Міністерство освіти і науки України

Національний університет “Львівська політехніка”

Інститут комп’ютерних наук та інформаційних технологій

Кафедра програмного забезпечення



**Звіт**

Про виконання лабораторної роботи №8

**На тему:**

“Виняткові ситуації в мові програмування С++”

**Лектор:**

доц. каф. ПЗ

Коротєєва Т.О.

**Виконав:**

ст. гр. ПЗ-18

Юшкевич А.І.

**Прийняв:**

асис. каф. ПЗ

Далявський В.С.

« … » … 2023 р.

∑ = \_\_\_\_\_\_\_\_\_\_\_\_

Львів – 2023

**Тема**: Виняткові ситуації в мові програмування С++.

**Мета**: Ознайомитися з синтаксисом та принципами використання винятків, навчитися передбачати виняткові ситуації, які можуть виникнути в процесі роботи програмного забезпечення, а також навчитися їх перехоплювати та опрацьовувати.

Теоретичні відомості

**Винятки**

В С++ винятки – це об‘єкт, який передається з тої області коду, де виникла проблема, в ту область коду, де проблема обробляється. Тип винятка визначає, яка область коду буде обробляти проблему, а вміст об’єкту може використовуватись для зворотного зв‘язку з користувачем.

Блок try містить код, який може викликати виняток. Якщо виняток виникає, то він "кидається" (thrown), і програма переходить до найближчого блоку catch, який збігається з типом винятку. Якщо виняток не збігається з жодним з типів, вказаних у блоках catch, то він переходить до блоку catch (...), який може обробити будь-який тип винятку.

Параметри винятку (у нашому прикладі параметр\_винятку\_1 та параметр\_винятку\_2) містять деталі про виняток, які можуть бути корисними для обробки.

У блоках catch можна використовувати будь-які команди для обробки винятків, наприклад, вивести повідомлення про помилку, записати її в лог-файл, повідомити користувача, тощо.

Оператор throw генерує виняткову ситуацію. Для цього оператор throw повинний знаходитися усередині блоку try або усередині функції, що викликається усередині блоку try. За ключовим словом throw вказується значення будь-якого типу даних, яке ви хочете задіяти, щоб сигналізувати про помилку. Як правило, цим значенням є код помилки, опис проблеми або спеціальний клас-виняток.

Завдання

1. Проаналізувати класи реалізовані у лабораторній роботі №5 відповідно до варіанту, визначити можливі виняткові ситуації, які можуть виникнути в процесі роботи програмного забезпечення.

2. Розробити набір або ієрархію класів-винятків і застосувати їх у класах відповідно до варіанту.

3. Функції, які не генерують винятків позначити специфікатором noexcept.

4. Продемонструвати роботу із винятками таким чином (на вибір):

1. За допомогою тестів Google Test, які генерують винятки.
2. За допомогою інтерактивної програми із демонстрацією її роботи.

5. Оформити і здати звіт про виконання лабораторної роботи. Звіт має містити варіант завдання, теоретичний матеріал, код розробленої програми, результати роботи (скріншоти) та повідомлення про винятки, висновок.

Варіант №13:

|  |  |
| --- | --- |
| 13 | Клас CTable – таблиця із колонками та рядками. Містить опис колонок та рядки із даними. Пам’ять під елементи масиву повинна виділятися динамічно.  Передбачити щонайменше такі виняткові ситуації:   * нестача динамічної пам’яті * помилки вводу/виводу при роботі із std::istream/std::ostream * спроба отримати назву колонки з індексом поза межами діапазону індексів * спроба отримати тип колонки з індексом поза межами діапазону індексів * спроба отримати значення рядка з індексом поза межами діапазону індексів * спроба додавання результатів із різною кількістю колонок * спроба множення результатів із нульовою кількістю колонок |

# Код програми

## Lab\_08\_OOP:

Table\_Header\_Unique.h:

#pragma once

#include <iostream>

#include <vector>

#include <variant>

#include "CExceptions.h"

using namespace std;

class CTable {

size\_t number\_of\_columns;

unique\_ptr<vector<variant<int, double, char, string>>[]> table{ nullptr };

vector<string> columns\_name;

size\_t max\_size\_of\_column;

void AddColumn(vector<variant<int, double, char, string>> column);

template <class T>

void GetNewColumn(T foo, int current\_index, istream& is) {

for (int i = 0; i < table[current\_index].size(); i++) {

cout << "\nEnter " << i + 1 << "element: ";

try {

try {

is >> foo;

}

catch (const exception& ex) {

throw CInputIssue();

}

}

catch (const CExceptions& issue) {

cout << "\n\n" << issue.what() << "\n\n";

}

table[current\_index][i] = foo;

}

}

public:

CTable() noexcept;

CTable(CTable& r\_other);

CTable(CTable&& rr\_other) noexcept;

//~CTable();

CTable& operator= (const CTable& r\_other) noexcept;

friend ostream& operator<< (ostream& os, const CTable& r\_other);

friend istream& operator>> (istream& os, CTable& r\_other);

CTable operator+(const CTable& r\_other) noexcept;

CTable operator\*(const CTable& r\_other);

template <class T>

void CreateNewColumn(T arr[], size\_t number\_of\_elements, string column\_name) {

if (number\_of\_elements > max\_size\_of\_column)

max\_size\_of\_column = number\_of\_elements;

vector<variant<int, double, char, string>> new\_column(number\_of\_elements);

for (int i = 0; i < number\_of\_elements; i++) {

new\_column[i] = arr[i];

}

AddColumn(new\_column);

columns\_name.push\_back(column\_name);

}

template<class T>

T GetValueOfTable(int column, int row) const {

T result{ 0 };

try {

table[column].at(row);

if (const T\* value\_2 = get\_if<T>(&table[column][row])) {

result = \*value\_2;

}

else {

try {

throw CGetValueOfColumnOutOfRange();

}

catch (const CExceptions& ex) {

cout << "\n\n" << ex.what() << "\n\n";

}

}

}

catch (const exception& issue) {

cout << "\n\n" << issue.what() << "\n\n";

}

return result;

}

void ShowTable() const noexcept;

int GetNumOfRows() const noexcept;

int GetNumOfColumns() const noexcept;

string GetNameOfColumn(int index) const;

string GetTypeOfColumn(int index) const;

};

Table\_Functions\_Unique.cpp:

#include "Table\_Header\_Unique.h"

CTable::CTable() noexcept : max\_size\_of\_column(0), number\_of\_columns(0) {};

CTable::CTable(CTable&& rr\_other) noexcept : max\_size\_of\_column(rr\_other.max\_size\_of\_column), number\_of\_columns(rr\_other.number\_of\_columns) {

this->table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[rr\_other.number\_of\_columns]);

try {

if (this->table == nullptr)

throw CBadAllocation();

}

catch (const CBadAllocation& ex) {

cout << "\n\n" << ex.what() << "\n\n";

}

this->columns\_name = rr\_other.columns\_name;

rr\_other.table = nullptr;

};

CTable::CTable(CTable& r\_other) : max\_size\_of\_column(r\_other.max\_size\_of\_column), number\_of\_columns(r\_other.number\_of\_columns) {

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[r\_other.number\_of\_columns]);

try {

if (new\_table == nullptr)

throw CBadAllocation();

}

catch (const CExceptions& ex) {

cout << "\n\n" << ex.what() << "\n\n";

}

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = r\_other.table[i];

}

if (this->table != nullptr)

this->table.reset();

this->columns\_name = r\_other.columns\_name;

this->table.swap(new\_table);

}

//CTable::~CTable() {

// if (this->table != nullptr)

// this->table.reset();

// this->table = nullptr;

//

//}

ostream& operator<<(ostream& os, const CTable& c\_table) {

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

os << c\_table.columns\_name[i] << "\t\t";

}

os << endl << endl;

for (int i = 0; i < c\_table.max\_size\_of\_column; i++) {

for (int j = 0; j < c\_table.number\_of\_columns; j++) {

if (i >= c\_table.table[j].size()) {

os << " \t\t";

continue;

}

if (auto value = get\_if<int>(&c\_table.table[j][0])) {

os << \*get\_if<int>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&c\_table.table[j][0])) {

os << \*get\_if<double>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&c\_table.table[j][0])) {

os << \*get\_if<char>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&c\_table.table[j][0])) {

os << \*get\_if<string>(&c\_table.table[j][i]) << "\t\t";

}

else {

try {

throw COutputIssue();

}

catch(const CExceptions& ex){

cout << "\n\n" << ex.what() << "\n\n";

}

}

}

os << endl;

}

os << endl;

return os;

}

istream& operator>> (istream& is, CTable& c\_table) {

c\_table.number\_of\_columns = 0;

cout << "Enter number of columns: ";

is >> c\_table.number\_of\_columns;

cout << endl;

if (c\_table.number\_of\_columns <= 0) {

c\_table.number\_of\_columns = 0;

return is;

}

if (c\_table.table != nullptr)

c\_table.table.reset();

c\_table.table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[c\_table.number\_of\_columns]);

c\_table.columns\_name.clear();

c\_table.columns\_name.resize(c\_table.number\_of\_columns);

c\_table.max\_size\_of\_column = 0;

int num\_of\_elements{ 0 };

string type{ 0 };

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

cout << "\n\nEnter number of elements in " << i + 1 << "column: ";

is >> num\_of\_elements;

cout << endl;

if (c\_table.max\_size\_of\_column < num\_of\_elements)

c\_table.max\_size\_of\_column = num\_of\_elements;

cout << "Enter type of elements in this column: ";

is >> type;

cout << endl;

c\_table.table[i].resize(num\_of\_elements);

if (type == "int" || type == "Int" || type == "INT" || type == "Int32") {

int foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "double" || type == "Double" || type == "DOUBLE") {

double foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "char" || type == "Char" || type == "CHAR") {

char foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "string" || type == "String" || type == "STRING") {

string foo{ "\0" };

c\_table.GetNewColumn(foo, i, is);

}

else {

cout << "\n\nWrong type, try again: \n\n";

i--;

continue;

}

c\_table.columns\_name[i] = type;

num\_of\_elements = 0;

}

cout << endl << endl;

return is;

}

CTable& CTable::operator= (const CTable& c\_table) noexcept {

this->max\_size\_of\_column = c\_table.max\_size\_of\_column;

this->number\_of\_columns = c\_table.number\_of\_columns;

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[number\_of\_columns]);

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = c\_table.table[i];

}

if (this->table != nullptr)

this->table.reset();

this->columns\_name = c\_table.columns\_name;

this->table.swap(new\_table);

return \*this;

}

CTable CTable::operator+(const CTable& r\_other) noexcept {

CTable c\_result\_table;

c\_result\_table.table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[number\_of\_columns]);

int i = 0;

for (int j = 0; j < this->number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(this->table[j]);

c\_result\_table.columns\_name.push\_back(this->columns\_name[j]);

}

i--;

for (int j = 0; j < r\_other.number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(r\_other.table[j]);

c\_result\_table.columns\_name.push\_back(r\_other.columns\_name[j]);

}

return c\_result\_table;

}

CTable CTable::operator\*(const CTable& r\_other) {

CTable additional\_table;

try {

if (this->number\_of\_columns == 0 || r\_other.number\_of\_columns == 0)

throw CZeroColumnsMultiplication();

}

catch(const CExceptions& ex){

cout << "\n\n" << ex.what() << "\n\n";

}

for (int i = 0; i < r\_other.number\_of\_columns; i++) {

for (int j = 0; j < r\_other.table[i].size(); j++) {

vector<variant<int, double, char, string>> new\_column;

for (int k = 0; k < this->max\_size\_of\_column; k++) {

new\_column.push\_back(r\_other.table[i][j]);

}

additional\_table.AddColumn(new\_column);

additional\_table.columns\_name.push\_back(r\_other.columns\_name[i]);

new\_column.clear();

}

}

return \*this + additional\_table;

}

void CTable::ShowTable() const noexcept {

for (int i = 0; i < number\_of\_columns; i++) {

cout << columns\_name[i] << "\t\t";

}

cout << endl << endl;

for (int i = 0; i < max\_size\_of\_column; i++) {

for (int j = 0; j < number\_of\_columns; j++) {

if (i >= table[j].size()) {

cout << " \t\t";

continue;

}

if (auto value = get\_if<int>(&table[j][0])) {

cout << \*get\_if<int>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&table[j][0])) {

cout << \*get\_if<double>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&table[j][0])) {

cout << \*get\_if<char>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&table[j][0])) {

cout << \*get\_if<string>(&table[j][i]) << "\t\t";

}

else break;

}

cout << endl;

}

}

int CTable::GetNumOfColumns() const noexcept {

return number\_of\_columns;

}

int CTable::GetNumOfRows() const noexcept {

return max\_size\_of\_column;

}

string CTable::GetNameOfColumn(int index) const {

string name{ 0 };

try {

try {

name = columns\_name.at(index);

}

catch(const exception& ex){

throw CGetNameOfColumnOutOfRange();

}

}

catch(const CExceptions& issue){

cout << "\n\n" << issue.what() << "\n\n";

}

return name;

}

string CTable::GetTypeOfColumn(int index) const {

string result = "unknown";

if (auto value = get\_if<int>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<double>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<char>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<string>(&table[index][0])) {

result = typeid(\*value).name();

}

else {

try {

throw CGetTypeOfColumnOutOfRange();

}

catch (const CExceptions& ex) {

cout << "\n\n" << ex.what() << "\n\n";

}

}

return result;

}

void CTable::AddColumn(vector<variant<int, double, char, string>> column) {

number\_of\_columns++;

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[number\_of\_columns]);

try {

if (new\_table == nullptr)

throw CBadAllocation();

}

catch (const CExceptions& ex) {

cout << "\n\n" << ex.what() << "\n\n";

}

int i = 0;

for (; i < number\_of\_columns - 1; i++) {

new\_table[i] = table[i];

}

new\_table[i] = column;

if (column.size() > max\_size\_of\_column)

max\_size\_of\_column = column.size();

if (this->table != nullptr)

table.reset();

table.swap(new\_table);

}

CExceptions.h:

#pragma once

#include <iostream>

#include <exception>

using namespace std;

class CExceptions : public exception

{

public:

CExceptions() = default;

char const\* what() const override;

};

class CBadAllocation : public CExceptions{

public:

CBadAllocation() = default;

char const\* what() const override;

};

class CInputIssue : public CExceptions{

public:

CInputIssue() = default;

char const\* what() const override;

};

class COutputIssue : public CExceptions{

public:

COutputIssue() = default;

char const\* what() const override;

};

class CGetNameOfColumnOutOfRange : public CExceptions {

public:

CGetNameOfColumnOutOfRange() = default;

char const\* what() const override;

};

class CGetTypeOfColumnOutOfRange : public CExceptions {

public:

CGetTypeOfColumnOutOfRange() = default;

char const\* what() const override;

};

class CGetValueOfColumnOutOfRange : public CExceptions {

public:

CGetValueOfColumnOutOfRange() = default;

char const\* what() const override;

};

class CZeroColumnsMultiplication : public CExceptions {

public:

CZeroColumnsMultiplication() = default;

char const\* what() const override;

};

Lab\_08\_OOP.cpp:

#include "CExceptions.h"

char const\* CExceptions::what() const {

return "Some of exceptions, I don't know what you want from me!";

}

char const\* CBadAllocation::what() const {

return "Your shit doesn't work cause your compiler is suck in allocating merely few bytes of memory :)";

}

char const\* CInputIssue::what() const {

return "How dare you to enter your shitty wrong input values?! You gonna die for this \*sounds of chainsaw\*";

}

char const\* COutputIssue::what() const {

return "What?! You're so pathetic you even not able to print some fuckin' text you've written? Idiot";

}

char const\* CGetNameOfColumnOutOfRange::what() const {

return "Moron. There is no such a column your index as big as my dick. That's to much. Or you stupid enough to enter negative number, comparable with size of your stuff in pants?";

}

char const\* CGetTypeOfColumnOutOfRange::what() const {

return "Hey man, I'm with you. No, really, bein imbecile is a heavy burden, and I'll try to help you, you should only ask :)";

}

char const\* CGetValueOfColumnOutOfRange::what() const {

return "Now you come to me and say: \"Code, give me normal result\", but you don't ask with respect.";

}

char const\* CZeroColumnsMultiplication::what() const {

return "You're hopeless fatty clown. Just fuck off.";

}

Lab\_08\_OOP.cpp:

#include "Table\_Header.h"

#include "Table\_Header\_Unique.h"

#include <iostream>

#include <vector>

#include <variant>

int main()

{

const size\_t size{ 4 }, size2{ 5 };

CTable table, table\_2, table\_3, table\_4;

int arr\_1[size2]{ 3, 5, 3, 67, 4 };

double arr\_2[size]{ 3.43, 5.77, 3.27, 67.43 };

char arr\_3[size2]{ 'd', 'u', '3', 'i', '.' };

table.CreateNewColumn(arr\_1, size2, "ints");

table.CreateNewColumn(arr\_2, size, "doubles");

table.CreateNewColumn(arr\_3, size2, "chars");

table\_2.CreateNewColumn(arr\_1, size2, "ints");

CTable t(table);

//table.ShowTable();

cout << table;

//cin >> table;

//cout << table;

//table\_2 = table;

//cout << table\_2;

table\_3 = table + table\_2;

cout << table\_3;

table\_4 = table \* table\_2;

cout << table\_4;

cout << "Name of firs Column is: " << table.GetNameOfColumn(-3) << ", it's type: " << table.GetTypeOfColumn(38) << endl;

string d = table.GetValueOfTable<string>(1, 2);

cout << "Value of second column and third row: " << d << endl;

}

### pch.h:

#pragma once

#include "gtest/gtest.h"

#include "../Lab\_08\_OOP/Table\_Header\_Unique.h"

#include "../Lab\_08\_OOP/CExceptions.h"

### 

### test.cpp:

#include "pch.h"

TEST(CTableTest, AllocationFault) {

CTable c\_table;

int arr[] = { 1, 2, 3, 4, 5 };

size\_t number\_of\_elements = 50;

string column\_name = "test\_column";

}

TEST(CTableTest, MultiplicationByZeroColumns) {

CTable c\_table1, c\_table2, c\_table3;

int arr3[] = { 11, 12, 13, 14, 15 };

c\_table2.CreateNewColumn(arr3, 5, "column3");

c\_table3 = c\_table1 \* c\_table2;

}

TEST(CTableTest, GetType) {

CTable table;

int arr[] = { 1, 2, 3, 4, 5 };

size\_t number\_of\_elements = 5;

string column\_name = "test\_column";

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.GetTypeOfColumn(-2);

}

TEST(CTableTest, GetName) {

CTable table;

int arr[] = { 1, 2, 3, 4, 5 };

size\_t number\_of\_elements = 5;

string column\_name = "test\_column";

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.GetNameOfColumn(-2);

}

TEST(CTableTest, GetValue) {

CTable table;

int arr[] = { 1, 2, 3, 4, 5 };

size\_t number\_of\_elements = 5;

string column\_name = "test\_column";

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

table.CreateNewColumn(arr, number\_of\_elements, column\_name);

double d{ 0 };

string s{ 0 };

d = table.GetValueOfTable<double>(1, -2);

s = table.GetValueOfTable<string>(1, 2);

}

# Результат виконання

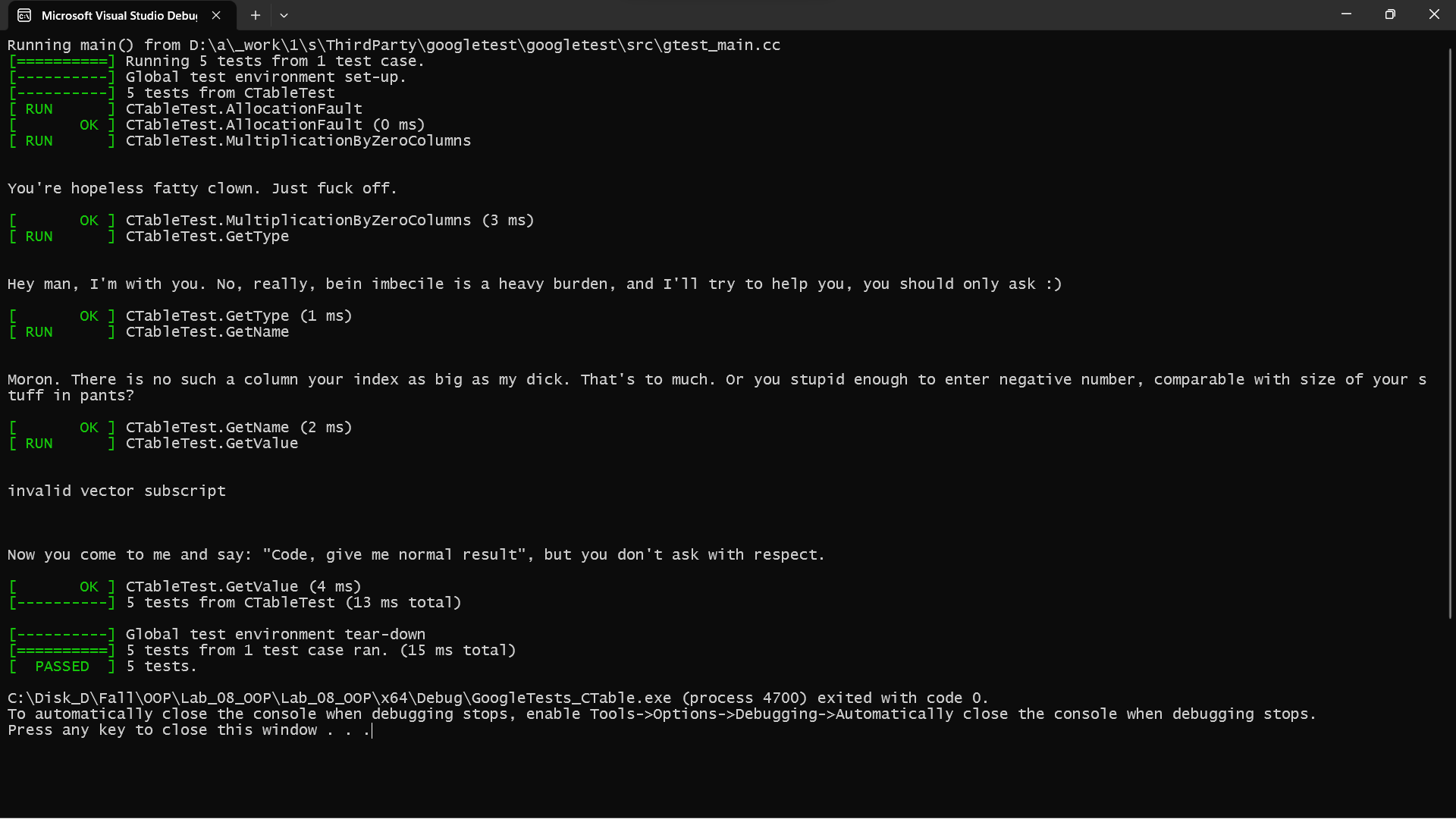


Рис. 1. Вивід повідомлень про помилки на екран

# Висновок

Під час виконання лабораторної роботи я ознайомився з синтаксисом та принципами використання винятків, навчився передбачати виняткові ситуації, які можуть виникнути в процесі роботи програмного забезпечення, а також навчився їх перехоплювати та опрацьовувати.