**תכנות מערכות ב' - התשפ"ד 2024 – פתרון מועד א'**

להלן פתרון מוצע של הבחינה

**פתרון שאלה 1**

1.1

העמסת פונקציות (Function Overloading)

היא יכולת בשפות תכנות שמאפשרת להגדיר מספר פונקציות עם אותו שם,

כל עוד יש להן חתימות שונות (המשמעות היא שמספר, סדר או סוג הפרמטרים שונה).

באופן כללי, החתימה של פונקציה כוללת את שמה ואת רשימת הפרמטרים שלה (סוג הפרמטרים וסדרם).

חשוב לציין ששינוי סוג הערך המוחזר מהפונקציה בלבד אינו נחשב כהעמסת פונקציות.

1.2

ניהול זיכרון סטטי (Static Memory Management)\*:

- האובייקטים מוקצים בזיכרון הסטטי בזמן הקומפילציה ומשוחררים אוטומטית כאשר התוכנית מסתיימת.

- אחראי: הקומפיילר.

cpp

static int globalVar = 10; // מוקצה בזיכרון סטטי

\*ניהול זיכרון אוטומטי (Automatic Memory Management)\*:

- (stack) האובייקטים מוקצים בזיכרון המחסנית כאשר הפונקציה נקראת ומשוחררים אוטומטית כאשר הפונקציה מסתיימת.

- אחראי: קומפיילר.

cpp

void func(){

int localVar = 5; // מוקצה בזיכרון המחסנית

}

\*ניהול זיכרון דינמי (Dynamic Memory Management)\*:

- מתבצעת בזיכרון הערימה באמצעות מילה שמורה

new

- שחרור מתבצע ידנית על ידי המתכנת באמצעות קריאה למילה שמורה

delete

- אחראי: המתכנת.

int\* dynamicVar = new int[10]; // מוקצה בזיכרון הערימה

delete []dynamicVar; // שחרור הזיכרון

1.3

1 - ניתן לשנות את הערך של המשתנה אבל לא דרך המצביע עצמו כי האו מצביע לערך שלם קבוע

2 - מצביע קבוע לערך שלם כלומר ניתן לשנות את הערך השלם אבל לא ניתן להפנות את המצביע לכתובת אחרת

3- המילה קונסט מופיעה בסו חתימה וזה אומר שבתוך פונקציה זו לא ניתן לשנות את הששדות של המחלקה

4 - המילה קונסט שבסוף החתימה כמו מקודם , המילה קונסט שבתחית החתימה אומרת שהרפרנס שמוחזר הוא קבוע ולא ניתן לשנות את ערך המחרוזת דרך רפרנס זה

5 - המילה קונסט שמופיעה בסוך החתימה כמו קודם, המילה קונדט שבתחית החתימה אומרת שהערך של המחרוזת המוחזר אינו ניתן לשינוי

1.4

implict - המרה המסוגלת להתרחש ללא צורך בהכרזה מפורשת על ידי המתכנת.

במקרים אלו, הקומפיילר מבצע את ההמרה באופן אוטומטי במסגרת הכללים של השפה

int num1 = 10;

double num2 = num1;

explict - המרה המתרחשת רק כאשר המתכנת מבקש ודורש זאת , המרה שבה המתכנת חייב להשתמש באופרטורים או בפונקציות מובנות לצורך ביצוע ההמרה.

double num1 = 3.14;

int num2 = static\_cast<int>(num1); // המרה מ-double ל-int באופן מפורש

1.5

מימוש פרטני של תבנית מאפשר לך להגדיר פעולות שונות או מבנים שונים לסוג נתונים מסוימים.

בעזרתו, ניתן לספק פתרונות ספציפיים למקרים מיוחדים שבהם התבנית הכללית אינה מתאימה באופן מלא

יתרונות

התאמה מדויקת לדרישות המקרה: זה מאפשר לנו לקבל פתרון מדויק ומותאם לסוג הנתונים או למקרה המסוים שאנו רוצים לטפל בו.

יעילות בביצועים ובמשאבים: מימוש פרטני יכול לאפשר שיפורים בביצועים על ידי התאמה של קוד מסוגל למצבים מיוחדים שבהם יש יכולות יעילות גבוהות יותר.

קלות בתחזוקה ובהבנה: במקרים בהם יש מימוש פרטני, הקוד מובן יותר במהירות וניתן לתחזק אותו בצורה יותר טובה.

**שאלה 2**

#include <iostream>

#include <initializer\_list>

template<typename T, int ROWS, int COLS>

class Matrix {

private:

T data[ROWS][COLS];

public:

Matrix() {

for (int i = 0; i < ROWS; ++i)

for (int j = 0; j < COLS; ++j)

data[i][j] = T();

}

Matrix(std::initializer\_list<std::initializer\_list<T>> init) {

int i = 0;

for (auto row : init) {

int j = 0;

for (auto elem : row) {

if (i < ROWS && j < COLS) {

data[i][j] = elem;

}

++j;

}

++i;

}

}

class Iterator {

private:

Matrix& matrix;

int row, col;

public:

Iterator(Matrix& matrix, int row, int col) : matrix(matrix), row(row), col(col) {}

Iterator& operator++() {

if (++col == COLS) {

col = 0;

++row;

}

return \*this;

}

bool operator!=(const Iterator& other) const {

return row != other.row || col != other.col;

}

T& operator\*() {

return matrix.data[row][col];

}

};

Iterator begin() { return Iterator(\*this, 0, 0); }

Iterator end() { return Iterator(\*this, ROWS, 0); }

class ColIterator {

private:

Matrix& matrix;

int row, col;

public:

ColIterator(Matrix& matrix, int row, int col) : matrix(matrix), row(row), col(col) {}

ColIterator& operator++() {

if (++row == ROWS) {

row = 0;

++col;

}

return \*this;

}

bool operator!=(const ColIterator& other) const {

return row != other.row || col != other.col;

}

T& operator\*() {

return matrix.data[row][col];

}

};

ColIterator colsBegin() { return ColIterator(\*this, 0, 0); }

ColIterator colsEnd() { return ColIterator(\*this, 0, COLS); }

};

int main() {

Matrix<int, 3, 4> matrix = {

{9, 8, 7, 0},

{6, 5, 4, 5},

{3, 2, 1, 9}

};

std::cout << "Matrix elements by rows: ";

for (auto it = matrix.begin(); it != matrix.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl << "Matrix elements by columns: ";

for (auto it = matrix.colsBegin(); it != matrix.colsEnd(); ++it) {

std::cout << \*it << " ";

}

return 0;

**פתרון שאלה 3**

א.

A:

func1: A::func1

func3: A::func3

B:

func1: B::func1

func3: B::func3

func4: B::func4

C:

func1: C::func1

func3: C::func3

func4: B::func4

D:

func1: D::func1

func3: D::func3

func4: D::func4

ב.

D::func1  
B::func2  
D::func3  
D::func4

ג.

מטרת הגדרת פונקציה כוירטואלית היא לאפשר פולימורפיזם בזמן ריצה. זה מאפשר לקרוא לפונקציות של אובייקטים לפי הסוג בפועל בזמן ריצה, ולא לפי סוג המשתנה שהאובייקט שמור בו.

אם נשכח להגדיר פונקציה כפונקציה וירטואלית כאשר נדרשת הגדרה זו, אזי הקריאות לפונקציה יבוצעו לפי סוג המשתנה ולא לפי סוג האובייקט. כלומר, לא יתבצע פולימורפיזם בזמן ריצה והפונקציות של המחלקה אב יופעלו במקום הפונקציות של המחלקות הנגזרות.

לא כדאי להוסיף ולהשתמש במנגנון וירטואליות לכלל הפונקציות במחלקה אם אין הכרח או צורך בכך. הגדרת פונקציות כווירטואליות גוררת עלויות נוספות של ביצועים וזיכרון. בנוסף, זה יכול להוביל לקוד פחות ברור ומסובך יותר לתחזוקה.

**פתרון שאלה 4**

א. יש הקצאה (NEW) ואין שחרור של הזכרון – מפרק destructur (מימוש משולב ביחד בקוד עם חלק ב')

ב. חוק השלושה – מפרק , בנאי מעתיק, אופרטור השמה

לאור הוספת המפרק , נדרש גם לממש קוד גם של בנאי מעתיק וגם של = אופרטור השמה

להלן מימוש המחלקה הכוללת את כלל הדברים הנדרשים בקוד

\*/

class DMatrix {

private:

int rows, cols;

int\*\* data;

public:

DMatrix(int r, int c) : rows(r), cols(c), data(new int\*[r]) {

for (int i = 0; i < r; ++i) {

data[i] = new int[c];

}

}

~DMatrix() {

for (int i = 0; i < rows; ++i) {

delete[] data[i];

}

delete[] data;

}

DMatrix(const DMatrix& other) : rows(other.rows), cols(other.cols), data(new int\*[other.rows]) {

for (int i = 0; i < rows; ++i) {

data[i] = new int[cols];

std::copy(other.data[i], other.data[i] + cols, data[i]);

}

}

DMatrix& operator=(const DMatrix& other) {

if (this != &other) {

for (int i = 0; i < rows; ++i) {

delete[] data[i];

}

delete[] data;

rows = other.rows;

cols = other.cols;

data = new int\*[rows];

for (int i = 0; i < rows; ++i) {

data[i] = new int[cols];

std::copy(other.data[i], other.data[i] + cols, data[i]);

}

}

return \*this;

}

int getRows() const { return rows; }

int getCols() const { return cols; }

int\*\* getData() const { return data; }

};

**פתרון שאלה 5**

a.

#include <iostream<

#include <vector<

#include <iterator<

// Template function to find the smallest element in a container

template <typename Iterator<

Iterator findSmallest(Iterator begin, Iterator end) {

if (begin == end) {

return end; // Return end if the container is empty

}

Iterator smallest = begin;

for (Iterator it = std::next(begin); it != end; ++it) {

if (\*it < \*smallest) {

smallest = it;

}

}

return smallest;

}

b.

The code that is needed to add is an implementation of the < opeartor for this class.

c.

19a: can't access private fields of car, need to initialize car and not the fields themselves

19b: need to add std:: to string since we aren't using namespace std

20: functions written as override, but the original function is also const, so the new function doesn't override, need to add const

22: same as 19b for both cout and endl