

AKADEMIA NAUK STOSOWANYCH W NOWYM SĄCZU

Wydział Nauk Inżynierskich
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA ZAAWANSOWANE PROGRAMOWANIE

Operacje na macierzy z zastosowaniem GitHub Copilot

Autor:
Gabriela Bugajska
Wiktoria Fałowska

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań	3
2. Analiza problemu	4
2.1. Zastosowanie Macierzy	4
2.2. Działanie programu	4
2.2.1. Plik <code>main.cpp</code>	5
2.2.2. Plik <code>matrix.h</code>	6
2.2.3. Plik <code>matrix.cpp</code>	7
2.3. Narzędzie GitHub Copilot	8
3. Projektowanie	11
3.1. Narzędzia używane w projekcie	11
3.2. UML	12
4. Implementacja	13
4.1. Implementacja klasy <code>matrix c++</code>	13
4.2. Ciekawe fragmenty kodu oraz szczegóły implementacji.	13
4.2.1. Klasa <code>matrix</code>	13
4.2.2. Kluczowe elementy implementacji	15
4.2.3. Podsumowanie	18
4.3. Inicjalizacja w GIT	19
4.3.1. Praca z systemem kontroli wersji GIT	19
4.3.2. Doxygen	20
4.3.3. Wdrożenie Doxygena	20
5. Wnioski	22
Literatura	23
Spis rysunków	24
Spis tabel	25
Spis listingów	26

1. Ogólne określenie wymagań

Projekt zakłada stworzenie klasy ‘Matrix’ obsługującej macierze kwadratowe $n \times n$, gdzie n to rozmiar macierzy, przechowywanej w pamięci dynamicznej.

Klasa będzie posiadać różne typy konstruktorów: domyślny, z wymiarem n , z danymi wejściowymi oraz konstruktor kopiujący. W ramach zarządzania pamięcią zaimplementujemy dynamiczną alokację, realokację i kontrolę poprawności rozmiaru. Destruktor zwolni zaalokowaną pamięć.

Funkcjonalności klasy obejmą m.in. wstawianie i odczyt wartości, transpozycję, losowe wypełnianie, tworzenie macierzy diagonalnych, kolumnowych, wierszowych oraz wzorców (np. szachownica). Klasa obsłuży operacje matematyczne (dodawanie, mnożenie, odejmowanie) oraz porównania między macierzami. Zostaną przeciążone operatory takie jak $+$, $-$, $*$, $==$, $<$, $>$, a także inkrementacja, dekrementacja i wyświetlanie macierzy.

W programie głównym przetestujemy wszystkie metody, wykorzystując dane wczytywane z plików. Zapewnimy zgodność wymiarów macierzy przy operacjach matematycznych, aby uniknąć błędów.

Projekt wymaga dokumentacji w LaTeX i automatycznego generowania opisów kodu za pomocą Doxygen. Kod i dokumentacja będą zarządzane w repozytorium na GitHubie, umożliwiając równoległą pracę dwóch osób.

W dokumentacji uwzględnimy doświadczenia z GitHub Copilot – jego zalety, napotkane trudności i błędy, z omówieniem sytuacji, w których okazał się pomocny. Projekt rozwija umiejętności zarządzania pamięcią dynamiczną, pracy z dużymi strukturami danych oraz nowoczesnych narzędzi programistycznych.

2. Analiza problemu

Operacje na macierzach¹ są podstawą wielu dziedzin, takich jak analiza danych, algorytmy grafowe czy modelowanie matematyczne. Wykorzystuje się je do reprezentacji i manipulacji dużych zbiorów danych w sposób uporządkowany i efektywny.

Projekt koncentruje się na implementacji klasy ‘matrix’, która umożliwia łatwe tworzenie i przetwarzanie macierzy kwadratowych. Klasa obsługuje operacje takie jak dodawanie, mnożenie, transpozycja czy manipulacja przekątnymi, zapewniając jednocześnie prostotę użytkowania i wydajność.

Celem jest dostarczenie narzędzia wszechstronnego, wydajnego i intuicyjnego, które znajdzie zastosowanie w obliczeniach numerycznych, przetwarzaniu obrazów czy symulacjach komputerowych.

2.1. Zastosowanie Macierzy

Macierze² są kluczowym narzędziem w wielu dziedzinach nauki i technologii, oferując sposób na reprezentowanie i przetwarzanie danych w sposób uporządkowany. W matematyce stosowanej macierze odgrywają centralną rolę w algebrze liniowej, umożliwiając rozwiązywanie układów równań liniowych, analizowanie transformacji liniowych czy modelowanie zjawisk fizycznych.

W informatyce macierze są powszechnie wykorzystywane do reprezentacji grafów w postaci macierzy sąsiedztwa, co pozwala na efektywne przeszukiwanie grafów oraz analizę relacji między obiektami. Są również podstawą wielu algorytmów uczenia maszynowego i sztucznej inteligencji, gdzie operacje na macierzach pozwalają na przetwarzanie dużych zbiorów danych, np. w sieciach neuronowych.

2.2. Działanie programu

Działanie programu opiera się na współpracy trzech głównych plików: `matrix.h`, `matrix.cpp` oraz `github.cpp`. Plik `matrix.h` definiuje strukturę klasy `matrix` oraz deklaruje jej metody. Plik `matrix.cpp` zawiera implementację tych metod, umożliwiając wykonywanie operacji na macierzach. Plik `github.cpp` natomiast demonstruje działanie programu poprzez serię testów, które ilustrują zastosowanie poszczególnych funkcji i operacji na macierzach. Poniżej przedstawiono szczegółowy opis funkcjonalności każdego pliku.

¹Źródło: [1]

²Źródło: [2]

2.2.1. Plik `main.cpp`

Plik `github.cpp` pełni rolę głównej części programu, zawierającej funkcję `main`. Jego głównym celem jest testowanie funkcjonalności klasy `matrix` oraz demonstracja możliwości implementacji na konkretnych przykładach. Struktura pliku została podzielona na logiczne sekcje, z których każda realizuje test konkretnej funkcjonalności. Dzięki zastosowanym komentarzom kod jest przejrzysty i łatwy do analizy.

Struktura pliku

Na początku pliku znajdują się dyrektywy `#include`, które włączają plik nagłówkowy `matrix.h` oraz niezbędne biblioteki standardowe, takie jak `<iostream>`. Następnie funkcja `main` jest podzielona na serie testów, z których każdy ilustruje wybraną funkcję klasy `matrix`.

Opis testów

- **Test 1: Tworzenie macierzy i losowe wypełnienie**

Tworzona jest macierz o zadanym rozmiarze (np. 3x3), która zostaje losowo wypełniona wartościami przy użyciu metody `losuj`. Wynikowa macierz jest następnie wyświetlana.

- **Test 2: Wstawianie i pobieranie wartości**

W tym teście pokazano, jak wstawiać wartości do wybranych komórek macierzy przy użyciu metody `wstaw` oraz odczytywać je za pomocą metody `pokaz`.

- **Test 3: Odwracanie macierzy (transpozycja)**

Demonstruje transpozycję macierzy za pomocą metody `odwroc`. Wyświetlone są macierz przed i po transpozycji.

- **Test 4: Wypełnianie przekątnej**

Wypełnia główną przekątną macierzy wartościami z tablicy przy pomocy metody `diagonalna`.

- **Test 5: Wypełnianie kolumny i wiersza**

Prezentuje wypełnianie wybranych wierszy i kolumn wartościami z tablic przy użyciu metod `wiersz` i `kolumna`.

- **Test 6: Wyświetlanie przekątnej, nadprzekątnej i podprzekątnej**

W teście pokazano sposób wyświetlania wartości znajdujących się na głównej przekątnej (`przekatna`), nad nią (`nad_przekatna`) oraz pod nią (`pod_przekatna`).

- **Test 7: Operacje arytmetyczne na macierzach**

Test prezentuje działanie przeciążonych operatorów, takich jak dodawanie (+), mnożenie (*) oraz operacje skalowania przez liczbę.

- **Test 8: Operatory porównania**

Sprawdzone są operacje porównania macierzy, takie jak równość (==), większość (>) i mniejszość (<).

- **Test 9: Operatory inkrementacji i dekrementacji**

W teście tym przedstawiono użycie operatorów ++ i --, które odpowiednio zwiększają i zmniejszają wartości wszystkich elementów macierzy.

- **Test 10: Generowanie wzoru szachownicy**

Ostateczny test ilustruje generowanie macierzy w formie szachownicy za pomocą metody `szachownica`, gdzie pola czarne są zastąpione zerami.

Każdy z testów pokazuje działanie jednej lub kilku metod klasy `matrix`, co pozwala na dokładne sprawdzenie funkcjonalności programu oraz jego poprawności.

2.2.2. Plik `matrix.h`

Plik `matrix.h` jest plikiem nagłówkowym, który definiuje klasę `matrix`. Klasa ta reprezentuje macierz kwadratową i zawiera metody pozwalające na wykonywanie operacji matematycznych, logicznych oraz manipulacyjnych na macierzach. Plik ten stanowi deklarację struktury i funkcji, które są zaimplementowane w pliku `matrix.cpp`.

Struktura klasy `matrix`

Klasa `matrix` zawiera dwa główne elementy:

- **Atrybuty prywatne:**

- `int n` – Rozmiar macierzy ($n \times n$).
- `int** data` – Wskaźnik na dynamicznie alokowaną tablicę dwuwymiarową, przechowującą dane macierzy.

- **Metody publiczne:** Klasa oferuje bogaty zestaw metod do tworzenia, modyfikacji i operacji na macierzach, w tym:

- **Konstruktory i destruktor** – Umożliwiają tworzenie macierzy (pustych, z wartościami lub kopiowanych) oraz zwalnianie pamięci.
- **Metody manipulacyjne** – Takie jak `wstaw`, `pokaz`, `alokuj`.

- **Metody matematyczne** – Obejmują operacje dodawania, mnożenia, transpozycji czy skalowania.
- **Operacje logiczne** – Porównywanie macierzy pod względem równości, większości i mniejszości.
- **Operatory przeciążone** – Przeciążone operatory umożliwiają intuicyjne wykonywanie działań matematycznych na macierzach.
- **Specjalne funkcje wyświetlania** – Wyświetlanie elementów macierzy, takich jak przekątna, elementy nad i pod przekątną oraz macierz w formie szachownicy.

Plik `matrix.h` pełni kluczową rolę w strukturze programu, zapewniając czytelną i intuicyjną deklarację funkcjonalności klasy `matrix`. Dzięki niemu możliwe jest łatwe zarządzanie oraz rozbudowa programu o dodatkowe funkcje w przyszłości.

2.2.3. Plik `matrix.cpp`

Plik `matrix.cpp` zawiera implementację metod i funkcji zadeklarowanych w pliku nagłówkowym `matrix.h`. Odpowiada za rzeczywiste działanie klasy `matrix`, umożliwiając operacje na macierzach kwadratowych, takie jak wstawianie wartości, wykonywanie działań matematycznych oraz manipulacje strukturą macierzy.

Struktura pliku

Plik `matrix.cpp` zaczyna się od włączenia pliku nagłówkowego `matrix.h` oraz potrzebnych bibliotek standardowych, takich jak `<iostream>`, `<ctime>` i `<cstdlib>`. Następnie zdefiniowane są wszystkie metody klasy `matrix`.

Główne funkcjonalności:

- **Konstruktory i destruktor:**

Plik implementuje różne sposoby tworzenia obiektów klasy `matrix`, takie jak:

- Konstruktor domyślny – Tworzy pustą macierz.
- Konstruktor parametryczny – Tworzy macierz o zadanym rozmiarze $n \times n$.
- Konstruktor kopiujący – Kopiuje dane z innej macierzy.
- Destruktor – Zwalnia dynamicznie alokowaną pamięć.

- **Metody manipulacyjne:**

Plik zawiera implementacje metod pozwalających na manipulowanie strukturą macierzy:

- `wstaw(int x, int y, int wartosc)` – Wstawia wartość do określonej komórki macierzy.
- `pokaz(int x, int y)` – Zwraca wartość znajdującą się w określonej pozycji macierzy.
- `alokuj(int size)` – Dynamicznie alokuje pamięć dla macierzy o zadanym rozmiarze.

- **Operacje matematyczne:**

Klasa obsługuje różnorodne operacje matematyczne na macierzach, w tym:

- Dodawanie i mnożenie macierzy (`operator+`, `operator*`).
- Skalowanie macierzy przez liczbę (`operator*`, `operator+` dla liczb).
- Transpozycję macierzy za pomocą metody `odwroc()`.

- **Specjalne metody wyświetlania:**

Plik implementuje metody pozwalające na wizualizację specyficznych elementów macierzy, takich jak:

- `przekatna()` – Wyświetla wartości na głównej przekątnej macierzy.
- `pod_przekatna()` – Wyświetla wartości poniżej głównej przekątnej.
- `nad_przekatna()` – Wyświetla wartości powyżej głównej przekątnej.
- `szachownica()` – Generuje wzór szachownicy na podstawie macierzy.

- **Operatory przeciążone:**

Klasa wykorzystuje przeciążenie operatorów, takich jak:

- `operator+`, `operator*`, `operator-` – Działania arytmetyczne.
- `operator==`, `operator>`, `operator<` – Operacje porównania.
- `operator++`, `operator--` – Inkrementacja i dekrementacja elementów macierzy.

Plik `matrix.cpp` jest kluczowym elementem programu, odpowiadającym za realizację logiki operacji na macierzach i umożliwiającym praktyczne zastosowanie klasy `matrix`.

2.3. Narzędzie GitHub Copilot

GitHub Copilot³ to zaawansowane narzędzie wspomagające programistów w pisanii kodu. Działa jako inteligentny asystent, który na podstawie kontekstu kodu oraz

³Źródło: [3]

komentarzy potrafi generować fragmenty, funkcje, a nawet całe klasy. Narzędzie to wykorzystuje modele sztucznej inteligencji oparte na technologii OpenAI, umożliwiając znaczące przyspieszenie procesu programowania.

Czy GitHub Copilot jest potrzebny?

GitHub Copilot jest szczególnie przydatny w projektach wymagających intensywnej pracy programistycznej, ponieważ:

- Przyspiesza proces kodowania, sugerując gotowe rozwiązania.
- Pomaga w unikaniu błędów składniowych i logicznych.
- Oferuje wsparcie w nauce nowych technologii i frameworków dzięki podpowiedzom.
- Redukuje czas poświęcony na wyszukiwanie przykładów w dokumentacji czy w internecie.

Jednak skuteczność Copilota zależy od doświadczenia użytkownika i stopnia zrozumienia kodu. Narzędzie nie zawsze generuje optymalne rozwiązania, dlatego jego propozycje wymagają weryfikacji.

Rysunek 2.1 przedstawia rozwiązanie błędu natrafionego podczas pisania kodu, które zaproponował Copilot.

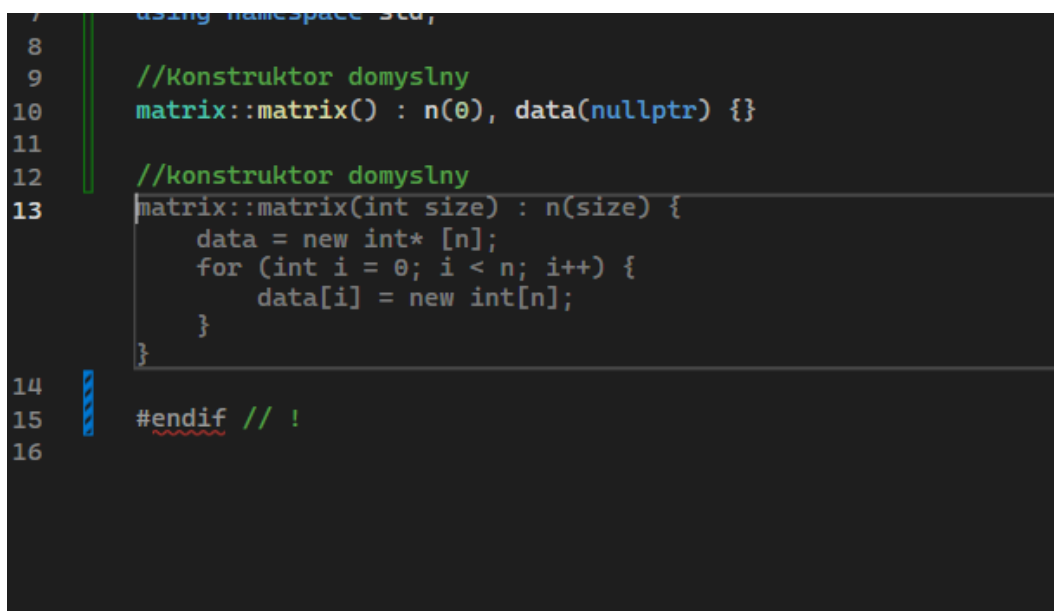


Rys. 2.1. Przykład działania Copilota

Czy GitHub Copilot jest warty używania?

GitHub Copilot radzi sobie doskonale w przypadku powtarzalnych lub standardowych zadań programistycznych takich jak na przykładowym zrzucie ekranu 2.2. Generowane przez niego fragmenty kodu są zwykle poprawne i zgodne z najlepszymi praktykami programowania. Jednak w przypadku bardziej zaawansowanych problemów, wymagających głębszego zrozumienia specyfiki projektu, Copilot może nie być wystarczający. W takich sytuacjach jego propozycje mogą wymagać znacznej modyfikacji.

Podsumowując, GitHub Copilot to potężne narzędzie, które może znacząco wspierać programistów w codziennej pracy. Mimo swoich ograniczeń, jego zastosowanie pozwala oszczędzić czas i zwiększyć produktywność, szczególnie w standardowych zadaniach programistycznych.

A screenshot of a C++ code editor showing a class named 'matrix'. Line 9 has a comment '//Konstruktor domyslny' and line 10 has the code 'matrix::matrix() : n(0), data(nullptr) {}'. Line 12 has another comment '//konstruktor domyslny'. Line 13 has the code 'matrix::matrix(int size) : n(size) {' followed by an indented block of code: 'data = new int* [n];', 'for (int i = 0; i < n; i++) {' followed by an indented line 'data[i] = new int[n];', and closing braces. Line 15 has the code '#endif // !'. A blue vertical bar on the left side of the editor indicates the current cursor position. The code is color-coded: comments are green, keywords are blue, and identifiers are white.

Rys. 2.2. Uzupełnienia Copilota

3. Projektowanie

3.1. Narzędzia używane w projekcie

W projekcie wykorzystaliśmy kilka narzędzi, które pomogły w realizacji kodu i dokumentacji. Do pisania dokumentacji użyliśmy LaTeX, który pozwala na tworzenie profesjonalnych raportów i dokumentów technicznych. Dzięki niemu mogliśmy estetycznie i przejrzysto opisać nasz projekt, dodając wzory matematyczne i schematy.

Kod źródłowy w języku C++ tworzyliśmy w Visual Studio 2022, które jest bardzo wygodnym środowiskiem programistycznym. Ułatwia ono pisanie kodu, oferuje podpowiedzi składni i narzędzia do testowania oraz debugowania, co znacząco przyspieszyło naszą pracę.

W projekcie korzystaliśmy także z GitHub Copilot, który uzyskaliśmy dzięki naszej legitymacji studenckiej. To narzędzie wspomagane sztuczną inteligencją podpowiadało fragmenty kodu w czasie rzeczywistym, co ułatwiło pisanie funkcji i rutynowych elementów programu. Choć wymagało sprawdzania, przyspieszyło naszą pracę.

Całość projektu zarządzana była za pomocą GitHub, co pozwoliło nam przechowywać kod, śledzić zmiany i pracować nad projektem równolegle. GitHub zapewnił nam bezpieczeństwo danych i łatwy dostęp do projektu, co było szczególnie przydatne podczas pracy zespołowej. Dzięki tym narzędziom projekt został zrealizowany sprawnie i efektywnie.

3.2. UML

<<class>> matrix
- n: int - data: int**
+ matrix() + matrix(size: int) + matrix(m: const matrix&) + matrix(size: int, t: int*) + ~matrix() + alokuj(size: int): matrix& + wstaw(x: int, y: int, wartosc: int): matrix& + pokaz(x: int, y: int): int + odwroc(): matrix& + losuj(): matrix& + losuj(x: int): matrix& + diagonalna(t: int*): matrix& + diagonalna_k(k: int, t: int*): matrix& + kolumna(x: int, t: int*): matrix& + wiersz(y: int, t: int*): matrix& + przekatna(): matrix& + pod_przekatna(): matrix& + nad_przekatna(): matrix& + szachownica(): matrix& + operator+(m: matrix&): matrix& + operator*(m: matrix&): matrix& + operator+(a: int): matrix& + operator*(a: int): matrix& + operator-(a: int): matrix& + operator+=(a: int): matrix& + operator-=(a: int): matrix& + operator*=(a: int): matrix& + operator==(m: const matrix&): bool + operator>(m: const matrix&): bool + operator<(m: const matrix&): bool + operator++(int): matrix& + operator--(int): matrix& + operator<<(o: ostream&, m: const matrix&): ostream& + operator+(a: int, m: matrix&): matrix + operator*(a: int, m: matrix&): matrix + operator-(a: int, m: matrix&): matrix

Rys. 3.1. UML Macierz

4. Implementacja

4.1. Implementacja klasy `matrix c++`

Projekt opiera się na stworzeniu klasy `matrix`, która pozwala na reprezentację oraz wykonywanie operacji matematycznych na macierzach kwadratowych. Klasa została zaprojektowana w sposób uniwersalny, z uwzględnieniem zarządzania pamięcią dynamiczną i przeciążania operatorów, co czyni jej wykorzystanie intuicyjnym i wygodnym dla użytkownika. Implementacja jest podzielona na pliki nagłówkowe (`matrix.h`) i pliki z definicjami metod (`matrix.cpp`), co sprzyja czytelności i modularności kodu. Dodatkowo, funkcja `main` została zaprojektowana tak, aby przeprowadzać szczegółowe testy funkcji i operatorów zaimplementowanych w klasie.

4.2. Ciekawe fragmenty kodu oraz szczegóły implementacji.

4.2.1. Klasa `matrix`

Klasa `matrix` definiuje podstawowe struktury i metody pozwalające na operacje na macierzach kwadratowych. Głównymi składnikami klasy widocznej na listingu 1 (s. 14) są:

- Pola prywatne:
 - `int n` — przechowuje rozmiar macierzy $n \times n$
 - `int** data` — dynamicznie alokowana tablica wskaźników przechowująca elementy macierzy.
- Konstruktory i destruktor:

Klasa implementuje cztery różne konstruktory:

 - Konstruktor domyślny, który tworzy pustą macierz
 - Konstruktor parametryczny, który alokuje pamięć dla macierzy o zadanym rozmiarze.
 - Konstruktor kopiujący, który umożliwia skopiowanie istniejącej macierzy do nowego obiektu.
 - Konstruktor inicjalizujący macierz z tablicy, który pozwala na szybkie zainicjowanie macierzy wartościami.

Destruktor zapewnia poprawne zwolnienie pamięci, co zapobiega wyciekom pamięci.

```
1 #pragma once
2 #ifndef MATRIX_H
3 #define MATRIX_H
4
5 #include <iostream>
6 using namespace std;
7
8 class matrix {
9 private:
10     int n;
11     int** data;
12
13 public:
14     matrix();
15     matrix(int size);
16     matrix(const matrix& m);
17     matrix(int size, int* t);
18     ~matrix();
19
20     matrix& alokuj(int size);
21     matrix& wstaw(int x, int y, int wartosc);
22     int pokaz(int x, int y);
23     matrix& odwroc();
24     matrix& losuj();
25     matrix& losuj(int x);
26     matrix& diagonalna(int* t);
27     matrix& diagonalna_k(int k, int* t);
28     matrix& kolumna(int x, int* t);
29     matrix& wiersz(int y, int* t);
30     matrix& przekatna();
31     matrix& pod_przekatna();
32     matrix& nad_przekatna();
33     matrix& szachownica();
34
35     matrix& operator+(matrix& m);
36     matrix& operator*(matrix& m);
37     matrix& operator+(int a);
38     matrix& operator*(int a);
39     matrix& operator-(int a);
40     friend matrix operator+(int a, matrix& m);
41     friend matrix operator*(int a, matrix& m);
42     friend matrix operator-(int a, matrix& m);
43     matrix& operator++(int);
44     matrix& operator--(int);
45     matrix& operator+=(int a);
```

```

46     matrix& operator--=(int a);
47     matrix& operator*=(int a);
48     bool operator==(const matrix& m) const;
49     bool operator>(const matrix& m) const;
50     bool operator<(const matrix& m) const;
51     friend ostream& operator<<(ostream& o, const matrix& m);
52 };
53
54 #endif
55
56 }

```

Listing 1. plik matrix.h

4.2.2. Kluczowe elementy implementacji

1. Konstruktor parametryczny i destruktor.

Konstruktor parametryczny pozwala na dynamiczne tworzenie macierzy o zadanym rozmiarze. Kod inicjalizuje wskaźnik data jako dynamiczną tablicę wskaźników, które z kolei wskazują na kolejne wiersze macierzy. Destruktor zwalnia dynamicznie zaalokowaną pamięć, zapewniając, że nie dochodzi do wycieków. Oba elementy są widoczne na listingu 2.

```

1 matrix::matrix(int size) : n(size) {
2     data = new int* [n];
3     for (int i = 0; i < n; i++) {
4         data[i] = new int[n];
5     }
6 }
7
8 matrix::~~matrix() {
9     for (int i = 0; i < n; i++) {
10         delete[] data[i];
11     }
12     delete[] data;
13 }

```

Listing 2. Konstruktor i destruktor

2. Zarządzanie pamięcią dynamiczną

Funkcja alokuj widoczna na listingu 3 pozwala na alokację lub realokację pamięci dla macierzy. Jeśli pamięć jest już zajęta, funkcja zwalnia ją przed przydzieleniem nowej.

```
1 matrix& matrix::alokuj(int size) {
2     if (data) {
3         for (int i = 0; i < n; i++) {
4             delete[] data[i];
5         }
6         delete[] data;
7     }
8     n = size;
9     data = new int* [n];
10    for (int i = 0; i < n; i++) {
11        data[i] = new int[n]();
12    }
13    return *this;
14 }
15
16 }
```

Listing 3. Funkcja alokuj

3. Losowe wypełnianie macierzy

Funkcja losuj widoczna na listingu 4 wypełnia macierz losowymi liczbami w zakresie 0–9. Jest to przydatne narzędzie do testowania funkcji.

```
1 matrix& matrix::losuj() {
2     srand(time(NULL));
3     for (int i = 0; i < n; i++) {
4         for (int j = 0; j < n; j++) {
5             data[i][j] = rand() % 10;
6         }
7     }
8     return *this;
9 }
```

Listing 4. Funkcja losuj

4. Przeciążanie operatorów

Przeciążone operatory, takie jak + i *, umożliwiają łatwe wykonywanie operacji na macierzach, takich jak dodawanie i mnożenie. Wykorzystanie operatorów w naszym kodzie widczne jest na listingu 5.

```
1 matrix& matrix::operator+(matrix& m) {
2     matrix* wynik = new matrix(n);
```



```

3   for (int i = 0; i < n; i++) {
4       for (int j = 0; j < n; j++) {
5           wynik->data[i][j] = data[i][j] + m.data[i][j];
6       }
7   }
8   return *wynik;
9 }
10
11 matrix& matrix::operator*(matrix& m) {
12     matrix* wynik = new matrix(n);
13     for (int i = 0; i < n; i++) {
14         for (int j = 0; j < n; j++) {
15             wynik->data[i][j] = data[i][j] * m.data[i][j];
16         }
17     }
18     return *wynik;
19 }

```

Listing 5. operator + oraz *

5. Transpozycja macierzy

Funkcja odwróć widoczna na listingu 6 zamienia wiersze z kolumnami, realizując transpozycję macierzy.

```

1 matrix& matrix::odwroc() {
2     for (int i = 0; i < n; i++) {
3         for (int j = 0; j < i; j++) {
4             swap(data[i][j], data[j][i]);
5         }
6     }
7     return *this;
8 }

```

Listing 6. Funkcja odwróć

6. Funkcja main – testowanie

W funkcji main widocznej na listingu 7 testowane są wszystkie kluczowe funkcje klasy wumienione w naszej dokumentacji. Przykładowo, tworzona jest macierz, wypełniana losowymi wartościami, a następnie wykonywane są operacje takie jak transpozycja, dodawanie wartości do przekątnej czy wstawianie elementów w określone pozycje.

```

1 int main() {
2     matrix m1(3);
3     m1.losuj();
4     cout << "Macierz losowa:" << endl << m1 << endl;

```

```
5
6  m1.wstaw(0, 0, 5);
7  cout << "Po wstawieniu 5 na (0, 0):" << endl << m1 << endl;
8
9  m1.odwroc();
10 cout << "Po odwróceniu macierzy:" << endl << m1 << endl;
11
12 int diag[] = { 1, 2, 3 };
13 m1.diagonalna(diag);
14 cout << "Macierz z przekatna [1, 2, 3]:" << endl << m1 << endl;
15
16 return 0;
17 }
```

Listing 7. Kod main (wybrane części)

4.2.3. Podsumowanie

Klasa `matrix` została zaprojektowana jako elastyczne narzędzie do pracy z macierzami kwadratowymi. Dzięki dynamicznemu zarządzaniu pamięcią i przeciążeniu operatorów, implementacja umożliwia wygodne operowanie na macierzach różnej wielkości. Dodatkowo, dokładne testy w funkcji `main` gwarantują poprawność działania wszystkich funkcji. Całość kodu jest przejrzysta i przyjemna w odczycie.

4.3. Inicjalizacja w GIT

4.3.1. Praca z systemem kontroli wersji GIT

Praca z systemem kontroli wersji rozpoczęła się od utworzenia repozytorium na GitHubie, które zostało sklonowane na komputery obu członków zespołu. Następnie, w lokalnym folderze, wykonano podstawowe polecenia inicjalizujące Git, takie jak `git init`, dodanie pierwszego commita oraz skonfigurowanie zdalnego repozytorium.

1. Przykłady Komend Git używanych do projektu:

```
1 cd /sciezka/do/folderu
2 git init
3 git add .
4 git commit -m "first commit"
5 git branch -M main
6 git remote add origin https://github.com/WiktoriaGit/GitHub-Copilot
  .git
7 git push -u origin main
```

Listing 8. git_1

2. Tworzenie gałęzi

Podczas pracy nad różnymi funkcjonalnościami projektu, tworzone były nowe gałęzie, co pozwalało na oddzielenie wprowadzanych zmian od głównej wersji kodu. Dzięki temu możliwe było równoległe rozwijanie funkcji bez ryzyka przypadkowego nadpisania stabilnego kodu.

```
1 git checkout -b galaz
```

Listing 9. git_2

3. Commitowanie zmian

Regularne zapisywanie postępów w kodzie odbywało się za pomocą commitów. Dzięki nim możliwe było łatwe śledzenie historii zmian oraz cofanie się do poprzednich wersji w razie potrzeby. Każdy commit był opatrzony odpowiednim opisem, co zapewniało przejrzystość pracy.

```
1 git add .
2 git commit -m "nowa funkcja w klasie matrix"
```

Listing 10. git_3

4. Wysyłanie zmian na GitHub

Aby zsynchronizować lokalne zmiany zdalnie z repozytorium na GitHubie, wykorzystywano polecenie `git push`. Dzięki temu wszystkie zmiany były dostępne dla innych członków zespołu.

```
1 git push origin new_branch
```

Listing 11. git_4

5. Scalanie gałęzi

Po zakończeniu implementacji nowych funkcji, gałęzie były scalane z główną wersją kodu (`main`). Przed scaleniem każda funkcjonalność była testowana, aby upewnić się, że nie wprowadza błędów. Proces scalania przebiegał w kilku etapach: przełączenie na główną gałąź, połączenie gałęzi oraz rozwiązanie ewentualnych konfliktów.

```
1 git checkout main
2 git merge galaz
```

Listing 12. git_5

6. Rozwiązywanie konfliktów

Podczas scalania równolegle rozwijanych funkcji czasami dochodziło do konfliktów, które rozwiązywano ręcznie, edytując odpowiednie pliki i zatwierdzając poprawki. Konflikty mogły dotyczyć fragmentów kodu, w których kilka osób wprowadziło zmiany w tym samym czasie.

```
1 git add .
2 git commit -m "Rozwiazano konflikty"
```

Listing 13. git_6

4.3.2. Doxygen

Doxygen⁴ służy do generowania dokumentacji kodu źródłowego. Na podstawie specjalnych komentarzy w kodzie tworzy automatycznie dokumentację w różnych formatach, ułatwiając zrozumienie i utrzymanie projektów programistycznych.

4.3.3. Wdrożenie Doxygena

Aby wygenerować dokumentację przez Doxygena należy przejść przez następujące kroki:

⁴Źródło: [4]

1. Zainstaluj Doxygen

Pobierz Doxygen z oficjalnej strony.

2. Dodaj komentarze do kodu

W kodzie dodaj specjalne komentarze do funkcji.

3. Utwórz plik konfiguracyjny

Uruchom komendę `doxygen -g`, aby wygenerować domyślny plik konfiguracyjny `Doxyfile`. Ten plik zawiera ustawienia, które kontrolują, co Doxygen będzie przetwarzał i jak generować dokumentację.

4. Dostosuj plik `Doxyfile`

Otwórz plik `Doxyfile` i dostosuj opcje, takie jak:

```
1 INPUT = ./sciezka/do/kodow/zrodlowych
2 FILE_PATTERNS = *.cpp *.h
3 SOURCE_BROWSER = YES
4 OUTPUT_DIRECTORY = przykladowy_folder
```

Listing 14. Ustawienia Doxygen

Na listingu 14 znajdują się podstawowe ustawienia pliku `Doxyfile` takie jak: ścieżka wskazująca na folder z plikami, przetwarzane typy plików, uwzględnienie plików źródłowych, oraz określenie nazwy folderu w którym wygenerują się pliki.

5. Uruchom Doxygena

W terminalu wpisz `doxygen Doxyfile`, aby uruchomić generowanie dokumentacji na podstawie konfiguracji. Dokumentacja zostanie zapisana w folderze określonym w `OUTPUT_DIRECTORY`.

6. Otwórz wygenerowaną dokumentację

Najczęściej Doxygen generuje dokumentację w formacie HTML (znajduje się w folderze `html`). Otwórz plik `index.html` w przeglądarce, aby przeglądać dokumentację. W przypadku chęci wygenerowania dokumentacji w `latex`, spakuj folder `latex` i wypakuj w OverLeaf.

5. Wnioski

W trakcie pracy nad projektem mieliśmy okazję zapoznać się z różnymi kombinacjami zadań związanymi z zarządzaniem pamięcią dynamiczną, optymalizacją kodu, a także implementacją zaawansowanych funkcji matematycznych. Ostatecznie udało się osiągnąć zakładane cele i stworzyć wszechstronną klasę umożliwiającą efektywne operacje na macierzach kwadratowych.

Implementacja klasy `matrix` pozwoliła nam zrozumieć kluczową rolę zarządzania pamięcią dynamiczną w języku C++. Dzięki zastosowaniu konstruktorów, destruktorów oraz odpowiednich mechanizmów realokacji pamięci, testowaliśmy, jak unikać wycieków pamięci i zapewnić stabilność aplikacji.

Przeciążenie operatorów znacząco ułatwiło korzystanie z klasy, czyniąc jej API bardziej intuicyjnym i czytelnym.

Podział projektu na pliki nagłówkowe (`matrix.h`), implementacyjne (`matrix.cpp`) oraz testowe (`main.cpp`) pomógł w utrzymaniu porządku w kodzie. Dzięki temu łatwiej było zarządzać poszczególnymi funkcjonalnościami i wprowadzać zmiany.

Funkcja `main` została zaprojektowana jako kompleksowy zestaw testów, co pozwoliło na wczesne wykrywanie błędów i szybkie ich naprawianie. Narzędzie Git oraz platforma GitHub były kluczowe dla płynnej współpracy zespołu. Dzięki gałęziom mogliśmy pracować nad różnymi funkcjonalnościami równolegle, a system kontroli wersji umożliwił łatwe śledzenie zmian i rozwiązywanie konfliktów.

GitHub Copilot okazał się pomocny w przyspieszaniu procesu kodowania, proponując fragmenty kodu i rozwiązania. Niemniej jednak zauważyliśmy, że narzędzia oparte na sztucznej inteligencji wymagają nadzoru i krytycznej oceny, ponieważ nie zawsze generują optymalne rozwiązania.

Bibliografia

- [1] *Strona Internetowa Wyznacznik*. URL: <https://wyznacznik.pl/podstawowe-dzialania-na-macierzach-teoria>.
- [2] *Strona Internetowa Wikipedia*. URL: <https://pl.wikipedia.org/wiki/Macierz>.
- [3] *Strona/Platforma GitHub*. URL: <https://github.com/features/copilot>.
- [4] *Strona internetowa Doxygen*. URL: <https://www.doxygen.nl/manual/>.

Spis rysunków

2.1. Przykład działania Copilota	9
2.2. Uzupełnienia Copilota	10
3.1. UML Macierz	12

Spis tabel

Spis listingów

1.	plik matrix.h	14
2.	Konstruktor i destruktor	15
3.	Funkcja alokuj	16
4.	Funkcja losuj	16
5.	operator + oraz *	16
6.	Funkcja odwróć	17
7.	Kod main (wybrane części)	17
8.	git_1	19
9.	git_2	19
10.	git_3	19
11.	git_4	20
12.	git_5	20
13.	git_6	20
14.	Ustawienia Doxygen	21