Основы разработки на C++: красный пояс

Неделя 2

Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода

Оглавление

Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода

2.1. Принципы оптимизации кода

2.1.1. Первое правило оптимизации кода

Избегайте преждевременной оптимизации. Преждевременная оптимизация – это упор на производительность в ущерб простоте, дизайну, понятности, поддерживаемости и так далее. В соответствии с принципом Парето 80% работы программы тратится на исполнение 20% кода. Эти 20% и нужно оптимизировать, но для начала их нужно найти. Делать максимально быстрым весь код тяжело, время на это будет уходить впустую. Преждевременная оптимизация приводит к необоснованному усложнению кода, затруднению поддержки и замедлению разработки. Как правило код надо стараться сделать правильным, простым, а только потом быстрым. Это не значит, что о производительности вообще не надо думать.

2.1.2. Второе правило оптимизации кода

Код не может быть просто быстрым или медленным. Он может быть достаточно или не достаточно быстрым для задачи, которую он решает. Рассмотрим пример.

```
vector<string> GenerateBigVector() {
  vector<string> result;
  for (int i = 0; i < 28000; ++i) {
    result.insert(begin(result), to_string(i));
  }
  return result;
```

```
}

int main() {
  cout << GenerateBigVector().size() << endl;
  return 0;
}
```

Функция `GenerateBigVector` генерирует вектор из 28000 элементов, программа выводит размер этого вектора. Замерим время работы программы. Программа работает 4.337 секунды. Понятно, что для данной задачи это медленно.

Мы можем бороться с проблемой интуитивно. Создадим версию `GenerateBigVector`, которая будет принимать вектор по ссылке, тогда не возникнет лишнего копирования и предположительно программа станет работать быстрее. Сделаем вектор из 32000 элементов.

```
void GenerateBigVector(vector<string>& result) {
  for (int i = 0; i < 32000; ++i) {
    result.insert(begin(result), to_string(i));
  }
}
```

Запустим программу с такой версией функции `GenerateBigVector` и замерим время работы. Программа отработала за 5.638 секунды. Это говорит о том, что интуиция нас подвела. Таким образом, мы приходим ко второму правилу оптимизации, которое звучит просто: **замеряйте!**

Интуиция часто даёт неверные результаты, а измерения объективны, они позволяют найти то место программы, из-за которой она работает медленно.

Замерим, сколько работает программа, а также замерим, сколько работает цикл по формированию вектора.

```
vector<string> GenerateBigVector() {
  vector<string> result;
  auto start = steady_clock::now();
  for (int i = 0; i < 28000; ++i) {
    result.insert(begin(result), to_string(i));
  }
  auto finish = steady_clock::now();
  auto duration = finish - start;
  cerr << "Cycle: "
      << duration_cast<milliseconds>(duration).count()
```

```
      << endl;
  return result;
}

int main() {
  auto start = steady_clock::now();
  cout << GenerateBigVector().size() << endl;
  auto finish = steady_clock::now();
  auto duration = finish - start;
  cerr << "Total: "
    << duration_cast<milliseconds>(duration).count()
    << endl;
  return 0;
}
```

Замерив время, мы понимаем, что общее время работы программы и время формирования вектора совпадает. Большая часть времени уходит на работу в цикле.

```
result.insert(begin(result), to_string(i));
```

Здесь мы вставляем каждый следующий элемент в начало вектора. Чтобы вставить элемент в начало, нужно сдвинуть текущее содержимое вектора вправо. Изменим программу и будем вставлять каждый новый элемент не в начало вектора, а в конец.

```
result.push_back(to_string(i));
```

Такая программа отработает всего за 6 миллисекунд.

Прежде чем ускорять код, замерьте, сколько он работает. Если это недостаточно быстро, то начинайте искать узкие места. Интуиция не работает – замеряйте!

2.1.3. Разработка своего профайлера

На практике для нахождения медленно работающего места в программе используются специальные инструменты, которые называются профайлерами. Некоторые из них:

- gperftools (Linux, Mac OS);

- perf (Linux);
- VTune (Windows, Linux).

Мы сделаем свой простой профайлер и будем им пользоваться.

Замерять время работы с помощью способа из предыдущего пункта громоздко. Напишем специальный класс.

```
class LogDuration {
public:
  LogDuration()
    : start(steady_clock::now())
  {
  }

  ~LogDuration() {
    auto finish = steady_clock::now();
    auto dur = finish - start;
    cerr << duration_cast<milliseconds>(dur).count()
      << " ms" << endl;
  }
private:
  steady_clock::time_point start;
};
```

`start` – это момент начала измерений. В конструкторе мы записываем в это поле `steady_clock::now()`. В деструкторе мы запоминаем момент окончания работы. Вычисляем длительность работы (разность). Выводим длительность в миллисекундах.

Теперь для измерения длительности работы блока кода мы пишем перед ним `LogDuration input;` и оборачиваем конструкцию в фигурные скобки:

```
{
  LogDuration input;
  for (int i = 0; i < element_count; ++i) {
    int x;
    cin >> x;
    elements.insert(x);
  }
}
```

Улучшим читаемость результатов работы программы.

```
class LogDuration {
  public:
    explicit LogDuration(const string& msg = "")
      : message(msg + ": ")
      , start(steady_clock::now())
    {
    }

    ~LogDuration() {
      auto finish = steady_clock::now();
      auto dur = finish - start;
      cerr << message
        << duration_cast<milliseconds>(dur).count()
        << " ms" << endl;
    }
  private:
    string message;
    steady_clock::time_point start;
};
```

Теперь мы можем добавлять сообщение, например:

```
{
  LogDuration input("Input");
  for (int i = 0; i < element_count; ++i) {
    int x;
    cin >> x;
    elements.insert(x);
  }
}
// Input: 113 ms
```

2.1.4. Совершенствование своего профайлера

Посмотрим, как мы пользуемся классом `LogDuration`.

```
{
  LogDuration input("Input");
  ...
}
```

Мы объявляем переменную с именем `input` и в качестве сообщения тоже используем `"Input"`. Имена переменных нам не нужны, мы не обращаемся к ним в коде. Нам нужен способ, который позволит объявлять объекты класса `LogDuration`, но избавит от необходимости придумывать имя для переменных. Тут нам помогут макросы.

```
#define LOG_DURATION(message)  \
  LogDuration UNIQ_ID(__LINE__){message};
```

Макрос `UNIQ_ID` имеет следующее устройство (не будем останавливаться на том, как оно получилось):

```
#define UNIQ_ID_IMPL(lineno) _a_local_var_##lineno
#define UNIQ_ID(lineno) UNIQ_ID_IMPL(lineno)
```

Просто поверьте на слово: конструкция `UNIQ_ID(__LINE__)` позволяет объявить уникальный идентификатор в пределах заданного .cpp-файла.

Теперь мы можем замерять время с помощью конструкции

```
{
  LOG_DURATION("Input");
  ...
}
```

Наша программа работает. Ввод длится 102 миллисекунды, обработка запросов – 76, вся программа – 181. При этом мы добились желаемого удобства. Нам больше не нужно придумывать имя для объекта, мы просто пишем `LOG_DURATION` и сообщение и получаем в стандартный поток ошибок длительность работы соответствующего блока `count`. Это действительно удобно.

Допустим, мы захотим померить, сколько работает одна итерация нашего цикла. Мы просто пишем `LOG_DURATION("Iter " + to_string(i))`.

Остался один штрих, чтобы сделать класс `LogDuration` переиспользуемым. Давайте его вынесем в отдельный файл. Заведем заголовочный файл в нашем проекте. Назовем его profile.h и вынесем сюда класс `LogDuration` и необходимые `include`'ы. Это `#include <chrono>`, `using`

`namespace std;`, `using namespace std::chrono;`. И так как мы выводим в `cerr`, нам понадобится ещё `<iostream>`.

Подведём итоги. Мы смогли разработать класс `LogDuration`, который можно использовать для замера времени работы программы и отдельных её блоков, а также сделали специальный макрос `LOG_DURATION`, который нас избавляет от придумывания ненужных идентификаторов и упрощает работу с этим профайлером.

2.2. Эффективное использование потоков ввода/вывода

В этом модуле мы посмотрим, какие проблемы могут возникнуть при использовании потоков ввода/вывода и как эти проблемы решаются.

2.2.1. Буферизация в выходных потоках

Файловый поток `ofstream` не сразу пишет выводимые в него данные в файл, а накапливает их в промежуточном буфере и сбрасывает его в файл, когда он наполнился. Поток вывода `cout` ведет себя точно так же. Манипулятор `endl` не только выводит перевод строки, но и сбрасывает буфер потока в файл.

Сравним производительность потоков. Будем измерять, за какое время выведутся в файл 15000 строк при использовании `endl` и при использовании `'\n'`. Будем использовать профайлер profile.h.

```
int main() {
  {
    LOG_DURATION("endl");
    ofstream out("output.txt");
    for (int i = 0; i < 15000; ++i) {
      out << "London is the capital of Great Britain. "
          << "I am travelling down the river"
          << endl;
    }
  }
  {
    LOG_DURATION("'\n'");
    ofstream out("output2.txt");
```

```
    for (int i = 0; i < 15000; ++i) {
      out << "London is the capital of Great Britain. "
          << "I am travelling down the river"
          << '\n';
    }
  }
}
// endl: 137 ms
// '\n': 16 ms
```

Пусть теперь выводится в 10 раз больше строчек, снова выполним наш код, и увидим поразительную огромную разницу: `endl` – 530 миллисекунд, `'\n'` – 168 миллисекунд.

Тот факт, что `endl` сбрасывает буфер потока, имеет значительное влияние на производительность. Дело в том, что при использовании `endl` мы пишем в файл каждый раз, а при использовании `'\n'` – только когда буфер заполнился.

Таким образом, использование `endl` может приводить к снижению скорости вывода в файл или `cout`.

2.2.2. Когда нужно использовать `endl`, а когда – `'\n'`

Возникает вопрос: зачем использовать `endl`? Он нужен в ситуациях, когда нам не важна скорость вывода всей информации, а важно увидеть последнее выведенное сообщение, как только оно было выведено в выходной поток. Самый простой пример – отладка программы с использованием отладочного вывода в консоль. Использовать буферизованный вывод в данном случае неудобно, потому что вероятна ситуация, когда вы выводите отладочное сообщение, а на следующей команде программа падает. Таким образом, есть вероятность, что это сообщение не будет выведено в консоль, хотя оно может содержать важную информацию об ошибке.

Рассмотрим пример с классом `LogDuration`. Эта программа состоит из двух функций. Тела функций `CouldBeSlowOne` и `CouldBeSlowTwo` скрыты, потому что сейчас они не важны. Одна из них работает медленно, мы не знаем, какая. Чтобы узнать это, мы оборачиваем их в `LOG_DURATION`.

```
int main() {
  {
    LOG_DURATION("One");
```

```
    CouldBeSlowOne();
  }
  {
    LOG_DURATION("Two");
    CouldBeSlowTwo();
  }
}
// One: 9 ms
// Two: 7093 ms
```

Если мы реализуем `LOG_DURATION` с помощью `'\n'`, то оба сообщения выведутся в конце работы программы, ждать придётся долго. Если же в реализации использовать `endl`, то первое сообщение выведется сразу после завершения работы первой функции. Это удобнее, мы можем не ждать выполнения второй функции.

Теперь давайте посмотрим на `TestRunner` из предыдущего курса (курс "Основы разработки на C++: жёлтый пояс"), давайте посмотрим на `class TestRunner` и его метод `RunTest`. `Endl` здесь используется во всех сообщениях. Когда тест выполнился успешно, выводится `OK` и `endl`. И когда случилось падение теста, мы тоже выводим его имя, говорим, что он упал, сообщение из исключения и `endl`.

Если здесь заменить `endl` на символ перевода строки, то мы не будем видеть сразу результат выполнения тестов. Конечно, когда у нас простые юнит-тесты, они все отрабатывают очень быстро, и нам не важно, что эти сообщения буферизируются и потом выводятся целиком. Но если у нас есть юнит-тесты, которые занимают заметное время, то нам лучше сразу же получать информацию о том, прошли они или нет.

Перевод строки можно использовать, когда вам важна скорость вывода. Например, если вы пишете консольное приложение, которое превращает данные из стандартного ввода в какие-то другие данные и выводит их в стандартный вывод.

2.2.3. Связанность потоков

Итак, теперь мы с вами знаем, что замена `endl` на символ перевода строки может ускорять вывод программ на C++. Рассмотрим пример:

```
int main() {
  for (int i = 0; i < 100000; ++i) {
```

```
    int x;
    cin >> x;
    cout << x << endl;
  }
}
```

В цикле 100000 раз из стандартного ввода считывается число и 100000 раз записывается в стандартный вывод. Логично ожидать, что при замене `endl` на `'\n'` программа заработает быстрее. Однако стоит проверить это и сделать измерения.

```
int main() {
  {
    LOG_DURATION("endl");
    for (int i = 0; i < 100000; ++i) {
      int x;
      cin >> x;
      cout << x << endl;
    }
  }
  {
    LOG_DURATION("'\n'");
    for (int i = 0; i < 100000; ++i) {
      int x;
      cin >> x;
      cout << x << endl;
  }
  }
}
// endl: 387 ms
// '\n': 373 ms
```

Реализация с `'\n'` работает немного быстрее, однако мы ожидали, что она отработает в разы быстрее реализации с `endl`. Рассмотрим другой пример:

```
cout << "Enter two integers: ";
for (;;) {
}
//
```

Данный код не выведет на экран ничего, потому что передаваемое сообщение попало в буфер и остаётся там на протяжении бесконечного цикла.

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
// Enter two integers:
```

Такой код выводит `Enter two integers:` . Получается, произошёл сброс внутреннего буфера `cout`. Этот пример нам показывает, что по умолчанию поток `cout` связан с потоком `cin`. Это специально сделано для интерактивных приложений вроде нашего калькулятора. Этим же объясняется неожиданно медленная работа реализации с `'\n'` из прошлого примера. Мы можем разорвать связь между `cout` и `cin`:

```
cin.tie(nullptr);
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
```

Что такое `nullptr`, будет рассказано далее в модуле «Модели памяти».

Ещё больше ускорить работу программ можно, если прописать строку:

```
ios_base::sync_with_stdio(false);
```

Что это за команда? Это выходит за рамки нашего курса. Важно помнить, что вызывать её надо до первой операции ввода/вывода.

2.3. Сложность алгоритмов

Мы научились замерять время работы кода. В частности, сравнивать два алгоритма по времени работы. Уже в предыдущем курсе возникали ситуации, когда один алгоритм заведомо эффективнее другого. Например, мы проходили контейнер `deque`. Чем он был лучше, чем вектор?

2.3.1. Введение

Рассмотрим пример. Попробуем вставлять элементы в начало вектора и в начало `deque`. Вспомним, что вставлять в начало вектора гораздо менее эффективно, чем вставлять в начало `deque`.

```
{
  LOG_DURATION("vector");
  vector<int> v;
  for (int i = 0; i < 100000; ++i) {
    v.insert(begin(v), i);
  }
}
{
  LOG_DURATION("deque");
  deque<int> v;
  for (int i = 0; i < 100000; ++i) {
    v.insert(begin(v), i);
  }
}
// vector: 1661 ms
// deque: 9 ms
```

Рассмотрим ещё один пример. Продемонстрируем, что метод `lower_bound` эффективнее, чем одноимённая функция на примере множества.

```
set<int> numbers;
for (int i = 0; i < 3000000; ++i) {
  numbers.insert(i);
}
const int x = 1000000;

{
  LOG_DURATION("global_lower_bound");
  cout << *lower_bound(begin(numbers), end(numbers), x);
}
{
  LOG_DURATION("lower_bound_method");
  cout << *numbers.lower_bound(x);
}
// global lower_bound: 944 ms
// lower_bound method: 0 ms
```

Рассмотрим третий пример. Сравним функции `lower_bound` и функции `find_if`. Функция `lower_bound` делает бинарный поиск, в то время как функция `find_if` запускает цикл `for`. `lower_bound` заведомо эффективнее, проверим это.

```
const int NUMBER_COUNT = 1000000;
const int NUMBER = 7654321;
const int QUERY_COUNT = 10;

int main() {
  vector<int> v;
  for (int i = 0; i < NUMBER_COUNT; ++i) {
    v.push_back(i * 10);
  }

  {
    LOG_DURATION("lower_bound");
    for (int i = 0; i < QUERY_COUNT; ++i) {
      lower_bound(begin(v), end(v), NUMBER);
    }
  }
  {
    LOG_DURATION("find_if");
    for (int i = 0; i < QUERY_COUNT; ++i) {
      find_if(begin(v), end(v)),
              [NUMBER](int y) { return y >= NUMBER});
    }
  }
  return 0;
}
// lower_bound: 0 ms
// find_if: 123 ms
```

`lower_bound` быстрее, об этом мы и так заранее знали.

Чтобы понять, будет ли эффективен алгоритм, не обязательно его программировать и замерять. Иногда можно заранее знать, какой алгоритм больше подходит.

2.3.2. Оценка сложности

Обозначим за N размер диапазона. Нам известно, что `lower_bound` работает за время порядка $\log N$, а `find_if` в худшем случае работает за N. Не будем учитывать множители при $\log N$ и N, так как при достаточно большом N, N будет больше $\log N$ при любом константном множителе при них. В частности, будем считать, что логарифм двоичный.

Рассмотрим более сложный пример. В цикле пройдёмся по контейнеру `numbers_to_find` и для каждого элемента этого контейнера вызовем `lower_bound`. Поищем это число в другом контейнере. Затем выполним простую операцию, например, `x` умножим на 2.

```
for (int& x : numbers_to_find) {
  lower_bound(begin(v), end(v), x);
  x *= 2;
}
```

Внешний цикл делает столько итераций, каков размер контейнера `numbers_to_find`. Дальше в каждой из итераций мы запускаем `lower_bound`, который работает $\log N$ времени, потом еще одна операция тратится на умножение `x` на 2. Получаем выражение для грубой оценки количества операций.

```
numbers_to_find.size() * (log(v.size()) + 1)
```

Это оценка для худшего случая. В реальности она может не достигаться.

Выполним программу для `NUMBER = 1` и `QUERY_COUNT = 1000000`.

```
// lower_bound: 701 ms
// find_if: 81 ms
```

`lower_bound` выполнял логарифм операций, в то время как `find_if` сходился практически сразу, выполнив всего несколько операций.

Обсудим сложность алгоритмов.

- **Константная** (1): арифметические операции, обращение к элементу вектора;
- **Логарифмическая** $(\log N)$: двоичный поиск (`lower_bound` и пр.), поиск в `set` или `map`;
- **Линейная** (N): цикл `for` (с лёгкими итерациями), алгоритмы `find_if`, `min_element` и пр.;

- $N \log N$: алгоритм `sort` (с лёгкими сравнениями).

Часто используют O-символику. Например, $O(\log N)$ означает, что алгоритм выполняет не больше, чем $\log N$ операций с точностью до константы. «O большое» означает оценку сверху. В принципе она может не достигаться вообще никогда.

Как узнать сложность работы незнакомого алгоритма? Можно вычислить самостоятельно, если известны детали работы алгоритма. Также сложность можно посмотреть в документации.

2.3.3. Практические применения

Скорость алгоритмов стоит сравнивать по следующему несложному правилу:

$$1 < \log N < N < N \log N < N^2 < \dots$$

При сложении большее поглощает меньшее:

$$O(N) + O(\log N) = O(N)$$

$O(5N)$ и $O(8N)$ надо сравнивать измерениями.

Как оценить, подходит ли алгоритм под заданные ограничения? Нужно рассматривать худший случай, брать сложность алгоритма и подставлять туда худшие значения входных данных. Так получается количество операций алгоритма. Далее нужно примерно прикинуть константу, умножить количество операций на неё. В результате получается грубая оценка количества элементарных операций. Далее можно прикинуть время работы алгоритма, если учесть, что на хороших процессорах за секунду можно успеть сделать миллиард простых операций.

Рассмотрим примеры.

Задача «Синонимы»:

Поступает три типа запросов.

- `ADD word1 word2`: добавить пару синонимов (`word1`, `word2`);

- `COUNT word`: узнать количество синонимов слова `word`;
- `CHECK word1 word2`: узнать, являются ли слова `word1` и `word2` синонимами.

В задаче приходит 70000 запросов, слова имеют длину не более 100 символов, решить задачу надо за 1 секунду.

```
int main() {
  int q;
  cin >> q;
  map<string, set<string>> synonyms;  // для каждого слова запоминаем множество его
                                      // синонимов
  for (int i = 0; i < q; ++i) {  // обрабатываем Q запросов (будет Q итераций)
    string operation_code;
    cin >> operation_code;
    if (operation_code == "ADD") {
      string first_word, second_word;
      cin >> first_word >> second_word;  // если слова длины не более чем L, то
                                         // считываем их за L простейших операций
      synonyms[first_word].insert(second_word);  // поиск в словаре конкретного
                                                 // ключа работает за логарифм от
                                                 // размера словаря. Он в худшем
                                                 // случае равен количеству слов,
                                                 // которые уже туда вставили.
                                                 // Сложность равна количеству
                                                 // запросов, умноженная на
                                                 // стоимость сравнения двух строк
                                                 // длины не более L.
                                                 // итоговая сложность: L logQ
      synonyms[second_word].insert(first_word);
      // O(L) + O(L logQ) = O(L logQ)
    } else if (operation_code == "COUNT") {
      string word;
      cin >> word;
      cout << synonyms[word].size() << endl;
      // поиск в словаре занимает L logQ. Итоговая сложность: O(L logQ)
    } else if (operation_code == "CHECK") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (synonyms[first_word].count(second_word) == 1) {
        cout << "YES" << endl;
```

```
      } else {
        cout << "NO" << endl;
      }
      // считываем два слова, ищем их в словаре, потом во множестве
      // O(L logQ)
    }
  }

  // обрабатываем Q запросов за L logQ
  // суммарная сложность программы: O(Q L logQ)
  return 0;
}
```

Задача «Имена и фамилии – 1»:

- `ChangeFirstName(year, first_name)`: добавить факт изменения имени на `first_name` в год `year`;
- `ChangeLastName(year, last_name)`: добавить факт изменения фамилии на `last_name` в год `year`;
- `GetFullName(year)`: получить имя и фамилию по состоянию на конец года `year`;

Были следующие ограничения: 100 запросов, слова длины 10, на выполнение дана одна секунда.

```
string FindNameByYear(const map<int, string>& names, int year) {
  string name;  // изначально имя неизвестно
  for (const auto& item : names) {  // O(Q) итераций
    if (item.first <= year) {  // сравниваем два числа за O(1)
      name = item.second;  // присвоить одну строку в другую стоит столько же
                           // операций, как длина строки: O(L)
    } else {
      // иначе пора остановиться, так как эта запись и все последующие относятся к
      // будущему
      break;
    }
  }
  return name;
  // O(QL) операций
}
```

```
class Person {
  public:
    void ChangeFirstName(int year, const string& first_name) {
      first_names[year] = first_name;  // ищем year как ключ в словаре и сохраняем
                                       // туда строчку
    }  // O(logQ + L)
    void ChangeLastName(int year, const string& last_name) {
      last_names[year] = last_name;
    }  // O(logQ + L)
    string GetFullName(int year) {
      const string first_name = FindNameByYear(first_names, year);
      const string last_name = FindNameByYear(last_names, year);

      if (first_name.empty() && last_name.empty()) {
        return "Incognito";
      } else if (first_name.empty()) {
        return last_name + " with unknown first name";
      } else if (last_name.empty()) {
        return first_name + " with unknown last name";
      } else {
        return first_name + " " + last_name;
      }
      // сложность O(L), которая тратится на сложение строк. Также тут вызывается
      // FindNameByYear. Итоговая сложность: O(L) + O(QL)
    }

  private:
    map<int, string> first_names;
    map<int, string> last_names;
};
// итоговая сложность программы: O(Q * Q * L)
```

В задаче «Имена и фамилии – 4» оптимизирована функция `FindNameByYear`. В ней теперь вызывается метод `upper_bound`, которая работает за логарифм от размера контейнера, то есть за $\log Q$. Тогда суммарная сложность будет $O(\log Q + L)$. Метод `GetFullName` также отработает за $O(\log Q + L)$. Получаем суммарную сложность $O(Q(L + \log Q))$.

2.3.4. Амортизированная сложность

Рассмотрим задачу. К нам приходят события с временными метками. Нужно уметь добавлять событие с временной меткой и находить, сколько событий случилось за последние пять минут. Для решения задачи напишем класс:

```
class EventManager {
  public:
    void Add(uint64_t time);
    int Count(uint64_t time);
}
```

Метод `Add` добавляет события, метод `Count` узнаёт количество событий за последние пять минут. Задачу можно решать с помощью очереди. Подключим `<queue>`.

```
private:
  queue<uint64_t> events;
```

В очередь мы будем добавлять новые события и удалять старые:

```
void Add(uint64_t time) {
  events.push(time);
}
int Count(uint64_t time) {
  return events.size();
}
```

Добавим в `private` метод `Adjust`, который принимает новые временные метки и удаляет старые.

```
void Adjust(uint64_t time) {
  while (!events.empty() && events.front() <= time - 300) {
    events.pop();
  }
}
```

Удалять старые события нужно и при добавлении в очередь, и при вычислении количества событий в ней.

```
void Add(uint64_t time) {
  Adjust(time);
  events.push(time);
}
```

```
int Count(uint64_t time) {
  Adjust(time);
  return events.size();
}
```

Попробуем оценить сложность каждого запроса.

```
class EventManager {
  public:
    void Add(uint64_t time) {  // O(Q)
      Adjust(time);
      events.push(time);  // O(1)
    }
    int Count(uint64_t time) {  // O(Q)
      Adjust(time);
      return events.size();  // O(1)
    }
  private:
    queue<uint64_t> events;
    void Adjust(uint64_t time) {  // O(Q)
      while (!events.empty() && events.front() <= time - 300) {  // в худшем
                                                                 // случае O(Q)
        events.pop();  // O(1)
      }
    }
}
```

Получается, суммарная сложность будет квадратичной? Дело в том, что это верхняя оценка и она достигаться не будет. Каждый конкретный вызов `Adjust`, `Add` или `Count` может работать долго. Суммарно все эти события будут работать за $O(Q)$, а не за $O(Q * Q)$. В таком случае говорят, что сложность `Add`, `Count`, `Adjust` не $O(Q)$, а amortized $O(1)$, то есть в среднем каждый метод работает как $O(1)$.