Профилируем и ускоряем. Конспект темы

Зачем нужна профилировка

Профилирование — выборочное измерение времени выполнения участков кода. Используется, чтобы отследить проблемы в коде.

Профилировщик — инструмент, показывающий, сколько времени работает функция.

Популярные профилировщики:
- Intel®VTune™ Profiler (Windows, Linux).
- Microsoft Visual Studio (Windows).
- Консольные инструменты (Linux). В ряде UNIX-подобных операционных систем есть инструменты strace, ltrace и gprof.

Измеряем время

Время выполнения операций измеряется так:

- запомнить начальный момент (до операции),
- запомнить конечный момент (после операции),
- вычесть из значения времени в конечный момент значение времени в начальный, получится продолжительность операции.

Для этого используют `<chrono>` — библиотеку часов для различных целей. Часы `chrono::steady_clock` определяют момент времени. Разность моментов времени — это продолжительность.

Когда пишете свой профилировщик, результат нужно выводить не в `cout`, а в `cerr`. `cerr` можно отделить от `cout`, перенаправив его в отдельный лог-файл с отладочной информацией.

`chrono::duration` — универсальная продолжительность. Это шаблонный класс, и его шаблонные параметры определяют:

- числовой тип, который будет представлять количество тиков,
- размер одного тика в виде дробного количества секунд.

Функция ожидания `this_thread::sleep_for` из библиотеки `<thread>` останавливает программу на заданное время.

Чтобы привести продолжительность в понятный вид — например, в миллисекунды — используют функцию `chrono::duration_cast`.

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std;
using namespace chrono;

int main() {
```

```
    cout << "Ожидание 5s..."s << endl;
    const auto start_time = steady_clock::now();

    // операция - ожидание 5 секунд
    this_thread::sleep_for(seconds(5));
    const auto end_time = steady_clock::now();

    const auto dur = end_time - start_time;
    cerr << "Продолжительность сна: "s << duration_cast<milliseconds>(dur).count() << " ms"s << endl;

    cout << "Ожидание завершено"s << endl;
}
```

Этот код можно упростить — подключите пространство имён `std::literals`. Оно позволяет использовать различные единицы измерения времени — от наносекунд (ns) до часов (h):

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std;
using namespace chrono;
// хотите немного магии? тогда используйте namespace literals
using namespace literals;

int main() {
    cout << "Ожидание 5s..."s << endl;
    const auto start_time = steady_clock::now();

    // операция - ожидание 5 секунд
    this_thread::sleep_for(5s);
    const auto end_time = steady_clock::now();

    const auto dur = end_time - start_time;
    cerr << "Продолжительность сна: "s << chrono::duration_cast<chrono::milliseconds>(dur).count() << " ms"s << endl;

    cout << "Ожидание завершено"s << endl;
}
```

Компилятор понял, что 5s — это пять секунд.

Упрощаем логирование

Конструктор и деструктор класса позволяет выполнять парные действия без риска ошибок.

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std;
using namespace chrono;
using namespace literals;

class LogDuration {
public:
    LogDuration() {
    }

    ~LogDuration() {
        const auto end_time = steady_clock::now();
        const auto dur = end_time - start_time_;
```

```
        cerr << duration_cast<milliseconds>(dur).count() << " ms"s << endl;
    }

private:
    const steady_clock::time_point start_time_ = steady_clock::now();
};

int main() {
    cout << "Ожидание 5s..."s << endl;

    {
        LogDuration sleep_guard;
        // операция - ожидание 5 секунд
        this_thread::sleep_for(5s);
    }

    cout << "Ожидание завершено"s << endl;
}
```

Призываем макросы

Для упрощения профилировки использована **RAII** (от англ. Resource Acquisition is Initialization, «получение ресурса есть инициализация»). Это идиома, основанная на том, что получение объекта связывается с созданием объекта, а освобождение — с уничтожением объекта.

Вы можете настроить специальный режим, в котором компилятор будет запускать только препроцессор. В GCC для этого укажите флаг `gcc -E in.cpp -o in.i`.

Препроцессор позволяет скрыть ненужную информацию — название переменной, которая позволила создать объект `LogDuration`. Определите макрос:

```
#define LOG_DURATION(x) LogDuration UNIQUE_VAR_NAME_PROFILE(x)
```

Затем разработайте макрос `UNIQUE_VAR_NAME_PROFILE`, который будет выдавать уникальное имя переменной. Для профилировщика это не особо важно, но удобно.

Понадобится макрос `__LINE__`. Препроцессор заменяет его на номер строки, в которой использован этот макрос. Также

`##` — оператор слияния лексем. Лексема — одно «слово» в коде с точки зрения препроцессора. Например, служебное слово, литерал, имя переменной, спецсимвол.

```
#define PROFILE_CONCAT_INTERNAL(X, Y) X ## Y
#define PROFILE_CONCAT(X, Y) PROFILE_CONCAT_INTERNAL(X, Y)
#define UNIQUE_VAR_NAME_PROFILE PROFILE_CONCAT(profile_guard_, __LINE__)

int main() {
    int UNIQUE_VAR_NAME_PROFILE;
}
```

Препроцессор выдаст:

```
#line 1 "main.cpp"
```

```
int main() {
    int profile_guard_6;
}
```

Измеряем и ускоряем

Чтобы ускорить работу программы, начните с улучшения её самой медленной части.

Вставлять элементы в конец вектора эффективнее, чем в начало или середину.

Итераторы `rbegin` и `rend` работают так же, как `begin` и `end`, но проходят контейнер в обратном направлении:

```
vector<int> ReverseVector2(const vector<int>& source_vector) {
    vector<int> res;

    // проходим source_vector задом наперёд,
    // используя обратный итератор
    for (auto iterator = source_vector.rbegin(); iterator != source_vector.rend(); ++iterator) {
        res.push_back(*iterator);
    }

    return res;
}
```

При случайном заполнении вектора требуется только один бит информации, а `rand()` выдаёт минимум 15 случайных бит. Используйте все 15, это увеличит производительность программы:

```
// <algorithm> нужен для функции min
#include <algorithm>
...
void FillRandom2(vector<int>& v, int n) {
    for (int i = 0; i < n; i += 15) {
        int number = rand();

        // мы можем заполнить 15 элементов вектора,
        // но не более, чем нам осталось до конца:
        int count = min(15, n - i);

        for (int j = 0; j < count; ++j)
            // таким образом, получим j-й бит числа.
            // операцию побитового сдвига вы уже видели в этой программе
            // на этот раз двигаем вправо, чтобы нужный бит оказался самым последним
            v.push_back((number >> j) % 2);
    }
}
...
```

Чтобы ещё ускорить работу программы, зарезервируйте место в векторе. Используйте метод `reserve` и укажите количество элементов, которые будут в векторе в итоге:

```
vector<int> ReverseVector3(const vector<int>& source_vector) {
    vector<int> res;
    res.reserve(source_vector.size());

    // проходим source_vector задом наперёд,
    // используя обратный итератор
```

```
    for (auto iterator = source_vector.rbegin(); iterator != source_vector.rend(); ++iterator) {
        res.push_back(*iterator);
    }

    return res;
}
```

Два важных правила оптимизации

1. Избегайте преждевременной оптимизации.

Главные достоинства программы — надёжность, понятность и скорость.

Усложняя код ради скорости, вы делаете его менее понятным и надёжным. В сложных программах проще допустить ошибку.

Понять, где правда нужна оптимизация, поможет профилировка.

2. Измеряйте.

Только измерения помогут выявить места, где программа проводит больше всего времени. Их расположение часто противоречит интуиции.

Вглубь процессора

Кэш процессора — специальная быстрая память, с которой процессор оперирует напрямую.

Когда читаете один символ из объекта `string`, все соседние загружаются в кэш вместе с ним, и доступ к ним происходит быстрее, чем к далёким символам и элементам других строк.

Процессоры имеют несколько кэш-линий, обычно размером 64 байта. Каждая кэш-линия хранит участок значений, к которым недавно обращалась программа.

Если обработаете все близкие значения сразу, процессору не придётся перезаписывать кэш-линии и обращаться к медленной памяти RAM (от англ. random access memory — «память с произвольным доступом») каждый раз при чтении символа из строки.

В критических местах программы избегайте `if`. `if` вставляет специальную процессорную инструкцию — **условный переход**. Процессор от этого впадает в ступор. Он не может работать эффективно, когда не знает, что ему делать дальше. Используйте лучше тернарный оператор.

Другие способы оптимизации:

- Специальные функции, которые позволяют производить одну и ту же операцию над несколькими числами одновременно. Эта технология называется Single Instruction Multiple Data.
- Разворачивание цикла. `for` тоже содержит условный переход в начале каждой итерации. Для оптимизации можно обработать в одной итерации сразу несколько элементов — например 16. Так вы сократите количество итераций и условных переходов в 16 раз. Это спорный подход, так как оптимизаторы иногда делают разворачивание цикла автоматически, а код становится менее понятным.
- Распараллеливание. Можно поручить нескольким потокам выполнения обрабатывать разные участки вектора.