Лекция 33

Иерархия исключений

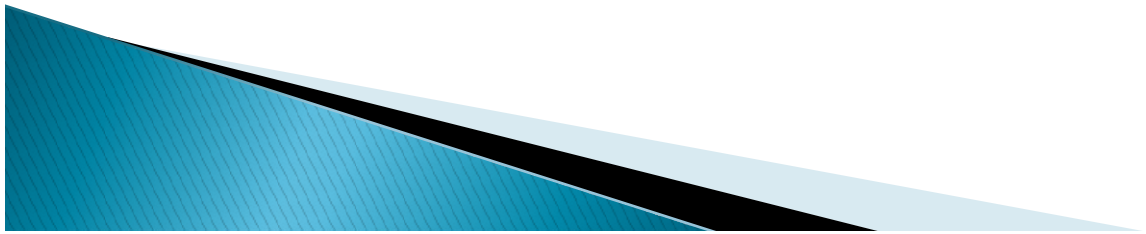
Использование классов для обработки исключений

Исключения – это механизм обработки ошибок с использованием систематического, объектно-ориентированного подхода.

Причиной исключительных ситуаций могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и др.

Особый интерес представляют исключения, сгенерированные самим пользователем.

- При генерации собственных исключений имеет смысл создать специальный класс для работы с ними.
- Рассмотрим пример такого класса, который позволяет получать и обрабатывать строковые сообщения, в которых поясняется причина возникшей исключительной ситуации.



Пример

```
class TMyException {
    char* msg;
    void operator =(const TMyException& e) {}
public:
    TMyException(const char* str) {
        assert(str != NULL);
        msg = new char[strlen(str) + 1];
        strcpy(msg, str);
    }
    TMyException(const TMyException& e) {
        msg = new char [strlen(e.msg) + 1];
        strcpy(msg, e.msg);
    }
    const char* getMessage() const { return msg;}

    ~TMyException() { delete [] msg; }
};
```

Перегруженная операция присваивания в описании, защищенном спецификатором доступа **private**, блокирует выполнение этой операции. Конечно, сложно предположить, для чего нужно присваивать значение одного объекта-исключения другому, но мы должны застраховаться и от таких ситуаций!

Функция **assert** аварийно завершает работу программы, если выражение, заданное в параметре, ложно (вызывает **abort()**). (В примере выражение будет ложно, если строка будет пустой.)

Написание конструктора копирования обязательно – ведь он может неявно вызываться в блоке **catch**.

Генерация исключения из описанного класса может выглядеть так:

```
throw TMyException
        ("Extra data in  input file");
```

Обработчик исключения:

```
try {
// здесь может быть сгенерировано собственное исключение
}
catch (TMyException e) {
 cout << "ERROR! " << e.getMessage() << endl;
}
```

Проблемы при работе с классом exception

Использование только класса стандартного exception при выбросе исключений приводит к тому, что анализ перехваченного исключения и его правильная обработка становится сложной задачей:

```
void f1() { …
   throw exception(”Error”);
… }
void f2() { …
   throw exception(”Error”);
… }
try {
   f1();
   f2();
}
catch (const exception& e) {
// непонятно, где выброшено исключение
}
```

Построение иерархии исключений

Одно из решений этой проблемы состоит в создании собственных классов-исключений – потомков класса exception. Работа этих исключений ничем не отличается от работы exception, поэтому достаточно написать конструкторы (которые не наследуются):

```
#include <exception>
...
class MyException: public exception {
private:
    MyException();
public:
    MyException(const char * const s): exception(s){}
    MyException(const MyException& e): exception(e){}
};
```

Обработка исключения

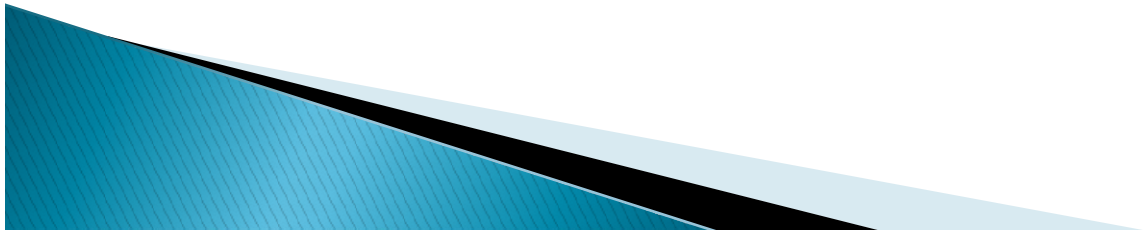
```
try {
…
   throw(MyException(”Unknown internal error!”));
…
}
catch (MyException e)
    {cout << e.what() << endl;}
```

Обработка иерархии исключений

Важен порядок перехвата исключений: вначале обрабатываются потомки, а потом – предки! Несоблюдение этого правила приведёт к тому, что специфическая обработка исключений-потомков будет пропущена.

```
try {
  …
}
catch (const MyException& e) {
  …
}
catch (const exception& e) {
  …
}
catch (...) {
  …
}
```

Конец лекции



Спасибо за внимание