

Санкт-Петербургский политехнический университет Петра Великого  
Институт машиностроения, материалов и транспорта  
Кафедра «Мехатроника и роботостроение (при ЦНИИ РТК)»»

## **Курсовой проект**

по дисциплине «Объектно-ориентированное программирование»  
«Потокобезопасная очередь без блокировок»

Студент гр. 3331506/00401

Земский С. А.

Преподаватель

Ананьевский М. С.

«    » \_\_\_\_\_ 2023 г.

Санкт-Петербург  
2023 г.

## Введение

В настоящее время многопоточность стала неотъемлемой частью разработки программного обеспечения. Она позволяет увеличить производительность приложений и обеспечить более эффективное использование ресурсов компьютера. Однако, при работе с многопоточностью необходимо учитывать возможность одновременного доступа к общим ресурсам из разных потоков, что может привести к проблемам синхронизации и блокировок.

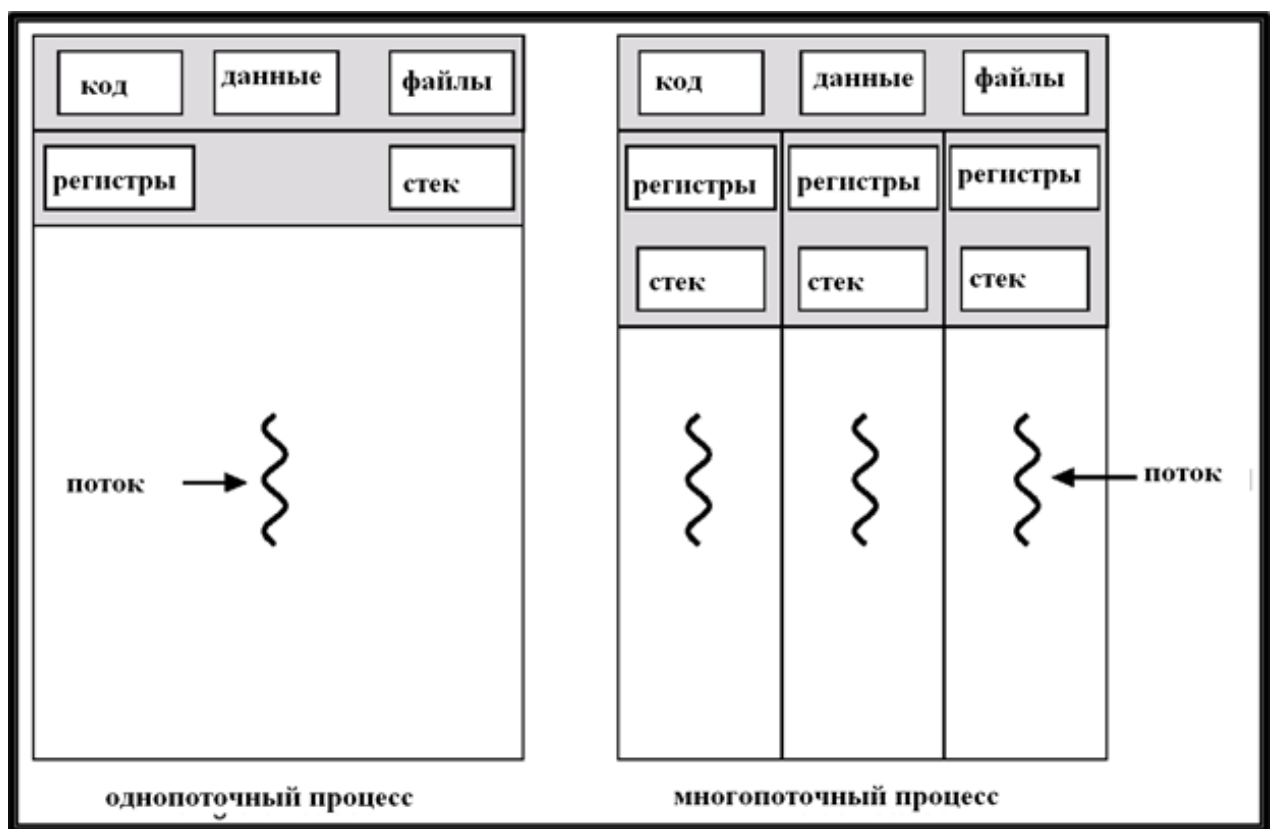


Рисунок 1 – Принципы работы однопоточного/многопоточного проекта

Одной из наиболее распространенных проблем при работе с многопоточностью является проблема блокировок. Блокировки возникают в тех случаях, когда один поток пытается получить доступ к ресурсу, который уже занят другим потоком. В результате этого поток, который ждет освобождения ресурса, блокируется и не может продолжить свою работу до тех пор, пока ресурс не будет освобожден.

Для решения проблемы блокировок и обеспечения безопасной записи и чтения из разных потоков была разработана очередь без блокировок на языке программирования C++. Очередь без блокировок позволяет избежать блокировок и увеличить производительность приложений, работающих с многопоточностью.

Цель данного курсового проекта - разработать и реализовать очередь без блокировок на языке программирования C++, которая будет обеспечивать безопасную запись и чтение из разных потоков. В рамках работы будет проведен анализ существующих решений, разработана архитектура очереди без блокировок, реализованы необходимые методы и проведены тесты на производительность и безопасность работы очереди.

## 1. Технические требования

Реализовать простую шаблонную циклическую очередь, безопасную для одновременной записи и чтения из двух потоков (один поток читает, другой пишет), не использующей механизмы взаимной блокировки потоков.

### Требования:

- Память для хранения данных аллоцируется статически внутри очереди.
- Два шаблонных параметра: `T` – тип данных, `CAPACITY` – ёмкость очереди (максимальное количество хранимых элементов типа `T`)
- Очередь должна поддерживать конструктор копирования и оператор присваивания

### Важные замечания, возникшие во время написания программы:

- Т. к. очередь предназначена для работы с шаблонами всё описание класса было в header-based библиотеке.
- Для того, чтобы очередь могла использоваться для произвольных типов данных нельзя было использовать `std::atomic<T>`, который сильно ограничивает возможные типы данных. Для решения данной задачи использовалась библиотека [`std::aligned\_storage`](#). С её помощью шаблонный тип `T` приводился к тривиальному типу стандартной компоновки, подходящим для использования в качестве неинициализированного хранилища для любого объекта.
- Из-за большой сложности обработки исключений они не использовались.
- Безопасность обеспечивалась не использованием блокировок(`mutex`-ов), а тем, что методы добавления элемента (`push`) и извлечения (`pop`) модифицировали только указатели на конец (`rear`) и начало (`front`) очереди соответственно.

## 2. Синтаксис программы

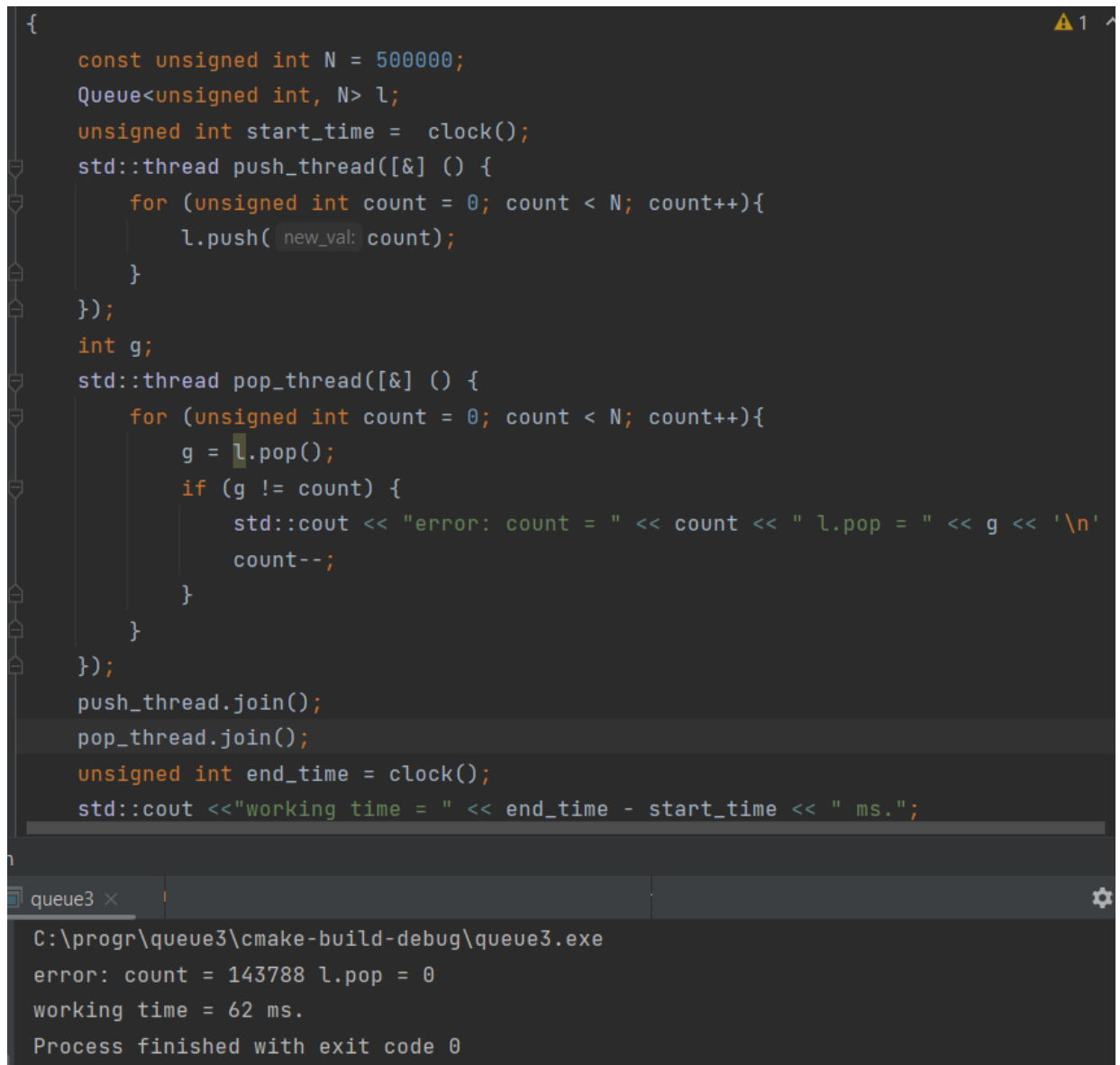
·Библиотека <queue.h> включает в себя описание класса Queue, который содержит :

Таблица 1 — Описание класса

1 Данные:	Указатели на первый, последний элемент, массив данных T, флаг пустоты очереди.
2 Конструкторы	Конструктор по умолчанию, копирования, деструктор
3 Операторы	Операторы присваивания и обращения по интдексу
4 Функции	Вставки (push) и извлечения (pop) элемента.

Для оценки корректности работы алгоритма была создана соответствующая программа, использующая описанную библиотеку. Результаты исследования представлены в следующем разделе.

### 3. Анализ результатов



```
{
    const unsigned int N = 500000;
    Queue<unsigned int, N> l;
    unsigned int start_time = clock();
    std::thread push_thread([&] () {
        for (unsigned int count = 0; count < N; count++){
            l.push( new_val: count);
        }
    });
    int g;
    std::thread pop_thread([&] () {
        for (unsigned int count = 0; count < N; count++){
            g = l.pop();
            if (g != count) {
                std::cout << "error: count = " << count << " l.pop = " << g << '\n';
                count--;
            }
        }
    });
    push_thread.join();
    pop_thread.join();
    unsigned int end_time = clock();
    std::cout << "working time = " << end_time - start_time << " ms.";
}

queue3 x
C:\progr\queue3\cmake-build-debug\queue3.exe
error: count = 143788 l.pop = 0
working time = 62 ms.
Process finished with exit code 0
```

Рисунок 2 – Оценка безопасности хранимых данных

В данном эксперименте создавалась очередь на 500000 элементов, которая параллельно заполняется и опустошается. Для оценки возможной потери данных была разработана программа, в которой:

- 1) 1 Поток добавляет элементы от 0 до  $N - 1$ .
- 2) 2 Поток считывает эти данные, и если они не равны ожидаемым, выводит в консоль номер итерации, на котором произошла ошибка и пробует повторить её.

По выводу в консоль видно, что из 500000 элементов лишь 1 раз не удалось извлечь элемент. По этим данным можно сказать, что очередь

является безопасной для записи/чтения из 2 потоков и может использоваться для соответствующих задач. Исследовательская задача выполнена.

#### **4. Список использованной литературы**

1. [https://en.cppreference.com/w/cpp/types/aligned\\_storage](https://en.cppreference.com/w/cpp/types/aligned_storage)
2. <https://habr.com/ru/articles/219201/>



## 5. Приложение

### Приложение A queue3.h

```
#ifndef QUEUE3_QUEUE3_H
#define QUEUE3_QUEUE3_H

#include <iostream>
#include <cstring>
#include <type_traits>

template<typename T, const unsigned int CAPACITY>
class Queue {
private:
    long front;
    long rear;
    std::aligned_storage_t<sizeof(T), alignof(T)> queue[CAPACITY];

public:
    bool empty;

public:
    Queue() : front(0), rear(0), empty(true) {};
    Queue(const Queue &other);
    ~Queue() = default;

public:
    Queue& operator=(const Queue &other);

private: // private for impossibility user's "breaking" structure of class
    const T& operator[](unsigned int pos) const {return
        *std::launder(reinterpret_cast<const T*>(&queue[pos]));}

public:
    bool push(const T& new_val);
    T pop();
};

template<typename T, const unsigned int CAPACITY>
Queue<T, CAPACITY>::Queue(const Queue &other) {
    for (int index = 0; index < CAPACITY; index++) {
        ::new(static_cast<std::aligned_storage_t<sizeof(T), alignof(T)>*>
            (&queue[index]))
            T(*std::launder(reinterpret_cast<const T*>(&other.queue[index])));
    }
    front = other.front;
    rear = other.rear;
    empty = other.empty;
}

template<typename T, const unsigned int CAPACITY>
Queue<T, CAPACITY>& Queue<T, CAPACITY>::operator=(const Queue &other) {
    for (int index = 0; index < CAPACITY; index++) {
        ::new(static_cast<std::aligned_storage_t<sizeof(T), alignof(T)>*>
            (&queue[index]))
            T(*std::launder(reinterpret_cast<const T*>(&other.queue[index])));
    }
    front = other.front;
    rear = other.rear;
    empty = other.empty;
}
```

```

        return *this;
    }

template<typename T, const unsigned int CAPACITY>
bool Queue<T, CAPACITY>::push(const T &new_val) {
    if (front == (rear + 1) % CAPACITY) return false;
    new(static_cast<std::aligned_storage_t<sizeof(T), alignof(T)>*>
        (&queue[empty? rear: (rear + 1) % CAPACITY])) T(new_val);
    rear = empty ? rear: (rear + 1) % CAPACITY;
    empty = false;
    return true;
}

template<typename T, const unsigned int CAPACITY>
T Queue<T, CAPACITY>::pop() {
    if (empty) return T();
    long temp_front = front;
    if (front == rear) empty = true;
    else front = (front + 1) % CAPACITY;
    T output_val;
    ::new(&output_val) T(*std::launder(reinterpret_cast<const
T*>(&queue[temp_front])));
    memset(&queue[temp_front], 0, sizeof(T));
    return output_val;
}

#endif //QUEUE3_QUEUE3_H

```

## Приложение В

### main.cpp

```
#include <iostream>
#include "queue3.h"
#include <thread>
#include <ctime>

int main()
{
    const unsigned int N = 500000;
    Queue<unsigned int, N> l;
    unsigned int start_time = clock();
    std::thread push_thread([&] () {
        for (unsigned int count = 0; count < N; count++){
            l.push(count);
        }
    });
    int g;
    std::thread pop_thread([&] () {
        for (unsigned int count = 0; count < N; count++){
            g = l.pop();
            if (g != count) {
                std::cout << "error: count = " << count << " l.pop = " << g
<< '\n' ;
                count--;
            }
        }
    });
    push_thread.join();
    pop_thread.join();
    unsigned int end_time = clock();
    std::cout << "working time = " << end_time - start_time << " ms.";
    return 0;
}
```