Packaged Task

std::packaged_task

- Defined in <future>
- Encapsulates a task
  - A callable object for the task's code
  - An std::promise for the result of the task
- Provides a higher level of abstraction

std::packaged_task

- Template class
  - The parameter is the callable object's signature

```
// std::packaged_task object
// The callable object takes int and int arguments.
// It returns int
std::packaged_task<int(int, int)> ptask(...);
```

- The constructor takes the callable object as argument

std::packaged_task Interface

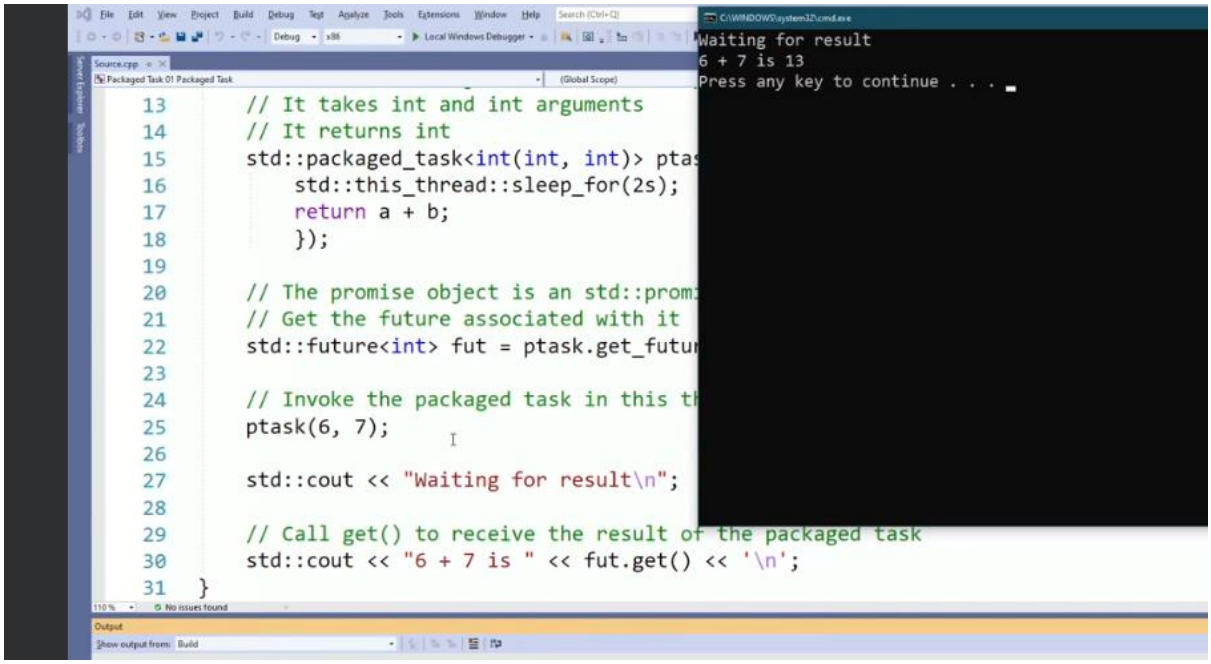
- Functor class
- Overloaded operator()
  - Invokes the callable object
  - Stores the return value in the promise object
- get_future()
  - Returns the std::future object associated with the promise
- std::packaged_task is a move-only class

Using an std::packaged_task Object

- Pass a callable object to the constructor
- The packaged task starts when operator() is called
  - In the same thread, by calling it directly
  - In a new thread, by passing the task to std::thread's constructor
- We call get_future()
- We call get() on the returned future object
  - Or wait() and friends

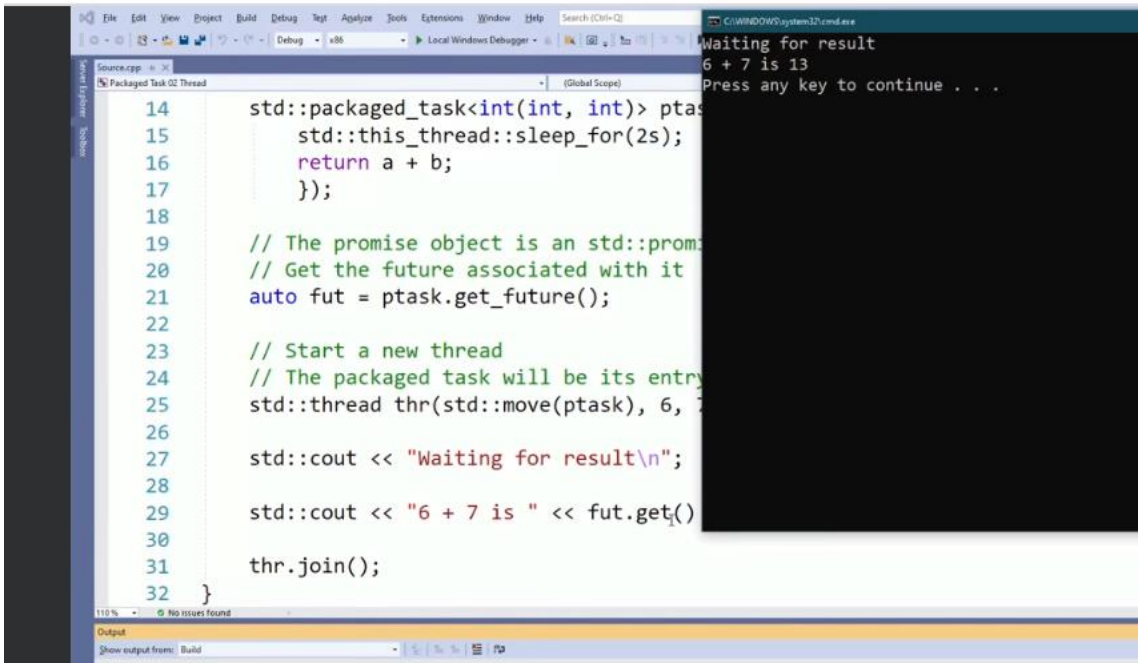
Example of std::packaged_task in Same Thread

```
// Packaged task using lambda expression
std::packaged_task<int(int, int)> ptask( [](int a, int b) {
   return a + b;
});

// The future associated with the packaged_task's promise
std::future<int> fut = ptask.get_future();

// Invoke the packaged task in this thread
ptask(6, 7);

// Call get() to receive the result
fut.get();
```



Example of std::packaged_task in New Thread

```
// Create a thread
// The packaged task will be its entry point
std::thread thr(std::move(ptask), 6, 7);
thr.join();
```



Advantages of std::packaged_task

- Avoids boilerplate code
  - Create std::promise object
  - Pass it to task function
  - Etc

Applications of std::packaged_task

- Create a container of packaged_task objects
  - The threads do not start up until we are ready for them
- Useful for managing threads
  - Each task can be run on a specified thread
  - Thread scheduler runs threads in a certain order
  - Thread pool consists of threads waiting for work to arrive

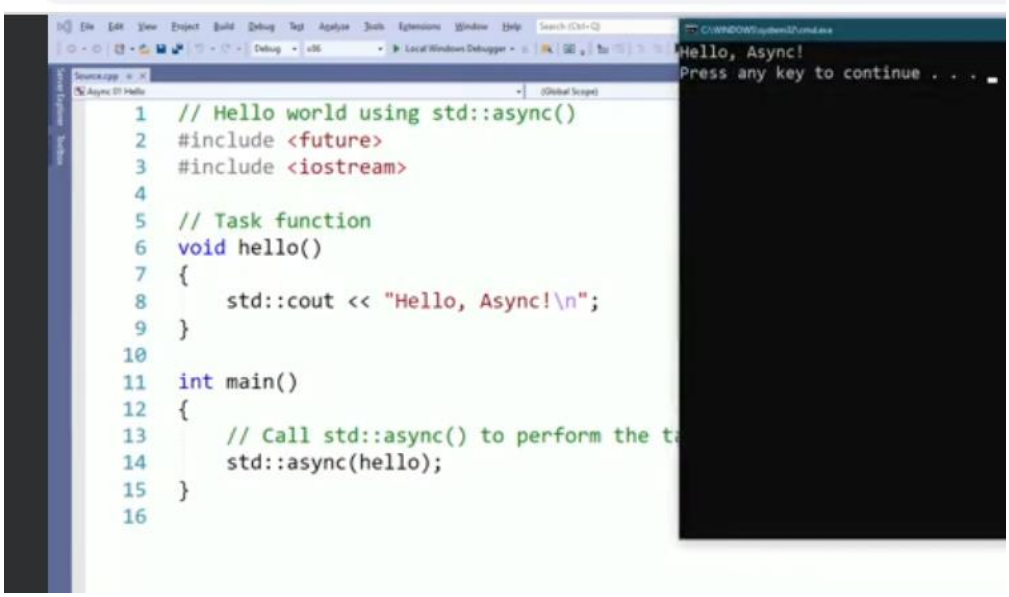
std::async()

std::async()

- Defined in <future>
- Higher-level abstraction than std::thread
  - We can execute a task with std::async() which runs in the background
  - This allows us to do other work while the task is running
  - Alternatively, it can run synchronously in the same thread
- Similar syntax to std::thread's constructor
  - The task function is the first argument
  - Followed by the arguments to the task function

Hello, Async!

```
// The task function
void hello()
{
   std::cout << "Hello, Async!\n";
}

int main()
{
   // Call std::async() to perform the task
   std::async(hello);
}
```

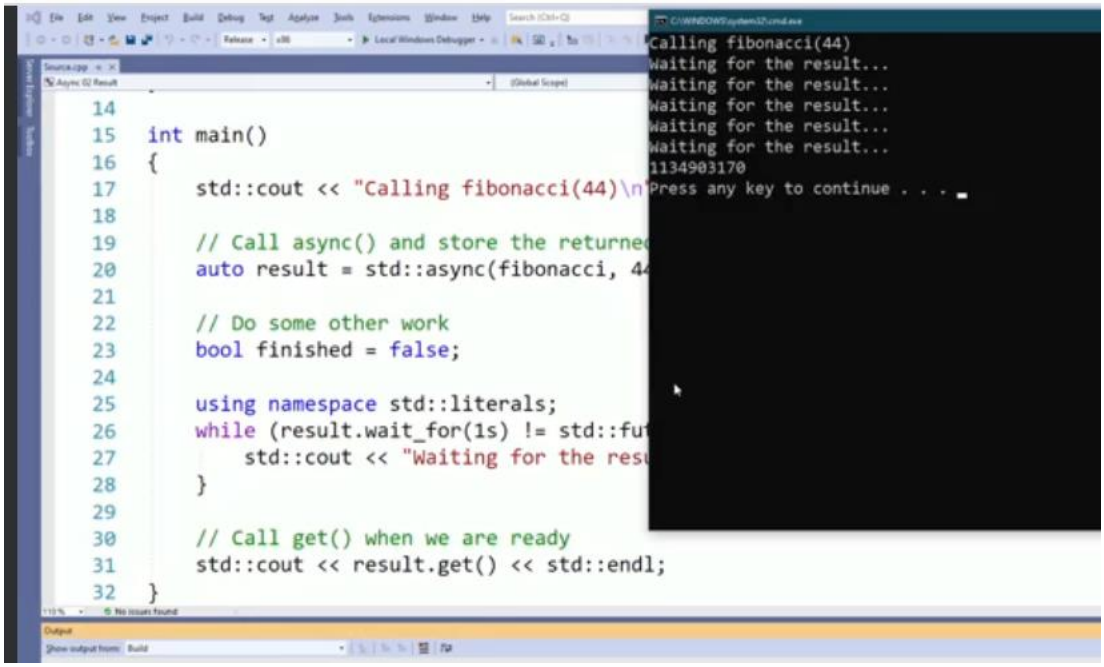


std::async() with std::future

- std::async() returns an std::future object
  - This contains the result of the task
- We can call get() on the future
  - Or wait() and friends
- This can be in a different thread from the call to std::async()

Returning a Value

```
// Task which returns a value
int func()
{
   return 42;
}

// Call async() and store the returned future object
auto result = std::async(func);

// Do some other work
...

 // Call get() when we are ready
int answer = result.get();
```

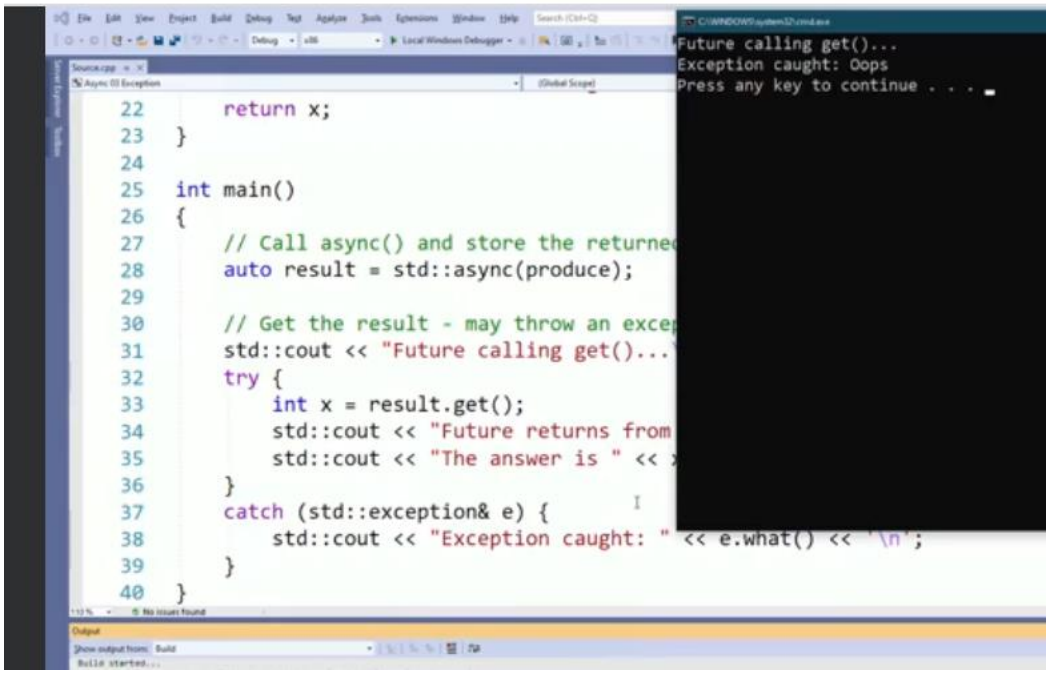


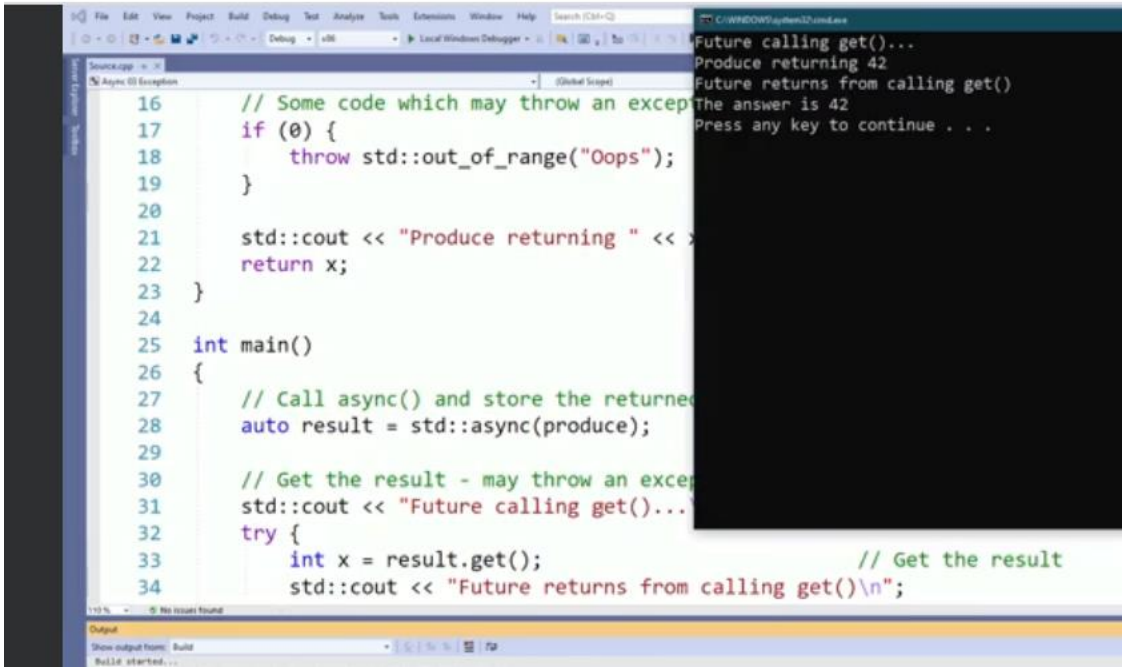
std::async() and Exceptions

- The task may throw an exception
- The exception is stored in the future object
- It will be re-thrown when get() is called
  - Similar to using an explicit std::promise

std::async() and Exceptions

```
int produce()
{
  int x = 42;
  // Code which may throw an exception
  ...
  return x;
}

auto result = std::async(produce);

try {
  // Get the result - may throw an exception
  int x = result.get();
}
catch(std::exception& e) {
  // Handle the exception
  ...
}
```





std::async() and Launch Options

Launch Options

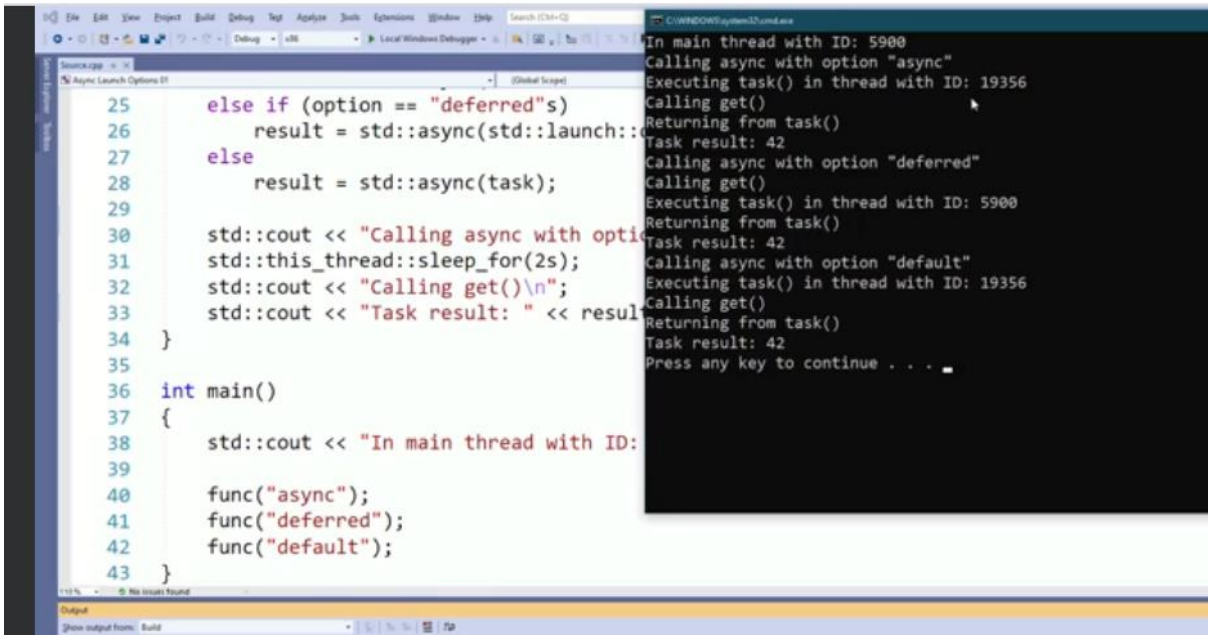
- std::async() may start a new thread for the task
- Or it may run the task in the same thread
- Controlled by the "launch flag"
  - Optional argument to std::async()

Launch Flags

- std::launch::async
  - A new thread is started for the task
  - The task is executed as soon as the thread starts
- std::launch::deferred
  - Nothing happens until get() is called on the returned future
  - The task is then executed ("lazy evaluation")
- If both flags are set
  - The implementation decides whether to start a new thread
  - This is the default

Default Launch Policy

- Lack of certainty
  - The task could execute synchronously with the initiating thread
  - The task could execute concurrently with the initiating thread
  - It could execute concurrently with the thread that calls get()
  - If get() is not called, the task may not execute at all
- Thread-local storage (TLS)
  - We do not know which thread's data will be used



Launch Policy Recommendations

- Use the async launch option if any of these are true
  - The task must execute in a separate thread
  - The task must start immediately
  - The task will use thread-local storage
  - The task function must be executed, even if get() is not called
  - The thread receiving the future will call wait_for() or wait_until()

Launch Policy Recommendations

- Use the deferred launch option if
  - The task must be run in the thread which calls get()
  - The task must be executed, even if no more threads can be created
  - You want lazy execution of the task
- Otherwise, let the implementation choose

Return Value from wait() and Friends

- wait()
  - Returns nothing
- wait_for()
- wait_until()
  - Return std::future_status::ready if the result is available
  - Return std::future_status::timeout if the timeout has expired
  - Return std::future_status::deferred if the result is being lazily evaluated
- In lazy evaluation, the task does not run until get() is called

Choosing a Thread Object

Choosing a Thread Object

- We now have three different ways to execute a task
  - Create an std::thread object
  - Create an std::packaged_task object
  - Call std::async()

Advantages of std::async()

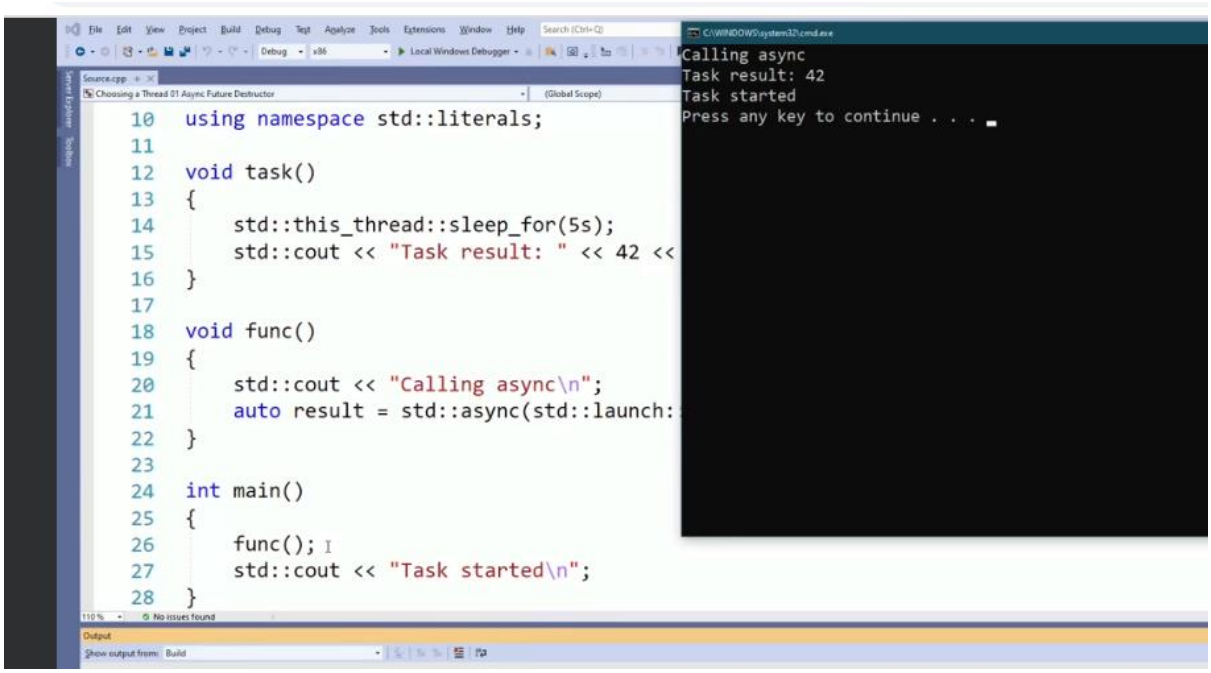
- The simplest way to execute a task
  - Easy to obtain the return value from a task
  - Or to catch any exception thrown in the task
  - Choice of running the task synchronously or asynchronously
- Higher-level abstraction than std::thread
  - The library manages the threads for the programmer
  - And the inter-thread communication
  - No need to use shared data

Disadvantages of async()

- Cannot detach tasks
- A task executed with std::launch::async is "implicitly joined"

```
{
   auto fut = std::async(std::launch::async, hello);
} // Calls ~fut()
```

- The returned future's destructor will block
  - Until the task completes



Advantages of std::packaged_task

- The best choice if we want to represent tasks as objects
  - e.g. to create a container of tasks
- A lower-level abstraction than std::async()
  - Can control when a task is executed
  - Can control on which thread it is executed

Advantages of std::thread

- The most flexible
  - Allows access to the underlying software thread
  - Useful for features not supported by standard C++
  - Can be detached

Recommendations

- For starting a new thread in general
  - Use std::async()
- For containers of thread objects
  - Use std::packaged_task
- For starting a detachable thread
- For more specialized requirements
  - Use std::thread

C++ Asynchronous Programming Limitations

- Lacks a number of important features
  - Continuations - "do this task, then do that task"
  - Only supports waiting on one future at a time
  - Waiting on multiple threads has to be done sequentially
  - Concurrent queue
- These were planned for C++20, but were not included
  - Were to be implemented in C++23 using "executors"
  - Executors will not be in C++23
  - Perhaps in C++26?

Third-party Libraries

- Microsoft Parallel Patterns Library
  - Windows only
- Apple Grand Central Dispatch
  - Open source, runs on Linux and Android
- Intel oneAPI Thread Building Blocks
  - Open source, runs on many platforms
- HPX
  - Open source, runs on many platforms

Parallelism Overview

Concurrency

- Sometimes useful to distinguish from "parallelism"
- Describes conceptually distinct tasks
  - Separation of concerns
  - Can run on a single core
- These tasks often interact
  - Wait for an event
  - Wait for each other
- The tasks often overlap in time
- Concurrency is a feature of the program structure

Analogies to Concurrency

- Musicians in a jazz band
- Team sport
- Traffic at a junction (intersection)

Parallelism

- The tasks are identical
- They all run at the same time
  - Run on multiple cores to improve scalability
- These tasks run independently of each other
- Parallelism is a feature of the algorithm being run

Analogies to Parallelism

- Competitive individual sport

Explicit Parallelism

- The programmer specifies how to parallelize the work
  - e.g. divide the data into four parts
  - Then start four threads to process each part
- Pros and cons
  - Involves more work for the programmer
  - Can produce better performance
  - Not scalable
- Mainly useful when writing for specific hardware
  - e.g. a game console
- Or if the work naturally divides into a fixed number of tasks

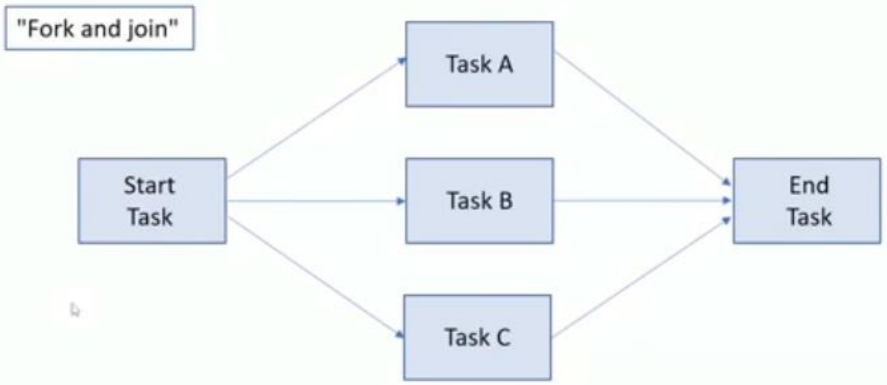
Implicit Parallelism

- The decision is left to the implementation
  - Makes best use of available resources
  - Usually the best option

Task Parallelism

- Distributed processing
  - Also known as "Thread-level Parallelism (TLP)"
- Split a large task into smaller tasks
- The sub-tasks are run concurrently on separate threads
  - e.g. Task A on core 1, Task B on core 2, Task C on core 3
- e.g. Database server runs many threads to reduce latency
  - A thread is waiting to access data on disk
  - Other threads can do useful work

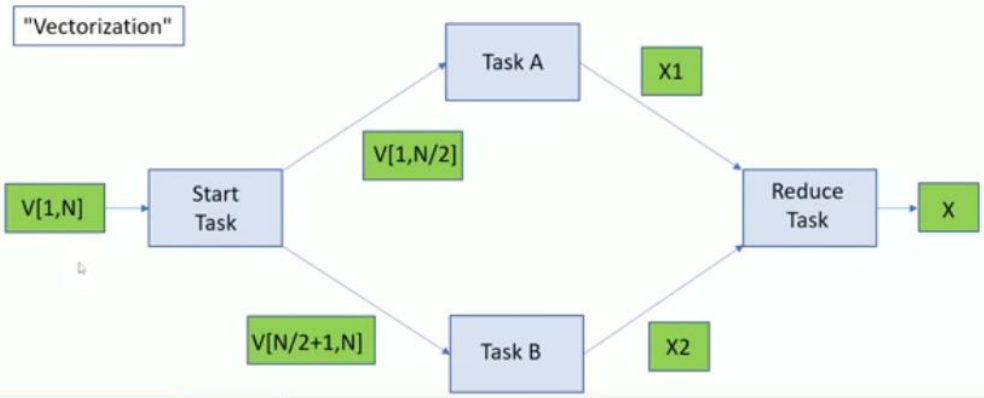
Task Parallelism



Data Parallelism

- Distributed data
  - A data set is divided up into several subsets
  - Process all the subsets concurrently
- Each thread works on one of the subsets
  - e.g. Core 1 processes the first half of the data
  - Core 2 processes the second half
- A final "reduce" step
  - Collects the result for each subset
  - Combines the partial results into the overall result

Data Parallelism



Data Parallelism

- Also known as "vector processing" or vectorization"
  - Used in Graphic Processor Units (GPU)
- Modern CPUs have support for vectorization
  - A single instruction can operate on multiple arguments
  - Known as Single Instruction/Multiple Data architecture or "SIMD"
  - x86 provided SSE family which performs 128-bit SIMD instructions
  - Superseded by 256-bit AVX family

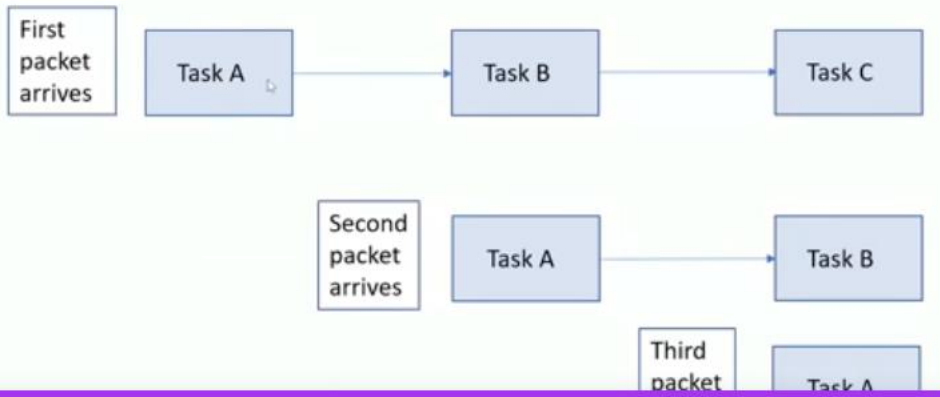
Locality of Data References

- Data parallelism can improve data locality
  - e.g. a program processes 20MB of data
  - A core has 4MB of cache
- Each core processes 1/5 of the data
- All the data that each core needs is in cache
  - No fetches from RAM
  - No interaction with cache controller

Pipelining

- Dependent tasks
  - Task B requires output from Task A, task C requires output from B
  - B cannot start until A has completed and produced its output
- If A, B and C are processing a stream of data
  - A processes first item in stream
  - B starts processing A's output
  - A starts processing the next item

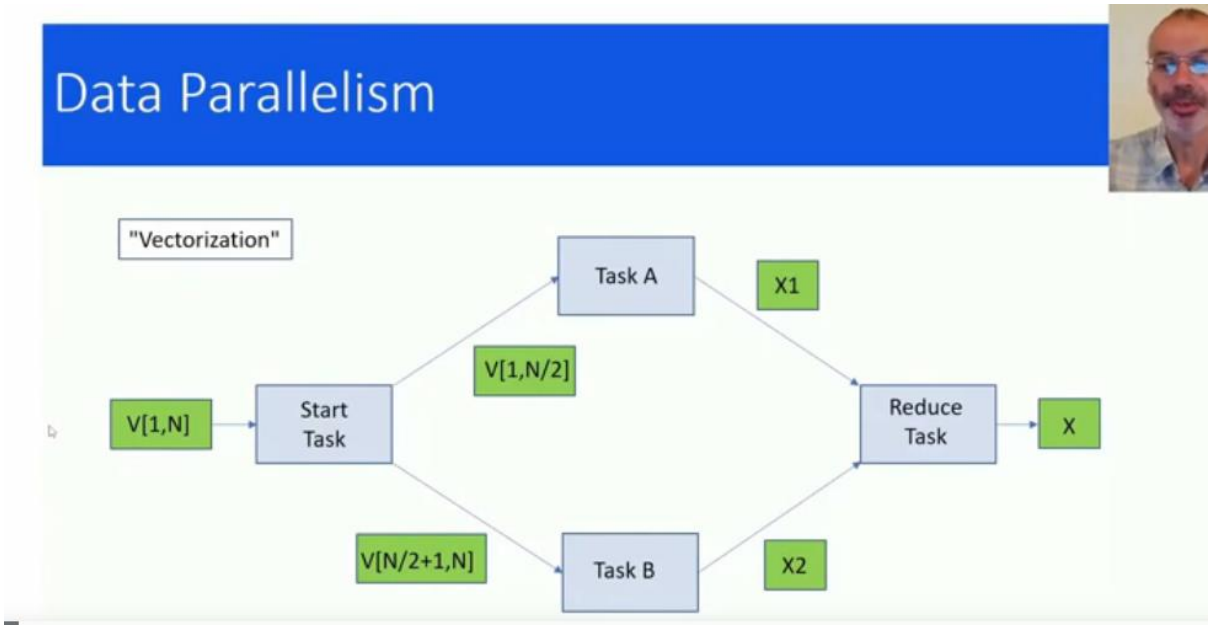
Pipelining

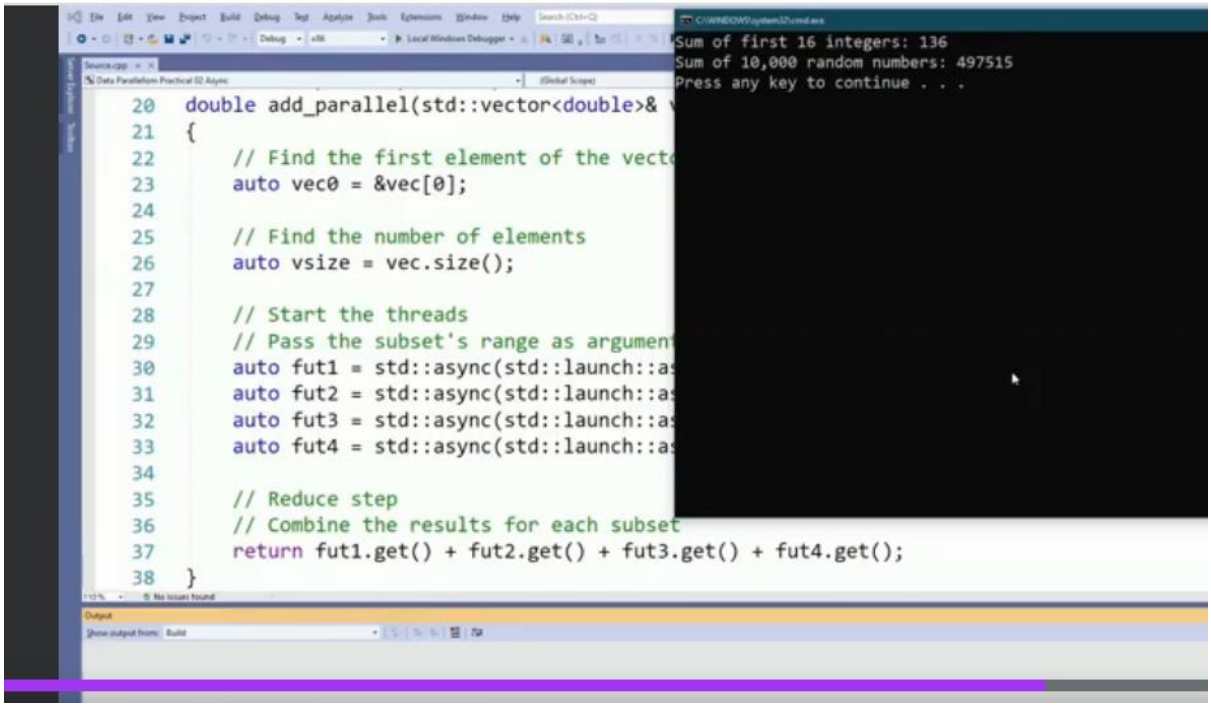


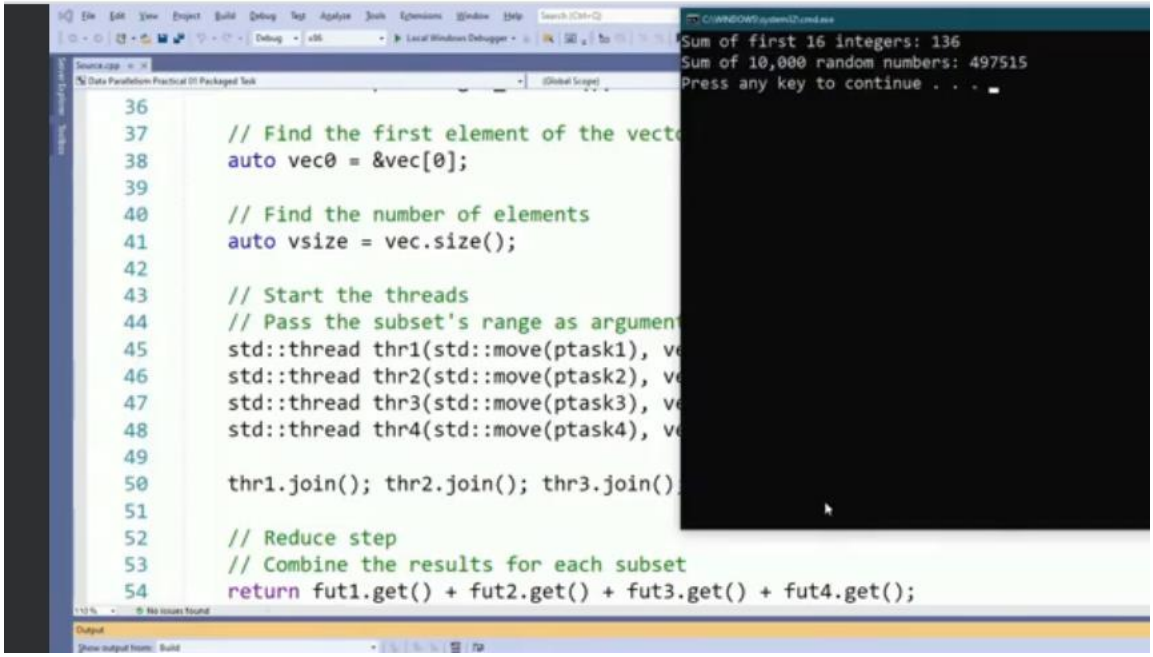
Parallelism

- Task parallelism
- Data parallelism
- Pipelining
  - Perform dependent tasks sequentially
  - Process data concurrently
- Graph parallelism
  - Similar to pipelining, but with an arbitrary graph of dependencies

Data Parallelism Practical


Data Parallelism
"Vectorization"
Task A
X1
V[1,N/2]
V[1,N]
Start Task
Reduce Task
X
V[N/2+1,N]
Task B
X2


Sum of first 16 integers: 136
Sum of 10,000 random numbers: 497515
Press any key to continue . . .
20 double add_parallel(std::vector<double>& v
21 {
22 // Find the first element of the vecto
23 auto vec0 = &vec[0];
24
25 // Find the number of elements
26 auto vsize = vec.size();
27
28 // Start the threads
29 // Pass the subset's range as argument
30 auto fut1 = std::async(std::launch::as
31 auto fut2 = std::async(std::launch::as
32 auto fut3 = std::async(std::launch::as
33 auto fut4 = std::async(std::launch::as
34
35 // Reduce step
36 // Combine the results for each subset
37 return fut1.get() + fut2.get() + fut3.get() + fut4.get();
38 }



Standard Algorithms Overview

Standard Algorithms

- A set of functions in the Standard Library
  - Implement classic algorithms, such as searching and sorting
  - Plus populating, copying, reordering etc
  - Operate on containers and sequences of data
- Most are in <algorithm>
  - A few are in <numeric>

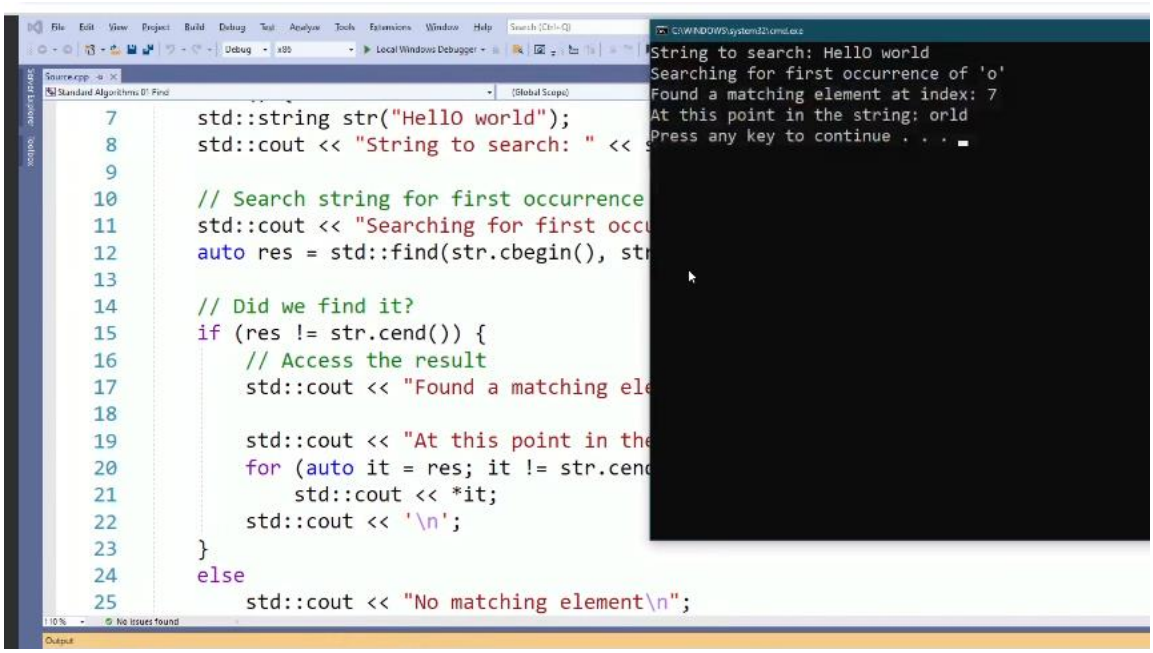
Algorithm Execution

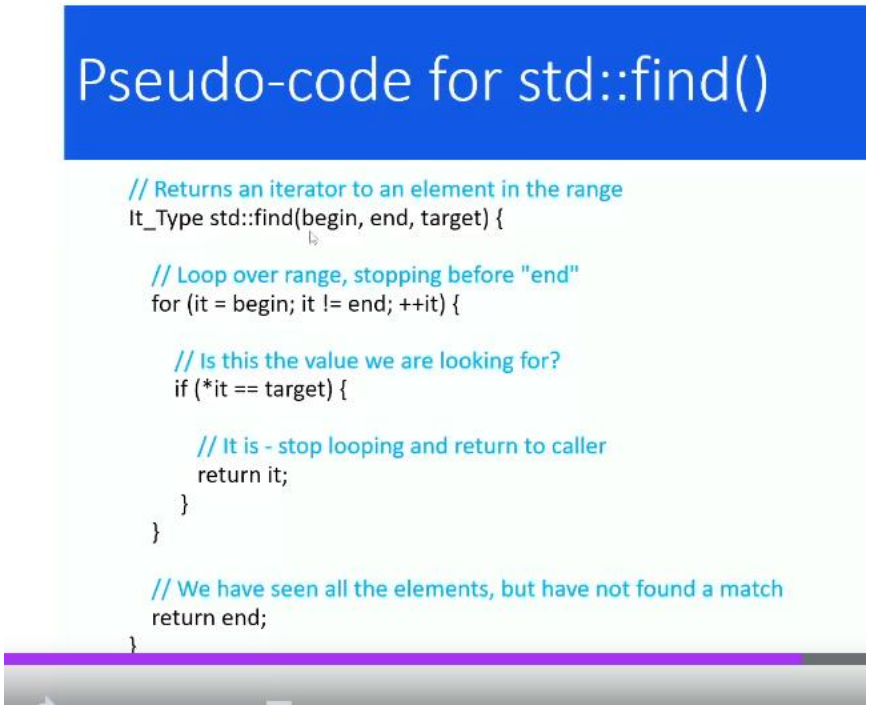
- Function call which takes an iterator range
  - Usually corresponds to a sequence of elements in a container
  - Often begin() and end(), to process the entire container
- Iterates over the range of elements
- Performs an operation on the elements
- Returns either
  - An iterator representing an element of interest
  - The result of the operation on the elements

std::find() Algorithm

- Returns an iterator to the first matching element
- Or the end of the range if no matching element

```
std::string str("Hello world");

// Search for the first occurrence of 'o'
auto res = std::find(str.cbegin(), str.cend(), 'o');

// Did we find it?
if (res != str.cend()) {
  // Access the result
  // ...
}
```


String to search: HelIO world
Searching for first occurrence of 'o'
Found a matching element at index: 7
At this point in the string: orld
Press any key to continue . . .
7 std::string str("HelIO world");
8 std::cout << "String to search: " <<
9
10 // Search string for first occurrence
11 std::cout << "Searching for first occ
12 auto res = std::find(str.cbegin(), st
13
14 // Did we find it?
15 if (res != str.cend()) {
16 // Access the result
17 std::cout << "Found a matching el
18
19 std::cout << "At this point in th
20 for (auto it = res; it != str.cen
21 std::cout << *it;
22 std::cout << '\n';
23 }
24 else
25 std::cout << "No matching element\n";


Pseudo-code for std::find()
// Returns an iterator to an element in the range
It_Type std::find(begin, end, target) {
// Loop over range, stopping before "end"
for (it = begin; it != end; ++it) {
// Is this the value we are looking for?
if (*it == target) {
// It is - stop looping and return to caller
return it;
}
}
// We have seen all the elements, but have not found a match
return end;
}

Predicates

- Many algorithms use a "predicate"
  - A function which returns bool
- std::find() uses the == operator
  - Compares each element to the target value
- The == operator for char is case-sensitive
  - 'o' is not regarded as equal to 'O'

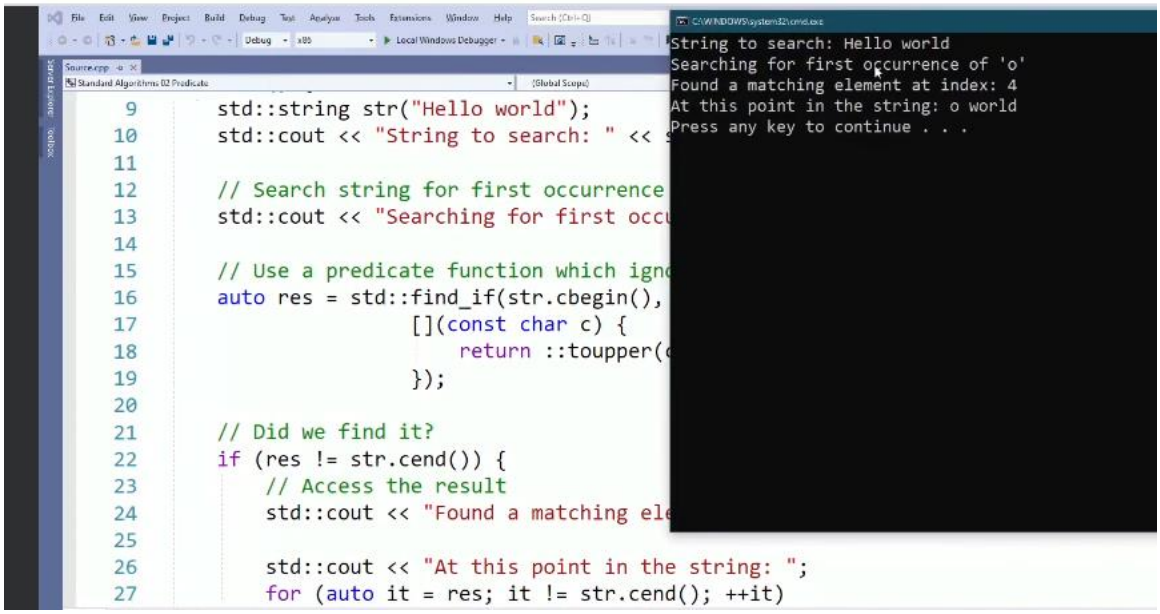
std::find_if()

- std::find_if() allow us to supply our own predicate
  - Pass a callable object as an extra argument
  - Allow us to change the definition of "equality"
- The predicate
  - Takes an argument of the element type
  - Returns a bool
- We will write a predicate which ignores case

std::find_if() Example

- Use a lambda expression for the predicate

```
// Use a predicate function which ignores case
auto res = std::find_if(str.cbegin(), str.cend(),
                        [](const char c) {
                            return ::toupper(c) == 'O';
                        }
);
```



Execution Policies

Standard algorithms

- C++1998 introduced a new set of functions to the Standard Library
  - These operate on containers and sequences of data
  - They implement classic algorithms, such as searching and sorting
  - Plus populating, copying, reordering etc
- They are mostly in <algorithm>, although a few are in <numeric>
  - Usually they take an iterator range which corresponds to the data sequence
  - Often they take a callable object which is "applied to" the elements in the sequence
- An algorithm execution performs a series of operations on the elements
  - modification
  - swap
  - comparison
  - function call application
  - etc

Code execution and parallelism

- Modern hardware supports four different ways of executing code
- Sequential
  - A single instruction processes a single data item, e.g. traditional loop
- Vectorized
  - A single instruction processes several data items at the same time
  - Requires suitable data structure and hardware
- Parallelized
  - Several instructions each process a single data item at the same time
  - Requires suitable algorithm
- Parallelized + vectorized
  - Several instructions each process several data item at the same time
  - Requires suitable algorithm, data structure and hardware

Execution policy

- C++17 added "execution policies"
- These let us choose how a Standard algorithm call should be executed
  - seq - do not use parallel execution
  - par - use parallel execution
  - par_unseq - use parallel and vectorized execution
  - unseq - use vectorized execution (C++20)
- These are only "requests" and may be ignored, e.g.
  - If parallel and/or vectorized execution is not supported
  - If not enough system resources available to do it efficiently
  - If parallel and/or vectorized version has not been implemented

Execution policy objects

- These global execution policy objects are in <execution>, in the std::execution namespace
- To specify the execution policy, we pass the appropriate object as the first argument of the algorithm call:

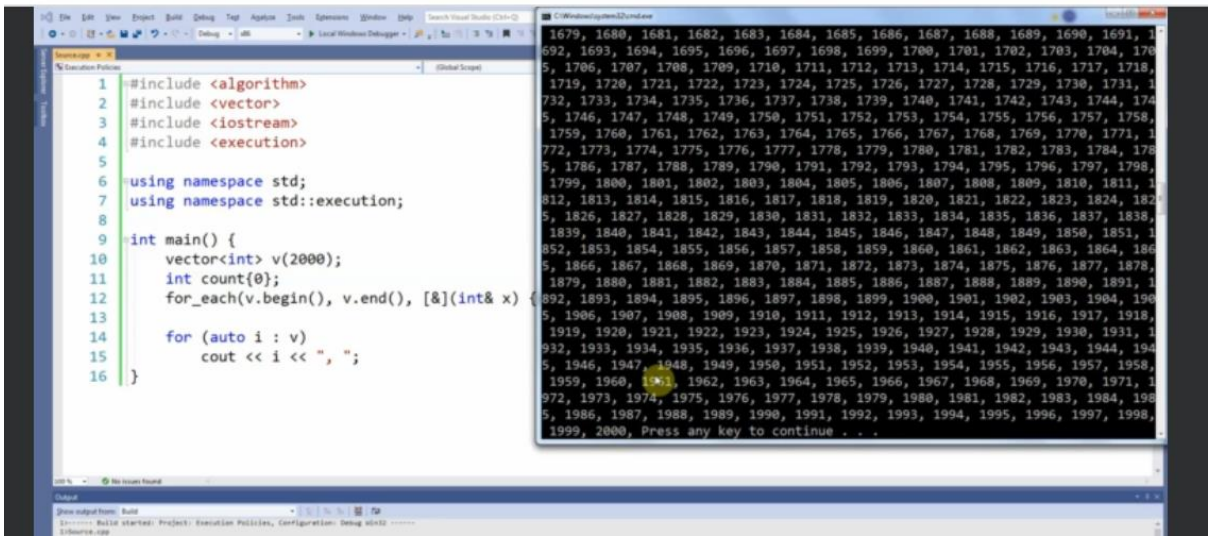
```
sort(v.begin(), v.end());                  // Non-policy (sequential)
sort(seq, v.begin(), v.end());             // Sequential
sort(par, v.begin(), v.end());             // Parallel
sort(par_unseq, v.begin(), v.end());       // Parallel and vectorized
sort(unseq, v.begin(), v.end());           // Vectorized (C++20)
```

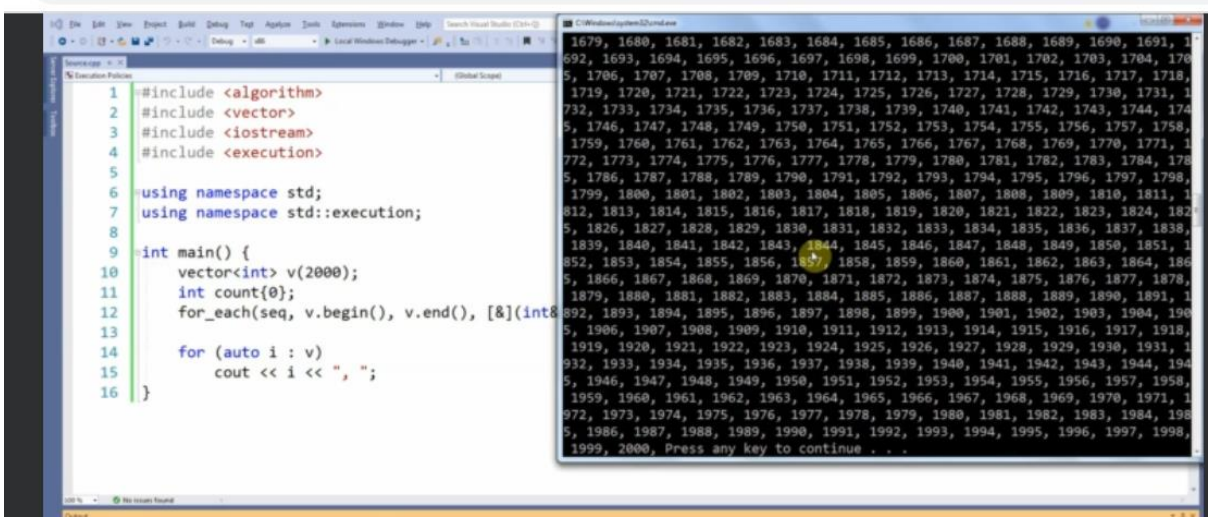
Sequenced execution

- With sequenced policy execution, all the operations in the algorithm execution are performed on a single thread
  - i.e., the thread which calls the algorithm
- Operations will not be interleaved, but may not necessarily be executed in a specific order

```
vector<int> v(2000);
int count {0};
for_each(v.begin(), v.end(), [&] (int& x) { x = ++count; });          // v -> {1, 2, ..., 2000}
for_each(seq, v.begin(), v.end(), [&] (int& x) { x = ++count; });
```

- In the second case, the elements of v will have the same values as in the first case, but may not be in the same order



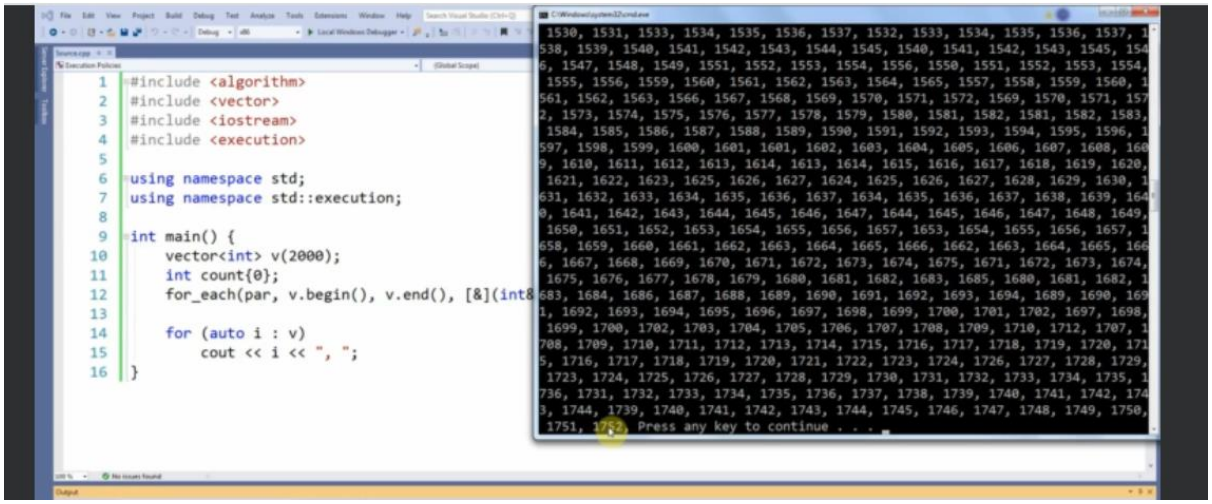


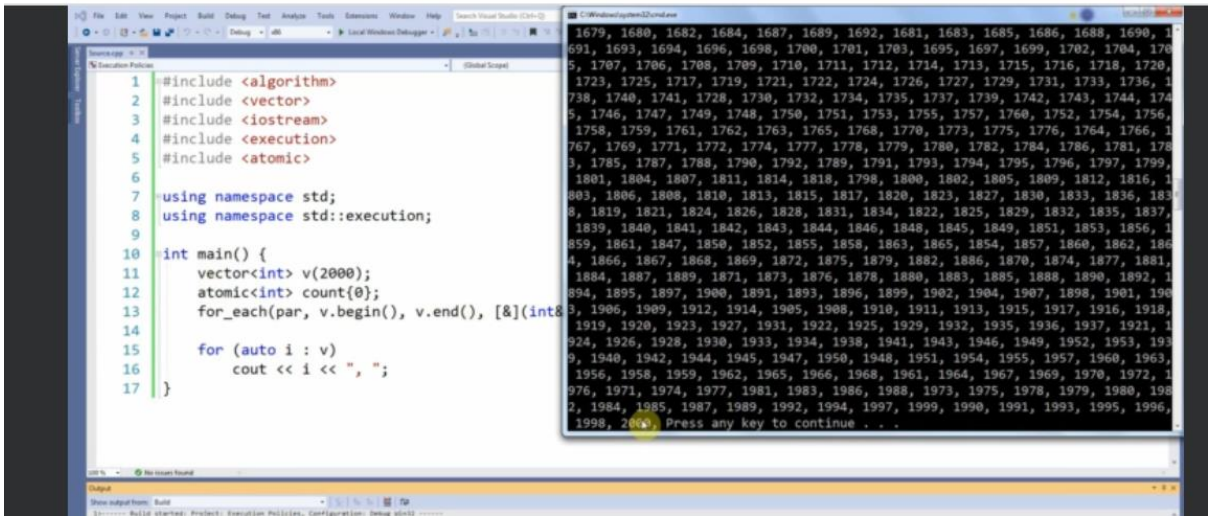
Parallel execution

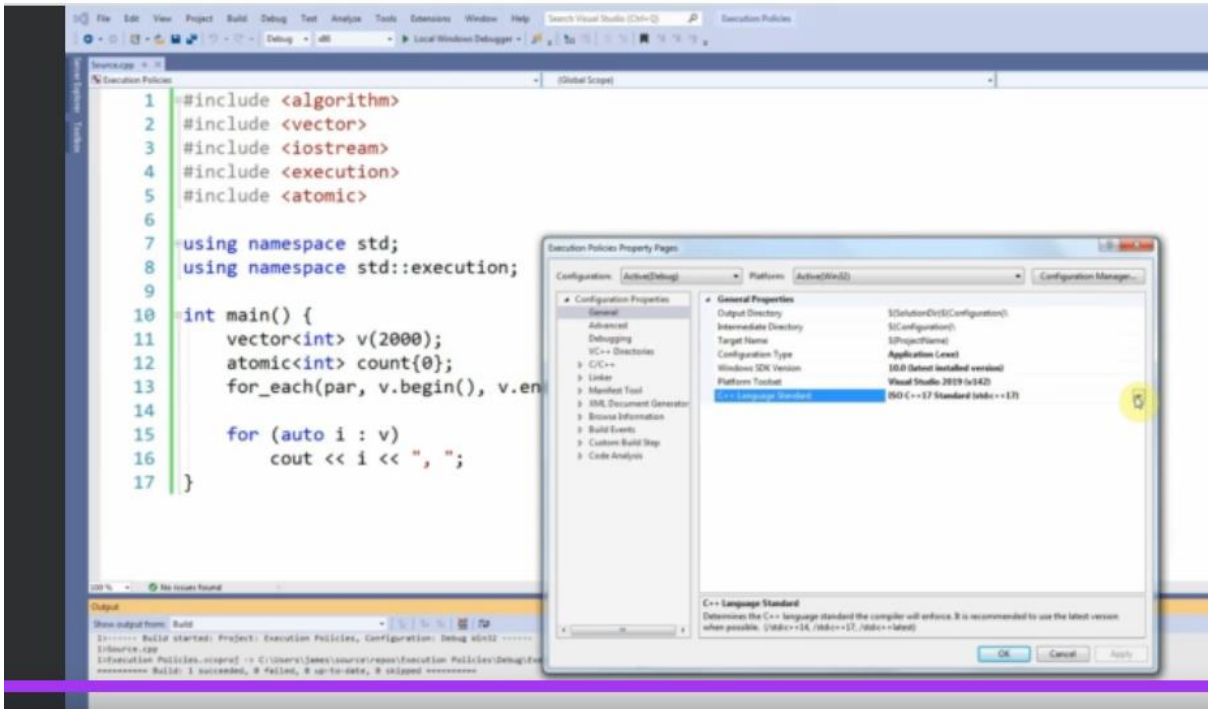
- The executions are performed in parallel across a number of threads
  - This may include the thread that called the algorithm function
- An operation will run on the same thread for its entire duration
- Operations performed on the same thread will not be interleaved, but may not necessarily be executed in a specific order
- The programmer must prevent data races

```
vector<int> v(2000);
int count {0};
for_each(par, v.begin(), v.end(), [&] (int& x) { x = ++count; });     // Data race!
```

- Here, the variable count is shared between threads without any protection

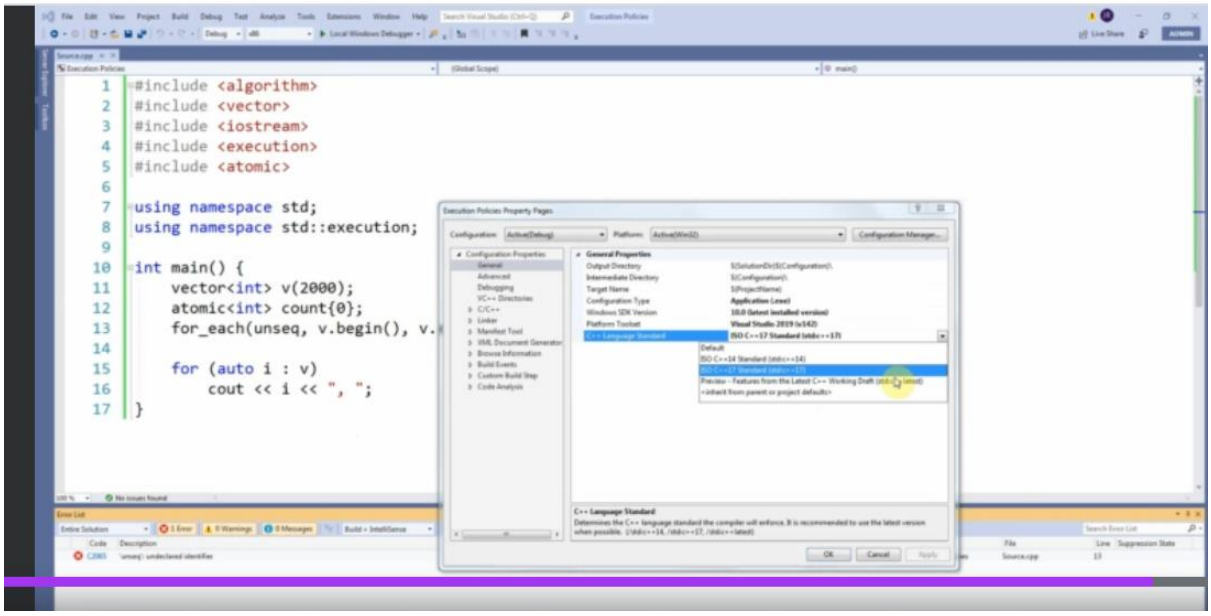






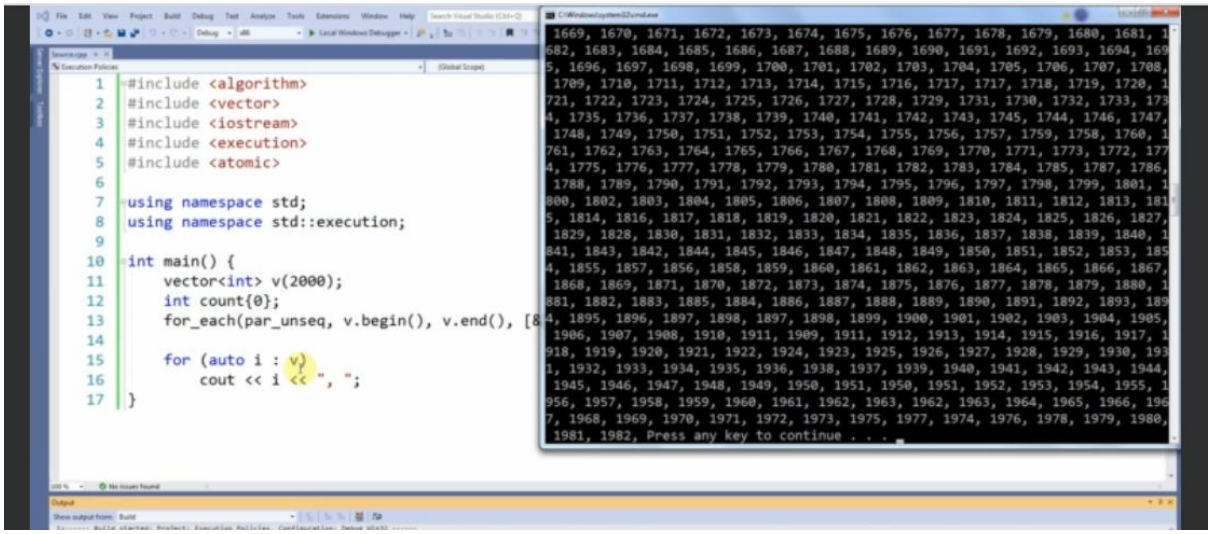
Unsequenced execution

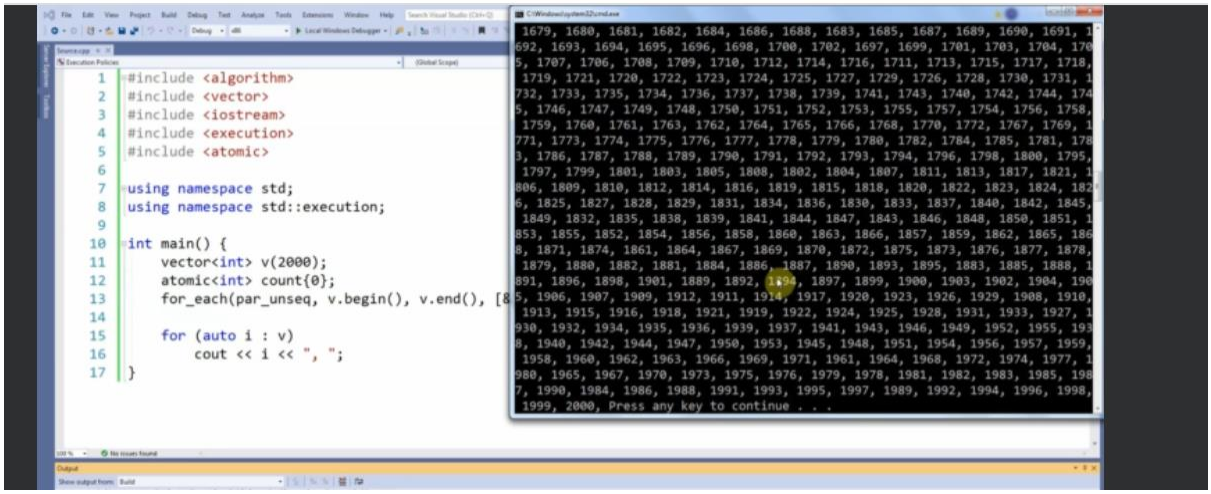
- The operations are performed on a single thread
  - i.e., the thread which calls the algorithm
- Operations will not be interleaved, but may not necessarily be executed in a specific order
- The programmer must avoid any modification of shared state between elements or between threads
  - Memory allocation and deallocation
  - Mutexes, locks and other forms of synchronization



Parallel unsequenced execution

- The executions are performed in parallel across a number of threads
  - This may include the thread that called the algorithm function
- An operation may be migrated from one thread to another
- Operations performed on the same thread may be interleaved and may not necessarily be executed in a specific order
- The programmer must avoid data races
- The programmer must avoid any modification of shared state between elements or between threads
  - Memory allocation and deallocation
  - Mutexes, locks and other forms of synchronization





Algorithms and Execution Policies

Algorithms and execution policies

- Most standard algorithms were respecified in C++17 to support execution policies
  - There are a few exceptions which are naturally sequential (e.g. equal_range, iota)
- Some algorithms in <numeric> which are specified as sequential were implemented with policies and given new names
  - accumulate -> reduce
  - partial_sum -> inclusive_scan, exclusive_scan
- A new "fused" algorithm
  - transform + inner_product -> transform_reduce

Algorithms and exceptions

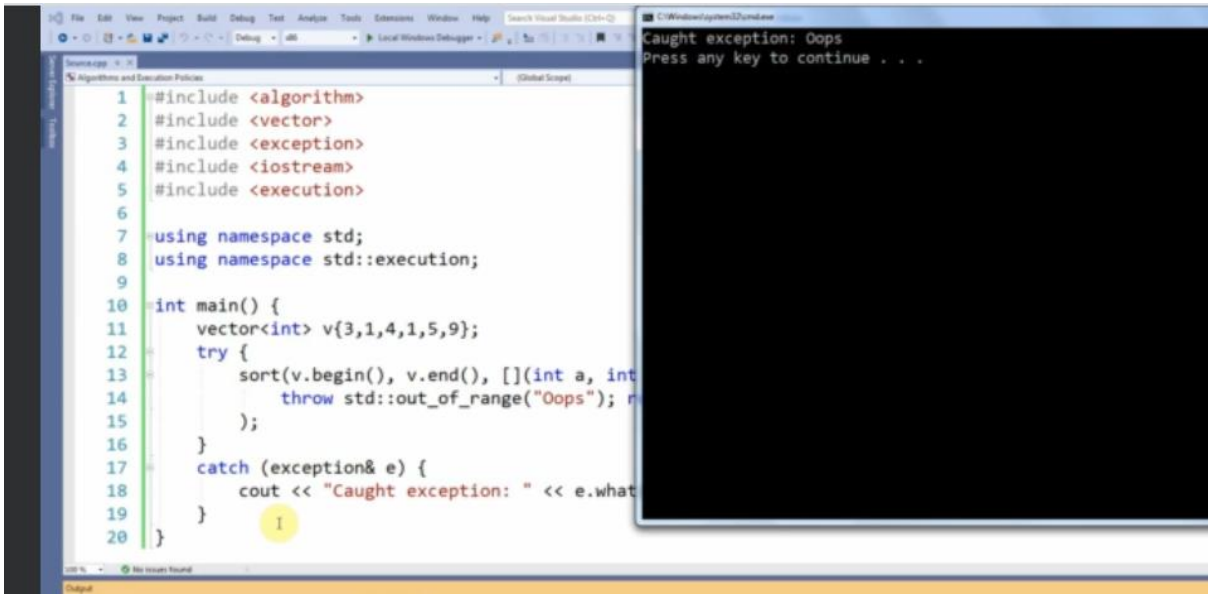
- When an exception is thrown during a non-policy algorithm call, it can be handled in a try-catch block

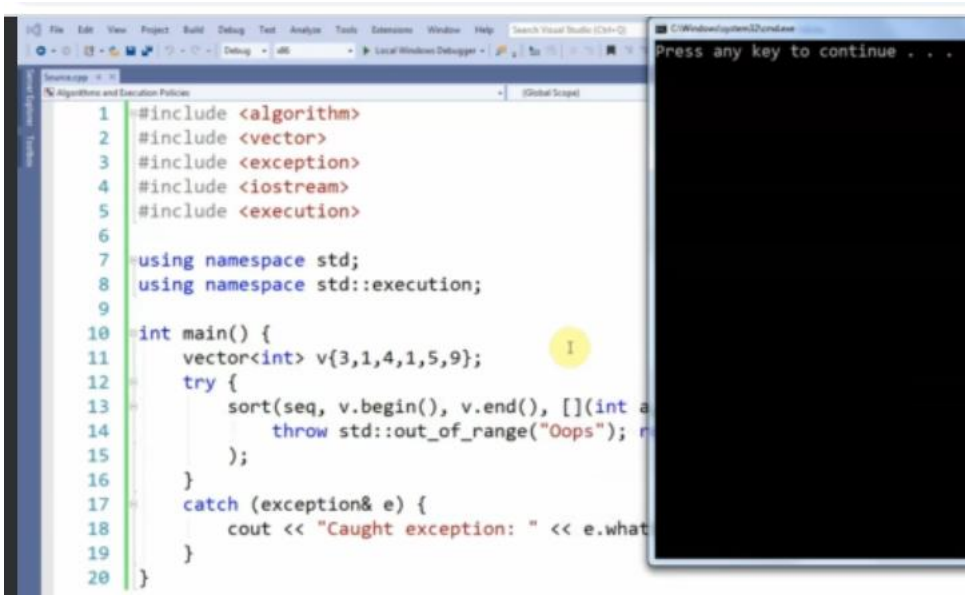
```
try {
   sort(v.begin(), v,end(), [] { ... } );      // Lambda expression, functor, etc which could throw
}
catch (...) {
   ...                                         // Handle exception
}
```

- This is not possible with an execution policy, as there may be multiple threads and each thread has its own stack

Execution policies and exceptions

- Algorithms with execution policies behave differently
- If an uncaught exception is thrown, std::terminate() is called
- e.g. an algorithm call which applies a function on every element
  - If the function throws an exception, and handles it itself, execution continues
  - If the function does not handle it, or if it rethrows the exception, execution ends
- The algorithm call itself may throw
  - If temporary memory resources are required and none are available, an std::bad_alloc exception will be thrown





Disadvantages of execution policies

- May not have any effect
  - Many compilers have not fully implemented algorithms with execution policies
  - Are allowed to fall back to sequential algorithms
- Involves extra overhead
  - The algorithm has to set up threads for the parallel processing and manage them
- Different complexity requirements
  - The C++ Standard often has complexity requirements for non-policy algorithms
    - e.g. number of swaps must be equal to the number of elements (or lower)
  - Algorithms which parallelize well often perform more operations than the sequential equivalent, but reduce overall execution time
  - Parallel algorithms may have weaker complexity requirements, or even none at all

When to use an execution policy?

- Generally, it is best not to use an execution policy
  - "Premature optimization is the root of all evil" - Knuth
- Do not use an execution policy if
  - The task is essentially sequential
  - Operation order is important
  - The algorithm call throws exceptions, and immediate termination is not acceptable
  - The code involves data races, and the synchronization overhead is worse than the cost of not using an execution policy
- Consider using an execution policy if
  - Measurement shows a worthwhile improvement in performance
  - It can be done safely and correctly

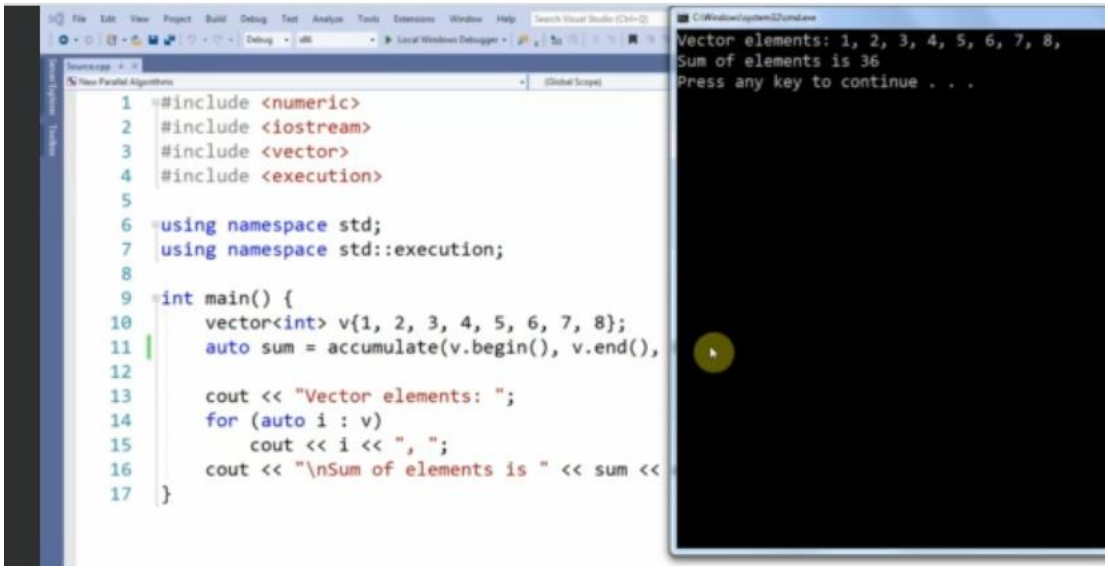
Which execution policy to use?

- Sequenced execution. Mainly used for debugging. Same as non-policy, but
  - Allows out of order execution
  - Terminates on exceptions
- Parallel unsequenced execution. Has the most potential to improve performance, but the strictest requirements. Use when
  - Data races cannot occur
  - Code does not modify shared state
- Parallel execution. Use when vectorization is not safe
  - Data races cannot occur, but code modifies shared state
- Unsequenced execution. Can be used with single threading
  - Code does not modify shared state

New Parallel Algorithms

std::accumulate()

- std::accumulate() adds each element to an initial value and returns the sum
- By default, the operator + is used to perform the addition
- To perform a different operation, we can pass a callable object as an additional fourth argument



std::accumulate() execution

- std::accumulate() is specified to execute sequentially

```
vector<int> v {1, 2, 3, 4};
auto sum = accumulate(v.begin(), v.end(), 0);        // Sum elements of v using initial value 0
```

- The execution is performed as

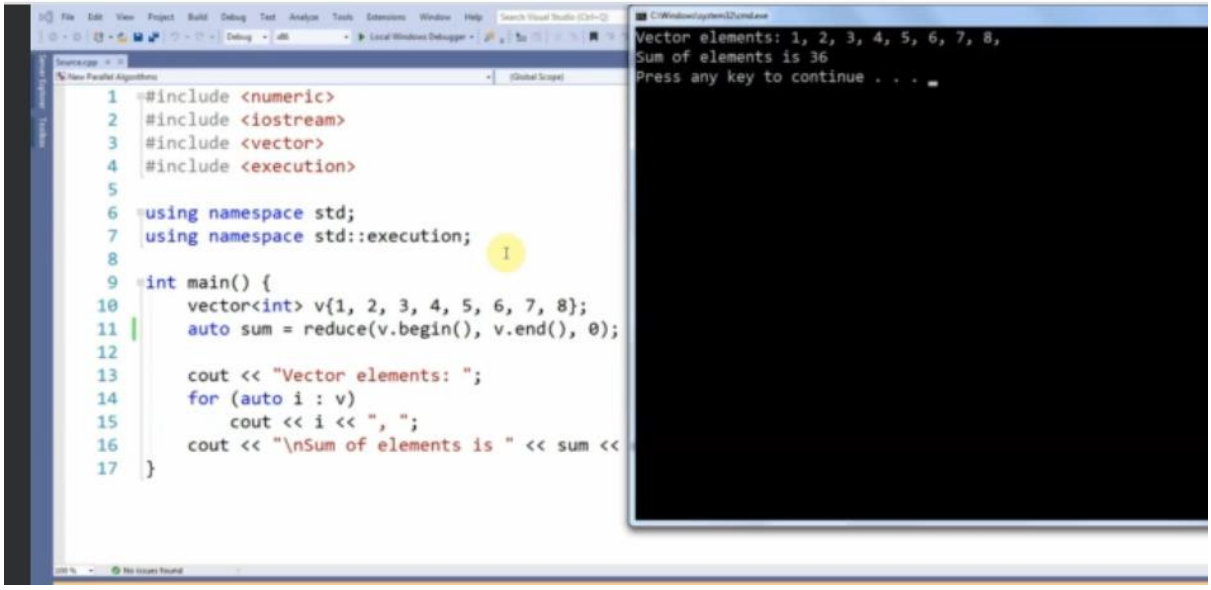
```
(((1 + 2) + 3) + 4)        // Must be performed in left-to-right order, one addition at a time
```

- This cannot be parallelized
  - Each operation must be completed before the next one can start
- This cannot be vectorized
  - Each operation can only take two arguments

std::reduce() and parallelization

- std::reduce() is a version of std::accumulate() which supports execution policies

```
vector<int> v {1, 2, 3, 4, 5, 6, 7, 8};

// Can perform one unsequenced addition (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8)
auto sum = reduce(seq, v.begin(), v.end(), 0);

// Can perform four parallel additions ((1 + 2) + (3 + 4) + (5 + 6) + (7 + 8))
auto sum = reduce(par, v.begin(), v.end(), 0);
```



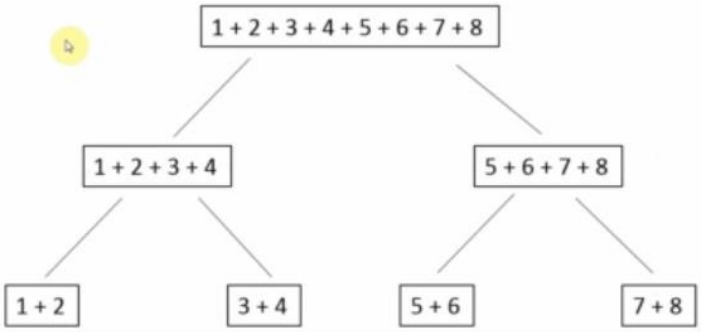
std::reduce() and vectorization

- std::reduce() also supports the other execution policies

```
vector<int> v {1, 2, 3, 4, 5, 6, 7, 8};

// Could be implemented as two vectorized additions (1 + 2 + 3 + 4) + (5 + 6 + 7 + 8)
auto sum = reduce(unseq, v.begin(), v.end(), 0);

// Could be implemented as parallel vectorized additions ((1 + 2 + 3 + 4) + (5 + 6 + 7 + 8))
auto sum = reduce(par_unseq, v.begin(), v.end(), 0);
```

std::reduce

- Reduce with parallel execution resembles a tree of operations



std::reduce() restrictions

- If reordering the operations or regrouping them alters the result, reduce() will not give the correct answer
- The operator used must be associative and commutative
  - Commutative - order of operations is not important
  - x + y == y + x
  - Associative - grouping operations does not change the result
  - (x + y) + z == x + (y + z)
- Not true for subtraction, floating-point arithmetic, etc

std::inclusive_scan

- std::partial_sum calculates the sum of the elements so far

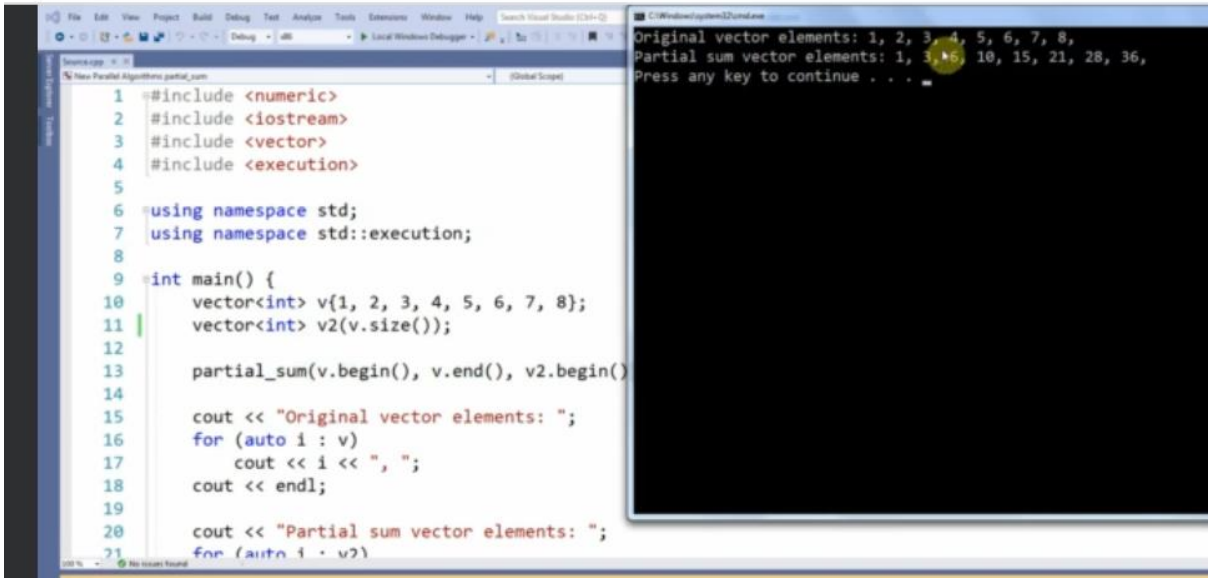
```
vector<int> v {1, 2, 3, 4};
vector<int> v2(v.size());
partial_sum(v.begin(), v.end(), v2.begin());          // v2 will contain { 1, 3, 6, 10 };
```

- As with std::accumulate, the calculation must be done in a fixed order
- std::inclusive_scan works the same way as std::partial_sum

```
inclusive_scan(v.begin(), v.end(), v2.begin());       // v2 will contain { 1, 3, 6, 10 };
```

- We can optionally add an execution policy

```
inclusive_scan(par, v.begin(), v.end(), v2.begin()); // Request parallel execution
```



std::exclusive_scan

- std::exclusive_scan calculates the sum of the elements so far, not including the current element
- It takes an extra argument, which is used as if it were the first element of the input vector

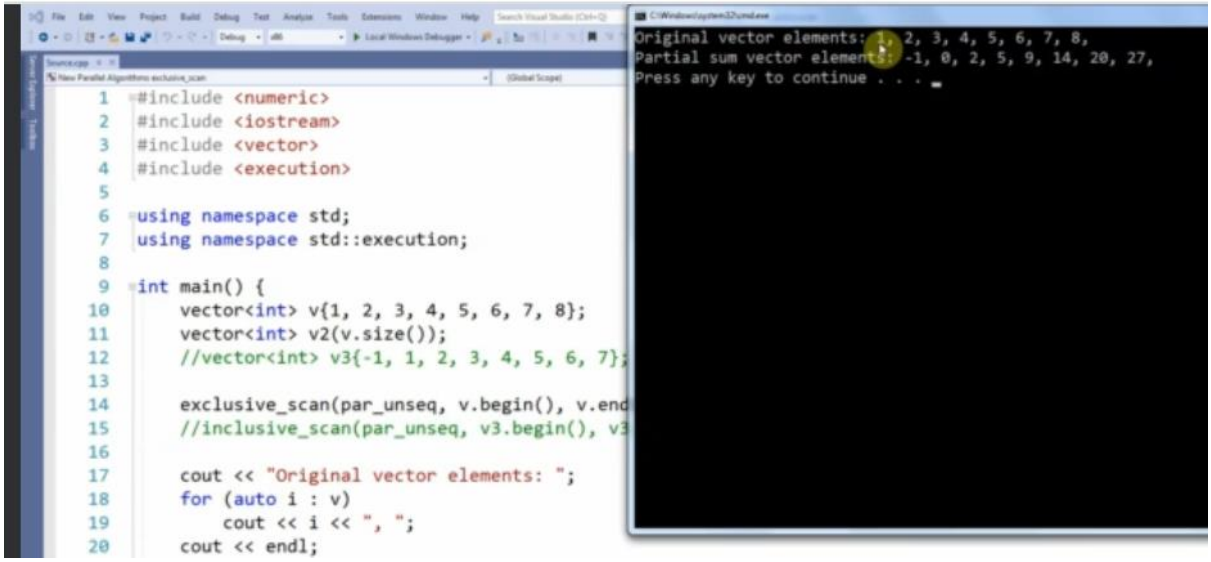
```
vector<int> v {1, 2, 3, 4};
exclusive_scan(v.begin(), v.end(), v2.begin(), -1);      // v2 will contain { -1, 0, 2, 5 };
```

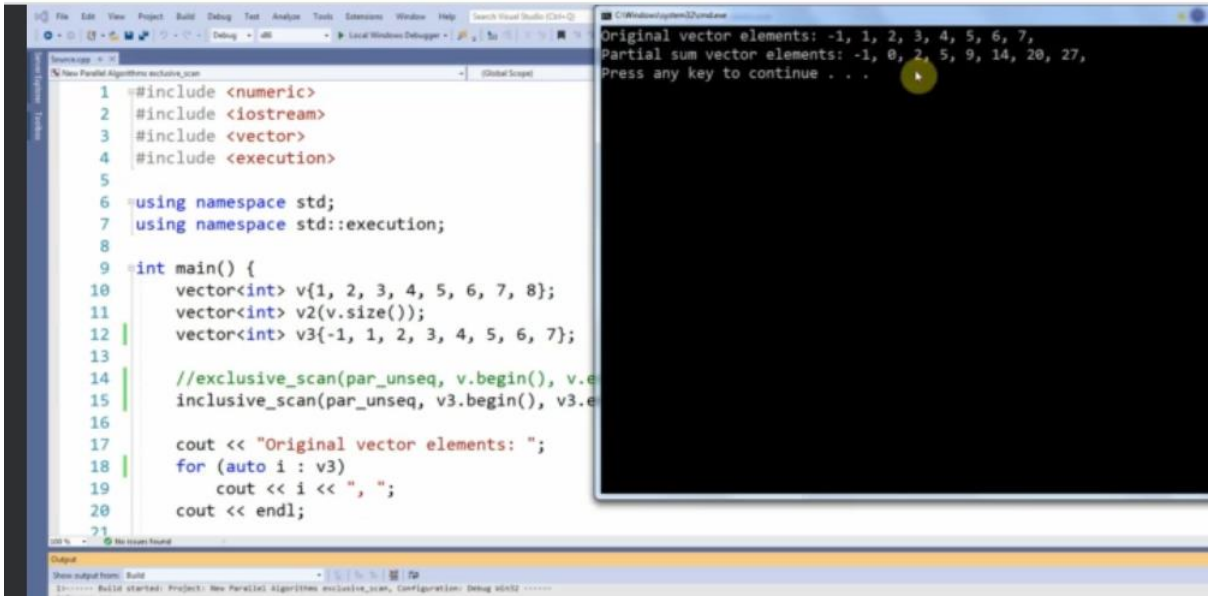
- This produces the same result as

```
vector<int> v3 {-1, 1, 2, 3};
inclusive_scan(v3.begin(), v3.end(), v2.begin();
```

- We can optionally add an execution policy

```
exclusive_scan(par, v.begin(), v.end(), v2.begin(), -1);   // Request parallel execution
```

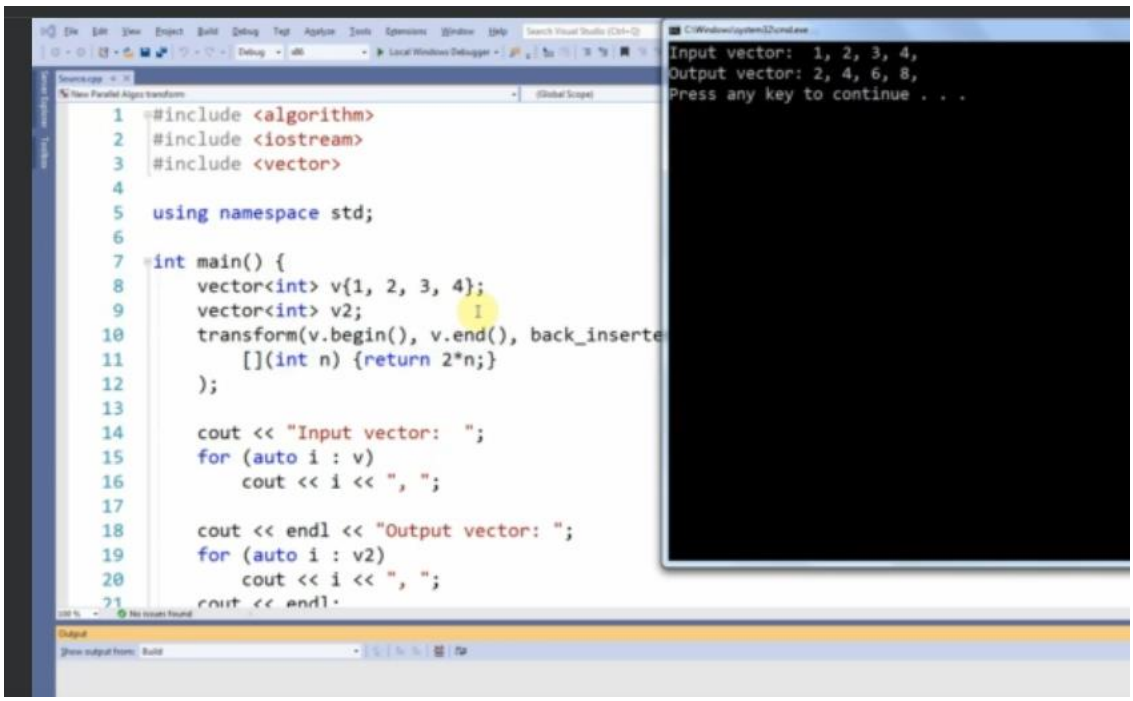




std::transform

- transform() will call a given function object on every element in an iterator range and store the result in a destination
- Equivalent to "map" in functional languages

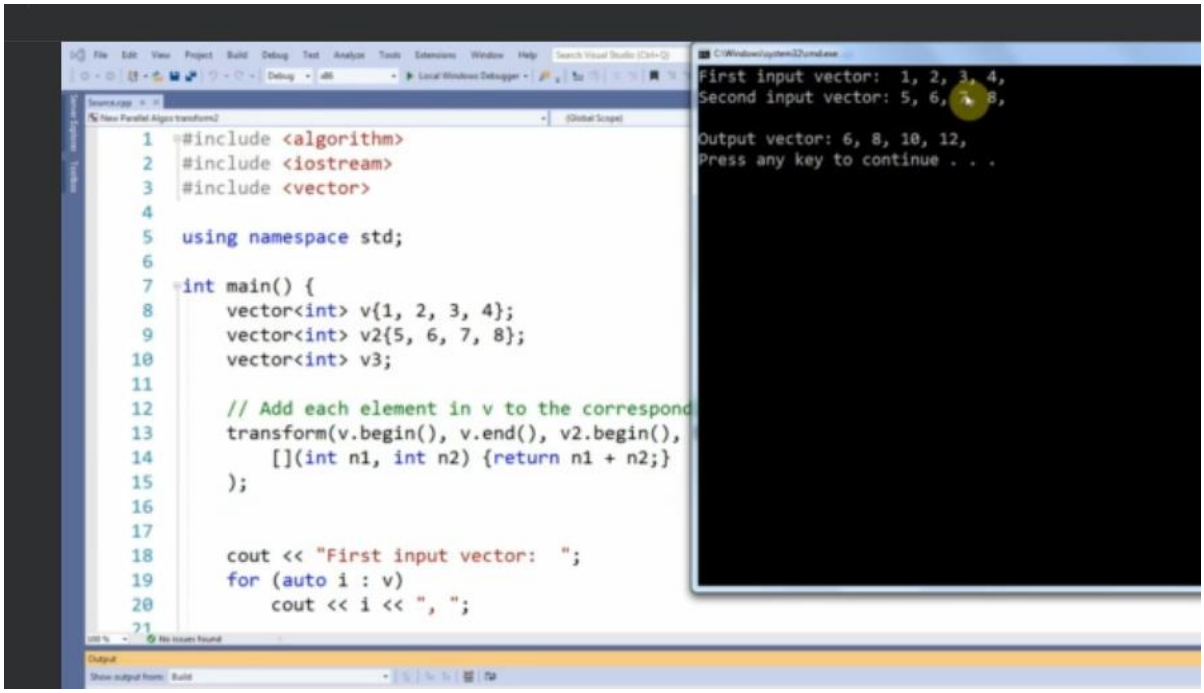
```
// Double each element in v and store the results in v2
transform(v.begin(), v.end(), back_inserter(v2),
                              [] (int n) { return 2*n; }
);
```



Binary overload of transform()

- There is an overload of transform() which takes an extra argument, representing a second source container
- The function object is a binary operator
- It is called on every corresponding pair of elements from each source container
- The result is stored in the destination

```
// Add each element in v to the corresponding element in v2 and store the result in v3
transform(v.begin(), v.end(), v2.begin(), back_inserter(v3),
            [] (int n1, int n2) { return n1 + n2; }
);
```



transform and reduce

- A very common pattern in parallel programming is transform and reduce (or "map" and reduce)
  - Divide the data into subsets
  - Start a number of threads, each of which calls transform() to perform some operation on a subset of the data
  - Call reduce() to combine the results from each thread into the final answer
- Using separate algorithms introduces overhead
  - Each transform() thread has to write an output container
  - reduce() cannot run until all the transform() threads have finished
  - reduce() has to read the output containers from the transform() threads
  - reduce() has to start up its own threads

std::transform_reduce()

- C++17 provides a combined transform_reduce() function
- It is a new version of inner_product() which supports execution policies
- It also avoids the serialization overhead of calling separate transform() and reduce() functions

std::inner_product

- inner_product multiplies the corresponding elements of two containers together and calculates their sum

```
vector<int> x{1, 2, 3, 4, 5};
vector<int> y{5, 4, 3, 2, 1};

auto result = inner_product(x.begin(), x.end(),   // Iterator range for first vector
                            y.begin(),            // Start of second vector
                            0);                   // Initial value of sum
cout << "Result is " << result << endl;           // Displays 35
```

std::transform_reduce example

- transform_reduce works the same way, except we can specify an execution policy as the first argument

```
vector<int> x{1, 2, 3, 4, 5};
vector<int> y{5, 4, 3, 2, 1};

auto result = transform_reduce(std::execution::par,    // Execution policy
                               x.begin(), x.end(),     // Iterator range for first vector
                               y.begin(),              // Start of second vector
                               0);                     // Initial value of sum
cout << "Result is " << result << endl;                // Displays 35
```

Overloads of std::transform_reduce

- There are various overloads of inner_product and transform_reduce
- The most interesting ones allow us to use our own binary functions instead of the default + and * operators
- We can replace the * operator by a "transform" function
  - This takes two arguments of the element type, and returns a value of some result type
- We can replace the + operator by a "reduce" function
  - This takes two arguments of the return type, and returns a value of the result type

Overloaded std::transform_reduce example

- The results of a scientific experiment are stored in a vector
- We have another vector which contains the theoretically predicted values
- We want to find the biggest error (the maximum difference between an expected result and the corresponding actual result)
- We can do this using an overloaded version of std::transform_reduce
  - The transform operation will find the differences between corresponding elements
  - The reduce operation will find go through the differences and find the largest

Overloaded std::transform_reduce example

- Replace the * operator by a function that finds the difference between corresponding elements
- Replace the + operator by a function that finds the largest difference

```
transform_reduce(std::execution::par,
                 begin(expected), end(expected),
                 begin(actual),
                 0.0,                                  // Initial max error
                 [](auto a, auto b){ return max(a,b); },  // Reduce operation
                 [](auto l, auto r) { return fabs(r-l); } );  // Transform operation
```

Concurrent Queue Implementation

std::queue

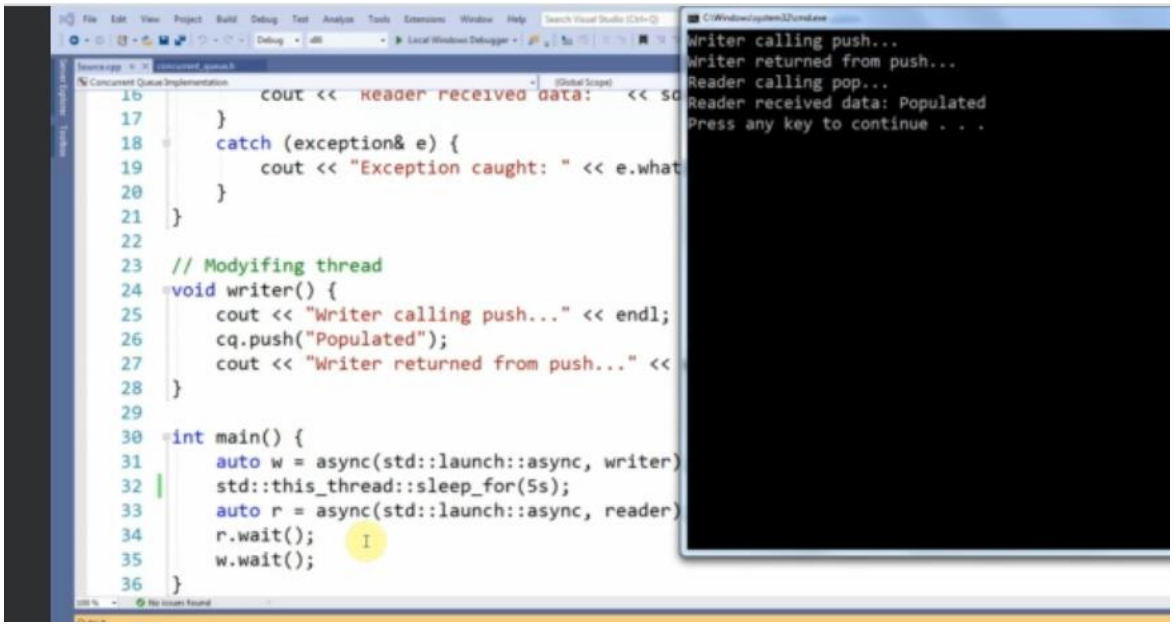
- FIFO data structure
  - Elements are stored in the order they were inserted
  - The oldest element is at the "front" and the newest element is at the "back"
  - Elements are "pushed" onto the back of the queue and "popped" off the front
- Removing an element involves two operations
  - front() returns a reference to the element at the front
  - pop() removes the element at the front, without returning anything
  - This was required in C++98 for exception safety (in case copy constructor throws)
  - If pop() is called on an empty container, the behaviour is undefined

Concurrent queue

- std::queue is not suitable for use as a concurrent queue
  - Data race if the same instance is accessed from multiple threads
  - Race condition between front() and pop()
  - Undefined behaviour if we pop() an empty queue
- The simplest way to implement a concurrent queue is to write a wrapper class
  - std::queue instance is class member
  - std::mutex is class member
  - Each member function locks the mutex, then calls the corresponding member function in the std::queue

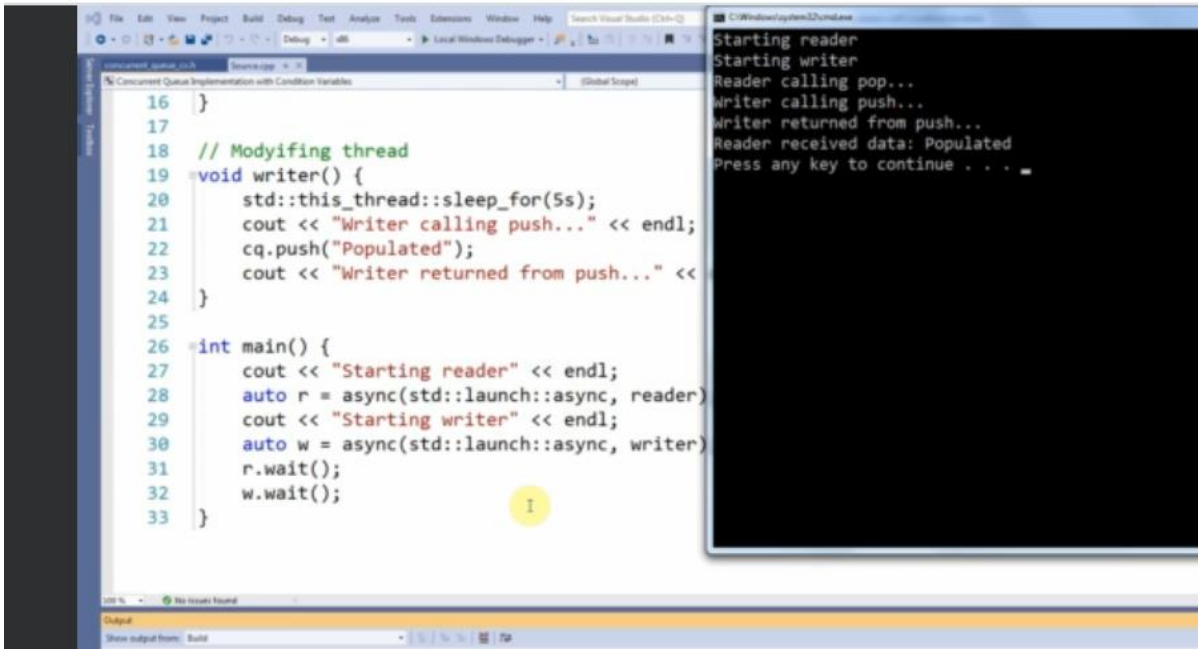
Concurrent queue member functions

- Rule of five
  - Nothing special required. The defaults are sufficient
- push()
  - Locks the mutex, calls the std::queue's push() with its argument, then unlocks the mutex
- pop()
  - Locks the mutex, calls the std::queue's front() and copies the returned value into its argument
  - Then calls the std::queue's pop() and unlocks the mutex
  - Does something sensible if called on an empty queue



Concurrent queue with condition variable

- Instead of throwing an exception in pop() when the queue is empty, we can wait for another thread to add an element
- We can do this with a condition variable
  - The thread that calls pop() calls wait() on the condition variable
  - The thread that calls push() notifies the condition variable
- To avoid spurious and lost wakeups, we add a predicate to the wait() call
  - If the queue is empty, we continue waiting
  - If it is not empty, then it is safe to continue and pop from the queue



Conclusion

- This is a simple concurrent queue
- It employs "coarse-grained" locking, in which only one thread can access the queue at any one time
- In effect, the program becomes single-threaded when accessing the queue
- Adding the condition variable improves this slightly
  - If the queue is empty and a thread is trying to pop(), other threads are allowed to run until the queue is no longer empty
- A more efficient solution can be written using lock-free programming, but is much more complex

Thread Pools

Thread creation overhead

- Creating a thread involves a lot of overhead
  - Create an execution stack for the thread
  - Call a system API
  - The operating system must create the internal data to manage the thread
  - The scheduler must execute the thread
  - A context switch occurs to run the thread
- Creating a new thread can take 10,000 times as long as calling a function directly
- Is there any way we can reduce or avoid this overhead?

Typing pool

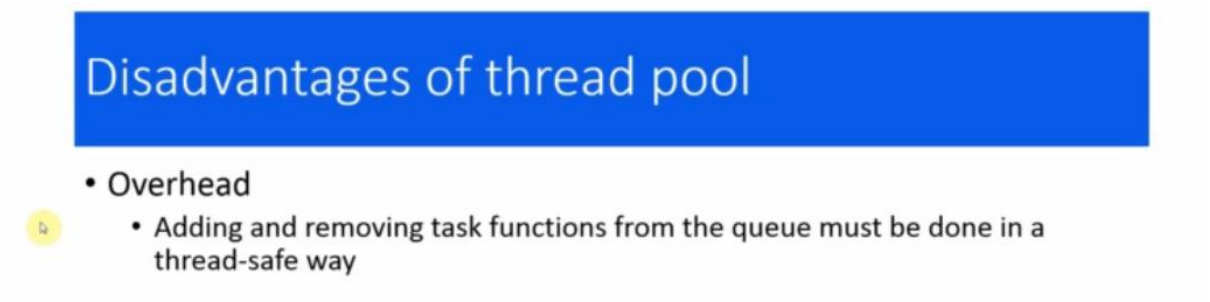
- Before computers, all business correspondence had to be typed
- Senior managers wrote enough letters and memos to justify having a dedicated secretary
- Other employees used a typing pool
  - A group of typists
  - Every time a new work item arrives, it is added to a pile of pending work
  - When a typist becomes free, they take the next item and work on it

Thread pool

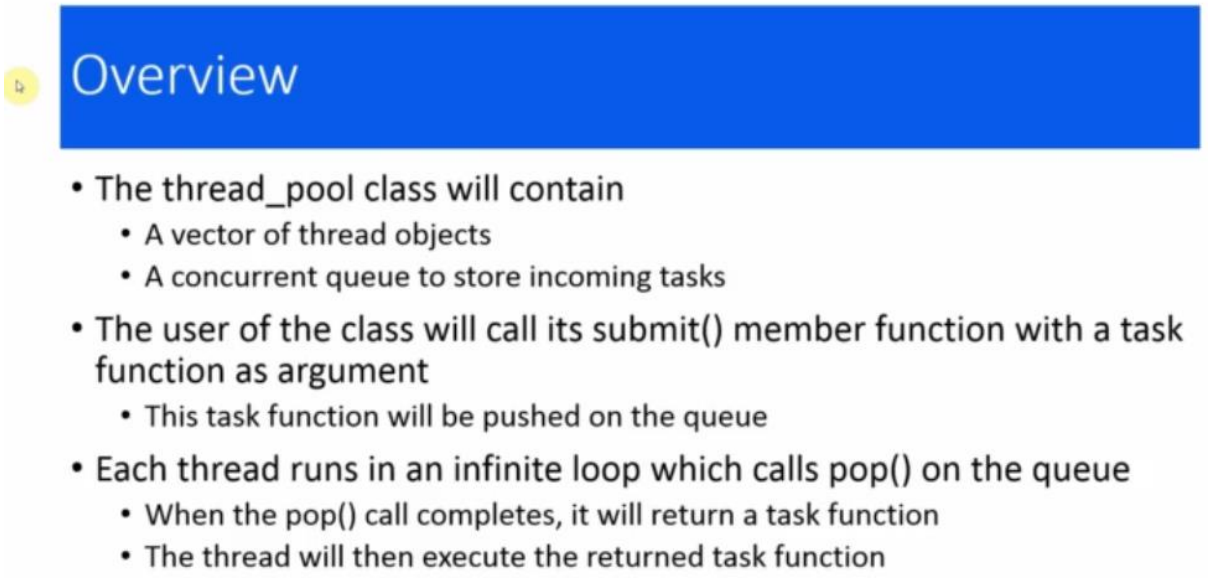
- A fixed-size container of thread objects
  - Usually equal to the number of cores on the machine
  - This can be found by calling std::thread::hardware_concurrency()
- A queue of task function objects
  - A thread object takes a task off the queue
  - It calls the task function
  - When finished, it takes the next task from the queue

Advantages of thread pool

- Easy scaling
  - The thread pool will automatically use all the available cores
- Makes efficient use of resources
  - Threads are never idle (unless there is no work for them to do)
  - As soon as a task finishes executing, the thread will take the next one from the queue
- Works best for short, simple "one-shot" tasks where
  - The overhead of creating a thread object is comparable to the task execution
  - The task does not block



Thread Pool Implementation



Multithreading libraries
Microsoft's Parallel Patterns Library is provided with Visual C++ as part of the "Concurrency Runtime", documented here: https://docs.microsoft.com/en-us/cpp/parallel/concrt/concurrency-runtime

Intel's Thread Building Blocks Library is part of their oneAPI: https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html

And finally, the "Awesome C++" github includes a number of multithreading libraries and frameworks: https://github.com/fffaraz/awesome-cpp