Исключения в ООП. // stack unwinding <https://www.youtube.com/watch?v=wAJB8gd2xvY>

Александр Смаль\часть 2 - 2017\Обработка ошибок.pdf

Какие бывают ошибки?

1.Ошибки по вине программиста (что-то не так написал, поделил на ноль, передал null, вышел за границы массива). Предусмотреть все места в программе, где могут произойти такие ошибки – трудоемкий процесс. Главное, если это сделать, обработать всё, программа будет медленной. Пример: при каждом обращении к массиву проверять не вышли ли мы за границы.

2.По вине среды, в которой программа запускается (могут быть всегда, например нет такого файла, отключено сетевое подключение, на диске закончилось место).

Общее правило: программа должна обрабатывать такие ошибки. Можно выдавать сообщение или предпринимать какие-то действия самостоятельно. Например подождать, повторить в цикле (например, в браузере так делают).

Что нас не устраивает в методах обработки ошибок в Си? int MyArray :: get (int index ) { ... } нас волновало, что нет красивого механизма, который бы позволил сообщить об ошибке. Почему? В виде возвращаемого значения? Что вернуть? -1 ? Не понятно, а что если в массиве по этому индексу и правда лежит -1 ? Сделать глобальную переменную, в которую сообщать об ошибке? Глобальные переменные – зло, плюс это плохо для многопоточных программ. Можно, наверное, дописать ещё один параметр в функцию, и в него записывать результат, такие примеры функций есть. Всё усугубится, если мы захотим об ошибке сообщать несколько больше, чем просто её код. Хочется говорить много, ну например строку выдать :)

Эстетически приятнее разделить код и обработку ошибок в разные места. Пример такой: пусть мы работаем с файлом

fopen, fread, fseek, fread – каждая из этих операций может вызвать ошибку, файл закончился или исчез и т.д. По идее после каждой такой операции нужно писать if() и проверять, не произошла ошибка? Программа становится громоздкой. Плюс сложно отследить логику. Если бы можно было в каком-то блоке описать операции, а потом где-то ниже описать обработку всех ошибок, которые могут возникнуть в таком блоке кода, было бы хорошо.

Также, если есть цепочка вызова функций f->g->h ... и если ошибка произошла в h, бывает важно обработать ошибку в f (допустим в графическом интерфейсе). Не будет толку писать в консоль внутри функции в h, f не увидит и не узнает. Можно было решить это с помощью механизма возвращаемых значений, но будет громоздко, т.к. нужно писать g() if h()==-1 то тоже вернуть -1 или добавить переменную, в которую записывать результат. Конструкция становится очень громоздкой. Хочется передавать сообщение об ошибке на нужный уровень, чтобы остальные уровни игнорировали.

При этом надо упомянуть, что перечисленные причины довольно субъективные. Не все программисты С++ используют исключения. У g++ есть часть операций в библиотеке STL, которая в случае ошибок бросает исключения, но есть ключ, который отключает этот механизм, и исключений в вашей программе не будет. В общем были придуманы исключения, чтобы, в том числе решить перечисленные выше проблемы. Синтаксис:

h() { // Если возникает ошибка, мы можем написать throw

throw 5; // либо throw 9.0; либо throw MyException(“Строка с ошибкой”,ip\_addr) любой объект можно передать

}

Как только выполняется инструкция throw, функция перестаёт выполняться, ошибка начинает распространяться по стеку вызовов. Попадает в вышестоящие функции, если ни одна из них не обработала, то ошибка доходит до main и программа аварийно завершается. Предположим, что f (вышестоящая) хочет ошибку обработать. Мы уже добились того, что можем любое количество информации об ошибке давать и распространять ошибку вверх по стеку вызовов.

void f() { // здесь конструкция будет состоять из двух блоков try и catch

try { // блок с кодом, который надо выполнить

g(); ... разные инструкции, функции, методы объектов и так далее, все что может сделать throw(5)

}

catch (int n) { // блок с обработкой ошибок, можно указать конкретный тип ошибки, которую ловим

} // сюда попадём, если ошибка была брошена throw 5;

catch (MyException& e) { // если написать не по ссылке, то лишний вызов конструктора копий, создание копии.

} // сюда попадём, если было throw MyException( с конструктором )

}

Что можно делать внутри блоков catch ? Можно просто сделать printf и забыть. Или же можно предприняв что-то ошибку бросить дальше, может она нужна выше! Например, в блоке try функции f был открыт файл fopen, вызвана функция g, внизу по стеку вызовов где-то произошла ошибка, то в catch у функции f нужно закрыть файл обязательно, а потом уже можно сообщать куда-то выше: thrown n; или throw e;

Поскольку бросить могут любой тип, то хочется уметь обрабатывать ошибки любого типа, иначе если она долетит до main, то программа аварийно завершится. Можно написать инструкцию, которая ловит любой тип ошибки.

catch (...) // ловим все что угодно, синтаксически здесь нет имени переменной, поэтому вытянуть параметры какие-то не получится, можно только предпринять какие-то общие меры и бросить ошибку дальше throw; без параметров. Если есть несколько блоков throw MyException; ниже наследник throw SuperException; то блоки catch можно располагать по-разному.

try { .... }

catch (MyException& e) { ... }

catch (SuperException& e) { ... }

Алгоритм работы такой: находится первый подходящий сверху вниз блок catch и выполняется. Выполнится первый подходящий, т.к. летит объект класса допустим SuperException, по основному правилу наследования, он является и объектом класса MyException тоже, поэтому нет смысла писать второй блок вообще, он не выполнится если есть наследование. Надо это учитывать и менять порядок блоков catch просто, сначала ловим SuperException. Вообще то, как распространяется Exception – это отдельный механизм, не в куче. Теперь особые случаи.

В STL есть базовый класс std::exception {

virtual const char \* what()=0; };

Это базовый класс для всех исключений STL. Написать throw exception; нельзя, он вроде бы абстрактный, нужно от него наследоваться. Для чего он нужен? Если есть какой-то код, работающий с STL, то очень просто поймать ошибки (не нужно использовать синтаксис с тремя точками от безысходности), мы просто ловим объект этого базового класса и хотя бы what() имеем. Пример при работе с STL:

try {....какой-то код работы с STL ...}

catch (std::exception & e) { ... обработка } // мы всегда точно поймаем всё здесь

Есть традиции в своей программе наследовать исключения от std::exception. Но это вовсе не обязательно. С точки зрения STL есть два больших класса ошибок: runtime\_error и logic\_error. Runtime можно выявить только на стадии исполнения, например: у BitSet есть метод ulong(), который берёт всё, что есть в объекте BitSet, пакует в long и вам возвращает. Если у вас в объекте BitSet лежит что-то, больше, чем 32 бита, то будет overflow. Ещё пример – класс locale, можно в конструкторе указать “Russian” или “SDFSDFSDF” – будет runtime exception. Logic – ошибки, которые можно выявить на этапе компиляции, неправильные индексы, выходы за границы и т.д. Пример: vector.at(i) где i>числа элементов. Есть ещё ошибки bad\_alloc (его кидает new, если память выделить не удалось). Ещё один наследник от exception: ios::base::failure – exception, которые падают при неправильной работе с файлами (fstream, cin, cout всё это бросает такие исключения).

Исключения в конструкторах.

class Storage {

Model\* m1;

Model\* m2;

Storage () {

m1=new Model(); m2=new Model(); }

~Storage () {

delete m1; delete m2; }

}; Беда в том, что у недостроенных объектов деструктор не вызывается, будет утечка памяти.

Storage s; // если например не хватит памяти для второго поля m2 и полетит bad\_alloc то деструктор не вызовется. Что можно сделать? Ну обычно просто завершают программу, памяти не хватает – что тут поделать. То есть в примере new можно оставить как есть. Но если так делать не хочется, важно, чтобы программа дальше жила, важно, чтобы не возникало утечки памяти, то можно исключение прямо здесь в конструкторе и обработать:

**Исключения, приведение типов С++.**

Ещё раз про конструкторы и исключения. Пусть есть приложение записная книжка, внутри есть класс, отвечающий за одну запись (имя+фотография):

class BookEntry {

private:

std::string name;

Image\* image;

public:

BookEntry (const std::string & name\_, const std::string & filename) {

name=name\_;

image = new Image();

image -> load (filename); }

~BookEntry () {

delete image; }

};

Проблема в том, что в строке new Image() может быть указан не тот файл, не того формата и так далее, полетело исключение. Но конструктор до конца не дошёл. В С++ есть правило: если объект не достроен, то деструктор вызывать не нужно. Проблема – утечка памяти. Соответственно, чтобы код работал корректно и не было утечки, нужно обработать исключение внутри конструктора:

try {

name = name\_;

image = new Image (filename); }

catch (...) {

delete image;

throw; } // с name компилятор сам справится. Недостаток в том, что пользователь класса никогда не узнает о проблеме какой-то, поэтому добавляем throw; Чтобы поговорить про остальные особенности использования исключений, надо понять, чем исключение отличается от goto.

Когда мы разбирали, зачем нужны исключения, и какие есть достоинства, мы смотрели примеры: когда исключение происходило где-то глубоко в стеке вызовов, а обработчики были где-то наверху. Удобно, не приходилось нагромождать кодом функции лишний раз, делать какие-то return с параметром и прочее. Если бы дело было только в этом, то в Си был страшный механизм, который позволял делать то же самое, типа goto. Сейчас приведём синтаксис и поймём, чем он плох.

#include <csetjmp> // setjmp.h здесь описана longjmp функция

class Bar {

char\* a;

void create() {

a = new char [100]; }

~Bar(){ delete [] a; }};

void bar() {

foo(); } // Т.е. есть несколько функций, ошибка происходит глубоко, а обработать нужно высоко.

void foo() {

Bar b; b.create(); … работа с сетью

if(error) longjmp(buf, 42); }

jmp\_buf buf;

int main() {

if(setjmp(buf)==0) {

bar(); }

else { …здесь находится обработка ошибки }

} // main – то место, где могут обработать нашу ошибку, только здесь могут решить, что с ней делать. Вызывается функция bar() -> foo() { тут есть действия, которые могут вызвать исключения }, когда ошибка произойдёт, надо сразу попасть в main: if. Как работает? Когда вы вызываете setjmp в буффер запоминается адрес этого места (где находится инструкция setjmp). Когда вызвали setjmp, сюда вернётся 0 и продолжится выполнение, т.е. вызовется bar(). Далее foo() и возможно произойдёт ошибка. Если произойдёт ошибка, то инструкция longjmp(buf,42); - инструкция говорит: “переведи счетчик команд у процессора в то место, которое было запомнено в buf”. Вместо 42 может быть любое число. Мы повторно попадём в инструкцию setjmp, только теперь он вернёт 42 вместо 0. Мы попадём в else, где сможем обработать ошибку. Ещё раз: при первом вызове setjmp в buf записывает адрес этого места в памяти (запоминает), потом longjmp берёт и переводит адрес выполнения на это запомненное место. При повторном вызове setjmp вернёт уже не 0, а ваш код, в данном случае 42. Вообще это такое длинное goto (в обычном goto разрешалось ходить только внутри одной функции, а здесь вообще между функциями). Какие проблемы? Этот механизм Сишный, а Си не знает про деструкторы. Механизм, при котором распространяются исключения (С++) знает про то, что нужно вызывать деструкторы у объектов при переходе по стеку вызовов вверх. Если бы были не longjmp, а throw, то деструкторы вызывались бы. Т.к. longjmp грубо говоря делает call, указатель передвинется в стеке вызовов и всё, а throw знает, что были объекты, надо по стеку пройтись, вызвать деструкторы. Это был пример, как было раньше в средневековье.

Исключения в деструкторах.

class NetWorkConnection {

void connect(); // тоже потенциально могут произойти ошибки

~NetWorkConnection() {

работа с файлами на STL (об ошибках можно сообщать исключениями)

} };

void foo() {

NetWorkConnection nc;

nc.connect(); }

Произошло исключение, началась процедура stack unwinding (проход по стеку вызовов в поиске обработчика исключений). Внутри foo нет обработчика, но есть созданный объект, поэтому вызывается деструктор. В деструкторе могут произойти ошибки, полететь своё исключение. Ситуация плохая, т.к. нет механизма позволяющего лететь одновременно двум исключениям. Если забыть про первый и работать со вторым, плохо, т.к. программист подумает, что проблемы с файлами, хотя на самом деле проблемы были с сетью. Такие ситуации в С++ не допустимы. Деструкторы не должны кидать исключения, иначе если кинет – программа сразу завершится аварийно (Имеется в виду, чтобы исключение не вылетало за него, вверх по стеку вызовов за пределы от деструктора). Как исправить такую ситуацию? Сделать try-catch внутри деструктора.

try { работа с файлами … STL }

catch(…) {обработка }

Идиома (приём, которым часто пользуются) RAII (Resource Acquisition Is Initialization – получение ресурса (выделение памяти и т.п.) должно происходить через инициализацию, т.е. через конструктор объекта, у которого есть деструктор, чтобы деструктор потом вызвался автоматически для освобождения ресурса). Пример:

void f() {

Person \*p = new Person();

работа с файлом на STL … чтение из файлов в p или наоборот

delete p; }

Создали объект в куче, работа с STL потенциально может бросить исключение. Проблема – допустим происходит исключение STL и началась процедура разматывания стека, до строчки delete не дошли.

Есть два решения: 1) Обернуть все try-catch:

try {

Person \*p = new Person;

работа…

delete p; }

catch(std::exception &e) {

delete p; throw e; } Не очень приятно т.к. дублирование кода, если будет больше выделение – нужно больше try/catch делать. Что ещё можно сделать? SmartPointers. В STL есть auto\_ptr. В Boost есть shared\_ptr. Пример:

void f() {

auto\_ptr p(new Person); либо auto\_ptr<Person>(new Person);

работа с файлами STL }

Здесь такой проблемы не будет, т.к. объект статический, выйдем из области видимости – естественным образом у auto\_ptr запустится деструктор, там внутри удалится Person. Либо когда произойдёт исключение, все деструкторы будут вызваны через auto\_ptr опять же. Идея в том, чтобы получение ресурсов перенести в класс, у которого есть деструктор, чтобы при произошедшем исключении всё удалилось. Чем хорошо vector вместо массива использовать? Можно забыть про исключения, т.к. при работе с массивом в куче надо не забыть delete[] вызвать, а vector всё сам сделает через деструктор.

Есть ещё такой синтаксис void foo() throw (AException, BException) { } Идея была в том, чтобы здесь указывались все исключения, которые может бросить эта функция. К сожалению никто не пользуется, т.к. сделать это не удалось (не удалось реализовать хороший анализ).

{

bar(); // но тут могут быть вызовы других функций

throw AException(…);

throw BException(…); // такое компилятор ещё смог бы проанализировать

} но хуже всего будет, если это уйдёт всё к библиотеке, распространённой через объектные файлы, без исходного кода. То есть на самом деле bar может бросить CException, компилятор не может подсказать, что ты должен в foo указать ещё и CException. Подход не прижился в общем-то. В Java используется активно. Если бы это работало, не было бы Catch(…), всегда можно было понять, какие исключения функция бросает.

Гарантии исключений. Если писать программу с использованием исключений и формулировать к ней требования, то это можно сделать следующим образом: несколько уровней гарантий:

а) no throw – если функция не бросает исключений, то для неё эта гарантия (а)

б) basic – то, чем мы занимались, состоит в том, что при произошедшем исключении все ресурсы будут корректно закрыты и система не перейдёт в противоречивое состояние (все delete будут вызваны и прочее)

в) strong – если произошло исключение в процессе операции, объект не должен измениться. Пусть выполняются 3 операции с объектом, на 2 операции произошла ошибка, 1 операцию надо откатить. Пример:

T stack :: pop() {

if(count!=0) return data[--count]; }

С точки зрения исключений здесь может произойти проблема при T каком-то пользовательском типе. При инструкции return data создается объект через конструктор копирования, чтобы в стеке создать возвращаемое значение. Может произойти исключение. С точки зрения strong мы потеряем объект, т.к. count мы уменьшили, а действие не сработало. Поидее count не должен был уменьшиться. Стек должен остаться в том же состоянии, как и был до исключения в return. Можно сделать два разных метода:

void stack :: pop(); // изменяет счетчик

T top(); // возвращает объект типа T

f () {

s.top();

s.pop(); } // всё корректно, т.к. при произошедшем в top исключении pop не выполнился и счетчик не уменьшился, состояние стека мы не разрушили. Казалось бы, всегда нужно делать string гарантии, однако не всегда действие можно откатить. Такой принцип – либо всё выполняется целиком корректно, либо все действия откатываются назад называется транзакцией. Пример: символы вывели на экран, их обратно ведь не засосёшь☺ Получается такая функция не соответствует сторогой гарантии. Из сети назад тоже не заберёшь пакеты. Люди договариваются, давайте будем писать классы, чтобы его методы соответствовали – такой гарантии, такой и такой … Ещё пример:

Идиома swap. То есть объект куда-то копируете, пытаетесь делать набор действий, если исключение происходит, то просто обратно swap и объект опять в исходном состоянии.

void doSomething(T& t) {

t.x+=1; // для этой строки выполняется гарантия no throw

X \*x=new X; // гарантия basic

t.list.push\_back(\*x); // strong, т.к. list не сломается при исключении

delete x; // no throw

} // Есть некоторый объект класса T, есть объект класса X и список list внутри t.

Про всю функцию можно сказать – никакую гарантию она не поддерживает.