C++20 Labs

Labs Introduction

This course introduces many of the new features. It is intended that these labs allow exploration of the features largely in isolation. Therefore, depending upon the development environment used, create new projects or new online sessions if using an online compiler. When projects are created check that the appropriate compiler flag is set to support C++20.

Template Features

These labs are an opportunity to try out some of the latest template features.

Abbreviated Syntax

1) Try writing a simple template function to square a numeric value. Call this function and then write the equivalent abbreviated template syntax. Call this function. Output the results.

   Build and test.

1a) Add a numeric constraint to the abbreviated template syntax function.

   Build and test

Lambda Template Syntax

2) Declare a variable to initialize with a generic lambda function. Use ‘decltype’ to discover the type of the template type parameter. Output the type name by using ‘typeid’.

   Write the equivalent lambda function using the template syntax to discover the type of the template type parameter. Output the type name by using ‘typeid’.

   Build and test.

Class Non-type Parameter

3) Define a template function to take template non-type parameter which is a class type.

   Define a suitable class to instantiate the template function. Try adding and changing the members to see what is an acceptable type for use as non-type parameter.

   Build and test.

Lambda as Parameter to template

4) Define a template class taking a template parameter of a function pointer type. Define an alias for the function pointer type.

   Try instantiating this template class with a lambda.

   Build and test.

Lambda Variadic Template Capture

5) Define a lambda function which uses variadic arguments and passes these on to another lambda which captures these variadic arguments.

   Implement the function using a fold to sum an arbitrary number of numbers.

   Build and test.

Templates and type traits

6) Create a new project to experiment with templates and concepts. Add a template function to implement ‘my_max’ which finds the maximum of two values. Try calling this function from within ‘main’ using both ‘int’ and ‘double’ variables. Output the results.

   Build and test.

   Try calling the function with ‘const char *’.

   Build and test. Does this work successfully?

   Add ‘static_assert’ to prevent calling the function if not integral or floating point types, using type traits.

   Try to build.

Templates and SFINAE

7) Continue from the previous lab by using enable_if to provide two versions of the ‘max’ function. One version for numeric values and one for const char *.

   Build and test.

Templates and predefined concept

8) Continue from the previous lab by implementing a 'cmy_max' function using the concept 'floating_point'.

Build and test.

Templates and requires

9) Continue from the previous lab by implementing a 'rmy_max' function using the requires keyword. Use an appropriate expression using type traits to restrict this template function to work with numeric types.

Build and test.

Templates and concepts

10) Define custom concepts for Integral and Floating types, using type traits. Define two template functions, one to output a floating point value and the other to output an integral value. Use the concepts in place of 'typename' within the template definition.

Call these functions from within main().

Build and test.

10a) Try using the concepts after the requires keyword.

Build and test.

Composite concept

11) Define a new concept 'Numeric' which is composite of the previous concepts.

Define a new function to compute cubes and restrict to working with numeric types.

Build and test.

Custom Concepts

12) Define a custom ‘concept’ called ‘Incrementor’, using requires with an appropriate parameter. Within braces define the signatures for incrementing (++).

Define a template function ‘output_values’ to output values from a STL container, using iterators. Iterate to output the values from the container. Use the Incrementor concept to restrict the template parameter using the concepts in place of ‘typename’ within the template definition.

Call this function from within main().

Build and test.

12a) Define a custom ‘concept’ called ‘Derefer’, using requires with an appropriate parameter. Within braces define the signatures for dereferencing (*).

Define a further ‘concept’ for ‘Iterator’ as a composite of ‘Incrementor’ and ‘Derefer’.

Use the Iterator concept to restrict the template parameter using the concepts in place of ‘typename’ within the template definition.

Build and test.

Interface like Concept

13) Define a custom ‘concept’ called ‘Stringable’, using requires with an appropriate parameter. Within braces define the signatures for a to_string member function which returns a std::string.

Define a template function ‘convert’ to call a ‘to_string’ member function on the template parameter. Use the Stringable concept to restrict the template parameter using the concept.

Define a class with the member function ‘to_string’ returning a std::string. Within main create an object of the class and pass this to a call to the ‘convert’ function.

Build and test.

Span

14) Within main declare and initialize an array of integers. Add a conventional loop to iterate through the values, outputting the values to cout.

Build and test.

Move the functionality of outputting the values to a function and call this function passing the array.

Build and test.

Which approach has been used to pass the ‘C’ array? Due to decay, typically the size needs to be passed in addition to the array pointer.

Rewrite the output function to take a span of integers.

Build and test.

Subspan

15) Modify the application to create two subspans and use the function to output the two subspans.

Build and test.

Ranges and Algorithms

16) Create a new project to try out Ranges. Declare and initialize a vector of int. Try out a number of algorithms using the vector, treating it as a range, using ‘std::ranges::’ algorthms (copy, generate).

Build and test.

16a) Try out the traditional algorithm ‘partition’ on a container of numbers (either odd/even or less than a value). Display the two partitions which are produced. Now try out the ranges ‘partition’ algorithm and display the two partitions produced.

Build and test.

16b) Try out ‘find’ algorithm on a container of struct (containing an int), and use a projection in order to select the data member for use within the find.

Build and test.

View Factories – iota_view

17) Use the ‘iota_view’ to allow traversing a vector of int. Create the view and use a range-based for to iterate across the view. Also try using a copy algorithm and ostream_iterator to output the values.

Build and test.

17a) Experiment with a number of other ‘iota’ algorithms to generate sequences.

Build and test.

Sentinel

18) Define a sequence (within and array or suitable container), which has a custom terminator (typically a specific value). Define a sentinel class and its associate operator to indicate termination using the custom terminator. First try using a the traditional for to iterate across a sequence using the iterator and sentinel to output the values. Then use the ranges copy algorithm to only output the values within the sequence.

Build and test.

View Factories – istream_view

19) Use the ‘istream_view’ to allow input from the keyboard. Create the view and use the copy algorithm to copy to a vector of string. Output the vector of strings using a range-based for. Also try using a copy algorithm and ostream_iterator to output the values.

Build and test.

19a) Use the ‘istream_view’ to allow input from an istringstream initialized with a string containing integers (space separated). Create the istream_view and use the copy algorithm to copy to a vector of int. Output the vector of integers.

Build and test.

Find Algorithm

20) Define a function to return a vector of int. Within main try using the returned vector within a std::ranges::find algorithm

    Build and test. What happens?

    Declare a temporary variable to copy the vector and pass this to the find algorithm.

    Build and test.

    Try using find to find a character within a std::string_view.

    Build and test.

Pipelines

21) Declare and initialize a vector of int and use a pipeline to filter out all of the even numbers. Output the results.

    Build and test.

    Debugging. Use a debugger to set a breakpoint to confirm that the filter 'callable' is called when the results are being iterated through.

Pipeline and Range factory

22) Using the range factory 'iota' filter out the all of the odd numbers. Output the results.

    Build and test.

Pipelines and Common

23) Use a collection and the take_while range adapter to obtain the result. Try using the result in a conventional algorithm.

    Try to build.

    Add the range adapter common, to the pipeline.

    Build and run.

Pipelines keys and values

24) Declare and initialize a map of int and string. Use a pipeline to filter for all of the keys. Output the results.

Build and test.

24a) Use a pipeline to filter for all of the values. Output the results.

Build and test.

24b) Use a pipeline to filter for all of the elements using the appropriate ‘index’ with <>. Output the results.

Build and test.

Custom Range Adapter

25) Define a template function taking a range to filter the range to return just the odd values. Use a concept to restrict to a range. Within the function use a pipeline to use ‘filter’. Call this function on a suitable range of values (vector or range view).

Output the results.

Build and test.

25a) Define a template function taking a range to filter the range to return just the odd values. Use a concept to restrict to a range. Within the function use the call to ‘filter’. Call this function on a suitable range of values (vector or range view).

Output the results.

Build and test.

25b) Define a constexpr function returning filter for a range to return just the odd values. Within the function use the call to ‘filter’ just passing a lambda. Use this function in a pipeline for a suitable range of values (vector or range view).

Output the results.

Build and test.