

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 13

??..12.2019

Оптимизация программ

- ▶ оптимальные алгоритмы и структуры данных;
- ▶ отсутствие лишних копирований;
- ▶ эффективный параллелизм;
- ▶ компиляторы и оптимизаторы;
- ▶ ...

«Premature optimization is the root of all evil» (с)

Оптимизация программ на C++

Особенности:

- ▶ Порядок инструкций в коде \neq порядок их исполнения;
- ▶ Потoki выполнения, средства синхронизации, `std::atomic`.
- ▶ STL – универсальная библиотека, часто ручную закодированный алгоритм работает быстрее.

Пример 1

Как оптимизировать код?

```
for (auto& str: cont) {  
    std::string s;  
    ParseInput(str, s);  
    Process(s);  
}
```

Пример 1

Как оптимизировать код?

```
for (auto& str: cont) {  
    std::string s;  
    ParseInput(str, s);  
    Process(s);  
}
```

```
std::string s;  
for (auto& str: cont) {  
    s.clear();  
    ParseInput(str, s);  
    Process(s);  
}
```

Пример 2

Как оптимизировать код?

```
char s[] = " ..... ";  
for (size_t i = 0; i < strlen(s); ++i) {  
    if (s[i] == ' ') s[i] = '*';  
}
```

Пример 2

Как оптимизировать код?

```
char s[] = " ..... ";  
for (size_t i = 0; i < strlen(s); ++i) {  
    if (s[i] == ' ') s[i] = '*';  
}
```

A:

```
for (size_t i = 0, len = strlen(s); i < len; ++i) {  
    if (s[i] == ' ') s[i] = '*';  
}
```

B:

```
for (int i = (int)strlen(s) - 1; i >= 0; --i) {  
    if (s[i] == ' ') s[i] = '*';  
}
```

Пример 3

Как оптимизировать код?

```
for (size_t i = 0; i < v.size(); ++i) {  
    double x = v[i].x, y = v[i].y;  
    v[i].x = cos(theta) * x - sin(theta) * y;  
    v[i].y = sin(theta) * x + cos(theta) * y;  
}
```


Пример 3

Как оптимизировать код?

```
for (size_t i = 0; i < v.size(); ++i) {  
    double x = v[i].x, y = v[i].y;  
    v[i].x = cos(theta) * x - sin(theta) * y;  
    v[i].y = sin(theta) * x + cos(theta) * y;  
}
```

Выносим инварианты:

```
cos_theta = cos(theta);  
sin_theta = sin(theta);  
for (size_t i = 0; i < v.size(); ++i) {  
    double x = v[i].x, y = v[i].y;  
    v[i].x = cos_theta * x - sin_theta * y;  
    v[i].y = sin_theta * x + cos_theta * y;  
}
```

Пример 4

Как оптимизировать код?

```
class A {  
    private:  
        int x, y;  
    public:  
        // ...  
        std::string get_name () const {  
            return "A";  
        }  
};
```

Пример 4

Как оптимизировать код?

```
class A {  
    private:  
        int x, y;  
    public:  
        // ...  
        std::string get_name () const {  
            return "A";  
        }  
};
```

static! Статические функции не должны вычислять неявный указатель this.

Пример 5

Как оптимизировать код?

```
y = a * x * x * x + b * x * x + c * x + d;
```

Пример 5

Как оптимизировать код?

```
y = a * x * x * x + b * x * x + c * x + d;
```

```
y = (((a * x + b) * x) + c) * x + d;
```

(схема Горнера)

Пример 6

Как оптимизировать код?

```
int f() {  
    // ...  
    seconds = 24 * days * 60 * 60;  
    // ...  
}
```

Пример 6

Как оптимизировать код?

```
int f() {  
    // ...  
    seconds = 24 * days * 60 * 60;  
    // ...  
}
```

```
constexpr float SecondsInDay = 24 * 60 * 60;  
int f() {  
    // ...  
    seconds = days * SecondsInDay;  
    // ...  
}
```

Пример 7

Как оптимизировать код?

```
y = x * 9;
```


Пример 7

Как оптимизировать код?

```
y = x * 9;
```

```
y = (x << 3) + x;
```

Эвристическое правило 90/10, bottleneck

90% времени работы программа тратит на исполнение 10% кода.

В программе есть «горячие» точки, где наиболее всего уместна оптимизация.

Если оптимизацией пренебречь в большом проекте, то впоследствии сложно будет что либо ускорить относительно всего дизайна!

Закон Амдала

$$S_T = \frac{1}{(1 - P) + \frac{P}{S_P}}$$

- ▶ S_T — ускорение в целом (улучшение времени выполнения программы в целом в результате оптимизации)
- ▶ P — доля под оптимизацию (сколько времени от общего оптимизируем)
- ▶ S_P — ускорение в оптимизированной части (насколько ускоряем в этой части)

Закон Амдала

Задача из теста: *Время выполнения программы занимает 50с. Из них почти 40с программа тратит на выполнение нескольких вызовов одной функции f . Существует способ оптимизации функции f , который сделает ее на 40% быстрее. Насколько ускорится выполнение программы в целом?*

Закон Амдала

Задача из теста: *Время выполнения программы занимает 50с. Из них почти 40с программа тратит на выполнение нескольких вызовов одной функции f . Существует способ оптимизации функции f , который сделает ее на 40% быстрее. Насколько ускорится выполнение программы в целом?*

$$S_T = \frac{1}{(1 - P) + \frac{P}{S_P}}$$

- ▶ Доля P оптимизированного общего времени: 0.8
- ▶ Показатель улучшения S_P в оптимизированной части: 1.4
- ▶

$$\frac{1}{(1 - 0.8) + \frac{0.8}{1.4}} = 35/27 = 1.296$$

Profiler

- ▶ Программа, которая генерирует статистические данные о том, как и на что программа тратит свое время работы.
- ▶ Может выдать частоты выполнения каждой инструкции или функции и суммарное время выполнения каждой функции.
- ▶ Влияние времени работы самого профайлера на время выполнения работы всей программы не велико и не имеет важного значения.
- ▶ Наиболее важное: увидеть самые проблемные места в коде
- ▶ Профилирование отладочной версии программы дает почти такой же результат, но нагляднее, поскольку ничего не скрывает.

Профайлер не в состоянии посоветовать более эффективный алгоритм решения задачи.

Строки в C++

Класс `std::string`: тысячи возможностей делают его эффективную реализацию невозможной. Существует множество компиляторов, в которых реализация строк не соответствует стандарту.

- ▶ используют динамическое выделение памяти
- ▶ ведут себя как значения в выражениях
- ▶ требуют много копирований

Идиома COW

Copy On Write: идиома для объектов с дорогим копированием.

- ▶ Одна динамическая память может использоваться несколькими значениями.
- ▶ Любая операция, которая изменяет значение строки, сначала убеждается что существует только один указатель на эту память.
- ▶ Если нет — выделяем новую и копируем.
- ▶ C++11 запрещает cow-строки. Можно реализовать с использованием `shared_ptr`.

Строки в C++

- ▶ C-строки.
- ▶ `std::string_view`: содержит указатель, которым не владеет, на строковые данные и их длину. Подстрока и отсечение суффиксов/префиксов более эффективны, чем в `std::string`. Полезен, когда хочется избежать ненужных копирований.
- ▶ Можно сделать другие реализации, но:
 - ▶ они должны давать профит везде, не только в узком кейсе;
 - ▶ замена требует большой работы;
 - ▶ найти среди множества альтернатив весьма трудно.

Оптимизация алгоритмов

- ▶ **Предвычисления.** Тут помогает `constexpr` и вычисления компилятором, подстановка шаблонных параметров. Компилятор сам оптимизирует.
- ▶ **Отложенные вычисления.** Как COW, когда копирование откладывается до тех пор, пока кто-то не захочет изменить данные.
- ▶ **Пакетирование.** Сбор элементов и их совместная обработка. Пример — буферизованный вывод.
- ▶ **Кеширование.** Повторное использование уже вычисленных результатов. Пример — не вычисляем длину строки каждый раз при вызове `size()`.
- ▶ **Специализация.** Не делаем те вычисления, которые в конкретном частном случае могут быть не нужны. Пример — `std::swap` использует семантику перемещения, если аргументы перемещаемы.

Оптимизация алгоритмов

- ▶ **Группировка.** Сокращение числа итераций повторов. Пример — запрос больших данных от ОС, чтобы вызывать функций ядра меньшее количество раз.
- ▶ **Подсказки.** Пример — необязательный аргумент `hint` у `std::map`, — итератор, с которого стоит начать поиск места для вставки.
- ▶ **Оптимизация ожидаемого пути.** Если есть много `else if`, то первыми лучше сделать те, которые встречаются чаще.
- ▶ **Двойная проверка.** Для исключения некоторых случаев используется недорогая проверка, а затем при необходимости используется дорогостоящая. Например, сравнение `std::string` можно проводить, сравнивая сначала длины, а потом уже посимвольно.
- ▶ **Хеширование.** Превращение объекта в целочисленное значение (хеш). Если хеши разные, то и объекты разные.