Лабораторная работа №7 «Обработка исключений»

Цель

Научиться обрабатывать исключительные ситуации различных типов, устанавливать исключения, создавать классы исключений, выводить данные о возникшей ошибке.

Теоретическая информация

Исключения в C++

C++ обеспечивает встроенный механизм обработки ошибок, называемый *обработкой исключительных ситуаций*. Благодаря обработке исключительных ситуаций можно упростить управление и реакцию на ошибки времени исполнения. Обработка исключительных ситуаций в C++ строится с помощью трех ключевых слов: **try, catch, throw.** Операторы программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке **try**. Если исключительная ситуация (т.е. ошибка) имеет место внутри блока **try**, искусственно она генерируется (с помощью **throw**). Перехватывается и обрабатывается исключительная ситуация с помощью ключевого слова **catch.** Любой оператор, который генерирует исключительную ситуацию, должен выполняться внутри блока **try**. (Функции, которые вызываются внутри блока **try** также могут генерировать исключительную ситуацию). Любая исключительная ситуация должна перехватываться оператором **catch**, который следует непосредственно за блоком **try**, генерирующим исключительную ситуацию.

```
try { // блок try
}
catch (type1 arg1) { // блок catch
}
catch (type2 arg2) { // блок catch
}
...
catch (typeN argN) { // блок catch
}
```

Блок **try** должен содержать ту часть программы, в которой вы хотите отслеживать ошибки. Это могут быть несколько операторов внутри одной

функции, так и все операторы функции **main()** (что естественно вызывает отслеживания ошибок во всей программе).

Когда исключительная ситуация возникает, она перехватывается соответствующим ей оператором **catch**, который ее обрабатывает. С блоком **try** может быть связано более одного оператора **catch.** То, какой конкретно оператор **catch** используется, зависит от типа исключительной ситуации. Т.е., если тип данных, указанный в операторе **catch**, соответствует типу исключительной ситуации, то выполняется данный оператор **catch**. А все другие операторы блока **try** пропускаются. Если исключительная ситуация перехвачена, то аргумент ***arg*** получает ее значение. Можно перехватить любые типы данных, включая и создаваемые вами типы.
Оператор **throw** должен выполняться либо внутри блока **try,** либо в любой функции, которую этот блок вызывает. Здесь *исключительная_ситуация* - это вызываемая оператором исключительная ситуация. Синтаксически выражение **throw** появляется в двух формах.

```
throw
throw выражение
```

Выражение **throw** устанавливает исключение. Самый внутренний блок **try**, в котором устанавливается исключение, используется для выбора оператора **catch**, который обрабатывает исключение. Выражение **throw** без аргумента повторно устанавливает текущее исключение. Обычно оно используется для дальнейшей обработки исключения вторым обработчиком, вызываемый из первого.

Рассмотрим пример работы исключительной ситуации.

```
void main(){
      try{ throw 10; }
      catch(int i) { cout << " error " << i << endl; }
      return;
}
```

На экран выведется сообщение: *error 10.* Значение целого числа, выданное через *throw i*, хранится до завершения работы обработчика с целочисленной сигнатурой *catch(int).* Это значение доступно для использования внутри обработчика в виде аргумента.

Пример переустановки исключения выглядит следующим образом:

```
catch(int n) { . . .
throw; // переустановка
}
```

Поскольку предполагается, что установленное выражение имело целый тип, переустановленное исключение также представляет собой постоянный целый объект, который обрабатывается ближайшим обработчиком, подходящим для этого типа. Концептуально установленное выражение передает информацию в обработчики. Часто обработчики не нуждаются в этой информации. Например, обработчик, который выводит сообщение и аварийно завершает работу, не нуждается ни в какой информации от окружения. Однако, пользователь может захотеть выводить дополнительную информацию или использовать ее для принятия решения относительно действий обработчика. В таком случае допустимо формирование информации в виде объекта.

В C++ блоки **try** могут быть вложенными. Если в текущем блоке **try** нет соответствующего обработчика, выбирается обработчик из ближайшего внешнего блока **try**. Если он не обнаружен и там, тогда используется поведение по умолчанию. **catch** выглядит подобно объявлению функции с одним параметром без возвращаемого типа:

```
catch (char *message) { cerr << message << endl; exit(1); }
catch (...) { // действие, которое нужно приянть по умолчанию.
      cerr << " That's all folks" << andl; abort();
}
```

В списке аргументов допускается сигнатура, которая соответствует любому параметру. Кроме того, формальный параметр может быть абстрактным объявлением. Это значит, что он может иметь информацию о типе без имени переменной. Обработчик вызывается соответствующим выражением **throw**. При этом, фактически, происходит выход из блока **try**. Система вызывает функции освобождения, которые включают деструкторы для любых объектов, локальных для блока **try**.

Синтаксически Спецификация исключения представляет собой части объявления функции и имеет форму:

```
заголовок_функции throw (список типов)
```

Список типов - это список типов, которые может иметь выражение **throw** внутри функции. Если этот список пуст, компилятор может предположить, что **throw** не будет выполняться функцией.

```
void foo() throw(int,over_flow);
void noex(int i) throw();
```

Если спецификация исключения оставлена, тогда возникает допущение, что такой функцией может быть установлено произвольное исключение. Хорошей практикой программирования будет показать с помощью спецификации, какие ожидаются исключения.

Пример кода, реализующего исключения при конструировании объекта

```
vect::vect(int n) {
      if(n <1) throw(n);
      p=new int [n];
      if(p==NULL) throw("NOT MEMORY");
}
void g(int m) {
      try { vect a(m); }
      catch (int n) {
            cerr << "SIZE ERROR" << n << endl;
            g(10); // повторить g с допустимым размером
      }
      catch(const char *error) { cerr << error << endl; abort(); }
}
```

Обработчик заменил запрещенное значение на допустимое значение по умолчанию. Это может быть приемлемо на этапе отладки системы, когда большинство подпрограмм объединяются и проверяются. Система пытается продолжать обеспечивать дальнейшую диагностику. Это аналогично компилятору, пытающемуся продолжать анализировать неправильную программу после синтаксической ошибки. Часто компилятор предоставляет дополнительны сообщения об ошибках, которые оказываются полезными.

Вышеупомянутый конструктор проверяет только одну переменную на допустимое значение. Однако, при такой форме записи разделение того, что является ошибкой и того, как она обрабатывается, очевидно. Это иллюстрирует ясную методологию разработки кода. Более обобщенно, конструктор объекта может выглядеть так:

```
Object::Object(аргументы) {
     if(недопустимый аргумент 1) throw выражение 1;
     if(недопустимый аргумент 2) throw выражение 2;
     . . . // попытка создания
}
```

Конструктор **Object** теперь обеспечивает набор выражений для установки запрещенного состояния. Теперь блок **try** может использовать информацию для восстановления или прерывания неправильного кода.

```
try { // оказоустойчивый код }
catch(объявление 1) { /* восстановление этого случая */ }
catch(объявление 2) { /* восстановление этого случая */ }
catch(объявление K) { /* восстановление этого случая */ }
// правильные или восстановленные переменные состояния теперь допустимы
```

Когда существует много определенных ошибочных условий, удобных для состояния данного объекта, может быть использована иерархия классов исключений. Эти иерархии позволяют соответственно упорядочному множеству **catch** обрабатывать исключения в логической последовательности. Тип базового класса в списке **catch** должен следовать после типа порожденного.

Философия восстановления после ошибок

Восстановление при возникновении ошибок - в основном имеет отношение к правильности написания программы. Восстановление при возникновении ошибок основывается на передачи управления. Недисциплинированная передача управления ведет к хаосу. При восстановлении отказов предполагается, что исключение нарушило вычисления. Продолжать вычисления становится опасно. Обработка исключений влечет за собой упорядочное восстановление при появлении отказа. В большинстве случаев программирование, которое вызывает исключения, должно выводить диагностическое сообщение и элегантно завершать работу. При специальных формах обработки, типа работы в режиме реального времени и при отказоустойчивом вычислении существует необходимость в том, чтобы система не приостанавливалась. В таких случаях героические попытки восстановления узаконены. C++ использует модель завершения, которая вынуждает завершать текущий блок **try**. При этом режиме пользователь или повторяет код, игнорируя исключение, или подставляет результат по умолчанию и продолжает. При повторении кода более вероятно получить правильный результат. Опыт показывает, что обычно код слабо

комментируется. Хорошо спланированный набор ошибочных условий, обнаруживаемых пользователями АТД - важная часть проекта. Если при нормальном программировании слишком часто обнаруживаются ошибки и происходит прерывание - это признак того, что программа плохо обдумана.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианта лабораторной работы №6.
2. Написать программу, в которой перехватываются исключения типа int, string. Сгенерировать исключительные ситуации.
3. Добавить к программе перехват любой исключительной ситуации catch(…).
4. Добавить к программе перехват 2-3 исключительных ситуаций стандартных типов (std::invalid_argument, std::length_error, std::out_of_range или другие).
5. Создайте два собственных класса ошибки, наследуемых от стандартного. Добавить к программе перехват исключительных ситуаций созданных типов.
6. Программа должна демонстрировать обработку исключительных ситуаций на верхнем уровне (функция main), возникающих при вложенных вызовах методов объектов.
7. Программа должна демонстрировать локальную обработку исключительных ситуаций без передачи ее обработчику более высокого уровня.
8. Сделать выводы.

Пример упрощенного варианта реализации

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class MyException : public invalid_argument {                  //собственный класс
                            //исключений, наследуемый от исключения invalid_argument
public:
	MyException(string str) : invalid_argument(str) {}
};

template<class T>
class MyArray {
	T* arr;
	int count = 0;
	int size;
public:
	MyArray(int n) {
		arr = new T[n];
```

```
		size = n;
	}
	~MyArray() {
		delete [] arr;
	}
	void addItem(T obj) {
		try {				//обработка исключения внутри метода
			if (obj == NULL) {
				throw MyException("Argument is null");
			}
			if (count < size) {
				arr[count] = obj;
				count++;
			}
		}
		catch (MyException& my) {
			cout << "Error: " << my.what() << endl;
		}
	}
	T getItem(int i) {
		if (i >= size) {
			throw exception();
		}
		return arr[i];
	}
	int findItem(T obj) {
		int index = -1;
		for (int i = 0; i < count; i++) {
			if (arr[i] == obj) {
				index = i;
				break;
			}
		}
		return index;
	}
};


class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
		author = "No author";
	}
	Article(string _name, string _author) {
		name = _name;
		author = _author;
	}
	~Article() {
	}
	virtual void print() = 0;
```

```
};

class Scientific : public Article {
	string science;
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
	}
	Scientific(string _name, string _author, string science): Article(_name, _author){
		this->science = science;
	}
	~Scientific() {
	}
	void print() {			//метод вывода
		cout << "Scientific -=-=- Name: " << getName() << " Author: "
			<< getAuthor() << " Science: " << getScience() << endl;
	}
};

class News : public Article {
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
		this->area = area;
	}
	News() {
		area = "No area";
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
	}
	~News() {
	}
	void print() {
		cout << "News -=-=- Name: " << getName() << " Author: " << getAuthor()
				<< " Area: " << getArea() << endl;
	}
};

int main()
{
	try {
		MyArray<int> intArr(3);
		intArr.addItem(12);
		intArr.addItem(22);
		cout << "Enter 0 for zero exception" << endl;
		char zero;
		cin >> zero;
		int izero = atoi(&zero);	//возможна ошибка конвертации в число
		if (izero == 0) {
			throw string("Error: Devide by zero or incorrect char");
							//генерация ошибки при делении на
0
		}
```

```
        else intArr.addItem(30 / izero);

        cout << "Enter count more then 3 for exception" << endl;
        int count;
        cin >> count;
        for (int i = 3; i < count; i++) {
               if (count > 3) {
                      throw count;         //генерация ошибки при попытке
                                           //выйти за границы массива
               }
               intArr.addItem(10);                        //возможна ошибка
                                                          //переполнения массива
        }
        cout << "Index of int: " << intArr.findItem(22) << endl;

        cout << "Enter index more then 3 for exception" << endl;
        int index;
        cin >> index;
        cout << "Item: " << intArr.getItem(index) << endl; //возможна ошибка
                                    //обращения к несуществующему элементу массива
}
catch (string str) {
        cout << str << endl;
}
catch (int i) {
        cout << "Error: Count is very large, count = " << i << endl;
}
catch (...) {
        cout << "Other ecxeption " << endl;
};

try {
        MyArray<char> charArr(3);
        charArr.addItem('a');
        charArr.addItem('b');

        cout << "Enter incorrect symbol for exception (correct is a-z)" << endl;
        char sym;
        cin >> sym;
        if (sym < 'a' || sym>'z') {
               throw invalid_argument("Incorrect symbol (correct is a - z)");
        }
        charArr.addItem(sym);

        cout << "Index of char: " << charArr.findItem('d') << endl;
}
catch (invalid_argument err) {
        cout << "Error: " << err.what() << endl;
}

try{
        Scientific* sc1 = new Scientific();
        Scientific* sc2 = new Scientific("Biology article", "Petrov", "ecology");
        News* n1 = NULL;
        News* n2 = new News("Top news", "Sidorov", "Politics");

        MyArray<Article*> myArr(4);
        myArr.addItem(sc1);
        myArr.addItem(sc2);
        myArr.addItem(n1);   //ошибка добавления null,
                                    //обрабатывается в методе addItem
        myArr.addItem(n2);
        cout << "Index of Article: " << myArr.findItem(n2) << endl;

        delete sc1;
```

```
			delete sc2;
			delete n2;
		}
		catch(...) {
			cout << "Error..." << endl;
		}
}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

Укажите в отчете:

1. Тему лабораторной работы, свою фамилию и имя, группу, вариант задания;
2. Листинг программы;
3. Скриншоты результатов выполнения программы;
4. Краткие пояснения к алгоритму работы;
5. Выводы.

Контрольные вопросы

1. Что такое исключительная ситуация?
2. Что произойдет, если поставить catch(…) первым в списке обработчиков?
3. Могут ли быть несколько обработчиков catch для одного блока try?
4. Как получить информацию о возникшей ошибке?
5. Какие классы исключений вы знаете?