TiDA 01amb – Bubble sort: indeksy vs wskaźniki

Cel: zrozumieć działanie bubble sort w dwóch wersjach (indeksowej i wskaźnikowej), nauczyć się mierzyć czas wykonania, poznać różnice między buildem Debug i Release oraz podstawy pracy z debuggerem w Visual Studio.

Kontekst i plan

- Mechanika bubble sort ($O(n^2)$, kiedy przerywamy).
- Indeksy vs wskaźniki – jak „chodzimy" po pamięci.
- Pomiar czasu `std::chrono` i higiena benchmarku.
- Debug vs Release – optymalizacje, inlining, rejestry.
- Debugger w Visual Studio – breakpoints, Watch, Memory, Disassembly.
- Cache procesora i zmienność czasów.
- Zadania.

Słowniczek pojęć (będziemy do nich wracać):
Aliasing – sytuacja, w której dwa różne wskaźniki/referencje mogą wskazywać na tę samą komórkę pamięci, co utrudnia kompilatorowi optymalizacje.
Inlining – „wklejenie" ciała funkcji w miejsce wywołania, bez skoku do osobnej funkcji (mniej narzutu, szansa na dalsze optymalizacje).
Rejestry – bardzo szybkie małe „pola" w CPU, gdzie trzymane są bieżące wartości (szybsze niż pamięć RAM i cache).
Cache – pamięć podręczna CPU (L1/L2/L3), przyspiesza dostęp do ostatnio używanych danych.

Szkielet programu i informacja o konfiguracji build'u

Fragment A – nagłówki i wykrywanie Debug/Release

```
#include <algorithm>
#include <chrono>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <random>
#include <string>
#include <vector>

static inline const char* build_config() {
#ifdef _DEBUG
    return "DEBUG (no optimizations, extra checks)";
#else
    return "RELEASE (optimized)";
#endif
}
```

Wyjaśnienie:

- Używamy standardowych nagłówków: losowanie danych, czas, wektory, itp.
- Funkcja `build_config()` zwraca napis zależny od makra `_DEBUG`. Dzięki temu na starcie programu zobaczymy, czy aktualny build to Debug czy Release.

Kontrola poprawności i narzędzia pomocnicze

Fragment B – sprawdzanie posortowania i porównanie zawartości

```
bool is_sorted_non_decreasing(const std::vector<int>& v) {
    for (size_t i = 1; i < v.size(); ++i)
        if (v[i - 1] > v[i]) return false;
    return true;
}

bool same_content(const std::vector<int>& a, const std::vector<int>& b) {
    return a.size() == b.size() && std::equal(a.begin(), a.end(), b.begin());
}
```

Po co?

- Po sortowaniu weryfikujemy, czy wynik faktycznie jest niemalejący i czy obie metody dały identyczny rezultat.

Fragment C – generator danych i „rozgrzewka”

```
std::vector<int> make_random_vector(size_t n, uint32_t seed) {
    std::mt19937 rng(seed);
    std::uniform_int_distribution<int> dist(-1'000'000, 1'000'000);
    std::vector<int> v(n);
    for (auto& x : v) x = dist(rng);
    return v;
}
```

Po co?

- Losujemy te same dane dla obu metod. Użycie ziarna (seed) zapewnia powtarzalność.

Fragment D – podgląd adresów w pamięci

```
void print_sample_addresses(const std::vector<int>& v) {
    if (v.size() < 5) return;
    const int* base = v.data();
    std::cout << "Sample addresses (int*):\n";
    for (size_t i = 0; i < 5; ++i) {
        std::cout << "  &v[" << i << "] = " << static_cast<const void*>(base + i)
                  << "  value=" << v[i] << "\n";
    }
}
```

Po co?

- Dzięki `v.data()` widzimy adresy kolejnych elementów – to pomoże w oknie Memory i zrozumieniu arytmetyki wskaźników.

Dwie implementacje bubble sort

Fragment E – wersja indeksowa

```cpp
void bubble_sort_indexed(std::vector<int>& a) {
    const size_t n = a.size();
    if (n < 2) return;

    for (size_t end = n - 1; end > 0; --end) {
        bool swapped = false;
        for (size_t i = 0; i < end; ++i) {
            if (a[i] > a[i + 1]) {
                std::swap(a[i], a[i + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Komentarz:

- Zewnętrzna pętla zmniejsza „koniec” zakresu, bo największy element „wypływa” na koniec.
- Gdy w pełnym przebiegu nie było zamiany (`swapped == false`), kończymy szybciej (tablica już posortowana).

Fragment F – wersja wskaźnikowa

```cpp
void bubble_sort_pointers(int* begin, int* end) {
    if (!begin || !end || end - begin < 2) return;

    for (int* last = end - 1; last > begin; --last) {
        bool swapped = false;
        for (int* p = begin; p < last; ++p) {
            int* q = p + 1;
            if (*p > *q) {
                std::swap(*p, *q);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Co jest inne w wersji wskaźnikowej?

- Iterujemy wskaźnikiem `p` po pamięci i porównujemy wartości przez dereferencję `*p`, `*(p+1)`.
- **Arytmetyka wskaźników** działa w „krokach” typu (tu `sizeof(int)`).
- Ta metoda ułatwia obserwację surowych adresów w debuggerze.

Pojęcie: Aliasing
Aliasing pojawia się, gdy dwa wskaźniki mogą wskazywać na ten sam obszar pamięci. Kompilator musi być ostrożniejszy z optymalizacjami (np. nie przetrzyma wartości z pamięci zbyt długo w rejestrze), bo zmiana przez jeden wskaźnik może wpłynąć na to, co „widzi” drugi.

⏱️ Rzetelny pomiar czasu

Fragment G – miernik czasu

```
template <typename F>
long long measure_us(F&& fn) {
    using clock = std::chrono::high_resolution_clock;
    auto t0 = clock::now();
    fn();
    auto t1 = clock::now();
    return std::chrono::duration_cast<std::chrono::microseconds>(t1 - t0).count();
}
```

🧪 Higiena benchmarku

„Higiena benchmarku” (ang. benchmark hygiene) oznacza zbiór zasad, które pozwalają mierzyć czas działania programu w sposób rzetelny i powtarzalny. Inaczej mówiąc: chodzi o to, żeby wyniki pomiaru faktycznie odzwierciedlały wydajność kodu, a nie przypadkowe czynniki zewnętrzne.

😌 Przykładowe zasady „higieny benchmarku”:

- Zawsze używaj trybu Release

  W Debug kod działa znacznie wolniej, bo kompilator nie optymalizuje i dodaje kontrole bezpieczeństwa. Pomiar w Debugu nie ma sensu.

- Uruchamiaj program bez debuggera (Ctrl+F5)

  Sam debugger potrafi spowalniać wykonywanie programu nawet kilkukrotnie.

- *„Rozgrzej”* program przed pomiarem

  Pierwsze uruchomienie może być wolniejsze (np. przez alokację pamięci, inicjalizację bibliotek, „zimny cache”). Dlatego zwykle pierwsze wykonanie odrzuca się, a mierzy dopiero kolejne.

- Wykonaj kilka prób i policz średnią

  Pojedynczy wynik może być zaburzony przez działanie systemu operacyjnego (np. procesy w tle, scheduler CPU).

- Używaj identycznych danych wejściowych

  Dla porównania dwóch wersji algorytmu (np. indeksy vs wskaźniki) dane muszą być dokładnie te same. Inaczej porównanie jest bez sensu.

- Nie mierz czasu generowania danych, tylko samego algorytmu

  Częsty błąd początkujących: pomiar obejmuje też przygotowanie tablicy, losowanie liczb itp. Pomiar powinien dotyczyć wyłącznie sortowania.

- Wyłącz niepotrzebne procesy w tle

  Gdy CPU jest zajęty czymś innym, czasy wykonania będą losowo się różnić.

💻 Funkcja `main` – kompletna demonstracja

Fragment H – `main` (konfiguracja, rozgrzewka, pętle pomiarowe)

```
int main() {
    std::cout << "Bubble Sort – indeksy vs wskaźniki\n";
    std::cout << "Build: " << build_config() << "\n\n";

    const std::vector<size_t> SIZES = {1000, 2000, 5000, 8000};
    const int TRIALS = 3;
    const uint32_t BASE_SEED = 12345;

    {
        auto warm = make_random_vector(2000, BASE_SEED);
        auto warm_copy = warm;
        bubble_sort_indexed(warm_copy);
        std::vector<int> warm_copy2 = warm;
        bubble_sort_pointers(warm_copy2.data(), warm_copy2.data() + warm_copy2.size());
    }

    std::cout << std::left
              << std::setw(8)  << "Size"
              << std::setw(16) << "Indexed [us]"
              << std::setw(16) << "Pointers [us]"
              << std::setw(10) << "Equal?"
              << "\n";
    std::cout << std::string(50, '-') << "\n";

    for (size_t n : SIZES) {
        long long acc_idx = 0;
        long long acc_ptr = 0;
        bool all_equal = true;

        for (int t = 0; t < TRIALS; ++t) {
            auto data = make_random_vector(n, BASE_SEED + t);
            auto a1 = data;
            auto a2 = data;

            if (n == SIZES.front() && t == 0) {
                print_sample_addresses(a1);
                std::cout << "\n";
            }

            auto time_idx = measure_us([&]() {
                bubble_sort_indexed(a1);
            });
            auto time_ptr = measure_us([&]() {
                bubble_sort_pointers(a2.data(), a2.data() + a2.size());
            });

            if (!is_sorted_non_decreasing(a1) || !is_sorted_non_decreasing(a2) ||
!same_content(a1, a2))
                all_equal = false;

            acc_idx += time_idx;
            acc_ptr += time_ptr;
        }
```

```
        long long avg_idx = acc_idx / TRIALS;
        long long avg_ptr = acc_ptr / TRIALS;

        std::cout << std::left
                  << std::setw(8)  << n
                  << std::setw(16) << avg_idx
                  << std::setw(16) << avg_ptr
                  << std::setw(10) << (all_equal ? "YES" : "NO")
                  << "\n";
    }

#ifdef _DEBUG
    std::cout << "\n[UWAGA] To jest build DEBUG – czasy będą zawyżone.\n"
                 "        Do rzetelnego pomiaru przełącz na Release i uruchom bez debuggera
(Ctrl+F5).\n";
#else
    std::cout << "\n[INFO] To jest build RELEASE – wyniki pomiarów są miarodajne.\n";
#endif

    std::cout << "\nWskazówka: uruchom program kilka razy – czasy mogą się różnić przez cache
i scheduler.\n";
    return 0;
}
```

Na co zwrócić uwagę?

- Te same dane wejściowe dla obu algorytmów w każdej próbie.
- Tabela wyników: średnie czasy w mikrosekundach i kontrola „Equal?”.
- **Jednorazowy** wydruk adresów pamięci ułatwi Wam obserwacje w debuggerze.

Debug vs Release – co naprawdę się zmienia?

- **Debug**: słabe/wyłączone optymalizacje, dodatkowe informacje dla debuggera. Kod bywa „grubszy”, częściej odwołuje się do pamięci.
- **Release**: włączone optymalizacje (np. /O2). Kompilator agresywnie stosuje **inlining**, eliminuje martwy kod, przetrzymuje wartości w **rejestrach**, lepiej układa pętle i instrukcje.
- **Wniosek**: pomiary wydajności wykonujemy **tylko w Release, bez debuggera** (Ctrl+F5), bo obecność debuggera i brak optymalizacji fałszują wyniki.

Pojęcie: Inlining
Kompilator może podmienić wywołanie krótkiej funkcji na jej treść w miejscu użycia. Znika koszt wywołania, a optimizer "widzi więcej" i może dalej upraszczać kod.

Pojęcie: Rejestry
Bardzo szybkie miejsca w CPU na bieżące dane/tymczasowe wyniki. Trzymanie zmiennych w rejestrach jest szybsze niż odczyt/zapis z RAM, a nawet z cache.

Debugger w Visual Studio – szybka ściąga

1. Ustaw **breakpointy**: początek obu funkcji sortujących, wewnętrzna pętla, linia ze `std::swap`.
2. Uruchom **F5** (Debug).
   - **F10 (Step Over)** – wykonaj linię bez wchodzenia do funkcji.
   - **F11 (Step Into)** – wejdź do funkcji.
   - **Shift+F11** – wyjdź z funkcji.
3. Otwórz **Watch/Locals** i obserwuj: `i`, `end`, `p`, `q`, `*p`, `*(p+1)`.
4. Otwórz **Debug → Windows → Memory → Memory 1**. Wklej adres wskaźnika (np. z `p`) i obejrzyj bajty.
5. Otwórz **Disassembly** i porównaj ten sam fragment w Debug vs Release (różnice w liczbie instrukcji, obecny inlining, inne użycie rejestrów).

Cache i zmienność czasów

- Dane „leżące" obok siebie w pamięci częściej trafiają do cache → szybciej.
- Kilka uruchomień da kilka różnych czasów (stan cache, inne procesy, scheduler).
- Dlatego: rozgrzewka, powtórzenia, średnie.

Stabilność pomiarów i wariancja

W sekcji z zadaniami pojawi się takie polecenie:

> „Zmień **SIZES** (np. `{1500, 3000, 6000, 9000}`) i porównaj **stabilność czasów (wariancja)**."

Co to znaczy?

1. **Zmieniasz rozmiary testowych tablic** – program mierzy czas działania dla tablic o innych długościach.

   ```
   const std::vector<size_t> SIZES = {1500, 3000, 6000, 9000};
   ```

2. **Uruchamiasz program kilka razy** i zapisujesz czasy dla każdego rozmiaru.

3. **Porównujesz stabilność wyników** – sprawdzasz, czy dla tego samego rozmiaru czasy są podobne między uruchomieniami.

Rozmiar tablicy	Pierwszy pomiar	Drugi pomiar	Różnica
1500	210 ms	213 ms	+3 ms
3000	810 ms	812 ms	+2 ms
6000	3090 ms	3200 ms	+110 ms

   Mała różnica → **stabilny wynik**, duża różnica → **niestabilny wynik**.

4. **Wariancja** oznacza, jak bardzo wyniki różnią się od średniej.
   - Mała wariancja → wyniki podobne (pomiar wiarygodny).
   - Duża wariancja → wyniki się wahają (coś wpływa na wydajność, np. inne procesy lub cache).

🎓 Zadania (do oddania w formie sprawozdania)

1. **Skopiuj** fragmenty A–H do projektu „Console App” (C++17+, x64). Zbuduj i uruchom.
2. **Debugowanie (F5, konfiguracja Debug):**
   - Breakpointy: początek obu sortów, pętla wewnętrzna, `std::swap`.
   - Zrób **zrzuty ekranu** z: *Locals/Watch* (z `i`, `end`, `p`, `q`, `*p`), *Memory 1* (widok bajtów dla `p`), *Disassembly* (ten sam fragment w Debug i w Release).
3. **Pomiary (Release, Ctrl+F5):**
   - Uruchom program **co najmniej 3 razy**.
   - Zapisz tabelę wyników (kolumny „Indexed [us]”, „Pointers [us]”, „Equal?”) dla każdego rozmiaru.
4. **Badanie stabilności pomiarów**
   - zmień `SIZES` (np. `{1500, 3000, 6000, 9000}`) i porównaj stabilność czasów (wariancja).
5. **Wnioski (≤ 1 strona):**
   - Różnice Debug vs Release (inlining, rejestry, aliasing – kiedy może blokować optymalizacje).
   - Czy wersja wskaźnikowa bywa szybsza? Czemu tak/nie?
   - Jak rośnie czas wraz z rozmiarem danych ($O(n^2)$), wpływ cache).
   - Dlaczego pomiary robimy w Release bez debuggera?

Tipy dla dociekliwych

- Porównaj kod maszynowy `bubble_sort_indexed` i `bubble_sort_pointers` w Release – czy widzisz różnice w użyciu rejestrów/instrukcji?
- Spróbuj oznaczyć parametry wskaźnikowe jako `restrict` (w MSVC – `__restrict`) i sprawdź, czy poprawia to optymalizację (ostrożnie: tylko jeśli **na pewno** nie aliasują!).