**Виняткові ситуації**

Одним з найбільш яскравих утілень принципу об'єктно-орієнтованого програмування є механізм обробки виняткових ситуацій у мові С++. У ході виконання програми можуть виявитися різні помилки. Вони можуть бути позв'язані з неправильним програмуванням (наприклад, вихід індексу масиву за межі припустимого чи переповнення пам'яті), а іноді їхня причина не залежить від програміста (наприклад, розрив зв'язку при мережевому з'єднанні). У кожній з цих ситуацій реакція програми непередбачена. Іноді вона завершує виконання, і лише після закінчення деякого інтервалу часу починають позначатися наслідку помилки, а частіше програма негайно припиняє роботу, піддаючи ризику дані, що знаходяться в пам'яті чи у файлі. Якщо не передбачити акуратне завершення роботи, використовуючи обробку виняткових ситуацій, результати можуть виявитися неприємними.

В подальшому ми будемо називати винятковою ситуацією будь-яку подію, що вимагає особливої обробки. При цьому зовсім неважливо, чи є ця подія фатальною чи простою помилкою. Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається обробкою виняткової ситуації. Ця задача покладається на оброблювача виняткової ситуації.

**Механізм обробки виняткових ситуацій**

Обробка виняткових ситуацій у мові С++ є об’єктно-орієнтованою. Це значить, що виняткова ситуація є об*'*єктом, що генерується при виникненні незвичайних умов, передбачених програмістом, і передається оброблювачу, що її перехоплює. Об'єктом, що описує природу виняткової ситуації, може бути будь-яка сутність — літерал, рядок, об'єкт класу, число і т.д. Не слід думати, що виняткова ситуація обов'язково повинна бути об'єктом якого-небудь класу.

**Обробка виняткових ситуацій**

В основі обробки виняткових ситуацій у мові С++ лежать три ключових слова: try, catch і throw. Якщо програміст підозрює, що визначений фрагмент програми може спровокувати помилку, він повинний занурити цю частину коду в блок try. Необхідно мати на увазі, що зміст помилки (за винятком стандартних ситуацій) визначає сам програміст. Це значить, що програміст може задати будь-яку умову, що приведе до створення виняткової ситуації. Після цього необхідно вказати, у яких умовах варто генерувати виняткову ситуацію. Для цієї мети призначене ключове слово throw. І нарешті, виняткову ситуацію потрібно перехопити й обробити в блоці catch. Ось як виглядає ця конструкція.

try {

// Тіло блоку try

if (умова) throw виняткова\_ситуація

}

catch (тип1 аргумент) {

// Тіло блоку catch

}

catch (тип2 аргумент) {

// Тіло блоку catch

}

. . .

catch (тип N аргумент) {

// Тіло блоку catch

}

Розмір блоку try не обмежений. У нього можна занурити як один оператор, так і цілу програму. Один блок try можна зв'язати з довільною кількістю блоків catch. Оскільки кожен блок catch відповідає окремому типу виняткової ситуації, програма сама визначить, який з них виконати. У цьому випадку інші блоки catch не виконуються. Кожен блок catch має аргумент, що приймає визначене значення. Цей аргумент може бути об'єктом будь-якого типу. Якщо програма виконана правильно й у блоці try не виникло жодної виняткової ситуації, усі блоки catch будуть зігноровані. Якщо в програмі виникла подія, яку програміст вважає небажаною, оператор throw генерує виняткову ситуацію. Для цього оператор throw повинний знаходитися усередині блоку try або усередині функції, викликуваної усередині блоку try.

Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort(). Утім, іноді виняткова ситуація не є небезпечної. У цьому випадку можна виправити помилку (наприклад, привласнити нульовому знаменнику ненульове значення) і продовжити виконання програми.

Розглянемо найпростіший приклад.

Обробка виняткової ситуації

int n = 10, m = 0;

cout << "Початок";

try {

cout << "У блоці try\n";

if (m == 0) throw "Divide by zero \n"; else n = n / m;

cout <<"Подальша частина блоку не виконується!";

}

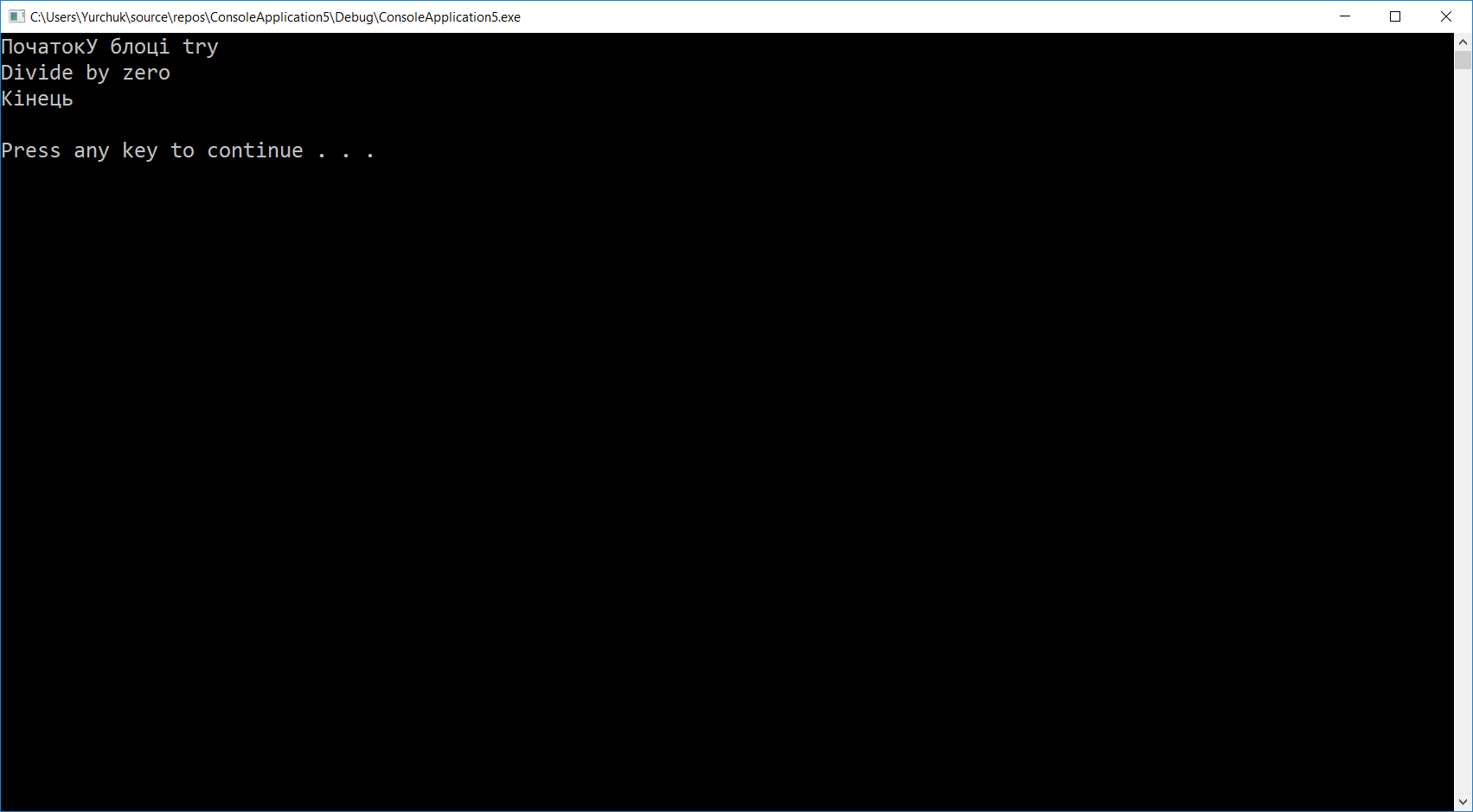
catch (const char\* s) {

cout << s;

}

cout << "Кінець\n";

Ця програма виводить на екран наступні рядки.



Простежимо за потоком керування при виконанні цієї програми. Спочатку з'являються і ініціалізуються дві цілочисельні змінні (одна з них дорівнює нулю). Потім виводиться повідомлення про початок виконання програми, і потік керування входить у блок try. Після виводу рядка повідомлення про вхід у блок try, потік керування переходить до перевірки рівності m==0. Оскільки ця рівність є істиною, генерується виняткова ситуація (у даному випадку — константний рядок). Керування негайно передається блоку catch, аргументом якого є константний символьний вказівник, ігноруючи всі інші оператори в блоці try. У цій програмі блок catch не робить жодних спроб виправити помилку. Замість цього він просто видає повідомлення — рядок, отриманий як аргумент — і передає керування оператору, що слідує за блоком. На закінчення виводиться на екран рядок Кінець, і програма завершує свою роботу.

Тип виняткової ситуації повинен збігатися з типом аргументу розділу catch. Поглянемо, що відбудеться, якщо цією умовою зневажити.

Порушення угоди про тип виняткової ситуації

int n = 10, m = 0;

cout << "Початок";

try {

cout << "У блоці try\n";

if (m == 0) throw "Divide by zero\n"; else n = n / m;

cout <<"Подальша частина блоку не виконується!";

}

catch (const char s) // Помилка! Необхідно const char\* s! {

cout << s;

}

cout << "Кінець\n";

У цій програмі ми зробили цілком “природну” помилку — забули поставити зірочку в оголошенні аргументу. Тепер блок catch очікує виняткову ситуацію, що представляє собою константний символ, а не вказівник. Ця помилка приводить до аварійного завершення роботи програми.

Покажемо, що відбудеться, якщо виняткова ситуація генерується усередині функції, яка викликається в блоці try.

Виняткова ситуація**,** згенерована усередині функції

int Denominator(int);

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n = 10, m;

cout << "Початок";

try {

cout << "У блоці try\n";

m = Denominator(0);

cout << "Подальша частина блоку не виконується!";

}

catch (const char \*s) // Помилка! Необхідно const char\* s!

{

cout << s;

}

cout << "Кінець\n";

cout << endl;

system("pause");

return 0;

}

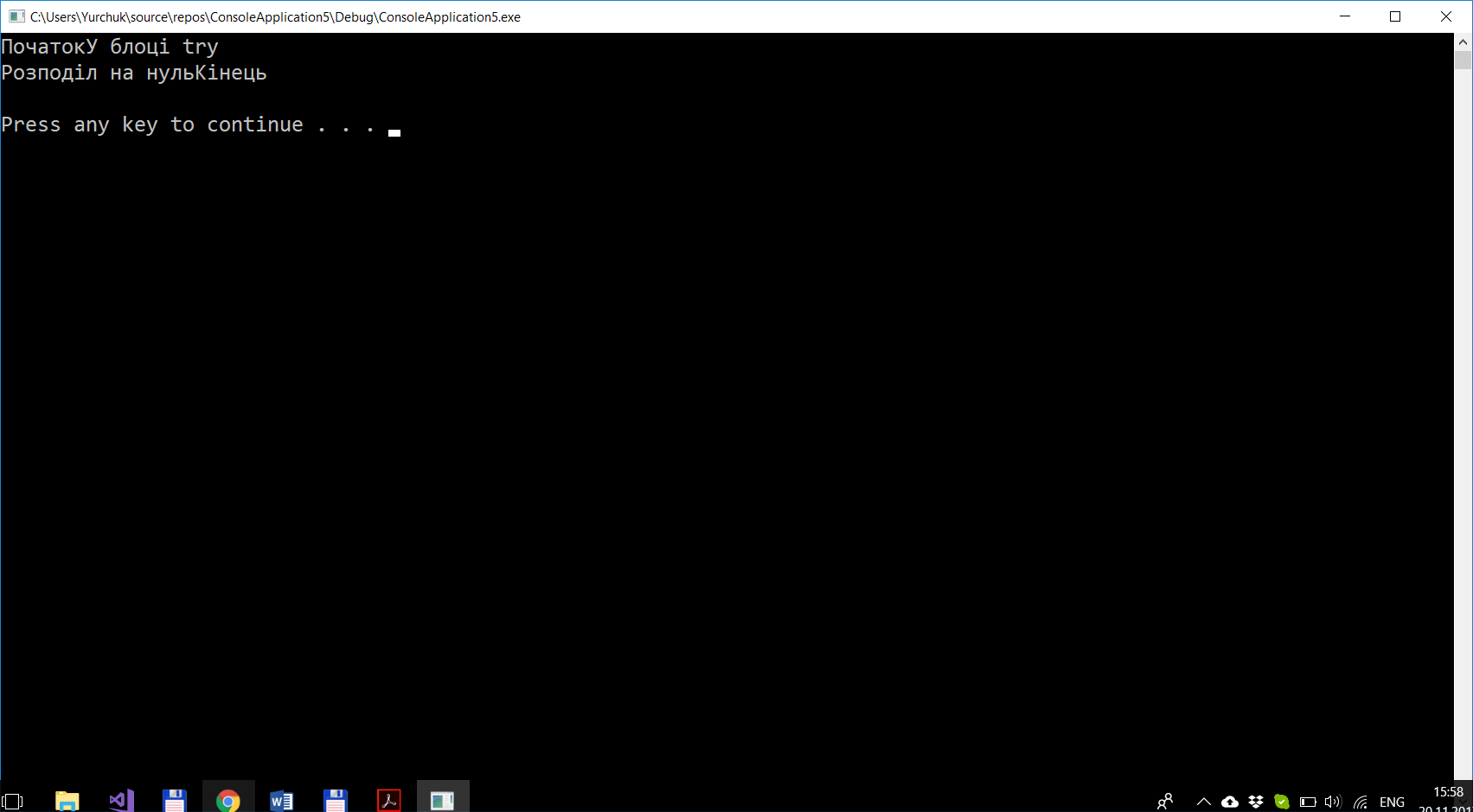
int Denominator(int i) {

if (i == 0)throw "Розподіл на нуль";

return i;

}

У цій програмі виняткова ситуація генерується у функції Denominator(), яка викликається в блоці try. Завдяки цьому результати роботи програми цілком збігаються з попередніми.



Якщо блок try знаходиться усередині функції, обробка виняткової ситуації виконується при кожному виклику.

Розміщення блоку tryусередині функції

int Denominator(int);

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n = 10, m;

printf("Початок\n");

m = Denominator(0);

n = Denominator(11);

printf("Кінець\n");

return 0;

}

int Denominator(int i) {

cout << "У функції Denominator\n";

try

{

cout << "У блоці try\n";

if (i == 0) throw("Розподіл на нуль!");

if (i>10) throw 10;

cout << "Подальша частина блоку не виконується!";

}

catch (const char\* s)

{

cout << s;

}

catch (int n)

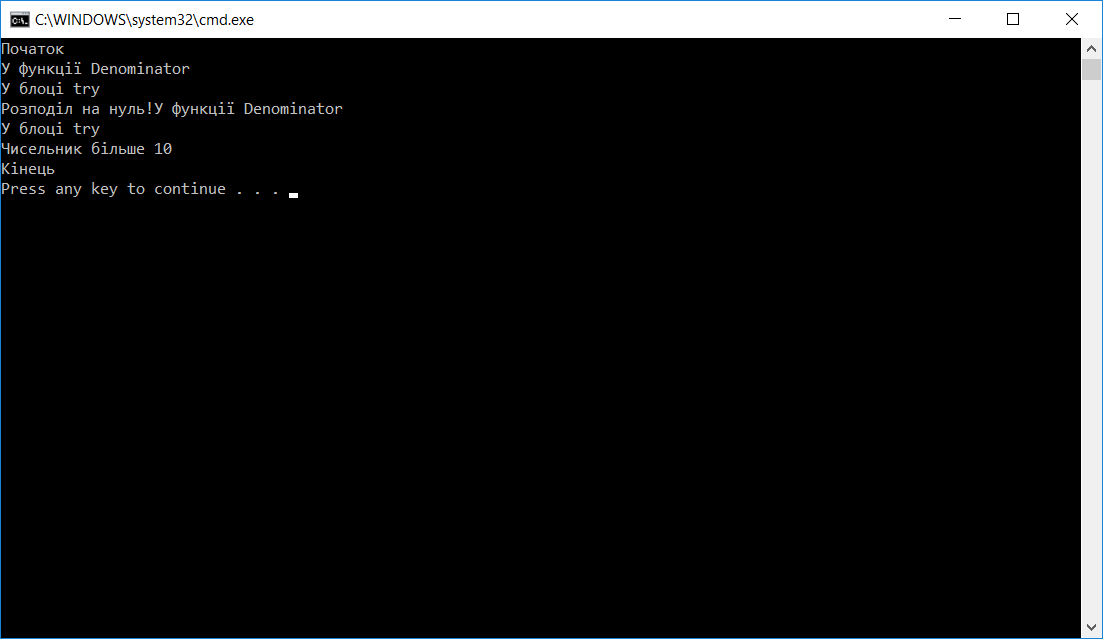
{

cout << "Чисельник більше " << n << endl;

return i;

}

}



У цій програмі передбачене перехоплення двох виняткових ситуацій. Перша з них має тип const char\* і генерується, коли знаменник дорівнює нулю, а друга — тип int і генерується, коли чисельник перевищує 10. Як бачимо, ці виняткові ситуації перевіряються і перехоплюються при кожнім виклику функції Denominator().

Розглянемо тепер приклад, у якому функція Denominator() лише генерує виняткові ситуації, а їх обробка здійснюється у функції main().

Окрема обробка виняткових ситуацій

#include "windows.h"

#include <iostream>

using namespace std;

int Denominator(int);

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n = 10, m;

cout << "Початок\n";

try {

cout << "У блоці try\n";

m = Denominator(0);

n = Denominator(11);

cout << "Подальша частина блоку не виконується!";

}

catch (const char\* s) {

cout << s;

}

catch (int n) {

cout << "Чисельник більше 10\n" << n;

}

cout << "Кінець\n";

cout << endl;

system("pause");

return 0;

}

int Denominator(int i) {

cout << "Усередині функції Denominator\n";

if (i == 0) throw("Ділення на нуль!");

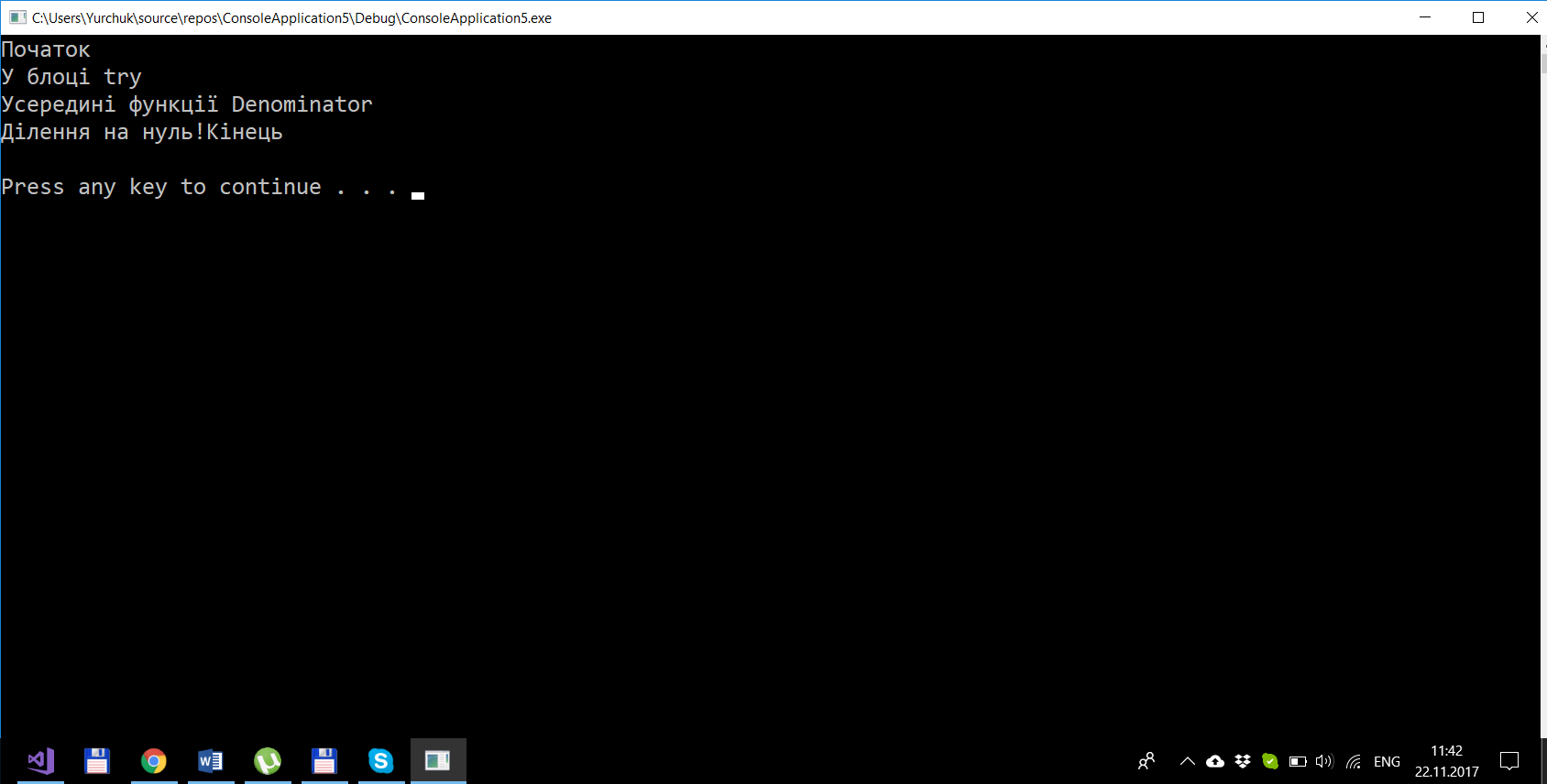
if (i>10) throw 10;

cout << "Кінець функції Denominator\n";

return i;

}

Результат демонструє декілька важливих особливостей, властивим функціям, що спричиняють, але не обробляють виняткову ситуацію.



По-перше, блоки try і catch нерозривні. Не можна помістити блок try у функцію, залишивши блок catch у функції main(). Необхідно або обробити виняткову ситуацію усередині функції, як це зроблено в одному із попередніх прикладів, або перенести обробку в модуль виклику. У першому випадку функція, завершивши обробку, повертає визначене її специфікацією значення, а в другому — виняткову ситуацію. Таким чином, можна обійти обмеження мови С++, відповідно до якого функція може повертати лише одне значення, тип якого визначений заздалегідь. По-друге, механізм обробки виняткових ситуацій дозволяє створювати альтернативні значення, що повертаються. По-третє, функції можуть генерувати декілька виняткових ситуацій. Згенерувавши одну з них, вони негайно припиняють своє виконання і повертають виняткову ситуацію в модуль виклику. Необхідно враховувати, що присвоювання m=Denominator(0) чи n=Denominator(11) у цьому випадку не виконуються.

Представимо тепер ланцюжок викликів функцій. Ланцюгове генерування виняткових ситуацій**:** перший варіант

#include "windows.h"

#include <iostream>

using namespace std;

int Check(int);

int Divide(int, int);

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n = 10, m = 0, l;

cout << "Початок\n"; l = Divide(n, m); cout << "Кінець\n";

cout << endl;

system("pause");

return 0;

}

int Check(int i) {

cout << "Усередині функції Check\n";

if (i == 0) throw("Ділення на нуль усередині функції Check!");

cout << "Кінець функції Check\n";

return i;

}

int Divide(int n, int m)

{

cout << "Усередині функції Divide\n";

try { m = Check(m); }

catch (const char\* s) {

printf("%s\n", s);

return 1;

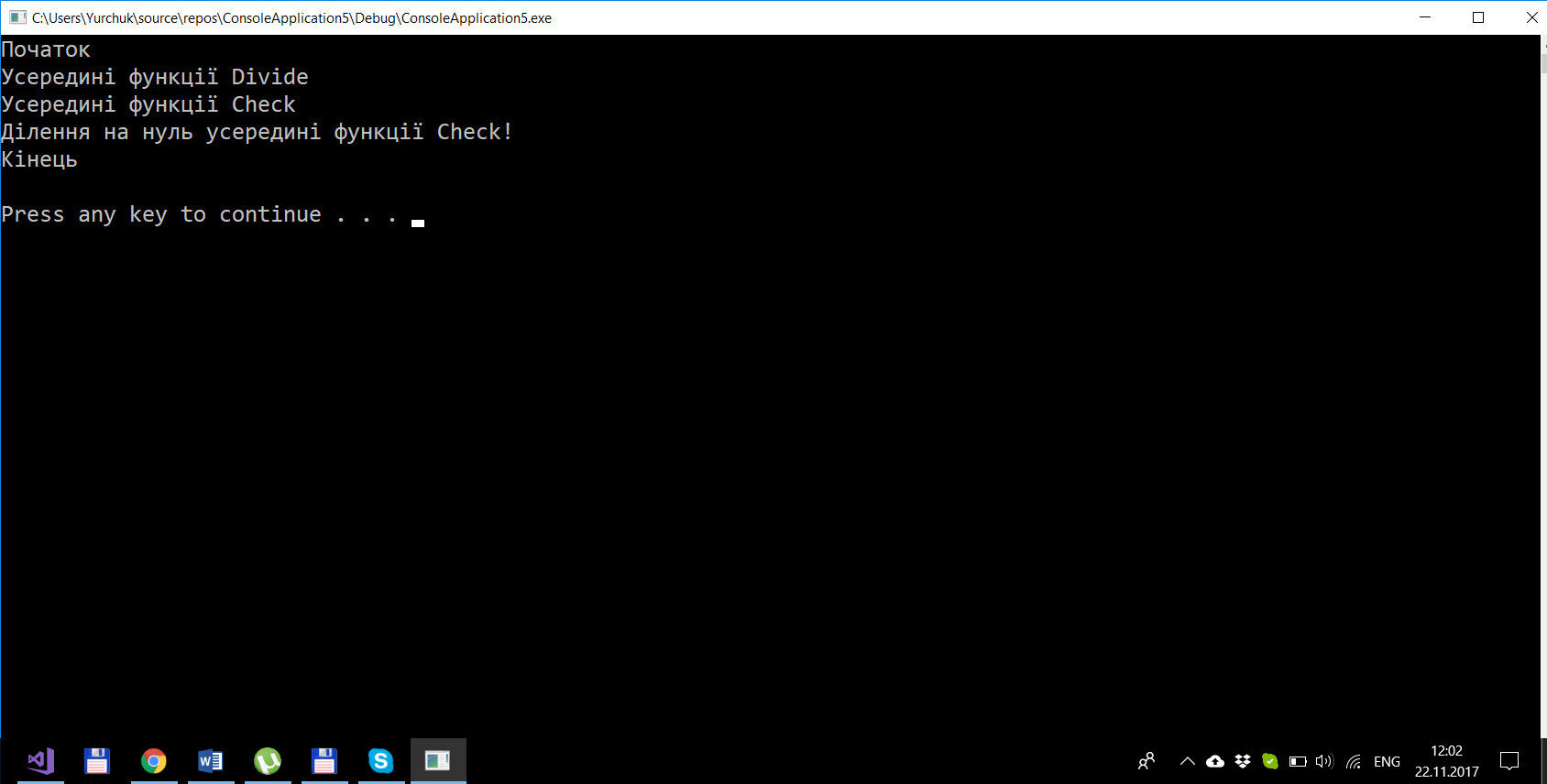
}

cout << "Кінець функції Divide\n";

return n / m;

}

Простежимо за передачею виняткової ситуації.



При виклику функції Divide() перевіряється знаменник m. Для цього викликається функція Check(). Якщо знаменник дорівнює нулю, усередині цієї функції генерується виняткова ситуація, що має тип const char\*. Обробка цієї виняткової ситуації усередині функції Check() не передбачена, тому вона передається нагору по ланцюжку викликів — функції Divide(). Потім керування передається функції main(), і виконання програми завершується.

Перенесемо обробку виняткової ситуації у функцію main().

Ланцюгове генерування виняткових ситуацій**:** другий варіант

#include "windows.h"

#include <iostream>

using namespace std;

int Check(int);

int Divide(int, int);

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n = 10, m = 0, l;

cout << "Початок\n";

try

{

cout << "Усередині функції try\n"; l = Divide(n, m);

}

catch (const char\* s) {

cout<<s<<endl;

}

cout<<"Кінець\n";

cout << endl;

system("pause");

return 0;

}

int Check(int i) {

cout << "Усередині функції Check\n";

if (i == 0) throw("Ділення на нуль усередині функції main!"); cout << "Кінець функції Check\n";

return i;

}

int Divide(int n, int m) {

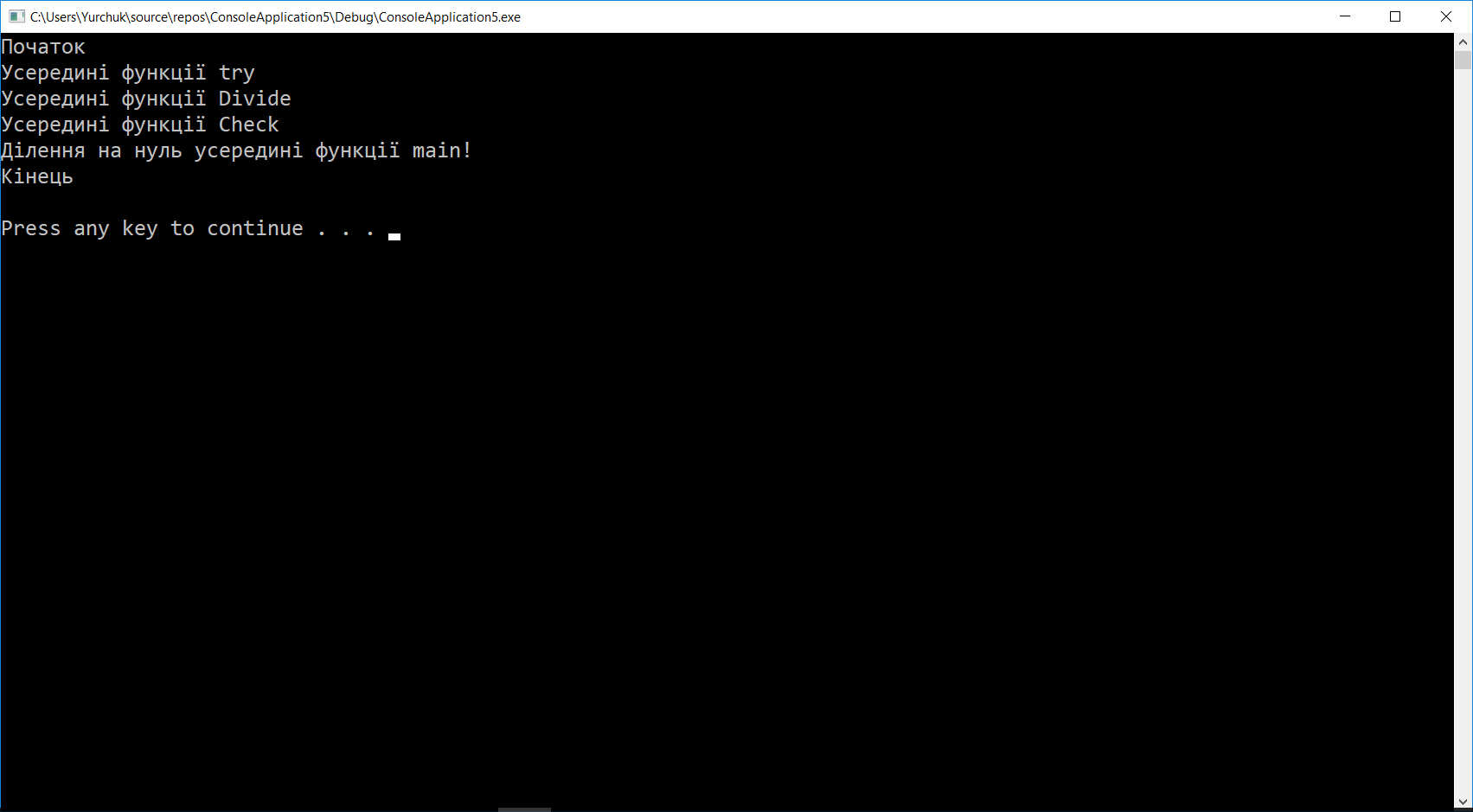
cout<<"Усередині функції Divide\n"; m = Check(m);

cout<<"Кінець функції Divide\n";

return n / m;

}

Результат роботи цієї функції такий.



Оскільки усередині функцій Check() і Divide() обробка виняткової ситуації не передбачена, вона передається в головний модуль, про що свідчить представлена нижче рядок.