

# TiDA 02<sup>amb</sup> – Tablice dynamiczne i statyczne, zarządzanie pamięcią, testy funkcjonalne i wydajnościowe

## CELE

- Zrozumiesz różnicę pomiędzy pamięcią stosu a sterty – od strony fizycznej i organizacyjnej.
  - Poznasz sposób działania alokacji pamięci w systemie operacyjnym.
  - Zrozumiesz, jak architektura x86 vs x64 wpływa na wydajność programu.
  - Nauczysz się korzystać z new, delete, unique\_ptr i RAII.
  - Zbudujesz testy funkcjonalne i wydajnościowe z pomiarem czasu.
  - Zrozumiesz błędy pamięci: **stack overflow, memory leak, dangling pointer**.
- 

## CZĘŚĆ TEORETYCZNA

### Stos i sterta w szczegółach

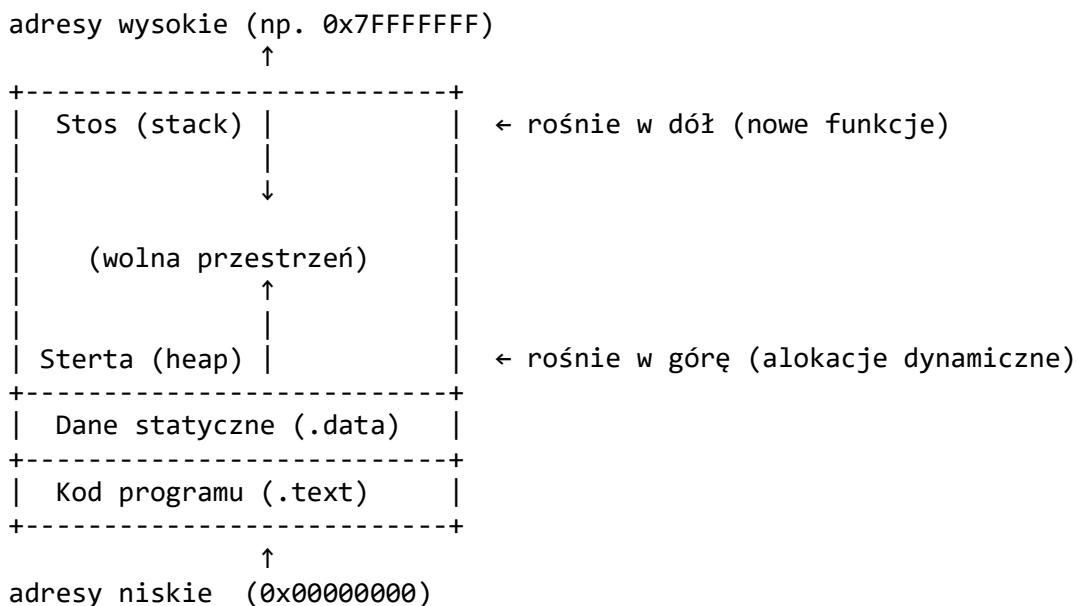
Cecha	Stos (stack)	Sterta (heap)
Przydział pamięci	Automatyczny (kompilator)	Ręczny lub półautomatyczny (new, delete, RAII)
Prędkość	Bardzo szybka (LIFO)	Wolniejsza (alokator systemowy)
Struktura	LIFO – Last In, First Out	Dynamiczna, nieuporządkowana
Kierunek wzrostu	W dół (adresy maleją)	W górę (adresy rosną)
Typowe błędy	Stack overflow	Memory leak, dangling pointer
Typowe zastosowanie	Zmienne lokalne, parametry funkcji	Obiekty dynamiczne, duże tablice

W fizycznym ujęciu zarówno stos, jak i sterta to fragmenty tej samej pamięci RAM.

Różnią się **sposobem zarządzania** przez system operacyjny i kompilator.

---

## Układ pamięci procesu



Procesor korzysta z rejestrów RSP (stack pointer) i RBP (base pointer), a system przydziela pamięć dla sterty dynamicznie.

---

## Architektura x86 vs x64

- W architekturze x64 rejesty mają 64 bity (np. RAX, RBP), a w x86 – 32 bity (EAX, EBP).
  - x64 ma więcej rejestrów ogólnego przeznaczenia – kod działa szybciej.
  - Nie zawsze jednak x64 wygrywa – dla małych danych może być wolniejszy z powodu większych wskaźników.
- 

## RAI<sup>I</sup> – automatyczne zarządzanie pamięcią w C++

RAI<sup>I</sup> (Resource Acquisition Is Initialization) to zasada, według której zasób (np. pamięć) jest zwalniany automatycznie, gdy obiekt wychodzi z zakresu.

W praktyce używamy `std::unique_ptr`:

```
#include <memory>

auto tablica = std::make_unique<int[]>(rozmiar);
// pamięć zostanie zwolniona automatycznie po zakończeniu funkcji
```

---

## CZĘŚĆ PRAKTYCZNA

### Pomocnicze funkcje testowe

```
#include <iostream>
#include <chrono>
#include <memory>
#include <cstdlib>
#include <ctime>

using namespace std;
using namespace std::chrono;

bool czyTablicaPoprawna(int* tablica, int rozmiar, int MIN, int MAX) {
    if (!tablica) return false;
    for (int i = 0; i < rozmiar; ++i)
        if (tablica[i] < MIN || tablica[i] > MAX) return false;
    return true;
}

bool czyTablicaPosortowana(int* tablica, int rozmiar) {
    for (int i = 1; i < rozmiar; ++i)
        if (tablica[i - 1] > tablica[i]) return false;
    return true;
}

bool testFunkcjonalny(int* tablica, int rozmiar, int MIN, int MAX) {
    return czyTablicaPoprawna(tablica, rozmiar, MIN, MAX) &&
           czyTablicaPosortowana(tablica, rozmiar);
}
```

---

### Sortowanie i pomiar czasu

```
void sortBubble(int* tablica, int n) {
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (tablica[j] > tablica[j + 1])
                swap(tablica[j], tablica[j + 1]);
}

template <typename F>
long long measure_us(F&& fn) {
    auto start = high_resolution_clock::now();
    fn();
    auto end = high_resolution_clock::now();
    return duration_cast<microseconds>(end - start).count();
}
```

---

## Porównanie tablic statycznych i dynamicznych

```
void porownajTablice(int rozmiar, int MIN, int MAX) {
    // tablica statyczna (na stosie)
    int* statyczna = new int[rozmiar];
    srand(time(nullptr));
    for (int i = 0; i < rozmiar; ++i)
        statyczna[i] = rand() % (MAX + 1);

    long long t1 = measure_us([&]() { sortBubble(statyczna, rozmiar); });
    cout << "Czas sortowania (tablica statyczna): " << t1 << " µs" << endl;

    // tablica dynamiczna (na stercie, RAII)
    auto dynamiczna = make_unique<int[]>(rozmiar);
    for (int i = 0; i < rozmiar; ++i)
        dynamiczna[i] = rand() % (MAX + 1);

    long long t2 = measure_us([&]() { sortBubble(dynamiczna.get(), rozmiar); });
    cout << "Czas sortowania (tablica dynamiczna): " << t2 << " µs" << endl;

    cout << (testFunkcjonalny(statyczna, rozmiar, MIN, MAX) ? "Statyczna OK\n" : "Statyczna
    błędna\n");
    cout << (testFunkcjonalny(dynamiczna.get(), rozmiar, MIN, MAX) ? "Dynamiczna OK\n" :
    "Dynamiczna błędna\n");

    delete[] statyczna;
}
```

---

## Funkcja główna

```
int main() {
    int rozmiar = 5000;
    int MIN = 0;
    int MAX = 1000;

    cout << "Porównanie tablic statycznych i dynamicznych:\n";
    porownajTablice(rozmiar, MIN, MAX);
}
```

---

## Zadania (do oddania w formie sprawozdania)

1. **Zbadaj granicę stosu** – zwiększa rozmiar tablicy, aż pojawi się *stack overflow*.
  2. **Zastąp sortowanie bąbelkowe std::sort()** i porównaj czasy.
  3. **Uruchom w wersji x86 i x64** – porównaj różnice.
  4. **Sprawdź adresy tablic** w debuggerze – czy są zbliżone?
  5. **Dodaj test unikalności elementów** – rozszerz testFunkcjonalny.
  6. **Wykonaj raport** z porównania czasów i poprawności dla różnych rozmiarów tablic.
-

## DODATKOWE WYZWANIE

Zaimplementuj własny prosty **monitor pamięci** – licząc, ile razy użyto new i delete, aby wykryć ewentualne wycieki pamięci.