Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка
Факультет комп’ютерних наук та кібернетики

Д.А. Клюшин

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

“Оптимізація обчислень на C++”

для студентів факультету комп’ютерних наук та кібернетики

Київ 2025

Методичні рекомендації “Оптимізація обчислень на C++” для студентів факультету комп’ютерних наук та кібернетики / Д.А. Клюшин. — К.: Київський національний університет імені Тараса Шевченка, 2025. — 36 с.

Укладач:

Д.А. Клюшин, доктор фізико-математичних наук, професор, професор кафедри обчислювальної математики факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка

Рецензенти:

Карнаух Т.О., кандидат фізико-математичних наук, доцент кафедри теоретичної кібернетики факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка

Омельчук Л.Л., кандидат фізико-математичних наук, доцент кафедри теорії та технології програмування факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка

Ухвалено на засіданні науково-методичної комісії факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, протокол № 6 від 17 лютого 2025 року.

Рекомендовано до друку на засіданні вченої ради факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, протокол № 10 від 19 лютого 2025 року.

ЗМІСТ

ВСТУП

Епоха індивідуальних програмістів залишилася у минулому. В наш час масштабні програмні проєкти реалізують великі колективи спеціалістів, члени яких часто перебувають в різних частинах країни, або навіть світу. Для успішної спільної праці багатьох програмістів критично важливі ясний план роботи, її грамотна організація та чітка координація між учасниками. Ігнорування цих аспектів може привести до втрати часу і фінансів, вкладених у проєкт. Сучасному процесу розробки програмного забезпечення допомагають спеціальні технології проєктування і програмування. Отже, перш ніж переходити до вивчення синтаксичних конструкцій, слід уяснити основні принципи проєктування програм, оскільки вони сильно впливають на структуру самої мови програмування. Для цього ми розглядатимемо комп'ютерні програми як засіб для розв'язування задач. Розв’язок — це алгоритм і структура даних. В цьому рівнянні немає головного і допоміжного елемента. І алгоритм, і структура даних однаково важливі, адже саме алгоритм розв'язує задачу, а його швидкість і ефективність використання пам'яті залежить від правильно обраної структури даних.

Слід пам'ятати, що робота над програмним забезпеченням не закінчується в момент завершення його розробки. У кожного програмного забезпечення є свій *життєвий цикл* — процес, який починається з постановки задачі, продовжується написанням та відлагоджуванням програми, а потім триває певний період, протягом якого програма модифікується та оптимізується.

За визначенням програма має правильно вирішувати поставлене завдання. Однак те саме завдання можна вирішувати по-різному. Отже, існує безліч варіантів програми, серед яких необхідно вибрати оптимальний. Зазвичай реалізується дещо інший варіант: створюючи програму, автор вибирає різні розв’язки підзадач, які ведуть до мети, дотримуючись певних критеріїв. Для цього він вибудовує ланцюжок операцій, кожна з яких має свої особливості. Можливо, серед них виявиться “слабка ланка”, яка гальмуватиме роботу всієї програми.

Виявити це “вузьке місце” та усунути проблему — ось у цьому і полягає мета оптимізації обчислень.

Як це пов’язано з об’єктно-орієнтованим програмуванням? Основним інструментом об’єктно-орієнтованого програмування є класи, які об’єднують дані та алгоритми в єдине ціле — об’єкти, а всередині класів застосовується модульна парадигма, основою якого є функції, які реалізують алгоритми, що обробляють інкапсульовані дані. Таким чином, модульна парадигма не меш важлива, ніж об’єктно-орієнтована, адже саме її оптимальне застосування дозволяє розробляти ефективні об’єктно-орієнтовані програми.

Оптимізація – багатокритеріальна задача. З одного боку, програма має працювати швидко, з іншого — не повинна витрачати багато пам’яті. Окрім того, створення програми не повинно вимагати надто багато робочого часу. Якщо оптимізація змушує програміста довше працювати над розв'язанням задачі, це збільшує вартість програми та погіршує її конкурентоспроможність.

Сучасні компілятори мають можливості оптимізації коду. Для цього достатньо зазначити відповідні опції.

```
#pragma optimize("", on)
// Код
#pragma optimize("", off)
```

Проте якими б досконалими не були компілятори, головне в програмуванні — ефективний стиль. Компілятор проводить оптимізацію формально і лише програміст знає, яке завдання він вирішує.

Ця методична розробка призначена для студентів факультету комп’ютерних наук та кібернетики, щоб допомогти їм покращити стиль написання програм на C++. Усі експерименти були проведені на комп’ютері з процесором Intel Core i3 за допомогою компілятора VC 2022. Усі програми наводяться в завершеному вигляді, щоб студенти могли їх скопіювати, скомпілювати і провести експерименти самостійно.

1. ОПТИМІЗАЦІЯ ШВИДКОДІЇ

Де розташоване “вузьке місце” програми? Яка операція надто затратна? Не вирішивши цих питань, програміст ризикує виправляти і так досить швидкий код, не звернувши уваги на справді неефективний фрагмент. Як правило, для відповіді на ці питання застосовують програму-профілювальник.

Визначивши “слабку ланку”, наприклад довгий цикл, що містить неефективний код, програміст повинен вибрати одне з двох можливих рішень: або модифікувати цикл (розгорнувши його, винісши інваріанти і так далі), або переписати весь алгоритм. Не слід забувати, що інколи програма працює довго, не тому, що написана погано, а тому, що обрано неефективний алгоритм розв'язання задачі.

1.1. Вибір алгоритму

Коли потрібно виконувати оптимізацію? Простіше відповісти, коли її не слід виконувати: якщо програма ще не завершена, не протестована та не налагоджена. Є ще одна відповідь: коли вона і так працює швидко. “Краще — ворог хорошого!” Прагнення довести програму до досконалості також потребує часу. Вишукуючи тонкощі та видаляючи порошинки, програміст витрачає додатковий робочий час і, отже, підвищує вартість програми.

Виконуючи оптимізацію, необхідно враховувати призначення програми. Якщо це програма, яка запускається раз на тиждень швидкість її роботи не є критично важливою величиною. У цій ситуації найважливіше її зручність та вимоги до ресурсів. Якщо ж програма працює практично неперервно, її швидкість має вкрай важливе значення.

Оптимізацію слід проводити не на модельних, а на реальних прикладах, взятих із практики. Наприклад, розглянемо поширене твердження, що “швидке сортування” є найефективнішим алгоритмом упорядкування даних. При цьому, зазвичай, забувають зазначити, що оцінка швидкості цього алгоритму є асимптотичною, тобто його перевага проявляється лише на дуже великих наборах даних. Просте тестування показує, що серед відомих методів сортування (метод

бульбашки, сортування вставками, пірамідальне сортування, сортування обмінами, порозрядне сортування, сортування Шелла та швидке сортування) на масивах помірного розміру (5–10 тисяч елементів) повним аутсайдером є лише алгоритм бульбашки. Однак логічна (не обчислювальна!) складність цих алгоритмів неоднакова: один довший — інший коротший, один простий — інший заплутаний. Вочевидь, якщо час роботи алгоритмів приблизно однаковий, слід вибирати найпростіший і зрозумілий алгоритм. Час, витрачений на налагодження такої програми, буде набагато меншим.

Цілком інша ситуація складається, якщо доводиться сортувати величезні масиви інформації, що складаються з мільйонів елементів. Зрозуміло, у цьому разі слід віддавати перевагу найбільш швидкому алгоритму, який має найкращу асимптотичну оцінку швидкості.

Крім алгоритму, на ефективність програми впливає вибір структури даних. Наприклад, якщо в ході виконання алгоритму необхідно часто вставляти та видаляти елементи, розташовані у довільних місцях, найбільш прийнятною структурою є зв'язаний список. Якщо алгоритм передбачає пошук у суміжних комірках за допомогою прямого доступу, краще записувати дані в масив.

1.2. Ясність та простота коду

Серед основних критеріїв якості програми часто називають її наочність чи читабельність. Це означає, що будь-яка людина, яка знає мову програмування, може розібратися у структурі та порядку виконання такої програми. Однак у цієї властивості є й інший аспект. Ясні та прості вирази полегшують не лише читання, а й компіляцію програми. Складні та заплутані конструкції набагато гірше піддаються оптимізації, ніж прості та зрозумілі.

Одним із основних способів підвищення наочності програми є модульний підхід. Він дозволяє розбити алгоритм на чітко окреслені вузькоспеціалізовані підзавдання, які можна “загорнути” в клас.

1.3. Швидкість виконання операцій

Слід пам'ятати, що в мові C++ є як швидкі, так і повільні операції. Навіть якщо виконання цих операцій займає лише кілька мікросекунд, у великій програмі їх може бути занадто багато. Крім того, як правило, вони розташовуються всередині циклів і виконуються багаторазово. Навіть найшвидша операція може бути причиною гальмування, якщо вона виконується надто часто.

Розглянемо тепер деякі прийоми, які, як вважається, дозволяють оптимізувати програму.

1.4. Спрощення виразів

Як відомо, ділення виконується набагато повільніше, ніж множення. У свою чергу операція множення повільніша від складання або віднімання. Проте програмісти часто ігнорують цей факт. Ось кілька прикладів, у яких спрощення виразів може підвищити швидкість виконання програми. Звісно, якщо операція виконується лише декілька разів, ефект від її оптимізації буде непомітним, але якщо операція є масовою і виконується `N` разів (в нашому випадку для експерименту ми вибрали один мільйон разів), то різниця стає помітною. Для обчислення тривалості виконання операцій ми обраховуємо кількість тактів процесора, протягом яких був виконаний відповідний фрагмент програми. Кількість тактів, що пройшли з моменту запуску програми, повертає функція `clock()`, яка описана в заголовочному файлі `ctime`. Для усереднення ефекту втручання побічних процесів, запустимо наші функції `M` разів (в нашому випадку для експерименту ми вибрали одну тисячу разів) та усереднимо результат.

Для уникнення дублювання коду, який обчислює середні значення, запишемо у файл `test` такий код.

```
using T = void (Test::*)();
void test(T f, Test& a, long int m, const char* s)
{
    clock_t start, finish;
    double sum = 0.0;
    for (long int p = 0; p < m; p++)
    {
        start = clock();
        (a.*f)();
        finish = clock();
```

```
        sum = sum + (finish - start);
    }
    sum = sum / m;
    cout << s << (int)sum << endl;
}
```

Об'єднаємо наші тестові функції в клас `Test`, який містить чотири функції-члени, які здійснюють відповідні обчислення. Заодно перевіримо, чи впливає на швидкодію різниця між префіксною та постфіксною формою інкрементації лічильника циклу. Для локалізації виводу на консоль використаємо функцію

```
system("chcp 1251 > 0");
```

яка описана в заголовку `locale`.

Крім того, для того щоб оцінити, наскільки корисним для швидкодії є описувані прийоми, кожну програму виконаємо два рази: без директиви `#pragma optimize("", on)` і з цією директивою. Якщо компілятор виконує ту саму оптимізацію, що й запропонований прийом, то результати будуть однаковими. Якщо компілятор виконує інші види оптимізації, але саме таку, як описано в тексті, не виконує, то ми отримаємо додатковий виграш в часі.

Оптимізація математичних виразів

```
#include <iostream>
#include <ctime>
#include <locale>
using namespace std;

struct Test
{
    static const long N = 10000000L;
    static const int  M = 1000;

    double a, b, c, r;
    Test(double x, double y, double z) :a(x), b(y), c(z) {}

    void timetest1()
    {
        //Два множення і одне віднімання
        int r;
        for (long int p = 0; p < N; p++) r = a * b - a * c;
    }

    void timetest2()
    {
        //Одне множення і одне віднімання"
        int r;
        for (long int p = 0; p < N; p++) r = a * (b - c);
    }
```

```
    void timetest3()
    {
        // Два ділення і одне віднімання
        double q;
        for (long int p = 0; p < N; p++) q = b / a - c / a;
    }

    void timetest4()
    {
        // Одне ділення і одне віднімання
        double q;
        for (long int p = 0; p < N; p++) q = (b - c) / a;
    }

    void timetest5()
    {
        //Два множення і одне віднімання
        int r;
        for (long int p = 0; p < N; ++p) r = a * b - a * c;
    }

    void timetest6()
    {
        //Одне множення і одне віднімання"
        int r;
        for (long int p = 0; p < N; ++p) r = a * (b - c);
    }

    void timetest7()
    {
        // Два ділення і одне віднімання
        double q;
        for (long int p = 0; p < N; ++p) q = b / a - c / a;
    }

    void timetest8()
    {
        // Одне ділення і одне віднімання
        double q;
        for (long int p = 0; p < N; ++p) q = (b - c) / a;
    }

}; // End Test

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a(2., 4., 3.);

    test(&Test::timetest1, a, a.M, "Два множення і одне віднімання (такти)  p++: ");
    test(&Test::timetest2, a, a.M, "Одне множення і одне віднімання (такти) p++: ");
    test(&Test::timetest3, a, a.M, "Два ділення і одне віднімання (такти)   p++: ");
    test(&Test::timetest4, a, a.M, "Одне ділення і одне віднімання (такти)  p++: ");
    test(&Test::timetest5, a, a.M, "Два множення і одне віднімання (такти)  ++p: ");
    test(&Test::timetest6, a, a.M, "Одне множення і одне віднімання (такти) ++p: ");
```

```
    test(&Test::timetest7, a, a.M, "Два ділення і одне віднімання (такти)   ++p: ");
    test(&Test::timetest8, a, a.M, "Одне ділення і одне віднімання (такти)  ++p: ");

    return 0;
}
```

На екран виводяться такі результати.

Без директиви `#pragma optimize("", on)`

```
Два множення і одне віднімання (такти)  p++: 31
Одне множення і одне віднімання (такти) p++: 27
Два ділення і одне віднімання (такти)   p++: 37
Одне ділення і одне віднімання (такти)  p++: 27
Два множення і одне віднімання (такти)  ++p: 30
Одне множення і одне віднімання (такти) ++p: 26
Два ділення і одне віднімання (такти)   ++p: 36
Одне ділення і одне віднімання (такти)  ++p: 27
```

З директивою `#pragma optimize("", on)`

```
Два множення і одне віднімання (такти)  p++: 26
Одне множення і одне віднімання (такти) p++: 24
Два ділення і одне віднімання (такти)   p++: 35
Одне ділення і одне віднімання (такти)  p++: 24
Два множення і одне віднімання (такти)  ++p: 27
Одне множення і одне віднімання (такти) ++p: 24
Два ділення і одне віднімання (такти)   ++p: 35
Одне ділення і одне віднімання (такти)  ++p: 24
```

Якщо включена оптимізація, компілятор дійсно помітно зменшує час виконання програми, але оптимізація математичних виразів зменшує час виконання програми ще більше. Якщо б це було не так, то після оптимізації компілятора усі перетворення виразів мали б призводити до однакових результатів. Таким чином, оптимізація математичних виразів є корисною. Ці два методи доповнюють один одного.

Як бачимо, суттєвої різниці між префіксною та постфіксною формою інкрементації лічильника циклу немає, а от форма виразу може мати відчутний вплив.

Слід зауважити, що при різних запусках програми на екран виводяться дещо різні, але схожі результати, оскільки у кількість тактів, які були витрачені на виконання фрагмента програми, можуть включатися такти, які були витрачені процесором на інші операції, не пов’язані з програмою. Тому для більш ґрунтовного висновку слід провести достатньо велику кількість запусків і обчислити середні показники. Дослідження точних залежностей тривалості

виконання операцій від обраних способів обчислення не є метою цієї роботи. Наша мета — продемонструвати, які прийоми зменшують час виконання програми при масовому виконанні цих операцій, а які на нього не впливають.

Оскільки теоретично двійкові операції повинні виконуватися швидше, ніж десяткові чи шістнадцяткові, може здатися, що заміна множення або ділення цілого числа на ступінь двійки зсувом розрядів праворуч або ліворуч відповідно призведе до виграшу часу. Однак сучасні компілятори роблять цю заміну автоматично, тому для степені двійки жодного ефекту досягти неможливо. Порозрядний зсув навіть трохи пригальмовує програму, оскільки для подання множників з урахуванням двійкового вигляду треба виконати додаткові операції.

Розрядний зсув

```
#include <ctime>
#include <iostream>
#include <locale>

using namespace std;

struct Test
{
    static const long N = 10000000L;
    static const int  M = 1000;

    int n, m;

    Test(int a, int b) : n(a), m(b) {}

    void timetest1() // Звичайне множення
    {
        int var;
        for (long int p = 0; p < N; p++)var = n * m;
    }

    void timetest2() // Множення на парні числа шляхом зсуву
    {
        int var;
        for (long int p = 0; p < N; p++)var = n << (m - 1);
    }

    void timetest3() // Множення на непарні числа шляхом
    {
        int var;
        for (long int p = 0; p < N; p++) { var = (n << (m - 2)) + n; }
    }

}; // End Test

#include "test"
```

```
int main()
{
    system("chcp 1251 > 0");

    Test a(5, 2);

    test(&Test::timetest1, a, a.M, "Множення на 2 (звичайне)  : ");
    test(&Test::timetest2, a, a.M, "Множення на 2 (зсув бітів): ");

    Test b(5, 3);

    test(&Test::timetest1, b, b.M, "Множення на 3 (звичайне)  : ");
    test(&Test::timetest3, b, b.M, "Множення на 3 (зсув бітів): ");

    return 0;
}
```

На екран виводяться такі результати.

Без директиви `#pragma optimize("", on)`

```
Множення на 2 (звичайне)  : 28
Множення на 2 (зсув бітів): 32
Множення на 3 (звичайне)  : 26
Множення на 3 (зсув бітів): 34
```

З директивою `#pragma optimize("", on)`

```
Множення на 2 (звичайне)  : 24
Множення на 2 (зсув бітів): 29
Множення на 3 (звичайне)  : 24
Множення на 3 (зсув бітів): 29
```

Як бачимо, результати оптимізації компілятора і за допомогою розрядного зсуву відрізняються один від одного, щоправда, не на користь розрядного зсуву.

1.5. Макроси та inline-функції

Виклик функції належить до однієї з найбільш повільних операцій. Заміна функцій макросами або inline-функціями дозволяє досягти значного прискорення. Проілюструємо це твердження наступним прикладом.

Виклик функції і макрос

```
#include <cmath>
#include <iostream>
#include <locale>

#define SQUARE(x) ((x)*(x))

using namespace std;

struct Test
{
    static const long N = 1000000L;
```

```
    static const int  M = 1000;

    double x, y;
    Test(double u) : x(u) {}
    void timetest1() { for (long i = 1; i < N; i++) y = pow(x, 2); }
    void timetest2() { for (long i = 1; i < N; i++) y = SQUARE(x); }
    void timetest3() { for (long i = 1; i < N; i++) y = x * x; }
};

#include "test"

int main()
{

    system("chcp 1251 > 0");

    Test a(2);

    test(&Test::timetest1, a, a.M, "Кількість тактів (pow)   : ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (макрос): ");
    test(&Test::timetest3, a, a.M, "Кількість тактів (x*x)   : ");

    return 0;
}
```

В результаті виконання цієї програми ми отримаємо таке повідомлення.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (pow)   : 55
Кількість тактів (макрос): 3
Кількість тактів (x*x)   : 3
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (pow)   : 54
Кількість тактів (макрос): 3
Кількість тактів (x*x)   : 3
```

В даному випадку результати практично однакові.

Зазначимо три обставини. По-перше, використання макросу і оператора `x*x` зменшує кількість тактів на порядок. По-друге, результати використання макросу і оператора є майже однаковими (причому в переважній більшості запусків оператор виконувався трохи швидше). По-третє, при різних запусках програми кількість тактів процесора, витрачених на виконання програми, незначно коливається.

Може здатися, що макрос містить зайві дужки. Однак такий висновок був би поспішним. Річ у тім, що підставлення виконується “наосліп”. Компілятор не перевіряє контекст оператора. Отже, можливі неоднозначності. Припустимо, ми знехтували дужками і визначили макрос так.

```
#define SQUARE(x) x*x
```

Припустимо, що у програмі цей макрос виконує таку підстановку:

```
y = SQUARE(x+2);
```

Компілятор формально замінить праву частину `SQUARE(x+2)` оператор `x+2*x+2` (нагадаємо, що дужки ми не передбачили!). Отже, результат є зовсім іншим.

Якщо є ми поставимо дужки

```
#define SQUARE(x) (x)*(x),
```

то отримаємо правильну відповідь. Однак цього все ще недостатньо, щоб запобігти помилкам. Якщо в програмі є вираз `y = i / SQUARE(X+2)`, то він буде розгорнутий у вираз `y = i/(x+2)*(x+2)`. Оскільки оператори `*` та `/` мають однаковий пріоритет, вони будуть виконуватися зліва направо, і замість числа `i/((x+2)*(x+2))` ми отримаємо число `(i/(x+2))*(x+2)`, тобто чисельник `i`.

Зазначимо кілька недоліків, властивих макросам. По-перше, аргументи макросів не запам'ятовуються у тимчасових змінних, а обчислюються щоразу заново. Якщо аргументом макросу є досить складний вираз, то робота програми може навіть сповільнитись. По-друге, довжина макросів обмежена, а їх синтаксис іноді виглядає заплутаним, що зменшує читабельність програми. Макроси — дуже потужний і корисний механізм, якщо його правильно використовувати у відповідних ситуаціях.

З іншого боку, цих проблем можна уникнути, використавши замість макросів inline-функції, тіло яких, як і макроси, безпосередньо вставляються в текст програми. Проте треба пам'ятати, що використання inline-функції компілятор тлумачить лише як рекомендацію.

1.6. Розгортання циклів

Часто програмісти не беруть до уваги, що конструкція циклу з лічильником є дещо громіздкою. У ній передбачені три операції — ініціалізація (виконується на початку циклу), а також перевірка та збільшення лічильника (виконуються на кожній ітерації). Для оптимізації циклів застосовують два прийоми: розгортання

циклів та об’єднання циклів. Розгортання циклів полягає у тому, що тіло циклу повторюють фіксовану кількість разів (наприклад, десять разів). Вважається, що такий прийом дозволяє зменшить в десять разів звертання до перевірки виходу за межі діапазону і таким чином прискорити обчислення, хоча читабельність програми при цьому знижується.

Розгортання циклу

```
#include <iostream>
#include <locale>

using namespace std;

struct Test
{

    static const long N = 10000000L;
    static const int  M = 1000;

    double x, y;
    Test(double a, double b) :x(a), y(b) {}
    void timetest1() { for (long int i = 0; i < N;i++) y = x / i; }
    void timetest2()
    {
        for (long int i = 1; i < N;)
        {
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
            y = x / i; i++;
        }
    }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a(2, 0);

    test(&Test::timetest1, a, a.M, "Кількість тактів (звичайний цикл):    ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (розгорнутий цикл):  ");

    return 0;
}
```

На екрані ми побачимо такі повідомлення.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (звичайний цикл):    79
Кількість тактів (розгорнутий цикл):  79
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (звичайний цикл):    79
Кількість тактів (розгорнутий цикл):  79
```

Виявляється, що прискорити програму із звичайним циклом за рахунок розгортання неможливо. Цю задачу вирішує компілятор.

1.7. Об’єднання циклів

Якщо кілька циклів мають однаковий діапазон зміни лічильника, можна заощадити на операціях порівняння та збільшення, об'єднавши цикли в один. Припустимо, потрібно обчислити матрицю Гільберта та відповідну нижню трикутну матрицю.

Об`єднання циклів

```
#include <iostream>
#include <locale>

using namespace std;

struct Test1
{
    static const long N   = 1000000L;
    static const int  M   = 1000;
    static const int  MAX = 10;

    double array1[MAX][MAX], array2[MAX][MAX];

    Test1()
    {
        for (long i = 1; i < N;i++)
        {
            for (int k = 0; k < MAX; k++)
                for (int l = 0; l < MAX; l++)
                    array1[k][l] = 1. / (k + l + 1);
            for (int k = 0; k < MAX; k++)
                for (int l = 0; l < MAX; l++)
                    if (k <= 0)array2[k][l] = 0; else array2[k][l] = array1[k][l];
        }
    }
};

struct Test2
{
    static const long N   = 1000000L;
    static const int  M   = 1000;
    static const int  MAX = 10;
```

```
    double array1[MAX][MAX], array2[MAX][MAX];

    Test2()
    {
        for (long i = 1; i < N;i++)
        {
            for (int k = 0; k < MAX; k++)
                for (int l = 0; l < MAX; l++)
                {
                    array1[k][l] = 1. / (k + l + 1);
                    if (k <= 0)array2[k][l] = 0; else array2[k][l] = array1[k][l];
                }
        }

    }
};


int main()
{
    clock_t start, finish;

    system("chcp 1251 > 0");

    //  Необ'єднані цикли

    start = clock();
    Test1 a;
    finish = clock();

    cout << "Кількість тактів (неоптимальний варіант): " << finish - start << endl;

    // Об'єднані цикли

    start = clock();
    Test2 b;
    finish = clock();

    cout << "Кількість тактів (оптимальний варіант)  : " << finish - start << endl;

    return 0;
}
```

Програма повідомляє таке.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (неоптимальний варіант): 79
Кількість тактів (оптимальний варіант)  : 79
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (неоптимальний варіант): 79
Кількість тактів (оптимальний варіант)  : 78
```

Як бачимо, об'єднання циклів трохи скорочує час виконання програми, якщо включена оптимізація компілятора.

1.8. Інверсія циклу

На деяких комп'ютерах декрементація та порівняння з нулем виконуються швидше, ніж інкрементація та порівняння з цілим числом. Перевірити цей факт можна за допомогою наступної програми.

Прямий і обернений цикл

```
#include <iostream>
#include <locale>

using namespace std;


struct Test
{
    static const long N = 10000000L;
    static const int  M = 1000;
    int x;

    Test() :x(0) {}
    void timetest1() { for (long int i = 0; i < N;i++) x++; } // Прямий цикл
    void timetest2() { for (long int i = N; i > 0;i--) x++; } // Обернений цикл

};

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a;

    test(&Test::timetest1, a, a.M, "Кількість тактів прямого циклу    : ");
    test(&Test::timetest2, a, a.M, "Кількість тактів оберненого циклу : ");

    return 0;
}
```

Результат роботи програми виглядає так.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів прямого циклу    : 32
Кількість тактів оберненого циклу : 32
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів прямого циклу    : 32
Кількість тактів оберненого циклу : 32
```

Очевидно, що компілятор здійснює оптимізацію так, що додаткова інверсія циклу ефекту не дає.

Ця програма демонструє час роботи прямого і оберненого циклу. На сучасних комп'ютерах цей прийом не дає суттєвого виграшу: обернений цикл виконується з тією ж швидкістю, що й прямий.

1.9. Заміна виразів

Багато виразів можна спростити, замінивши їх фрагменти ефективнішими еквівалентами.

Заміна виразів

```
#include <iostream>
#include <cmath>
#include <locale>

using namespace std;

struct Test
{
    static const long N = 1000000L;
    static const int  M = 1000;
    long int x, y, z;

    Test(int a, int b, int c) : x(a), y(b), z(c) {}

    void timetest1() // Неоптимальні вирази
    {
        for (long i = 0; i < N;i++)
        {
            x = y % 8;
            y = pow(x, 2);
            z = y * 33;
        }
    }

    void timetest2()// Оптимальні вирази
    {
        for (long i = 0; i < N;i++)
        {
            x = y & 7;          // x = y % 8;
            y = x * x;          // y = pow(x, 2);
            z = (y << 5) + y;   // y*(32+1) = (y << 5) + y
        }
    }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a(0, 18, 0);
```

```
    test(&Test::timetest1, a, a.M, "Кількість тактів (неоптимальні вирази): ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (оптимальні вирази)  : ");

    return 0;
}
```

Ми замінили оператори у тілі циклу їх ефективними еквівалентами:

```
  x = y & 7;          // x = y % 8;
  y = x*x;            // y = pow(x, 2);
  z = (y << 5) + y;   // y*(32+1) = (y << 5) + y
```

Результати виглядають так.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (неоптимальні вирази): 25
Кількість тактів (оптимальні вирази)  : 6
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (неоптимальні вирази): 24
Кількість тактів (оптимальні вирази)  : 5
```

Час виконання зменшився майже в п’ять разів. Оптимізація дає додатковий виграш в один такт, але зауважимо, що основний виграш дає заміна виразів, яку компілятор, очевидно, не виконує.

1.10. Інваріанти циклів

Інваріантом циклу називається будь-який вираз, який ані прямо, ані опосередковано не залежить від лічильника. Їхня наявність у тілі циклу лише затримує його виконання. Кандидатами на видалення з тіла циклу мають бути такі операції: індексування масиву, розіменування вказівника та виклик функції.

Винесення інваріантів

```
#include <iostream>
#include <cmath>
#include <locale>

using namespace std;

struct Test
{
    static const int n = 2000;
    static const int M = 1000;

    int u[n], v[n], w[n];

    Test()
    {
        for (long i = 0; i < n; i++) { u[i] = 1; v[i] = 2; w[i] = 0; }
    }
```

```
    void timetest1() // Звичайний варіант
    {

        for (long i = 0; i < n; i++)
            for (long j = 0; j < n; j++)
                w[j] = u[j] * v[i];
    }

    void timetest2() // Винесення інваріантів
    {
        for (long i = 0; i < n; i++)
        {
            double x = v[i];
            for (int j = 0; j < n; j++)
                w[j] = x * u[j];
        }
    }
};

#include "test"


int main()
{
    system("chcp 1251 > 0");

    Test a;

    test(&Test::timetest1, a, a.M, "Кількість тактів (звичайний варіант)    : ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (винесення інваріантів): ");

    return 0;
}
```

Ця програма повертає такі результати.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (звичайний варіант)    : 13
Кількість тактів (винесення інваріантів): 15
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (звичайний варіант)    : 12
Кількість тактів (винесення інваріантів): 15
```

Як бачимо, суттєвої різниці немає, але оптимізація компілятора дозволяє виграти додатковий такт.

1.11. Використання констант

Зазвичай, використання літеральних констант в програмах засуджується, тому що вони зменшують гнучкість програми. Але такі константи іноді рекомендують використовувати для підвищення швидкодії. Наприклад,

припустимо, що ми хочемо присвоїти кожному другому елементу одного масиву відповідний елемент іншого масиву. Ми можемо просто вказати індекс, кратний двом. Гіпотетично, якщо ввести допоміжну цілочисельну змінну і змінювати її на бажаний крок, то швидкість обчислень може збільшитися.

Використання констант

```
#include <iostream>
#include <cmath>
#include <locale>
#pragma comment(linker, "/STACK:16777216")
using namespace std;

struct Test
{
    static const long N = 1000000L;
    static const int  M = 1000;

    char x[N];

    Test()
    {
        for (long i = 0; i < N; i++) x[i] = 0;
    }

    void timetest1() // Звичайний варіант
    {
        for (int i = 0; i <= N - 2; i = i + 2)
        {
            x[i] = 1;
        }
    }

    void timetest2() // Константа
    {
        long j = 0;
        for (long i = 0; i < N / 2; i++)
        {
            x[j] = 1;
            j = j + 2;
        }
    }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a;

    test(&Test::timetest1, a, a.M, "Кількість тактів (звичайний варіант)     : ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (використання константи): ");
```

```
    return 0;
}
```

Як результат маємо:

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (звичайний варіант)    : 1
Кількість тактів (використання константи: 1
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (звичайний варіант)    : 1
Кількість тактів (використання константи: 1
```

Отримані результати свідчать про те, що між звичайним варіантом і використанням констант, а також між оптимізацією компілятора та використанням констант різниці по швидкодії немає. Крім того, програма виконується практично миттєво.

1.12. Винесення інваріантних умов

Якщо вкладені цикли містять інваріантні умови, тобто умови. які не залежать від їх лічильників, ці умови слід винести в зовнішній цикл.

Винесення інваріантних умов

```
#include <iostream>
#include <cmath>
#include <locale>

using namespace std;

struct Test
{
    static const int n = 2000;
    static const long M = 1000;

    double u[n], v[n], w[n];

    Test()
    {
        for (long i = 0; i < n; i++) { u[i] = 1; v[i] = 2; w[i] = 0; }
    }

    void timetest1() // Звичайний варіант
    {

        for (long j = 0; j < n; j++)
        {
            for (long i = 0; i < n; i++)
            {
                u[i] = u[i] + v[i];
```

```
                if (w[j] > 0) v[i] = 0;
            }
        }

    }

    void timetest2() // Винесення інваріантних умов
    {
        for (long j = 0; j < n; j++)
        {
            if (w[j] > 0)
                for (long i = 0; i < n; i++)
                {
                    u[i] = u[i] + v[i];
                    v[i] = 0;
                }
            else
                for (long i = 0; i < n; i++)
                    u[i] += v[i];

        }
    }
};

#include "test"

int main()
{

    system("chcp 1251 > 0");

    Test a;

    test(&Test::timetest1, a, a.M, "Кількість тактів (неоптимальний варіант): ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (оптимальний варіант)  : ");

    return 0;
}
```

Як результат маємо:

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (неоптимальний варіант): 16
Кількість тактів (оптимальний варіант)  : 13
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (неоптимальний варіант): 15
Кількість тактів (оптимальний варіант)  : 13
```

Як бачимо, оптимізація компілятора покращує швидкодію програми, але не настільки, як винесення інваріантних умов.

Слід зауважити, що часто швидкодія програми залежить не тільки від кількості обчислень, а й від швидкості доступу до комірок пам’яті. У зв’язку з цим слід ретельно підходити до розташування великих наборів даних, вибираючи для них оптимальну структуру та спосіб розташування. Елементи, до яких програма звертається частіше, повинні міститись в більш доступних ділянках пам’яті (як правило, цій умові добре відповідає послідовний принцип зберігання однорідних даних в масивах або в кеші). При роботі з циклами необхідно в першу чергу оптимізувати вкладені цикли. Зберігаючи великі, але розріджені матриці (тобто матриці з великою кількістю нулів), слід намагатися економити пам’ять, вибираючи способи щільного зберігання чисел (без нулів). Це окрема і дуже глибока тема, пов’язана з матричними алгоритмами.

2. ОПТИМІЗАЦІЯ РОБОТИ З ПАМ’ЯТТЮ

Питання оптимізації обчислень на C++ не обмежується лише підвищенням швидкодії виконання операторів. Не менш важливі питання оптимізації роботи з пам’яттю, зокрема підвищення ефективності доступу до елементів та їх економного зберігання. Ключовим моментом під час оптимізації роботи з пам’яттю є локалізація посилань. Цей термін означає здатність програми використовувати близько розташовані адреси пам’яті. Час доступу до таких осередків пам’яті набагато менший, отже швидкість роботи програми збільшується. Програміст може впливати на локалізацію посилань, змінюючи порядок доступу та виділення пам’яті. Для цього слід розділити об'єкти на “часто використовувані” та “рідко використовувані”.

Існує кілька практичних прийомів, які дозволяють підвищити швидкість роботи програми за рахунок ефективної роботи з пам'яттю.

2.1. Доступ до елементів багатовимірних масивів

Вважається, що при доступі до елементів багатовимірних масивів слід спочатку змінювати індекс стовпця. Проілюструємо цю тезу наступною програмою.

Доступ до елементів масиву (зміни індексу рядка)

```
#include <iostream>
#include <cmath>
#include <locale>
#pragma comment(linker, "/STACK:16777216")

using namespace std;

struct Test
{
    static const int  n = 500;
    static const long M = 1000;

    double x[n][n];

    void timetest1() // Першим змінюється індекс рядка
    {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                x[i][j] = 1. / (i + j + 1);
    }

    void timetest2()// Першим змінюється індекс стовпця
```

```
    {
        for (int j = 0; j < n; j++)
            for (int i = 0; i < n; i++)
                x[i][j] = 1. / (i + j + 1);
    }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");

    Test a;

    test(&Test::timetest1, a, a.M, "Кількість тактів (індекс рядка)  :     ");
    test(&Test::timetest2, a, a.M, "Кількість тактів (індекс стовпця):     ");

    return 0;
}
```

На екран виводяться такі результати.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (індекс рядка)  :     2
Кількість тактів (індекс стовпця):     1
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (індекс рядка)  :     2
Кількість тактів (індекс стовпця):     1
```

Як бачимо, якщо змінити порядок доступу так, щоб у першу чергу змінювався індекс рядка, середній час виконання програми зменшується. У міру зростання розміру масиву цей ефект підсилюється, а оптимізація компілятора на це не впливає.

2.2. Переупорядкування членів структури

Іноді вирівнювання машинних слів може призвести до неекономного використання пам’яті. У цьому випадку необхідно переупорядкувати елементи класу. Розглянемо приклад.

```
struct Test
{
    float  a1;
    double a2;
    float  a3;
    double a4;
    short  a5;
    long   a6;
    short  a7;
```

```
    long   a8;
    char   a9;
    int    a10;
    char   a11;
    int    a12;
}
```

Згрупуємо елементи цього класу за типами.

```
struct Test
{
    double a4;
    double a2;
    long   a6;
    long   a8;
    float  a1;
    float  a3;
    int    a10;
    int    a12;
    short  a5;
    short  a7;
    char   a9;
    char   a11;
}
```

Перевіримо, наскільки змінився розмір структури після переупорядкування.

Переупорядкування членів структури

```
#include <iostream>
#include <locale>

using namespace std;

struct Test1
{
public:
    float   a1;
    double  a2;
    float   a3;
    double  a4;
    short   a5;
    long    a6;
    short   a7;
    long    a8;
    char    a9;
    int     a10;
    char    a11;
    int     a12;
};

struct Test2
{
public:
    double  a2;
    double  a4;
    float   a1;
    float   a3;
```

```
    long    a6;
    long    a8;
    short   a5;
    short   a7;
    int     a10;
    int     a12;
    char    a11;
    char    a9;
};

int main()
{
    system("chcp 1251 > 0");

    Test1 Obj1;
    Test2 Obj2;

    cout << "Розмір Obj1 = " << sizeof(Obj1) << " байтів" << endl;
    cout << "Розмір Obj2 = " << sizeof(Obj2) << " байтів" << endl;
    return 0;
}
```

Увімкнувши опцію вирівнювання машинних слів, скомпілюємо цю програму. У результаті виявляємо таке.

```
Розмір Obj1 = 64 байтів
Розмір Obj2 = 48 байтів
```

Як бачимо, виграш виявився значним (майже 30%).

2.3. Копіювання великих масивів

Існує декілька способів копіювання великих масивів, зокрема, функція `memcpy`, алгоритм `copy` та оператор присвоєння стандартного шаблонного масиву `array` із стандартної бібліотеки шаблонів STL. Спробуємо з'ясувати, який з цих способів швидше працює з великими масивами.

Порівняння memcpy, copy та оператора присвоєння

```
#pragma comment(linker, "/STACK:16777216")
#include <iostream>
#include <locale>
#include <array>
#include <algorithm>

using namespace std;

struct Test
{
    static const long M = 100000L;
    static const long N = 1;

    void timetest1()
    {
```

```
        int x[M], y[M];
        for (int i = 0; i < M; i++) { x[i] = i; }
        memcpy(x, y, sizeof(int) * M);;
    }

    void timetest2()
    {
        array<int, M> x, y;
        for (int i = 0; i < M; i++) { x[i] = i; }
        copy(x.begin(), x.end(), y.begin());
    }

    void timetest3()
    {
        array<int, M> x, y;
        for (int i = 0; i < M; i++) { x[i] = i; }
        y = x;
    }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");
    Test a;

    test(&Test::timetest1, a, a.N, "Кількість тактів (memcpy): ");
    test(&Test::timetest2, a, a.N, "Кількість тактів (copy)  : ");
    test(&Test::timetest3, a, a.N, "Кількість тактів (=)     : ");

    return 0;
}
```

Результат є цілком очікуваним. Всі алгоритми однаково швидкі.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (memcpy): 1
Кількість тактів (copy)  : 1
Кількість тактів (=)     : 1
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (memcpy): 1
Кількість тактів (copy)  : 1
Кількість тактів (=)     : 1
```

Очевидно, що оптимізація компілятора на такий код не впливає, тому що оптимізація здійснена в самих алгоритмах.

2.4. Алгоритмічні трюки

Виконуючи алгоритми, потрібно прагнути використовувати будь-яку можливість оптимізації. Згадаємо ще кілька прийомів, які дозволяють заощаджувати пам’ять та підвищувати швидкість роботи програми.

У алгоритмах сортування часто виконується операція перестановки елементів масиву. Традиційно для цього використовується тимчасова змінна, в яку записується елемент, що замінюється. Однак деякі трюки дозволяють уникнути додаткових витрат часу та пам’яті при перестановці великих об'єктів. Для цього слід застосовувати обмін вказівників та спеціальні способи перестановки цілих чисел (адресів). Проілюструємо це на прикладі програми, яка економить об’єм пам'яті за допомогою заміни функції перестановки елементів, яка використовує три елементи, на функцію, яка використовує два елементи. Зауважимо, що цей виграш досягається за рахунок швидкості. Не забувайте, що оптимізація — багатокритеріальна задача!

Сортування методом бульбашки

```
#include <iostream>
#include <locale>
#include <cmath>

using namespace std;

struct Test
{
    static const int N = 1000;
    static const int M = 1000;
    int array[M];

    Test() { for (int i = 0; i < M; ++i) array[i] = M - i; }

    void swap1(int x, int y)
    {
        int tmp = x;
        x = y;
        y = tmp;
    }

    void swap2(int x, int y)
    {
        x = x ^ y;
        y = x ^ y;
        x = x ^ y;
    }
```

```
    void timetest1() // Звичайний варіант перестановки
    {
        for (int i = M - 1; i > 0;i--)
            for (int j = 1; j < i + 1; j++)
                if (array[j] < array[j - 1]) swap1(array[j], array[j - 1]);

    }

    void timetest2()// Першим змінюється індекс стовпця
    {
        for (int i = M - 1; i > 0;i--)
            for (int j = 1; j < i + 1; j++)
                if (array[j] < array[j - 1]) swap2(array[j], array[j - 1]);
     }
};

#include "test"

int main()
{
    system("chcp 1251 > 0");
    Test a;

    test(&Test::timetest1, a, a.N, "Кількість тактів (звичайна перестановка): ");
    test(&Test::timetest2, a, a.N, "Кількість тактів (варіант XOR)          : ");

    return 0;
}
```

Результат роботи програми свідчить про майже однаковий час виконання програми при застосуванні звичайної перестановки і перестановки за допомогою оператора XOR.

Без директиви `#pragma optimize("", on)`

```
Кількість тактів (звичайна перестановка): 5
Кількість тактів (варіант XOR)          : 5
```

З директивою `#pragma optimize("", on)`

```
Кількість тактів (звичайна перестановка): 4
Кількість тактів (варіант XOR)          : 5
```

Приклади програм, наведені вище, демонструють, що не всі прийоми, які вважаються корисними, дійсно дають бажаний ефект. Більшість задач з оптимізацію коду виконує компілятор, але є прийоми (оптимізація математичних виразів, використання макросів та inline-функцій, заміна виразів та зміна індексу масиву), які підвищують швидкість виконання програм. Сполучення оптимізації за допомогою компілятора та цих прийомів дозволяє досягти значного виграшу.

КОНТРОЛЬНІ ПИТАННЯ

1. Як пов’язані об’єктно-орієнтована та модульна парадигми?
2. У чому полягає модульний підхід?
3. Опишіть життєвий цикл програми.
4. У чому полягають ясність і простота коду?
5. Як вимірюється швидкість виконання операцій в C++?
6. Які способи спрощення математичних виразів ви можете назвати?
7. Що таке макроси? Назвіть їх переваги і недоліки.
8. Назвіть способи роботи з циклами, які прискорюють швидкість виконання програми.
9. Назвіть особливості копіювання великих масивів.
10. Як оптимально працювати із структурами?
11. Які алгоритмічні трюки для прискорення роботи програми ви знаєте?
12. Які алгоритмічні трюки для економії пам’яті ви знаєте?

КОНТРОЛЬНІ ЗАВДАННЯ

1. Напишіть програму, яка обчислює значення полінома за схемою Горнера. Порівняйте кількість тактів, що витрачаються на безпосередні обчислення значення полінома та за схемою Горнера.
2. Напишіть програму для обчислення дробу $a = \dfrac{(b-c)\left((b-c)^5 + b\right)}{(b-c)^3 + b}$, за допомогою мінімальної кількості операцій.
3. Для двох довільних цілих чисел `X` та `Y` знайдіть мінімальну кількість операцій додавання та віднімання, після виконання яких числа `X` та `Y` стануть однаковими. Напишіть програму, яка виконує відповідний алгоритм.
4. Для довільного цілого числа `N` знайдіть мінімальну кількість операцій подвоєння та додавання одиниці, щоб, починаючи з нуля, дістатися до `N`. Напишіть програму, яка виконує відповідний алгоритм.
5. Для довільного додатного числа `N` знайдіть мінімальну кількість операцій `N = N - P`, де `P` — найменший простий дільник числа `N`, за допомогою яких число `N` зменшується до нуля. Напишіть програму, яка виконує відповідний алгоритм.
6. Для двох довільних цілих чисел `X` та `Y`, таких що `X < Y`, знайдіть мінімальну кількість операцій множення на два та віднімання одиниці, щоб перетворити число `X` на число `Y`. Напишіть програму, яка виконує відповідний алгоритм.
7. Задано масив `X` довжини `N` і ціле число `M`. Всі елементи масиву більше одиниці і менші `N`. Знайдіть мінімальну кількість операцій вибору двох елементів з наступним додаванням до них одиниці і вибору одного елемента з наступним додаванням до нього одиниці, після виконання яких усі елементи масиву стають однаковими. Напишіть програму, яка виконує відповідний алгоритм.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Optimizing C and C++ code. https://www.eventhelix.com/embedded/optimizing-c-and-cpp-code/
2. Obregon A. Optimizing C++ Code for Performance. https://medium.com/@AlexanderObregon/optimizing-c-code-for-performance-a6b89e2b09bf
3. Fog A. Optimazing software in C++ https://www.agner.org/optimize/optimizing_cpp.pdf
4. Godbolt M. Optimizations in C++ compilers // Acmqueue, 2019, vol. 17, issue 5.
5. Optimization best practices. https://learn.microsoft.com/en-us/cpp/build/optimization-best-practices?view=msvc-170
6. Isensee P. C++ Optimization Strategies and Techniques. https://www.tantalon.com/pete/cppopt/main.htm
7. CPP optimization diary. https://cpp-optimizations.netlify.app/
8. Tips for Optimizing C/C++ Code. https://people.computing.clemson.edu/~dhouse/courses/405/papers/optimize.pdf
9. 10 Ways Optimize C++ Code. https://blog.devgenius.io/10-ways-optimize-c-code-cd21a2583d39
10. Optimize C++ code. https://iq.opengenus.org/optimize-cpp-code/