Лекция 14. Исключения.

Евгений Линский

Виды ошибок

Виды ошибок:

- Ошибки “по вине программиста”. Примеры:

```
char *s = NULL;
size_t l = strlen(s);
Array a(-1);
```

  Обработка ошибок:

  - Лучше выявить на стадии тестирования (assert, unit test, etc).
  - При выполнении “идеальной” программы их не происходит.
  - Библиотека C подобные ошибки не обрабатывает.
  - Библиотека C++ — по-разному в разных местах: vector.at(i) и vector.operator[i].
  - Обрабатывть или нет — на усмотрение программиста.

- Ошибки “по вине окружения программы”. Примеры:
  - Файл не существует.
  - Сервер разорвал сетевое соединение.
  - Пользователь вместо числа ввел букву.

  Обработка ошибок:

  - Могут произойти и при выполнении “идеальной” программы.
  - **Обязательно надо обрабатывать!**

Обработка ошибок

- Проверка на наличие ошибки (if)
- Освободить ресурсы

```
delete [] array;
fclose(f);
```

- Сообщить пользователю и/или вызывающей функции

```
FILE *f = fopen("a.txt", "r");
if( f == NULL ) {
  printf("File a.txt not found\n");
}
//or
if( f == NULL ) {
  return -1;
}
```

- Предпринять действия по восстановлению от ошибки (например, не смогли соединиться — попробовать еще три раза)

Обработка ошибок в C-style

Информация об ошибки: через возвращаемое значение и через глобальную переменную

```
FILE* fopen(...) {
  if(file not found) {
    errno = 666;
    return NULL;
  }
  if(permission denied) {
    errno = 777;
    return NULL;
  }
  ...
}
```

Обработка ошибок в C-style

- Мало информации! Не знаем причину: нет файла, нет прав доступа, ...

```
FILE *f = fopen(...);
if( f == NULL ) {
  ...
}
```

- Глобальная переменная *errno* хранит код ошибки (*strerror(..)* — сообщение об ошибки)

```
#include <errno.h>
FILE *f = fopen(...);
if( f == NULL ) {
  switch(errno) {
    ...
  }
}
```

Почему не всем нравится C-style?

Attention! Holy war!

- Не всегда хватает диапазона возвращаемых значений функции

```
class Array {
  int *a;
public:
  //return -1 in case of index out-of-bound ???
  int get(size_t index);
};
int r1 = atoi("0");
int r2 = atoi("a");
```

- Код логики и обработка ошибок перемешены

```
r = fread(...);
if (r < ...) {
  //error
}
r = fseek(...);
if (r != 0 )  {
  //error
}
```

C++-style: исключения (exception)

```
class MyException {
private:
  char message[256];
  // possible fields: filename, line, function name
public:
  const char* get();

};

double divide(int a, int b) {
  if(b == 0) {
    throw MyException("Devision by zero");
  }
  return a/b;
}
```

C++-style: исключения (exception)

```
try {
  x = divide(c, d);
}
catch(MyException& e) {
  std::cout << e.get(); // 1. tell user
  // 2. delete [] ...; free resources
  // 3. throw e; inform caller function
}
```

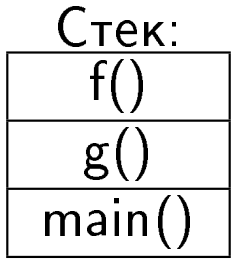
Stack unwinding - I

```
f() {
   if(...) throw MyException("Error: ....");
   printf(...);
}

g() { f(); }

main() {
   try {
      g();
   }
   catch(MyException& e) { ... }
}
```

Stack unwinding - II



Если “брошено” исключение:

- Нормальный процесс выполнения программы заканчивается, т.е. поток управления до *printf* в *f()* не дойдет.
- Начинается stack unwinding: последовательный просмотр стека до тех пор, пока не будет найден подходящий по типу исключения (в нашем примере тип *MyException*) блок *try/catch*.
- Если подходящий блок не был найден, и исключениие “вылетело” за *main()*, то программа аварийно завершается.

Несколько типов исключений - I

Если в программе несколько подсистем (GUI, Network, Model), то можно:

1. у каждой подсистемы сделать свой тип исключения (GuiException, NetworkException, ModelException)
2. обрабатывать их по-разному

```
main() {
  try {
    doGame();
  }
  catch(GuiException& e) {
    showMessageBox(...);
  }
  catch(NetworkException& e) {
    showMessageBox(...);
    logger.log(...)
  }
  catch(ModelException &e) {
    logger.log(...)
  }
}
```

Несколько типов исключений - II

- Можно организовать иерархию наследования исключений, чтобы не пропустить какое-нибудь в main.

```
class MyException {};
class GuiException : public MyException {};
class NetworkException : public MyException {};
class ModelException : public MyException {};
```

- Однако, надо помнить, что в блоке *try/catch* выбирается первый *catch*, подходящий по типу.
  - Всегда будет срабатывать первый *catch*

```
try { ... }
catch(MyException &e) { ... }
catch(GuiException &e) { ... }
```

  - Правильный порядок обработки

```
try { ... }
catch(GuiException &e) { ... }
catch(MyException &e) { ... }
```

- В STL все исключения — наследники *std::exception*

Как поймать исключение любого типа?

```
try {
  doMainWork();
}
catch(...) {   // ...  -- catch anything
  throw;       // without argument -- rethrow anything
}
```

Stack unwinding - III

Во время stack unwinding вызываются деструкторы!

```
f() {
   MyArray a(100);
   if (...) throw MyException(...);
} a.~MyArray()

g() {
   Matrix m(10, 30);
   f();
} m.~Matrix()

main() {
   try {
      g();
   }
   catch(...) { }
}
```

Утечка памяти

```
f() {
  int *buffer = new int [n];
  if( ... ) throw MyExcpetion(...);
  delete [] buffer;
}
```

```
g() {
  Person *p = new Person("Jenya", 36, true);
  divide(c, e); // could throw exception
  delete p;
}
```

При возникновении исключения поток управления до *delete* не дойдет.

Идиома RAII

- Идиома в данном контексте — “так все делают =)”
- RAII — Resource Acquisition Is Initialization (“Взятие Ресурса Должно Происходить через Инициализацию” или как-то так)
- Взятие ресурса нужно “инкапсулировать” в класс, чтобы в случае исключения вызвался деструктор и освободил ресурс.

```
f() {
  MyArray buffer(n);
  if( ... ) throw MyExcpetion(...);
}
```

```
g() {
  unique_ptr p(new Person("Jenya", 36, true)); // or another smart ptr
  divide(c, e); // could throw exception
}
```

Исключения в конструкторе - I

Произошло исключение в divide, объект “недостроен”. Деструкторы у недостроенных объектов не вызываются.

```
class PhoneBookItem {
    PhoneBookItem(const char *audio, const char *pic) {
        af = fopen(audio, "r");
        pf = fopen(pic, "r");
        divide(c, e); // could throw excpetion
        f();
    }

    ~PhoneBookItem() {
        fclose(af);
        fclose(pf);
    }

};
```

Исключения в конструкторе - II

Надо предусмотреть такую ситуацию.

```
class PhoneBookItem {
    PhoneBookItem(const char *audio, const char *pic) {
        try {
            af = fopen(audio, "r");
            pf = fopen(pic, "r");
            divide(c, e); // could throw excpetion
            f();
        }
        catch(MyException& e) {
            fclose(af);
            fclose(pf);
            throw e; //inform caller
        }
    }

    ...
};
```

Исключения в деструкторе - I

Логи посылаются на сервер. За это отвечает объект networkLogger, методы которого могут бросать исключения.

```
class PersonDatabase {
  ~PersonDatabase() {
    // throws exception if server is unavailable.
    networkLogger.log("Database is closed.");
    ...
  }
};

f() {
  PersonDatabase db;
  if(...) throw MyException("Error: disk is full.")
}
```

- Исключение от networkLogger может “подменить” исключение о том, что “места на диске больше нет”, и мы не узнаем истинную причину ошибки.
- Поэтому исключения в деструкторах бросать запрещено!
- Если это происходит, то программа аварийно завершается.

Исключения в деструкторе - II

Надо предусмотреть такую ситуацию.

```
class PersonDatabase {
  ~PersonDatabase() {
    try {
      // throws exception if server is unavailable.
      networkLogger.log("Database is closed.");
    }
    catch(...) { }//catch everything
  }
};
```

Гарантии при работе с исключениями

Гарантии:

- обязательства функции (метода) с точки зрения работы с исключениями
- документация для программиста, работающего с функцией (методом)

Виды гарантий:

- no throw guarantee — не бросает исключений вообще
- basic guarantee — в случае возникновения исключения ресурсы не утекают
- strong guarantee — переменные принимают те же значения, что были до возникновения ошибки

no throw

```
void strlen(const char *s) {
  int count = 0;
  while(*s != 0) {
    s++; count++;
  }
  return count;
}
```

```
void f() {
  try {
    strlen(...);
    divide(a, b);
  }
  catch(...) { //catch everything

  }
}
```

basic guarantee

- Если произойдет исключение, то память “течь” не будет, но измененные элементы array свои значения не восстановят.

```
class PersonDatabase {
  MyVector<Person> array;
  void process() {
    unique_ptr<Person> p(new Person(...));
    for(int i = 0; i < array.length; i++) {
      int a = divide( rand(), rand() ); // could
throw exception
      array[i]->setAge(a);
      std::cout << p;
    }
  }
};
```

- Функции, в которых мы в этой лекциях применяли RAII, обеспечивают как минимум basic guarantee.

strong guarantee

Идиома: copy-and-swap

```
class PersonDatabase {
  MyVector<Person> array;
  void process() {
    unique_ptr<Person> p(new Person(...));
    MyVector<Person> copy(array);

    for(int i = 0; i < array.length; i++) {
      int a = divide( rand(), rand() ); // could throw
exception
      copy[i]->setAge(a);
    }

    array.swap(copy); // swap полей

  }
};
```