Виняткові ситуації

Одним з найбільш яскравих утілень принципу об'єктно-орієнтованого програмування є механізм обробки виняткових ситуацій у мові C++. 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 проявлятися наслідки 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 C++ 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 C++ 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.

Генерація винятків.

Для використання оператора throw застосовується ключове слово throw, а за ним 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. Наприклад:

```
throw - 1; // генерація винятку типу int
throw ENUM_INVALID_INDEX; // генерація винятку типу enum
throw "Can not take square root of negative number"; // генерація винятку типу const char*
(рядок C-style)
throw dX; // генерація винятку типу double (змінна типу double, яка була визначена раніше)
throw MyException("Fatal Error"); // генерація винятку з використанням об'єкту класу
MyException
```

Кожен з цих рядків сигналізує про те, що виникла якась помилка, яку потрібно обробити.

Пошук винятків

Генерація винятків — це лише одна частина процесу обробки винятків. Повернемося до нашої аналогії з баскетболом: як тільки просвистів арбітр, що відбувається далі? Гравці зупиняються, і гра тимчасово припиняється. Звичайний хід гри порушений.

У мові C++ ми використовуємо ключове слово try для визначення блоку стейтментів (так званого «блоку try»). Блок try діє як спостерігач в пошуках винятків, які були викинуті будь-яким з операторів в цьому ж блоці try, наприклад:

```
try
{
	// Тут ми пишемо стейтменти, які генеруватимуть наступний виняток
	throw - 1; // типовий стейтмент throw
}
```

Обробка винятків

Поки арбітр не оголосить штрафний кидок, і поки цей штрафний кидок не буде виконано, гра не відновиться. Іншими словами, штрафний кидок повинен бути “оброблений” до відновлення гри.

Фактично, обробка винятків — це робота блоку(ів) catch. Ключове слово catch використовується для визначення блоку коду (так званого «блоку catch»), який обробляє винятки певного типу даних.

Ось приклад блоку catch, який обробляє (ловить) винятки типу int:

```
catch (int a)
{
	// Обробляємо виняток типу int
	cerr << "We caught an int exception with value" << a << '\n';
}
```

Блоки try і catch працюють разом. Блок try виявляє будь-які винятки, які були викинуті в ньому, і направляє їх до відповідного блоку catch для обробки. Блок try повинен мати, принаймні, один блок catch, який знаходиться відразу ж за ним, але також може мати і кілька блоків catch, розміщених послідовно (один за одним).

Як тільки виняток було спіймано блоком try і направлено в блок catch для обробки, він вважається обробленим (після виконання коду блоку catch), і програма відновлює своє виконання.

Параметри catch працюють так само, як і параметри функції, причому параметри одного блоку catch можуть бути доступні і в іншому блоці catch (який знаходиться за ним). Винятки фундаментальних типів даних можуть бути спіймані по значенню (параметром блоку

catch є значення), але винятки нефундаментальних типів даних повинні бути спіймані по константному посиланню (параметром блоку catch є константне посилання), щоб уникнути непотрібного копіювання.

Як і у випадку з функціями, якщо параметр не використовується в блоці catch, то ім'я змінної можна не вказувати:

```
catch (double) // примітка: Ми не вказуємо ім'я змінної, тому що в цьому немає необхідності (ми її ніде в блоці не використовуємо)
{
	// Обробляємо виняток типу double тут
	cerr << "We caught an exception of type double" << '\n';
}
```

Це запобіжить виведенню попереджень від компілятора про невикористані змінні.

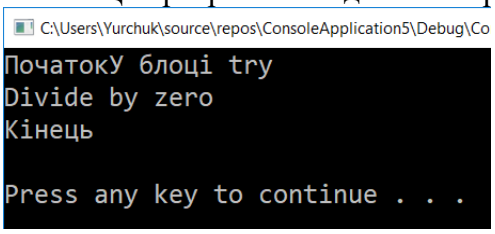
Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort(). Утім, іноді виняткова ситуація не є небезпечна. У цьому випадку можна виправити помилку (наприклад, привласнити нульовому 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";
```

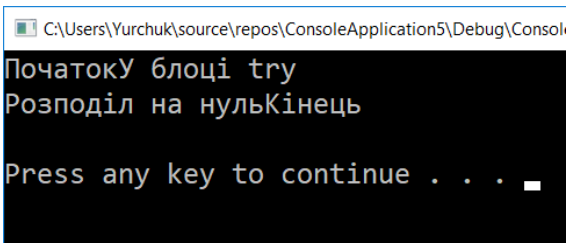
У цій програмі ми зробили цілком “природну” помилку — забули поставити зірочку в оголошенні аргументу. Тепер блок catch очікує виняткову ситуацію, що представляє собою константний символ, а не вказівник. Ця помилка приводить до аварійного завершення роботи програми.

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;
}
```

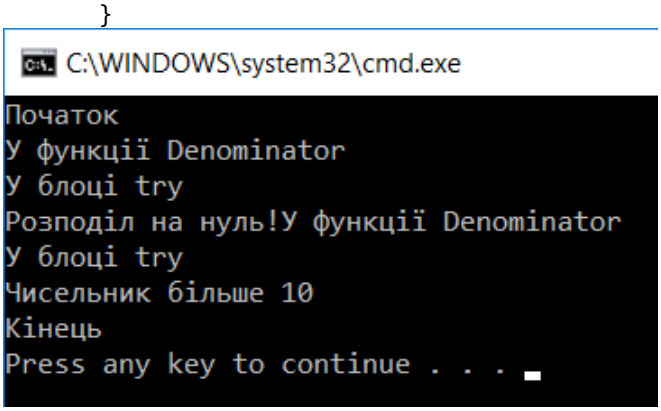
У цій програмі виняткова ситуація генерується у функції Denominator(), яка викликається в блоці try. Завдяки цьому результати роботи програми цілком збігаються з попередніми.



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

Розміщення блоку try усередині функції

```
int Denominator(int);
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);


	int n = 10, m;
	cout << "Початок\n";
	m = Denominator(0);
	n = Denominator(11);
	cout << "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().

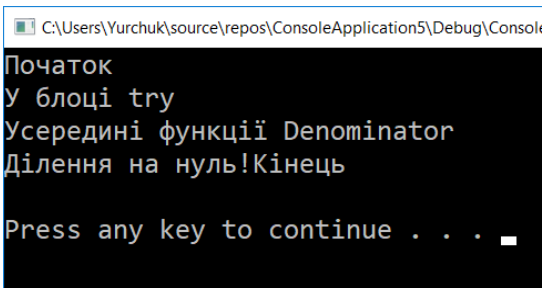
Розглянемо тепер приклад, у якому функція Denominator() лише генерує виняткові ситуації, а їх обробка здійснюється у функції main().

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 C++, 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.

Приклад розкручування стеку

Коли функція викликає іншу функцію, а та в свою чергу ще іншу функцію, то такі функції поміщаються в стек, відповідно вверху стеку знаходиться функція, яка обробляється в даний момент часу, при завершенні роботи вона видаляється із стеку, на її місце підсувається попередня функція, і так далі. Такий процес називається розкручування стеку.

Тут у нас вже більший стек. Хоча все здається занадто складним, але насправді це не

так:

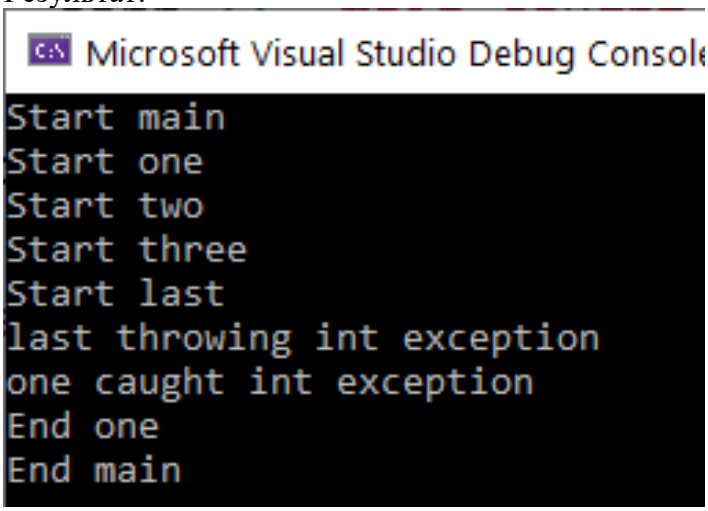
main() викликає one();
one() викликає two();
two() викликає three();
three() викликає last();
last() викидає виняток.

Дивимося:

```
#include <iostream>
using namespace std;
void last() // викликається функцією three()
{
    cout << "Start last\n";
   cout << "last throwing int exception\n";
    throw - 1;
   cout << "End last\n";

}

void three() // викликається функцією two()
{
   cout << "Start three\n";
    last();
    cout << "End three\n";
}

void two() // викликається функцією one()
{
   cout << "Start two\n";
    try
    {
        three();
    }
    catch (double)
    {
       cerr << "two caught double exception\n";
    }
    cout << "End two\n";
}

void one() // викликається функцією main()
{
   cout << "Start one\n";
    try
    {
        two();
    }
    catch (int)
    {
        cerr << "one caught int exception\n";
    }
    catch (double)
    {
        cerr << "one caught double exception\n";
    }
   cout << "End one\n";
}

int main()
{
   cout << "Start main\n";
    try
    {
        one();
    }
    catch (int)
    {
```

```
        cerr << "main caught int exception\n";
    }
    cout << "End main\n";

    return 0;
}
```

Погляньте на цю програму ще раз. Чи можете ви зрозуміти, що виведеться на екран? Результат:



Розглянемо хід виконання програми детально. Думаю не потрібно пояснювати виведення рядків Start. Функція last() виводить last throwing int exception, а потім викидає виняток типу int. Ось де починається найцікавіше.

Оскільки функція last() не обробляє винятки самостійно, то стек починає розкручуватися. Функція last() негайно завершує своє виконання, і точка виконання повертається назад в caller (в функцію three()).

Функція three() не обробляє жодних винятків, тому стек розкручується далі, виконання функції three() припиняється, і точка виконання повертається в two().

Функція two() має блок try, в якому знаходиться виклик three(), тому компілятор намагається знайти обробник винятків типу int, але, так як його не знаходить, точка виконання повертається назад в one(). Зверніть увагу, компілятор не виконує неявну конвертацію, щоб зіставити виняток типу int з обробником типу double.

Функція one() також має блок try з викликом two() всередині, тому компілятор дивиться, чи є відповідний обробник catch. Є — функція one() обробляє виняток і виводить one caught int exception.

Оскільки виняток було оброблено, то точка виконання переміщається в кінець блоку catch всередині one(). Це означає, що one() виводить End one, а потім завершує своє виконання, як зазвичай.

Точка виконання повертається назад в main(). Хоча main() має обробник винятків типу int, але наш виняток вже був оброблений функцією one(), тому блок catch всередині main() не виконується. Функція main() виводить End main, а потім завершує своє виконання.

З цієї програми можна зробити кілька цікавих висновків:

- По-перше, безпосередній caller, який викликає функцію, в якій викидається виняток, не зобов'язаний обробляти цей виняток, якщо він цього не хоче. У прикладі, наведеному вище, функція three() не обробляє виняток, який генерується функцією last(). Вона делегує цю відповідальність на іншого caller-а зі стеку.
- По-друге, якщо блок try не має обробника catch відповідного типу, то розкручування стеку відбувається так, наче цього блоку try не було взагалі. У прикладі, наведеному вище, функція two() не обробляє виняток, тому що у неї немає відповідного обробника catch.
- По-третє, коли виняток оброблено, виконання коду продовжується як зазвичай, починаючи з кінця блоку catch (в якому цей виняток був оброблений). У прикладі, наведеному вище, функція one() обробила виняток, а потім продовжила своє виконання

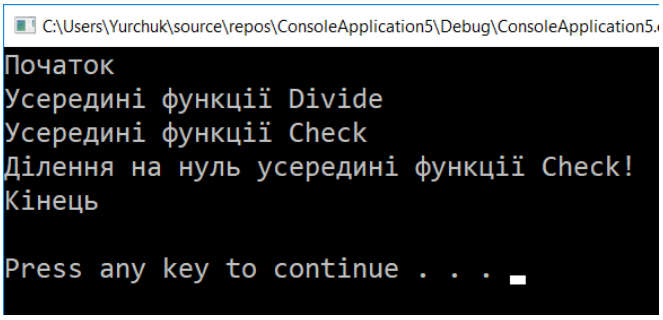
виведенням рядка End one. На той час, коли точка виконання повертається назад в функцію main(), виняток вже був згенерований і оброблений. Функція main() виконується так, наче цього винятку не було взагалі!

Розкручування стеку є дуже корисним механізмом, тому що дозволяє функціям не обробляти винятки, якщо вони цього не хочуть. Операція розкручування стеку виконується до тих пір, поки не буде виявлений відповідний блок catch! Таким чином, ми можемо самі вирішувати, де саме слід обробляти винятки.

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) {
		cout << s;
		return 1;
	}
	cout << "Кінець функції Divide\n";
	return n / m;
}
```

Простежимо за передачею виняткової ситуації.



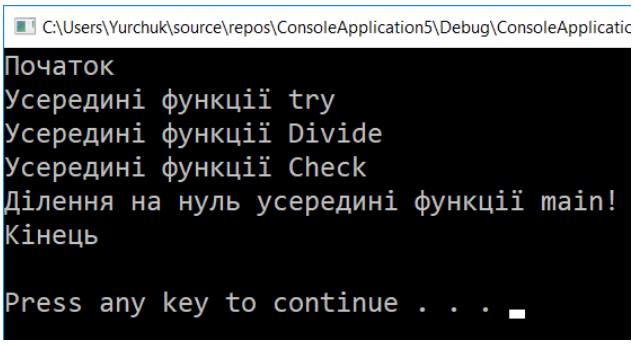
При виклику функції Divide() перевіряється знаменник m. Для цього викликається функція Check(). Якщо знаменник дорівнює нулю, усередині цієї функції генерується виняткова ситуація, що має тип const char*. Обробка цієї виняткової ситуації усередині функції Check() не передбачена, тому вона передається 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 представлений нижче рядок.

Неспіймані винятки

У наступному прикладі функція mySqrt() викидає виняток і припускає, що його хтось обробить. Але що станеться, якщо цього ніхто не зробить?

Ось наша програма обчислення квадратного кореня числа без блоку try в функції main():

```
#include <iostream>
#include <cmath> // для sqrt()
using namespace std;
// Окрема функція обчислення квадратного кореня числа
double mySqrt(double a)
```

```
{
    // Якщо користувач ввів від'ємне число,
    if (a < 0.0)
        throw "Can not take sqrt of negative number"; // то генерується виняток типу const
char*

    return sqrt(a);
}

int main()
{
   cout << "Enter a number: ";
    double a;
    cin >> a;

    // Тут немає ніякого обробника винятків!
   cout << "The sqrt of " << a << " is " << mySqrt(a) << '\n';

    return 0;
}
```

Тепер припустимо, що користувач ввів -5, і mySqrt(-5) згенерувало виняток. Функція mySqrt() не обробляє свої винятки самостійно, тому стек починає розкручуватися, і точка виконання повертається назад в функцію main(). Але, оскільки в main() також немає обробника винятків, виконання main() і всієї програми припиняється.

Коли main() завершує своє виконання з необробленим винятком, то операційна система зазвичай повідомляє нас про те, що відбулася помилка необробленого винятку. Як вона це зробить — залежить від кожної операційної системи окремо:

або виведе повідомлення про помилку;
або відкриє діалогове вікно з помилкою;
або просто збій.

Це те, що ми не повинні допускати, як програмісти!

Обробники всіх типів винятків

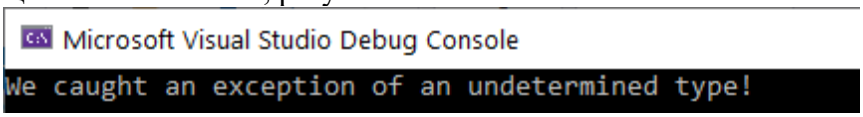
А тепер загадка: “Функції можуть генерувати винятки будь-якого типу даних, і, якщо виняток не спіймано, це призведе до розкручування стеку і потенційному завершенню виконання програми. Оскільки ми можемо викликати функції, не знаючи їх реалізації (і, відповідно, які винятки вони можуть генерувати), то як ми можемо цьому запобігти?”.

На щастя, мова C++ надає нам механізм виявлення/обробки всіх типів винятків — обробник catch-all. Обробник catch-all працює так само, як і звичайний блок catch, за винятком того, що замість обробки винятків певного типу даних, він використовує еліпсис (...) в якості типу даних.

А як ми вже знаємо, еліпсиси можуть використовуватися для передачі аргументів будь-якого типу даних в функцію. У цьому контексті вони представляють собою винятки будь-якого типу даних. Ось простий приклад:

```
#include <iostream>
using namespace std;
int main()
{
	try
	{
		throw 7; // генерується виняток типу int
	}
	catch (double a)
	{
		cout << "We caught an exception of type double: " << a << '\n';
	}
	catch (...) // обробник catch-all
	{
		cout << "We caught an exception of an undetermined type!\n";
	}
}
```

Оскільки для типу int не існує спеціального обробника catch, то обробник catch-all ловить цей виняток. Отже, результат:



Обробник catch-all повинен знаходитися останнім в ланцюжку блоків catch. Це робиться для того, щоб винятки спочатку могли бути спіймані обробниками catch, адаптованими до конкретних типів даних (якщо вони взагалі існують). У Visual Studio це контролюється, щодо інших компіляторів не впевнений, чи є таке обмеження.
Часто блок обробника catch-all залишають порожнім:

```
catch (...) {} // ігноруюься будь-які непередбачувані винятки
```

Цей обробник ловить будь-які непередбачені винятки і запобігає розкручуванню стеку (і потенційному завершенню виконання програми), але тут він не виконує ніякої обробки винятків.

Використання обробника catch-all в функції main()

Розглянемо наступну програму:

```
#include <iostream>
using namespace std;
int main()
{

    try
    {
        runGame();
    }
    catch (...)
    {
        cerr << "Abnormal termination\n";
    }

    saveState(); // зберігаємо поточний стан гравця
    return 1;
}
```

В цьому випадку, якщо функція runGame() або будь-яка інша з функцій, які викликаються в runGame(), викине виняток, який не буде спіймано функціями в стеці вище, то, в кінцевому підсумку, він потрапить в обробник catch-all. Це запобіжить завершенню виконання функції main() і дасть нам можливість вивести повідомлення із зазначенням помилки на наш розсуд, а потім зберегти стан користувача до виходу з програми. Це може бути корисно для виявлення і усунення непередбачених проблем.

Специфікації винятків

Специфікації винятків — це механізм оголошення функцій із зазначенням того, чи генеруватиме функція винятки (і які саме) чи ні. Це може бути корисно при визначенні необхідності розміщення виклику функції в блоці try.

Існують три типи специфікації винятків, кожен з яких використовує так званий синтаксис throw (…).

По-перше, ми можемо використовувати порожній оператор throw для позначення того, що функція не генерує ніяких винятків, які виходять за її межі:

```
int doSomething() throw(); // не генеруються винятки
```

Зверніть увагу, функція doSomething() все ще може генерувати винятки, тільки обробляти вона повинна їх самостійно. Будь-яка функція, оголошена з використанням throw() (як у вищенаведеному прикладі), повинна негайно припинити виконання програми, якщо вона спробує згенерувати виняток, який призведе до розкручування стеку. Іншими словами, ми повідомляємо, що всі винятки функції doSomething(), функція doSomething() оброблятиме самостійно.

По-друге, ми можемо використовувати оператор throw із зазначенням типу винятку, який може генерувати ця функція:

```
int doSomething() throw(double); // можуть генеруватися винятки типу double
```

Нарешті, ми можемо використовувати еліпсис з оператором throw для позначення того, що функція може генерувати різні типи винятків:

```
int doSomething() throw(...); // можуть генеруватися будь-які винятки
```

Через погану (неповну) реалізацію і сумісність з компіляторами, і з огляду на той факт, що специфікації винятків більше нагадують заяви про наміри, ніж гарантії чого-небудь, і те, що вони погано сумісні з шаблонами функцій, і те, що більшість програмістів C++ не знають про їх існування, призводить до того, що використовувати специфікації винятків не рекомендується.