Семестр 2. Лекция 3. Исключения. Приведение типов в C++.

Евгений Линский

10 Марта 2017

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() {
  auto_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(...) throws 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() {
    auto_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() {
    auto_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 = copy;

  }
};
```

Приведение типов в C. Воспоминания.

Неявное приведение (может вызвать warning, но не error).
Но можно сделать так: - Wall -Werror (включить все warnings, трактовать их как ошибки).

```
void f(char *p);
int *pi = malloc(100 * sizeof(int));
f(p1);

int a = 65535;
char b = a;
```

Явное (“это неслучайно, я так действительно хочу)”:

```
int a = 5; int b = 6;
double c = a / (double)b;
```

Приведение типов в C++. Воспоминания.

Требуется явное приведение для указателей (кроме приведения к void* и кроме приведения к базовому классу).

```
void* f();
int* pi = (int*)f();

// class B : public class A
void print(const A* p);

B b;
print(&b);
```

Для классов:

```
class BigInt {
  BigInt(int a);   // from int
  operator int(); // to int

  BigInt(const Complex&); // from Complex
  operator Complex();     // to Complex
};
```

C++: синтаксис для явного приведения.

Зачем?Проще искать с помощью grep в коде. Более точно выражается намерение программиста:

- разные cast'ы для разных случаев (static_cast, reinterpret_cast, cont_cast)
- компилятор делает более точную проверку

static_cast (примитивные типы; классы, связанные наследованием; приведение к void*; пользовательские преобразования BigInteger->int).

```
int a = 65535;
char b = static_cast<char>(a);

// class B: public class A
void f(B *b);
A *a = new B();
f(static_cast<B*>(a));

char *pc = ...; int *pi = ...;
pc = static_cast<char*> pi; //error
```

C++: синтаксис для явного приведения.

reinterpret_cast (указатели разных типов).

```
void* f();
int* pi = reinterpret_cast<int*>(f());

char *pc = ...; int *pi = ...;
pc = reinterpret_cast<char*> pi; //ok
```

const_cast (добавление/удаление const; лучше не использовать)

```
char const *p1 = "Hello";
char *p2 = const_cast<char*>(p1);
p2[0] = 'h'; // undefined behaviour
```

RTTI.

Технология RTTI (Run-Time Type Information):

- Оператор dynamic_cast осуществляет безопасное преобразование указателя на базовый класс в указатель на производный класс (ссылки).
- Оператор typeid возвращает фактический тип объекта для указателя (ссылки).

dynamic_cast

```
// class B: public class A;
// class C: public class A;
void f(B *b);

A *a = new C();
f(static_cast<B*>(a)); // no errors in compile time
                       // undefined behaviour in runtime

if (dynamic_cast<B*>(a) != 0) {
  f(static_cast<B*>(a));
}
```

RTTI.

```
#include <typeinfo>
// class C: public class A;
A   *a = new C();
type_info ti = typeid(*a);   // reference required
ti.name();   // "C"
```

RTTI.

- Работает только для классов с виртуальными функциями (инфо о типе хранится в таблице виртуальных функций)
- Чаще всего используется, когда надо “сделать костыль” для существующего кода, который нельзя перепроектировать (например, большая система с долгой историей)

```
class Shape {
  virtual draw() = 0;
};

draw_all(Shape* shapes, size_t n) {
  for(int i = 0; i < n; i++) {
    Shape *p = shapes[i];
    p->draw();
    if(dynamic_cast<AnimatedShape*>(p) != 0) {
      p->animate_draw();
    }
  }
}
```