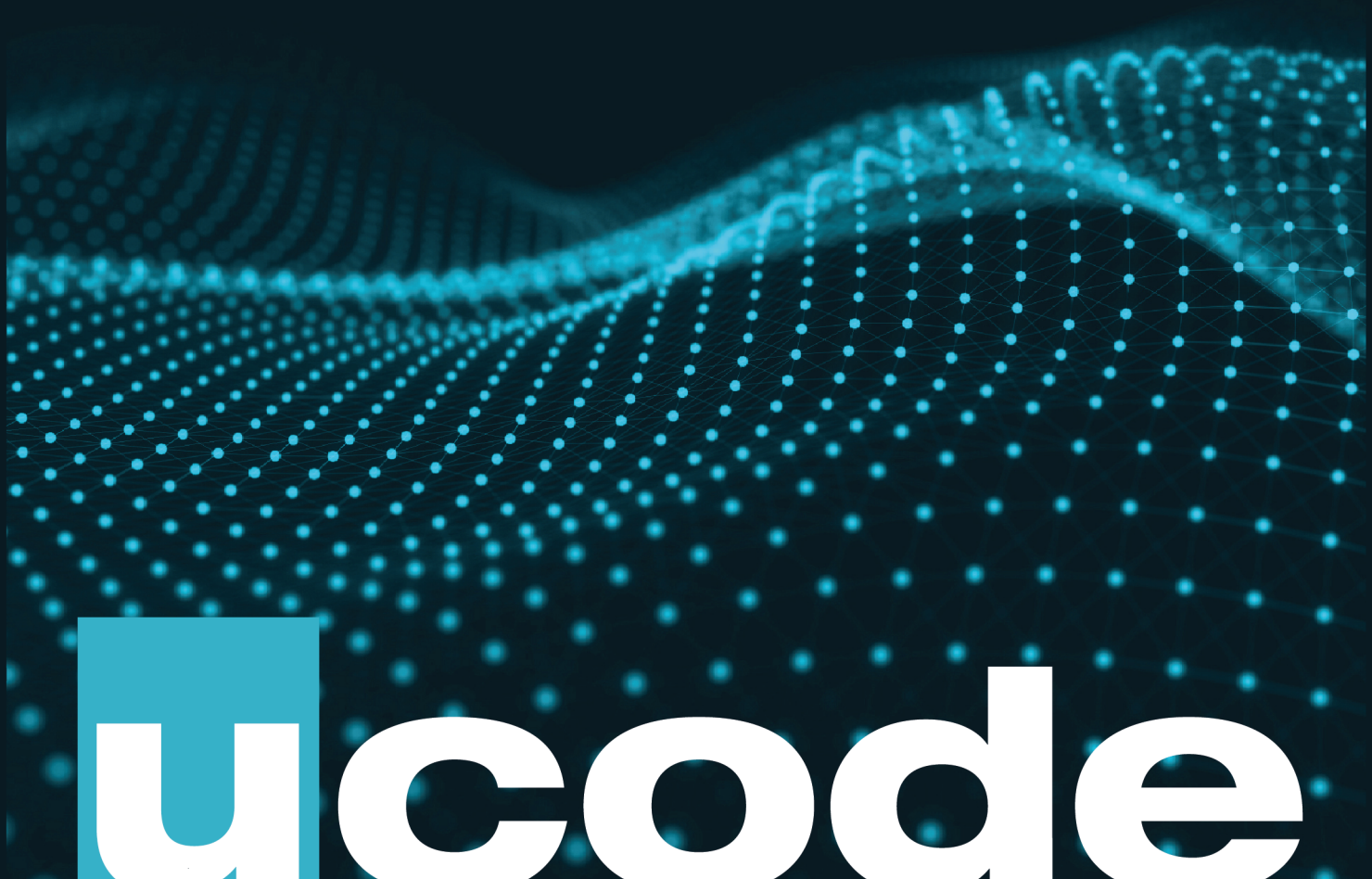
Sprint 09

Half Marathon C++

August 25, 2020



ucode

Contents

ucode

Engage

DESCRIPTION

Welcome to Sprint 09!
You're finally here. You've done such a great job!

During your previous Sprints you've learned a lot.

Today, the world is evolving faster than ever. Everyone looks for new ways to achieve more with the same resources and maximize the effectiveness of programs. Nobody wants to wait for long execution of a program. And after this Sprint, you will learn how to develop high-performance apps and games. Fortunately, there are many ways to significantly speed up computing. One of them is concurrent programming.

It allows you to build advanced systems where every process can execute without waiting for all others to complete. In short, you can multiple tasks at the same time. At first glance, it might seem complicated, but systematic approach and perseverance will lead you to a whole new world in programming!

Good luck!

BIG IDEA

Concurrent computing.

ESSENTIAL QUESTION

How to use concurrency in C++ effectively?

CHALLENGE

Learn multithreading.

Investigate

GUIDING QUESTIONS

We invite you to find answers to the following questions. By researching and answering them, you will gain the knowledge necessary to complete the challenge. To find answers, ask the students around you and search the internet. We encourage you to ask as many questions as possible. Note down your findings and discuss them with your peers.

- How do you understand the concept of concurrent computing?
- What is the difference between concurrent and parallel computing?
- What are the use cases of `std::thread`?
- What operations can be done with threads?
- What is a `lambda expression` and how to use it?
- What are the use cases of `std::lock_guard`?
- What are the use cases of `std::mutex`?
- What are the use cases of `std::condition_variable`?
- What are the use cases of `std::atomic`?
- What are the use cases of `std::async`?
- How does `auto` determine the type of expression to be assigned?

GUIDING ACTIVITIES

Complete the following activities. Don't forget that you have a limited time to overcome the challenge. Use it wisely. Distribute tasks correctly.

- Read the tasks carefully and try to find as much information as possible about them.
- Consider the algorithms found in the tasks.
- Allocate your resources and time.
- Arrange to brainstorm tasks with other students.
- Clone your git repository that is issued on the challenge page in the LMS.
- Try to implement your thoughts in code.
- Push solutions to the repository.

ANALYSIS

Analyze your findings. What conclusions have you made after completing guiding questions and activities? In addition to your thoughts and conclusions, here are some more analysis results.

- Be attentive to all statements of the story.
- Analyze all information you have collected during the preparation stages. Try to define the order of your actions.
- Perform only those tasks that are given in this document.

- Submit your files using the layout described in the story. Only useful files allowed, garbage shall not pass!
- Tasks in `shell` must be executed with `zsh`.
- Compile files with commands `cmake . -Bbuild && cmake --build ./build` that will call `CMake` and build an app.
- Pay attention to what is allowed in a certain task. Use of forbidden stuff is considered a cheat and your tasks will be failed.
- Complete tasks according to the rules specified in the Google C++ Style Guide.
  But there are several exceptions for the guide listed below:
  - you can use `#pragma once` directive instead of `#ifndef ... #define`
  - variables can be written in `mixed case`
  - class data members must begin with `m_` prefix (m for member)
  - indent 4 spaces at a time
  - each line of text in your code must be at most 120 characters long
  - ignore the sections Inputs and Outputs and Legal Notice and Author Line in the style guide
- The solution will be checked and graded by students like you. Peer-to-Peer learning.
- You must use this project structure for each task:

```
>tree project_dir --dirsfirst --charset=ascii
project_dir
|-- app
|   |-- src
|   |   |-- CMakeLists.txt
|   |   `-- ...
|   |-- resources
|   |   `-- ...
|   |-- CMakeLists.txt
|   `-- main.cpp
|-- lib1
|   |-- CMakeLists.txt
|   `-- ...
...
|-- libN
|   |-- CMakeLists.txt
|   `-- ...
`-- CMakeLists.txt
```

- If you have any questions or don't understand something, ask other students or just Google it.
- In the name of Talos, use your brain!

Act: Task 00

NAME

Simple Worker V1

DIRECTORY

`t00/`

BINARY

`simpleWorkerV1`

DESCRIPTION

Create a program with threads. Implement the `Worker` class according to the conditions listed below:

- is not copyable or movable
- ontains member functions:
  - `joinThread` waits for the thread to finish in `m_thread` and then deletes it
  - `startWorker`:
    * accepts both regular expressions (lambdas) and member functions as a first parameter
    * calls `joinThread`, if a thread already exists
    * launches a new thread in `m_thread` with the given parameters
- has a destructor that calls `joinThread` and then deletes `m_thread`

Test with the `main.cpp` from the SYNOPSIS.

SYNOPSIS

```
/* Worker.h */
class Worker {
 public:
    Worker() = default;
    ~Worker();

    template <typename Function, class... Args>
    void startWorker(Function&& func, Args&&... args);

    void joinThread();

 private:
    std::thread* m_thread{nullptr};
};

/* main.cpp */
```

```cpp
class TestClass {
 public:
    TestClass() = default;
    ~TestClass() = default;

    void testMember1() { std::cout << "testMember1" << std::endl; }

    void testMember2(int i) { std::cout << "testMember2 " << i << std::endl; }
};

int main() {
    const auto strLambda = [](std::string&& str) { std::cout << str << std::endl; };

    Worker worker;
    worker.startWorker(strLambda, "Main thread 1");

    worker.joinThread();

    worker.startWorker(strLambda, "Main thread 2");
    worker.startWorker(strLambda, "Main thread 3");

    worker.joinThread();

    const auto testLambda = [] () {
        for (auto i = 0; i < 100; i += 2) {
            --i;
        }
        std::cout << "testLambda" << std::endl;
    };

    worker.startWorker(testLambda);

    TestClass obj;
    worker.startWorker(&TestClass::testMember1, &obj);
    worker.startWorker(&TestClass::testMember2, &obj, 10);

    testLambda();

    return 0;
}
```

CONSOLE OUTPUT

```
>./simpleWorkerV1 | cat -e
Main thread 1$
Main thread 2$
Main thread 3$
testLambda$
testMember1$
testLambda$
testMember2 10$
```

```
>
```

SEE ALSO

std::thread
Lambda expressions

Act: Task 01

NAME

Simple Worker V2

DIRECTORY

`t01/`

BINARY

`simpleWorkerV2`

DESCRIPTION

Create a program that outputs the result of add and subtract functions using threads. The program accepts three command-line arguments as integers and contains:

- improved `Worker` class that supports multiple threads. Requirements:
  - is not copyable or movable
  - contains two member functions:
    * `startNewThread` that creates a new thread and does in-place insertion of it to the end of the `vector`
    * `joinAllThreads` that expects all threads to complete in `m_workerThreads`
  - has a destructor that uses `joinAllThreads`
- `MultithreadedClass` class that controls thread access to the variable `m_int` and contains the following member functions:
  - `getInt` returns the value of `m_int`
  - `add` manages a mutex manually and increases `m_int` by one on each iteration of the `for loop`. `for loop` iterates `addValue` times to simulate some work
  - `subtract` uses the advantages of `std::lock_guard` and decreases `m_int` by one on each iteration of the `for loop`. `for loop` iterates `subtractValue` times to simulate some work
- `[addValue]` and `[subtractValue]` command-line arguments are in the range of -2000 and 2000.
- `[count]` command-line argument is in the range of 5 and 10
- `main.cpp` snippet from the SYNOPSIS. Complete `main.cpp` with missing source code.

Error handling. The program prints to the standard error:

- in case of an incorrect number of arguments - `usage: ./simpleWorkerV2 [addValue] [subtractValue] [count]`
- if the values of the command-line arguments are incorrect - `Incorrect values`

SYNOPSIS

```cpp
/* Worker.h */

class Worker {
 public:
    Worker() = default;
    ~Worker();

    template <typename Function, class... Args>
    void startNewThread(Function&& func, Args&&... args);

    void joinAllThreads();

 private:
    std::vector<std::thread> m_workerThreads;
};

/* MultithreadedClass.h */

class MultithreadedClass {
 public:
    MultithreadedClass() = default;
    ~MultithreadedClass() = default;

    int getInt() const;

    void add(int addValue);

    void subtract(int subtractValue);

 private:
    int m_int{0};
    std::mutex m_mutex;
};

/* main.cpp */

int main(int argc, char** argv) {

//...
    MultithreadedClass obj;

    Worker worker;

    for (auto i = 0; i < count; ++i) {
        worker.startNewThread(&MultithreadedClass::add, &obj, addValue);
    }

    for (auto i = 0; i < count; ++i) {
        worker.startNewThread(&MultithreadedClass::subtract, &obj, subtractValue);
```

```
    }

    worker.joinAllThreads();

    std::cout << obj.getInt() << std::endl;
//...

}
```

CONSOLE OUTPUT

```
>./simpleWorkerV2 | cat -e
usage: ./simpleWorkerV2 [addValue] [subtractValue] [count]
>./simpleWorkerV2 2000 2000 5 | cat -e
0$
>./simpleWorkerV2 20 10 10 | cat -e
100$
>./simpleWorkerV2 20 50 6 | cat -e
-180$
>./simpleWorkerV2 2000 30000 5 | cat -e
Incorrect values
>./simpleWorkerV2 20 3 4 | cat -e
Incorrect values
>
```

SEE ALSO

std::mutex
std::lock_guard

Act: Task 02

NAME

Multithreaded File Handler

DIRECTORY

`t02/`

BINARY

`multithreadedFileHandler`

DESCRIPTION

Create a program that reads two files using threads and prints them to the standard output.

- Implement a `MultithreadedFileHandler` class that processes and loads a file in different threads
- `processFile` member function waits for a file to be loaded into `loadFile` member function and displays the contents of the file, which is stored in the `m_file` variable.
  Use `std::condition_variable` for this
- For launching these functions in different threads use `Worker` class from previous task
- Before starting threads for the second file, the program blocks the execution of the threads for the first file for 1 second. The message `-----1 second sleep-----` is printed while waiting
- `joinAllThreads()` member function is called at the end of the program execution
- The output of the program is similar to the output listed in the CONSOLE OUTPUT

Error handling. The program prints to the standard error:

- in case of an incorrect number of arguments -
  `usage: ./multithreadedFileHandler [file1] [file2]`
- if `file1` or `file2` is invalid - `error`

SYNOPSIS

```
/* MultithreadedFileHandler.h */

class MultithreadedFileHandler {
 public:
    MultithreadedFileHandler() = default;
    ~MultithreadedFileHandler() = default;

    void processFile();
    void loadFile(const std::string& fileName);
```

```
 private:
    std::string m_file;
    std::mutex m_mutex;
    std::condition_variable m_condVar;
    bool m_fileLoaded{false};
};
```

CONSOLE OUTPUT

```
>./multithreadedFileHandler | cat -e
usage: ./multithreadedFileHandler [file1] [file2]
>./multithreadedFileHandler file1.txt file2.txt | cat -e
*file1.txt content*$
-----1 second sleep-----$
*file2.txt content*$
>./multithreadedFileHandler qwe qwe | cat -e
error
>
```

SEE ALSO

std::condition_variable

Act: Task 03

NAME

Class With Atomic

DIRECTORY

`t03/`

BINARY

`classWithAtomic`

DESCRIPTION

Create a program that accepts three command-line arguments as integers and implements the `ClassWithAtomic` class that handles its operations and contents asynchronously.

A class has member functions:

- that don't use a mutex:
  - `addToInt` increases `m_int` by one on each iteration of the `for loop`. `for loop` iterates `addValue` times to simulate some work
  - `subtractFromInt` decreases `m_int` by one on each iteration of the `for loop`. `for loop` iterates `subtractValue` times to simulate some work
- that use a mutex:
  - `pushToVector` adds element `value` to the end of the `m_vector`
  - `eraseFromVector` deletes all elements from `m_vector` that are equal to the `value`
- `getInt` returns `m_int`
- `getVector` returns `m_vector`

The program:

- starts threads using input data from command-line arguments:
  - `[addValue]` is used in `addToInt` member function
  - `[subtractValue]` is used in `subtractFromInt` member function
  - `[size]` indicates the number of items that need to be pushed with the `pushToVector` member function
- uses `main.cpp` snippet from the SYNOPSIS. Complete `main.cpp` with missing source code
- the output is similar to the output listed in the CONSOLE OUTPUT
- prints `usage: classWithAtomic [add] [subtract] [size]` to the standard error, if the number of command-line arguments is not in the valid range

SYNOPSIS

```cpp
/* ClassWithAtomic.h */

class ClassWithAtomic {
 public:
    ClassWithAtomic() = default;
    ~ClassWithAtomic() = default;

    void addToInt(int addValue);
    void subtractFromInt(int subtractValue);

    void pushToVector(int value);
    void eraseFromVector(int value);

    int getInt() const;
    std::vector<int> getVector() const;

 private:
    std::mutex m_vecMutex;
    std::atomic<int> m_int;
    std::vector<int> m_vector;
};

/* main.cpp */

int main(int argc, char** argv) {

//...

    Worker worker;

    ClassWithAtomic obj;

    worker.startNewThread(&ClassWithAtomic::addToInt, &obj, addValue);
    worker.startNewThread(&ClassWithAtomic::subtractFromInt, &obj, subtractValue);

    for (auto i = 0; i < pushSize; ++i) {
        worker.startNewThread(&ClassWithAtomic::pushToVector, &obj, i);
    }

    for (auto i = 1; i <= pushSize; ++i) {
        if (i % 2 == 0) {
            worker.startNewThread(&ClassWithAtomic::eraseFromVector, &obj, i);
        }
    }

    worker.joinAllThreads();

    std::cout << "Result: " << obj.getInt() << std::endl;
```

```
    auto vec = obj.getVector();

    std::cout << "Size of vector: " << vec.size() << std::endl;

    for (int i = 0; i < vec.size(); ++i) {
        std::cout << vec[i];
        if (i != vec.size() - 1) {
            std::cout << " ";
        } else {
            std::cout << std::endl;
        }
    }

//...

}
```

CONSOLE OUTPUT

```
>./classWithAtomic | cat -e
usage: classWithAtomic [add] [subtract] [size]
>./classWithAtomic 5 10 20  | cat -e
Result: -5$
Size of vector: 11$
0 3 5 7 1 9 11 13 15 17 19$
>./classWithAtomic 25 13 25 | cat -e
Result: 12$
Size of vector: 13$
0 1 3 5 7 9 11 13 15 17 19 21 23$
>
```

SEE ALSO

std::atomic

Act: Task 04

NAME

Simple Worker V3

DIRECTORY

`t04/`

BINARY

`simpleWorkerV3`

DESCRIPTION

Create a program that creates asynchronous threads and prints their results to the standard output.
Implement a new `Worker` class:

- is not copyable or movable
- `startAsync` member function that returns the resulting object

Test `Worker` class with `main.cpp` from the SYNOPSIS but do not stop here. Provide wider range of test cases to prove that you have fully completed this task.

SYNOPSIS

```
/* Worker.h */

class Worker {
 public:
    Worker() = default;
    ~Worker() = default;

    template <typename Function, class... Args>
    auto startAsync(Function&& func, Args&&... args);
};

/* main.cpp */

int main() {
    Worker worker;

    const auto lambda1 = [](int k) {
        k += 69;
        return k;
    };

    const auto lambda2 = [](std::string s) {
        s.append(" appended str");
        return s;
    };
```

```
    auto fut1 = worker.startAsync(lambda1, 1);
    auto fut2 = worker.startAsync(lambda2, "Str");

    auto res1 = fut1.get();
    auto res2 = fut2.get();

    std::cout << res1 << std::endl;
    std::cout << res2 << std::endl;
    return 0;
}
```

CONSOLE OUTPUT

```
>./simpleWorkerV3 | cat -e
70$
Str appended str$
>
```

SEE ALSO

std::async

Act: Task 05

NAME

Logger

DIRECTORY

`t05/`

BINARY

`logger`

DESCRIPTION

Create a program that writes logs into a file using threads.
Implement your logger.
You must:

- create `Worker` abstract class that has:
  - `m_isRunning` member variable that indicates whether a thread is running in `m_worker`
  - member functions:
    * `start` launches a new thread in `m_worker` using `run` member function and sets `m_isRunning` to `true`, if thread isn't running in `m_worker`
    * `stop` sets `m_isRunning` to `false`
    * `join` waits for the end of the thread in `m_worker`
    * `isRunning` returns `m_isRunning`
- create `Log` class that has:
  - private constructor that initializes `m_logLevel` with `level`
  - destructor that uses `log` member function from `LoggerWorker` class to create log with `m_logLevel` and text from `m_inputStream`
  - `createLog` member function that returns a new instance of `Log` class with the given `LogLevel`
  - overloaded operator `<<` that appends `value` to the end of a `m_inputStream` member variable, and then appends a single space character to the end of the `value`
  - defines:
    * `logDebug` that creates a log with `Debug` log level
    * `logWarning` that creates a log with `Warning` log level
    * `logInfo` that creates a log with `Info` log level

- Create `LoggerWorker` class that must be a `Singleton` class and has a:
  - constructor that creates a log file with the following name format: `<day>_<month>_<year>T<hours>_<minutes>_<seconds>.txt` and launches the thread in `m_worker`
  - destructor that:
    * sets `m_isRunning` to `false`
    * notifies one waiting thread and waits till it's finished
    * closes file in `m_logFileStream`
  - `getLogger` member function that:
    * creates `LoggerWorker` object in `m_logger`, if `m_logger` is empty
    * returns a pointer to `m_logger`
  - `log` member function that:
    * inserts object with `level` and `logMessage` data to the end of the `m_logQueue`
    * notifies one waiting thread
  - `run` member function that:
    * loops infinitely until the thread is started in `m_worker` or the queue is not empty
    * waits for work using `m_condVar`
    * gets the first element from `m_logQueue`
    * inserts the output to the `m_logFileStream` in the next format followed by a newline:
      `[<day>:<month>:<year>T<hours>:<minutes>:<seconds>] <log_level> <element_message>`
    * deletes the first element from `m_logQueue`
- use `main.cpp` from the SYNOPSIS where you need to replace `//replace me` with one line of code that causes the calling thread to sleep for 1 second but do not stop here. Provide wider range of test cases to prove that you have fully completed this task

SYNOPSIS

```
/* Worker.h */

class Worker {
 public:
    Worker() = default;
    virtual ~Worker() = default;

    Worker(const Worker&) = delete;
    Worker& operator=(const Worker&) = delete;
```

```
    void start();
    void stop();
    void join();
    bool isRunning() const;

 protected:
    virtual void run() = 0;

 private:
    std::atomic_bool m_isRunning{false};
    std::thread m_worker;
};

/* LoggerWorker.h */

class LoggerWorker final : public Worker {
 public:
    ~LoggerWorker() override;

    static LoggerWorker& getLogger();

    void log(Log::LogLevel level, const std::string& logMessage);

 private:
    struct LogMessage {
        std::string message;
        Log::LogLevel logLevel;
    };

    LoggerWorker();

    void run() override;

    std::mutex m_runMutex;
    std::condition_variable m_condVar;
    std::ofstream m_logFileStream;
    std::queue<LogMessage> m_logQueue;

    inline static std::shared_ptr<LoggerWorker> m_logger{nullptr};
};

/* Log.h */

class Log final {
 public:
    enum class LogLevel { Debug, Warning, Info };

    ~Log();

    static Log createLog(LogLevel level);
```

```
    template <class T>
    Log& operator<<(T const& value);

 private:
    Log(LogLevel level);

    LogLevel m_logLevel;
    std::stringstream m_inputStream;
};

template <class T>
Log& Log::operator<<(T const& value) {
    m_inputStream << value << " ";
    return *this;
}


/* main.cpp */

int main() {
    std::cout << "I am sworn to carry your logs." << std::endl;

    logInfo << "Main thread info" << 1;

    logDebug << "Main thread debug" << 2.0;

    const auto l1 = [] {
        logWarning << "Thread 1 warning" << true;

        std::string str = "!!!";
        // replace me

        logDebug << "Thread 1 debug" << str << false;
        logInfo << "Thread 1 info" << 3.14f;
    };

    const auto l2 = [] {
        logInfo << "Thread 2 info" << -5;
        logDebug << "Thread 2 debug" << 'k';

        std::string str = "???";
        // replace me

        logWarning << "Thread 2 warning" << str;
    };

    std::thread t1(l1);
    std::thread t2(l2);

    t1.join();
```

```
    t2.join();

    return 0;
}
```

CONSOLE OUTPUT

```
>./logger | cat -e
I am sworn to carry your logs.$
>cat -e 06_10_2019T14_27_29.txt
[06:10:2019T14:27:29] [Info] Main thread info 1 $
[06:10:2019T14:27:29] [Debug] Main thread debug 2 $
[06:10:2019T14:27:29] [Warning] Thread 1 warning 1 $
[06:10:2019T14:27:29] [Info] Thread 2 info -5 $
[06:10:2019T14:27:29] [Debug] Thread 2 debug k $
[06:10:2019T14:27:30] [Warning] Thread 2 warning ??? $
[06:10:2019T14:27:30] [Debug] Thread 1 debug !!! 0 $
[06:10:2019T14:27:30] [Info] Thread 1 info 3.14 $
>
```

Act: Task 06

NAME

Thread Pool

DIRECTORY

`t06/`

BINARY

`threadPool`

DESCRIPTION

Create a program that simulates the work of a thread pool.
Thread pool completes the given tasks asynchronously and requires to:

- implement `ThreadPool` class that has a:
  - constructor that:
    * creates and launches the requested number of threads
    * waits for a new task appearing in a `m_taskQueue` member function
    * uses a thread to execute task that appears in queue and pops it out of the queue
  - destructor that:
    * sets `m_isRunning` to `false`
    * notifies all threads using `m_condVar` member variable
    * waits for all threads to complete
  - `enqueueTask` member function that:
    * creates a task
    * moves it to `m_taskQueue`
    * notifies `m_condVar`
- use `main.cpp` from the SYNOPSIS to test Thread pool but do not stop here. Provide wider range of test cases to prove that you have fully completed this task

SYNOPSIS

```
/* ThreadPool.h */

class ThreadPool final {
 public:
    explicit ThreadPool(size_t threads);
    ~ThreadPool();

    ThreadPool(const ThreadPool&) = delete;
    ThreadPool(const ThreadPool&&) = delete;
```

```
    ThreadPool& operator=(const ThreadPool&) = delete;

    template <typename Function, class... Args>
    auto enqueueTask(Function&& func, Args&&... args);

 private:
    std::vector<std::thread> m_workerThreads;
    std::queue<std::packaged_task<void()>> m_taskQueue;
    std::condition_variable m_condVar;
    std::mutex m_queueMutex;
    bool m_isRunning{true};
};

/* main.cpp */

int compute_ackermann(int m, int n) {
    if (m == 0) {
        return n + 1;
    } else if (m > 0 && n == 0) {
        return compute_ackermann(m - 1, 1);
    } else if (m > 0 && n > 0) {
        return compute_ackermann(m - 1, compute_ackermann(m, n - 1));
    }

    return -1;
}

int main() {
    ThreadPool pool(5);
    auto af1 = pool.enqueueTask(compute_ackermann, -1, 7);
    auto af2 = pool.enqueueTask(compute_ackermann, 1, 5);
    auto af3 = pool.enqueueTask(compute_ackermann, 2, 3);
    auto af4 = pool.enqueueTask(compute_ackermann, 3, 10);

    const auto a5 = compute_ackermann(1, 5);
    const auto a1 = af1.get();
    const auto a2 = af2.get();
    const auto a3 = af3.get();
    const auto a4 = af4.get();

    std::cout << a1 << std::endl;
    std::cout << a2 << std::endl;
    std::cout << a3 << std::endl;
    std::cout << a4 << std::endl;
    std::cout << a5 << std::endl;

    return 0;
}
```

CONSOLE OUTPUT

```
>./threadPool | cat -e
-1$
7$
9$
8189$
7$
>
```

Share

PUBLISHING

Last but not least, the final stage of your work is to publish it. This allows you to share your challenges, solutions, and reflections with local and global audiences. During this stage, you will discover ways of getting external evaluation and feedback on your work. As a result, you will get the most out of the challenge, and get a better understanding of both your achievements and missteps.

To share your work, you can create:

- a text post, as a summary of your reflection
- charts, infographics or other ways to visualize your information
- a video, either of your work, or a reflection video
- an audio podcast. Record a story about your experience
- a photo report with a small post

Helpful tools:

- Canva - a good way to visualize your data
- QuickTime - an easy way to capture your screen, record video or audio

Examples of ways to share your experience:

- Facebook - create and share a post that will inspire your friends
- YouTube - upload an exciting video
- GitHub - share and describe your solution
- Telegraph - create a post that you can easily share on Telegram
- Instagram - share photos and stories from ucode. Don't forget to tag us :)

Share what you've learned and accomplished with your local community and the world. Use #ucode and #CBLWorld on social media.