

# 8- Практическое задание — Реализация параллельной обработки данных с использованием лямбда-выражений, многопоточности и наследования

## Table of Contents

Цель .....	2
1. Введение в лямбда-выражения с использованием наследования .....	2
2. Основы многопоточности .....	7
3. Реализация параллельной обработки данных с использованием лямбда-выражений, многопоточности и наследования.....	9
Самостоятельная задача: Параллельная обработка текстовых файлов.....	12

# Цель

Цель этой лабораторной работы заключается в изучении и применении продвинутых концепций программирования на C++, включая лямбда-выражения, многопоточность, и наследование в объектно-ориентированном программировании. Студенты начнут с освоения лямбда-выражений и базовых принципов многопоточности, применяя эти концепции для создания простой многопоточной программы. Затем, они перейдут к более сложной задаче, комбинируя эти элементы с наследованием для разработки системы параллельной обработки данных, что позволит им увидеть, как разные потоки могут работать одновременно над разными частями данных. Наконец, в самостоятельно реализованном задании студенты применят эти навыки на практике, создавая программу для параллельной обработки текстовых файлов, демонстрируя глубокое понимание многопоточности, лямбда-выражений и эффективности работы с файлами. Эта лабораторная работа не только укрепит понимание студентами ключевых аспектов современного C++, но и разовьет их навыки решения комплексных программных задач, актуальных в реальных проектах.

## 1. Введение в лямбда-выражения с использованием наследования

**Введение:** Лямбда-выражения в C++ предоставляют компактный и выразительный способ определения анонимных функций. Эта задача поможет понять, как лямбда-выражения могут быть использованы для создания мощных и гибких программных решений в контексте наследования ООП. Мы увидим на конкретном примере, как изменение логики обработки данных может быть легко реализовано через лямбда-выражения без необходимости изменения структуры класса или интерфейса.

**Лямбда-выражение:** Функция без имени, которая может быть определена внутри других функций для выполнения коротких фрагментов кода.

Лямбда-выражение можно представить как небольшую временную функцию, которую можно создавать прямо внутри вашего кода, без необходимости предварительно определять отдельную функцию. Это похоже на то, как написать быструю инструкцию на месте для выполнения конкретной задачи.

Лямбда-выражение следует следующему общему формату:

**[список\_захвата] (параметры) -> возвращаемый\_тип { тело }**

1. **Список захвата (необязательно):** Эта часть (в квадратных скобках) сообщает лямбда-выражению, к каким переменным извне его кода он может получить доступ. Мы подробно рассмотрим это позже.
2. **Параметры (необязательно):** это входные данные, которые может принимать лямбда-выражение, так же, как и обычная функция. Если входов нет, можно оставить скобки пустыми.
3. **Возвращаемый тип (необязательно):** Он указывает тип данных значения, которое вернет лямбда-выражение, аналогично обычной функции. Если не указано, по умолчанию он равен `void` (что означает, что ничего не возвращается).
4. **Тело (обязательно):** здесь вы пишете фактический код, который будет выполнять лямбда-выражение, заключенный в фигурные скобки `{}`. Это похоже на инструкции, которым следует функция.

Хотя указание возвращаемого типа и параметров необязательно (и список захвата тоже может быть пустым), возвращаемый тип лямбда-выражения не всегда равен `void` по умолчанию. Возвращаемый тип может быть автоматически выведен компилятором на основе выражения, возвращаемого из тела лямбда-выражения. Если в теле лямбда-выражения присутствует оператор возврата значения, компилятор определит возвращаемый тип на основе этого значения. Если оператор возврата отсутствует, тогда да, можно считать, что

возвращаемый тип — `void`. При этом, если в лямбда-выражении опущены все элементы, кроме тела, то есть используется самая простая форма `[] { тело }`, это означает, что лямбда:

- Не захватывает никаких переменных извне (пустой список захвата).
- Не принимает параметров (пустые круглые скобки, которые могут быть опущены).
- Возвращаемый тип может быть автоматически выведен или явно указан как `void`, если лямбда не возвращает значение.
- Содержит обязательное тело, где выполняется непосредственно логика лямбда-выражения.

Такая лямбда-функция может использоваться, например, для выполнения простого действия, не требующего входных данных и не возвращающего результата, или же для возвращения значения, тип которого будет выведен автоматически, основываясь на содержимом тела функции.

### Зачем использовать лямбда-выражения?

Лямбда-выражения удобны тем, что позволяют писать короткий, чистый код для небольших задач, особенно при работе с алгоритмами или функциями, которые ожидают другие функции в качестве аргументов. Вот некоторые причины, по которым они полезны:

1. **Краткость:** они могут сделать ваш код более компактным, избегая необходимости определять отдельные именованные функции для простых задач.
2. **Читаемость:** при правильном использовании они могут улучшить читаемость, сохраняя код, который оперирует данными, рядом с самими данными.
3. **Гибкость:** они позволяют легко передавать небольшие фрагменты функциональности, делая ваш код более адаптивным.

Посмотрим, как лямбда-выражение можно использовать для сортировки списка чисел:

```
std::vector<int> numbers = {3, 1, 4, 5, 2};

// Сортировка чисел по возрастанию с помощью лямбда-выражения
std::sort(numbers.begin(), numbers.end(), [](int a, int b) { return a < b; });
// Теперь numbers будет равно {1, 2, 3, 4, 5}
```

- `std::vector<int> numbers = {3, 1, 4, 5, 2};` инициализирует вектор целых чисел значениями {3, 1, 4, 5, 2}.
- `std::sort(numbers.begin(), numbers.end(), [](int a, int b) { return a < b; });` сортирует элементы `numbers` от начала до конца в порядке возрастания, потому что лямбда-выражение `(int a, int b) { return a < b; }` возвращает `true`, если первый аргумент (`a`) меньше второго (`b`), указывая `std::sort` расположить `a` перед `b`.
- После сортировки `numbers` действительно будет равен {1, 2, 3, 4, 5}, как и предполагалось. Это стандартное использование `std::sort` с лямбда-выражением для определения пользовательских критериев сортировки, в данном случае, простой сортировки по возрастанию.

### Список захвата - Взять вещи с собой (необязательно):

Список захвата позволяет лямбда-выражению получать доступ к переменным извне его собственного кода. Вот некоторые распространенные способы его использования:

- `[]`: Это ничего не захватывает извне, делая лямбда-выражение самодостаточным.
- `[переменная]`: это захватывает одну переменную по значению, то есть лямбда-выражение получает копию значения переменной.
- `[&]`: Это захватывает все переменные извне по ссылке, то есть лямбда-выражение может модифицировать исходные переменные.

**Помните:** Лямбда-выражения - мощный инструмент, но используйте их разумно. Если логика становится сложной, подумайте о создании отдельной именованной функции для улучшения читаемости.

Теперь наш полный Пример:

```
#include <iostream>
#include <vector>
#include <algorithm> // Для std::for_each
#include <functional> // Для std::function

// Базовый класс Processor с виртуальной функцией process для обработки данных
class Processor {
public:
    // Функция для обработки данных, принимает вектор и лямбда-выражение
    virtual void process(std::vector<int>& data, std::function<void(int)> func) = 0;
};

// Производный класс LambdaProcessor, наследующий от Processor
class LambdaProcessor : public Processor {
public:
    // Переопределение функции process с использованием лямбда-выражения в качестве параметра
    void process(std::vector<int>& data, std::function<void(int)> func) override {
        // Применение лямбда-выражения к каждому элементу вектора
        std::for_each(data.begin(), data.end(), func);
    }
};

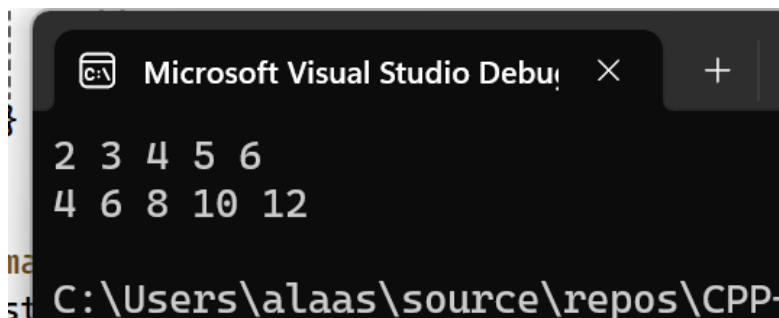
int main() {
    std::vector<int> data = { 1, 2, 3, 4, 5 }; // Исходные данные

    LambdaProcessor processor; // Создание экземпляра производного класса

    // Вызов функции обработки данных с лямбда-выражением для увеличения каждого элемента на 1
    processor.process(data, [](int& val) { val += 1; });
    // Вывод результатов после первой обработки
    for (int val : data) {
        std::cout << val << " "; // Должно вывести 2 3 4 5 6
    }
    std::cout << std::endl;

    // Вызов функции обработки данных с другим лямбда-выражением для умножения каждого элемента на 2
    processor.process(data, [](int& val) { val *= 2; });
    // Вывод результатов после второй обработки
    for (int val : data) {
        std::cout << val << " "; // Должно вывести 4 6 8 10 12
    }
    std::cout << std::endl;

    return 0;
}
```



Этот код демонстрирует использование наследования в ООП и лямбда-выражений в контексте обработки данных. Он предназначен для начинающих студентов, чтобы помочь им понять:

- **Наследование:** как базовый класс может использоваться для создания производных классов.
- **Виртуальные функции:** как базовый класс может содержать виртуальные функции, которые переопределяются в производных классах.
- **Лямбда-выражения:** как анонимные функции могут быть использованы в качестве параметров функций.

Объяснение кода:

### 1. Базовый класс Processor:

```
// Базовый класс Processor с виртуальной функцией process для обработки данных
class Processor {
public:
    // Функция для обработки данных, принимает вектор и лямбда-выражение
    virtual void process(std::vector<int>& data, std::function<void(int)> func) = 0;
};
```

- Класс Processor объявляет виртуальную функцию process, которая принимает два параметра:
  - std::vector<int>& data: ссылка на вектор целых чисел, содержащий данные для обработки.
  - std::function<void(int)> func: лямбда-выражение, которое будет применяться к каждому элементу вектора.

### 2. Производный класс LambdaProcessor:

```
// Производный класс LambdaProcessor, наследующий от Processor
class LambdaProcessor : public Processor {
public:
    // Переопределение функции process с использованием лямбда-выражения в качестве параметра
    void process(std::vector<int>& data, std::function<void(int)> func) override {
        // Применение лямбда-выражения к каждому элементу вектора
        std::for_each(data.begin(), data.end(), func);
    }
};
```

- Класс LambdaProcessor наследует от Processor.
- Функция process переопределяется в классе LambdaProcessor.
- Внутри переопределенной функции process используется std::for\_each для применения лямбда-выражения func к каждому элементу вектора data. std::for\_each — это алгоритм из стандартной библиотеки C++, который применяет функцию или лямбда-выражение к каждому элементу в диапазоне, заданном двумя итераторами. Этот алгоритм не модифицирует диапазон, но функция, применяемая к элементам, может изменять их. std::for\_each возвращает функцию после её применения ко всем элементам, что позволяет выполнить дополнительные действия с этой функцией после обхода диапазона.

В предоставленном код символ ->, который используется для указания возвращаемого типа лямбда-выражения, действительно отсутствует. Однако это не ошибка, поскольку в данном случае лямбда-выражение не является частью непосредственно этого кода. Вместо этого, в функцию process передается аргумент func типа std::function<void(int)>, который может быть лямбда-выражением, но само лямбда-выражение формируется и передается в process извне. Сигнатура std::function<void(int)> указывает, что func должно быть вызываемым объектом (например, лямбда-выражением, функцией или объектом с перегруженным оператором ()), который принимает один параметр типа int& и не возвращает значение (то есть имеет возвращаемый тип void). В этом контексте -> не требуется, потому что возвращаемый тип уже ясно указан в типе std::function. Если бы вы хотели создать лямбда-выражение для передачи в process в качестве параметра

func, вы бы могли сделать это без явного указания возвращаемого типа, так как он уже известен из контекста (void). Например:

```
std::vector<int> data = {1, 2, 3, 4, 5};
```

```
process(data, [](int& x) {
```

```
// Модификация x или выполнение действия с x
```

```
});
```

В этом случае, лямбда принимает один аргумент по ссылке (int& x) и не возвращает значение, что соответствует требованию std::function<void(int&)>.

### 3. Функция main:

```
int main() {
    std::vector<int> data = { 1, 2, 3, 4, 5 }; // Исходные данные

    LambdaProcessor processor; // Создание экземпляра производного класса

    // Вызов функции обработки данных с лямбда-выражением для увеличения каждого элемента на 1
    processor.process(data, [](int& val) { val += 1; });
    // Вывод результатов после первой обработки
    for (int val : data) {
        std::cout << val << " "; // Должно вывести 2 3 4 5 6
    }
    std::cout << std::endl;

    // Вызов функции обработки данных с другим лямбда-выражением для умножения каждого элемента на 2
    processor.process(data, [](int& val) { val *= 2; });
    // Вывод результатов после второй обработки
    for (int val : data) {
        std::cout << val << " "; // Должно вывести 4 6 8 10 12
    }
    std::cout << std::endl;

    return 0;
}
```

- Функция main создает вектор data с пятью элементами.
- Создается экземпляр класса LambdaProcessor под названием processor.
- Функция process вызывается дважды:
  - Первый раз с лямбда-выражением, которое увеличивает каждый элемент вектора на 1.
  - Второй раз с лямбда-выражением, которое умножает каждый элемент вектора на 2.
- После каждого вызова process элементы вектора выводятся на консоль.

#### Этот код иллюстрирует:

- Как базовый класс может использоваться для определения общего интерфейса (функция process).
- Как производный класс может переопределять виртуальную функцию для реализации конкретного поведения (использование лямбда-выражения).
- Как лямбда-выражения могут быть использованы для реализации анонимных функций, которые могут быть переданы в качестве параметров.

## 2. Основы многопоточности

**Введение:** Многопоточность — это способность программы выполнять несколько задач одновременно, используя для этого множество потоков выполнения. Это позволяет увеличить эффективность и производительность программы, особенно на компьютерах с многоядерными процессорами. В этой задаче мы рассмотрим основы создания многопоточных программ в C++ с использованием библиотеки `<thread>`. Наша цель - создать простую многопоточную программу, которая демонстрирует, как разные потоки могут работать параллельно для обработки данных.

### Определения:

1. **Поток (Thread):** Базовая единица выполнения кода. В одном процессе может быть множество потоков, которые могут выполняться параллельно.
2. **Многопоточность (Multithreading):** Способность приложения одновременно выполнять несколько задач, распределяя их по разным потокам.
3. **Мьютекс (Mutex):** Средство синхронизации, используемое для предотвращения одновременного доступа нескольких потоков к общему ресурсу.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <locale.h>

// **Мьютекс для синхронизации доступа к стандартному выводу (cout)**
std::mutex coutMutex;

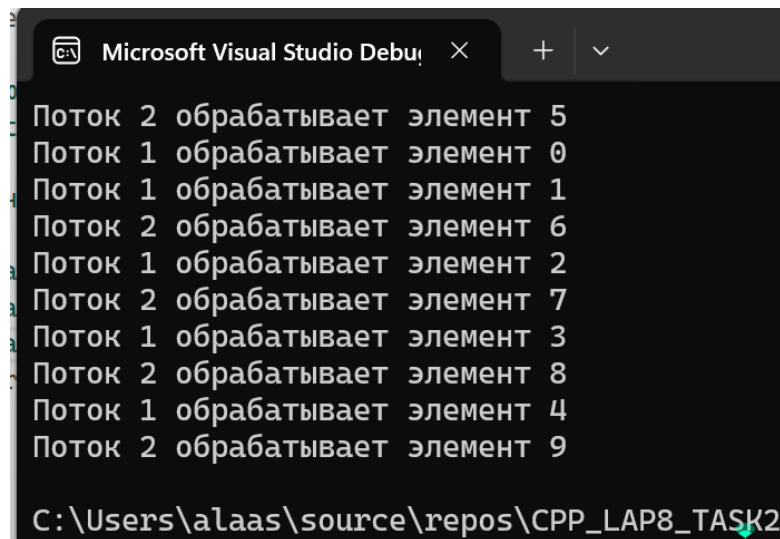
// **Функция, демонстрирующая работу потока с определенной частью данных**
//
// @param threadNum - Номер потока
// @param start      - Начальный индекс обрабатываемых данных
// @param end        - Конечный индекс обрабатываемых данных
void threadWork(int threadNum, int start, int end) {
    for (int i = start; i < end; ++i) {
        // **Блокировка мьютекса для обеспечения синхронного доступа к cout**
        {
            std::lock_guard<std::mutex> guard(coutMutex);
            // **Вывод сообщения о том, что поток работает над определенным элементом**
            std::cout << "Поток " << threadNum << " обрабатывает элемент " << i << std::endl;
        }
        // **Имитация работы потока путем введения небольшой задержки**
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}

int main() {
    setlocale(LC_ALL, "Russian");

    // **Запуск двух потоков, каждый из которых обрабатывает свою часть данных**
    std::thread t1(threadWork, 1, 0, 5);
    std::thread t2(threadWork, 2, 5, 10);

    // **Ожидание завершения работы потоков**
    t1.join();
    t2.join();

    return 0;
}
```



```

Microsoft Visual Studio Debug Console
Поток 2 обрабатывает элемент 5
Поток 1 обрабатывает элемент 0
Поток 1 обрабатывает элемент 1
Поток 2 обрабатывает элемент 6
Поток 1 обрабатывает элемент 2
Поток 2 обрабатывает элемент 7
Поток 1 обрабатывает элемент 3
Поток 2 обрабатывает элемент 8
Поток 1 обрабатывает элемент 4
Поток 2 обрабатывает элемент 9

C:\Users\alaas\source\repos\CPP_LAP8_TASK2

```

#### Объяснение кода:

- Определение мьютекса: `std::mutex coutMutex` используется для синхронизации доступа к стандартному выводу, предотвращая наложение вывода от разных потоков.
- Функция `threadWork`: Эта функция демонстрирует выполнение работы потоком на определенном диапазоне данных. Использование `std::lock_guard` с мьютексом гарантирует, что при выводе сообщений не будет конфликта между потоками.
- Создание и запуск потоков: В `main` функции создаются и запускаются два потока с использованием функции `threadWork`, каждый из которых обрабатывает свою уникальную часть данных.
- Чередование выполнения: Введение задержек между операциями в `threadWork` увеличивает шансы на то, что операционная система будет переключаться между потоками, демонстрируя чередование их работы. Это помогает понять, как многопоточность позволяет различным частям программы выполняться одновременно.

Эта задача предназначена для демонстрации основ многопоточного программирования и важности синхронизации при работе с общими ресурсами. Чередование выполнения потоков — ключевое понятие, которое необходимо понимать при разработке многопоточных приложений, поскольку оно влияет на производительность и корректность программы.



### 3. Реализация параллельной обработки данных с использованием лямбда-выражений, многопоточности и наследования

**Введение:** Этот пример кода демонстрирует мощную комбинацию трех ключевых концепций современного программирования на C++: лямбда-выражений, многопоточности и наследования. Задача реализует систему, в которой данные обрабатываются параллельно в нескольких потоках. Каждый поток работает с собственным сегментом данных, применяя к каждому элементу заданную функцию. Это упражнение предназначено для обучения студентов принципам параллельного программирования и демонстрации возможностей C++ для работы с анонимными функциями и потоками.

```
#include <iostream>
#include <vector>
#include <thread>
#include <functional>
#include <mutex>
#include <chrono>
#include <locale.h>

// **Мьютекс для синхронизации доступа к стандартному выводу (cout)**
std::mutex coutMutex;

// **Базовый класс для обработки данных**
//
// Абстрактный метод `process` должен быть переопределен в производных классах
// для реализации конкретной логики обработки данных.
class DataProcessor {
public:
    virtual void process(std::vector<int>& data, std::function<void(int)> func) = 0;
};

// **Производный класс, реализующий параллельную обработку данных**
class ParallelDataProcessor : public DataProcessor {
public:
    // **Переопределение метода `process` из базового класса**
    void process(std::vector<int>& data, std::function<void(int)> func) override {
        // **Создание контейнера для потоков**
        std::vector<std::thread> threads;

        // **Вычисление размера части данных для каждого потока**
        // (Предполагается, что размер данных кратен 4)
        int partSize = data.size() / 4;

        // **Запуск потоков для обработки каждой части данных**
        for (int i = 0; i < 4; ++i) {
            // Создание нового потока с использованием лямбда-выражения:
            // * [&]: Захват внешних переменных по ссылке (для доступа к coutMutex, data и др.).
            // * [i, func]: Захват `i` (индекс части данных) и `func` (функции обработки) по значению
            // * для использования внутри лямбда-функции.
            threads.emplace_back([&, i, func]() {
                // **Определение диапазона обрабатываемых данных для текущего потока**
                int start = i * partSize;
                int end = (i + 1) * partSize;

                // **Обработка данных в цикле**
                for (int j = start; j < end; ++j) {
                    // **Имитация задержки для демонстрации чередования потоков**
                    std::this_thread::sleep_for(std::chrono::milliseconds(50 * (j - start + 1)));

                    // **Выполнение функции `func` для каждого элемента данных**
                    func(data[j]);
                }

                // **Вывод сообщения о выполнении операции**
            });
        }
    }
};
```

```

        std::lock_guard<std::mutex> guard(coutMutex);
        std::cout << "Поток " << i << " обработал элемент " << j << std::endl;
    }
    });
}

// **Ожидание завершения всех потоков**
for (auto& t : threads) {
    t.join();
}
}
};

int main() {
    setlocale(LC_ALL, "Russian");

    // **Пример данных**
    std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8 };

    // **Создание объекта класса ParallelDataProcessor**
    ParallelDataProcessor processor;

    // **Обработка данных с помощью лямбда-выражения, увеличивающего каждый элемент на 1**
    processor.process(data, [](int& n) { n += 1; });

    // **Вывод результатов обработки**
    std::lock_guard<std::mutex> guard(coutMutex);
    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

```

Microsoft Visual Studio Debug Console
Поток 1 обработал элемент 2
Поток 0 обработал элемент 0
Поток 3 обработал элемент 6
Поток 2 обработал элемент 4
Поток 0 обработал элемент 1
Поток 1 обработал элемент 3
Поток 2 обработал элемент 5
Поток 3 обработал элемент 7
2 3 4 5 6 7 8 9
C:\Users\alaaas\source\repos\CPP

```

#### Объяснение кода:

- Мьютекс для синхронизации вывода: используется для предотвращения одновременного доступа к cout из разных потоков, что могло бы привести к наложению вывода.
- Абстрактный базовый класс DataProcessor: определяет интерфейс для классов обработки данных с абстрактным методом process, который должен быть реализован в производных классах.
- Производный класс ParallelDataProcessor: реализует параллельную обработку данных, создавая для каждого сегмента данных отдельный поток.
- Лямбда-выражения для обработки данных: передается в метод process и применяется к каждому элементу данных в параллельных потоках.

- Демонстрация чередования выполнения потоков: через задержки и вывод сообщений о прогрессе работы потоков, позволяя наблюдать параллельную обработку данных в реальном времени.

Этот пример демонстрирует, как можно эффективно использовать многопоточность для ускорения обработки данных, применяя функции обработки к большим массивам данных параллельно. Такой подход является ценным навыком в современной разработке программного обеспечения, особенно в задачах, требующих интенсивных вычислений и обработки больших объемов данных.

# Самостоятельная задача: Параллельная обработка текстовых файлов

Создайте систему на C++, которая читает текст из нескольких файлов, обрабатывает данные параллельно с использованием лямбда-выражений и многопоточности, а затем сохраняет результат в новых файлах. Каждый поток должен обрабатывать текст из своего файла, например, выполняя подсчет слов или модифицируя текст определенным образом. Используйте наследование для создания общего интерфейса обработчиков данных и мьютексы для синхронизации доступа к ресурсам, если это необходимо. Результатом работы каждого потока должен быть новый файл, в котором содержится обработанный текст. Это задание поможет вам понять, как эффективно использовать многопоточность и лямбда-выражения для выполнения параллельной обработки данных, а также даст практику в работе с файлами в C++.