ВИНЯТКОВІ СИТУАЦІЇ

Механізм обробки виняткових ситуацій

Тонкощі обробки виняткових ситуацій

Стандартні виняткові ситуації

Одним з найбільш яскравих втілень принципу ООП 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 допустимого чи переповнення пам'яті), а іноді їхня причина не залежить від програміста (скажімо, розрив зв'язку при мережевому з'єднанні). У кожній з цих ситуацій реакція програми непередбачувана. Іноді програма завершує виконання і лише після закінчення деякого інтервалу часу починають позначатися 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 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.ін. 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 повинен заключити 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 і 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(типN аргумент)

{ // тіло блоку catch

}

Розмір блоку try не обмежений. У нього можна помістити 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 і у блоці try не виникло жодної виняткової ситуації, усі блоки catch будуть проігноровані. Якщо в програмі виникла подія, яку програміст вважає небажаною, оператор throw згенерує виняткову ситуацію. Для цього оператор throw повинен знаходитися усередині блоку try або усередині функції, викликуваної усередині блоку try.

Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення і обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort().

Утім, іноді виняткова ситуація не є небезпечною. У цьому випадку можна виправити помилку (наприклад, привласнити нульовому знаменнику ненульове значення) і продовжити виконання програми.

Розглянемо найпростіший приклад.

int main()

{ int n = 10, m = 0;

printf("Початок\n");

try

{ printf("У блоці try\n");

If (m==0) throw "Divide by zero";

else n=n/m;

printf("Подальша частина блоку не виконується!");

}

catch (const char\* s)

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

}

printf("Кінець\n");

getch();

}

Ця програма виводить на екран наступні рядки.

Початок

Усередині блоку try

Divide by zero

Кінець

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 функція printf() виводить на екран рядок Кінець, і програма завершує свою роботу.

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

int main()

{ int n = 10, m = 0;

printf("Початок\n");

try

{ printf("У блоці try\n");

if(m==0) throw "Ділення на нуль";

else n=n/m;

printf("Подальша частина блоку не виконується!");

}

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

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

}

printf("Кінець\n");

getch();

}

У цій програмі зроблена цілком “природна” помилка — не поставлена зірочка 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.

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

int Denominator(int);

int main()

{ int n = 10, m;

printf("Початок\n");

try

{ printf("У блоці try\n");

m=Denominator(0);

printf("Подальша частина блоку не виконується!");

}

catch (const char\* s)

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

}

printf("Кінець\n");

getch();

}

int Denominator(int i)

{ if(i==0)throw "Ділення 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.

int Denominator(int);

int main()

{ int n = 10, m;

printf("Початок\n");

m=Denominator(0);

n = Denominator(11);

printf("Кінець\n");

getch();

}

int Denominator(int i)

{ printf("У функції Denominator\n");

try

{ printf("У блоці try\n");

if(i==0) throw("Розподіл на нуль!");

if(i>10) throw 10;

printf("Подальша частина блоку не виконується!");

}

catch (const char\* s)

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

}

catch (int n)

{ printf("Чисельник більше %d\n",n);

}

return i;

}

На екрані з'являться наступні рядки.

Усередині функції Denominator

У блоці try

Ділення на нуль!

Усередині функції Denominator

У блоці try

Чисельник більше 10

Кінець

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 кожному виклику функції Denominator().

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

int Denominator(int);

int main()

{ int n = 10, m;

printf("Початок\n");

try

{ printf("У блоці try\n");

m=Denominator(0);

n = Denominator(11);

printf("Подальша частина блоку не виконується!");

}

catch (const char\* s)

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

}

catch (int n)

{ printf("Чисельник більше 10 %d\n",n);

}

printf("Кінець\n");

getch();

}

int Denominator(int i)

{ printf("Усередині функції Denominator\n");

if(i==0) throw("Ділення на нуль!");

if(i>10) throw 10;

printf("Кінець функції Denominator\n");

return i;

}

Результат демонструє декілька важливих особливостей, властивих функціям, що генерують, але не обробляють виняткову ситуацію.

Початок

Усередині блоку try

Усередині функції Denominator

Ділення на нуль!

Кінець

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, 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.

int Check(int);

int Divide(int, int);

int main()

{ int n = 10, m=0, l;

printf("Початок\n");

l = Divide(n,m);

printf("Кінець\n");

getch();

}

int Check(int i)

{ printf("Усередині функції Check\n");

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

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

try

{ m=Check(m);

}

catch (const char\* s)

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

return 1;

}

printf("Кінець функції Divide\n");

return n/m;

}

Простежимо за передачею виняткової ситуації:

Початок

Усередині функції Divide

Усередині блоку try

Усередині функції Check

Ділення на нуль усередині функції Check!

Кінець

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().

int Check(int);

int Divide(int, int);

int main()

{ int n = 10, m=0, l;

printf("Початок\n");

try

{ printf("Усередині блоку try\n");

l = Divide(n,m);

}

catch (const char\* s)

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

}

printf("Кінець\n");

getch();

}

int Check(int i)

{ printf("Усередині функції Check\n");

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

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

m=Check(m); printf("Кінець функції Divide\n"

);

return n/m;

}

Результат роботи цієї функції такий.

Початок

Усередині блоку try

Усередині функції Divide

Усередині функції Check

Ділення на нуль усередині функції main!

Кінець

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

Ділення на нуль усередині функції main!

## Перехоплення класів виняткових ситуацій

Як відзначено вище, *виняткова ситуація* — це об'єкт, що може мати будь-який тип, як вбудований, так і тип користувача. Створення класів виняткових ситуацій дозволяє програмісту точніше визначати реакцію програми на небажану ситуацію. Розглянемо приклад, що ілюструє ця теза.

class Error

{ public:

int m;

Error() { printf("Помилка\n");}

Error(int x):m(x) {}

void Message() { printf("Ділення на нуль!"); }

};

class Rational

{ int n;

int m;

public:

Rational(int x, int y)

{ n = x;

if(y==0)throw Error(y);

else n = y;

}

~Rational() { printf("Dtor Rational"); }

};

int main()

{ try

{ Rational q(1,0);

}

catch(Error& zero)

{ zero.Message();

}

getch();

}

Клас Rational реалізує концепцію раціонального числа. Зрозуміло, необхідно заборонити створення раціональних чисел, у яких знаменник дорівнює нулю. Для цього передбачений клас Error, що представляє собою виняткову ситуацію. Його об'єкт створюється тільки в тому випадку, якщо поле m в об'єкті класу Rational дорівнює нулю.

В цьому прикладі має місце спроба створити об'єкт виду 1/0. Конструктор класу Rational згенерував виняткову ситуацію класу Error, що була по посиланню передана в розділ catch. У цьому розділі відбувається звертання до функції Message() — члена класу Error. У підсумку, на екрані з'явиться таке повідомлення.

Ділення на нуль!

Зверніть увагу на те, що виняткова ситуація передається за посиланням. Це зовсім не обов’язково, але дуже бажано, оскільки передача за посиланням ефективніша, ніж передача за значенням.

## Ієрархія виняткових ситуацій

Оскільки виняткова ситуація може бути об'єктом класу, у мові С++ існує можливість створювати ієрархію виняткових ситуацій. У цьому випадку блок catch перехоплює об'єкти не тільки базового, але і похідних класів. При генерації похідних виняткових ситуацій це приводить до непорозумінь — їх перехоплює блок catch, призначений для обробки базових виняткових ситуацій.

class ErrorBase

{ public:

long m;

ErrorBase() { printf("ErrorBase\n"); }

ErrorBase(long x):m(x) { printf("ErrorBase\n"); }

void Message() { printf("Ділення на нуль!");}

};

class ErrorDerived:public ErrorBase

{ public:

long m;

ErrorDerived() { printf("ErrorDerived\n"); }

ErrorDerived(long x):m(x) { printf("ErrorDerived\n"); }

void Message() { printf("Ділення на нескінченність!"); }

};

class Rational

{ long n;

long m;

public:

Rational(long x, long y)

{ n = x;

if(y==0)throw ErrorBase(y);

else n = y;

if(y>=1000000)throw ErrorDerived(y);

else n = y;

}

~Rational(){printf("Dtor Rational");}

};

int main()

{ try

{ Rational q(1,1000000000);

}

catch(ErrorBase& zero)

{ zero.Message();

}

catch(ErrorDerived& infinity)

{ infinity.Message();

}

getch();

}

У цій програмі оголошена ієрархія виняткових ситуацій — базовий клас ErrorBase і похідний від нього клас ErrorDerived. Виняткові ситуації базового класу генеруються, якщо конструктор класу Rational намагається створити об'єкт із нульовим знаменником, а похідний клас ErrorDerived описує реакцію програми, коли знаменник занадто великий (більше мільйона). Програма виводить на екран наступні рядки:

ErrorBase

ErrorDerived

Ділення на нуль!

Як бачимо, при спробі створити об'єкт, знаменник якого більше мільйона, була згенерована ситуація ErrorDivide, однак її перехопив блок catch, надбудований над базовим класом ErrorDivide.

Для розв’язання цієї проблеми необхідно розмістити блок catch, що відповідає похідному класу, вище блоку, призначеного для перехоплення об'єктів класу ErrorBase:

class ErrorBase

{ public:

long m;

ErrorBase() { printf("ErrorBase\n");}

ErrorBase(long x):m(x) {printf("ErrorBase\n");}

virtual void Message() { printf("Ділення на нуль!");}

};

class ErrorDerived:public ErrorBase

{ public:

long m;

ErrorDerived(){ printf("ErrorDerived\n");}

ErrorDerived(long x):m(x){printf("ErrorDerived\n");}

void Message(){ printf("Ділення на нескінченність!");}

};

class Rational

{ long n;

long m;

public:

Rational(long x, long y)

{ n = x;

if(y==0)throw ErrorBase(y);

else n = y;

if(y>=1000000)throw ErrorDerived(y);

else n = y;

}

~Rational() { printf("Dtor Rational"); }

};

int main()

{ try { Rational q(1,1000000000); }

catch(ErrorDerived& infinity)

{ infinity.Message();

}

catch(ErrorBase& zero)

{ zero.Message();

}

getch();

}

На екрані ми побачимо наступні рядки.

ErrorBase ErrorDerived Ділення на нуль!

Зауважимо, що ця задача має ще одне розв’язок — можна оголосити функцію-член Message() віртуальною. У цьому випадку оператор catch, призначений для обробки виняткових ситуацій базового типу, буде як і раніше перехоплювати об'єкти похідного типу, вважаючи їх базовими. Однак механізм заміщення віртуальних функцій-членів базового класу дозволяє правильно обробити виняткову ситуацію. Утім, цей спосіб вимагає визначеної обережності — вся обробка повинна бути передбачена у функціях-членах похідного класу. Не слід забувати, що керування передається в блок catch, призначений для перехоплення базових виняткових ситуацій!

class ErrorBase

{ public:

long m;

ErrorBase() { printf("ErrorBase\n"); }

ErrorBase(long x):m(x) { printf("ErrorBase\n"); }

virtual void Message() { printf("Ділення на нуль!\n"); }

};

class ErrorDerived:public ErrorBase

{ public:

long m;

ErrorDerived() { printf("ErrorDerived\n"); }

ErrorDerived(long x):m(x) { printf("ErrorDerived\n"); }

void Message() { printf("Ділення на нескінченність!\n"); }

};

class Rational

{ long n;

long m;

public:

Rational(long x, long y)

{ n = x;

if(y==0)throw ErrorBase(y);

else n = y;

if (y>=1000000) throw ErrorDerived(y);

else n = y;

}

~Rational() { printf("Dtor Rational");}

};

int main()

{ try

{ Rational q(1,1000000000); }

catch(ErrorBase& zero)

{ zero.Message();

printf("Усередині блоку сatch для класу ErrorBase!\n");

return -1;

}

catch(ErrorDerived& infinity)

{ infinity.Message();

printf("Усередині блоку catch для класу ErrorDerived!\n");

}

getch();

}

Тепер програма виводить на екран очікуваний результат, хоча керування передається блоку catch, призначеному для перехоплення базових ситуацій, оскільки він розташований вище блоку, що відповідає похідній винятковій ситуації.

ErrorBase

ErrorDerived

Ділення на нуль!

Усередині блоку catch для класу ErrorBase!

Зверніть увагу на те, що перехоплення виняткових ситуацій, що виникли в конструкторі, припиняє створення об'єкта. З цієї причини деструктор наприкінці програми не викликається.

## Тонкощі обробки виняткових ситуацій

Розглянемо декілька корисних прийомів, що дозволяють ефективно використовувати механізм виняткових ситуацій.

## Тотальне перехоплення виняткових ситуацій

Іноді ретельна деталізація виняткових ситуацій не потрібна. Наприклад, у попередніх прикладах їх обробка проводилася майже однаково, за винятком супутніх повідомлень про помилки. Отже, було б зручно, якби блок catch можна було настроїти на будь-яку виняткову ситуацію. Зробити це дозволяє наступна конструкція:

catch(...)

{ // перехоплення усіх виняткових ситуацій

}

Трикрапка в дужках означає, що блок catch здатний перехопити і обробити будь-яку виняткову ситуацію.

Повернемося до попереднього прикладу і замінимо блоки catch новою конструкцією.

void Message();

class ErrorBase

{ public:

long m;

ErrorBase(){ printf("ErrorBase\n");}

ErrorBase(long x):m(x){printf("ErrorBase\n");}

virtual void Message(){ printf("Ділення на нуль!");}

};

class ErrorDerived:public ErrorBase

{ public:

long m;

ErrorDerived(){ printf("ErrorDerived\n");}

ErrorDerived(long x):m(x){printf("ErrorDerived\n");}

void Message(){ printf("Ділення на нескінченність!!");}

};

class Rational

{ long n;

long m;

public:

Rational(long x, long y)

{ n = x;

if(y==0)throw ErrorBase(y);

else n = y;

if(y>=1000000)throw ErrorDerived(y);

else n = y;

}

~Rational(){printf("Dtor Rational");}

};

int main()

{ int n = 5;

for (int i = 1; i<=n; i++)

{ try

{ if(i%2) Rational q(1,0);

else Rational q(1,100000000);

}

catch(...)

{ Message(); }

}

getch();

}

void Message() { printf("Виняткова ситуація!\n"); }

Ця програма генерує різні виняткові ситуації: при непарних індексах циклу i — базову виняткову ситуацію класу ErrorBase, а при парних — похідну виняткову ситуацію класу ErrorDivide. Усі вони перехоплюються тим самим блоком catch.

ErrorBase

Виняткова ситуація!

ErrorBase

ErrorDerived

Виняткова ситуація!

ErrorBase

Виняткова ситуація!

ErrorBase

ErrorDerived

Виняткова ситуація!

ErrorBase

Виняткова ситуація!

Крім того, якщо програміст сумнівається в тому, що він передбачив усі можливі виняткові ситуації, після всіх розділів catch можна поставити розділ catch(...). У цьому випадку оператор catch(...) використовується для підстрахування, перехоплюючи виняткові ситуації, не передбачені програмістом. Це нагадує застосування мітки default в операторі switch.

Для ілюстрації цього прийому повернемося до попередньої програми. Нагадаємо, що виняткові ситуації базового класу ErrorBase генеруються при спробі створити об'єкт, що імітує раціональне число з нульовим знаменником, а похідний клас ErrorDerived описує ситуацію, коли знаменник більше мільйона. Ясно, що це не усі виняткові ситуації, що можуть виникнути. Наприклад, обмежений обсяг комп'ютерної пам'яті не дозволяє працювати з числами типу long, довжина яких перевищує максимальну. Звичайно, можна було б передбачити і це, написавши клас ErrorLong:

catch(const char\* s)

{ printf("%s",s);

}

І все ж, щоб перестрахуватися, можна поставити на останнє місце в ланцюжку блоків catch універсальний перехоплювач.

class ErrorBase

{ public:

long m;

ErrorBase(){ printf("ErrorBase\n");}

ErrorBase(long x):m(x){printf("ErrorBase\n");}

virtual void Message(){ printf("Ділення на нуль!\n");}

};

class ErrorDerived:public ErrorBase

{ public:

long m;

ErrorDerived(){ printf("ErrorDerived\n");}

ErrorDerived(long x):m(x){printf("ErrorDerived\n");}

void Message(){ printf("Ділення на нескінченність!\n");}

};

class Rational

{ long n;

long m;

public:

Rational(long x, long y)

{ if(x >= LONG\_MAX) throw "Занадто великий чисельник";

if(y >= LONG\_MAX) throw "Занадто великий знаменник";

n = x;

if(y==0)throw ErrorBase(y);

else n = y;

if(y>=1000000)throw ErrorDerived(y);

else n = y;

}

~Rational() { printf("Dtor Rational"); }

};

int main()

{ try

{ Rational q(1,LONG\_MAX); }

catch(ErrorBase& zero)

{ zero.Message();

printf("Усередині блоку catch для класу ErrorBase!\n");

}

catch(ErrorDerived& infinity)

{ infinity.Message();

printf("Усередині блоку catch для класу ErrorDerived!\n");

}

catch(...)

{ printf("Непередбачена виняткова ситуація!");

}

getch();

}

Оскільки блок catch(...) не має аргументів, у ньому важко передбачити точну реакцію на виниклу виняткову ситуацію, наприклад вивести діагностичний рядок. Мабуть, єдине, що залишається — припинити роботу програми і видати на екран відповідне повідомлення .

## Генерація виняткових ситуацій

Оголошуючи функцію, можна перелічити типи виняткових ситуацій, які їй дозволено генерувати. Типи виняткових ситуацій, не включені в список, функція генерувати не зможе. Спроба порушити це обмеження приведе до негайного припинення роботи програми за допомогою ланцюжка викликів стандартних функцій: unexpected() — abort(). Якщо список порожній, функція взагалі не повинна генерувати ніяких виняткових ситуацій.

Для оголошення списку дозволених виняткових ситуацій використовується наступна синтаксична конструкція:

тип\_ результату ім'я\_функції(аргументи) throw(виняткові ситуації)

{ // ...

}

Повернемося до демонстраційної програми, що використовує функцію Check().

#define MAX 1000000

int CheckZero(int) throw (const char\*);

int CheckMax(int) throw (int);

int Divide(int, int);

int main()

{ int n = MAX, m=0, l;

printf("Початок\n");

try

{ printf("Усередині блоку try\n");

l = Divide(n+1,m);

}

catch (const char\* s)

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

catch (int k)

{ printf("Число більше чи дорівнює %d\n",k);

}

printf("Кінець\n");

getch();

}

int CheckZero(int i) throw (const char\*)

{ printf("Усередині функції Check\n");

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

printf("Кінець функції Check\n");

return i;

}

int CheckMax(int i) throw (int)

{ printf("Усередині функції CheckMax\n");

if(i>=MAX) throw MAX;

printf("Кінець функції CheckMax\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=CheckMax(n);

m=CheckMax(m);

m=CheckZero(m);

printf("Кінець функції Divide\n");

return n/m;

}

У цій програмі, що виконує ділення двох цілих чисел, можливі наступні виняткові ситуації:

1) чисельник або знаменник перевищує максимально допустиме число;

2) знаменник дорівнює нулю.

Для порівняння числа з максимальним використовується функція CheckMax(), а для порівняння з нулем — функція CheckZero(). Функції CheckMax() дозволено генерувати виняткову ситуацію типу int, а функція CheckZero() може генерувати виняткову ситуацію типу const char\*. Якщо ці функції спробують згенерувати виняткову ситуацію, не зазначену в списку throw, виконання програми завершиться аварійно.

Початок

Усередині блоку try

Усередині функції Divide

Усередині функції CheckMax

Число більше 100000

Кінець

Припустимо тепер, що в програмі ані чисельник, ані знаменник не повинні дорівнювати нулю. Отже, функція, що виконує перевірку, повинна мати можливість генерувати обидві виняткові ситуації: число дорівнює нулю і число більше максимального. Модифікований варіант програми виглядає так:

#define MAX 1000000

int Check(int) throw (const char\*, int);

int Divide(int, int);

int main()

{ int n = 0, m=0, l;

printf("Початок\n");

try

{ printf("Усередині блоку try\n");

l = Divide(n,m);

}

catch (const char\* s)

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

}

catch (int k)

{ printf("Число більше чи дорівнює %d\n",k);

}

printf("Кінець\n");

getch();

}

int Check(int i) throw (const char\*, int)

{ printf("Усередині функції Check\n");

if(i==0) throw("Число дорівнює нулю!");

if(i>=MAX) throw MAX;

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=Check(n);

m=Check(m);

printf("Кінець функції Divide\n");

return n/m;

}

Результат роботи цієї програми такий:

Початок

Усередині блоку try

Усередині функції Divide

Усередині функції Check

Число дорівнює нулю

Кінець

Список дозволених виняткових ситуацій поширюється лише на типи значень, що повертаються функцією в модуль виклику. Інакше кажучи, ці обмеження не стосуються блоків try, що знаходяться усередині функції. Якщо ми перенесемо обробку виняткових ситуацій усередину функції Check(), їй можна взагалі заборонити генерувати виняткові ситуації і передавати їх по ланцюжку модулів виклику.

#define MAX 100000

int Check(int) throw();

int Divide(int, int);

int main()

{ int n = 0, m=MAX, l;

printf("Початок\n");

l = Divide(n,m+1);

printf("Кінець\n");

getch();

}

int Check(int i) throw ()

{ printf("Усередині функції Check\n");

try

{ printf("Усередині блоку try\n");

if(i==0) throw("Число дорівнює нулю!");

if(i>=MAX) throw MAX;

}

catch (const char\* s)

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

}

catch (int k)

{ printf("Число більше чи дорівнює %d\n",k);

}

printf("Кінець функції Check\n"); return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=Check(n);

m=Check(m);

if (n==-1 || m==-1)

{ printf("Аварійне завершення!");

}

printf("Кінець функції Divide\n");

return n/m;

}

Результат роботи цієї програми виглядає в такий спосіб:

Початок

Усередині функції Divide

Усередині функції Check

Усередині блоку try

Число дорівнює нулю!

Усередині функції Check

Усередині блоку try

Число чи більше дорівнює 100000

Аварійне завершення

## Повторні виняткові ситуації

Іноді зручно обробляти виняткову ситуацію в декількох розділах catch, ніби передаючи її по конвеєру. Уявимо собі, що на першому етапі оброблювач просто констатує наявність проблеми і пропонує користувачу вирішити, що робити — намагатися виправити помилку чи припинити роботу програми. Якщо програміст вирішить продовжувати роботу, необхідно згенерувати ситуацію знову і вжити заходів, що дозволяють врятувати положення. Якщо ж програміст вирішить припинити роботу програми, викликається відповідна функція чи виконується оператор return.

#define MAX 100000

int Check(int) throw();

int Divide(int, int);

int main()

{ int n = 0, m=MAX, l;

printf("Початок\n");

l = Divide(n,m+1);

printf("Кінець\n");

getch();

}

int Check(int i) throw ()

{ int choice;

printf("Усередині функції Check\n");

try

{ printf("in try Усередині блоку try\n");

if(i==0) throw("Число дорівнює нулю!");

if(i>=MAX) throw MAX;

}

catch (const char\* s)

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

printf("Ваш вибір? 0 - Вихід; 1 - Продовжити");

scanf("%d",&choice);

if(!choice)

{ printf("Виходимо\n");

}

else throw;

}

catch (int k)

{ printf("Число більше чи дорівнює %d\n",k);

}

printf("Кінець функції Check\n"); return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

try

{ n=Check(n);

}

catch (const char\* s)

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

printf("Продовжуємо\n");

}

try

{ m=Check(m);

}

catch (const char\* s)

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

printf("Продовжуємо\n");

}

if (n==-1 || m==-1)

{ printf("Аварійне завершення!\n");

}

printf("Кінець функції Divide\n");

return n/m;

}

Проаналізуємо хід виконання цієї програми.

Початок

Усередині функції Divide

Усередині функції Check

Усередині блоку try

Число дорівнює нулю!

Ваш вибір? 0 - Вихід; 1 - Продовжити 1

Число дорівнює нулю!

Продовжуємо

Кінець

Чисельник дробу дорівнює нулю, а знаменник — максимальному значенню. Усередині функції Divide() викликається функція Check(), що повинна перевірити коректність чисельника. Знайшовши, що чисельник дорівнює нулю, функція Check() генерує виняткову ситуацію типу const char\*. Її перехоплює блок catch, усередині якого користувачу пропонується або вийти з програми, ввівши число 0, або продовжити виконання, ввівши 1 чи будь-яке інше число. У даному випадку користувач ввів одиницю. У відповідь на це функція Check() повторно згенерувала виняткову ситуацію типу const char\*. Для цього використовується оператор throw без параметрів, що генерує повторно поточну ситуацію. Оскільки тому самому блоку try не можна поставити у відповідність декілька блоків catch з однаковим набором аргументів, виняткова ситуація передається по ланцюжку в модуль виклику. Про це свідчить рядок Продовжуємо.

Аналогічна обробка виняткової ситуації передбачена і для знаменника. Якщо у відповідь на запит програми користувач введе нуль, на екрані з'являться наступні рядки:

Виходимо

Усередині блоку Check

Усередині блоку try

Число більше чи дорівнює 100000

Аварійне завершення!

Кінець

## Непередбачені виняткові ситуації

Для того, щоб реагувати на виняткові ситуації, обробка яких програмістом не передбачена, у мові С++ використовуються функції terminate() і unexpected(), оголошені в заголовку <exception>, а також функція abort(), прототип якої міститься в заголовках <process.h> і <stdlib.h>.

Механізм роботи цих функцій виглядає в такий спосіб. Виняткова ситуація, яку не перехопив жоден оператор catch, передається нагору по ланцюжку викликів. Якщо вона не обробляється в жодному з модулів виклику, викликається terminate(), що за замовчуванням викликає функцію abort(). Такі виняткові ситуації називаються *непередбаченими* (*unexpected*).

За допомогою стандартних засобів простежити, коли і які функції викликаються, неможливо. Зазвичай вважається, що спочатку викликається функція unexpected(), що автоматично викликає функцію terminate(), а та у свою чергу — функцію abort() за замовчуванням. На щастя, у мові С++ існує механізм заміни функцій terminate() і unexpected() власними оброблювачами непередбачених ситуацій. Для цього призначені функції set\_terminate() і set\_unexpected(), специфікація яких міститься в заголовку <exception>:

new\_unexpected set\_unexpected(new\_unexpected) throw();

new\_terminate set\_terminate(new\_terminate) throw();

Ідентифікатори new\_unexpected і new\_terminate — імена нових оброблювачів. Аргументами функцій set\_unexpected() і set\_terminate() є вказівники на ці оброблювачі. Оскільки цими вказівниками автоматично є імена нових функцій, синтаксична конструкція виглядає досить просто. Одержуючи адресу нового оброблювача, функції set\_unexpected() і set\_terminate() повертають адреси старих оброблювачів. Це дозволяє зберегти їх у пам'яті і відновити при необхідності стару систему обробки непередбачених ситуацій. Замінники функцій unexpected() і terminate() не повинні повертати керування програмі. Вони зобов'язані поводитися, як їхні прототипи.

Продемонструємо систему обробки непередбачених ситуацій наступними прикладами. Спочатку розглянемо програму, у якій не передбачена обробка виняткової ситуації, що виникає, коли число перевищує максимально допустиме.

#define MAX 100000

int Check(int) throw();

int Divide(int, int);

int main()

{ int n = 0, m=MAX, l;

printf("Початок\n");

l = Divide(n,m+1);

printf("Кінець\n");

getch();

}

int Check(int i) throw ()

{ printf("Кінець функції Check\n");

try

{ printf("Усередині блоку try\n");

if(i==0) throw("Ділення на нуль!");

if(i>=MAX) throw MAX;

}

catch(const char\* s)

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

}

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=Check(n);

m=Check(m);

if (n==-1 || m==-1)

{ printf("Аварійне завершення!\n");

}

printf("Кінець функції Divide\n");

return n/m;

}

У цьому випадку на екрані з'являться такі рядки:

Початок

Усередині функції Divide

Усередині функції Check

Усередині блоку try

Число дорівнює нулю!

Кінець функції Check

Усередині функції Check

Усередині блоку try

На цьому виконання програми аварійно завершується. Справа в тому, що в ній не передбачена реакція на виняткову ситуацію типу int. Пошук оброблювачів у модулях виклику нічого не дав, і ОС викликала функції unexpected() і terminate(). Щоб відповісти на питання “в якому порядку це зроблено”, замінимо стандартні функції unexpected() і terminate() своїми версіями. Зверніть увагу на те, що знайти неопрацьовану виняткову ситуацію можна також за допомогою функції uncaught\_exception(), що повертає значення типу bool.

#define MAX 100000

int Check(int) throw (const char\*, int);

int Divide(int, int);

void new\_unexpected();

void new\_terminate();

int main()

{ int n = 0, m=MAX, l;

typedef void (\*pNew\_unexpected)();

typedef void (\*pNew\_terminate)();

pNew\_unexpected oldAddress1;

pNew\_terminate oldAddress2;

set\_unexpected(new\_unexpected); // встановлюємо новий обробник unexpected set\_terminate(new\_terminate); // встановлюємо новий обробник terminate printf("Початок\n");

l = Divide(n,m+1);

printf("Кінець\n");

oldAddress1 = set\_unexpected(oldAddress1);

oldAddress2 = set\_terminate(oldAddress2);

getch();

}

int Check(int i) throw (const char\*, int)

{ printf("Усередині функції Check\n");

try

{ printf("Усередині блоку try\n");

if(i==0) throw("Число дорівнює нулю!");

if(i>=MAX) throw MAX;

}

catch(const char\* s)

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

}

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=Check(n);

m=Check(m);

if (n==-1 || m==-1)

{ printf("Аварійне завершення!\n");

}

printf("Кінець функції Divide\n");

return n/m;

}

void new\_unexpected()

{ printf("Новий оброблювач unexpected\n");

}

void new\_terminate()

{ if (!uncaught\_exception()) printf("Виявлена непередбачена виняткова ситуація!\n");

else printf("Ok!\n");

printf("Новий оброблювач terminate\n");

}

Хід виконання програми ілюструється наступними рядками:

Початок

Усередині функції Divide

Усередині функції Check

Усередині блоку try

Число дорівнює нулю!

Кінець функції Check

Усередині функції Check

Усередині блоку try

Виявлена непередбачена виняткова ситуація!

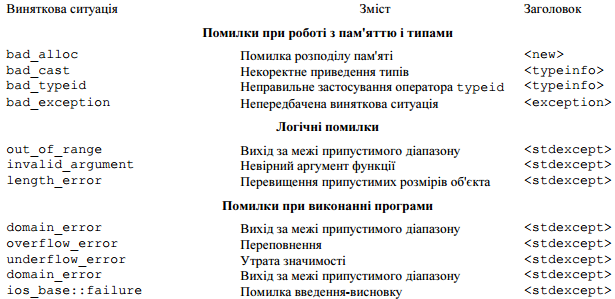
Новий оброблювач terminate

Як бачимо, при виявленні непередбаченої виняткової ситуації викликається оброблювач new\_terminate, що завершує роботу програми.

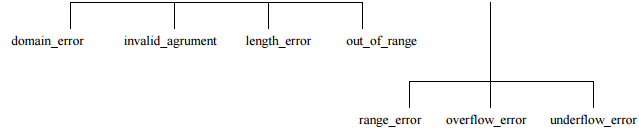
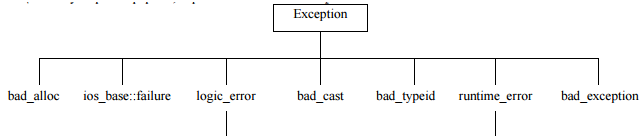
## Стандартні виняткові ситуації

Стандартна бібліотека мови C++ містить власну ієрархію класів винятків, що є прямими або непрямими нащадками базового класу exception. Нащадки класу exception умовно представляють дві категорії помилок: логічні помилки і помилки часу виконання

У мові С++ передбачено вісім стандартних виняткових ситуацій, що генеруються операторами і конструкторами стандартних класів.



Ці класи утворять ієрархію, коренем якої є клас exception.



Частина з них зв'язана з класами, оголошеними в стандартній бібліотеці шаблонів. Зупинимося поки на тих стандартних виняткових ситуаціях, що не належать до бібліотеки STL: bad\_alloc, bad\_cast, bad\_typeid і bad\_exception. Виняткові ситуації, що належать цим класам, поєднує одна загальна властивість — усі вони є об'єктами, а не вказівниками. Отже, вони передаються або за значенням, або по посиланню.

## Клас bad\_alloc

Мабуть, найбільш важлива виняткова ситуація описується класом std::bad\_alloc.

int main()

{ long\* q = NULL;

try

{ q = new long[5000000000000];

}

catch(const bad\_alloc&)

{ cout << "Bad\_alloc";

}

catch(...)

{ cout << "Інші";

}

cout << q << endl;

delete[] q;

getch();

}

Ця програма не обов'язково викликає виняткову ситуацію. Однак якщо виникне недостача пам'яті, буде згенерирований об'єкт класу bad\_alloc, а на екран буде виведена назва виняткової ситуації і нульове значення вказівника q.

## Клас bad\_cast

Виняткова ситуація bad\_cast виникає, коли спроба приведення типу є некоректною. Наприклад, не можна приводити поліморфний тип до посилання на інший поліморфний тип.

class A

{ int n;

public:

A(int k):n(k) { }

virtual void view() { cout << n << endl; }

};

class B: public A

{ int m;

public:

B(int l): A(l) { m=l; }

virtual void view() { cout << m << endl; }

};

int main()

{ A \*p = new A(0);

try

{ B& r = dynamic\_cast<B&>(\*p); // генерує виняткову ситуацію

}

catch (const bad\_cast& ex)

{ cout << ex.what() << endl;

}

getch();

}

При виконанні другого оператора в блоці try буде згенерована виняткова ситуація, повідомлення про яку виводить на екран функція what(), що належить класу bad\_alloc. Зверніть увагу на те, що об'єкт класу bad\_cast передається за допомогою константного посилання.

## Клас bad\_typeid

Виняткова ситуація, що належить класу bad\_typeid, генерується, якщо оператор dynamic\_cast застосовується до нульового вказівника. Повернемося до попереднього прикладу, злегка змінивши його.

class A

{ int n;

public:

A(int k):n(k) { }

virtual void view() { cout << n << endl; }

};

class B: public A

{ int m;

public:

B(int l):A(l) { m = l; }

virtual void view() { cout << m << endl; }

};

int main()

{ A\* p = new A(0);

try

{ B\* p = dynamic\_cast(p); // Некоректне приведення

cout << typeid(\*p).name() << endl;

}

catch (const bad\_typeid& ex)

{ cout << ex.what() << endl;

}

getch();

}

У цій програмі виконується спроба привести вказівник на базовий клас до вказівника на похідний клас. Однак у даному випадку це приведення неможливе, оскільки вказівник p посилається на об'єкт базового класу A. Для того, щоб це приведення стало коректним, варто було б установити цей вказівник на об'єкт класу B. Таким чином, реагуючи на помилку, оператор dynamic\_cast повертає нульовий вказівник. Спроба розіменувати його, виконуючи оператор typeid, породжує виняткову ситуацію bad\_typeid.

## Клас bad\_exception

Виняткова ситуація bad\_exception генерується функцією unexpected() у тих випадках, для яких не була передбачена обробка. Повернемося до прикладу, що ілюструє застосування списку дозволених виняткових ситуацій. Припустимо, у нас немає бажання перелічувати усі виняткові ситуації, що можуть виникнути при виконанні програми, оскільки програма повинна однаково реагувати на них (наприклад, виводити повідомлення на екран).

#define MAX 1000000

int Check(int) throw (const char\*, bad\_exception);

int Divide(int, int);

int main()

{ int n = MAX, m=0, l;

printf("Початок\n");

try

{ printf("Усередині блоку try\n");

l = Divide(n+1,m);

}

catch (const char\* s)

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

}

catch (int k)

{ printf("Число більше чи дорівнює %d\n",k);

}

printf("Кінець\n");

getch();

}

int Check(int i) throw (const char\*, bad\_exception)

{ printf("Усередині функції Check\n");

If (i==0) throw("Ділення на нуль усередині функції main!");

if (abs(i)>=MAX) throw "Bad\_exception...";

printf("Кінець функції Check\n");

return i;

}

int Divide(int n, int m)

{ printf("Усередині функції Divide\n");

n=Check(n);

m=Check(m);

printf("Кінець функції Divide\n");

return n/m;

}

Нагадаємо, що ця програма виконує ділення двох цілих чисел. Тепер для порівняння знаменника з нулем і максимально допустимим значенням використовується одна функція — Check(). Вона реагує на ситуацію, коли знаменник дорівнює нулю. В усіх інших випадках функція unexpected() виявляє непередбачену виняткову ситуацію і генерує об'єкт класу bad\_exception.

Початок

Усередині блоку try

Усередині функції Divide

Усередині функції Check

Bad\_exception...

Кінець

У завершення відзначимо, що обробка виняткових ситуацій приводить до додаткових витрат ресурсів комп'ютера. Програми, що передбачають обробку виняткових ситуацій, містять додатковий код і працюють повільніше. Таким чином, до цього механізму мови С++ варто прибігати тільки в дійсно виняткових випадках, коли без нього не можна обійтися. Наприклад, у попередній програмі ігнорування виняткових ситуацій зменшує час її виконання в два рази.

## Резюме

1. Винятковою ситуацією називається будь-яка подія, що вимагає особливої обробки
2. Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається обробкою виняткової ситуації. Ця задача покладається на оброблювача виняткової ситуації.
3. 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 неї перехоплює. Об'єктом, що описує природу виняткової ситуації, може бути будь-як 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.

В основі обробки виняткових ситуацій три ключових слова: try, catch і throw. Оператор try визначає область програми, де може виникнути виняткова ситуація, оператор throw генерує об’єкт — виняткову ситуацію, а оператор catch перехоплює цей об’єкт.

1. Розмір блоку try не обмежений. У нього можна занурити як один оператор, так і цілу програму.
2. Кожен блок catch відповідає окремому типу виняткової ситуації. Програма сама визначає, 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.
3. Якщо в програмі виникла подія, що програміст вважає небажаним, оператор throw генерує виняткову ситуацію. Для цього оператор throw повинний знаходитися усередині блоку try або усередині функції, викликуваної усередині блоку try.
4. Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort().
5. Якщо блок try знаходиться усередині функції, обробка виняткової ситуації виконується при кожнім виклику.
6. 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 більш ранньому прикладі, або перенести обробку в модуль виклику. У першому випадку функція, завершивши обробку, повертає визначене її специфікацією значення, а в другому — виняткову ситуацію.
7. Оскільки виняткова ситуація може бути об'єктом класу, у мові С++ існує можливість створювати ієрархію виняткових ситуацій. У цьому випадку блок catch перехоплює об'єкти не тільки базового, але і похідних класів. При генерації похідних виняткових ситуацій це приводить до непорозумінь — їх перехоплює блок catch, призначений для обробки базових виняткових ситуацій.
8. Для того щоб реагувати на виняткові ситуації, обробка яких програмістом не передбачена, у мові С++ використовуються функції terminate() і unexpected(), оголошені в заголовку, а також функція abort().
9. Ідентифікатори new\_unexpected і new\_terminate - імена нових оброблювачів. Аргументами функцій set\_unexpected() і set\_terminate() є вказівники на ці оброблювачі. Оскільки цими вказівниками автоматично є імена нових функцій, синтаксична конструкція виглядає досить просто. Одержуючи адресу нового оброблювача, функції set\_unexpected() і set\_terminate() повертають адреси старих оброблювачів. Це дозволяє зберегти їх у пам'яті і відновити при необхідності стару систему обробки непередбачених ситуацій. Замінники функцій unexpected() і terminate() не повинні повертати керування програми. Вони зобов'язані поводитися, як їхні прототипи.