AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

...Algorytm listy dwukierunkowej z zastosowaniem GitHub...

Autor:
Imie Nazwisko

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem zadania jest stworzenie klasy `matrix`, która umożliwia zarządzanie macierzami o wymiarach $n \times n$. Klasa powinna obsługiwać podstawowe operacje na macierzach, takie jak ich alokacja, modyfikacja, transpozycja, a także operacje matematyczne, takie jak dodawanie, mnożenie, czy obliczenia z macierzami skalarowymi. Wymagana jest również możliwość inicjowania macierzy wartościami z tablicy jednowymiarowej oraz losowanie wartości w macierzy.

1.1. Wymagania dotyczące implementacji

1. **Reprezentacja macierzy** – Macierz jest reprezentowana jako dwuwymiarowa tablica wskaźników (tablica wskaźników na wskaźniki).
2. **Alokacja pamięci** – Należy zaimplementować funkcje alokujące pamięć dla macierzy o zadanym rozmiarze oraz zwalniające pamięć po zakończeniu operacji na obiekcie.
3. **Operacje na macierzach**:
   - Inicjowanie macierzy z wartościami przekazanymi w postaci tablicy.
   - Modyfikacja elementów w macierzy (wstawianie wartości w określonym wierszu i kolumnie).
   - Transpozycja macierzy – zamiana wierszy z kolumnami.
   - Losowanie wartości w macierzy lub tylko dla wybranych elementów.

1.2. Przykład

Poniżej przedstawiono przykład użycia klasy `matrix`. Program demonstruje tworzenie macierzy, wstawianie wartości, transponowanie oraz wyświetlanie wyników. Przykład użycia klasy matrix[1].

```
1
2 int main() {
3     int n = 4;
4     int t[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
     16 };
5
6     matrix m1;
```

[1]Przykład użycia klasy matrix[1].

```
    matrix m2(n, t);

    std::cout << "Macierz m2:\n" << m2;

    m2.wstaw(0, 0, 42);
    std::cout << "Po wstawieniu 42 do (0, 0):\n" << m2;

    m2.dowroc();
    std::cout << "Po transpozycji:\n" << m2;

    return 0;
}
```

Listing 1. Przykład użycia klasy matrix

W przykładzie:

- Tworzymy macierz `m2` o wymiarach 4×4, wypełnioną wartościami z tablicy `t`.
- Wstawiamy wartość `42` do elementu w lewym górnym rogu macierzy (0, 0).
- Następnie wykonujemy transpozycję macierzy, zamieniając wiersze z kolumnami.
- Na końcu wypisujemy macierz po każdej operacji.

1.3. Instalacja

Aby rozpocząć korzystanie z klasy `matrix`, należy wykonać kilka prostych kroków w celu zainstalowania odpowiednich narzędzi oraz skompilowania programu. Poniżej przedstawiono szczegółową instrukcję instalacji i konfiguracji środowiska, które pozwoli na użycie stworzonej klasy.

1. Wymagania systemowe

Do kompilacji programu potrzebne jest środowisko obsługujące język C++. Wymagane wersje narzędzi to:

- **Kompilator C++** – zaleca się używanie kompilatora GCC lub Clang, które są dostępne na większości systemów operacyjnych.
- **System operacyjny** – program można kompilować na systemach Windows, macOS lub Linux.

- **Edytor kodu** – można używać dowolnego edytora kodu, np. Visual Studio Code, CLion, czy Sublime Text.

1.4. Instalacja kompilatora i środowiska programistycznego

- **Windows**:
  - Zainstaluj kompilator `MinGW` lub korzystaj z `Microsoft Visual Studio`.
  - Skonfiguruj środowisko w terminalu, aby kompilator był dostępny w systemowej ścieżce.
- **Linux / macOS**:
  - Kompilator GCC lub Clang jest zwykle zainstalowany domyślnie. W przypadku braku, można go zainstalować za pomocą menedżera pakietów (np. `sudo apt install g++` na systemach opartych na Debianie).

1.5. Kompilacja programu

Po pobraniu kodu źródłowego i ustawieniu środowiska, należy przejść do katalogu z plikiem źródłowym i wykonać następujące kroki:

- Otwórz terminal lub konsolę w katalogu z plikiem `main.cpp`.
- Użyj polecenia `g++` lub innego kompilatora C++ do kompilacji programu:
  - `g++ main.cpp -o program`
- Uruchom program:
  - `./program`

Po tych krokach program zostanie skompilowany, a jego wykonanie wyświetli wyniki na ekranie.

1.6. Wymagane pliki

Aby program działał poprawnie, należy upewnić się, że wszystkie pliki źródłowe są dostępne w katalogu roboczym:

- `matrix.h` – plik nagłówkowy zawierający deklaracje klasy `matrix`.

- `matrix.cpp` – plik źródłowy implementujący funkcje klasy `matrix`.
- `main.cpp` – plik główny, w którym znajduje się kod programu wykonującego operacje na macierzach.

1.7. Testowanie i uruchamianie

Po poprawnym skompilowaniu programu, należy uruchomić go w celu testowania funkcjonalności. Program powinien poprawnie obsługiwać podstawowe operacje na macierzach, takie jak ich tworzenie, wstawianie wartości, transponowanie i wyświetlanie wyników.

2. Analiza problemu

Celem projektu jest stworzenie klasy `matrix`, która umożliwia wykonywanie operacji na macierzach o wymiarach $n \times n$. Aby skutecznie zrealizować ten cel, musimy rozwiązać szereg problemów związanych z reprezentacją macierzy, alokacją pamięci, efektywnością operacji oraz interfejsem użytkownika. Poniżej przedstawiona zostanie szczegółowa analiza tych zagadnień.

2.1. Reprezentacja macierzy

Pierwszym krokiem w implementacji jest odpowiednia reprezentacja macierzy. W rozwiązaniu przyjęto podejście oparte na dwuwymiarowej tablicy wskaźników, czyli tablicy wskaźników na tablice, które przechowują poszczególne elementy macierzy. Jest to podejście wystarczająco elastyczne, by mogło obsługiwać macierze o dowolnym rozmiarze, zachowując jednocześnie względnie prostą strukturę pamięciową.

Zalety i wady

Zaletą takiej reprezentacji jest łatwość w alokacji i zwalnianiu pamięci oraz prostota dostępu do elementów macierzy. Jednakże, takim rozwiązaniem mogą być obarczone pewne wady:

- Problemy z fragmentacją pamięci — przy dynamicznej alokacji pamięci dla każdej z linii macierzy może występować fragmentacja, zwłaszcza przy dużych macierzach.

- Trudności z przetwarzaniem macierzy o dużych rozmiarach — z racji, że każdy element jest przechowywany w oddzielnej komórce pamięci, operacje na takich strukturach mogą być mniej wydajne w porównaniu do macierzy reprezentowanych jako jednowymiarowe tablice (z jednym dużym blokiem pamięci).

2.2. Alokacja i zwalnianie pamięci

Klasa `matrix` musi dynamicznie przydzielać pamięć dla przechowywania elementów macierzy. W przypadku operacji takich jak inicjalizacja, zmiana rozmiaru macierzy, czy zwalnianie pamięci, należy zadbać o prawidłowe zarządzanie tymi procesami.

Alokacja pamięci

Do alokacji pamięci wykorzystano funkcję `allocateMemory(int size)`, która dynamicznie tworzy tablicę wskaźników oraz przydziela pamięć dla każdej z linii macierzy. Każda linia jest tablicą dynamiczną, a sama pamięć jest przydzielana za pomocą operatora `new`. Dzięki temu rozwiązaniu, rozmiar macierzy może być dowolnie modyfikowany w trakcie działania programu.

Zwalnianie pamięci

Po zakończeniu operacji na macierzy ważne jest zwolnienie przydzielonej pamięci. W tym celu w konstruktorze klasy `matrix` zastosowano odpowiednią funkcję destruktora, która odpowiada za zwolnienie pamięci przypisanej do macierzy, zapewniając tym samym jej prawidłowe usunięcie z pamięci.

Zalety i wady

Zalety tej techniki to możliwość dynamicznej zmiany rozmiaru macierzy oraz pełna kontrola nad pamięcią. Wadą jest jednak konieczność ręcznego zarządzania pamięcią, co zwiększa złożoność kodu i wymaga ostrożności przy alokacji i dealokacji.

2.3. Operacje na macierzach

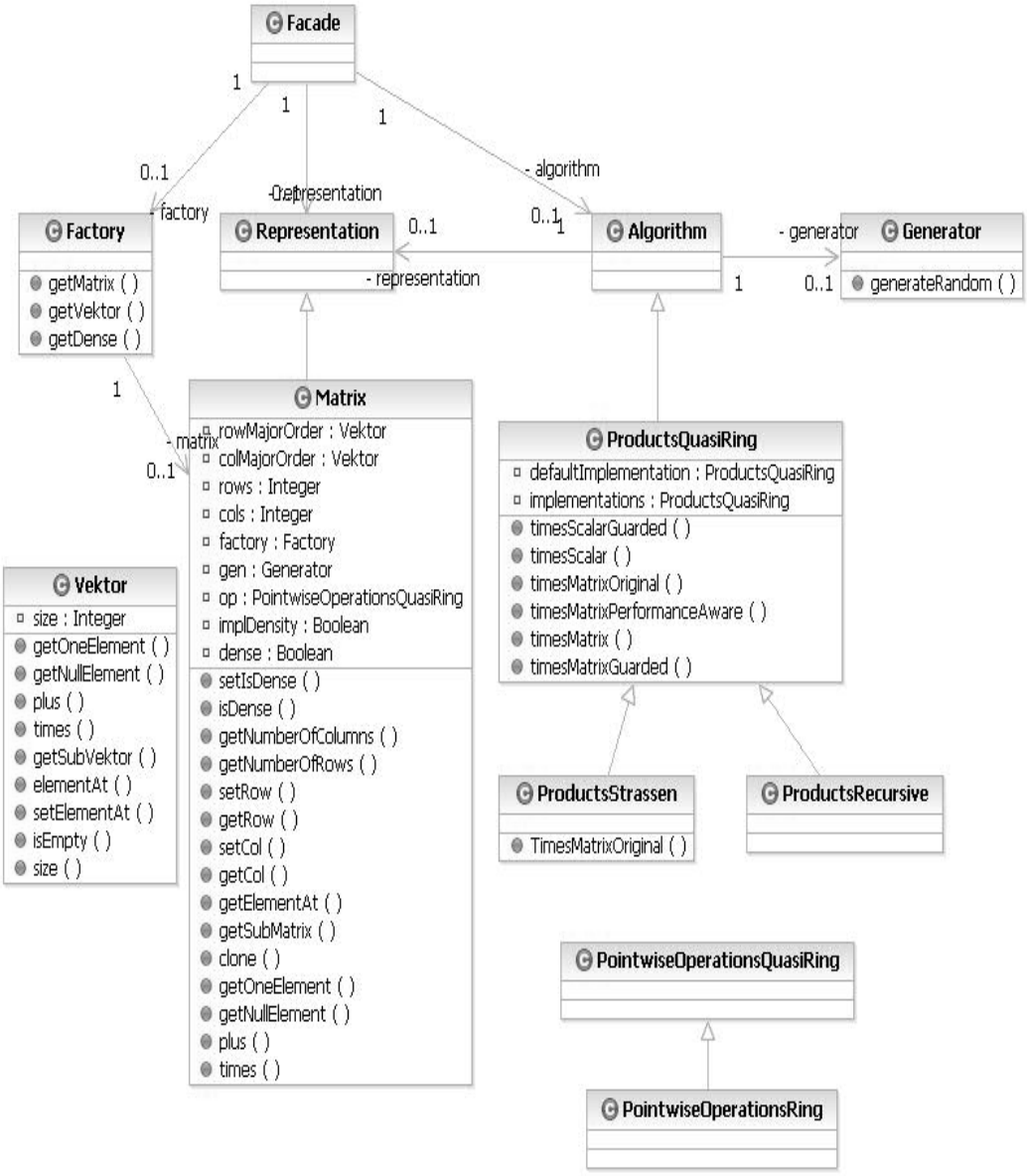
Klasa `matrix` oferuje szereg operacji, które mogą być wykonywane na macierzach, w tym:

- Wstawianie wartości w konkretne komórki macierzy.
- Transpozycja macierzy, czyli zamiana wierszy z kolumnami.
- Losowanie wartości w macierzy.

Operacja wstawiania

Funkcja `wstaw(int x, int y, int wartosc)` umożliwia wstawienie wartości w wybrane miejsce w macierzy. Funkcja ta sprawdza, czy wskazane współrzędne mieszczą się w granicach macierzy i wstawia wartość w odpowiednią komórkę. Jest to stosunkowo prosta operacja, ale niezbędna do modyfikacji macierzy w trakcie jej użytkowania. Diagram UML klasy matrix [2].

[2]Diagram UML klasy matrix[2].



Rys. 2.1. Diagram UML klasy matrix

Transpozycja

Funkcja `dowroc()` wykonuje transpozycję macierzy, zamieniając wiersze z kolumnami. Transpozycja macierzy jest operacją, która może być używana w wielu algorytmach numerycznych i statystycznych. Warto zwrócić uwagę, że transponowanie macierzy jest operacją czasochłonną, szczególnie dla dużych macierzy, ponieważ wymaga odwrotnej iteracji przez wszystkie elementy.

Losowanie wartości

Funkcje `losuj()` i `losuj(int x)` służą do losowego przypisania wartości do elementów macierzy. W przypadku pierwszej funkcji losowane są wartości dla wszystkich elementów macierzy, a w przypadku drugiej funkcji wartości są losowane tylko dla określonej liczby elementów. Losowanie wartości może być wykorzystywane do testowania funkcji lub generowania macierzy o losowych danych.

Zalety i wady

Operacje te są stosunkowo proste w implementacji, ale mogą generować wysokie koszty obliczeniowe, zwłaszcza w przypadku bardzo dużych macierzy. Operacja transpozycji jest szczególnie wymagająca pod względem złożoności czasowej, ponieważ wymaga dwóch zagnieżdżonych pętli, z których każda iteruje po n elementach macierzy, co daje łączną złożoność $O(n^2)$.

2.4. Efektywność algorytmów

Ważnym zagadnieniem, które należy rozważyć przy projektowaniu klasy `matrix`, jest efektywność operacji, zwłaszcza w kontekście macierzy o dużych rozmiarach. Złożoność czasowa operacji, takich jak dodawanie, mnożenie czy transpozycja, rośnie w zależności od rozmiaru macierzy.

Optymalizacja pamięciowa

Optymalizacja pamięciowa jest kluczowa, szczególnie w przypadku operacji na dużych macierzach. Można rozważyć zastosowanie innych struktur pamięciowych, takich jak jednowymiarowe tablice, które mogą pomóc w zredukowaniu fragmentacji pamięci oraz poprawić dostępność danych. Jednakże, zmiana struktury pamięciowej może wiązać się z dodatkowymi trudnościami przy implementacji niektórych operacji, jak np. transpozycja.

Złożoność czasowa

Wszystkie operacje w klasie `matrix` posiadają złożoność czasową co najmniej $O(n^2)$, ze względu na konieczność iteracji przez wszystkie elementy macierzy. Przy dużych macierzach, złożoność może stać się problematyczna, zwłaszcza jeśli operacje będą wykonywane wielokrotnie w ramach algorytmów numerycznych.

2.5. Interfejs użytkownika

Interfejs użytkownika został zaprojektowany w sposób umożliwiający łatwe wykonywanie podstawowych operacji na macierzach. Funkcje takie jak wstawianie, transpozycja, oraz losowanie wartości są dostępne w prosty sposób, a użytkownik może łatwo manipulować macierzami i wyświetlać wyniki. Klasa `matrix` pozwala na wydajną pracę z macierzami, zapewniając intuicyjny sposób dostępu do danych.

3. Projektowanie

Projektowanie klasy `matrix` jest kluczowym etapem, który determinuje efektywność i funkcjonalność całego projektu. W tym rozdziale przedstawiono założenia projektowe, opisano strukturę klasy, szczegółowo omówiono metody i operatory oraz podjęto próbę zaprojektowania modułowego i rozszerzalnego rozwiązania.

3.1. Struktura klasy

Klasa `matrix` została zaprojektowana jako klasa reprezentująca kwadratową macierz o wymiarach $n \times n$. Kluczowymi elementami składowymi klasy są:

- `int n` — rozmiar macierzy.
- `int** data` — wskaźnik na tablicę wskaźników, które przechowują wiersze macierzy.

Taka struktura zapewnia odpowiednią elastyczność przy alokacji pamięci oraz łatwość w implementacji operacji takich jak transpozycja czy losowanie elementów. Dodatkowo, konstruktor klasy umożliwia różne sposoby inicjalizacji macierzy:

- Konstruktor domyślny — tworzy pustą macierz.
- Konstruktor parametryczny — tworzy macierz o zadanym rozmiarze n, zainicjalizowaną zerami lub wartościami podanymi w tablicy.
- Konstruktor kopiujący — umożliwia utworzenie kopii istniejącej macierzy.

3.2. Projektowanie metod

Metody klasy `matrix` zostały zaprojektowane tak, aby umożliwiały łatwą manipulację macierzami oraz wykonywanie standardowych operacji. Kluczowe metody to:

Alokacja pamięci

Funkcja `allocateMemory(int size)` odpowiada za dynamiczną alokację pamięci dla macierzy. Dzięki zastosowaniu tej funkcji wszystkie konstruktory mogą korzystać z jednolitego mechanizmu zarządzania pamięcią.

Modyfikacja elementów macierzy

Funkcja `wstaw(int x, int y, int wartosc)` umożliwia wstawianie wartości do macierzy w wybranym miejscu. Projektując tę funkcję, zadbano o sprawdzanie poprawności indeksów, aby zapobiec błędom wynikającym z nieprawidłowego dostępu do pamięci.

Transpozycja macierzy

Funkcja `dowroc()` realizuje transpozycję macierzy. W projekcie zastosowano tymczasową kopię macierzy, co upraszcza implementację i pozwala na zachowanie oryginalnej macierzy do momentu zakończenia transpozycji.

Losowanie wartości

Funkcje `losuj()` oraz `losuj(int x)` umożliwiają przypisanie losowych wartości do elementów macierzy. Funkcja `losuj()` działa na całej macierzy, natomiast `losuj(int x)` ogranicza losowanie do określonej liczby elementów, co może być przydatne w testach.

3.3. Projektowanie operatorów

Klasa `matrix` obsługuje szereg operatorów, takich jak `+`, `-`, `*`, co umożliwia przeprowadzanie standardowych operacji na macierzach w intuicyjny sposób. W projekcie uwzględniono również operatory inkrementacji, porównania oraz modyfikacji.

Dodawanie i mnożenie macierzy

Operatory `+` i `*` zostały zaimplementowane tak, aby umożliwiały operacje arytmetyczne zarówno pomiędzy macierzami, jak i pomiędzy macierzą a liczbą całkowitą. Przykład użycia: Przykład dodawania dwóch macierzy [3].

```
1 matrix m1(4);
2 matrix m2(4);
3 matrix m3 = m1 + m2;
```

Listing 2. Przykład dodawania dwóch macierzy

[3]Przykład dodawania dwóch macierzy[1].

Porównywanie macierzy

Operator `==` umożliwia porównywanie dwóch macierzy pod kątem ich zawartości. Jest to przydatne w testach jednostkowych oraz przy weryfikacji wyników obliczeń.

Inkrementacja i dekrementacja

Operatory `++` oraz `--` wprowadzają możliwość łatwego zwiększania lub zmniejszania wszystkich elementów macierzy o jeden. Implementacja ta jest intuicyjna i zgodna z konwencjami stosowanymi w języku C++.

3.4. Modułowość i rozszerzalność

W projekcie uwzględniono możliwość łatwego rozszerzania funkcjonalności klasy. Dzięki odpowiedniej organizacji kodu, nowe metody lub operatory mogą być dodawane bez konieczności modyfikowania istniejących komponentów. W szczególności:

- Funkcja `allocateMemory()` może być wykorzystywana w przyszłych rozszerzeniach związanych z operacjami na macierzach.
- Projektując operatory arytmetyczne, zastosowano zasadę separacji odpowiedzialności, co ułatwia dodanie nowych operatorów (np. dzielenia elementów macierzy).

3.5. Obsługa błędów

W projekcie zadbano o podstawowe mechanizmy obsługi błędów, takie jak:

- Sprawdzanie poprawności indeksów w metodach `wstaw()` i `pokaz()`.
- Zwalnianie pamięci w destruktorze, co zapobiega wyciekom pamięci.
- Ochrona przed nieprawidłowym dostępem do macierzy, która nie została zainicjalizowana.

3.6. Interfejs użytkownika

Interfejs użytkownika został zaprojektowany w sposób intuicyjny, co umożliwia łatwe wykonywanie podstawowych operacji. Na przykład, wyświetlanie macierzy jest

realizowane za pomocą przeciążonego operatora `<<`, co pozwala na użycie standardowego strumienia wyjściowego: Przykład wyświetlania zawartości macierzy w C++[4].

```
matrix m(4);
std::cout << m;
```

Listing 3. Przykład wyświetlania zawartości macierzy w C++

Podsumowanie

Projekt klasy `matrix` opiera się na elastycznym i modułowym podejściu. Zastosowanie dynamicznej alokacji pamięci oraz przeciążenia operatorów sprawia, że klasa ta jest zarówno wydajna, jak i łatwa w użyciu. Dzięki zaprojektowanym metodom możliwe jest wygodne przeprowadzanie operacji na macierzach, co czyni ją odpowiednią do zastosowań w różnych dziedzinach obliczeń numerycznych.

[4]Przykład wyświetlania zawartości macierzy w C++[1].

4. Implementacja

Implementacja klasy `matrix` obejmuje szczegółową realizację metod, operatorów oraz zarządzania pamięcią. W tym rozdziale opisano kluczowe aspekty implementacji, zapewniając wgląd w mechanizmy stojące za poszczególnymi funkcjonalnościami.

4.1. Struktura danych

Podstawową strukturą danych w klasie `matrix` jest dynamicznie alokowana tablica dwuwymiarowa: Deklaracja wskaźnika na dane macierzy [5].

```
int** data;
```

Listing 4. Deklaracja wskaźnika na dane macierzy

Pamięć dla tej tablicy jest alokowana przez metodę prywatną `allocateMemory(int size)`: Funkcja alokująca pamięć dla macierzy kwadratowej[6].

```
void matrix::allocateMemory(int size) {
    data = new int*[size];
    for (int i = 0; i < size; ++i) {
        data[i] = new int[size]();
    }
}
```

Listing 5. Funkcja alokująca pamięć dla macierzy kwadratowej

Każdy wiersz jest alokowany indywidualnie, co pozwala na łatwe zarządzanie pamięcią.

4.2. Konstruktory

Klasa `matrix` obsługuje trzy główne typy konstruktorów: Domyślny konstruktor klasy matrix [7].

- Konstruktor domyślny:

```
matrix::matrix() : n(0), data(nullptr) {}

```

Listing 6. Domyślny konstruktor klasy matrix

Tworzy pustą macierz bez alokacji pamięci.

[5] Deklaracja wskaźnika na dane macierzy[1].
[6] Funkcja alokująca pamięć dla macierzy kwadratowej[1].
[7] Domyślny konstruktor klasy matrix[1].

- Konstruktor parametryczny: Inicjalizacja macierzy na podstawie tablicy jednowymiarowej [8].

```
matrix::matrix(int n, int* t) : n(n) {
    allocateMemory(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            data[i][j] = t[i * n + j];
}

```

Listing 7. Inicjalizacja macierzy na podstawie tablicy jednowymiarowej

Pozwala na inicjalizację macierzy o zadanym rozmiarze n oraz wypełnienie jej wartościami z tablicy `t`. Konstruktor kopiujący klasy matrix [9].

- Konstruktor kopiujący:

```
matrix::matrix(const matrix& m) : n(m.n) {
    allocateMemory(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            data[i][j] = m.data[i][j];
}

```

Listing 8. Konstruktor kopiujący klasy matrix

Umożliwia utworzenie kopii istniejącej macierzy.

4.3. Zarządzanie pamięcią

Destruktor klasy `matrix` dba o zwolnienie pamięci alokowanej dynamicznie: Destruktor klasy matrix [10].

```
matrix::~matrix() {
    if (data) {
        for (int i = 0; i < n; ++i)
            delete[] data[i];
        delete[] data;
    }
}
```

Listing 9. Destruktor klasy matrix

[8] Inicjalizacja macierzy na podstawie tablicy jednowymiarowej[1].
[9] Konstruktor kopiujący klasy matrix[1].
[10] Destruktor klasy matrix[1].

Dzięki temu unika się wycieków pamięci.

4.4. Implementacja metod

Transpozycja macierzy

Funkcja `dowroc()` realizuje transpozycję macierzy, wykorzystując tymczasową kopię: Implementacja metody transpozycji macierzy [11].

```
matrix& matrix::dowroc() {
    matrix temp(*this);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            data[i][j] = temp.data[j][i];
    return *this;
}
```

Listing 10. Implementacja metody transpozycji macierzy

Losowanie wartości

Funkcja `losuj()` wypełnia całą macierz losowymi wartościami: Implementacja metody losowego wypełniania macierzy [12].

```
matrix& matrix::losuj() {
    std::srand(std::time(nullptr));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            data[i][j] = std::rand() % 10;
    return *this;
}
```

Listing 11. Implementacja metody losowego wypełniania macierzy

Modyfikacja elementów

Funkcja `wstaw()` pozwala na zmianę wartości konkretnego elementu: Implementacja metody wstawiania wartości do macierzy [13].

```
matrix& matrix::wstaw(int x, int y, int wartosc) {
    if (x >= 0 && x < n && y >= 0 && y < n) {
        data[x][y] = wartosc;
```

[11] Implementacja metody transpozycji macierzy[1].
[12] Implementacja metody losowego wypełniania macierzy[1].
[13] Implementacja metody wstawiania wartości do macierzy[1].

```
    }
    return *this;
}
```

Listing 12. Implementacja metody wstawiania wartości do macierzy

4.5. Implementacja operatorów

Dodawanie macierzy

Operator + został zaimplementowany jako metoda przyjazna: Przeciążenie operatora + dla macierzy[14].

```
matrix operator+(const matrix& m1, const matrix& m2) {
    matrix result(m1.n);
    for (int i = 0; i < m1.n; ++i)
        for (int j = 0; j < m1.n; ++j)
            result.data[i][j] = m1.data[i][j] + m2.data[i][j];
    return result;
}
```

Listing 13. Przeciążenie operatora + dla macierzy

Porównywanie macierzy

Operator == umożliwia porównywanie dwóch macierzy: Przeciążenie operatora [15].

```
bool matrix::operator==(const matrix& m) const {
    if (n != m.n) return false;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (data[i][j] != m.data[i][j])
                return false;
    return true;
}
```

Listing 14. Przeciążenie operatora

4.6. Testowanie i debugowanie

Dla zapewnienia poprawności implementacji zaimplementowano zestaw testów jednostkowych, sprawdzających:

[14]Przeciążenie operatora + dla macierzy[1].
[15]Przeciążenie operatora[1].

- Poprawność operacji arytmetycznych na macierzach.
- Działanie funkcji losowania i transpozycji.
- Poprawność alokacji i zwalniania pamięci.

Przykład testu: Przeciążenie operatora [16].

```
matrix m1(2, new int[4]{1, 2, 3, 4});
matrix m2(2, new int[4]{5, 6, 7, 8});
matrix m3 = m1 + m2;
std::cout << m3;
```

Listing 15. Przeciążenie operatora

Podsumowanie

Implementacja klasy `matrix` została wykonana zgodnie z założeniami projektowymi. Wykorzystano dynamiczną alokację pamięci, co umożliwia elastyczne zarządzanie rozmiarem macierzy. Zaimplementowane metody i operatory zapewniają pełną funkcjonalność, a mechanizmy obsługi błędów minimalizują ryzyko nieprawidłowego działania programu.

[16] Przeciążenie operatora[1].

5. Wnioski

Przeprowadzenie implementacji klasy `matrix` pozwoliło na dogłębne zrozumienie problematyki operacji na macierzach, zarządzania pamięcią dynamiczną oraz obsługi struktur danych. W tym rozdziale przedstawiono podsumowanie osiągniętych wyników, analizę mocnych i słabych stron projektu oraz propozycje przyszłych usprawnień.

5.1. Podsumowanie osiągnięć

Realizacja projektu przyniosła następujące rezultaty:

- Opracowano klasę `matrix`, która umożliwia dynamiczne zarządzanie macierzami kwadratowymi o dowolnym rozmiarze.
- Zaimplementowano szeroki zakres operacji, takich jak:
  - dodawanie i mnożenie macierzy,
  - transpozycja,
  - wypełnianie losowymi wartościami,
  - manipulowanie pojedynczymi elementami.
- Udało się zapewnić wydajne zarządzanie pamięcią poprzez dynamiczną alokację i zwalnianie zasobów.
- Przeprowadzono wstępne testy funkcjonalności, które potwierdziły poprawność kluczowych operacji.

5.2. Mocne strony rozwiązania

Podczas realizacji projektu zauważono kilka istotnych zalet:

- Elastyczność klasy – możliwość dostosowania rozmiaru macierzy oraz wypełnienia jej różnymi wartościami.
- Intuicyjna i przejrzysta implementacja operatorów pozwala na łatwe manipulowanie macierzami.
- Modularność kodu – podział na logiczne funkcje i metody pozwala na łatwe wprowadzanie zmian i rozszerzeń.
- Zgodność z zasadami programowania obiektowego, co zwiększa możliwość ponownego wykorzystania kodu w przyszłych projektach.

5.3. Wyzwania i ograniczenia

Mimo licznych osiągnięć, w trakcie realizacji napotkano pewne wyzwania:

- Ograniczenie klasy do pracy jedynie z macierzami kwadratowymi – w przyszłości warto rozważyć wsparcie dla macierzy prostokątnych.
- Brak rozbudowanego systemu obsługi błędów – obecnie nie są zgłaszane wyjątki w przypadku niepoprawnych operacji.
- Zależność od dynamicznej alokacji pamięci może prowadzić do potencjalnych problemów z wydajnością w przypadku dużych macierzy.

5.4. Propozycje usprawnień

Na podstawie przeprowadzonej analizy sugeruje się następujące kierunki rozwoju:

- Rozszerzenie funkcjonalności o obsługę macierzy prostokątnych.
- Dodanie metod do zapisywania i odczytywania macierzy z plików, co zwiększy ich praktyczne zastosowanie.
- Wprowadzenie bardziej zaawansowanych algorytmów, takich jak:
  - obliczanie wyznacznika,
  - eliminacja Gaussa,
  - dekompozycja LU.
- Implementacja rozbudowanego systemu logowania i obsługi błędów, który pozwoli na szybsze diagnozowanie problemów.
- Optymalizacja pamięci – wykorzystanie jednowymiarowych tablic zamiast tablic dynamicznych.

5.5. Wnioski końcowe

Realizacja projektu klasy `matrix` była wartościowym doświadczeniem, które umożliwiło zdobycie praktycznych umiejętności w programowaniu obiektowym i zarządzaniu pamięcią. Pomimo pewnych ograniczeń, stworzona implementacja stanowi solidną bazę do dalszych prac nad bardziej zaawansowanymi systemami operującymi na macierzach. Kluczowe aspekty, które wymagają dalszej pracy, to rozszerzenie funkcjonalności oraz poprawa wydajności. Projekt ten stanowi fundament dla bardziej złożonych rozwiązań matematycznych i numerycznych w przyszłości.

Bibliografia

[1] *Strona internetowa WikipediA*. URL: https://pl.wikipedia.org/wiki/Wikipedia:Strona_g%C5%82%C3%B3wna.

[2] *Strona z rysunkiem UML*. URL: https://www.google.com/search?q=uml+diagram+of+a+matrix+class&oq=&gs_lcrp=EgZjaHJvbWUqCQgGECMYJxjqAjIJCAAQIxgnGOoCMgkI
sourceid=chrome&ie=UTF-8.

Spis rysunków

Spis tabel

Spis listingów