МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

**Лабораторная работа №21**

по дисциплине: Основы программирования

тема: «Исключения»

Выполнил: ст. группы ПВ-223

Пахомов Владислав Андреевич

Проверили:

Притчин Иван Сергеевич

Черников Сергей Викторович

Код-ревьер: ст. группы ПВ-223

Голуцкий Георгий Юрьевич

Белгород 2023 г.

**Лабораторная работа № 21**

**Содержание отчёта:**

* Тема лабораторной работы.
* Цель лабораторной работы.
* Тексты заданий с набранными фрагментами кода к каждому из пунктов и ответами на вопросы по ходу выполнения работы.
* Работа над ошибками (код ревью)
* Вывод по работе.

**Тема лабораторной работы:** Исключения

**Цель лабораторной работы:** получение навыков работы с исключениями, осознание необходимости данных языковых средств.

Задания к лабораторной работе

В процессе запуска программ, взаимодействуя с ней, могут происходить различные нештатные ситуации: выход за пределы массива, некорректные аргументы, получаемые функциями и т. п.. Данные ситуации называются исключительными. Есть несколько способов, как вы можете реагировать на это:

1. Не реагировать. Умолчать об ошибке. Что будет, если мы умолчим о выходе за пределы массива? Это является неопределенным поведением и может произойти что угодно:
   1. Программа завершится с ошибкой
   2. Программа благополучно отработает и выдаст правильный результат.
   3. Программа отработает и выдаст какой-нибудь результат: число, строку, может изобрести искусственный интеллект, активировать SkyNet и уничтожить человечество (да, у нас опасная работа!).
2. Остановить работу программы во избежание рисков, описанных выше. Использовать exit.
3. Вывести сообщение об ошибке (если кому-то от этого легче)
4. Использовать специальные языковые средства, для обработки исключений

Ещё примеры исключительных ситуаций:

* деление на ноль
* нехватка оперативной памяти при использовании оператора new для её выделения (или другой функции)
* доступа к элементу массива за его пределами (ошибочный индекс)
* переполнение значения для некоторого типа
* взятие корня из отрицательного числа

В языке C++ **исключения** – это специальные объекты класса или значения базового типа, которые описывают (определяют) конкретную исключительную ситуацию и соответствующим образом обрабатываются.

* Рассмотрим всё же плохой сценарий обработки (вывод сообщения). Самостоятельно наберите пример ниже и опишите его поведение при b = 0 и b != 0

#include <iostream>  
  
int division(int a, int b) {  
 if (b != 0)  
 return a / b;  
 else  
 std::cerr << "b must not be equal to 0" << std::endl;  
}  
  
int main() {  
 int a, b;  
 std::cin >> a >> b;  
  
 auto res = division(a, b);  
 std::cout << res;  
  
 return 0;  
}

При b != 0 программа корректно выводит целочисленный результат деления a/b

При b = 0 программа выводит сообщение об ошибке b must not be equal to 0 и затем выводит 1, иногда в обратном порядке: сначала число, потом сообщение об ошибке

Какие недостатки вы можете выделить?

Программа всё равно пытается вывести результат деления, даже если оно невозможно.

* Каким образом функция main из прошлого примера поймёт, что что-то в функции division пошло не так? Да, вы можете сказать, что это было известно ещё до вызова функции. Однако заранее знать ситуации, при которых могут возникнуть исключительные ситуации не всегда возможно. Функция division хотела бы как-нибудь информировать main о том, что произошла проблема. Она могла бы делать это при помощи возвращаемого значения.

#include <iostream>  
  
bool division(int a, int b, int &res) {  
 if (!b) return false;  
  
 res = a / b;  
 return true;  
}  
  
int main() {  
 int a, b;  
 std::cin >> a >> b;  
  
 int res;  
 if (division(a, b, res))  
 std::cout << res;  
 else  
 std::cerr << "b must not be equal to 0" << std::endl;  
  
 return 0;  
}

Выделите недостатки.

Текущее решение довольно громоздкое, при вызове функции division логично получить результат деления, а не bool. Не совсем логичен с первого взгляда третий параметр функции division.

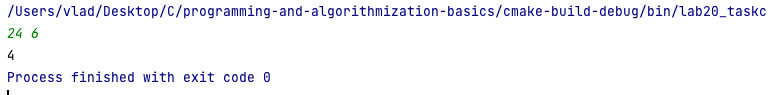
* Для сигнализирования исключительной ситуации в языке программирования C++ используется ключевое слово throw

#include <iostream>  
#include <string>  
  
int division(int a, int b) {  
 if (!b) throw std::string("Division by zero");  
  
 return a / b;  
}  
  
int main() {  
 int a, b;  
 std::cin >> a >> b;  
  
 auto res = division(a, b);  
 std::cout << res;  
  
 return 0;  
}

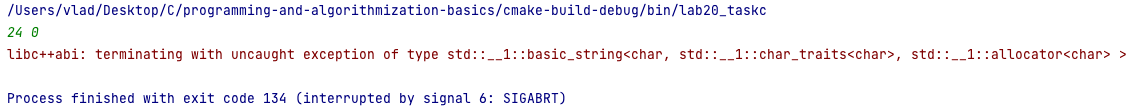
За которым следует исключение.

Проверьте работу функции main с данным примером. Результат приложите.

При b != 0 программа корректно выводит целочисленный результат деления a/b



При b = 0 программа выводит сообщение об ошибке и завершается с ошибкой



* throw прокидывает исключение вызывающей стороне. Считается, что она (вызывающая сторона) каким-то образом знает, как обработать данную исключительную ситуацию на основании полученного исключения. Для перехвата исключения используется конструкция вида:

try {  
 // тело блока try.  
 // В нём могут располагаться произвольные операторы  
}  
catch( тип1 идентификатор1 )  
{  
 // тело блока catch  
}  
catch( тип2 идентификатор2 )  
{  
 // тело блока catch  
}  
...  
catch( типN идентификаторN )  
{  
 // тело блока catch  
}

Если в блоке try генерируется соответствующая исключительная ситуация, то она перехватывается подходящим блоком catch. Выбор того или иного блока catch осуществляется в зависимости от типа исключительной ситуации. После возникновения исключительной ситуации определенного типа, вызывается блок catch с соответствующим типом аргумента.

Если в блоке try возникнет исключительная ситуация, которая не предусмотрена блоком catch, то вызывается стандартная функция terminate(), которая по умолчанию вызовет функцию abort(). Эта стандартная функция останавливает выполнение программы.

В блоке catch часто выполняются следующие действия:

* Выводится сообщение об ошибке в поток.
* Ошибка может быть прокинута дальше в вызывающий код.
* Осуществлён возврат значения для вызывающей функции.

Ознакомьтесь с примерами (наберите, опишите поведения):

#include <iostream>  
  
int division(int a, int b) {  
 if (!b) throw std::string("Division by zero");  
  
 return a / b;  
}  
  
int main() {  
 int a, b;  
 std::cin >> a >> b;  
  
 try {  
 auto res = division(a, b);  
 std::cout << res;  
 } catch (const std::string &s) {  
 std::cerr << s;  
 }  
  
 return 0;  
}

При b != 0 программа корректно выводит целочисленный результат деления a/b

При b = 0 программа выводит сообщение об ошибке и завершается без ошибки.

На самом деле, вы можете использовать более короткий пример, чтобы убедиться в том, что исключения перехватываются блоком catch (наберите, опишите поведение):

#include <iostream>  
  
int main() {  
 try {  
 throw 42;  
   
 std::cout << "Success";  
 } catch (int a) {  
 std::cerr << a;  
 }  
   
 return 0;  
}

В консоль выводится число 42, программа завершается без ошибки.

Механизм перехвата исключения следующий:

* Генерируется исключение
* Точка выполнения немедленно переходит к ближайшему блоку try внутри которого располагался код, в результате которого произошла исключительная ситуация.
* Затем проверяются обработчики catch на соответствие типу исключения. Если обработчик найден, исключение обрабатывается, иначе передаётся дальше (функции, которая вызвала эту функцию, вдруг е неё есть соответствующий обработчик catch).
* На основании описания поведения абзацем выше объясните, каким образом работает данный фрагмент

#include <iostream>  
  
void f1() {  
 throw std::string("ERROR!!!");  
}  
  
void f2() {  
 try {  
 f1();  
 } catch (int a) {  
 std::cerr << "CATCH IN F2";  
 std::cerr << "(" << a << ")";  
 }  
}  
  
int main() {  
 try {  
 f2();  
 } catch (std::string &s) {  
 std::cerr << "CATCH IN MAIN";  
 std::cerr << "(" << s << ")";  
 }  
  
 return 0;  
}

1. При выполнении f1 генерируется исключение
2. Точка выполнения переходит к try внутри которого было сгенерировано это исключение (9 линия)
3. Проверяются обработчики catch, среди них нет обработчика, обрабатывающего std::string, поэтому исключение прокидывается дальше в main
4. Точка выполнения переходит к try на 18 линии.
5. Проверяются обработчики catch, среди них есть обработчик, удовлетворяющий типу ошибки std::string, поэтому будут выполнены команды на 20-21 линиях.
6. main продолжит своё выполнение, программа успешно завершится, и жили они долго и счастливо.

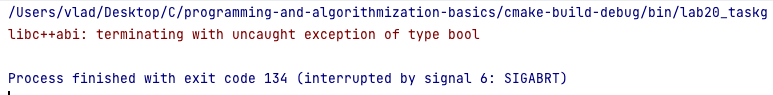
* Напишите фрагмент кода, который выбирает необходимый catch в зависимости от типа исключения (конструкцию с несколькими блоками catch).

#include <iostream>  
  
int main() {  
 try {  
 int errorType;  
 std::cin >> errorType;  
  
 switch (errorType) {  
 case 0:  
 throw 42;  
 case 1:  
 throw 42ULL;  
 case 2:  
 throw std::string("42");  
 default:  
 throw std::vector<int>(42, 42);  
 }  
 } catch (int &error) {  
 std::cerr << "You've been int excepted!" << std::endl;  
 } catch (unsigned long long &error) {  
 std::cerr << "You've been unsigned long long excepted!" << std::endl;  
 } catch (std::string &error) {  
 std::cerr << "You've been string excepted!" << std::endl;  
 } catch (std::vector<int> &error) {  
 std::cerr << "You've been vector excepted!" << std::endl;  
 }  
  
 return 0;  
}

* Напишите фрагмент кода в котором выкидывается исключение, но никак не обрабатывается. Опишите наблюдаемое поведение.

#include <iostream>  
  
int main() {  
 try {  
 throw false;  
 } catch (int &error) {  
 std::cerr << "You've been int excepted!" << std::endl;  
 } catch (unsigned long long &error) {  
 std::cerr << "You've been unsigned long long excepted!" << std::endl;  
 } catch (std::string &error) {  
 std::cerr << "You've been string excepted!" << std::endl;  
 } catch (std::vector<int> &error) {  
 std::cerr << "You've been vector excepted!" << std::endl;  
 }  
  
 return 0;  
}

В консоль выводится сообщение об ошибке, программа завершается с ошибкой



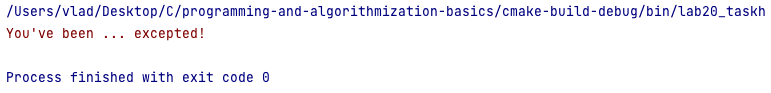
* Можно перехватывать исключения любого типа используя такой синтаксис:

try {  
 // блок кода с потенциальной ошибкой  
} catch (...) {

}

Сгенерируйте какое-нибудь исключение и проверьте, что решение через ... работает.

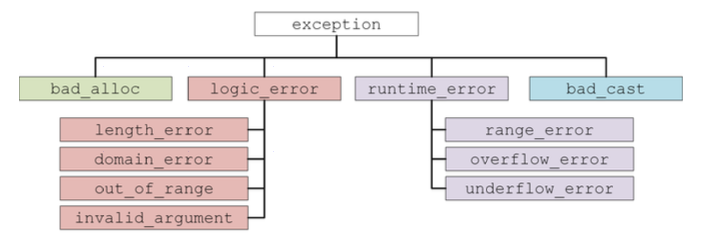
#include <iostream>  
  
int main() {  
 try {  
 throw false;  
 } catch (int &error) {  
 std::cerr << "You've been int excepted!" << std::endl;  
 } catch (unsigned long long &error) {  
 std::cerr << "You've been unsigned long long excepted!" << std::endl;  
 } catch (std::string &error) {  
 std::cerr << "You've been string excepted!" << std::endl;  
 } catch (std::vector<int> &error) {  
 std::cerr << "You've been vector excepted!" << std::endl;  
 } catch (...) {  
 std::cerr << "You've been ... excepted!" << std::endl;  
 }  
  
 return 0;  
}



* Оператор throw может и не содержать параметров. Он используется тогда, когда возникает необходимость прокинуть исключение дальше. В примере ниже функция f() с какой-то вероятностью не может выделить память. Предположим, если память не смогла быть выделена, наша программа должна освободить память под все ранее выделенные объекты, и при этом main должен узнать, что произошла ошибка:

#include <iostream>  
#include <vector>  
#include <ctime>  
  
#define **SUCCESS\_PROB** 0.95  
  
int\* getmem(float successProb) {  
 float r = static\_cast<float>(rand()) / static\_cast<float>(**RAND\_MAX**);  
  
 if (r > successProb)  
 throw std::bad\_alloc();  
  
 return new int[10];  
}  
  
std::vector<int\*> getParts(int amountParts, float successProb) {  
 std::vector<int\*> parts;  
  
 for (int i = 0; i < amountParts; i++) {  
 try {  
 parts.push\_back(getmem(successProb));  
 } catch (const std::bad\_alloc &e) {  
 std::clog << "free dynamic memory\n";  
  
 for (auto &part : parts)  
 delete[] part;  
  
 throw;  
 }  
 }  
  
 return parts;  
}  
  
int main() {  
 srand(time(**nullptr**));  
  
 try {  
 auto parts = getParts(10, **SUCCESS\_PROB**);  
 std::cout << "Success!";  
 } catch (const std::exception &e) {  
 std::cerr << e.what();  
 }  
  
 return 0;  
}

* Как выше было замечено, перехватывалась переменная класса exception. Все стандартные исключения наследуются от него



Из этой схемы следует, что если вы выкинете исключение out\_of\_range, то оно может быть перехвачено в блоке catch переменными типа logic\_error и exception

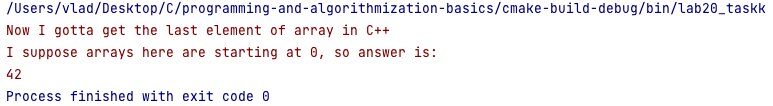
#include <iostream>  
  
int main() {  
 try {  
 throw std::logic\_error("<Error description>");  
 } catch (const std::exception &e) {  
 std::cerr << e.what();  
 }  
  
 return 0;  
}

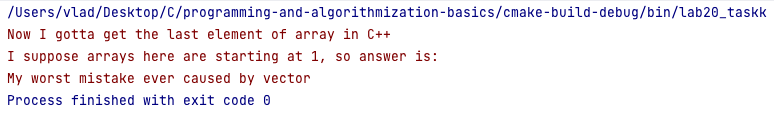
Перехватите исключение out\_of\_range при помощи logic\_error. Чтобы исключение сгенерировалось более осмысленно, создайте вектор размера n, и обратитесь к элементу вектора при помощи метода .at(), допуская выход за пределы массива. Напишите код таким образом, чтобы при каких-то значениях исходных данных исключение выкидывалось, а при каких-то – нет.

Приложите написанный код, результат работы программы, когда исключение было и не было возбуждено.

В задании к пункту опишите, когда именно выбрасываются исключения классов, представленных на схеме.

#include <vector>  
#include <iostream>  
#include <ctime>  
  
int main() {  
 srand(time(**nullptr**));  
 rand();  
  
 try {  
 size\_t n = 42;  
 std::vector<int> arr(n, 42);  
  
 std::clog << "Now I gotta get the last element of array in C++\n";  
  
 float r = static\_cast<float>(rand()) / static\_cast<float>(**RAND\_MAX**);  
  
 if (r > 0.5) {  
 std::clog << "I suppose arrays here are starting at 0, so answer is: \n";  
 std::clog << arr.at(n - 1);  
 } else {  
 std::clog << "I suppose arrays here are starting at 1, so answer is: \n";  
 std::clog << arr.at(n);  
 }  
 } catch (std::logic\_error &e) {  
 std::cerr << "My worst mistake ever caused by " << e.what();  
 }  
  
 return 0;  
}





* bad\_alloc – выбрасывается при невозможности выделить память
* logic\_error – базовое исключение, которое служит для описания ошибок, которые могут быть обнаружены до выполнения программы, таких как нарушение логических предварительных условий
  + length\_error – исключение выбрасываемое при попытке создать слишком длинный объект
  + domain\_error - исключение выбрасываемое при ошибках домена (области определения)
  + out\_of\_range - исключение выбрасываемое при условии что аргумент выходит за пределы допустимого диапазона
  + invalid\_argument - исключение выбрасываемое при невалидном аргументе
* runtime\_error - базовое исключение, которое служит для описания ошибок, которые не могут быть обнаружены до выполнения программы
  + range\_error - исключение выбрасываемое при ошибках диапазона
  + overflow\_error - исключение выбрасываемое при переполнении
  + underflow\_error - исключение выбрасываемое при невозможности выразить число ввиду недостаточной точности на которую способен процессор, память
* bad\_cast - исключение выбрасываемое при невозможности привести значение одного типа к другому типу (яблоки привести к кирпичу)
* Пусть есть некоторая функция, которая возвращает фрагмент свободной динамической памяти:

int\* getmem(size\_t partSize) {  
 return new int[partSize];  
}

Напишите функцию

vector<int\*> getParts(int nParts, size\_t partSize)

которая по окончании своей работы должна вернуть вектор указателей на выде- ленные фрагменты памяти. Если память выделить при помощи new не удалось, будет выброшен bad\_alloc, который должен быть перехвачен в main.

#include <iostream>  
#include <vector>  
#include <ctime>  
  
int\* getmem(size\_t partSize) {  
 return new int[partSize];  
}

std::vector<int\*> getParts(int amountParts, size\_t partSize) {  
 std::vector<int\*> parts;  
  
 for (int i = 0; i < amountParts; i++) {  
 try {  
 parts.push\_back(getmem(partSize));  
 } catch (const std::bad\_alloc &e) {  
 std::clog << "free dynamic memory\n";  
  
 for (auto &part : parts)  
 delete[] part;  
  
 throw;  
 }  
 }  
  
 return parts;  
}  
  
int main() {  
 srand(time(**nullptr**));  
  
 try {  
 auto parts = getParts(10, 100000000000000000);  
 std::cout << "Success!";  
 } catch (const std::exception &e) {  
 std::cerr << e.what();  
 }  
  
 return 0;  
}

**Вывод:** в ходе лабораторной работы получили навыки работы с исключениями, осознали необходимость данных языковых средств.