processing the data
  - R values cannot be used in a pipeline
  - Views can be used within a pipeline

| operator

- To achieve pipelining the overloaded or '|' operator is used
- The value(s) on the left of the operator are fed into the expression on the right of the operator

```
int data[] = {1,2,3,4,5};

auto results = data
             | std::views::filter([](auto n)
                                  {return n % 2 == 0;});
```

Range Adapter

Pipeline operator

Range Adaptors

- Range Adaptors provide various functionality
- Range Adaptors only work with viewable ranges, these include:

Adaptor	Returned view	Description
filter	filter_view	Filters elements based on a predicate
transform	transform_view	Transforms elements based on a 'callable'
take	take_view	First set of elements
take_while	take_while_view	First set of elements whilst predicate true
drop	drop_view	Skip the first set of elements
drop_while	drop_while_view	Skip the first set of elements whilst predicate true

Range Adaptors

- Range Adaptors provide various functionality
- Range Adaptors only work with viewable ranges, these include:

Adaptor	Returned view	Description
join	join_view	Flatten the sequence
reverse	reverse_view	Reverse elements
split	split_view	Split sequence of elements

Pipelining (lazy evaluation)

- Sequences of data can be processed by pipelining, using range adaptors:

```
auto data = std::vector<int>(20);
std::iota(data.begin(), data.end(), 1);

auto results = data
        | std::views::filter([](auto n) { return n % 2 != 0; });

std::cout << "Odd numbers: ";

for (const auto& n : results) {
        std::cout << n << ", ";
}
```

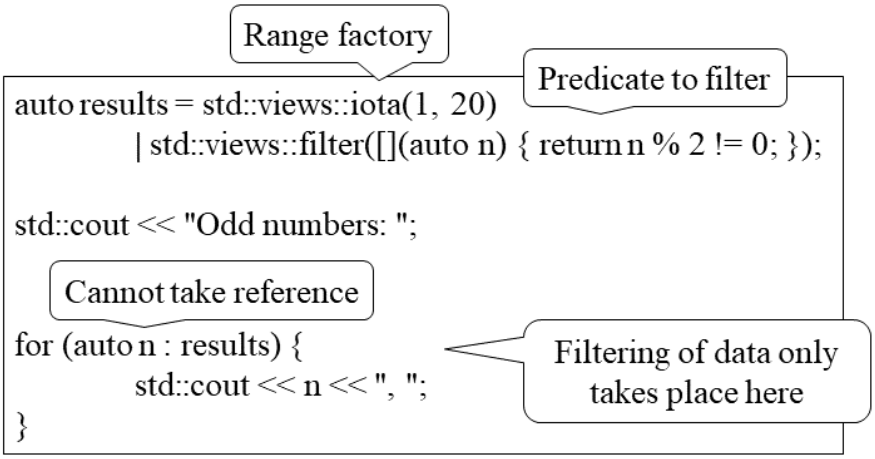
Predicate used to filter

Reference to value

Filtering of data only takes place here

Pipelining (lazy evaluation)

- Sequences of data can be processed by pipelining, using range adaptors:



Common

- One of the issues mentioned earlier was that of algorithms requiring pairs of iterators of the same type (common range)
  - This may not be the case for some range adapters:

```
std::vector<int> data{ 1,2,3,4,5 };
std::vector<int> data2;
auto take_view = data
        | std::views::take_while([](auto n) {return n < 4; })
        | std::views::common;
std::copy(take_view.begin(), take_view.end(), std::back_inserter(data2));
```

Make common_range

Range Adaptors for Elements

- Where sequences of objects are processed within a pipeline, elements can be selected:

Adaptor	Description
elements	Select specific element from tuples
keys	Select key from pair
values	Select value from pair

```
std::map<int, std::string> data{ {1, "one"}, {2,"two"}, {3, "three"} };

auto keys_results = data | std::views::keys;

std::ostream_iterator<int> osi_int(std::cout, ", ");
std::ranges::copy(keys_results, osi_int);
```

1, 2, 3,

Range Adaptor for Values

- Similarly values can be obtained from pairs:

```
std::map<int, std::string> data{ {1, "one"}, {2,"two"}, {3, "three"} };

auto values_results = data | std::views::values;

std::ostream_iterator<std::string> osi_string(std::cout, ", ");

std::ranges::copy(values_results, osi_string);
```

one, two, three,

Range Adaptor for Elements

- Sequence of tuples to be processed:

```
std::vector<std::tuple<int, std::string>> data{ {1, "one"}, {2,"two"}, {3, "three"} };

auto elements0_results = data | std::views::elements<0>;
auto elements1_results = data | std::views::elements<1>;

std::ostream_iterator<int> osi_int(std::cout, ", ");
std::ostream_iterator<std::string> osi_string(std::cout, ", ");

std::ranges::copy(elements0_results, osi_int);
std::cout << std::endl;
std::ranges::copy(elements1_results, osi_string);
std::cout << std::endl;
```

```
1, 2, 3,
one, two, three,
```


Custom Range Adaptor

- The following is a step towards creating and range adaptor to filter out even numbers:

```
template<typename T>
inline auto myodd(T&& t)
        requires std::ranges::range<T> {
    return std::views::filter(std::forward<T>(t),
        [](auto n) {return n % 2 != 0; }); }
```

Range constraint

Filter range adaptor

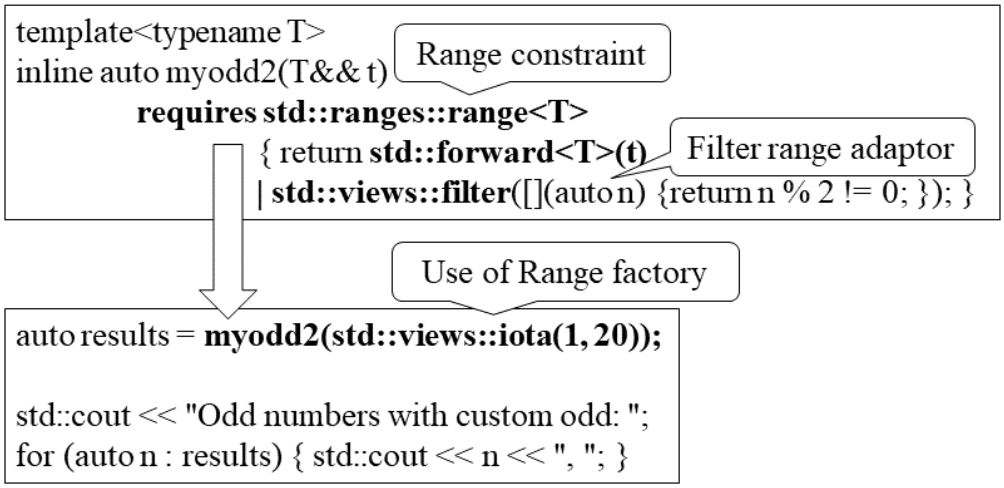
Use of Range factory

```
auto results = myodd(std::views::iota(1, 20));

std::cout << "Odd numbers with custom odd: ";
for (auto n : results) { std::cout << n << ", "; }
```

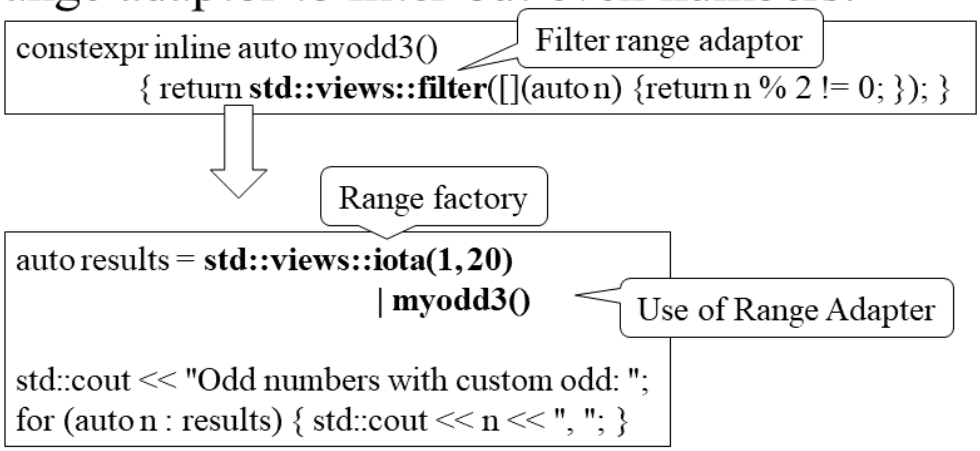
Custom Range Adaptor - continued

- The following is a step towards creating and range adaptor to filter out even numbers:



Custom Range Adaptor - continued

- The following is a step towards creating and range adaptor to filter out even numbers:



Threading

- This section considers the new thread related features:
  - jthread
  - stop_source and stop_token
  - counting_semaphore
  - latch
  - barrier

Thread class

- The ‘thread’ class provides the ability to create a single thread of execution
  - If ‘join’ is required, explicit call to the ‘join’ is required
  - All paths would require to ‘join’
    - Complicated by possible throwing of exceptions
  - The thread starts when the thread is created (subject to scheduling)

jthread

- The ‘jthread’ class is similar to thread except provides for ‘RAII’
- ‘jthread’ joins on destruction
  - No need to add call to ‘join’ on multiple paths
- ‘jthread’ also supports cancellation
  - Thread callable can take a ‘stop_token’

jthread ‘joining’

- The following use of jthread will complete successfully:

```
int main()
{
   {
      std::jthread th{ []() {
         for (size_t i = 0; i < 10; i++)
         {
            std::cout << i << std::endl;
         }
      } }; // No need to call ‘join’ member function
   }
   std::cout << "Hello World!\n";
}
```

stop_source and stop_token

- The stop_token can be used to control the termination of a thread
  - A stop_token must be associated with a stop_source
  - When creating a thread the ‘callable’ can take a stop_token
  - Request for stop can be made and checked for within thread

stop_token and stop_callback

- The stop_token allows ‘observation’:

Member function	Description
stop_requested	Returns Boolean to indicate if a stop has been requested
stop_possible	Returns Boolean to indicate if a stop is possible

- A stop ‘callback’ can be created to execute when a stop is requested:

Associate with stop_token for thread

```
std::stop_callback sc(th.get_stop_token(),
        []() { std::cout << "Callback for stop!!" << std::endl; });
```

Lambda callback, called when stop is requested

stop_source

- A stop_source has an associated ‘stop state’
  - Use to indicate if threads should stop

Member function	Description
request_stop	Request stop of thread and may change stop state (subsequent requests have no effect)
get_token	Returns a stop_token
stop_requested	Returns Boolean to indicate if a stop has been requested
stop_possible	Returns Boolean to indicate if a stop is possible

counting_semaphore

- Counting semaphore is a lightweight synchronisation object to protect resource(s)
  - Contains an internal count
  - Count can be ‘acquired’ (decremented)
  - Count can be ‘released’ (incremented)
- ‘binary_semaphore’ equivalent to counting_semaphore<1>
- Count can be configured at declaration and initialisation

counting_semaphore

- Member functions:

Member function	Description
release	Increment count, to allow another thread to acquire count
acquire	Decrement count, if available, otherwise blocks
try_acquire	Attempts to acquire and returns Boolean indicating success of failure
try_acquire_for	Attempts to acquire for a given duration
try_acquire_until	Attempts to acquire until a given time

std::latch

- The latch is a single use barrier
- The latch has a count which is decremented
  - Count initialised on creation
- Threads wait on the latch until the count is zero
  - Thus creating a form of barrier

std::latch members

- The member functions are:

Member functions	Description
count_down	Decrement the counter
try_wait	Returns Boolean indicating count is zero
wait	Wait on count to become zero
arrive_and_wait	Decrement the counter and wait on count to become zero

- Constant:

Constant	Description
max	Implementation max value for count

latch example

- The following illustrates the use of a latch:

```
using std::chrono_literals::operator"" s;
{
  std::latch l(3);
  std::jthread th1{ [&l](int n) {
    for (size_t i = 0; i < n; i++){
      std::this_thread::sleep_for(1s);
      std::cout << i << std::endl;
    }
    std::cout << "Completed..." + std::to_string( n) << std::endl;
    l.count_down();
  },20 };
  std::jthread th2{ [&l](int n) { ...}};
  l.wait();
}
```

Latch with count of three

Decrement

Total of three threads decrementing latch

Wait for count to reach zero

std::barrier

- Barrier are effectively multi use, whereas latch is single use
  - Meaning that threads can go through multiple stages
  - The threads wait for all eligible threads to reach the barrier
  - A completion function is called when threads reach the barrier

std::barrier members

- The member functions are:

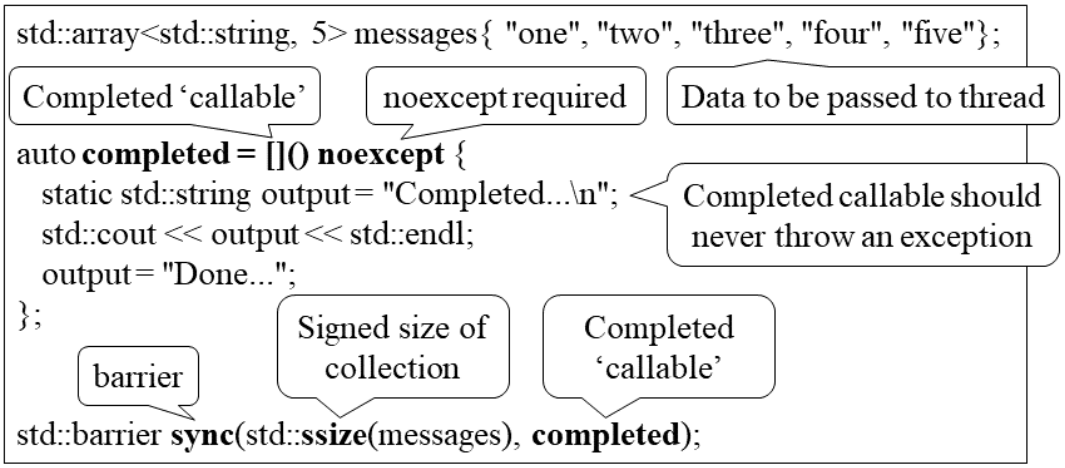
Member functions	Description
arrive	Arrival of thread ‘at’ the barrier (decrement count)
wait	Blocks at sync point for phase completion
arrive_and_wait	Arrival of thread ‘at’ the barrier (decrement count) and blocks at sync point for phase completion
arrive_and_drop	Decrement expected count and expected count

- Constant:

Constant	Description
max	Implementation max value for count

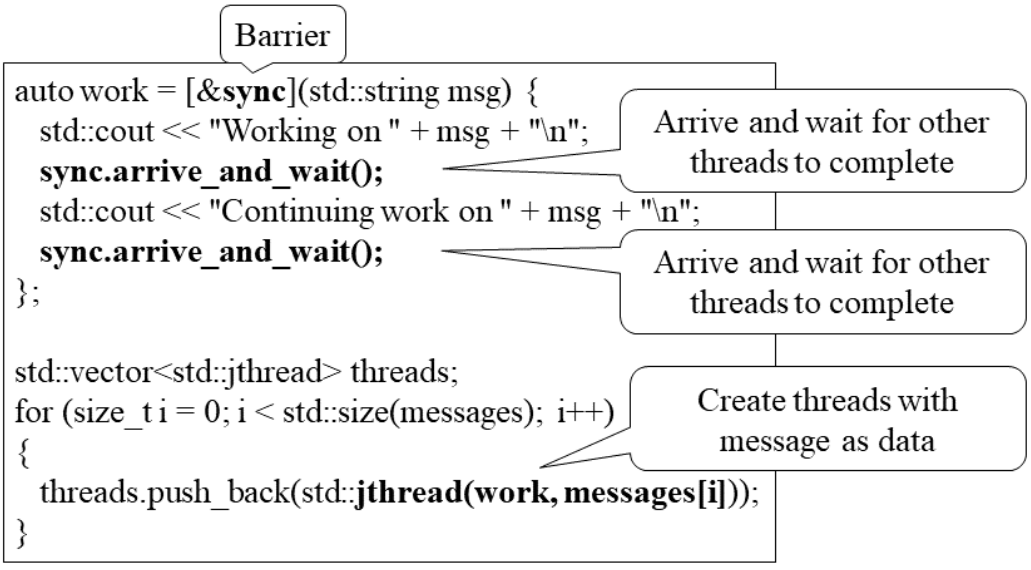
barrier example

- The following illustrates the use of a barrier:



barrier example - continued

- The following illustrates the use of a barrier:



barrier - output

- The following would be the resulting output from the barrier example:

```
Working on one
Working on two
Working on four
Working on five
Working on three
Completed...

Continuing work on three
Continuing work on four
Continuing work on five
Continuing work on two
Continuing work on one
Done...
```


Atomic

- Many new features have been added in relation to Atomic:
  - Atomic shared_ptr
  - Atomic – Wait and Notify
  - Atomic reference

Atomic shared_ptr

- The management block for std::shared_ptr was thread safe, however the use of the pointer was not thread safe
  - Previously synchronisation could be used (std::mutex) to protect the pointer
  - std::atomic_load and std::atomic_store now deprecated (for std::shared_ptr)
- Can now use std::atomic<std::shared_ptr< >> with member functions (specialised)

Atomic std::shared_ptr

- Shared pointers can now be atomic
  - Atomic shared_ptr cannot be copy or move constructable or assignable

```
{
  std::atomic<std::shared_ptr<int>> ptr{ std::make_shared<int>(7) };
  std::cout <<"Current: " << * ptr.load() << std::endl;

  std::shared_ptr<int> val{std::make_shared<int>(8)};
  ptr.store(val);

  auto result = ptr.load();

  std::cout << "Replaced: " << *result << std::endl;
}
```

Atomic Wait and Notify

- Atomic variables now support the member functions:

Member function	Description
wait	Waits on an atomic variable until notified Wait starts if the 'old' value matches the value currently stored, otherwise returns
notify_one	Notifies and ends the wait of one waiting thread
notify_all	Notifies and ends the wait of all waiting thread

Producer Thread

- The following illustrates the use of atomic wait:

```
std::atomic_int val{ 0 };
std::jthread producer{ [&val]() {
 int cnt{0};

 while (cnt < 10) {
   val.store(cnt++);
   val.notify_all();
   auto second = std::chrono::seconds{ 1 };
   std::this_thread::sleep_for(second);
 }}
};
```

```
std::jthread consumer{ [&val]() {
 int cnt{0};
 while (true) {
   val.wait(cnt);
   cnt=val.load();
   std::cout << cnt << std::endl;
 }}
};
```

Atomic Reference

- The following illustrates the use of atomic wait:

```
int val{ 0 };
std::atomic_ref ref{ val };
std::jthread producer{ [&ref]() {
  int cnt{0};

  while (cnt < 10) {
    ref = cnt++;
    ref.notify_all();
    auto second = std::chrono::seconds{ 1 };
    std::this_thread::sleep_for(second);
  }
} };
```

```
std::jthread consumer{ [&ref]() {
  int cnt{0};
  while (true) {
    ref.wait(cnt);
    cnt = ref;
    std::cout << cnt << std::endl;
  }
} };
```


General C++ 20 Features

- C++ 20 has introduced many new features:
  - Using enum
  - Lambda Capture of ‘this’
  - constexpr and consteval
  - Constinit
  - Designated Initialiser
  - const virtual functions
  - Conditional explicit
  - [[likely]] and [[unlikely]] attributes

Using enum

- It is now possible to introduce an enum into scope with a using statement:

```
enum class Number { One = 1, Two = 2, Three = 3 };

struct SomeNumbers
{
  using enum Number;
};
int main()
{
  SomeNumbers sn;

  std::cout << (int)sn.One << std::endl;
  std::cout << (int)SomeNumbers::Two << std::endl;
}
```

Number introduced into structure

Access enum value in a number of ways

Using enum - continued

- The using for enum is useful within switch statements:

```
enum class Number { One = 1, Two = 2, Three = 3 };
int main(){
  Number num{ Number::Two };
  switch (num){
  case Number::One:
     std::cout << "One" << std::endl;
  break;
  case Number::Two:
    std::cout << "Two" << std::endl;
  break;
  …
}
```

```
switch (num){
  using enum Number;
  case One:
    std::cout << "One" << std::endl;
  break;
  case Two:
    std::cout << "Two" << std::endl;
  break;
  …
```

Lambda Capture of ‘this’

- Whilst this change is in one sense minor, it could potentially be a breaking change:
  - Prior to C++ 20 the use of ‘[=]’ could capture everything from the scope, including ‘this’ for member functions
    - Thus it is proposed to deprecate the use of [=] for the capture of ‘this’
  - In C++ 20 to produce the same effect would require the use of ‘[=,this]’

constexpr and consteval

- constexpr was introduced with C++11 to allow evaluation at compile time of expressions involving function calls
  - C++11 placed a number of restrictions on the definition of the function
  - These restrictions were relaxed in later standards to allow use of assignment and loops
  - constexpr functions may also be used at runtime where an argument is a variable

Return Type

- The return type and parameters to constexpr functions have the named requirement LiteralType:
- Requirements include being one of:
  - void
  - Scalar or reference type
  - Array of literal type
  - Class (with various requirements)!

constexpr and consteval continued…

- constexpr was technically required on both the function and the variable being initialised
  - Although some compilers were a little relaxed
- constexpr is used widely within the standard library
  - Many algorithms are now constexpr
- consteval is similar to constexpr but with the restriction that a consteval function can only be evaluated at compile time (immediate function)
- An ‘immediate function’ is one which is evaluated at compile time to return a constant

constexpr in C++ 20

- The use of constexpr has now been expanded further
  - virtual function can now be constexpr
  - Whilst constexpr functions require to be self contained, they can now use:
    - Dynamic memory allocation
      - Using only global ‘new’ and ‘delete’
    - try/catch blocks (although be noexcept)
    - ‘dynamic_cast’ and ‘typeid’

constexpr virtual functions

- Clearly there will be limitations on how this can be applied!

```
struct A {
  constexpr virtual int value() const { return 7; }
};
struct B : A {
  constexpr int value() const override { return 9; }
};
constexpr int add(const A& x, const A& y) {
  return x.value() + y.value();
}
```

```
int main()
{
  constexpr auto result = add(B{}, A{});
  ...
}
```

std::string and std::vector

- The changes in support for constexpr have now made it possible to use both std::string and std::vector in constexpr functions
  - Now possible to use within constexpr functions
  - The following can now be treated as constexpr:

```
constexpr int sum_vec(const std::vector<int>& data)
{
  int result{};
  for (const auto& n : data) { result += n; }
  return result;
}
```

```
constexpr auto result = sum_vec(std::vector{1, 2, 3, 4});
```


constexpr Algorithms

- Throughout the C++20 standard library constexpr is now widely used
- Many Algorithm are now constexpr

```
constexpr int sum(const std::vector<int>& data){
    return std::reduce(data.cbegin(), data.cend(), 0,
              [](auto a, auto b){return a+b;});
}
int main()
{
  constexpr int result = sum(std::vector{3,4,5,6});
  ...
}
```

constinit

- The variable has ‘static’ initialisation
- Whilst appearing similar constinit is used for initialising ‘static’ and thread local values at compile time
- The keywords constexpr, consteval and constinit cannot be used in combination

constinit

- Initialising a static variable:

```
constexpr int factorial(int n)
              { return n==0? 1 : n*factorial(n-1);}

int main()
{
  int n = 2;

  static constinit int fact7 = factorial(7);

  // static constinit int factn = factorial(n);

  return 0;
}
```

Variable parameter mean is cannot be evaluate at compile time

Designated Initialiser

- For simple aggregate types (struct) designated initialisers allow initialisation of members by member name:

```
struct some_data {
   int a{1}, b{2};
};
int main()
{
          some_data sd{ .b = 42 };
          ...
}
```

Named member to be given value. Must be named in declared order

Value of 'a' is default

explicit

- Constructors and type conversion operators can be used for implicit type promotion or type conversion
- The compiler is allowed to use one level of these in statements and expressions
  - This can be useful to avoid cluttering code with constructor calls and casts
- However, especially where operators are defined, ambiguities can arise

explicit

- The explicit keyword has been available for use on one parameter constructors for a long time
  - Which forces an explicit call to constructor
- The explicit keyword was made available for type conversion operators from C++ 11
  - Which forces an explicit cast for type conversion
- The application of the explicit keyword can be used to resolve ambiguities mentioned

Conditional explicit

- Explicit can now be conditional, which means that within template classes, one parameter constructors and type conversion operators can have finer grain control:

```
template<typename T>
class Num {
      ...
public:
   ...
   explicit( std::is_integral_v<T> || std::is_floating_point_v<T>)
         operator T() const {
      return … }
};
```

Use appropriate type traits

[[likely]] and [[unlikely]] attributes

- The [[likely]] and [[unlikely]] attributes are to be used to give a hint to the compiler for optimisation purposes
  - Can be applied within conditional statements to indicate likely and unlikely 'branches'

```
int num{};
std::cin >> num;
if(num > 0)[[unlikely]]{
  std::cout << "Greater than zero!" << std::endl;}
else{
  std::cout << "Less that or equal zero!" << std::endl;
}
```

Hint that branch is unlikely!

Switch with Attributes

- [[likely]] and [[unlikely]] can be used within a switch statement:

```
int num{};
std::cin >> num;
switch(num){
  [[unlikely]]case 1:std::cout << "One" << std::endl;
    break;
  [[likely]]case 2:std::cout << "Two" << std::endl;
    break;
  case 3:std::cout << "Three" << std::endl;
    break;
}
```

- Cannot use [[likely]] and [[unlikely]] on same statement

General C++ 20 Features - continued

- C++ 20 has introduced many new features:
  - Spaceship Operator
  - bind_front function
  - Source Location
  - Initialisation in Range based For
  - Shared pointer for arrays
  - Removing Items
  - Format Library

Spaceship Operator

- This section is to introduction to the new spaceship operator:
  - Spaceship Operator <=>
  - Relational Operators
  - Expression Rewriting
  - Simple Class with Spaceship
  - Operator Symmetry
  - Custom Spaceship Operator
  - Spaceship Operator Functions

Spaceship Operator <=>

- The spaceship operator (or three way comparison) is new to C++20 and reduces the need to explicitly provide relational operators
  - Traditionally many operators could be provided:

Operator	Description
operator<	Greater than
operator<=	Greater than or equal
operator>	Less than
operator>=	Less than or equal
operator==	Equals
operator!=	Not equals

Relational Operators

- The traditional way relational operators are defined is as non member functions (in order to be symmetric)
  - Thus defined as friend operator function or global function outside of the class
  - Define one or two and define others in terms of those defined (in order to avoid mistakes)!
  - operator< should satisfy ‘strict weak ordering’ in order to be used with containers and algorithms

Expression Rewriting

- Defining the Spaceship operator now uses another new feature of C++20
- If the expression ‘a < b’ is written for variables of type with the spaceship operator, this expression is rewritten as ‘ (a <=> b) < 0’
- The equality and inequality are defined separately but nevertheless generated when the spaceship operator is provided

Simple class with Spaceship

- Defined class:

```
class ValueClass                          #include <compare>
{
  int _val{};
public:
  ValueClass(int val):_val{val}{}
  auto operator<=>( const ValueClass& rhs)const = default;
};
```

- Generated:

```
class ValueClass
{
  int _val;
public:
  inline ValueClass(int val) : _val{val} { }
  inline constexpr std::strong_ordering operator<=>(const ValueClass & rhs)
                                                    const noexcept = default;
  inline constexpr bool operator==(const ValueClass & rhs) const = default;
};
```

Strong ordering as comparing integral types

<=> Generated Code

- In the previous example the return type of the operator is std::strong_ordering
  - This was due to the one data member being int
- If the data members include a floating point number the return type will be std::partial_ordering as some values are not comparable
- Where a class has multiple data members the operator compares the data members in order

Class Containing Double

- Use the default implementation of <=>:

```
struct TheDouble {
  double d;
  auto operator<=>(const TheDouble& f)const = default;
};
```

```
int main() {
{
  TheDouble td1{ 34.2 }, td2{ NAN };

  auto result = td1 <=> td2;
}
```

NAN or INFINITE could be tried

'unordered' for NAN and 'greater' for INFINITE

Operator Symmetry

- When defining operators manually symmetry had to be considered
- Expression rewriting takes this into account, e.g.

```
ValueClass vc1{1},vc2{2};

if(1 < vc2){  std::cout << "1 less than 2!\n" << std::endl; }
```

- Generated code:

```
ValueClass vc1 = ValueClass{1};
ValueClass vc2 = ValueClass{2};

if(operator<(__cmp_cat::__unspec(0),vc2.operator<=>(ValueClass(1)))) {
    std::operator<<(std::cout, "1 less than 2!\n").operator<<(std::endl);
}
```

Right hand operand on left

Type promotion used

std::strong_ordering

- The previous generated code return type for the spaceship operator is not Boolean!
  - Return type was std::strong_ordering
- std::string_ordering has four static constants

Constants	Description
less	Represents a less-than relationship (ordered before)
greater	Represents a greater-than relationship (ordered after)
equal	Representing equivalence (not ordered before or ordered after)
equivalent	Representing equivalence – same as equal

Custom Spaceship Operator

- It is possible to define a custom operator <=>:

```
class ValueClass
{
 int _val{};
 public:
 ValueClass(int val):_val{val}{}
 constexpr std::strong_ordering operator<=>(const ValueClass& rhs) const{
  if( _val < rhs._val) return std::strong_ordering::less;
  else if(_val > rhs._val) return std::strong_ordering::greater;
  else return std::strong_ordering::equal;
 }
 constexpr bool operator==(const ValueClass& rhs) const{
  return _val == rhs._val;
 }
};
```

operator!= will be generated

Orderings

- All orderings should be:

	Description
irreflexive	a < a is false
asymmetric	a < b true, implies b < a false
transitive	a < b true and b < c true, implies a < c true

- A number of orderings are defined:

Ordering	**Constants used**
Strong Ordering	greater/equal/equivalent/less (substitutable if equal)
Weak Ordering	greater/equivalent/less (not substitutable)
Partial Ordering	greater/equivalent/less/unordered

Equivalent means neither ordered before or after!

Incomparable value!

Spaceship Operator Functions

- New functions have been added which take the result from a spaceship operator

Function	Description
is_gt	Greater than
is_gteq	Greater than or equal
is_lt	Less than
is_lteq	Less than or equal
is_eq	Equals
is_neq	Not equals

Spaceship operator Example

- Alternative ways spaceship operator could be used explicitly:

```
struct some_data {
    int a{1};
    int b{2};
    auto operator<=>(const some_data&)const = default;
};
int main(){
   some_data sd;
   std::strong_ordering result = some_data{2, 3 } <=> sd;
   if (result == std::strong_ordering::greater)
                        std::cout << "greater than..." << std::endl;
   if (std::is_gt(result)) std::cout << "greater than..." << std::endl;
}
```

std::bind_front

- The std::bind function provides a very flexible means of creating a new ‘callable’ from an existing ‘callable’
- std::bind_front provides a more limited, but useful simple means of creating a new ‘callable’, by binding just the first arguments
  - The remaining arguments can then be passed to the returned ‘callable’

bind_front Example

- Using a simple function:

```
int calc(int a, int b, std::string c, std::string d) {

  return a + b - std::stoi(c) * std::stoi(d);
}
int main(){
                                  Bind first two arguments
  auto part = std::bind_front(calc, 3, 7);
                                  Pass remaining arguments
  std::cout << "Result: " << part("2", "3") << std::endl;;
}
```

bind_front Member Function Example

- Using a simple function:

```
class DataClass {
  int _a;
public:
  DataClass(int a) :_a{ a } {}
  int calc(int b, int c) const { return _a - b + c; }
};
int main(){
  DataClass dc{ 10 };
  auto member = std::bind_front(&DataClass::calc, dc);

  std::cout << "Result: " << member(1, 2) << std::endl;;
}
```

Member function

Bind object

Pass remaining arguments

std::source_location

- std::source_location allows obtaining location within code:

Member function	Description
current	Returns source_location corresponding to where the call to 'current' takes place
line	Returns line number where source location obtained
column	Returns column number where source location obtained
file_name	Returns file name where source location obtained
function_name	Returns function name where source location obtained

Source Location - Example

- Function to obtain location:

```
void logit(const std::string& msg,
           std::source_location sl = std::source_location::current()){

  std::cout << msg << std::endl;
  std::cout << "Function: " << sl.function_name()
            << ", Line number: " << sl.line() << std::endl;
}
```

Called when function is called

- Using the default parameter allows obtaining position for 'logit' function call

Initialisation in Range based For

- C++ 17 introduced initialisation within both switch and while
  - Now also available within range based for:

```
auto get_data() {
   return std::vector<int>{7, 6, 5, 4, 3, 2, 1};
}
int main()
{
  for (auto data = get_data(); const auto & n: data) {
     std::cout << n << std::endl;
  }
}
```

std::shared_ptr for Arrays

- Shared pointers to arrays can now be created using make_shared, e.g.:

```
class A {
   public:
      ~A(){ std::cout << "Destroying A..." << std::endl; }
};
```

```
int main()
{
      auto ptr = std::make_shared<A[]>(5);


}
```

Output:+

```
Destroying A...
Destroying A...
Destroying A...
Destroying A...
Destroying A...
```


Removing Items (pre C++ 20)

- Removing items from a container creates an interesting challenge, as it is important not to invalidate iterators
- There are a number of options for removing items:
  - Removing by moving values up, this does not change the size of the container
  - Remove copy, to populates a new container with only wanted items

std::remove and std::remove_if

- std::remove and std::remove_if simply moves values up in the container
  - The container size remains the same
  - Returns new 'end' iterator
    - Iterator to value after last wanted item.

```
void remove_value(std::vector<int>& data, int value)
{
  auto end = std::remove(std::begin(data), std::end(data), value);

  auto newEnd = std::remove_if(std::begin(data), std::end(data),
                    std::bind(std::less<int>(),std::placeholders::_1, 3));
}
```

Compared using == operator

C++ 20 erase and erase_if

- For std::vector the erase member function could then be used
- With C++ 20 there is std::erase and std::erase_if algorithms for vector and string

```
std::cout << "The C++ 20 way:" << std::endl;
std::vector data{ 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8 };

std::erase(data, 5);

std::ostream_iterator<int> osi(std::cout, ", ");
std::ranges::copy(data, osi);
```

```
The C++ 20 way:
1, 2, 3, 4, 6, 7, 8, 1, 2, 3, 4, 6, 7, 8,
```

Format Library

- This section introduces the format library:
  - Format Library Introduction
  - std::format
  - std::vformat
  - Formatting String
  - Numbered Placeholders
  - Format String Specification
  - Format Integral Types
  - Format Floating Point Types

Formatting Library Introduction

- C++ already has the stream library and the legacy printf function, however the new formatting library provides an extensible alternative
  - Complements stream library and reuses some features
  - std::format allows formatting of a std::string using a format string and arguments

Formatting Library continued

- The library contains many functions and overloads for formatting strings
  - Overloads for wide characters and locale
- String can be created in a number of ways:

Function	Description
format	Returns a string using format and arguments
format_to	Outputs a string using an iterator
vformat	Returns a string using format (view) and arguments (using make)!
vformat_to	Outputs a string using an iterator, from a format (view) and arguments (using make)!

std::format

- std::format can be used in a number different ways (include <format>):

```
auto n = 2;
auto d = 3.2;
auto f = 65.3f;

auto message = std::format( "int: {}, double: {}, float: {}", n, d, f);

std::cout << message << std::endl;
```

string

Format string

Arguments to populate {}

{} placeholders filled in order

std::vformat

- std::vformat can be used in a number different ways (include <format>):

```
auto n = 2;
auto d = 3.2;
auto f = 65.3f;
   string            Format string
std::string fmt = "int: {}, double: {}, float: {}";
                  string_view passed
auto message = std::vformat( fmt, std::make_format_args(n, d, f));

std::cout << message << std::endl;        Make format args store
```


Formatting String

- Formating strings is straightforward:

```
using std::string_literals::operator "" s;
auto b{ true };
auto ch {'g'};
auto str{ "Greetings"s };
try {                          Output Boolean as 'string'    Output Character
   auto message = std::format( "s:{0:s}, s:{1:c}, s:{2:s}", b, ch, str);
                                          s:true, s:g, s:Greetings
   std::cout << message << std::endl;
}
catch (const std::format_error& fe) {
std::cout << fe.what() << std::endl;
}
```

Numbered Placeholders

- Placeholders can be numbered
- The format function can throw an exception

```
auto n = 2;
auto d = 3.2;
auto f = 65.3f;
try {
  auto message = std::format( "float: {1}, int: {0}, double: {2}", n, d, f);

  std::cout << message << std::endl;
}
catch (const std::format_error & fe) {
  std::cout << fe.what() << std::endl;
}
```

Format string

Arguments to populate {}, additional arguments will be ignored

Number to identify argument

Format String Specification

- The specification has many options ([] indicates optional):

[fill and align][sign][#][0][width][precision][L][type]

– Fill and align

Symbol	Meaning
<	Left justified (default none numeric)
>	Right justified (default numeric)
^	Centered

– Sign

Symbol	Meaning
+	Both negative and positive sign shown
-	Only negative sign shown (default)
space	Space for positive sign

Format String Specification continued

- The specification has many options ([] indicates optional):

[fill and align][sign][#][0][width][precision][L][type]

  - [#] indicates prefix should be shown for Boolean, Octal and Hexadecimal
  - [0] indicates pad with zeros
  - [width] is the minimum field width
  - [precision] is a '.' and the number of digits
  - [L] formatting specific to locale

Format Specifiers

- Following are some of the format specifiers:

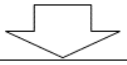
Format specifier	**Type**	**Description (# to show prefix)**
none, s	string	String copied
b	Integral (not chars)	Binary format
B	"	Binary format, but prefix 0B
c	"	Character
d	"	Decimal format
o	"	Octal format
x	"	Hexadecimal format
X	"	Hexadecimal format, but prefix 0X
none		Defaults to 'd'

Format Integral Type

- Use of format characters:

```
auto n = 287;

try {
auto message = std::format(
        "b:{0:#b}, B:{0:#B}, d:{0:d}, o:{0:#o}, x:{0:#x}, X:{0:#X}", n);
```



```
    b:0b100011111, B:0B100011111, d:287, o:0437, x:0x11f, X:0X11F
std::cout << message << std::endl;
}
  catch (const std::format_error& fe) {
  std::cout << fe.what() << std::endl;
}
```


Format Specifiers continued

- Following are some of the format specifiers:

Format specifier	Type	Description
a	Floating point	Floating point number
A	“	Floating point number (capital letters)
e	“	Scientific notation
E	“	Scientific notation, Uses E
f	“	Floating point number
F	“	Floating point number (capital letters)
g	“	General number format
G	“	General number format (capital letters)
none	“	Similar to g

Format Floating Point Type

- Use of format characters:

```
auto d = 2.2;

try {
  auto message = std::format(
          "a:{0:a}, A:{0:A}, e:{0:e}, E:{0:E}, f:{0:f}, F:{0:F}", d);
```

```
  std::cout << message << std::endl;
}
catch (const std::format_error& fe) {
  std::cout << fe.what() << std::endl;
}
```

a:1.199999999999ap+1, A:1.199999999999AP+1,
e:2.200000e+00, E:2.200000E+00, f:2.200000, F:2.200000

Specific Floating Point

- Use of options:

```
auto d{ 234.235 };

try {
  auto message = std::format("double precision number: '{0:^+12.4f}'", d);
```

double precision number: **' +234.2350 '**

```
  std::cout << message << std::endl;
}
catch (const std::format_error& fe) {
  std::cout << fe.what() << std::endl;
}
```

Print Library (C++23)

- This section gives an overview of the new C++23 print library:
  - Introduction to ‘print’
  - Printing with <iostream>
  - Printing with <print>

Introduction to ‘print'

- The format library provides a means of creating a string from a number of arguments and a format string
- This formatted string could then be output using streams
- The new ‘print’ capabilities provide functions which can be call to output arguments to an output stream in a formatted way

Printing with <iostream>

- The header file <iostream> now provides a number of new functions:

Function	Description
print	Outputs to an ostream a number of formatted arguments
println	Like print, but appends a new line character

```
{
 int n{5}, square{n * n};

 std::println(std::cout, "The square of {0} is {1}.", n, square);
}
```

The square of 5 is 25.

Printing with <print>

- The new header file <print> provides:

Function	Description
print	Outputs to stdout a number of formatted arguments
println	Like print, but appends a new line character

```
{
 int n{5}, square{n * n};

 std::println(std::cout, "The square of {0} is {1}.", n, square);
}
```

The square of 5 is 25.

Printing with <print>

- Print and println are overloaded to take file:

```
{
 FILE* aFile = new FILE();

 fopen_s(&aFile, "C:/Temp/Info.txt", "w");
 if (aFile) {
  int n{ 5 }, square{ n * n };

  std::println(aFile, "The square of {0} is {1}.", n, square);
  std::fclose(aFile);
 }
}
```


C++ 23 Operators

- This section gives an introduction to some of the new features in C++ 23:
  - Static Operators
  - Usage of static operator()
  - Static Lambdas
  - operator[]
  - operator[,]

Static Operators

- C++ 23 now allows the operator() and operator[] (for consistency) to be declared static
  - The use of operator() to allow definition of function object has been available for a long time
  - However, where the type contains no members there seems little necessity for an instance function
  - Thus, it is now possible to define a the operator() as static
  - A compiler can then optimise away the pointer to 'this'.

```
struct Square{
  static int operator()(int n){ return n*n;}
}
```


Usage of static operator()

- Whilst the operator() may be static its usage is the same as for the instance operator:

```
{
  std::vector<int> data(10);
  std::ranges::iota(data, 1);

  std::ranges::transform(data, data.begin(), Square{});

  std::ranges::copy(data, std::ostream_iterator<int>{std::cout, ", "});
}
```

```
1, 4, 9 16, 24, 36, 49, 64, 81, 100,
```

Static Lambdas

- A consequence of the static operator() is that the generated operator() for a lambda can now be static:
  - This cannot be done implicitly as this may break some existing code
  - Possible to opt-in to static operator by defining static lambda for capture-less lambdas

```
{
 auto fun = [](int n) static { return n*n;}
 std::cout << fun(12) << std::endl;
}
```

Empty capture

operator[]

- For consistency it is also possible to define the operator[] as static for types with no data members:

```
struct Echo{
   static int operator[](int n){ return  n;}
};
```

```
{
 Echo echo;

 std::cout << echo[7] << std::endl;
}
```

operator[,]

- The operator[] can now support multiple comma separated ‘indices’:

```
template<unsigned N>
struct MyData{
   int *_data;
public:
   MyData(int *data):_data{data}{}
   ~MyData(){ delete [] _data;}
   int operator[](int a, int b){ return _data[a*N+b];}
};
```

```
{
 int *data = new int[]{1,2,3,4,5,6};
 MyData<3> md{data};

 for (size_t i = 0; i < 2; i++)
 {
  for (size_t j = 0; j < 3; j++)
  {
   std::cout << md[i,j] << ", ";
  }
  std::cout << std::endl;
 }
}
```


C++ 23 Fold Algorithms

- This section gives an introduction to Fold Algorithms:
  - Fold Algorithms Introduction
  - Fold Expression (C++ 17)
  - Fold Algorithms
  - Fold Algorithms Example
  - Left Fold

Fold Algorithms Introduction

- C++ 23 now provides algorithms performing fold operations
  - Fold expressions have previously been available by the use of variadic templates
  - Algorithms correspond to these fold expressions
  - Whereas the use of variadic templates require parameters to be passed, the algorithms allow the use of containers

Fold Expressions (C++ 17)

- Fold expressions are used in conjunction with variadic templates
  - Makes implementing functions simpler
  - Does not require recursive implementation
- Four options:

Fold Expression	**Description**
(pack op …)	Right unary expression
(… op pack)	Left unary expression
(pack op … op initexpr)	Right binary expression (operators must be the same)
(initexpr op … op pack)	Left binary expression(operators must be the same)

Fold Algorithms

- Available algorithms are constrained algorithms and either take a range or iterator-sentinel pair:

Algorithm	Description	Returns
fold_right	Takes initial value	Result of fold
fold_right_last	Last element as initial value	Result of fold
fold_left	Take initial value	Result of fold
fold_left_first	First element as initial value	Result of fold
fold_left_with_iter	Take initial value	Result and end iterator
fold_left_first_with_iter	First element as initial value	Result and end iterator

Fold Algorithm Example

- The following illustrates the use of the left_fold:

```
{
 std::vector<int> data(7);
 std::ranges::iota(data, 1);

 auto result = std::ranges::fold_left(data, 0,
                                      [](int a, int b) { return a+b;});

 std::cout << result << std::endl;
}
```

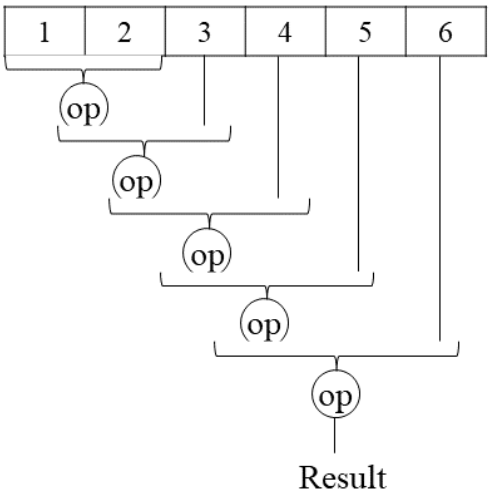
Range

Initial value

Binary Operation

Left Fold

- The left fold:
  - fold_left:



C++ Monad

- This section gives an introduction to implementing Monads in C++:
  - Monad Pattern
  - Implementing the Monad
  - Monad Usage
  - C++ 23 std::optional monadic functions
  - Monad in C++ 23
  - C++ 23 std::expected
  - Monad in C++ 23 (std::expected)

Monad Pattern

- The monad pattern allows involves a ‘value’ and associated ‘computation’
- The use of monads allows data to be handled in a structured way
  - Monads can be used in a sequence in order to perform multiple computations
- Monads are typically use in functional programming and are particularly common in languages like Haskell

Implementing the Monad

- Prior to C++ 23 monads could be implement by wrapping the std::optional type:

```
template<typename T>
class Maybe {
private:
  std::optional<T> value;
public:
  Maybe() : value(std::nullopt) {}
  Maybe(T val) : value(val) {}
  bool has_value() const {
    return value.has_value();
  }
  T unwrap() const { return *value; }
  ...
};
```

Check if value held

Return the value

Implementing the Monad - continued

- An important member of the type is ‘bind’, which allow the computation if value exists:

```
template<typename T> class Maybe {
public:
  ...
  template <typename U>
  auto bind(std::function<Maybe<U>(T)> f) {
    if (value) { return f(*value); }
    else { return Maybe<T>(); }
  }

  static Maybe<T> return_value(T val) {
    return Maybe<T>(val);
  }
};
```

Execute the function

Return nullopt

Monad Usage - computations

- Given the possible computations defined below, these can be performed in sequence:

```
Maybe<int> add_five(int x) {
  return Maybe<int>::return_value(x + 5);
}

Maybe<int> divide_by_two(int x) {
  if (x % 2 == 0) {
    return Maybe<int>::return_value(x / 2);
  }
  else {
    return Maybe<int>();
  }
}
```

Monad Usage

- Maybe can be used as:

```
int main() {
  Maybe<int> value = Maybe<int>::return_value(10);

  Maybe<int> result = value
    .bind<int>(add_five)
    .bind<int>(divide_by_two);

  if (result.has_value()) {
    std::cout << "Result: " << result.unwrap() << std::endl;
  }
  else {
    std::cout << "Failed." << std::endl;
  }
}
```

Perform multiple computations

C++ 23 std::optional monadic functions

- As of C++ 23 it is now possible to perform similar functionality using the new optional member functions:

Member function	Description
and_then	Compute function if value exists and return optional otherwise return empty optional
transform	Return transformed value if it exists and return optional otherwise return empty optional
or_else	Return optional if value exists otherwise result of give function

Monad is C++ 23

- The follow could now be used instead of the Maybe type
- First of all computations:

```
std::optional<int> add_five(int x) {
  return x + 5;
}
std::optional<int> divide_by_two(int x) {
  if (x % 2 == 0) {
    return x / 2;
  }
  else {
    return std::nullopt;
  }
}
```


Monad in C++ 23 - continued

- Usage of the ‘monad’, implemented using optional:

```
int main() {
  std::optional<int> value = 10;
  auto result = value
    .and_then(add_five)
    .and_then(divide_by_two);

  if (result.has_value()) {
    std::cout << "Result: " << *result << std::endl;
  }
  else {
    std::cout << "Computation failed." << std::endl;
  }
}
```

Perform multiple computations

C++ 23 std::expected

- A variation on the monad implementation can be achieved using std::expected
- std::expected either holds a value or an error
  - Similar ‘monadic’ members are available to those on std::optional
  - On ‘failure’ the std::expended holds an error and not ‘empty’

Monad is C++ 23 (std::expected)

- The follow could now be used instead of the Maybe type
- First of all computations:

```
std::expected<int, std::string> add_five(int x) {
  return x + 5;
}
std::expected<int, std::string> divide_by_two(int x) {
  if (x % 2 == 0) {
    return x / 2;
  }
  else {
    return std::unexpected(
       "Division by an odd number is not allowed");
  }
}
```

Monad in C++ 23 - continued

- Usage of the ‘monad’, implemented using optional:

```
int main() {
  std::expected<int, std::string> initial_value = 10;

  auto result = initial_value
    .and_then(add_five)
    .and_then(divide_by_two);
  if (result) {
    std::cout << "Result: " << *result << std::endl;
  }
  else {
    std::cout << "Computation failed: "
              << result.error() << std::endl;
  }
}
```

Perform multiple computations

The End

Congratulations on completing this C++ course

Appendix 1

Modules

Modules

- This section introduces the Modules feature:
  - Modules Introduction
  - Module Interface
  - Visual Studio Support
  - Module Interface Unit
  - Module Implementation Unit
  - Module Implementation Unit and Private
  - Module Partitions

Modules Introduction

- Modules essentially replace header files
- Header files suffered from many problems
  - No encapsulation
  - Leakage of defines/macros
  - Inconsistency
  - Multiple builds – may result in high build times
  - Order significant
- Modules resolve all of these problems
  - Module interface - specific ‘export’
  - Implementation can be hidden
  - Order not significant

Modules Support

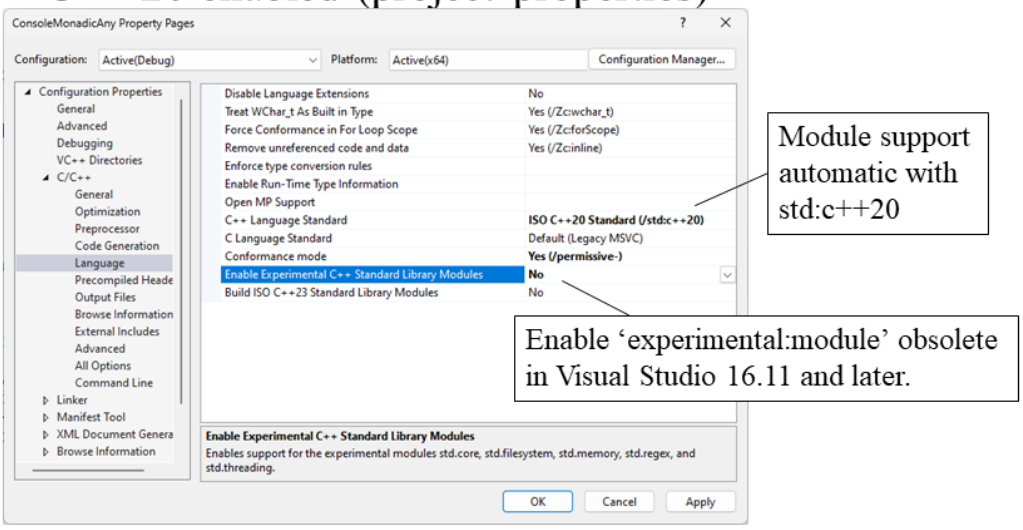
- Typically post standardisation it takes time for compiler vendors to implement all of the language features
- Modules is a feature taking time!
- Microsoft one of the first to implement Modules feature
- Also module support implemented differently by different compiler vendors
  - Different compiler options!

Module Interface

- Define a module interface and only export what is expected to be used externally
  - 'module' and 'import' are identifiers
- Import appropriate 'exported' modules
  - Only processed once
  - Improved build process/reduced build time
- Import standard library header files
  - The settings required for this are compiler specific

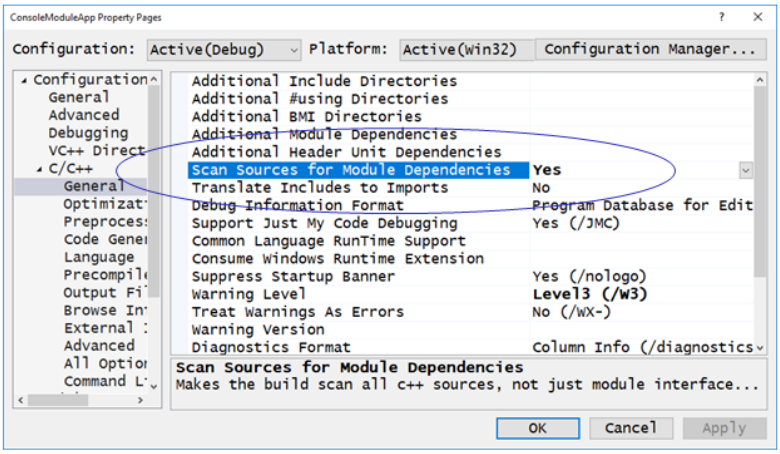
Visual Studio Support

- Settings for modules support:
  - C++ 20 enabled (project properties)



Visual Studio Support

- Settings required for supporting modules:
  - Scan Source for Module (project properties)



Module Interface Unit

- A common file extension for the Module Interface Unit is '.ixx'
  - Single Module Interface Unit per module
  - Can have multiple Module Implementation Units
- Visual Studio projects simply add a Module
  - Provided settings correct, project should build!
- For clang compiler
  - build the '.ixx' file in addition to the '.cpp' file
    - Creates a '.ifc' file in addition to '.obj' file

Export module

- Modules can be defined as '.ixx' files:

```
module;
export module flighthandling.passengers;
import <string>;
export namespace flighthandling {
  class PassengerDetails {
    std::string _name;
    int _weight;
  public:
    PassengerDetails(const std::string& name, const int& weight) :
            _name(name), _weight(weight) {}
    std::string get_name()const { return _name; }
    int get_weight()const { return _weight; }
    …
}
```

Global module fragment after 'module'

Export 'module'. Module name

Export namespace

Types and functions within exported namespace will be visible

Previous Module Explained

- The use of 'module' initially defines a 'global module fragment'
  - Allows including header files for use within the module
- The export of the module name defines how this module may be imported
  - Import other modules
  - Export namespaces/classes/functions
    - But not with static linkage

Modules continued…

- Many modules will need to include header files, but as mention, if supported these 'header' files may be 'imported' instead
- Many modules will need to import other modules, however these modules may also be exported, e.g.:

```
export import flighthandling.flights;
```


Module Implementation Unit

- All of the implementation could be within the Module Interface Unit, however Implementation Units can be added
  - Simply files with the extension '.cpp'
  - Use 'module' keyword with module name to be implemented, but no exports
  - Implement function signatures declared within Module Interface Unit

Implementation Unit Example

- Spliting implementation into separate file:

Simple.ixx

```
export module Simple;

export inline int factorial(int a);
```

```
import Simple;
int main()
{
  std::cout << factorial(5) << std::endl;
}
```

SimpleImpl.cpp

```
module Simple;

int factorial(int a){
   return a == 0 ? 1 : a * factorial(a - 1);
}
```


Implementation Units and Private

- One of the advantages of creating implementation files is that a change to this code would not require recompiling the Interface Unit
- When using just single file Module Interface Units a similar effect can be achieved by creating a private module fragment:

```
module : private;
```

Private Module Fragment

- A Single Module Interface Unit with Private Module Fragment:

```
export module Simple;

export int cube(int a);

module :private;

int cube(int a) { return a * a * a; }
```

```
import Simple;
int main()
{
  std::cout << cube(5) << std::endl;
}
```


Modules Partitions

- Modules may have partitions
  - Single partition within a ‘module unit’

```
module;
#include <iostream>
export module Utilities:A;
export void do_workA() { std::cout << "From A..." << std::endl; }
```

Export Module Partition ‘A’

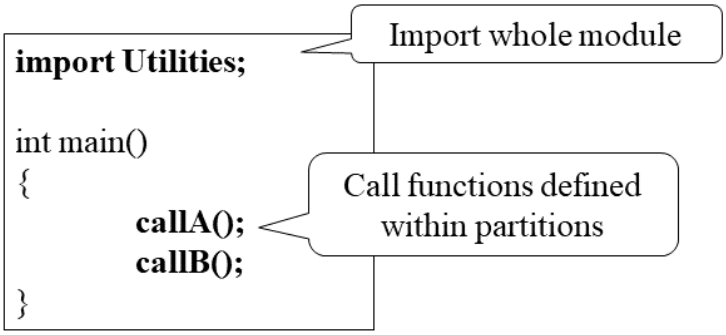
```
export module Utilities;
import :A;
import :B;
export void callA() { do_workA(); }
export void callB() { do_workB(); }
```

Export Module

Import Partition ‘A’

Using Module with Partitions

- To use the module defined in partitions simply import the module:



Reusing Modules

- To reuse the Modules these can be package in static libraries
- Using Microsoft tools a reference can be added to reference the library
- To create the library it needs to be compile with appropriate options

Static Library (Modules)

- Using Visual Studio:
  - Add a new Static Library project
  - To this project add a new item:
    - Module Interface Unit (.ixx)
    - Name this appropriate
    - Add class/functions for export
  - Edit the project properties
    - C/C++/Advanced/'Compile As'
      - Compile as C++ Module Code (/interface)

Appendix 2

Coroutines

Coroutines

- This section gives an overview to this new language feature:
  - Coroutine Introduction
  - Generator
  - Experimental Generator
  - Coroutines and the Stack
  - Subroutines and Coroutines
  - Experimental Libraries
  - Task and co_return

Coroutines Introduction

- The section is only an overview of Coroutines
- C++ 20 provides the language features however the library support will not be provided until C++ 23
- This feature allows asynchronous programming within C++
- Similar or related features have been supported in some languages/frameworks for some time, i.e. C#/.NET and Python

Coroutines - continued

- Coroutines are functions which contain at least one of the new keyword co_yield, co_await, co_return
- Whilst the language features have been added, the library features have not
  - Currently only provided in experimental form, such as (with Visual Studio):
    - std::experimental::generator

Generator

- An example of a coroutine is that of a function using a 'generator'
  - The motivation for this is to allow 'lazy' generation of data, rather than require all of the data to be 'created'/'returned' in one go as below:

```
std::vector<int> sequence(int start, int end) {
    std::vector<int> result(end - start);
    std::iota(result.begin(), result.end(), 1);
    return result;
}
```

Vector populated with increasing sequence

Whole vector returned

Usage of ‘sequence’ function

- The sequence function could be used in a number of ways, depending on the API to be used, such as:

```
std::vector<int> seq = sequence(1, 20);

for (auto& n : seq) {
  std::cout << n << ", ";
}
std::cout << std::endl;
```


Motivation

- The previous example of the ‘sequence’ function returned a vector
- Two particular problems arise:
  - It is of a predetermined fixed size
  - The whole vector is returned
- What if the size was not known or the sequence did not having a predetermined end or was ‘infinite’?

Experimental Generator

- Using the experimental Generator (Visual Studio) (available in C++23), the sequence function could be written as:

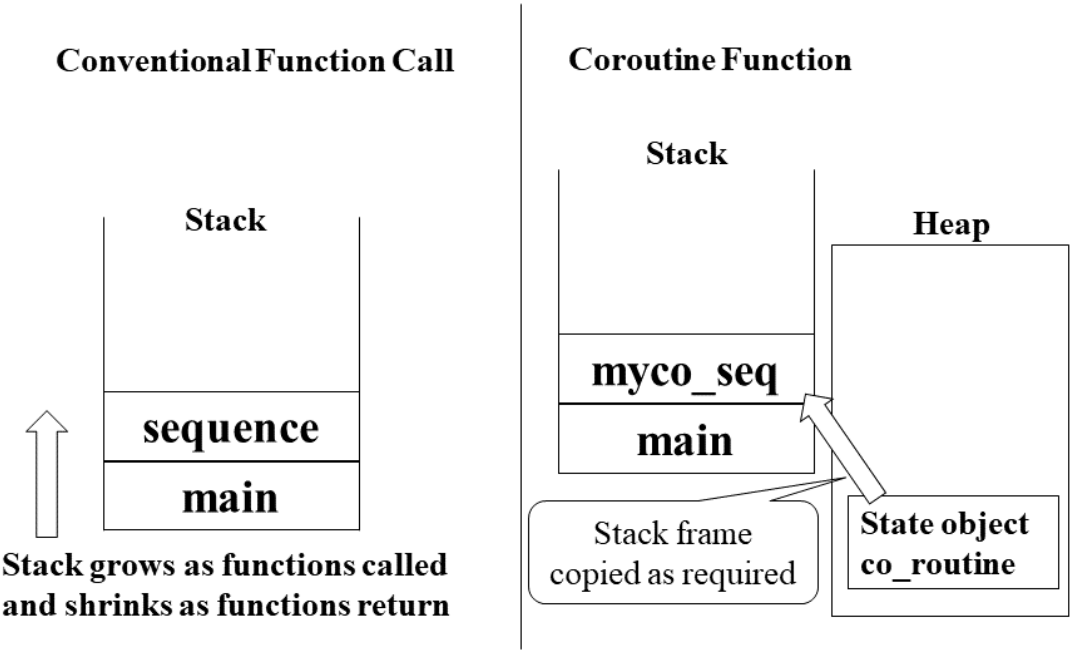
```
#include <experimental/generator>
std::experimental::generator<int> myco_sequence(int start , int end) {
    while (start != end)
        co_yield start++;
}
```

'Call back' on each iteration

- And used:

```
for (auto& n : myco_sequence(1,20) ){
   std::cout << n << ", ";
}
std::cout << std::endl;
```


Coroutines and the Stack



Subroutines and Coroutines

- So far the term function has been used, however the more general term subroutine is sometimes used
  - Subroutines allow:
    - Call
    - Return
  - Coroutines additionally allow:
    - Suspend
    - Resume

    } State needs to be maintained, as it is needed, in or to resumed
- Coroutine is a generalisation of a subroutine!

Coroutines keywords and Restrictions

- The meaning of the various keywords:

Keyword	Descritption
co_await	Suspends and returns control
co_return	Return statement for coroutine
co_yield	Returns value and suspends

- Restrictions:
  - Cannot use variadic arguments, conventional ‘return’ or return auto (or concept)
  - Coroutines cannot be consteval, constexpr, constructors, destructors or main!

Unhandled Exceptions

- Care needs to be taken to avoid unhandled exceptions within coroutines
  - An unhandled exception within the a coroutine is caught resulting in a call to promise.unhandled_exception()
  - promise.final_suspend() is called
  - Attempting to resume this coroutine results in undefined behaviour

Experimental Libraries

- The current language support is intended for library writers, but there are already a number of libraries which can be experimented with
  - Not recommended for release applications at this time (but useful to gain some familiarity)
  - One illustration is from Reza Arjmandi
    - https://github.com/reza-arjmandi/cpp20_coroutines
  - This later library also illustrates the use of the some other features

task and co_return

- Provided a suitable library implementation exist (C++23) the following form of code should be possible:

```
task<int> make_answer(){
   co_return 42;
}

task<std::string> make_question_answer(){
   auto task = make_answer();
   auto answer = co_await task;
   co_return "The answer: " + std::to_string(answer);
}
```


promise_type Implementation

- Implementing the ‘generate’ or ‘task’ class is not straightforward
  - Requires a nested type ‘promise_type’ with a number of member functions:

Member function
get_return_object
initial_suspend
final_suspend
yield_value
unhandled_exception