Лекція 15. Опрацювання помилок, винятки

1. *Традиційні способи опрацювання помилок.*
2. *Використання винятків (значення вбудованих типів).*
3. *Оголошення і використання класів винятків.*
4. *Стандартні класи винятків.*
5. *Розгортання стека, повторне генерування винятку.*
6. *Можливі проблеми, породжені опрацюванням винятку.*
7. *Додаткові можливості використання винятків.*

Сьогоднішню лекцію ми розпочнемо з однієї шкільної задачі.

Задача. Відомо, що автомобіль подолав першу половину шляху зі швидкістю 45 *км/год*., а другу – зі швидкістю 55 *км/год*. Знайдіть середню швидкість автомобіля.

Ми напишемо програму, що обчислюватиме середню швидкість V_c для будь-яких заданих швидкостей V_1 та V_2. Пригадаймо собі, що розв'язком поставленої задачі є $V_c = \frac{2V_1V_2}{V_1+V_2}$. Таке число називають середнім гармонійним. Його обчислення опишемо за допомогою функції:

```
double harmonic(double a, double b)
{
     return 2.0*a*b/(a + b);
}
```

Тепер можемо легко обчислювати розв'язок задачі за допомогою комп'ютера:

```
int main() {
     SetConsoleOutputCP(1251);
     double x, y, z;
     cout << "\nВведіть два числа: "; cin >> x >> y;
     z = harmonic(x, y);
     cout << "\nСереднє гармонійне чисел "<< x <<" та "<< y <<" є "<< z <<'\n';
     return 0;
}
```

Програма працюватиме правильно для довільних вхідних даних задачі, але чи буде це так для взагалі довільних дійсних чисел? Очевидно, що ні, бо для випадку $x = -y$ програма спробує виконати ділення на нуль, що призведе до аварійного завершення.

Виникнення проблемних ситуацій під час виконання програми не рідкість, а, швидше, правило. Помилки трапляються найрізноманітніші: математичні, що виникають у ході обчислень (переповнення розрядної сітки, втрата точності, ділення на нуль, обчислення логарифма від'ємного числа тощо), файлові (не знайдено файл чи шлях до файла, відмовлено в доступі, переповнено диск), керування пам'яттю (вичерпано динамічну пам'ять, звертання за межі пам'яті процесу) та інші. Як же діяти, якщо навколо стільки загроз?

Можна комплектувати свої програми докладними інструкціями, де описано, за яких умов, для яких вхідних даних програма працює правильно (наприклад, написана нами програма працюватиме для будь-яких двох додатніх чисел), і сподіватися на краще: користувач дотримається інструкції, операційна система дасть раду, якщо все-таки щось виникне. Але це не дуже добра тактика, навіть якщо пишете програми виключно для власних потреб. Надійна програма мала б якось захищатися від потенційних помилок. Отож розглянемо різні можливості удосконалення функції обчислення середнього гармонійного двох дійсних чисел.

Спосіб 1 (лінивий). Функція виявляє неправильні вхідні дані, інформує про помилку і завершує виконання програми (бо не знає, як правильно її продовжити).

```
double harmonic_1(double a, double b)
{    if (a == -b)
     {            cout << "Недопустимі аргументи для harmonic_1\n";
            abort(); }
     return 2.0*a*b/(a+b);
}
```

Цей варіант має єдину перевагу: змістовне повідомлення про причини передчасного завершення. Але програма все ж «вмерла»!

Спосіб 2 (традиційний). Функція повертає ознаку того, чи вдалось виконати обчислення. У цьому випадку для повернення результату обчислень використовують додатковий параметр.

```
bool harmonic_2(double a,double b,double * c)
{
     if (a == -b)
     {
            *c = -DBL_MAX; return false;
     }
     else
     {
            *c = 2.0*a*b/(a+b); return true;
     }
}
```

Досить добрий варіант, бо така функція завжди поводить себе осмислено, за її сигнатурою легко зрозуміти правила використання: вказівники традиційно використовують для повернення результатів, тип функції прозоро натякає на ознаку правильності завершення. Недоліки: треба, щоб користувач не лінувався перевіряти цю ознаку; такі перевірки заплутують код.

Спосіб 3 (сучасний) використовує *механізм винятків*. Функція, що виявила помилку, запускає виняткову ситуацію розраховуючи на те, що вирішити проблему можна зовні – в тій функції, що явно чи опосередковано її (функцію) викликала. Автор функції може виявити неправильні вхідні дані, але не знає, як виправити ситуацію. Користувач функції міг би правильно зреагувати на помилкові дані, але не знає, коли вони трапляються. Отже, потрібен інструмент, за допомогою якого функція могла б сигналізувати про виникнення помилки і передати керування кодові, що виправляє цю помилку. Таким інструментом і є виняток.

Виняток запускає інструкція *throw* ⟨*об'єкт*⟩. Тут об'єкт є "матеріальним носієм" винятку. Вирішальне значення для розпізнавання винятку має його тип. Дані, поміщені в об'єкт, надають додаткову інформацію про виняткову ситуацію. Функція, що здатна запустити виняток, відрізняється від усіх інших, тому в заголовку такої функції вказують ключове слово *throw* і список типів можливих винятків.

```
double harmonic_3(double a, double b) throw(const char*)
{    if (a == -b)
            throw "Недопустимі аргументи для harmonic_3: 'a = -b'";
     return 2.0*a*b/(a+b);
}
```

Правду кажучи, такий спосіб оголошення функції ви знайдете хіба в літературі, опублікованій декілька років назад. Сьогодні його вважають застарілим, а компілятор чудово дає собі раду зі звичайним оголошенням:

```
double harmonic_3(double a, double b)
{
     if (a == -b)
            throw "Недопустимі аргументи для harmonic_3: 'a = -b'";
     return 2.0*a*b/(a+b);
}
```

У наведеному прикладі інструкцію *throw* використано в тілі умовної інструкції. Вона запускає виняток-рядок. Таку функцію використовують трохи інакше. Для того, щоб перехопити й опрацювати виняток, виклик функції розташовують всередині *try*-блока, а після нього записують *catch*-блок з зазначенням типу винятка і кодом його опрацювання.

```
int main()
{
	cout << "Введіть два числа: ";
	int x, y, z;
	while (cin >> x >> y)
	{
		try
		{
			z = harmonic_3(x,y);
		}
		catch (const char * s)
		{
			cout << s <<"\n\n"
				<< "Введіть наступні два числа (або Q, щоб вийти): ";
			continue;
		}
		cout << "Середнє гармонійне чисел "<< x <<" та "<< y <<" є "<< z <<"\n\n"
			<< "Введіть наступні два числа (або Q, щоб вийти): ";
	}
	return 0;
}
```

Як це працює? Інструкція генерування винятку за своєю природою є своєрідним оператором переходу, що наказує передати керування іншій частині програми. Блок *try* є вказівкою компіляторові пильнувати за винятками. Блок *catch* перехоплює ті винятки, чий тип збігається з типом параметра блока, код в тілі блоку призначено для опрацювання перехопленого винятку. Якщо функція може запускати винятки різних типів, то після *try* записують декілька *catch*-блоків.

У наведеному прикладі інструкція *throw* припиняє виконання функції *harmonic_3*, звільняє стек від усіх локальних (автоматичних) змінних цієї функції і передає керування наступній в стеку функції в надії знайти блоки *try* і *catch* з відповідним типом – такою функцією є *main*. Далі керування передається на оператори *catch*-блока: буде надруковано вичерпне повідомлення про помилку, що сталася, та підказка про продовження роботи; інструкція *continue* передасть керування на наступну ітерацію циклу.

Оголошення параметра *catch*-блока схоже на оголошення параметра функції. Його аргументом стає запущений об’єкт-виняток. Але є й відмінності: компілятор створює копію запущеного об’єкта і саме її присвоює параметрові; зазначення імені параметра не є обов’язковим, якщо нема потреби передавати якісь дані, бо для розпізнавання вистачило б і самого типу.

Послідовність блоків *try { ⟨інструкція1⟩ } catch(T1) { опрацювання1 } catch(T2) { опрацювання2 } ⟨інструкція2⟩* працює за таким алгоритмом:

- виконати інструкцію 1;
- якщо винятків не сталося, то перейти до інструкції 2;
- якщо стався виняток, то шукати відповідний *catch*-блок:
  - якщо тип винятку збігся з T1, то виконати опрацювання 1 і перейти до інструкції 2;
  - у протилежному випадку перевірити наступний блок на співпадіння: якщо тип винятку збігся з T2, то виконати опрацювання 2 і перейти до інструкції 2;
  - якщо тип винятку не збігся з жодним типом *catch*-блока, завершити роботу функції і шукати опрацювання в зовнішній функції.

Виняток вважається опрацьованим, як тільки керування увійшло в *catch*-блок.

Для опрацювання різнотипних винятків використовують різні *catch*-блоки. Один *catch*-блок може опрацювати однотипні винятки з різних джерел:

```
double harmonic_3(double,double); // середнє гармонійне
double geometric(double,double);  // середнє геометричне

int main()
{
	SetConsoleOutputCP(1251);
	double x, y, z, g;
	cout << "\nВведіть два числа: ";
	while (cin >> x >> y)
	{
		try
		{
			z = harmonic_3(x,y);
			g = geometric(x,y);
		}
		catch (const char * s)
		{
			cout << s <<"\n\n"
				<< "Введіть правильні дані (або Q, щоб вийти): ";
			continue;
		}
		cout << "Введено числа: " << x << " та " << y
			<< "\nСереднє гармонійне: " << z << "\nСереднє геометричне: " << g
			<< "\n\nВведіть наступні два числа (або Q, щоб вийти): ";
	}
	return 0;
}

double geometric (double a, double b) // throw(const char*)
{
	if (a<0 || b<0)
		throw "Недопустимі аргументи для geometric: 'a<0 | b<0'";
	return sqrt(a*b);
}
```

У цьому прикладі джерелом винятку може стати одна з функцій *harmonic_3* чи *geometric*. Запущений виняток перехопить *catch*-блок, а користувач побачить відповідне повідомлення.

Ми вже згадували, що вирішальним для розпізнавання винятку є його тип. Природно, що для позначення винятків використовують класи – типи, визначені користувачем. Ми могли б використати ще один варіант розв'язування поставленої на початку лекції задачі: запускати як виняток екземпляр власноруч оголошеного класу. Підемо далі й оголосимо два класи для винятків, щоб сигналізувати про дві можливі помилки: ділення на нуль або одне з чисел рівне нулю.

```
// виняток можна позначати екземпляром порожнього класу
class DivByZero {};
class ZeroArg {};

double harmonic_4(double,double); // throw(DivByZero,ZeroArg);

int main()
{
	SetConsoleOutputCP(1251);
	double x, y, z;
	cout << "Введіть два числа: ";
	while (cin >> x >> y)
	{
		try
		{
			z = harmonic_4(x,y);
		}
```

```
            catch (DivByZero)
            {
                  cout << "Числа не можуть бути протилежними. Спробуйте ще.\n\n";
                  continue;
            }
            catch (ZeroArg)
            {
                  cout << "Жодне з чисел не може бути нулем. Спробуйте ще.\n\n";
                  continue;
            }
            cout << "Середнє гармонійне чисел "<< x <<" та "<< y <<" є "<< z <<"\n\n"
                  << "Введіть наступні два числа (або Q, щоб вийти): ";
      } return 0;
}

double harmonic_4(double a, double b)
{
      if (a == 0 || b == 0) throw ZeroArg();
      if (a == -b) throw DivByZero();
      return 2.0*a*b/(a + b);
}
```

Тут для позначення винятків оголошено класи *DivByZero* та *ZeroArg*. Вони не мають ніякої особливої функціональності, лише конструктори і деструктор, згенеровані компілятором. Та нам і цього вистачить, бо для потреб програми важливим є тільки ім'я класу. Інструкція *throw* запускає як виняток безіменний екземпляр класу, створений конструктором за замовчуванням. Блок *catch* впізнає тип винятка і перехоплює його.

Варто наголосити, що використання винятків не запобігає виникненню помилок і не виправляє їх якось автоматично. Механізм генерування та перехоплення винятків надає зручний і зрозумілий засіб для повідомлення про виникнення помилки та для програмного реагування на неї.

Наведемо ще один приклад оголошення і використання винятків. Клас *ArrayDbl*, що інкапсулює динамічний масив дійсних чисел і розширює можливості його використання, гарантує, що створений масив матиме правильний розмір, а звертання за індексом не виведе за межі масиву.

файл "arraydbl.h"

```
#ifndef array_dbe_h_
#define array_dbe_h_

#include <iostream>
using std::ostream;

class ArrayDbl
{
private:
      int size;
      double* arr;
      void checkSize(int n);
public:
      class BadSize {}; // позначатиме неправильний розмір масиву
      class BadIndex    // позначатиме неправильний індекс, а також міститиме
      {                 // його значення, буде вміти його повідомити
      public:
            int badInd;
            BadIndex(int i) : badInd(i) {}
            void report() const;
      };
      ArrayDbl() : arr(0), size(0) {}
      explicit ArrayDbl(int n, double val = 0.0) /*throw(BadSize)*/;
      ArrayDbl(const double* pn, int n) /*throw(BadSize)*/;
```

```
     ArrayDbl(const ArrayDbl& a);
     ArrayDbl& operator=(const ArrayDbl& a);
     ~ArrayDbl();
     int arSize() const { return size; }
     void resize(int n) /*throw(BadSize)*/;
     double average() const;
     double max() const;
     double& operator[](int i) /*throw(BadIndex&)*/;
     const double& operator[](int i) const /*throw(BadIndex&)*/;
     friend ostream& operator<<(ostream& os, const ArrayDbl& a);
};
#endif
```

файл "arraycpp.h"

```
#include "arraydbl.h"
// функціональність об’єкта-винятка; зверніть увагу на складену кваліфікацію імені
void ArrayDbl::BadIndex::report() const
{
     std::cerr << "Out of bounds index value: " << badInd << '\n';
}
// спеціалізований метод для перевірки правильності заданого розміру масиву:
// він не може бути порожнім, без жодного елемента, завеликі масиви також вважаються
// помилкою. Придумайте кращий спосіб задання верхньої межі дозволеного розміру.
void ArrayDbl::checkSize(int n)
{
     if (n <= 0 || n > 32000) throw BadSize();
}
// Конструктори з параметрами виділяють пам'ять тільки після перевірки
ArrayDbl::ArrayDbl(int n, double val) :size(0), arr(nullptr)
{
     checkSize(n);
     arr = new double[n];
     size = n;
     for (int i = 0; i < n; ++i) arr[i] = val;
}
ArrayDbl::ArrayDbl(const double *pn, int n) :size(0), arr(nullptr)
{
     checkSize(n);
     arr = new double[n];
     size = n;
     for (int i = 0; i < n; ++i) arr[i] = pn[i];
}
ArrayDbl::ArrayDbl(const ArrayDbl &a)
{
     size = a.size;
     arr = new double[size];
     for (int i = 0; i<size; ++i) arr[i] = a.arr[i];
}
ArrayDbl& ArrayDbl::operator =(const ArrayDbl &a)
{
     if (this == &a) return *this;
     delete[] arr;
     size = a.size;
     arr = new double[size];
     for (int i = 0; i < size; ++i) arr[i] = a.arr[i];
     return *this;
}
ArrayDbl::~ArrayDbl()
{
     delete[] arr;
}
// змінити розмір масиву можна тільки на правильний
void ArrayDbl::resize(int n)
{
     checkSize(n);
```

```
	if (n == size) return;
	double *newMem = new double[n];
	if (size < n)
	{
		for (int i = 0; i < size; ++i) newMem[i] = arr[i];
		for (int i = size; i < n; ++i) newMem[i] = 0.0;
	}
	else
		for (int i = 0; i < n; ++i) newMem[i] = arr[i];
	delete[] arr;
	arr = newMem;
	newMem = nullptr;
	size = n;
}
// "розумний" масив може дещо повідомити про себе
double ArrayDbl::average() const
{
	double sum = 0.0;
	for (int i = 0; i < size; ++i) sum += arr[i];
	return sum / size;
}

double ArrayDbl::max() const
{
	double res = arr[0];
	for (int i = 1; i < size; ++i)
		if (arr[i] > res) res = arr[i];
	return res;
}
// вихід індекса за межі масиву – поширена помилка
double& ArrayDbl::operator[](int i)
{
	if (i < 0 || i >= size) throw BadIndex(i);
	return arr[i];
}
const double& ArrayDbl::operator[](int i) const
{
	if (i < 0 || i >= size) throw BadIndex(i);
	return arr[i];
}

ostream& operator<<(ostream& os, const ArrayDbl& a)
{
	for (int i = 0; i<a.size; ++i)
	{
		os.width(8); os<<a.arr[i];
	}
	if (a.size % 10 != 0) os << '\n';
	return os;
}
```

Невелика тестова програма створює масив і провокує виникнення винятків. Спеціально задано неправильний розмір *Players* і виконано спробу доступитися до неіснуючих елементів. Усі ризиковані дії виконано в захищеному блоці *try*. Блоки *catch* перехоплюють повідомлення про помилки, що сталися, і дозволяють програмі працювати далі.

За змістом програма завантажує послідовність дійсних чисел і виводить їх на консоль у спеціальному форматі. Знайдіть, де в програмі працюють конструктори та оператори класу *ArrayDbl*, де випробувано методи.

```
#include <iostream>
#include "arraydbl.h"

using namespace std;
```

```
int main()
{
     int Players = -3;     // провокує виняток BadSize
     ArrayDbl Points(1);
rep:
     try {
           ArrayDbl Team(Players);
           cout << "Enter percentages for your " << Players
                 << " top players as a decimal fraction:\n";
           int player;
           for (player = 0; player < Players; ++player)
           {
                 cout << "Player " << (player + 1) << " % = ";
                 cin >> Team[player];
           }
           cout << Team << "max = " << Team.max()
                 << "\naverage = " << Team.average() << "\n\n";
           Points = Team;

           cout.precision(1);
           cout.setf(ios_base::showpoint);
           cout.setf(ios_base::fixed, ios_base::floatfield);
           cout << "Recapitulating, here are the percentages:\n";
           for (player = 0; player <= Players + 1; ++player) // провокує BadIndex
                 cout << "Player #"<<(player + 1)<<": "<<100.0*Team[player]<<"%\n";
     }
     catch (ArrayDbl::BadIndex & bi)
     {
           cout << "\nArrayDbE exception: "<<bi.badInd<<" is a bad index value\n"
                 << "Information from the exception object:\n";
           bi.report();
           cout << "--------------\n";
     }
     catch (ArrayDbl::BadSize)
     {
           cout << "\nArrayDbE exception: bad size of the array. "
                 << "\nEnter a valid size: ";
           cin >> Players; goto rep;
     }
     Points.resize(Players + 1);
     for (int player = 0; player <= Players; ++player)
           cout << "Player #"<<(player + 1)<<": "<<100.0*Points[player]<<"%\n";
     cout.precision(2);
     cout << "\nmax = "<<Points.max()<<"\naverage = "<<Points.average()<<'\n';

     cout << "Bye!\n";
     system("pause");
     return 0;
}
```

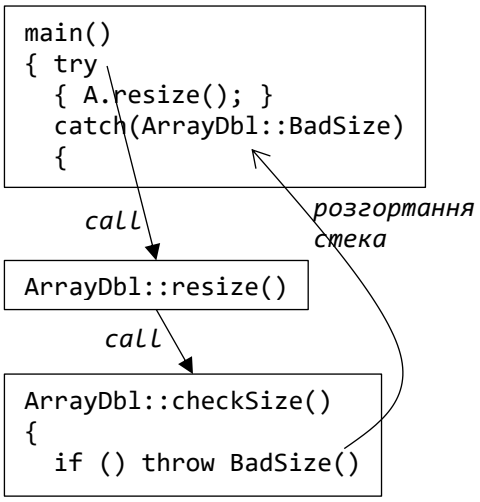
Зверніть увагу на задання типу винятка в *catch*-блоках – все правильно, класи винятків оголошено всередині класу *ArrayDbl*, тому потрібно використати кваліфікатор імені. Як і в попередньому прикладі, клас *BadSize* порожній, використовують тільки його ім'я. Зате клас *BadIndex* наділено і полями, і методами! Традиційно їх роблять відкритими.

Після захищеного блока розташовано два блоки перехоплення: кожен спеціалізується на винятку «свого» типу. Щоб отримати доступ до інформації, вкладеної в екземпляр винятку, в блоці *catch* оголошують локальну змінну, ніби параметр функції: *catch (ArrayDbl::BadIndex & bi)*. Змінна *bi* міститиме виняток після перехоплення.

Щоб перехоплювати одним блоком будь-які винятки, можна використати конструкцію *catch(...)* – «перехопити виняток довільного типу», але такий прийом хіба для налагодження програми, не для готового продукту.

Давайте розглянемо, як працюватиме виняток у такому коді

```
int main()
{
      ArrayDbl A(10);
      int N = 0;
      try
      {
            A.resize(N);
      }
      catch (ArrayDbl::BadSize)
      {
            cout << "Змінити розмір не вдалося\n";
      }
      cout << A.arSize(); ......
}
```



Запуск винятку розгортає стек викликів аж до того блока, де вперше зустрінеться *catch* з відповідним типом. Виконання методів *checkSize* та *resize* завершиться (нештатно), керування отримає функція *main*.

Окремо розглянемо генерування винятків у конструкторах. Багато авторів вказує, що виняток – єдиний спосіб, яким конструктор може повідомити про неправильно задані аргументи. Страуструп наводить приклад генерування винятку в конструкторі. Конструктори декількох контейнерів стандартної бібліотеки можуть генерувати винятки. Інші автори застерігають, що використання винятків у конструкторах можуть свідчити про неправильну архітектуру. Як має вести себе «недоконструйований» об'єкт? Поміркуйте, що станеться, якщо трапиться виняток у конструкторі динамічного об'єкта. Загальне правило таке: якщо в конструкторі потрібно використати виняток, потурбуйтеся про те, щоб ще перед генеруванням винятку всі поля об'єкта отримали змістовні значення. Гарантуйте, що об'єкт не захопить ніякого ресурсу, поки є ризик генерування винятку. Наші конструктори написані з дотриманням цих правил.

Зауважимо, що власні класи винятків ми оголосили тільки з навчальною метою. Замість них ми могли б використати і стандартні класи винятків. Приєднайте до проекту заголовковий файл *exception*, і у вашому розпорядженні буде низка спеціальних класів. (Деякі типи винятків оголошено в інших заголовкових файлах, описаних у документації.) Ось деякі з них: базовий тип

```
class exception // базовий для всіх винятків
{
      exception();
      exception(const char*);
      const char* what();
};
```

та ієрархія типів винятків за категоріями (відступ означає підкатегорію, в кутніх дужках вказано заголовковий файл типу, за ним – стисле пояснення щодо призначення; усі типи з простору імен *std*)

- exception <exception> interface (debatable if you should catch this)
  - bad_alloc <new> failure to allocate storage
    - bad_array_new_length <new> invalid array length
  - bad_cast <typeinfo> execution of an invalid dynamic-cast
  - bad_exception <exception> signifies an incorrect exception was thrown
  - bad_function_call <functional> thrown by "null" std::function
  - bad_typeid <typeinfo> using typeinfo on a null pointer
  - bad_weak_ptr <memory> constructing a shared_ptr from a bad weak_ptr
  - logic_error <stdexcept> errors detectable before the program executes

        - domain_error <stdexcept> parameter outside the valid range
        - future_error <future> violated a promise/future condition
        - invalid_argument <stdexcept> invalid argument
        - length_error <stdexcept> length exceeds its maximum allowable size
        - out_of_range <stdexcept> argument value not in its expected range
    - runtime_error <stdexcept> errors detectable when the program executes
        - overflow_error <stdexcept> arithmetic overflow error.
        - underflow_error <stdexcept> arithmetic underflow error.
        - range_error <stdexcept> range errors in internal computations
        - regex_error <regex> errors from the regular expression library.
        - system_error <system_error> from operating system or other C API
            - ios_base::failure <ios> Input or output error

Ми могли б використати *invalid_argument* замість *BadSize* та *out_of_range* замість *BadIndex*. Перевагу віддають власним класам винятків, якщо вони точніше діагностують причину помилки, є інформативнішими.

На ті класи, які використовують для позначення винятків, розповсюджуються ті самі правила, що й на всі інші класи: вони можуть утворювати ієрархію за відношенням наслідування, їх можна оголошувати всередині інших класів тощо. Досить часто класи винятків оголошують всередині тих класів, яким вони допомагають боротися з помилками. Це явно вказує на їхнє призначення та запобігає конфліктові імен. Саме так ми оголосили типи винятків для динамічного масиву.

Поділ на категорії полегшує опрацювання винятків: за допомогою перехоплення одного типу можна зловити винятки всіх його підкатегорій. Наприклад, *catch(logic_error)* перехопить і *logic_error*, і *invalid_argument*, і решту чотири типи винятків цієї категорії.

Якщо програма опрацьовує винятки, описані ієрархією класів, то блоки перехоплення розташовують у певному порядку: від конкретніших типів до загальніших. Тобто, спочатку перехоплюють винятки, описані підкласом, пізніше – базовим класом.

```
try
{
     DoSomethingChanceful();
}
catch(domain_error)
{
     // виправити значення, що не відповідає своєму діапазонові
}
catch(invalid_argument)
{
     // виправити невідповідний аргумент функції
}
catch(logic_error)
{
     // виправити всі інші логічні помилки
}
catch(exception)
{
     // виправити помилки всіх інших категорій
}
catch(...)
{
     // виправити всі некатегоризовані помилки
}
```

Згадаємо про ще одну можливість щодо опрацювання винятків – *повторне генерування* винятків. Якщо до виняткової ситуації привів ланцюжок викликів декількох функцій і є потреба опрацьовувати виняток у кожній з них, в блоках опрацювання використовують інструкцію *throw* без параметрів для повторного запуску щойно опрацьованого винятку. Наприклад:

```
class Ex {};
void first();  // throw(Ex)
void second(); // throw(Ex)

int main() {
     try { first(); }
     catch(Ex) { cout<<"Main work up\n"; }
     return 0;
}
void first() {
     try { second(); }
     catch(Ex)
     {
            cout<<"First work up\n";
            throw;
     }
}
void second() {
     bool cond = do_something();
     if (cond) throw Ex();
}
```

Згенерований в тілі *second()* виняток зазнає опрацювання в *catch*-блоці функції *first()*, але одразу ж буде запущений повторно. Наступне опрацювання відбудеться в *main()*.

Використання винятків допомагає виправити помилки, але може стати причиною нових. Якщо функція захопила якийсь ресурс (файл, пам'ять, пристрій тощо), то перед завершенням вона мала б його звільнити. Що станеться, якщо виняток трапиться раніше? Розглянемо приклад оголошення функції, що використовує для запису деякий файл. (Тут взаємодію з файлом організовано в стилі C.)

```
void use_file(const char* file_name) {
      FILE* f = fopen(file_name,"w"); //відкриваєм для запису файл з іменем file_name
      // працюємо з файлом f, тут може скластися ситуація, що запускає виняток
      // тоді керування до наступної інструкції не перейде, файл залишиться відкритим
      fclose(f);
}
```

Спробуємо виправити код і гарантувати закривання файла:

```
void use_file(const char* file_name) {
     FILE* f = fopen(file_name,"w");
     try {
     // працюємо з файлом f, тут може статися throw виняток
     }
     catch (...) { fclose(f); // що б не сталося, файл закриваємо
            throw; }          // і повідомляємо про виняток далі
     fclose(f); // нормальне закривання файла, спрацьовує, коли винятку не було
}
```

Такі заплутані алгоритми не бувають надійними. Нам потрібно, щоб функція закрила файл перед завершенням своєї роботи незалежно від того, чому це завершення відбулось (штатно чи завдяки запуску винятка). Саме так спрацьовують деструктори об'єктів: і виконання *return*, і виконання *throw* в тілі функції зумовлює вилучення з автоматичної пам'яті всіх локальних змінних (для вилучення об'єктів автоматично викликаються деструктори). Тому запит на використання ресурсу та його звільнення потрібно інкапсулювати в класі:

```
class FilePtr {
     FILE* p;
public:
     FilePtr(const char* n, const char* a) { p = fopen(n,a); } // захопити ресурс
     ~FilePtr() { fclose(p); }                 // звільнити за будь-яких обставин
     FILE* operator() { return p; }  };        // надати доступ до ресурсу
```

Тепер функція використання файла запишеться стисло і ясно:

```
void use_file(const char* file_name) {
     FilePtr f(file_name, "w");
// сміливо працюємо з файлом f
     fprintf(f()," value is %f", value);
}
```

Звичайно, якщо взаємодію з файлом організувати за допомогою потокових об’єктів, то Вам не доведеться турбуватися про власні класи – все вже є в стандартних потокових класах.

Описану техніку використання ресурсу можна використати для керування динамічною пам’яттю. Нижче наведено просту реалізацію «розумного вказівника», який сам «прибирає за собою» – звільняє захоплену динамічну пам’ять.

```
class DoublePtr {
     double * p;
public:
     DoublePtr() { p = new double; }
     ~DoublePtr() { delete p; }
     double& operator*() { return *p; }
};
```

Оголошення змінної типу *DoublePtr* автоматично створить нову динамічну змінну дійсного типу. Як тільки ця змінна перестане існувати, динамічна пам’ять автоматично звільниться.

На завершення згадаємо про ще одну можливість використання винятків. Вони не мусять сигналізувати про помилки! Ви можете використати *throw myObject* для того, щоб швидко припинити виконання цілого ланцюжка викликаних функцій і транспортувати дані в запущеному об’єкті з «найглибшої» функції кудись нагору, де цей об’єкт буде перехоплено. Такий спосіб використання винятка можна порівняти з використанням інструкції переходу назовні з цілої групи глибоко вкладених (один в одного) циклів. Суть така ж: швидко передати керування, оминаючи звичайні шляхи. Зрозуміло, що для застосування таких «екстремальних» підходів потрібні вагомі причини.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Бьерн Страуструп Язык программирования С++.