

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();
```

The screenshot shows a C++ development environment with a code editor and a terminal window. The code editor displays a file named 'Source.cpp' containing the provided C++ code. The terminal window, titled 'cmd.exe', shows the output of the program's execution. It prints 'Waiting for result', followed by '6 + 7 is 13', and then prompts the user to 'Press any key to continue . . .'. The terminal window has a blue header bar with the path 'C:\WINDOWS\system32\cmd.exe'.

```
Waiting for result
6 + 7 is 13
Press any key to continue . . .
```

```
13 // It takes int and int arguments
14 // It returns int
15 std::packaged_task<int(int, int)> ptask;
16     std::this_thread::sleep_for(2s);
17     return a + b;
18 };
19
20 // The promise object is an std::promise<int>
21 // Get the future associated with it
22 std::future<int> fut = ptask.get_future();
23
24 // Invoke the packaged task in this thread
25 ptask(6, 7);
26
27 std::cout << "Waiting for result\n";
28
29 // Call get() to receive the result of the packaged task
30 std::cout << "6 + 7 is " << fut.get() << '\n';
31 }
```

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();
```

The screenshot shows the Microsoft Visual Studio IDE. The Source window displays the following C++ code:

```
Source.cpp 14 // Create a thread
15 // The packaged task will be its entry point
16 std::thread thr(std::move(ptask), 6, 7);
17 thr.join();
18
19 // The promise object is an std::promise<int>
20 // Get the future associated with it
21 auto fut = ptask.get_future();
22
23 // Start a new thread
24 // The packaged task will be its entry point
25 std::thread thr(std::move(ptask), 6, 7);
26
27 std::cout << "Waiting for result\n";
28
29 std::cout << "6 + 7 is " << fut.get();
30
31 thr.join();
32 }
```

The Output window shows the command-line interface output:

```
C:\WINDOWS\system32\cmd.exe
Waiting for result
6 + 7 is 13
Press any key to continue . . .
```

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);
}
```



A screenshot of a Windows command prompt window titled 'cmd' at the top. The path 'C:\Windows\system32\cmd.exe' is visible. The window contains the text 'Hello, Async!' followed by 'Press any key to continue . . .'. To the left of the command prompt is a code editor window showing a file named 'Source.cpp'. The code in the editor matches the text above, with line numbers 1 through 16 on the left.

```
1 // Hello world using std::async()
2 #include <future>
3 #include <iostream>
4
5 // Task function
6 void hello()
7 {
8     std::cout << "Hello, Async!\n";
9 }
10 int main()
11 {
12     // Call std::async() to perform the t
13     std::async(hello);
14 }
15
16
```

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();
```

A screenshot of Microsoft Visual Studio. On the left, the code editor shows `Source.cpp` with the following content:

```
14
15 int main()
16 {
17     std::cout << "Calling fibonacci(44)\n";
18
19     // Call async() and store the returned future
20     auto result = std::async(fibonacci, 44);
21
22     // Do some other work
23     bool finished = false;
24
25     using namespace std::literals;
26     while (result.wait_for(1s) != std::future_status::ready)
27     {
28         std::cout << "Waiting for the result..." << std::endl;
29     }
30
31     // Call get() when we are ready
32     std::cout << result.get() << std::endl;
33 }
```

The terminal window on the right shows the output of the program:

```
C:\WINDOWS\system32\cmd.exe
Calling fibonacci(44)
Waiting for the result...
1134903170
Press any key to continue . . .
```

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
    ...
}
```

The screenshot shows a Windows command prompt window titled 'cmd.exe' with the path 'C:\WINDOWS\system32\cmd.exe'. The output text is:

```
Future calling get()...
Exception caught: Oops
Press any key to continue . . .
```

Below the command prompt is a code editor window for 'Source.cpp'. The code is as follows:

```
22     return x;
23 }
24
25 int main()
26 {
27     // Call async() and store the returned Future
28     auto result = std::async(produce);
29
30     // Get the result - may throw an exception
31     std::cout << "Future calling get()..." << endl;
32     try {
33         int x = result.get();
34         std::cout << "Future returns from get()..." << endl;
35         std::cout << "The answer is " << x << endl;
36     }
37     catch (std::exception& e) {
38         std::cout << "Exception caught: " << e.what() << endl;
39     }
40 }
```

The code editor has syntax highlighting for C++ and shows line numbers 22 through 40. The output window shows the text 'Future calling get()...', followed by 'Exception caught: Oops', and 'Press any key to continue . . .'. The status bar at the bottom of the code editor says 'No issues found'.

```
Source.cpp 16 // Some code which may throw an exception
17 if (0) {
18     throw std::out_of_range("Oops");
19 }
20
21 std::cout << "Produce returning " << x;
22 return x;
23 }
24
25 int main()
26 {
27     // Call async() and store the returned Future
28     auto result = std::async(produce);
29
30     // Get the result - may throw an exception
31     std::cout << "Future calling get()..." << std::endl;
32     try {
33         int x = result.get();                                // Get the result
34         std::cout << "Future returns from calling get()\n";
35     }
36 }
```

C:\WINDOWS\system32\cmd.exe
Future calling get()...
Produce returning 42
Future returns from calling get()
The answer is 42
Press any key to continue . . .

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

```
In main thread with ID: 5900
Calling async with option "async"
Executing task() in thread with ID: 19356
Calling get()
Returning from task()
Task result: 42
Calling async with option "deferred"
Calling get()
Executing task() in thread with ID: 5900
Returning from task()
Task result: 42
Calling async with option "default"
Executing task() in thread with ID: 19356
Calling get()
Returning from task()
Task result: 42
Press any key to continue . . .
```

Source.cpp

```
25     else if (option == "deferred")
26         result = std::async(std::launch::deferred,
27     else
28         result = std::async(task);
29
30     std::cout << "Calling async with option "
31     std::this_thread::sleep_for(2s);
32     std::cout << "Calling get()\n";
33     std::cout << "Task result: " << result;
34 }
35
36 int main()
37 {
38     std::cout << "In main thread with ID:
39
40     func("async");
41     func("deferred");
42     func("default");
43 }
```

No errors found

Output

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

The screenshot shows the Microsoft Visual Studio IDE. In the top-left pane, there is a code editor with the following C++ code:

```
Source.cpp: 10  using namespace std::literals;
11
12 void task()
13 {
14     std::this_thread::sleep_for(5s);
15     std::cout << "Task result: " << 42 <<
16 }
17
18 void func()
19 {
20     std::cout << "Calling async\n";
21     auto result = std::async(std::launch::async,
22                             task);
23
24 int main()
25 {
26     func();
27     std::cout << "Task started\n";
28 }
```

In the bottom-right pane, the output window displays the following text:

```
C:\WINDOWS\system32\cmd.exe
Calling async
Task result: 42
Task started
Press any key to continue . . .
```

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

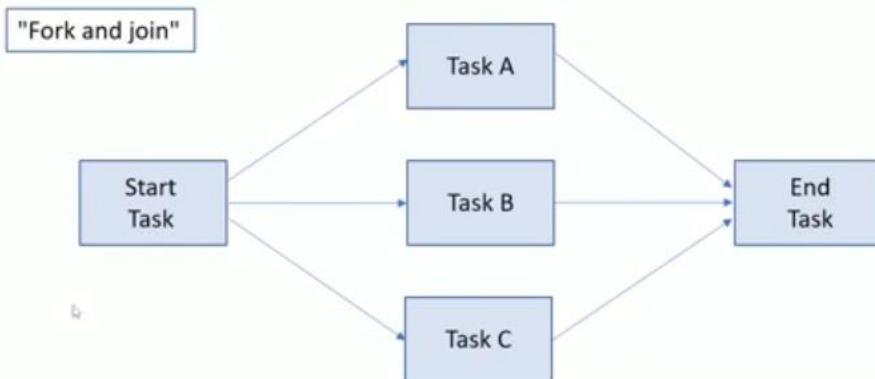
Implicit Parallelism

- 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

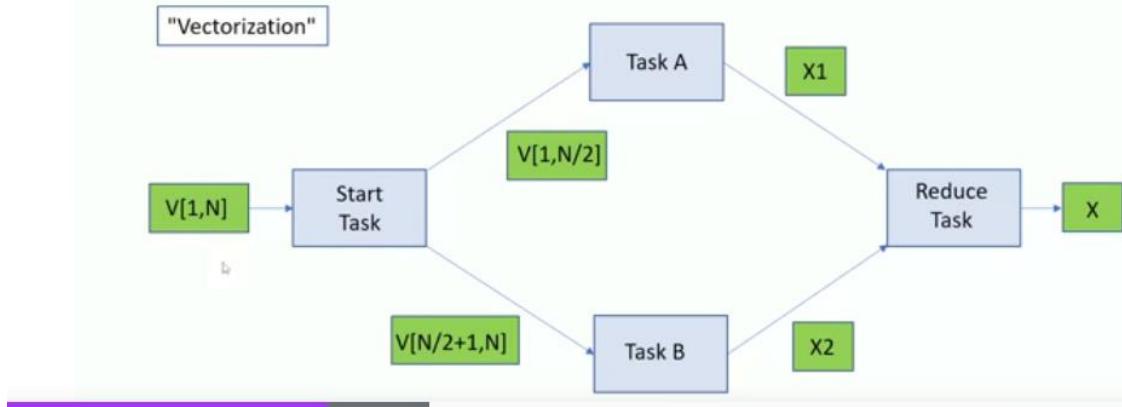
Task Parallelism



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

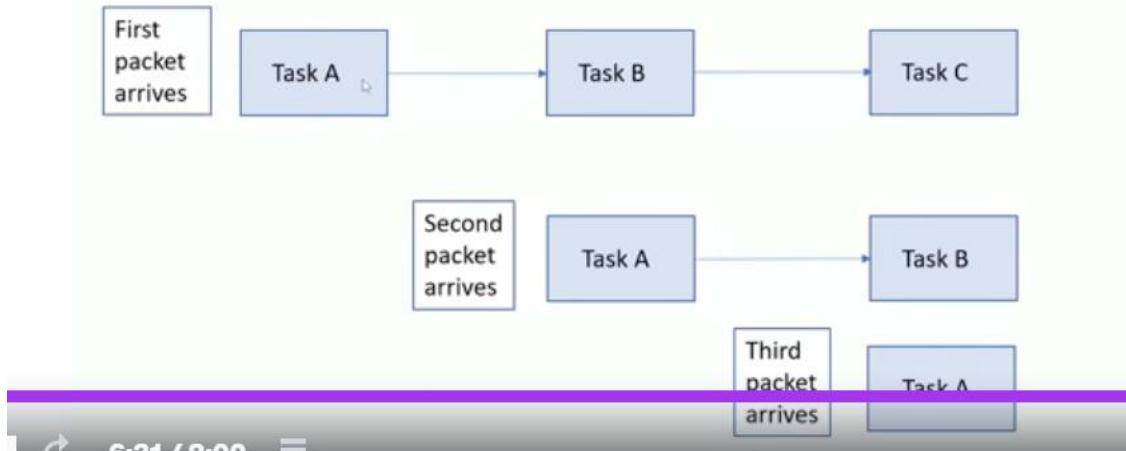
Locality of Data References

- 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
 - C starts processing the next item

Pipelining

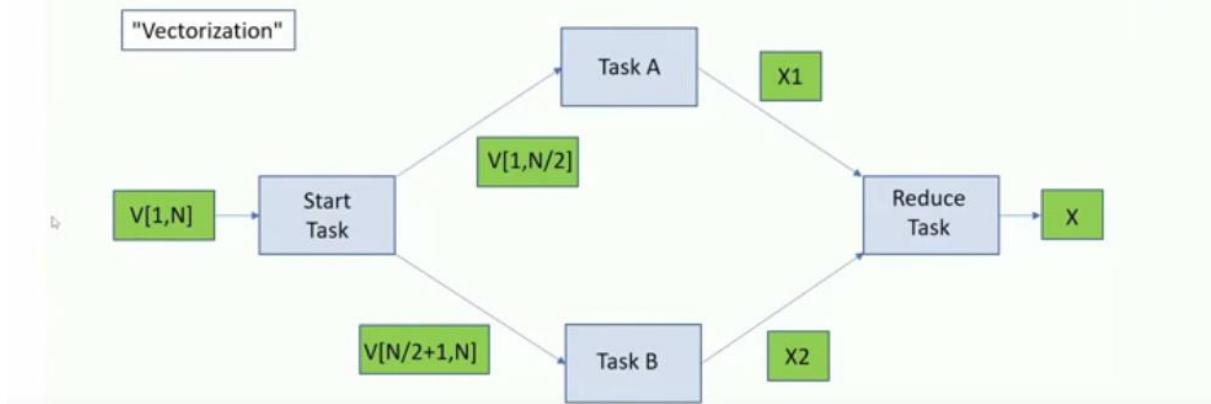


Parallelism

- 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



A screenshot of a Microsoft Visual Studio IDE. The code editor shows a file named 'Source.cpp' with the following content:

```
20 double add_parallel(std::vector<double>& v)
21 {
22     // Find the first element of the vector
23     auto vec0 = &vec[0];
24
25     // Find the number of elements
26     auto vszie = vec.size();
27
28     // Start the threads
29     // Pass the subset's range as arguments
30     auto fut1 = std::async(std::launch::async, add_parallel, vec0, vszie / 4);
31     auto fut2 = std::async(std::launch::async, add_parallel, vec0 + vszie / 4, vszie / 4);
32     auto fut3 = std::async(std::launch::async, add_parallel, vec0 + vszie / 2, vszie / 4);
33     auto fut4 = std::async(std::launch::async, add_parallel, vec0 + vszie / 2, vszie / 4);
34
35     // Reduce step
36     // Combine the results for each subset
37     return fut1.get() + fut2.get() + fut3.get() + fut4.get();
38 }
```

The output window shows the results of the parallel computation:

```
C:\Windows\system32\cmd.exe
Sum of first 16 integers: 136
Sum of 10,000 random numbers: 497515
Press any key to continue . . .
```

The screenshot shows a C++ development environment with a code editor and a terminal window. The code editor displays a file named 'Source.cpp' containing C++ code that uses the `std::parallel_for` function to calculate the sum of elements in a vector. The terminal window shows the execution of the program, displaying the sum of the first 16 integers (136) and the sum of 10,000 random numbers (497515), followed by a prompt to press any key to continue.

```
36 // Find the first element of the vector
37 auto vec0 = &vec[0];
38
39 // Find the number of elements
40 auto vsize = vec.size();
41
42 // Start the threads
43 // Pass the subset's range as argument
44 std::thread thr1(std::move(ptask1), vec0, 0, 4);
45 std::thread thr2(std::move(ptask2), vec0, 4, 8);
46 std::thread thr3(std::move(ptask3), vec0, 8, 12);
47 std::thread thr4(std::move(ptask4), vec0, 12, vsize);
48
49 thr1.join(); thr2.join(); thr3.join();
50
51 // Reduce step
52 // Combine the results for each subset
53 return fut1.get() + fut2.get() + fut3.get() + fut4.get();
```

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
    // ...
}
```

```
File Edit View Project Build Debug Tools Analyze Extensions Window Help Search (Ctrl-Q)
Source.cpp -> [Search]
Standard Algorithms 0 Find [Global Scope]
7     std::string str("Hello world");
8     std::cout << "String to search: " <<
9
10    // Search string for first occurrence
11    std::cout << "Searching for first occu
12    auto res = std::find(str.cbegin(), str
13
14    // Did we find it?
15    if (res != str.cend()) {
16        // Access the result
17        std::cout << "Found a matching ele
18
19        std::cout << "At this point in the
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";
100% No Issues found
Output
```

String to search: Hello 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 . . .

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';
});

```

The screenshot shows a debugger interface with two panes. The left pane displays the source code in a file named 'Source.cpp'. The right pane shows the output of the program's execution.

Source code (Source.cpp):

```
9     std::string str("Hello world");
10    std::cout << "String to search: " << str;
11
12    // Search string for first occurrence
13    std::cout << "Searching for first occurrence . . .
14
15    // Use a predicate function which ignores case
16    auto res = std::find_if(str.cbegin(),
17                           [] (const char c) {
18                               return ::toupper(c) == 'O';
19                           });
20
21    // Did we find it?
22    if (res != str.cend()) {
23        // Access the result
24        std::cout << "Found a matching element at index: ";
25        std::cout << res - str.cbegin();
26
27        std::cout << "At this point in the string: ";
28        for (auto it = res; it != str.cend(); ++it)
29            std::cout << *it;
30    }
31
32    std::cout << std::endl;
33}
```

Output window:

```
String to search: Hello world
Searching for first occurrence of 'o'
Found a matching element at index: 4
At this point in the string: o world
Press any key to continue . . .
```

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; });
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

```
1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 1772, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 1801, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812, 1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 1848, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, Press any key to continue... . . .
```

```
1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 1772, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 1801, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812, 1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 1848, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, Press any key to continue... . . .
```

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

This screenshot shows the Visual Studio IDE with two windows. The left window displays the source code for 'ExecutionPolicy.cpp' containing a parallel for_each loop that increments a shared variable 'count' without protection. The right window shows the output console with a long list of random integers from 1 to 2000, indicating that the variable 'count' has been corrupted by multiple parallel threads.

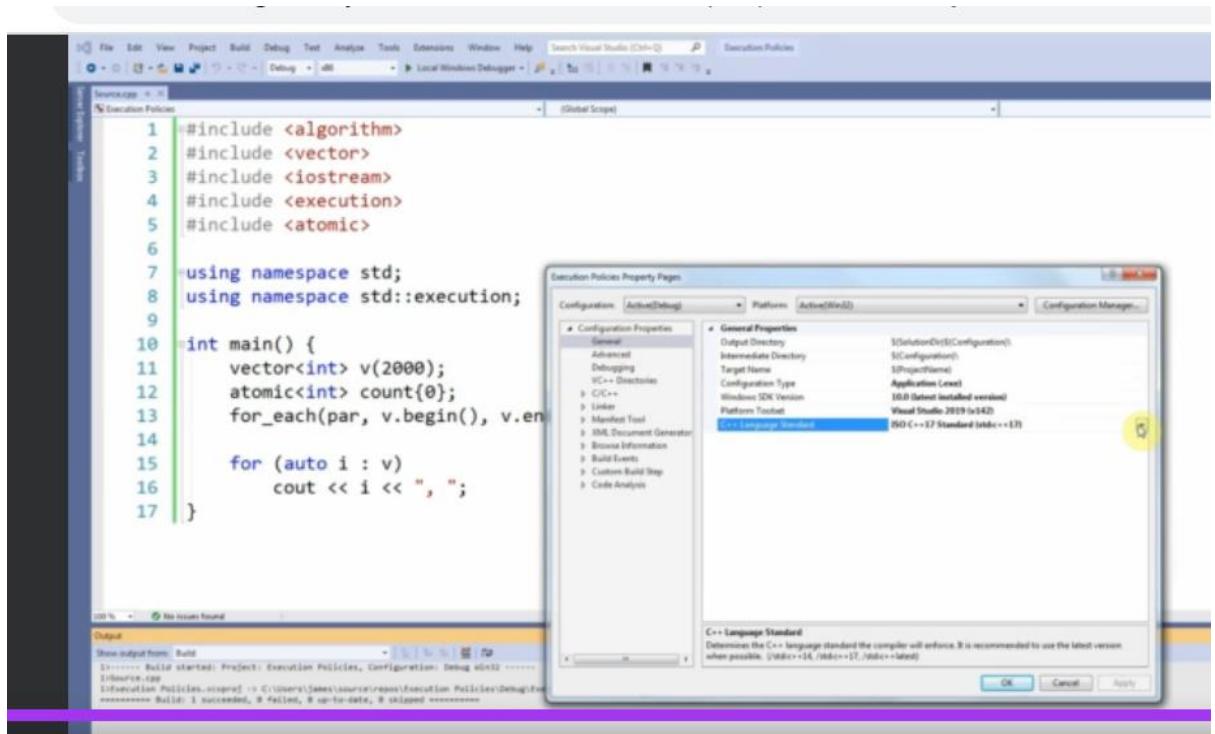
```
1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <execution>
5
6 using namespace std;
7 using namespace std::execution;
8
9 int main() {
10     vector<int> v(2000);
11     int count{0};
12     for_each(par, v.begin(), v.end(), [&] (int& x) {
13         cout << i << ", ";
14     });
15 }
```

```
1530, 1531, 1533, 1534, 1535, 1536, 1537, 1532, 1533, 1534, 1535, 1536, 1537, 1-
538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1540, 1541, 1542, 1543, 1545, 154-
6, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1556, 1550, 1551, 1552, 1553, 1554,
1555, 1556, 1559, 1560, 1561, 1562, 1563, 1566, 1567, 1568, 1569, 1570, 1571, 1572,
1561, 1562, 1563, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 1569, 1570, 1571, 157-
2, 1573, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1581, 1582, 1583,
1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1-
597, 1598, 1599, 1600, 1601, 1601, 1602, 1603, 1604, 1605, 1606, 1607, 1608, 160-
9, 1610, 1611, 1612, 1613, 1614, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620,
1621, 1622, 1623, 1625, 1626, 1627, 1624, 1625, 1626, 1626, 1627, 1628, 1629, 1630, 1-
631, 1632, 1633, 1634, 1635, 1636, 1637, 1634, 1635, 1636, 1637, 1638, 1639, 164-
0, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1644, 1645, 1646, 1647, 1648, 1649,
1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1653, 1654, 1655, 1656, 1657, 1-
658, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1662, 1663, 1664, 1665, 166-
6, 1667, 1668, 1669, 1670, 1671, 1672, 1673, 1674, 1675, 1671, 1672, 1673, 1674,
1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1685, 1686, 1688, 1681, 1682, 1-
683, 1684, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1689, 1690, 169-
1, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1697, 1698,
1699, 1700, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1712, 1707, 1-
708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 171-
5, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 1726, 1727, 1728, 1729,
1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1-
736, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 174-
3, 1744, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750,
1751, 1752. Press any key to continue . . .
```

This screenshot shows the Visual Studio IDE with two windows. The left window displays the source code for 'ExecutionPolicy.cpp' with a fix: it uses std::atomic instead of std::int for the shared variable 'count'. The right window shows the output console with a long list of random integers from 1 to 2000, indicating that the variable 'count' is now correctly updated by the parallel threads due to the atomicity guarantee.

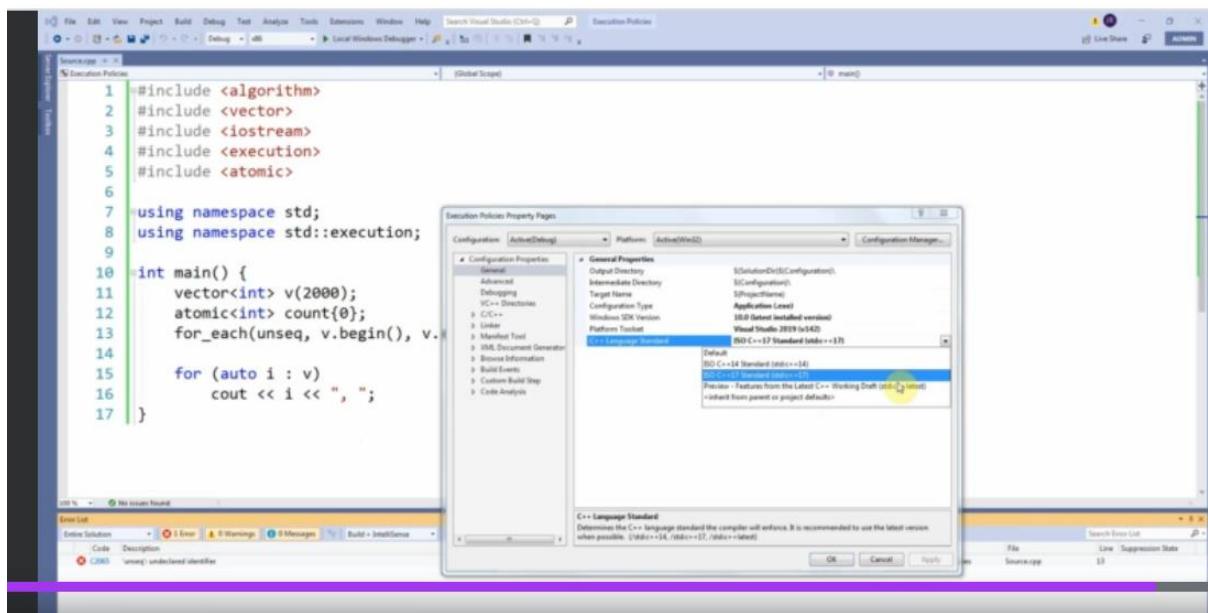
```
1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <execution>
5 #include <atomic>
6
7 using namespace std;
8 using namespace std::execution;
9
10 int main() {
11     vector<int> v(2000);
12     atomic<int> count{0};
13     for_each(par, v.begin(), v.end(), [&] (int& x) {
14         cout << i << ", ";
15     });
16 }
```

```
1679, 1680, 1682, 1684, 1687, 1689, 1692, 1681, 1683, 1685, 1686, 1688, 1690, 1-
691, 1693, 1694, 1696, 1698, 1700, 1701, 1703, 1695, 1697, 1699, 1702, 1704, 170-
5, 1707, 1706, 1708, 1709, 1710, 1711, 1712, 1714, 1713, 1715, 1716, 1718, 1720,
1723, 1725, 1717, 1719, 1721, 1722, 1724, 1726, 1727, 1729, 1731, 1733, 1735, 1-
736, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 174-
3, 1744, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750,
1751, 1752. Press any key to continue . . .
```



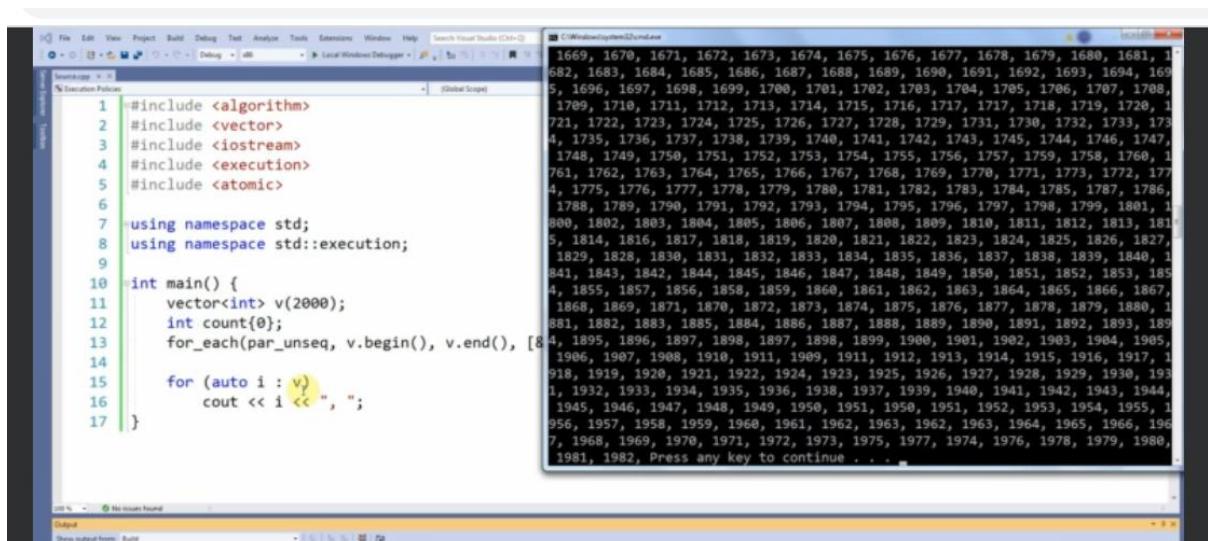
Unsequenced execution

- 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



```

1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <execution>
5 #include <atomic>
6
7 using namespace std;
8 using namespace std::execution;
9
10 int main() {
11     vector<int> v(2000);
12     atomic<int> count{0};
13     for_each(par_unseq, v.begin(), v.end(), [&count]);
14
15     for (auto i : v)
16         cout << i << ", ";
17 }

```

1679, 1680, 1681, 1682, 1684, 1686, 1688, 1683, 1685, 1687, 1689, 1690, 1691, 1
 692, 1693, 1694, 1695, 1696, 1698, 1700, 1702, 1697, 1699, 1701, 1703, 1704, 170
 5, 1707, 1706, 1708, 1709, 1710, 1712, 1714, 1716, 1711, 1713, 1715, 1717, 1718,
 1719, 1721, 1720, 1722, 1723, 1724, 1725, 1727, 1729, 1726, 1728, 1730, 1731, 1
 732, 1733, 1735, 1734, 1736, 1737, 1738, 1739, 1741, 1743, 1740, 1742, 1744, 174
 5, 1746, 1747, 1749, 1748, 1750, 1751, 1752, 1753, 1755, 1757, 1754, 1756, 1758,
 1759, 1760, 1761, 1763, 1762, 1764, 1765, 1766, 1768, 1770, 1772, 1767, 1769, 1
 771, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1782, 1784, 1785, 1781, 178
 3, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1796, 1798, 1800, 1795,
 1797, 1799, 1801, 1803, 1805, 1808, 1802, 1804, 1807, 1811, 1813, 1817, 1821, 1
 806, 1809, 1810, 1812, 1814, 1816, 1819, 1815, 1818, 1820, 1822, 1823, 1824, 182
 5, 1825, 1827, 1828, 1829, 1831, 1834, 1836, 1830, 1833, 1837, 1840, 1842, 184
 5, 1849, 1852, 1853, 1858, 1838, 1839, 1841, 1844, 1847, 1843, 1846, 1848, 1850, 1851, 1
 853, 1855, 1852, 1854, 1856, 1858, 1868, 1863, 1866, 1857, 1859, 1862, 1865, 186
 8, 1871, 1874, 1861, 1864, 1867, 1869, 1870, 1872, 1875, 1873, 1876, 1877, 1878,
 1879, 1880, 1882, 1881, 1884, 1886, 1887, 1890, 1893, 1895, 1883, 1885, 1888, 1
 891, 1896, 1898, 1901, 1889, 1892, 1894, 1897, 1899, 1900, 1903, 1902, 1904, 190
 5, 1906, 1907, 1909, 1912, 1911, 1914, 1917, 1928, 1923, 1926, 1929, 1908, 1910,
 1913, 1915, 1916, 1918, 1921, 1919, 1922, 1924, 1925, 1928, 1931, 1933, 1927, 1
 936, 1932, 1934, 1935, 1936, 1939, 1937, 1941, 1943, 1946, 1949, 1952, 1955, 193
 8, 1940, 1942, 1944, 1947, 1950, 1953, 1945, 1948, 1951, 1954, 1956, 1957, 1959,
 1958, 1960, 1962, 1963, 1966, 1969, 1971, 1961, 1964, 1968, 1972, 1974, 1977, 1
 980, 1965, 1967, 1970, 1973, 1975, 1976, 1979, 1978, 1981, 1982, 1983, 1985, 198
 7, 1990, 1984, 1986, 1988, 1991, 1993, 1995, 1997, 1989, 1992, 1994, 1996, 1998,
 1999, 2000, Press any key to continue . . .

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

The screenshot shows a Visual Studio IDE window with two panes. The left pane displays a C++ source code file named 'Source.cpp' containing the following code:

```
#include <algorithm>
#include <vector>
#include <exception>
#include <iostream>
#include <execution>

using namespace std;
using namespace std::execution;

int main() {
    vector<int> v{3,1,4,1,5,9};
    try {
        sort(v.begin(), v.end(), [](int a, int b) {
            throw std::out_of_range("Oops");
        });
    }
    catch (exception& e) {
        cout << "Caught exception: " << e.what();
    }
}
```

The right pane shows a terminal window with the output:

```
C:\Windows\system32\cmd.exe
Caught exception: Oops
Press any key to continue . . .
```

The screenshot shows a Visual Studio interface. On the left is the code editor with a file named 'AlgorithmsAndExecutionPolicies.cpp'. The code demonstrates the use of execution policies with std::sort. A yellow callout points to the line 'sort(seq, v.begin(), v.end(), [](int a, int b){ return a < b;});'. On the right is a terminal window titled 'C:\Windows\system32\cmd.exe' with the text 'Press any key to continue . . .'.

```
#include <algorithm>
#include <vector>
#include <exception>
#include <iostream>
#include <execution>

using namespace std;
using namespace std::execution;

int main() {
    vector<int> v{3,1,4,1,5,9};
    try {
        sort(seq, v.begin(), v.end(), [](int a, int b){ return a < b;});
        throw std::out_of_range("Oops");
    };
    catch (exception& e) {
        cout << "Caught exception: " << e.what();
    }
}
```

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

The screenshot shows the Visual Studio IDE. On the left is the code editor with `Source.cpp` containing C++ code. On the right is the Output window showing the execution results.

```
1 #include <numeric>
2 #include <iostream>
3 #include <vector>
4 #include <execution>
5
6 using namespace std;
7 using namespace std::execution;
8
9 int main() {
10     vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
11     auto sum = accumulate(v.begin(), v.end(),
12
13         cout << "Vector elements: ";
14         for (auto i : v)
15             cout << i << ", ";
16         cout << "\nSum of elements is " << sum <<
17 }
```

Output window content:

```
Vector elements: 1, 2, 3, 4, 5, 6, 7, 8
Sum of elements is 36
Press any key to continue . . .
```

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);
```

```
#include <numeric>
#include <iostream>
#include <vector>
#include <execution>

using namespace std;
using namespace std::execution;

int main() {
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
    auto sum = reduce(v.begin(), v.end(), 0);

    cout << "Vector elements: ";
    for (auto i : v)
        cout << i << ", ";
    cout << "\nSum of elements is " << sum <<
```

Vector elements: 1, 2, 3, 4, 5, 6, 7, 8
Sum of elements is 36
Press any key to continue . . .

std::reduce() and vectorization

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

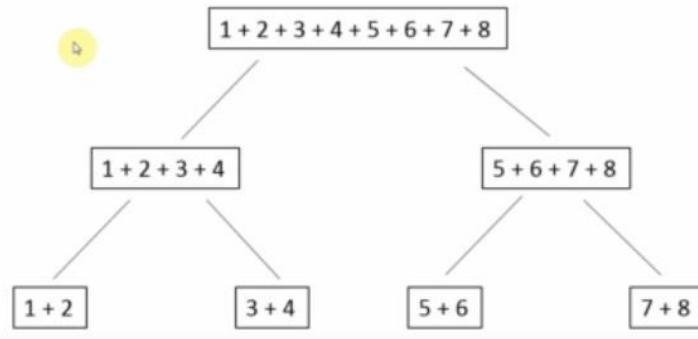
```
vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
```

```
// 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
```

The screenshot shows a Visual Studio IDE window with two panes. The left pane displays a code editor for a file named 'source.cpp' containing C++ code. The right pane shows the output window with the results of the program's execution.

Code Editor (source.cpp):

```
#include <numeric>  
#include <iostream>  
#include <vector>  
#include <execution>  
  
using namespace std;  
using namespace std::execution;  
  
int main() {  
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};  
    vector<int> v2(v.size());  
  
    partial_sum(v.begin(), v.end(), v2.begin());  
  
    cout << "Original vector elements: ";  
    for (auto i : v)  
        cout << i << ", ";  
    cout << endl;  
  
    cout << "Partial sum vector elements: ";  
    for (auto i : v2)
```

Output Window:

```
C:\Windows\system32\cmd.exe  
Original vector elements: 1, 2, 3, 4, 5, 6, 7, 8,  
Partial sum vector elements: 1, 3, 6, 10, 15, 21, 28, 36,  
Press any key to continue . . .
```

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
```

The screenshot shows a Visual Studio IDE window with two panes. The left pane displays the code for `exclusive_scan.cpp`, and the right pane shows the console output.

Code (Left Pane):

```
#include <numeric>  
#include <iostream>  
#include <vector>  
#include <execution>  
  
using namespace std;  
using namespace std::execution;  
  
int main() {  
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};  
    vector<int> v2(v.size());  
    //vector<int> v3{-1, 1, 2, 3, 4, 5, 6, 7};  
  
    exclusive_scan(par_unseq, v.begin(), v.end(), v2.begin());  
    //inclusive_scan(par_unseq, v3.begin(), v3.end(), v2.begin());  
  
    cout << "Original vector elements: ";  
    for (auto i : v)  
        cout << i << ", ";  
    cout << endl;
```

Output (Right Pane):

```
Original vector elements: 1, 2, 3, 4, 5, 6, 7, 8,  
Partial sum vector elements: -1, 0, 2, 5, 9, 14, 20, 27,  
Press any key to continue . . .
```

```

1 #include <numeric>
2 #include <iostream>
3 #include <vector>
4 #include <execution>
5
6 using namespace std;
7 using namespace std::execution;
8
9 int main() {
10     vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
11     vector<int> v2(v.size());
12     vector<int> v3{-1, 1, 2, 3, 4, 5, 6, 7};
13
14 //inclusive_scan(par_unseq, v.begin(), v.end(), v2,
15 inclusive_scan(par_unseq, v3.begin(), v3.end(), v2);
16
17 cout << "Original vector elements: ";
18 for (auto i : v)
19     cout << i << ", ";
20 cout << endl;
21

```

Original vector elements: -1, 1, 2, 3, 4, 5, 6, 7,
Partial sum vector elements: -1, 0, 2, 5, 9, 14, 20, 27,
Press any key to continue . . .

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; });

```

```

1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6
7 int main() {
8     vector<int> v{1, 2, 3, 4};
9     vector<int> v2;
10    transform(v.begin(), v.end(), back_inserter(v2),
11              [] (int n) { return 2*n; });
12
13    cout << "Input vector: ";
14    for (auto i : v)
15        cout << i << ", ";
16
17    cout << endl << "Output vector: ";
18    for (auto i : v2)
19        cout << i << ", ";
20    cout << endl;
21

```

Input vector: 1, 2, 3, 4,
Output vector: 2, 4, 6, 8,
Press any key to continue . . .

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; })
);
```

The screenshot shows a Visual Studio IDE window. On the left, the code editor displays a file named "Source.cpp" with the following content:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v{1, 2, 3, 4};
    vector<int> v2{5, 6, 7, 8};
    vector<int> v3;

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

    cout << "First input vector: ";
    for (auto i : v)
        cout << i << ", ";
}
```

On the right, the output window shows the execution results:

```
First input vector: 1, 2, 3, 4,
Second input vector: 5, 6, 7, 8,
Output vector: 6, 8, 10, 12,
Press any key to continue . . .
```

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(),  
                           y.begin(),  
                           0);  
cout << "Result is " << result << endl;
```

// Iterator range for first vector
// Start of second vector
// Initial value of sum
// Displays 35

std::transform_reduce example

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

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

auto result = transform_reduce(std::execution::par,
                             x.begin(), x.end(),
                             y.begin(),
                             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 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

- 4
 - 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()**

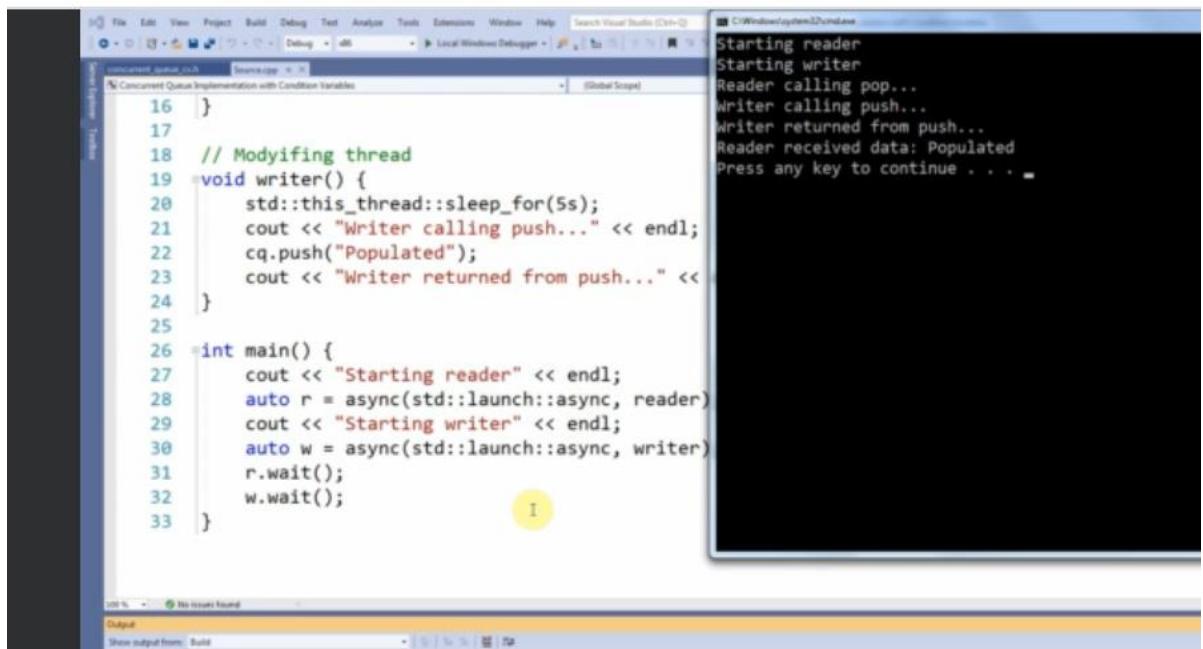
- Locks the mutex, calls the std::queue's front() and copies the returned value into its argument
 - Then calls the std::queue's pop() and unlocks the mutex
 - Does something sensible if called on an empty queue

The screenshot shows a Visual Studio IDE window with two main panes. The left pane displays a C++ code editor for a file named 'Concurrent Queue Implementation'. The code contains a writer thread function that pushes the string "Populated" into a concurrent queue and a reader thread function that retrieves and prints the data from the queue. The right pane shows the output window with the following text:

```
C:\Windows\system32\cmd.exe
Writer calling push...
Writer returned from push...
Reader calling pop...
Reader received data: Populated
Press any key to continue . . .
```

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



A screenshot of Visual Studio showing a C++ file named `concurrent_queue.cpp`. The code implements a concurrent queue using condition variables. It includes a `writer()` function that sleeps for 5 seconds, then pushes "Populated" into the queue and returns. It also includes a `main()` function that starts both a reader and a writer thread. The output window shows the execution of the program, with the reader starting first, followed by the writer, and then the reader popping the item from the queue.

```
16 }
17
18 // Modifying thread
19 void writer() {
20     std::this_thread::sleep_for(5s);
21     cout << "Writer calling push..." << endl;
22     cq.push("Populated");
23     cout << "Writer returned from push..." <<
24 }
25
26 int main() {
27     cout << "Starting reader" << endl;
28     auto r = std::async(std::launch::async, reader)
29     cout << "Starting writer" << endl;
30     auto w = std::async(std::launch::async, writer)
31     r.wait();
32     w.wait();
33 }
```

```
C:\Windows\system32\cmd.exe
Starting reader
Starting writer
Reader calling pop...
Writer calling push...
Writer returned from push...
Reader received data: Populated
Press any key to continue . . .
```

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

Disadvantages of thread pool

- Overhead
 - Adding and removing task functions from the queue must be done in a thread-safe way

Thread Pool Implementation

Overview

- The `thread_pool` class will contain
 - A vector of thread objects
 - A concurrent queue to store incoming tasks
- The user of the class will call its `submit()` member function with a task function as argument
 - This task function will be pushed on the queue
- Each thread runs in an infinite loop which calls `pop()` on the queue
 - When the `pop()` call completes, it will return a task function
 - The thread will then execute the returned task function

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>