Object Oriented programming and software engineering – Lab 20

Adam Korytowski – 2025

1. Asynchronous Programming in C++

Asynchronous programming in C++ allows a program to perform tasks in the background **without blocking** the main execution thread. This is useful when:

- Tasks are time-consuming (e.g., I/O, network operations).
- You want better responsiveness (especially in GUI or real-time systems).
- You need to run code concurrently for performance.

C++11 introduced basic asynchronous support using <future>, <thread>, and <async>. More advanced features came in C++20 with coroutines (co_await, co_yield, co_return).

Example:

```cpp
int longCalculation()
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 42;
}

int main()
{
    std::cout << "Starting async task..." << std::endl;

    std::future<int> result = std::async(std::launch::async, longCalculation);

    std::cout << "Doing other work in main..." << std::endl;

    int value = result.get(); // Waits for longCalculation to finish
    std::cout << "Async result: " << value << std::endl;
```

This program starts an async task, continues running the main thread, and then retrieves the result later.

2. Tasks (3 pts each)

- Create a virtual method in the base class that simulates a time-consuming operation by sleeping for a short duration and then outputs a message. Override this method in two derived classes with different durations and messages. Then, run

these methods asynchronously for instances of the derived classes and wait for all to finish. Example:

```
class Car
{
public:
    virtual void performTask() = 0;
    virtual ~Car() {}
};

class Honda : public Car
{
public:
    void performTask() override
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "Honda task complete." << std::endl;
    }
};

class Opel : public Car
{
public:
    void performTask() override
    {
        std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "Opel task complete." << std::endl;
    }
};
```

In Simulation class from previous lab (if don't have it, create a new empty class called Simulation):

```
void runPerformTask(Car* car)
{
    car->performTask();
}
```

In main():

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

Simulation* simulation = new Simulation();

std::future<void> f1 = std::async(std::launch::async, &Simulation::runPerformTask, simulation, honda);
std::future<void> f2 = std::async(std::launch::async, &Simulation::runPerformTask, simulation, opel);
```

- Implement a virtual method in the base class that returns a string after simulating some processing delay. Override this method in two derived classes with different

delays and returned messages. Use async calls to run these methods on multiple objects, then collect and print all the returned strings. Example:

Implementation in child classes:

```
std::string computeResult() override
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return "Result from Honda";
}
```

In Simulation class:

```
std::string runComputeResult(Car* car)
{
    return car->computeResult();
}
```

In main:

```
std::vector<Car*> cars = { new Honda(), new Opel(), new Honda() };
std::vector<std::future<std::string>> futures;

for (Car* car : cars)
{
    futures.push_back(std::async(std::launch::async, &Simulation::runComputeResult, simulation, car));
}

for (auto& f : futures)
{
    std::cout << f.get() << std::endl;
}

for (Car* car : cars)
{
    delete car;
}
```

- Create a virtual method in the base class that returns an integer after a short delay, and override it in two derived classes with different returned values. Then, in your program, call these methods asynchronously and pass their results into a second method (also virtual) that takes an integer and outputs a message. This simulates chaining asynchronous computations with intermediate processing. Example:

In child classes:

```cpp
int generateValue() override
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 10;
}

void processValue(int value) override
{
    std::cout << "Honda processed value: " << value * 2 << std::endl;
}
```

```cpp
int generateValue() override
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 20;
}

void processValue(int value) override
{
    std::cout << "Opel processed value: " << value + 5 << std::endl;
}
```

In Simulation class:

```cpp
int runGenerate(Car* car)
{
    return car->generateValue();
}

void runProcess(Car* car, int value)
{
    car->processValue(value);
}
```

in main():

```cpp
std::vector<std::future<int>> int_futures;

for (Car* car : cars)
{
    int_futures.push_back(std::async(std::launch::async, &Simulation::runGenerate, simulation, car));
}

for (size_t i = 0; i < cars.size(); ++i)
{
    int value = int_futures[i].get();
    simulation->runProcess(cars[i], value);
}
```