Лекція 10: Виключення

Поняття виключень

Під час роботи програм зустрічаються ситуації які перешкоджають подальшої виконанню нормальної їх роботи. Наприклад при діленні числа на нуль, програма завершить роботу не дивлячись на те, скільки користувач працював в програмі, і який обсяг даних вніс. Програма просто закриється.

І уявіть, якщо до цього користувач вносив в програму дані кілька годин і проводив розрахунки. При такому аварійному закритті програми всі ці дані і розрахунки пропадуть. Також може зустрітися така ситуація, коли програма намагається відкрити недоступний в цей момент файл, або запросити більше, ніж є пам'яті. В таких ситуаціях бажано вміти переривати небезпечний процес при цьому продовжуючи дію програми.

Такого роду ситуації програмістам треба намагатися передбачити і будувати програми так, щоб вони могли гнучко реагувати, а не аварійно закриватися.

Відоме таке визначення винятків в Сі ++:

В мові Сі++ виключення - це реакція на нештатну ситуацію, що виникла під час виконання програми, наприклад при відленні пам'яті або при відкритті файлу Виключення дозволяють передати управління програмою з однієї частини в іншу.

Розглянемо механізм роботи винятків в Сі++ на простому прикладі. У ньому ми передбачаємо той випадок, що в якийсь момент під час розрахунків програми, може зустрітися розподіл числа на 0. Наберіть і скомпілюйте код, записаний нижче. Щоб переконатися в тому, як реагує програма на таку ситуацію, внесіть число 0 в змінну **num2** (вона виступає дільником в прикладі).

Приклад 1:

```
setlocale(LC_ALL, "ukr");
int num1;
int num2;
int var = 3; // керуюча змінна для while
while (var != 0) {
    cout << "Введить число num1: ";
    cin >> num1;
    cout << "Введить число num2: ";
    cin >> num2;
    cout << "num1 + num2 = " << num1 + num2 << endl;
    cout << "num1 / num2 = " << num1 / num2 << endl;
    cout << "num1 - num2 = " << num1 - num2 << endl;
    cout << "==================================" << endl << endl;
   var--;
}
```

Так як змінна **var** дорівнює 3, цикл **while**, в нормальній ситуації, повинен відпрацювати три рази. З кожним кроком циклу **var** зменшується на одиницю за допомогою декременту. Але так як ми відразу внесли значення 0 в змінну **num2**, програма не пройде до кінця навіть перший крок циклу. вона перерветься.

У наступному лістингу, ми виправимо це упущення – додамо кілька компонентів, які допоможуть зреагувати на цю ситуацію без переривання роботи програми. А саме:

- блок **try** или **try**-блок(спроба, зразок);
- генератор виключення– блок **throw**(обробити, запустити);
- обробник виключення, який його перехоплює-команда **catch** (зловити, ловити)

Як працює виняток? – Програміст прописує в коді (в **try**-блоці) конкретна умова, що якщо змінна **num2** буде дорівнювати 0, то в такому випадку необхідно генерувати виняток в **throw**. Далі, то що згенерувано в **throw**, перехоплює **try**-блок (у вигляді параметра функції) і програма виконає той код, який прописаний в цьому блоці.

Приклад 2:

```
setlocale(LC_ALL, "ukr");
 int num1;
 int num2;
 int var = 3;
 while (var != 0) {
    cout << "Введить число num1: ";
    cin >> num1;
    cout << "Введить число num2: ";
    cin >> num2;
    cout << "num1 + num2 = " << num1 + num2 << endl;
    cout << "num1 / num2 = ";
    try { // тут код, який може викликати помилку
       if (num2 == 0) {
          throw 42; // генерувати ціле число 42
       }
       cout << num1 / num2 << endl;
    }
    catch (int thr) { // сюда передаєтбся число, яке згенерував throw
       cout << "Помилка №" << thr << " - ділення на 0!!!" << endl;
```

```
      }
       catch (exception& e)       {
      cerr << "exception caught: " << e.what() << '\n';
      }
      cout << "num1 - num2 = " << num1 - num2 << endl;
     cout << "================================" << endl << endl;
      var--;
   }
   cout << "Програма завершила роботу!" << endl << endl;
```

Синтаксіс блоку виключення

Винятки забезпечують спосіб реагування на виняткові обставини (наприклад, помилки під час виконання) в програмах шляхом передачі керування до спеціальних функцій, які називаються обробниками.
Щоб зловити винятки, частина коду поміщається під інспекцію виключення. Це робиться шляхом укладання тієї частини коду в try-block. Коли виникає виняткова обставина в межах цього блоку, вилучається виняток, який передає керування обробнику винятку. Якщо не виключено жодного виключення, код продовжується нормально, і всі обробники ігноруються. Виняток виникає за допомогою ключового слова throw з блоку try. Обробники винятку оголошуються ключовим словом catch, який має бути розміщений відразу після блоку try:

```
// exceptions
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

An exception occurred. Exception Nr. 20

Код під обробкою виключень укладений у блок спроби. У цьому прикладі цей код просто викидає виняток:

```
throw 20;
```

Вираз бросання приймає один параметр (в даному випадку це ціле значення 20), який передається як аргумент обробнику винятку.

Обробник винятку оголошується з ключовим словом catch відразу після закриття фігурної дужки блоку try. Синтаксис catch схожий на звичайну функцію з одним параметром. Тип цього параметра дуже важливий, оскільки тип аргументу, переданий виразом throw, перевіряється проти нього, і тільки в тому випадку, якщо вони збігаються, виняток виявляється цим обробником.

Кілька обробників (тобто, вирази уловлювання) можуть бути прив'язані; кожен з іншим типом параметрів. Виконується тільки обробник, тип аргументу якого відповідає типу винятку, зазначеного у операторі throw.

Якщо в якості параметра catch використовується еліпсис (...), то обробник вловлюватиме будь-яке виключення, незалежно від типу викинутого виключення. Це можна використовувати як обробник за умовчанням, який ловить усі винятки, які не виявлено іншими обробниками:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

У цьому випадку, останній обробник вловлюватиме будь-яке виключення типу, що не є ні int, ні char.

Після того, як виняток було оброблено програмою, виконання продовжується після блоку try-catch, а не після оператора throw !.

Також можна вкласти блоки try-catch в більш зовнішні блоки спроб. У цих випадках ми маємо можливість, що внутрішній блок лову переадресовує виняток на свій зовнішній рівень. Це робиться за допомогою виразів броску; без аргументів. Наприклад:

```
try {
  try {
    // code here
  }
  catch (int n) {
    throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Примітка: Старий код може містити динамічні специфікації виключень. Вони тепер застаріли в Сі++, але все ще підтримуються. Характеристика динамічного

виключення слідує за декларуванням функції, додаючи до неї специфікацію. Наприклад:

double myfunction (*char* param) *throw* (*int*);

Це оголошує функцію myfunction, яка приймає один аргумент типу char і повертає значення типу double. Якщо ця функція кидає виняток іншого типу, відмінного від int, функція викликає std :: unexpected замість пошуку обробника або виклику std :: terminate.

Якщо цей специфікатор *throw* залишиться порожнім без типу, це означає, що std :: unexpected викликається для будь-якого виключення. Функції без специфікатора виключення (регулярні функції) ніколи не називають std :: unexpected, але дотримуються звичайного шляху пошуку свого обробника виключень.

Тобто можливі наступні варіанти:

- *int* myfunction (*int* param) *throw*(); *//Не очікуютьмя жодні виключення*
- *int* myfunction (*int* param); *// звичайна обробка виключень*
- <new> std :: nothrow // не очікується виключення
- extern const nothrow_t nothrow;

Це значення константи використовується як аргумент для оператора new та operator new [] для вказівки, що ці функції не повинні викидати виняток при відмові, а замість цього повертають нульовий покажчик.

За замовчуванням, коли новий оператор використовується, щоб спробувати виділити пам'ять, а функція обробки не може зробити це, викидається помилка bad_alloc. Але коли nothrow використовується як аргумент для new, замість нього повертається нульовий покажчик.

Ця константа (nothrow) є лише значенням типу nothrow_t, з єдиною метою запуску перевантаженої версії оператора функції new (або оператора new []), що приймає аргумент цього типу.

У Сі++ оператор нової функції може бути перевантажений, щоб взяти більше одного параметра: Перший параметр, переданий оператору новою функцією, завжди є розміром пам'яті, яку слід виділити, але додаткові аргументи можуть бути передані цій функції, уклавши їх в дужки в новому виразі. Наприклад:

int * p = new (x) int;

є дійсним виразом, який у певний момент викликає оператор new(sizeof (int), x). За замовчуванням одна з версій оператора new перевантажена, щоб прийняти параметр типу nothrow_t (як nothrow). Саме значення не використовується, але ця версія оператора new повинна повернути нульовий покажчик у разі помилки замість того, щоб кидати виняток.

Те ж саме стосується new оператора [] і оператора функції new [].

// nothrow example

```
#include <iostream>    // std::cout
#include <new>         // std::nothrow
int main () {
  std::cout << "Attempting to allocate 1 MiB... ";
  char* p = new (std::nothrow) char [1048576];

  if (!p) {          // null pointers are implicitly converted to false
    std::cout << "Failed!\n";
  }
  else {
    std::cout << "Succeeded!\n";
    delete[] p;
  }
  return
}
```

Проблема відловлення виключення в Сі -коді.

код 1

```
#include <iostream>
#include <cstring>
#include <string>

int main(){
    int A = 25, B = 5, C = 0;
    std::cout << "A " << std::endl;
    std::cin >> A;
    std::cout << "B " << std::endl;
    std::cin >> B;

    try
    {
        if (B == 0)
            throw "Error";
        C = A / B;
    }
    catch (const char* e)
    {
        std::cout << e << std::endl;
    }
    std::cout << "Result is " << C << std::endl;

    return 0;
}
```

І запустіть цей код. Програма буде крашитись якщо B буде 0.

Наступний код може вирішити цю проблему:

▲код 2

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <string>
```

```
int main(){
    int A = 25, B = 5, C = 0;
    std::cout << "A " << std::endl;
    std::cin >> A;
    std::cout << "B " << std::endl;
    std::cin >> B;

    try    {
        //if (B == 0)
        //    throw "Error";
        C = A / B;
    }
    catch (const char* e)    {
        std::cout << e << std::endl;
    }
    std::cout << "Result is " << C << std::endl;

    return 0;
}

{
    __try
    {
        *pResult = dividend / divisor;
    }
    __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        std::cerr << "Error! Integer division by zero\n";
        return FALSE;
    }
    return TRUE;
}


int main()
{
    int A = 25, B = 5, C = 0;
    std::cout << "A " << std::endl;
    std::cin >> A;
    std::cout << "B " << std::endl;
    std::cin >> B;

    if(SafeDiv(A, B, &C))
        std::cout << "A/B="<<C<<std::endl;
    return 0;
}
```

А також зверніть увагу на ще один спосіб https://msdn.microsoft.com/ru-ru/library/5z4bw5h5.aspx

```
// crt_settrans.cpp
// compile with: /EHa
#include <stdio.h>
#include <windows.h>
#include <eh.h>

void SEFunc();
void trans_func( unsigned int, EXCEPTION_POINTERS* );
```

```
class SE_Exception
{
private:
    unsigned int nSE;
public:
    SE_Exception() {}
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};
int main( void )  {
    try      {
        _set_se_translator( trans_func );
        SEFunc();
    }
    catch( SE_Exception e )      {
        printf( "Caught a __try exception with SE_Exception.\n" );
    }
}
void SEFunc()  {
    __try      {
        int x, y=0;
        x = 5 / y;
    }
    __finally      {
        printf( "In finally\n" );
    }
}
void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )  {
    printf( "In trans_func.\n" );
    throw SE_Exception();
}
#include<iostream>
#include<csignal> /* for signal */
#include<cstdlib>

using namespace std;

void fpe_handler(int signal){
    cerr << "Floating Point Exception: division by zero" << endl;
    exit(signal);
}

int main(){
    // Register floating-point exception handler.
    signal(SIGFPE, fpe_handler);

    int a = 1;
    int b = 0;
    cout << a/b << endl;

    return 0;
}
```

Клас std::exception

Всі об'єкти, що є виключеннями від компонент стандартної бібліотеки, виводяться з цього класу. Таким чином, всі стандартні винятки можуть бути виявлені шляхом лову цього типу за посиланням.

Приклад.

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}
```

Member functions

(constructor)
Конструктор виключення (публічний метод)

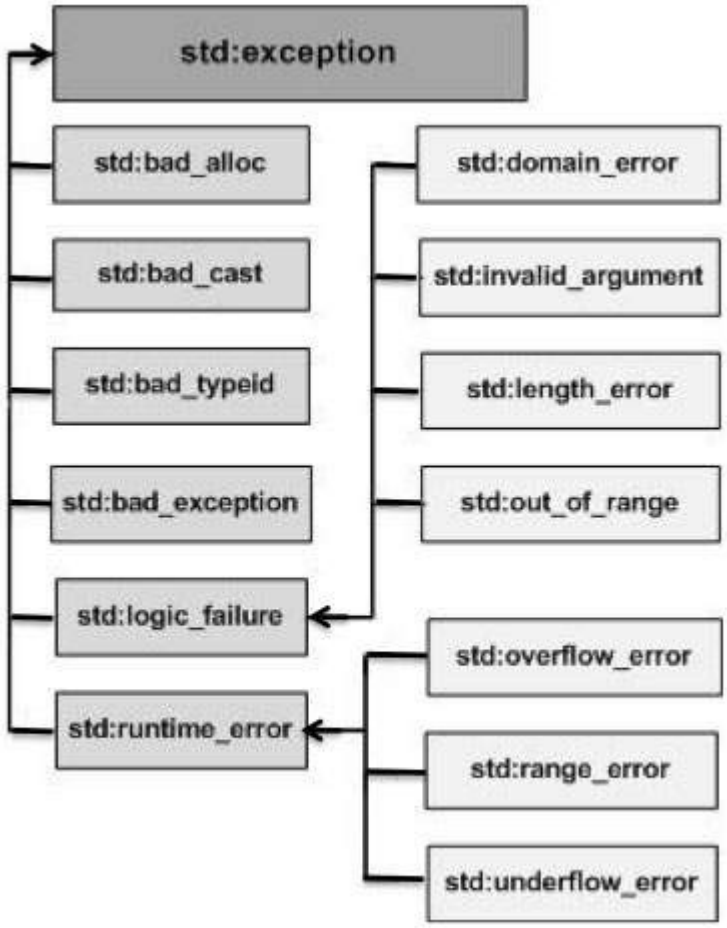
operator=
Оператор присвоєння - копіює виключення (публічний метод)

what (virtual)
Повертає рядок, що іденифікує виключення (публічний метод)

(destructor) (virtual)
Деструктор (знищувач) виключення (публічний віртуальний метод)

Стандартні виключення Ci++ (Standard Exceptions)

Ci++ містить деякі стандартні виключення, що містяться в заголовочному файлі **<exception>** який можна підляючати в програми. Ці класи-виключення підпорядковуються ієрархії класів, яка зображена нижче:



В талиці містяться описи даних виключень.

Sr.No	Виключення (Exception) та його опис
1	**std::exception** Виключення — батьківський клас для віх стандартних Сі++ виключень.
2	**std::bad_alloc** Кидається при не виконанні **new**.
3	**std::bad_cast** викидається при поганому **dynamic_cast**.
4	**std::bad_exception** Корисне виключення, що викидається при виключенні, що не оброблюється C++ програмою.
5	**std::bad_typeid** Виключення, що викидається **typeid**.
6	**std::logic_error** виключення про логічну помилку, що теоретично може бути визначено при читанні коду
7	**std::domain_error** виключення, що позначає, що математичний оператор застосовується в некоректній області визначення
8	**std::invalid_argument** Викидається при направильних аргументах функції
9	**std::length_error** Занадто вилике std::string створюється.

10	**std::out_of_range** Виключення що викидається методом 'at', наприклоад в std::vector та std::bitset<>::operator[]().
11	**std::runtime_error** виключення про логічну помилку, що теоретично не може бути визначено при читанні коду.
12	**std::overflow_error** Переповнення типу при математичній операції
13	**std::range_error** Виключення при роботі зі змінної з неправильного діапазону значень
14	**std::underflow_error** Недостаньої повний математичний тип.

Створення власного виключення

Можливо створити власне виключення шляхом наслідування та перевантаження функціональності класу exception. У вказаному прикладі, використовується клас std::exception для створення власного виключення:

Приклад.

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception {
  const char * what () const throw () {
    return "C++ Exception";
  }
};

int main() {
  try {
    throw MyException();
  } catch(MyException& e) {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  } catch(std::exception& e) {
    //Other errors
  }
}
```

Результат:

```
MyException caught
C++ Exception
```

Тут what() - публічний метод, що створений в класі виключення і ввін превантажується класами нащадками. Він повертає причину виключення як рядок:

```
// exception example
#include <iostream>      // std::cerr
#include <typeinfo>      // operator typeid
#include <exception>     // std::exception

class Polymorphic {virtual void member(){}};
int main () {
  try  {
    Polymorphic * pb = 0;
    typeid(*pb);  // throws a bad_typeid exception
  }
  catch (std::exception& e)  {
    std::cerr << "exception caught: " << e.what() << '\n';
  }
  return 0;
}
```

Приклад (Створення власного виключення):

```
class BasicQueueException : public std::logic_error {
public:
   explicit BasicQueueException(const char* message):std::logic_error(message) {}
};
class EmptyQueueException : public BasicQueueException {
public:
   explicit EmptyQueueException() : BasicQueueException("Queue is empty") {}
};
void Queue:::pop() {
   if(isEmpty()) throw EmptyQueueException();
   // ....
}

// ...
try{
   q.pop();
}
catch(BasicQueueException& e) {
   std::cerr << e.what() << std::endl;
}
```