Пермский филиал федерального государственного автономного
образовательного учреждения высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет экономики, менеджмента и бизнес-информатики

Чепоков Елизар Сергеевич

SEH

Реферат

студента образовательной программы «Программная инженерия»
по направлению подготовки *09.03.04 Программная инженерия*

Доцент кафедры
информационных
технологий в бизнесе
_____
Л. Н. Лядова

Пермь, 2020 год

Обработка исключительных ситуаций

Структурированная обработка исключений (SEH) – это предоставляемый системой сервис, вокруг которого библиотеки современных языков программирования реализуют свои собственные функции для работы с исключениями. Предназначена для описания реакции программы на ошибки времени выполнения и другие возможные исключения, которые могут возникнуть при выполнении программы и приводят к невозможности дальнейшей отработки программой её базового алгоритма.

Существует два принципиально разных механизма функционирования обработчиков исключений.

- Обработка с возвратом подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация и выполнение программы продолжается.
- Обработка без возврата заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на безусловный переход к заданному оператору.

Область действия обработчиков начинается специальным ключевым словом **try** или просто языковым маркером начала блока и заканчивается перед описанием обработчиков (catch, except, resque). Обработчиков может быть несколько, один за одним, и каждый может указывать тип исключения, который он обрабатывает. Как правило, никакого подбора наиболее подходящего обработчика не производится, и выполняется первый же обработчик, совместимый по типу с исключением. Поэтому порядок следования обработчиков имеет важное значение: если обработчик, совместимый с многими или всеми типами исключений, окажется в тексте прежде

специфических обработчиков для конкретных типов, то специфические обработчики не будут использоваться вовсе.

В качестве нарушения было выбрано исключение, возникающее при попытке записи значения элемента, память для которого не выделена. Так как обращение идет напрямую к памяти, то процессор может заранее предусмотреть ошибку, декодировав адрес, по которому обращаются к памяти.

1. EXCEPTION_EXECUTE_HANDLER – позволяет обработать получившееся нарушение, передав управление обработчику в блоке __except, следующем за данным фильтром.

```
void fault(){
	char* el = NULL;
	__try {
		*el = 'A';
	}
	__except (EXCEPTION_EXECUTE_HANDLER)  {
		cout << "Error\n";
	}
}
```

2. EXCEPTION_CONTINUE_EXECUTION – фильтр передает управление на инструкцию, которая и вызвала исключение, поскольку идет расчет на то, что значения изменились и теперь не вызовут ошибку, при этом само исключение не обрабатывается, а выполнение программы продолжается.

```
void fault2(){
	char* el = NULL;
	__try {
		*el = 'A';
	}
	__except (EXCEPTION_CONTINUE_EXECUTION)	{
		cout << "Error\n";
	}
}
```

3. EXCEPTION_CONTINUE_SEARCH – указывает на то, что обработка исключения может быть найдена выше по стеку.

```
void fault3(){
	char* el = NULL;
	__try {
		*el = 'A';
	}
	__except (EXCEPTION_CONTINUE_SEARCH)  {
		cout << "Error\n";
	}
}
```

Ловушки

Для демонстрации ловушки в рамках переполненного стека была организована бесконечная рекурсия внутри защищенного участка кода. Поскольку процессор заранее не может предусмотреть, какой результат получится при выполнении данной части кода, то и заранее предусмотреть исключение он не в силах.

1. EXCEPTION_CONTINUE_EXECUTION – при нахождении ошибки фильтр вновь направляет команду продолжить выполнение программы, проигнорировав исключение. В случае с ловушкой программа перестает работать из-за нарушения защиты, а не переполнения стека.

```
int recurtion(int num)
{
	return recurtion(num*num);
}

int trap()
{
	int res = 0;
	__try
	{
		res = recurtion(100);
	}
	__except (EXCEPTION_CONTINUE_EXECUTION)
	{
		cout << "Exception code: " << GetExceptionCode() << endl;
		cout << "DO NOT WORK" << "\n";
	}
	return res;
}
```

2. EXCEPTION_CONTINUE_SEARCH – при запуске с отладкой выводится сообщение, указывающее на то, что произошло переполнение стека, но ошибка была не обработана, поскольку фильтр указывает, что в данном случае ошибка не обрабатывается, а производится поиск обработчик выше

```
int trap2(){
	int res = 0;
	__try  {
		res = recurtion(100);
	}
	__except (EXCEPTION_CONTINUE_SEARCH)   {
		cout << "Exception code: " << GetExceptionCode() << endl;
		cout << "DO NOT WORK" << "\n";
	}
	return res;
}
```

3. EXCEPTION_EXECUTE_HANDLER – выполняется код в блоке __except, поскольку данный фильтр подразумевает, что обработчик данного исключения находится прямо здесь.

```
int trap3(){
	int res = 0;
	__try  {
		res = recurtion(100);
	}
```

```
	__except (EXCEPTION_EXECUTE_HANDLER)  {
		cout << "Exception code: " << GetExceptionCode() << endl;
		cout << "DO NOT WORK" << "\n";
	}
	return res;
}
```

Собственные исключения

Для выполнения обработки исключений собственными средствами реализованы следующие синтаксические конструкции: catch и throw.

```
class Exception : public std::exception{
private:
	std::string m_error;
public:
	MyException(std::string error) : m_error(error){}
};

int myexc(){
	int a = 0, b = 8;
	try   {
		if (a == 0) throw MyException("Division by 0");
		else a = b / a;
	}
	catch (MyException){
		return 0;
	}
	return a;
}
```

Если знаменатель равен 0, то выбрасывается исключение, которое было создано раннее в классе Exception, после чего catch ловит его и выполняет действия, заключенные в фигурных скобках. В противном случае происходит деление чисел.

```
int filter(int code, struct _EXCEPTION_POINTERS *ep){
	if (code == EXCEPTION_INT_DIVIDE_BY_ZERO) return
EXCEPTION_EXECUTE_HANDLER;
	else return EXCEPTION_CONTINUE_SEARCH;
}

int del(){
	int a = 0, b = 8;
	__try  {
		a = b / a;
		a = b + a;
	}
	__except (filter(GetExceptionCode(), GetExceptionInformation())){
		return 0;
	}
	return a;
}
```

В данном примере в зависимости от того, какой код ошибки произошел (код получаем с помощью функции GetExceptionCode()), выполняется либо ее обработка (EXCEPTION_EXECUTE_HANDLER) – при условии, что происходит деление на 0, либо дальнейший поиск обработчика нарушения (EXCEPTION_CONTINUE_SEARCH). В результате возвращается 0.

```
int myexc2(){
	short int el = 32766, el2 = 2;
	__try  {
```

```
            el = el1 + el2;
            if (el < 0) throw 1;
            return 0;
      }
      __except (EXCEPTION_EXECUTE_HANDLER)   {
            cout << "Exception code: " << GetExceptionCode() << endl;
            return 32767;
      }
}
```

el достигает границы short int, из-за чего становится отрицательным числом. Переполнение этого типа обнаруживается, и программа заходит в блок __except, где происходит обработка и возвращение максимального значения для переменной типа short int.