# Object Oriented programming and software engineering – Lab 19

Adam Korytowski – 2025

## 1. Multithreading

Multithread programming is a way to make your program do multiple things at the same time by using threads. A thread is a lightweight process—like a mini-program running inside your main program. In C++, multithreading is available through the <thread> library.

### Why Do We Need It?

Imagine you're building a game where:

- One thread handles the player input,
- Another thread processes enemy behavior,
- A third one updates graphics or saves the game.

If everything ran in a single thread, one slow task could block everything else. With multithreading, you get better **performance**, **responsiveness**, and **efficient CPU usage**, especially on multicore processors.

### When Should You Use It?

Use multithreading when your program:

- Has **tasks that can run independently** (e.g., saving to a file and updating the screen),
- Needs to **respond quickly** (e.g., games or real-time apps),
- Can benefit from **parallel processing** (e.g., AI, physics simulations, or bulk data processing).

Examples:

```cpp
void printMessage()
{
    std::cout << "Hello from thread" << std::endl;
}

int main()
{
    std::thread t(printMessage);  // Start the thread
    t.join();                     // Wait for it to finish
    std::cout << "Back in main." << std::endl;
    return 0;
}
```

Calling a function with arguments:

```cpp
void greet(const std::string& name)
{
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main()
{
    std::thread t(greet, "Alice");
    t.join();
    return 0;
}
```

Simulating work with *sleep*:

```cpp
void slowTask()
{
    std::cout << "Starting task..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Task done." << std::endl;
}

int main()
{
    std::thread t(slowTask);
    t.join();
    return 0;
}
```

**Simple Mutex Example (Adding Safely to a Shared Variable)**

**What it does:**

- Two threads each add numbers to a **shared total**.

- We use a std::mutex to make sure **only one thread changes the total at a time**.

```cpp
int total = 0;          // Shared variable
std::mutex mtx;         // Mutex to protect it

void addNumbers()
{
    for (int i = 0; i < 5; ++i)
    {
        mtx.lock();                         // Lock before changing shared data
        total += 1;
        std::cout << "Total: " << total << std::endl;
        mtx.unlock();                       // Unlock after done
    }
}

int main()
{
    std::thread t1(addNumbers);
    std::thread t2(addNumbers);

    t1.join();
    t2.join();

    std::cout << "Final Total: " << total << std::endl;
    return 0;
}
```

**Why use mutex here?**

Without the mutex, both threads might try to change total at the same time → this could cause incorrect results or even crash your program.

Or using lock_guard:

```cpp
void addNumbers()
{
    for (int i = 0; i < 5; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // Automatically locks & unlocks
        total += 1;
        std::cout << "Total: " << total << std::endl;
    }
}
```

2. Tasks

Use your code from Lab17 or Lab18 and perform the following: (2 pts each)

- Create two separate threads and a class called Simulation. Each thread should simulate one object of each subclass performing an action by calling a method. This simulates two actions being performed at the same time. Example:

```
void simulateHonda() //inside of Simulation class
{
    Car* honda = new Honda();
    honda->drive();
    delete honda;
}

void simulateOpel() //inside of Simulation class
{
    Car* opel = new Opel();
    opel->drive();
    delete opel;
}
```

```
std::thread t1(/*calling your first function*/);
std::thread t2(/*calling your second function*/);
```

```
t1.join();
t2.join();
```

- Create a function that simulates logging or updating the status of an object in a loop (e.g., printing health, speed, energy, etc.). Use two threads to run this function for two different object types at the same time.

In two of your child classes:

```
void logStatus() override
{
    std::cout << "Honda status: Speed OK, Fuel OK";
}
```

In Simulation class:

```
void runObjectWithStatus(Car* obj)
{
    obj->drive();
    obj->logStatus();
    delete obj;
}
```

In main:

```
Car* honda = new Honda();
Car* opel = new Opel();

std::thread t1(runObjectWithStatus, honda);
std::thread t2(runObjectWithStatus, opel);

t1.join();
t2.join();
```

- Create multiple objects from different derived classes. Split them into two groups, and use two threads to process each group separately. Each object in the group should perform its own action through a virtual method. Example:

In Simulation class:

```
void processObjects(const std::vector<Car*>& objects)
{
    for (Car* obj : objects)
    {
        obj->drive();
    }
}
```

In main:

```
Car* obj1 = new Honda();
Car* obj2 = new Opel();
Car* obj3 = new Honda();
Car* obj4 = new Opel();

std::vector<Car*> group1 = { obj1, obj2 };
std::vector<Car*> group2 = { obj3, obj4 };

std::thread t1(processObjects, group1);
std::thread t2(processObjects, group2);

t1.join();
t2.join();
```

- Simulate two different objects trying to access the same shared resource (e.g., a log, a charging station, a weapon rack). Use std::mutex to ensure only one object accesses the resource at a time, avoiding conflicts or race conditions. Example:

In Simulation class:

```
std::mutex resourceMutex;
```

```
void useSharedResource(const std::string& name)
{
    std::cout << name << " is waiting to access the shared resource...\n";

    {
        std::lock_guard<std::mutex> lock(resourceMutex);
        std::cout << name << " is using the resource...\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        std::cout << name << " is done.\n";
    }
}
```

In main:

```
std::thread t1(useSharedResource, "Honda");
std::thread t2(useSharedResource, "Opel");

t1.join();
t2.join();
```

___

Extra tips:

a. To call the functions from Simulation class (e. g. “useSharedResource” – you need to create an object of Simulation class, then call it on the object, e. g.

Simulation* simulation = new Simulation();

Then, e. g.:

t1 = std::thread(&Simulation::useSharedResource, simulation, "Honda");

b. How to reuse an existing thread variable:
use:

t1 = std::thread(<function name>, <object name>);

instead of:

std::thread t1((<function name>, <object name>););