Шестой семинар по программированию

Обработка различных ситуаций
Исключения

Были какие-то проблемы с передачей невалидные аргументов в функции (10 в арксинус), указателей на пустую область памяти и так далее.
Как мы обрабатывали такое в С?

1. *abort()* — просто экстренно предотвращает программу (не гарантирует, что будет с нашими файлами, буферами);
2. *exit()* — у нее есть отличие от аборта в том смысле что вот мы работаем с файлами, наружными буферами и вот вопрос, данные из буфера запишутся в файл или нет? В экзите буферы будут скинуты, данные не потеряются в том виде, в котором они были в буфере. Он принимает различные параметры. Если все плохо, то есть такая штука: *EXIT_FAILURE*;
3. *assert()* — помимо остановки программы он также выводит информацию о том, где произошла проблема. «В таком-то файле, на такой-то строке что-то там произошло»: *file.cpp: 80 Line*;

В целом, если мы хотим экстренно завершать, то assert весьма хороший вариант. Но иногда возникают проблемы, так что нужно помнить и про другие команды тоже.
Также можно возвращать наши ошибки, можно передавать указатель на место, где код ошибки сохранить. Int func(int* error); Третий вариант - глобальная переменная. Наиболее С-путь заранее подготовленная erno, erno.h;
В С++ посчитали, что эти методы не достаточны гибкие. Мы могли с этим столкнуться во время проектировки библиотеки. В с++ посидели и захотели создать механизм исключений. Он помогает обрабатывать ошибки, возникающие в коде программы.

throw, try, catch

throw smth; /* под этим smth может быть что угодно. Целочисленные значения, строки и объекты классов, стандартные исключения exception */

Чтобы поймать их нужно использовать другие слова:

try { … } // обрабатывает код внутри этих скобок
catch (exception & x) { … }

// обработка исключений, сложнее края, нужно понять, а что ловить? Если мы знаем, что исключения кидают массив, то нужно ловить массив. Если исключение кидает объект класса, то нужно ловить объект класса.

Обычно записывается
В 98 году в С++ добавили классификацию спецификацию исключений. Пускай у нас какая-то простая функция: *void func() throw (type_exeption t0, type_error t1) {…}* можно после объявления функции в throw блоке перечислить исключения, какие данная функция бросает, чтобы потом помочь разработчику, который будет использовать нашу функцию. На самом деле этот список - не все исключения, которые могут прилететь ввиду больших структур. Но в С++11 это считается устаревшим. Добавили новое слово для спецификаций исключений. Новый noexcept , мы уже использовали это в move конструкторе. Это не то чтобы считается хорошим патерном в использовании, но нужно знать. В С++14 это еще в стандартной библиотеке.
Есть еще одна нотация, точнее, продолжение той нотации;

А как работают функции:

```
void func (int arg0, int arg1);
```

int main () (… func(arg0, arg1); … } // сначала поместиться arg1, затем arg0 на стэк, затем адрес возврата return_agress. Берется адрес с вершины стэка и понимает, куда все же возвращаться. Таким образом работают функции. Помещаются аргументы функции и адресы возврата.

Исключению важно, где кинули и поймали его. То есть блоки throw и catch.

По сути стэк - минимальная вычислительная система, ведь его достаточно для выполнения любой последовательности математических операций. И достаточно для постройки Тьюринг полной машины на ней.

Отсюда есть множество разных интересных следствий:
Исключения очень медленно работают. То есть нам нужно пробежаться по стэку и найти try, catch блок, а это долго и не то чтобы сильно безопасно.
Более того, нам нарисовали не всю картину. Помимо сохранения адреса возврата. То в функции создаются автопеременные. При выходе из функции локальные объекты уничтожаются. Здесь тоже исключения замедляются (см. прата). Связи с этим могут быть различные не очень хорошие ситуации, к примеру, если мы создадим какой-нибудь объект. Даже простую стрингу: string …. То все будет хорошо, так как вызовется локальный объект, потом вызовется на него деструктор и будет все хорошо. Но иногда пробросила исключений не вызовет уничтожения памяти. Так что нужно все прописать руками: if все плохо, а потом освобождаем память.
Так что механизм исключений устроен весьма нетревиально. Эта реализация гигантская. И в реальности она является довольно монструозной с большим механизмом. Теперь немного про… мы подошли к теме, чем исключения в C++ не очень хороши.

1. Динамическая память.
2. Что вызвало исключение

Любая функция в блоке try, что может вызвать исключение, будет поймана и обработана в блоке catch. Ну тогда у меня будет блок try. В C и C++ довольно много функций может быть в одну строчку. Например: a = b + c. Здесь мы не поймем, вызвало исключение + или =. Так что это все надо продумывать.
Для упрощения нашей жизни разработчики сделали… они унаследованы от общего класса исключения.
Есть стандартный класс исключений: exception. std::exeption. От класса к классу это будет меняться. У этого класса есть минимальные требования к набору функций. А именно, виртулаьная: virtual what(); -> string. Все должно быть.
Мы хотим писать собственные исключения. И логично их наследовать от базового класса исключений:

class my_exceptions: public std::exception { … }

Типы ошибок:

logic_error
bad_alloc // когда мы вызываем оператор new, а у нас нет памяти. Выв. Эту ошибку.
runtime_error

Мы можем гарантированно заставить компилятор кидать или не кидать исключения. Можем специфицировать: *new(std::nothrow) int;* или, наоборот, заставить:

new (std::nowthrow) ing ;

bad_alloc — первый тип ошибок.

Второй тип ошибок для нас: logic_error. Это ошибки, который можно по-хорошему избежать. Но по неусмотрению разработчика они все же допускаются.

invalid_argument
domein_error
length_error
out_of_bounds //выход за пределы массива, например, выход за границы.

Третий тип: runtime_error

range_error
overflow_error
underflow_error // не относится к стюку, относится к представлениям.

2^(64) — очень много, человечество не произвело столько памяти.

Обработка исключений.
Самый интересный для нас, наверное, bad_alloc.

Два пути:
использовать стандарт
наследовать по паблику исключения (обязательно)

У catch есть довольно интересные нотации

try { … } catch (…) { std::cerr<<ex.what(); } exit(…); catch (…) — ловит все исключения, которые есть у нас.

В таком блоке мы можем не больше, не меньше: завершить нашу программу.

Предположим, что у нас есть некоторая иерархия исключений. Для этого достаточно перехватить указатель на exception. Хотелось бы получать нечто более конкретное, чем просто строка о том, что что-то произошло. Если у нас есть иерархия наследования. Можно задавать последовательность catch блоков, причем делать это надо в обратной последовательности.

try {…} catch (invalid_arguments) {…} catch (logic_error) {…} catch (exception &x) {…}

// принимается внутрь ссылка на копию объекта

Сначала система обработает все слева направо. Сначала проверит исполняемость invalid_arguments, затем мы проверяем, является ли это logic_error, а потом уже в общем по стандартным исключениям. Если не нашло ничего, то стэк раскрутится до конца и упадет в terminate. Это не хороший патерн, так как тогда не понятно, где проблема.

Принимаем ссылку на локальный объект. Как мы помним, локальные объекты перестают существовать за пределами функций. В любом случае в catch блок будет передана копия объекта. К примеру, у нас не хватило памяти и мы обрабатываем и логично позаботиться о том, чтобы там были move конструкторы с минимальными затратами памяти. Вот, опять, новые применения мув конструкторов.

Проблемы с работой с динамической памятью:

В исключениях мы столкнулись с тем, что программа бывает довольно не линейной. У нас есть гарантия, что между строками (заветы конструктивного программирования) мы сможем планировать выделение памяти и тд. У нас есть полный контроль над работой.

Для вызова мув конструктора: std::move();

В языке С очень много ошибок с работой с динамической памятью. Зато есть гибкость, возможность управлять ситуацией.

Мы перешли к умным указателям:
С динамической памятью здесь сложно. В С++98 добавили автоптр. Сейчас это считают устаревшим. В С++11 добавили новые штуки. Что интересного? Как использовать вообще? НАчинали мы с шаблонов функций. На самом деле шаблонизированными могут быть еще и классы.

auto_ptr — шаблонный класс. Нам нужно хранить указатель на объект. Соотв. автоптр. На стэк. Мы решили почему-то здесь динамически выделять. Внутри этого ст у нас указатель.

auto_ptr <Stack*> st0 = new Stack();

Для этого класса переопределны операторы ->, * .
При выходе из скопа у нас уничтожатся локальные объекты, вызовется дисктруктор для auto_ptr. Освободится и он, и наша выделенная память у rvalue. Какие минусы?

auto_ptr <Stack*> st1 = new St();

Память попытается освободиться два раза, что не хорошо. И здесь есть механизм «владения», позволяющий решить эту проблему. Но тогда лишь одна переменная освободиться (пример st1). А при обращении к st0 мы обратимся по нулевому указателю. И нельзя вызывать new[], так что работает только с new и delete.

Возникает вопрос, зачем мне смартпоинтеры с тем

Суть в том, что С++ долго не развивался, но он многим нравился. И энтузиасты создали библиотеку boost.

unique_ptr <stack*> st0 = new stack(); // st0 - владелец
unique_ptr <stack*> st1 = new st();

// здесь можно включить массив следующим образом:

unique_ptr <stack[]> st0 = new stack();

// проверяет ситуации с передачей управления. Он не даст:

unique_ptr <stack*> st1 = st0; // так как один владелец

Назначение, что здесь деструктор вызывает delete для того, на что указывает.

Есть также концепция раздельного управления.
У нас есть файл в сисетеме . И почему бы нескольким программам не работать с этим файлом. Когда количество ссылок = 0, то файл существовать не должен.

shared_ptr // объект перестанет существовать, когда все ссылки уйдут, организация
через static поле
~shared() {
if (ref_count = 0) { delete ptr; } else { - - ref count; }
}

Вывод:

У нас
Если пишем свое собственное исключение, то наследуемая от стандарта для общего исключения.
Проблемы исключений:

1. Производительность
2. Динамическая память
3. Определение источника проблем

Проблема (2) в целом проблема C++. В C++ нам добавили умные указатели. Которые могут помочь нам с этим. unique_ptr, shared_ptr и некоторые другие. Умные указатели решают проблему освобождения памяти.

KSS - Kaspersky Security System, встраивание в Linux, мобильные платформы, начало на windows.