Частное учреждение образования
«Колледж бизнеса и права»

УТВЕРЖДАЮ
Заведующий методическим кабинетом
_____Е.В.Фалей
«____»_____2017

Специальность: 2-40 01 01 «Программное обеспечение информационных технологий»	Дисциплина: «Основы алгоритмизации и программирование»

Лабораторная работа № 30
Инструкционно-технологическая карта

Тема: Разработка алгоритмов и программ по обработке исключительных ситуаций
Цель: Научиться создавать алгоритмы и программы с использованием обработки исключительных ситуаций
Время выполнения: 2 часа

Задания по написанию программ выделены зеленым фоном

1. Краткие теоретические сведения

Исключения в C++.

Ошибка (error) – это ошибка, которая возникает в процессе работы программы (запущенного на исполнение *.exe-файла). Ошибка свидетельствует о том, что, хотя код программы и не содержит синтаксических ошибок (ведь *.exe-файл скомпилировался), но выполнение кода программы привело к возникновению аварийной ситуации (например, произошло обращение к данным, которых нет в предполагаемом месте оперативной памяти, обратились к файлу, которого не оказалось на компьютере по указанному в программе пути, переполнение стека, попытка создать динамический массив отрицательной или очень большой размерности, которую не может обеспечить наличная свободная оперативная память и так далее). Ошибка возникает в ходе выполнения программы, код которой написан формально синтаксически правильно, но не учитывает случай возникновения такой ошибочной ситуации, которая может возникать всегда или только иногда, в зависимости от каких-то условий. Код программ, которые допускают ошибки в процессе своей работы и вследствие чего аварийно завершаются, неграмотный. Аварийное завершение программы обеспечивается операционной системой, которая всегда удаляет (убивает) процесс программы, в которой произошла ошибка, из оперативной памяти. Операционная система под каждую запускаемую на исполнение программу создает процесс (process), внутри которого будет как минимум один или несколько потоков (thread). Именно потоки ПРОЦЕССов (исполняемых программ) исполняются ПРОЦЕССором. Операционная система уничтожает процесс, в котором произошла ошибка, поскольку операционная система не может исправить код откомпилированного файла и стремится уничтожить процесс, в котором возникла ошибка, чтобы минимизировать ущерб от процесса, в котором произошла ошибка.

Идеально исключить вероятность возникновения ошибок в ходе работы программы на 100% нельзя, но нужно отлавливать и обрабатывать вероятные ошибки внутри программы, делая их исключительными ситуациями (exception'ами) чтобы операционная система не уничтожала вашу исполняемую программу и пользователь был доволен стабильной работой вашей программы.

Исключительная ситуация (исключение, exception) – это ошибка (error), возникшая в процессе работы программы и **обработанная в коде** программистом, то есть это ошибка, отловленная и обработанная специальным кодом, написанным программистом.

Примеры ошибок: выход за пределы выделенной области памяти, ошибка открытия файла, попытка инициализации объекта недопустимым значением, обращение к элементу массива с несуществующим индексом.

В языке Си функция возвращает значение. Если в функции произошла ошибка, то функция возвращает сигнализирующее об этом значение, которое отлавливается в main'е. Например:

```
//… код
if(fopen(…) == NULL) // if(fopen(…) == ERROR_RETURN_VALUE) // if(fopen(…) == bad_alloc)
{
	//код обработки ошибки
}
else
{
	// код работы с успешно открытым файлом
}
```

Недостатки такого подхода: код программы содержит больше вложенных участков кода; в программах возможны неявные вызовы функций другими функциями, и такие вызовы не отследятся.

Язык C++ предлагает развитие данного подхода и устранение вышеотмеченных недостатков с помощью блока кода try-catch-finally. Для использования такого блока в коде программы надо написать:

```
//код, возможное возникновение ошибок в котором не отслеживается
try
{//начало области кода, в которой отслеживаются ошибки
	//область кода, в которой будут отслеживаться возникновения ошибок
	if(…)
	{
		throw имяОшибки;//можно сгенерировать ошибку самому, то есть
		//искусственно вызвать возникновение ошибки – «пробросить» ошибку
	}
}//конец области кода, в которой отслеживаются ошибки
catch(типОшибки1 аргумент1)//обработчик ошибки вида типОшибки1
{//один или несколько блоков catch(){} должны сразу находиться за блоком try{…}
	//код, который выполнится в случае возникновения ошибки1 в блоке try{…}
```

```
}
catch(типОшибки2 аргумент2)//обработчик ошибки вида типОшибки2
{//один или несколько блоков catch(){} должны сразу находиться за блоком try{…}
	//код, который выполнится в случае возникновения ошибки2 в блоке try{…}
}
catch(типОшибки3 аргумент3)//обработчик ошибки вида типОшибки3
{//один или несколько блоков catch(){} должны сразу находиться за блоком try{…}
	//код, который выполнится в случае возникновения ошибки3 в блоке try{…}
}
catch(типОшибки4 аргумент4)//обработчик ошибки вида типОшибки
{//один или несколько блоков catch(){} должны сразу находиться за блоком try{…}
	//код, который выполнится в случае возникновения ошибки в блоке try{…}
}
catch(…)//”универсальный” обработчик всех сгенерированных типов ошибок пишется
{//с тремя точками в круглых скобках, он отловит все ошибки, поэтому он должен быть
	//записан ПОСЛЕДНИМ из всех отлавливателей catch()
	//код, который выполнится в случае возникновения ошибки в блоке try{…}
}
finally//необязательный участок кода (его можно не писать, если в нем нет потребности)
{
	//код внутри блока finally{…} выполняется в любом случае, вне зависимости от
	//возникновения ошибки (тогда блок finally{…} сработает вслед за сработавшим
	// catch’ем) или отсутствия ошибок (тогда блок finally{…} сработает вслед за
	//выполненным блоком try{…}, а ни один из catch’ей не сработает, поскольку
	//ошибок не возникло. В блок finally{…} помещают код, который должен
	//выполниться в любом случае, например, закрытие читаемых или
	//записываемых файлов, закрытие подключения к базе данных, серверу и т.д.
}
```

С помощью throw можно принудительно вызвать ошибку некоторого типа. Это может понадобиться для тестирования, но обычно генерирование ошибки зависит от некоторых условий, поэтому throw пишется внутри проверки, например, if(условие){…}, чтобы ошибка создавалась только при выполнении каких-либо условий. В программах обычно отлавливают общеизвестные типы ошибок, которые уже прописаны в библиотеках, и мы сначала подключаем библиотеки с типами ошибок и в catch’ах отлавливаем ошибки тех типов, возникновение которые мы ожидаем в блоке try{…} при некотором сценарии выполнения нашей программы на компьютере пользователем. Всегда сначала записывается самый узкоспециализированный отлавливатель ошибки catch(), за ним менее узкоспециализированный отлавливатель ошибки, за ним более универсальный отлавливатель и так далее до последнего самого универсального отлавливателя ошибки, который всегда должен писаться последним, поскольку, располагаясь перед узкоспециализированным отлавливателем, универсальный отлавливатель раньше узкоспециализированного получает входной аргумент (ошибку) и срабатывает, ведь

ему подходят ошибки любых типов. Таким образом, расположенный далее узкоспециализированный отлавливатель не сможет сработать никогда. Ошибка деления на ноль – пример конкретного типа ошибки (узкоспециализированная ошибка), арифметическая ошибка – более универсальная ошибка, а просто ошибка – самый универсальный тип ошибки как таковой. Нужно отлавливать ошибки разных типов, чтобы сообщить другому коду и пользователю, что именно за ошибочная ситуация произошла (какой именно у ошибки тип), поскольку дальнейшие действия программы в случае возникновения разных ошибок могут быть разными, специализированными (это грамотный подход в программировании). Если вы сообщите пользователю, что по указанному им пути файл не обнаружен, это вполне может устроить пользователя и подсказать ему, что делать, чтобы исправить ситуацию и помочь вашей программе отработать правильно следующий раз; но если вы просто сообщите пользователю, что произошла ошибка, то это пусть и лучше, чем аварийное завершение программы, но оставляет пользователя в неведении, что же сделать, чтобы исправить ситуацию, поскольку он не знает деталей об ошибке и даже при следующем запуске программы ситуация может повториться, но программа снова не сообщит конкретной информации об ошибке. Универсальный улавливатель ошибки располагают последним в качестве «последнего рубежа» на всякий случай: если мы не учли все специализированные типы ошибок, которые реально возникнут, то в случае возникновения ошибки НЕотлавливаемого нами типа последним примет эту ошибку универсальный отлавливатель и, поскольку он срабатывает на ошибку любого типа, то он предотвратит аварийное завершение программы операционной системой и хотя бы сообщит другому коду и пользователю (если вы это укажете в его теле), что произошла ошибка.

Если в блоке try{…} происходит первая ошибка, то в этом месте выполнение кода прекращается и управление сразу передается на первый следующий блок catch(), который сравнивает тип возникшей ошибки с типом ошибок, которые он должен отлавливать, и если есть совпадение типов, то срабатывает его тело и следующие за ним catch’и уже не получат ошибку и сработать не смогут в принципе. Если далее есть блок finally{…}, то он сработает последним обязательно и в случае возникновения ошибки, и при безаварийной работе программы; catch’и же срабатывают, только если произошла ошибка отлавливаемого ими типа. После срабатывания первого catch’а следующие за ним catch’и не сработают в принципе (до них просто не дойдет управление), но если первый catch() принял ошибку и она не подходит ему по типу, то эту ошибку получает следующий за ним catch(), который будет ее сравнивать с типом «своей» ошибки и сработает при совпадении, если совпадения не будет – ошибка передается следующему catch()’у и так далее, пока или очередной catch() не распознает эту ошибку как подходящую ему по типу, или ошибка по цепочке передастся до последнего универсального catch()'а, а ему подойдет ошибка любого типа и он сработает на нее в любом случае (поэтому узкоспециализированные catch()’и должны располагаться перед ним, причем между собой они должны идти начиная от самого конкретного и далее по нарастанию их «универсальности» вплоть до самого универсального catch()’а в конце на всякий случай.

После срабатывания первого catch()'а все расположенные за ним catch()'и уже не сработают. Блок кода try – catch напоминает по своему действию блок switch – case, но имеет вышеперечисленные особенности и не требует break'ов в конце каждого catch()'а, поскольку в конце каждого catch()'а break неявно подставляется компилятором по умолчанию. Если блок finally{…} написан, то он сработает последним и в случае возникновения ошибки, и без них. Отлавливатели catch()'и срабатывают только в случае возникновения ошибки в блоке try{…} перед ними. Если ошибка не произойдет, то никакие catch()'и не сработают и программа отработает обычным образом. Если ошибка в ходе выполнения вашей программы возникнет, но будет обработана catch()'ем в вашей программе, то это уже не ошибка, а исключительная ситуация (обработанная ошибка) и ваша программа считается правильной, по крайней мере, в данном тестовом запуске. Из catch()'ей сработает только один – первый catch(), которому подойдет возникшая ошибка по типу.

Обычно код тела функции main() помещают в блок try{…}, но можно помещать отдельные участки кода в отдельные try'и, можно писать блоки try{…} в отдельных пользовательских функциях, методах, свойствах, конструкторах, деструкторах. Например, в лабораторных примерах WindowsForms программ в блоки try – catch брались тела методов, где были возможны ошибки деления на ноль, неверного формата данных и так далее, причем тот код представляет из себя примеры прописывания блоков try – catch в языке Visual C++ (VC++) для платформы .NET, разработанной в компании Microsoft.

В обработчиках исключительных ситуаций catch() может использоваться код завершения программы, например:

```
return 0;
exit(0);
terminate();//вызовет функцию abort();
abort();//завершает работу программы (прекращает работу программы)
```

Исключительная ситуация может иметь любой тип, в том числе она может быть объектом класса ошибок, определенного в библиотеке (подключенном файле) или объектом класса, определенного программистом (класса, написанного вами; ваш класс выгодно унаследовать от готового класса Exception). Сам класс exception определен в файле <vcruntime_exception.h>, но обычно вы его подключаете косвенно через другие файлы, например, <stdexcept>.

Для работы с готовыми исключительными ситуациями (готовыми классами типов ошибок) надо подключить библиотеки и пространство имен std (для консольных приложений):

```
#include <stdexcept>//содержит определения ошибок std::logic_error,
//std::runtime_error и всех ошибок, от них унаследованных
#include <exception>//содержит определения ошибок std::bad_exception, std::bad_alloc
#include <new>//#include <new.h>//заголовочный файл для выделения динамической
//области оперативной памяти под динамические объекты
#include <typeinfo>//содержит определения ошибок std::bad_cast и std::bad_typeid
using namespace std;
```

В этих библиотеках есть готовые классы вероятных ошибок, то есть типы ошибок, которые обычно возникают:

std::range_error	Исключение, возникающее в случае, когда полученный вещественный результат превосходит допустимый диапазон, то есть полученное вещественное число либо слишком маленькое, либо слишком большое для вещественного типа данных. При таком исключении мы не узнаем конкретно, положительный результат или отрицательный, но узнаем, что он вышел за допустимый диапазон для вещественного типа данных
std::overflow_error	Исключение, возникающее в случае, когда полученный результат выходит за верхнюю границу диапазона допустимых значений (недопустимо большое положительное значение)
std::underflow_error	Исключение, возникающее в случае, когда полученный результат выходит за нижнюю границу диапазона допустимых значений (недопустимо большое отрицательное значение)
std::runtime_error	Исключение, возникающее во время выполнения программы. Подвиды ошибки std::runtime_error: std::range_error std::overflow_error std::underflow_error
std::out_of_range	Исключительная ситуация, возникающая при попытке доступа к элементам вне допустимого диапазона (например, выход за границы массива)
std::length_error	Исключительная ситуация, возникающая при попытке создать объект большего размера (большей длины в байтах), чем допустимый размер (длина) для объекта данного типа
std::invalid_argument	Исключительная ситуация, возникающая при попытке передачи в функцию неподходящего по типу входного аргумента (несоответствующего сигнатуре функции)
std::domain_error	Исключительная ситуация, возникающая при попытке передачи в функцию непроинициализированного входного аргумента (в функцию передается переменная, не содержащая в себе никакого значения)
std::logic_error	Исключительная ситуация, возникающая при наличии логической ошибки в коде программы. Подвиды ошибки std::logic_error: std::out_of_range std::length_error std::invalid_argument std::domain_error
std::bad_alloc	Исключительная ситуация, возникающая при невозможности выделить динамическую область оперативной памяти для создания и хранения динамического объекта. Сначала нужно

	подключить файлы: #include <new> #include <exception> , поскольку данный тип ошибки определен в них
std::bad_cast	Исключительная ситуация, возникающая при ошибочном приведении типа объекта к неприемлемому для этого объекта типу данных. Сначала нужно подключить файл: #include <typeinfo> , поскольку данный тип ошибки определен в нем
std::bad_typeid	Исключительная ситуация, возникающая при неудачной попытке определения типа некого объекта. Сначала нужно подключить файл: #include <typeinfo> , поскольку данный тип ошибки определен в нем
std::bad_exception	Исключительная ситуация, возникающая от иных причин, например, когда нарушается динамическая спецификация отлавливаемых исключительных ситуаций

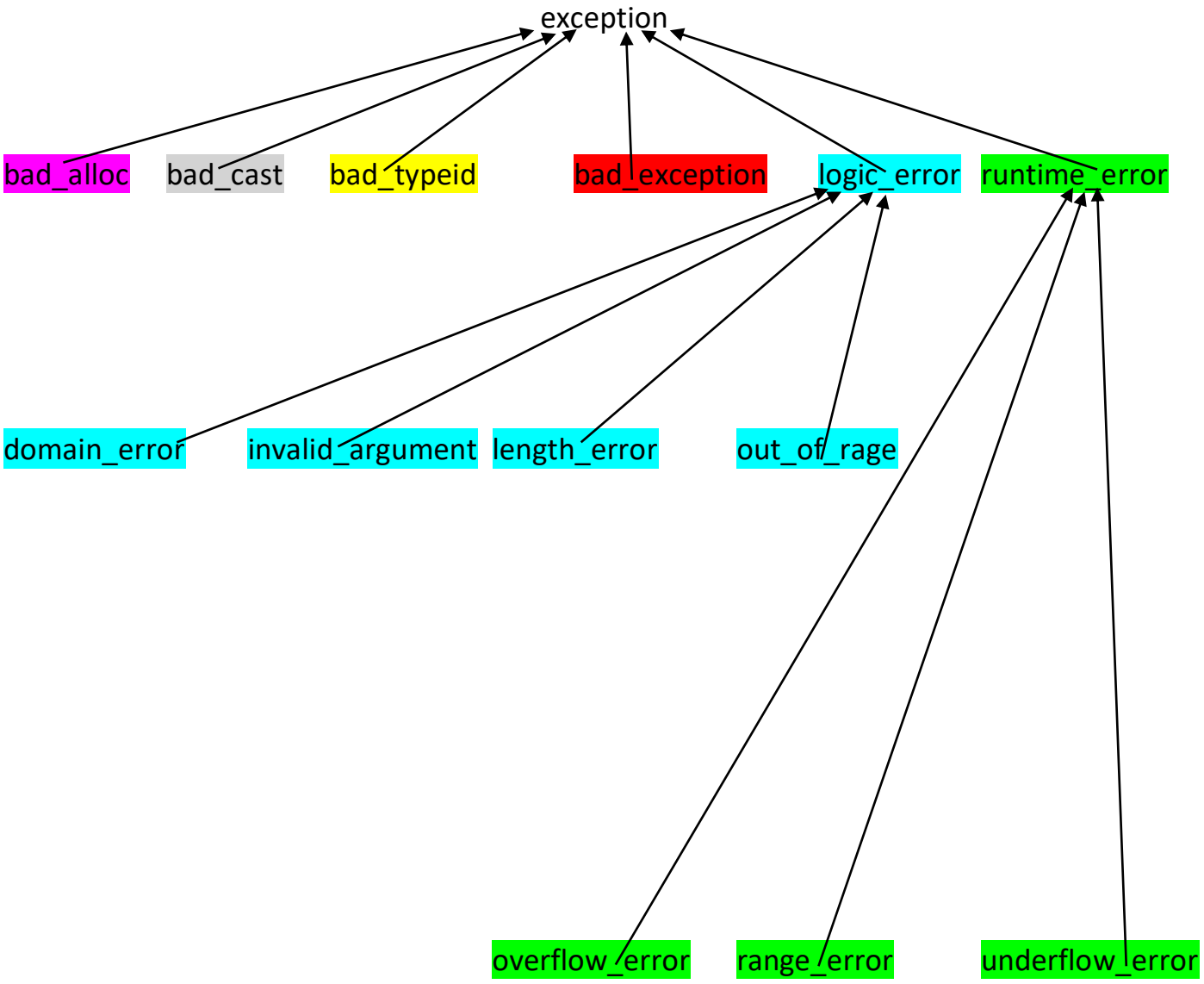


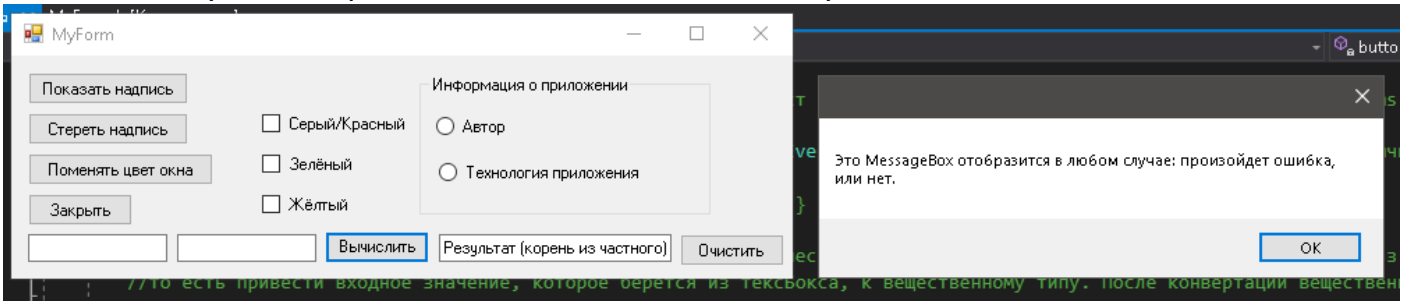
Схема 1 – Диаграмма иерархии классов наследования от класса exception

Ниже примеры использования обработчиков исключительных ситуаций в Visual C++ в Windows Forms. При наборе кода, если навести курсор мыши на имена создаваемых объектов, операторы и выражения, во всплывающей подсказке вам среда разработки указывает те типы ошибок, которые вероятны в этом выражении, чтобы вы написали обработчики этих вероятных ошибок. Также обратите внимание, если вероятных ошибок несколько типов, то среда разработки показывает вам их типы в порядке от самого узкоспециализированного до более общего, а если вы наберете их в другом порядке, поставив более универсальный обработчик ошибки перед более специализированным, то среда разработки подчеркнет такой код как ошибочный (последнее верно и для консольных программ). В Visual C++ в Windows Forms вам доступен и try – catch, и try – catch – finally. Блок кода finally{…} используется для кода, который должен выполниться безусловно в любом случае вне зависимости от того, произойдут ошибки или нет. Код в теле finally{} выполняется последним, после корректно отработавшего кода в блоке try{} или после сработавшего обработчика ошибки catch().

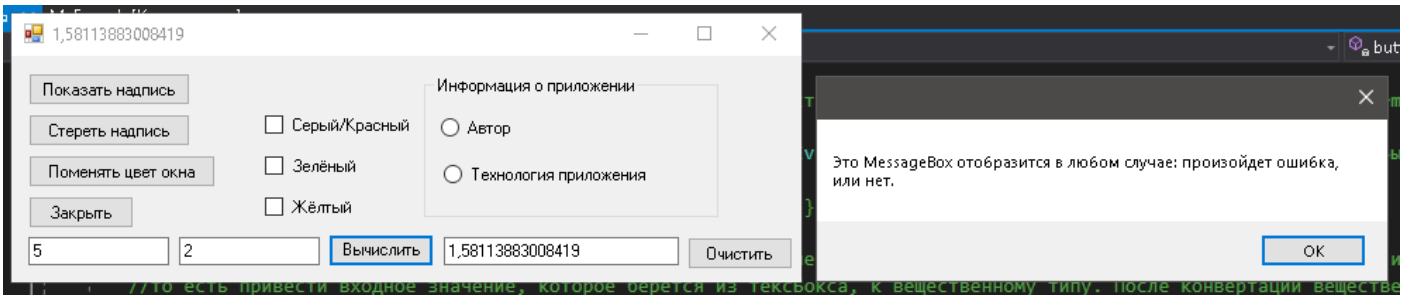
```
private: System::Void button5_Click(System::Object^ sender, System::EventArgs^ e)//обработчик события нажатия на кнопку "Вычислить"
{
    try//это блок try{}-catch(){}, отлавливающий в коде блока try{...} ошибки типов, указанные в блоке catch()
    {
        double a = Convert::ToDouble(this->textBox1->Text);//у статического класса Convert(т.е.Конвертация, перевод даных из одного типа данных в другой) вызываем метод ToDouble(),
        //то есть привести входное значение, которое берется из тексБокса, к вещественному типу. После конвертации вещественный результат справа помещаем в вещественную переменную,
        double b = Convert::ToDouble(this->textBox2->Text);//созданную слева
        if (b == 0 || a / b < 0)//если делитель равен нулю или хотя бы результат деления меньше нуля (значит, из него НЕЛЬЗЯ извлечь квадратный корень)
        {//то выводим мини-окно Сообщения с заголовком "Ошибка!" и текстом "Попытка деления на ноль или извлечения корня из отрицательного числа."
            MessageBox::Show("Попытка деления на ноль или извлечения корня из отрицательного числа.", "Ошибка!");
            button6_Click(sender, System::EventArgs::Empty);//после закрытия окна Сообщения, очищаем поля от старых ошибочных данных, вызвав обработчик кнопки "Очистить"
        }//обработчик события должен принимать имя отправителя, создающего событие, и передаваемые данные (тут передаем ничего - Empty)
        else//если проверку на математическую корректность данные прошли успешно
        {//то можно разделить делимое на делитель и извлечь из этого результата квадратный корень, который берется из статического класса Math, помещая итоговый результат в
            double c = Math::Sqrt(a / b);//вещественную переменную double c
            this->textBox3->Clear();//очищаем поле текстБокса3 от предыдущего любого текста
            this->textBox3->Text = c.ToString();//вещественное число приводим к строковому виду (символьный массив) и помещаем в поле текстБокса3
            this->Text = c.ToString();//вещественное число приводим к строковому виду (символьный массив) и помещаем в качестве заголовка основного окна
        }
    }//конец блока try - за ним сразу должен следовать блок кода catch(){...}, отлавливающий ошибки конкретного типа
    catch (System::FormatException^ e)//отлавливать ошибки типа System::FormatException, то есть ошибки, связанные с неподходящим форматом (типом) данных
    {//если возникла ошибка неподходящего форматирования данных (данные не соотвествуют ожидаемому от них типу данных), то
        MessageBox::Show(e->Message->ToString(), "Ошибка формата данных!");//вывести мини-окно Сообщения с заголовком "Ошибка формата данных!" и текстом сути ошибки форматирования
        button6_Click(sender, System::EventArgs::Empty);//после закрытия окна Сообщения очистить поля от старых ошибочных данных, вызвав обработчик кнопки "Очистить"
    }//конец тела блока catch(){...}
    finally//блок кода finally{...} выполнится в случае нажатия на кнопку button5 вне зависимости от корректной работы кода или генерации ошибки. Блок кода finally
    {//сработает в конце обработчика button5_Click(). Блок finally можно писать за последним из catch'ей или, если он не нужен, не писать его совсем
        MessageBox::Show("Это MessageBox отобразится в любом случае: произойдет ошибка, или нет.\n");
    }//конец тела блока кода finally
}//конец тела обработчика события клика по кнопке "Вычислить"
```

Ниже тестирование работы блока кода finally{}, который срабатывает в конце работы кода обработчика нажатия кнопки button5 «Вычислить». Конечно, в нашем примере блок кода finally{} сработает только если нажать на кнопку «Вычислить», но в таком случае уже обязательно безусловно сработает.

Поля пустые – сработала исключительная ситуация:



Данные корректны и результат отобразился без ошибок, но finally{} все равно в конце сработал:



Еще пример обработчика нажатия на кнопку с внутренней обработкой исключительных ситуаций:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)//обработчик события нажатия на кнопку button1
{
    double a;
    try//в блоке кода try{} отслеживаем возможные ошибки трех типов
    {
        a = Double::Parse(this->textBox1->Text);//статический класс Double содержит метод Parse(), который принимает значение из текстБокса1, конвертирует его в вещественной
        //значение и возвращает результат, который мы помещаем в вещественную переменную a. Если конвертируемое значение не походит на вещественное число, например, это слова,
        //буквы, нечисловые символы, отсутствие любых символов, то произойдет ошибка конвертирования. Разделителем целой и дробной частей должна выступать ЗАПЯТАЯ в нашем языке
        SecondForm^ f = gcnew SecondForm(a, this);//создать новую оконную форму типа класса SecondForm конструктором с вещественным параметром, куда передаем значение из
        //переменной a. Имя оконной формы ^f - это указатель на оконную форму
        f->ShowDialog();
        //f->Show();//отобразить НЕмодальное окно f на экране ПК, при этом главное окно остается активным и доступно для работы с пользователем
        //MyForm::Hide();//метод визуально скрывает главную оконную форму от пользователя на мониторе ПК, но в оперативной памяти она остается, пока не будет закрыто и
        //выгружено из ОП все приложение кодом Application::Exit();//метод закрытия всего приложения
        //this->Enabled = false;//это выражение делает главное окно приложение НЕреагирующим на дествия мыши пользователя. появившееся вспомогательное окно при этом доступно
        //для действий мыши пользователя, ведь его свойство по умолчанию this->Enabled = true;
    }
    catch (System::ArgumentNullException^ e)//отлавливаем недопустимую пустую (непроинициализированную) ссылку
    {//сообщаем пользователю информацию в мини-окне MessageBox'а, который является модальным окном (пока он существует, основное окно заблокировано для действий пользователя)
        MessageBox::Show(this, "Пустая ссылка.", "Внимание", MessageBoxButtons::OK, MessageBoxIcon::Error);//окну сообщения можно присваивать заголовок, текст, назначать
    }// кнопки, иконку
    catch (System::FormatException^ e)//отлавливаем ошибку некорректного формата данных
    {
        MessageBox::Show(this, "Введите вещественное число.", "Внимание", MessageBoxButtons::OK, MessageBoxIcon::Warning);
    }
    catch (System::OverflowException^ e)//отлавливаем ошибку переполнения значения переменной некоторого типа данных. Есть ли разница, в каком порядке располагать отлавливатели
    {// ошибок?
        MessageBox::Show(this, "ВВедите число подходящего размера.", "Слишком большое или маленькое число", MessageBoxButtons::OK, MessageBoxIcon::Stop);
    }
    finally//если finally{} не нужен - не пишем его
    {
        //безусловно выполняющийся в конце код (если такое нужно)
    }
}
```

Задание 1. Наберите код и протестируйте программу

```
#include <iostream>
#include <stdexcept>//содержит определения ошибок std::logic_error, std::runtime_error и всех ошибок, от них унаследованных
#include <exception>//содержит определения ошибок std::bad_exception, std::bad_alloc
#include <new>//#include <new.h>//заголовочный файл для выделения динамической области оперативной памяти под динамические объекты
#include <typeinfo>//содержит определения ошибок std::bad_cast и std::bad_typeid
#include <Windows.h>
using namespace std;
//сам класс exception определен в файле <vcruntime_exception.h>, но вы его подключаете косвенно через файлы, например, <stdexcept>
int main()
{
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    double z;
    int a = 1, x, y;
    while (a != 0)
    {
        try//блок кода try-catch помещен внутри цикла, поэтому возможные ошибки обрабатываются внутри цикла без завершения программы от возникших ошибок
        {//начало блока кода, в котором наша программа сама отслеживает ошибки
            cout << "Введите делимое: ";
            cin >> x;
            cout << "Введите делитель: ";
            cin >> y;
            if (y == 0)//если делитель у равен нулю, то пробросим (создадим, сгенерируем, вызовем) исключительную ситуацию типа класса ошибок std::logic_error
            {
                throw std::logic_error("Ошибка деления на ноль.\n");//создадим конструктором с параметром ошибку типа класса std::logic_error, передадим
            }// конструктору фразу "Ошибка деления на ноль.\n", которую он поместит в поле экземпляра ошибки
            if (x == 9999)//например, если пользователь введет 9999 в качестве делимого, то это неприемлемо для нас, поэтому мы отлавливаем это значение в иксе
            {
                throw std::runtime_error("Некоторая ошибка.\n");//и создаем конструктором с параметром ошибку типа класса std::runtime_error, передавая туда
            }// фразу "Некоторая ошибка.\n"
            if (x == 7777)//например, если пользователь введет 7777 в качестве делимого, то это неприемлемо для нас, поэтому мы отлавливаем это значение в иксе
            {
                throw 1;//и сгенерируем исключительную ситуацию, которая передается целочисленным значением, например, 1
            }
```

```
            z = (double)x / y;//делим, поскольку выше отловили интересовавшие нас ошибочные ситуации
            cout << "Частное от деления " << x << " на " << y << " равно: " << z << endl << "0-завершить; 1-повторить ввод. Выберите желаемое действие: ";
            cin >> a;
        }//конец блока кода, в котором наша программа сама отслеживает ошибки
        catch (int)//отловит ошибку, передающую целочисленное значение: throw 1;
        {
            cout << "Нельзя вводить делимое 7777.\n";
        }
        catch (std::logic_error e)//отлавливатель ошибок типа класса std::logic_error. Имя входному значению дает программист: e
        {
            cout << "Произошла логическая ошибка: " << e.what();//у входного объекта e вызовем метод that(), который вернет нам значение поля с текстом сохраненной
        }// в нем фразы об ошибке
        catch (std::exception e1)//отлавливатель ошибок типа класса std::exception. Имя входному значению дает программист: e1
        {//std::exception - самый базовый класс ошибок, поэтому он отловит ошибки любых типов, ведь они все будут экземплярами либо этого, либо его дочерних классов
            cout << "Произошла ошибка: " << e1.what();//универсальный отлавливатель ошибок располагаем последним, чтобы он отловил ошибки типов, которые мы забыли
        }//написать в отлавливателях перед ним. Грамотно отлавливать ошибки конкретных типов, поскольку они более узкоспециализированны, конкретны, более
        //информативны, сам их тип несет информацию о характере и роде ошибки
        //catch (...)//универсальный отлавливатель ошибки, фактический аналог catch (std::exception e), но последний более информативный
        //{
        //  cout << "Универсальный отлавливатель ошибки. Произошла ошибка.\n";
        //}
        //finally//в "чистом" консольном C++ не поддерживается использование блока кода finally{...}, но он поддерживается в Visual C++ в WindowsForms
        //{
        //  //код этой строки выполнился бы в любом случае, если бы "чистый" C++ для консоли поддерживал использование блока кода finally{...}
        //}
    }//в этом коде мы не отлавливаем ошибки типа ввода буквы вместо числа. Но эту проблему можно решить, например, принимать символьный массив, фильтровать его на
    system("pause");// наличие символов цифр и из них формировать целые числа функциями atoi(), atol() или вещественные числа функциями atof()
    return 0;
}
```

Тестируем программу на корректных и некорректных данных:

```
Консоль отладки Microsoft Visual Studio
Введите делимое: 78
Введите делитель: 9
Частное от деления 78 на 9 равно: 8.66667
0-завершить; 1-повторить ввод. Выберите желаемое действие: 1
Введите делимое: 98
Введите делитель: 0
Произошла логическая ошибка: Ошибка деления на ноль.
Введите делимое: 98
Введите делитель: 3
Частное от деления 98 на 3 равно: 32.6667
0-завершить; 1-повторить ввод. Выберите желаемое действие: 1
Введите делимое: 9999
Введите делитель: 7
Произошла ошибка: Некоторая ошибка.
Введите делимое: 9998
Введите делитель: 54
Частное от деления 9998 на 54 равно: 185.148
0-завершить; 1-повторить ввод. Выберите желаемое действие: 1
Введите делимое: 7777
Введите делитель: 567
Нельзя вводить делимое 7777.
Введите делимое: 7778
Введите делитель: 567
Частное от деления 7778 на 567 равно: 13.7178
0-завершить; 1-повторить ввод. Выберите желаемое действие: 89
Введите делимое: 9876
Введите делитель: 98760
Частное от деления 9876 на 98760 равно: 0.1
0-завершить; 1-повторить ввод. Выберите желаемое действие: 0
Для продолжения нажмите любую клавишу . . .

D:\2019\Labs\x64\Debug\Lab7_6.exe (процесс 16300) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите парамет
Нажмите любую клавишу, чтобы закрыть это окно…
```

Тестирование завершено. Сохраните программу.

Контекст для установки исключения – это блок try{…} catch(){…} finally{…}. Обработчики (отлавливатели ошибок) объявляются сразу за блоком кода try{} с использованием ключевого слова catch(…) {…}.

Пример:

```
vect::vect(int n)
{
	if(n < 1)
	{
		throw(n);
	}
	p = new int[n];
	if(p == 0)
	{
		throw("Exception: free store exhausted.\n");
	}
}
void g()
{
	try
	{
		vect a(n), b(n);
		//……..
	}
	catch(int n) //отслеживает все неправильные размеры массива
	{
		std::cout << “Exception: incorrect size of vector.\n”;
	}
	catch(char* error) //отслеживает запрос оперативной памяти, превышающий размер имеющейся свободной оперативной памяти
	{
		std::cout << “Exception: no memory.\n”;
	}
}
```

Установленные исключения.

Синтаксически выражение throw может записываться в двух формах:

throw

throw *выражение;*

Выражение throw устанавливает исключение. Выражение throw без аргумента повторно устанавливает текущее исключение. Обычно оно используется, когда для дальнейшей обработки исключения необходим второй обработчик, вызываемый из первого обработчика исключения.

```
void foo()
{
	int i;
	//……..
	throw (i);
```

```
}
main()
{
	try
	{
		foo();
	}
	catch(int i)
	{
		//……..
	}
}
```

Если пользователь хочет выводить дополнительную информацию или использовать ее для принятия решения относительно действий обработчика, то допустимо формирование исключения в виде объекта.

```
enum error {bounds, heap, other};
class vect_error
{
	private:
		error e_type;
		int ub, index, size;
	public:
		vect_error(error, int, int); //пакет вне пределов
		vect_error(error, int); //пакет вне памяти
}
```

Теперь выражение throw может быть более информативным, поскольку передает поясняющие данные о возникшей ошибке:

```
//...
throw vect_error(bounds, i, ub);
//...
```

Блоки *try*.

Синтаксически блок try имеет такую форму записи:

```
try
{
	составной оператор;
}
/*список обработчиков ниже*/
catch(…)
{
	//код, выполняющийся в случае возникновения ошибки данного типа
}
catch(…)
{
```

```
        //код, выполняющийся в случае возникновения ошибки данного типа
}
catch(…)
{
        //код, выполняющийся в случае возникновения ошибки данного типа
}
```

Блок try – контекст для принятия решения о том, какие обработчики вызываются для установленного исключения.

```
try
{
        //...
        throw("SOS.\n");
        //...
        io_condition eof(argv[i]);
        throw(eof);
        //...
}
catch (const char*)
{
        //...
}
catch (io_condition& x)
{
        //...
}
```

Выражение throw соответствует аргументу catch(входной аргумент), если он:

1. Точно соответствует.
2. Общий базовый класс порожденного типа представляет собой то, что устанавливается.
3. Объект установленного типа является типом указателя, преобразуемым в тип указателя, являющегося аргументом catch(аргумент).

Обработчики.

Синтаксически обработчик catch имеет следующую форму записи в коде:

```
catch (формальный аргумент)
{
        составной оператор;
}
```

Пример:

```
catch (char* message)
{
        cerr << message << endl;//cerr – это поток для ошибок, обычно по умолчанию
        //выводит текст ошибки на консоль cerr – Console ERRoor
}
```

```
catch (...) //действие по умолчанию
{
	cerr << "This is an exceptional situation. Do not worry, please.\n";
	abort();//убить этот процесс (прекратить выполнение этой программы и
	//освободить от нее место в оперативной памяти)
}
```

Спецификация исключения.

Синтаксис:

заголовок функции throw(*список типов входных аргументов*);

Пример:

void foo() throw(int, over_flow);//описание функции foo(), которая ничего не принимает и ничего не возвращает, но в которой генерится исключение, которое посылает два значения: первое – целочисленного типа, второе – типа класса over_flow

void noex(int i) throw();//описание функции noex(), которая принимает целочисленное значение и ничего не возвращает, но в которой декларируется нетипизированное универсальное исключение, которое может произойти, при этом никаких данных не посылается

***Обработчики terminate* и *unexpected*.**

Обработчик terminate() вызывается, когда для обработки исключения НЕ поставлен другой обработчик, то есть НЕ написан никакой обработчик ошибок. Обработчик terminate по умолчанию вызывает в своем теле функцию abort() для процесса вашей программы, тем самым уничтожая процесс вашей программы со всеми его внутренними потоками.

Обработчик unexpected() вызывается, когда исключения не было в списке спецификации исключений, то есть в списке catch'ей.

2. Пример выполнения программы

```
#include <iostream>
using namespace std;
const int MAX = 3;//пусть в нашем стеке может быть максимум 3 целых числа
class Stack
{
	private:
		int st[MAX];// стек: целочисленный массив
		int top;// индекс вершины стека
	public:
		class Range // класс исключений для Stack
		{
			// внимание: тело класса пусто
		};
		Stack() // конструктор
		{
```

```
                top = -1;
        }
        void push(int var)
        {
                if(top >= MAX - 1)// если стек заполнен,
                {
                        throw Range();// генерировать исключение
                }
                st[++top] = var;// внести число в стек
        }
        int pop()
        {
                if(top < 0) // если стек пуст,
                {
                        throw Range();// исключение
                }
                return st[top--]; // взять число из стека
        }
};
int main()
{
        Stack s1;
        try
        {
                s1.push(11);
                s1.push(22);
                s1.push(33);
                // s1.push(44);//Стек заполнен
                cout << "1: " << s1.pop() << endl;
                cout << "2: " << s1.pop() << endl;
                cout << "3: " << s1.pop() << endl;
                cout << "4: " << s1.pop() << endl; //Стек пуст
        }
        catch(Stack::Range) // обработчик
        {
                cout << "Исключение: Стек переполнен или пуст.\n";
        }
        cout << " Сюда после захвата исключения или нормального выхода.\n";
```

```
    system("pause");
    return 0;
}
```

3.Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.
2. **Задание 2.** Реализовать алгоритм решения задачи и написать программу (консольную или WindowsForms на ваш выбор). Постройте таблицу значений функции y() для x[a; b] с шагом h. Если в некоторой точке x *функция y(x)* не определена (y нельзя вычислить математически, обработайте это как исключение), то выведите на экран сообщение об этом.

$$y = \frac{\sqrt{x^3 - 1}}{\sqrt{x^2 - 1}}$$

3. Разработать на языке C++ программу вывода на экран решения задачи в соответствии с вариантом индивидуального задания, указанным преподавателем.
4. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.
5. **Задание 3.** Написать программу циклического вычисления значений функции, определенной для вас из таблицы вариантов заданий ниже. Значения R должны вводиться с клавиатуры. R1, R2, R3 – вещественные или целые числа (также R3 можно создать как комплексное число). Отлавливать несколько возможных исключительных ситуаций, среди которых должны быть исключения, указанные по вашему варианту, и «универсальное» исключение в конце.

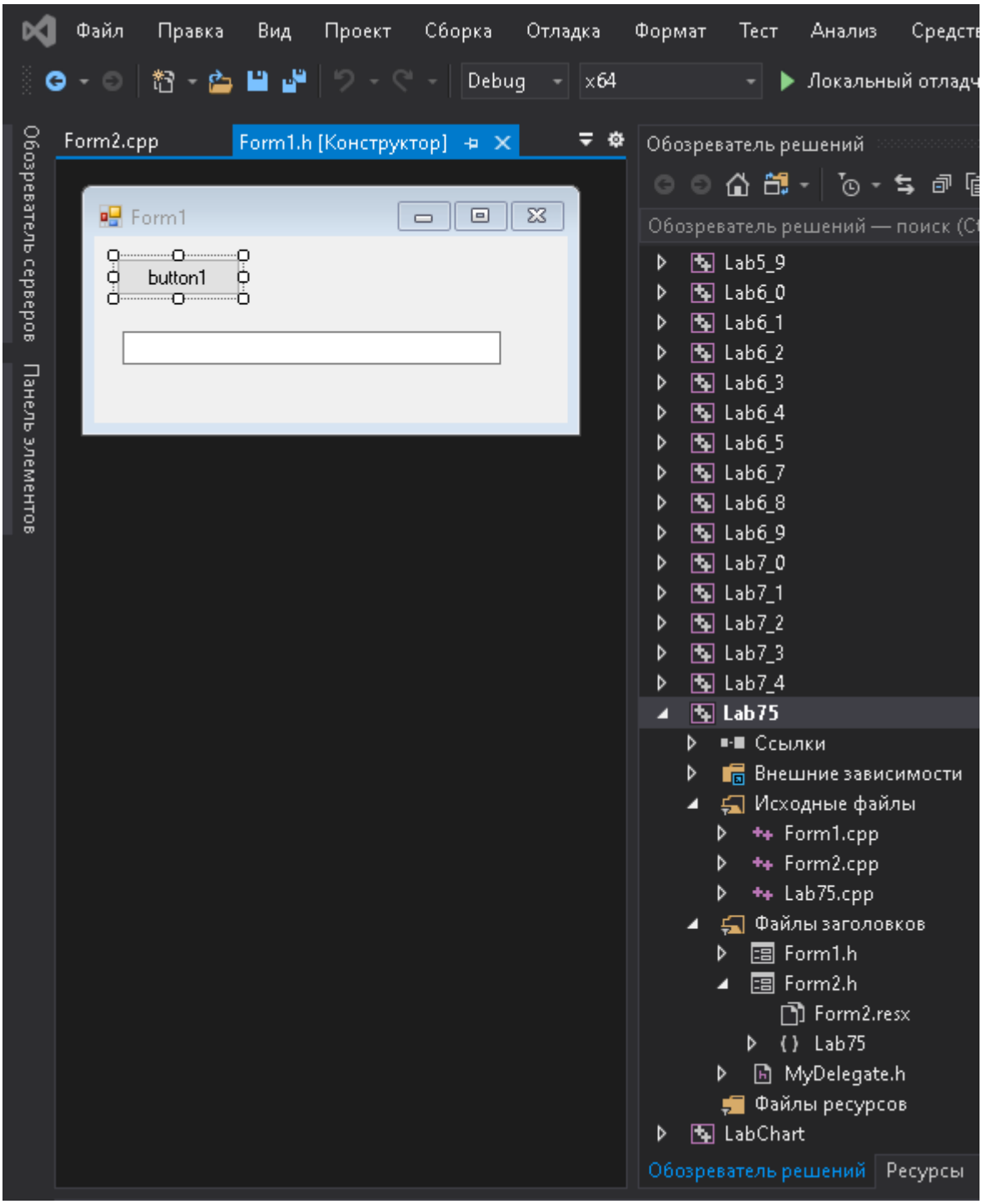
№	Функция	Исключительная ситуация
1	Sin(R1)*(pi)/R2-R3	Деление на 0
2	Sin(R2)/pi*R1+R3	Деление на 0; Переполнение
3	Sqrt(Sin(R1)+R2)+R3	Область определения аргумента, корень из отрицательного числа
4	Sqrt(Tan(R1)+R2+R3)	Корень из отрицательного числа; Прерывание
5	Ln(Cos(R1)-R2)/R2-R3	Область определения аргумента; деление на ноль
6	Tan(R2%R1)/R1+Cmod(R3)	Деление на 0
7	Cos(R1)/arctan(R2/R1)+R3	Прерывание; деление на ноль
8	Ln(R1-R2)*R2-R3	Область определения аргумента
9	Sin(R2)*Cos(R1)+Sqrt(R3)	Переполнение; корень из отрицательного числа
10	Ln(R1)-Exp(R2)+R3	Область определения аргумента
11	Sin(Pow(R1,R2))/(R2-R3)	Деление на 0
12	Sin(R1*R1)+R1/R2+Cmod(R3)	Переполнение; деление на ноль
13	Cosh(R1)/R2+R3	Область определения аргумента, деление на ноль
14	Sqrt(Arctan(R1)/(R2)-R3)	Деление на 0; корень из отрицательного числа
15	Tanh(R1)/R2-R3	Область определения аргумента, деление на ноль

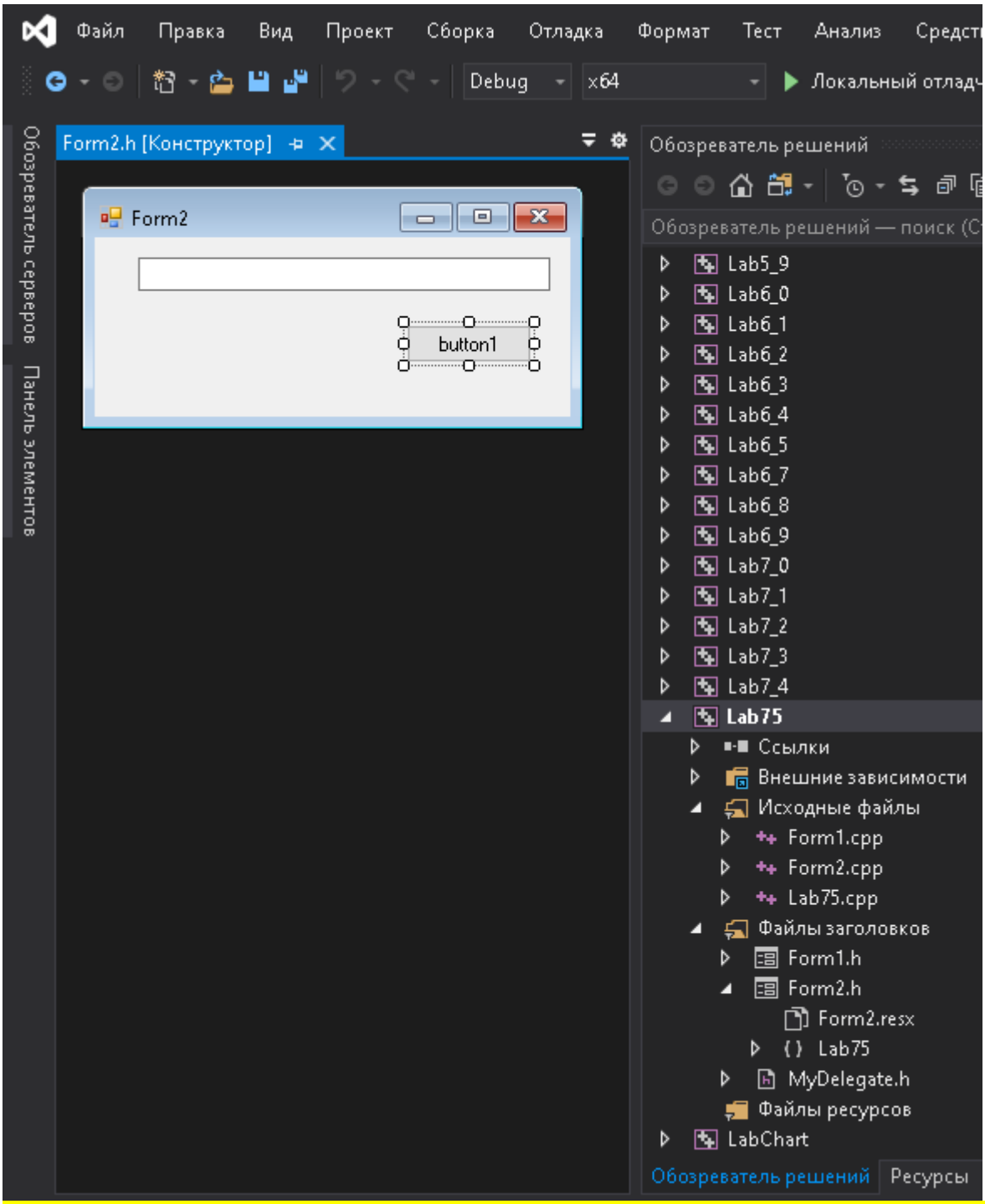
16	Tan(R1)/(R1+R2)-R3	Деление на 0
17	Tan(R1)/R3+Cmod(R3)	Прерывание; деление на ноль
18	Arctan(R1*Sqrt(R2))/R2-Cmod(R3)	Область определения аргумента, деление на ноль
19	Arctan(R2)/R1-Cmod(R3)	Переполнение, деление на ноль
20	Sqrt(R1)*R2-Cmod(R3)	Область определения аргумента; корень из отрицательного числа
21	Tan(R1)+Pow(R1,R2)+Cmod(R3)/R1	Переполнение; деление на ноль
22	Mod(R1,R2)/(Exp(R1)+R3)	Сверхмалый результат, деление на ноль
23	Sin(R2)/R1+Cmod(R3)	Деление на 0
24	Exp(R1)*Arctan(R2)/R2-R3	Сверхмалый результат; деление на ноль
25	Arctan(R1)/R1+R2+Cmod(R3)	Переполнение, деление на ноль
26	(R2*R1) / Sqrt(R2 * R1 * R3)	Деление на ноль, корень из отрицательного числа
27	(Sin(R2)-Cos(R1)) / Sqrt(R2 + R1)	Деление на ноль, корень из отрицательного числа
28	(Tan(R1)+Pow(R1,R2)) / R1	Переполнение, деление на ноль

Задание 4. Создайте оконную программу как в примере ниже

Создание проекта, в котором из второй оконной формы данные могут многократно передаваться в первую оконную форму по желанию (действию) пользователя. Использование делегатов (указателей на методы) в управляемом коде Visual C++. Размещение кода приложения в различных файлах проекта. Создание модальных и НЕмодальных окон.

1. создать пустой CLR-проект с названием, например, Lab75;
2. создать два окна – Form1 и Form2 – с кнопкой и текстбоксом в каждом из этих окон:


Файл
Правка
Вид
Проект
Сборка
Отладка
Формат
Тест
Анализ
Debug
x64
Локальный отладч
Form2.cpp
Form1.h [Конструктор]
Form1
button1
Обозреватель решений
Обозреватель решений — поиск
Lab5_9
Lab6_0
Lab6_1
Lab6_2
Lab6_3
Lab6_4
Lab6_5
Lab6_7
Lab6_8
Lab6_9
Lab7_0
Lab7_1
Lab7_2
Lab7_3
Lab7_4
Lab75
Ссылки
Внешние зависимости
Исходные файлы
Form1.cpp
Form2.cpp
Lab75.cpp
Файлы заголовков
Form1.h
Form2.h
Form2.resx
Lab75
MyDelegate.h
Файлы ресурсов
LabChart
Обозреватель решений
Ресурсы
Обозреватель серверов
Панель элементов



3. создать для функции WinMain() собственный файл Lab75.cpp (обычный *.cpp-файл, а не Оконную форму) с кодом:

```
#include "Form1.h"//подключаем оконную форму Form1
#include <Windows.h>
using namespace Lab75;

[STAThreadAttribute]
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	Application::Run(gcnew Form1());//вызываем конструктор Form1() и создаем динамический управляемый Сборщиком мусора объект
	return 0;
}
```

4. добавить в проект файл MyDelegate.h (обычный заголовочный *.h-файл, а не Оконную форму) с кодом:

```
#pragma once
using namespace System;

namespace Lab75
{
	public delegate void MyDelegate(String^ data);//в пространстве имен программы Lab75 глобально декларируем делегат как ссылку на метод, который принимает объект типа String^ (это управляемый Сборщиком мусора указатель на строку символов (char'овский массив)) и ничего не возвращает (void)
}
```

5. в файл Form2.h добавить код:

```
#pragma once
#include "MyDelegate.h"//надо подключить делегат MyDelegate
#include "Form1.h"//надо подключить оконную форму Form1

namespace Lab75 {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

	/// <summary>
	/// Сводка для Form2
	/// </summary>
	public ref class Form2 : public System::Windows::Forms::Form
	{
	public:
		Form2(MyDelegate^ sender);//декларация конструктора оконной формы Form2; его инициализация с телом в файле Form2.cpp
	private:
		MyDelegate^ deleg;//закрытое поле типа делегата (ссылки на метод)
	public:
		Form2(void)
		{
			InitializeComponent();
			//
			//TODO: добавьте код конструктора
			//
		}

	protected:
		/// <summary>
		/// Освободить все используемые ресурсы.
		/// </summary>
		~Form2()
		{
			if (components)
			{
				delete components;
			}
		}
	private: System::Windows::Forms::TextBox^ textBox1;
	private: System::Windows::Forms::Button^ button1;
	protected:

	private:
		/// <summary>
		/// Обязательная переменная конструктора.
		/// </summary>
		System::ComponentModel::Container ^components;
```

```
#pragma region Windows Form Designer generated code
		/// <summary>
		/// Требуемый метод для поддержки конструктора — не изменяйте
		/// содержимое этого метода с помощью редактора кода.
		/// </summary>
		void InitializeComponent(void)
		{
			this->textBox1 = (gcnew System::Windows::Forms::TextBox());
			this->button1 = (gcnew System::Windows::Forms::Button());
			this->SuspendLayout();
			//
			// textBox1
			//
			this->textBox1->Location = System::Drawing::Point(26, 12);
			this->textBox1->Name = L"textBox1";
			this->textBox1->Size = System::Drawing::Size(246, 20);
			this->textBox1->TabIndex = 0;
			//
			// button1
			//
			this->button1->Location = System::Drawing::Point(186, 52);
			this->button1->Name = L"button1";
			this->button1->Size = System::Drawing::Size(75, 23);
			this->button1->TabIndex = 1;
			this->button1->Text = L"button1";
			this->button1->UseVisualStyleBackColor = true;
			this->button1->Click += gcnew System::EventHandler(this,
&Form2::button1_Click);
			//
			// Form2
			//
			this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
			this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
			this->ClientSize = System::Drawing::Size(284, 107);
			this->Controls->Add(this->button1);
			this->Controls->Add(this->textBox1);
			this->Name = L"Form2";
			this->Text = L"Form2";
			this->ResumeLayout(false);
			this->PerformLayout();

		}
#pragma endregion
	private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e);//тут
декларация метода; а его инициализация с телом будет в файле Form2.cpp
	};
}
```

6. в файл Form1.h добавить код:

```
#pragma once

namespace Lab75 {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

	/// <summary>
	/// Сводка для Form1
	/// </summary>
	public ref class Form1 : public System::Windows::Forms::Form
	{
	public:
		Form1(void)
```

```
		{
			InitializeComponent();
			//
			//TODO: добавьте код конструктора
			//
		}

	protected:
		/// <summary>
		/// Освободить все используемые ресурсы.
		/// </summary>
		~Form1()
		{
			if (components)
			{
				delete components;
			}
		}
	private: System::Windows::Forms::Button^ button1;
	private: System::Windows::Forms::TextBox^ textBox1;
	protected:

	private:
		/// <summary>
		/// Обязательная переменная конструктора.
		/// </summary>
		System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
		/// <summary>
		/// Требуемый метод для поддержки конструктора — не изменяйте
		/// содержимое этого метода с помощью редактора кода.
		/// </summary>
		void InitializeComponent(void)
		{
			this->button1 = (gcnew System::Windows::Forms::Button());
			this->textBox1 = (gcnew System::Windows::Forms::TextBox());
			this->SuspendLayout();
			//
			// button1
			//
			this->button1->Location = System::Drawing::Point(13, 13);
			this->button1->Name = L"button1";
			this->button1->Size = System::Drawing::Size(75, 23);
			this->button1->TabIndex = 0;
			this->button1->Text = L"button1";
			this->button1->UseVisualStyleBackColor = true;
			this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
			//
			// textBox1
			//
			this->textBox1->Location = System::Drawing::Point(17, 57);
			this->textBox1->Name = L"textBox1";
			this->textBox1->Size = System::Drawing::Size(227, 20);
			this->textBox1->TabIndex = 1;
			//
			// Form1
			//
			this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
			this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
			this->ClientSize = System::Drawing::Size(284, 112);
			this->Controls->Add(this->textBox1);
			this->Controls->Add(this->button1);
			this->Name = L"Form1";
			this->Text = L"Form1";
			this->ResumeLayout(false);
```

```
                this->PerformLayout();

            }
#pragma endregion
        private:
            System::Void button1_Click(System::Object^ sender, System::EventArgs^ e);//тут
декларация метода; а его инициализация с телом будет в файле Form1.cpp
        public://метод GetData() общедоступный
            void GetData(String^ param)
            {
                this->textBox1->Text = "";
                this->textBox1->Text = param;
            }
        };
}
```

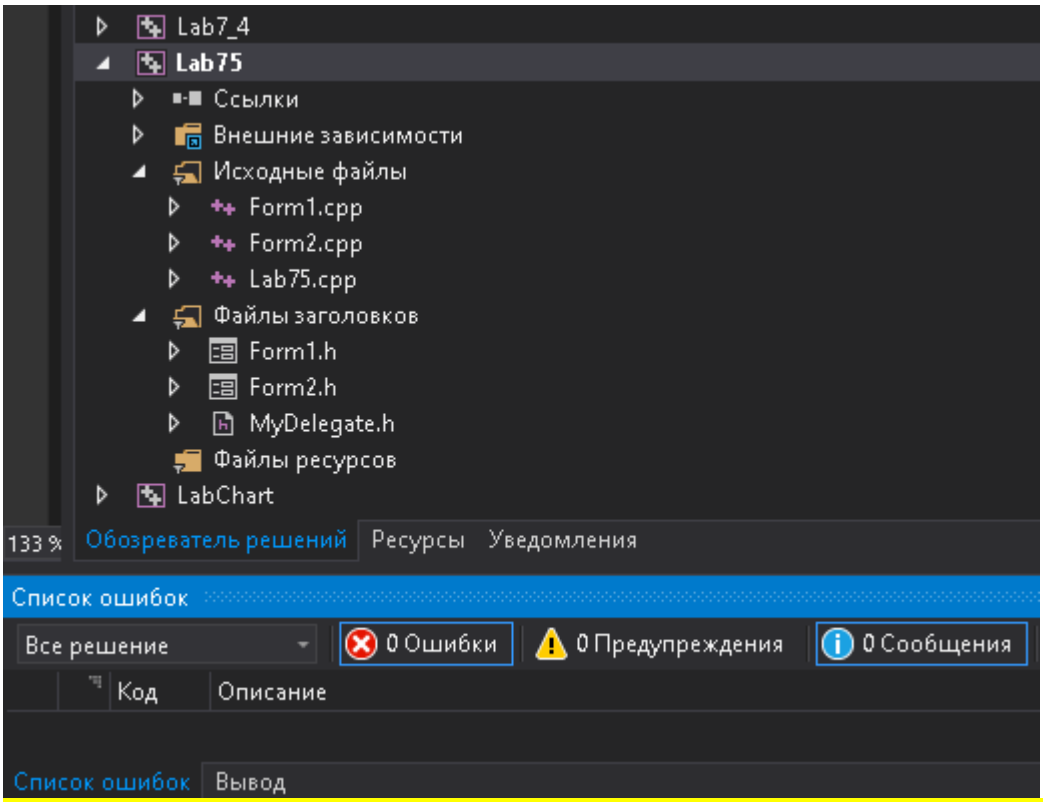
7. в файл Form1.cpp добавить код:

```
#include "Form1.h"//подключаем ОБА окна
#include "Form2.h"

using namespace Lab75;

void Form1::button1_Click(Object^ sender, EventArgs^ e)
{
        Form2^ f = gcnew Form2(gcnew MyDelegate(this, &Form1::GetData));//передается 2 параметра
        f->ShowDialog();//окно f откроется как МОДАЛЬНОЕ окно, а если f->Show();//окно f откроется
как НЕмодальное окно
}
```

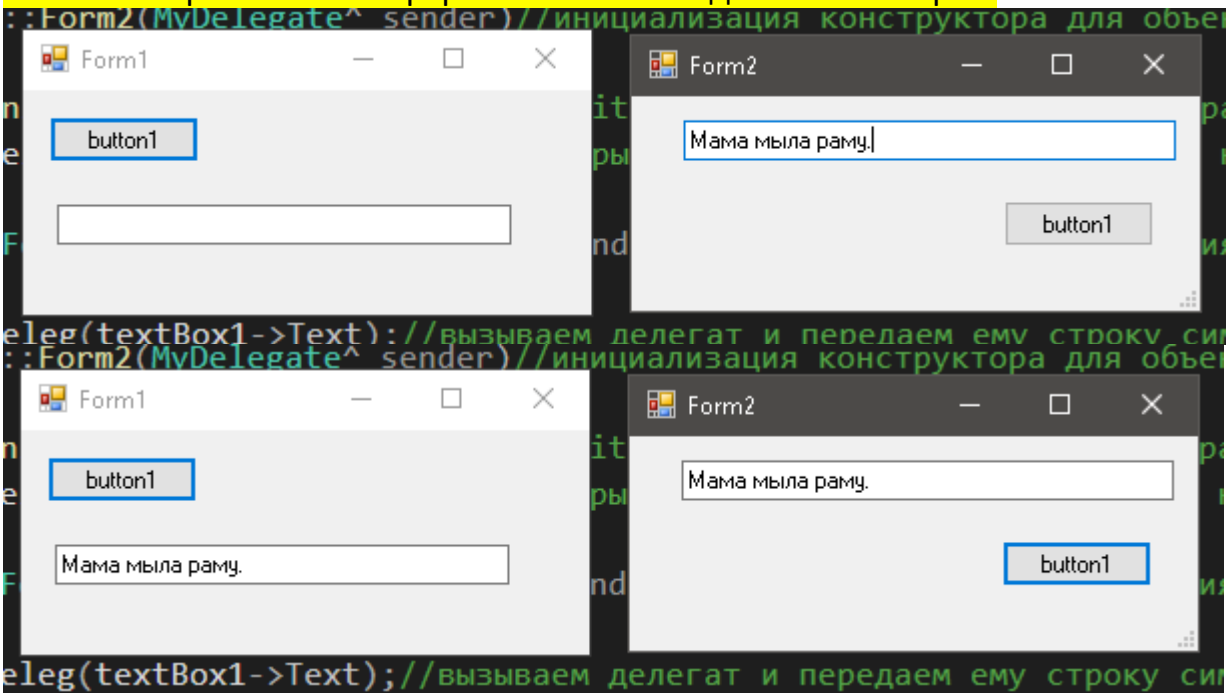
8. в файл Form2.cpp добавить код:

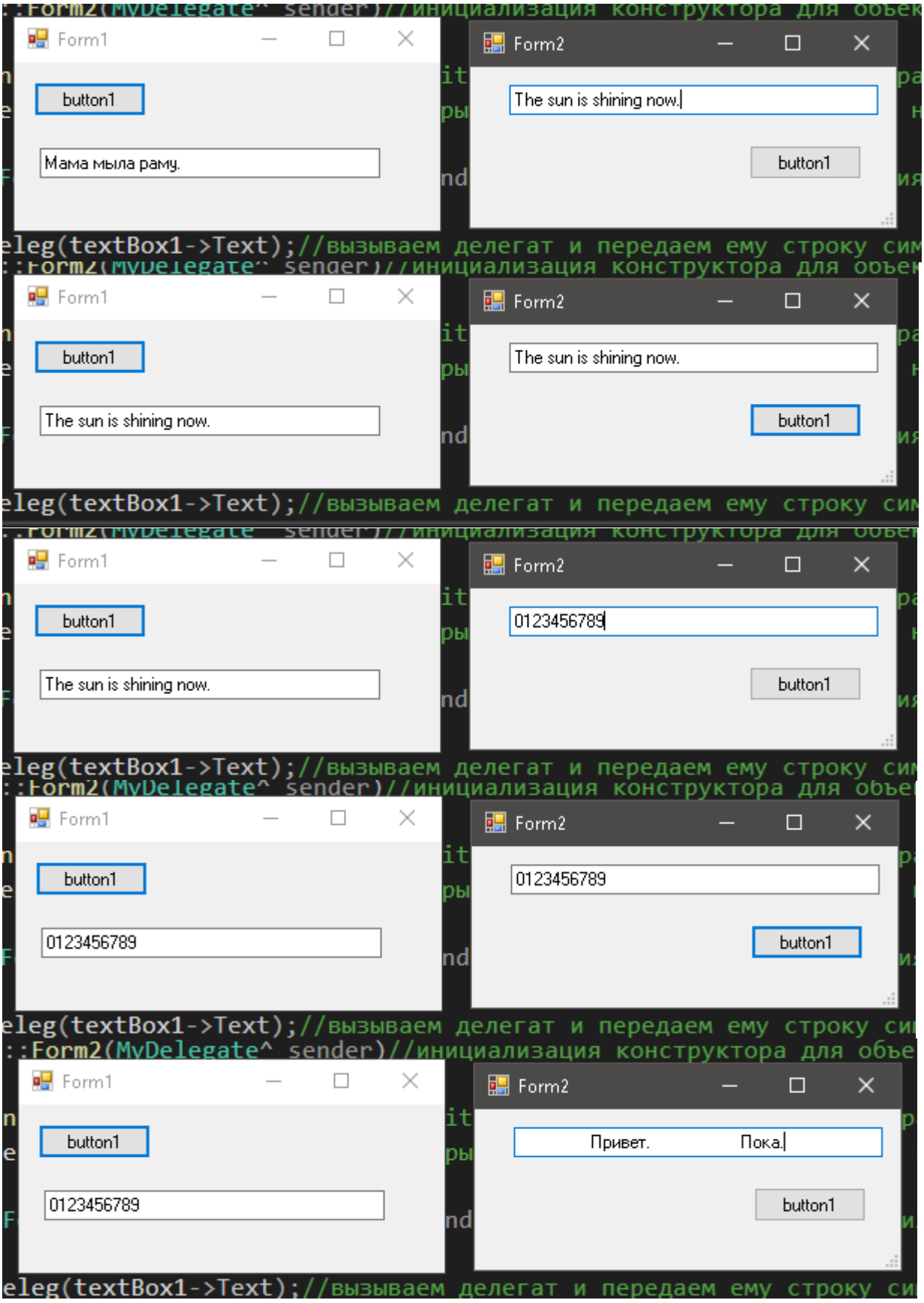
```
#include "Form1.h"//подключаем ОБА окна
#include "Form2.h"

using namespace Lab75;

Form2::Form2(MyDelegate^ sender)//инициализация конструктора для объектов класса Form2
{
    InitializeComponent();//метод InitializeComponent() в конструкторах всегда пишите ПЕРВЫМ
выражением
    deleg = sender;//deleg - это закрытое поле типа делегата (ссылки на метод), продекларированное в
файле Form2.h: private: MyDelegate^ deleg;
}
void Form2::button1_Click(Object^ sender, EventArgs^ e)//инициализация метода button1_Click() для
оконной формы Form2
{
    deleg(textBox1->Text);//вызываем делегат и передаем ему строку символов из тексБокса1. Делегат
вызовет метод &Form1::GetData(String^ param), который принимаемую строку символов поместит в поле
текстБокса1 оконной формы Form1
}
```

9. компилируем, исправляем ошибки, проверяем в Обозревателе решений, чтобы заголовочные *.h и исходного кода *.cpp файлы в рамках проекта Lab75 находились по своим группам:



10. тестируем возможность передать данные (строку символов) из ВТОРОЙ оконной формы в ПЕРВУЮ. Поскольку второе окно МОДАЛЬНОЕ, то при запущенном втором окне вводить данные в окно первой оконной формы (или даже просто нажать на button1 первой оконной формы при запущенном втором окне) пользователю нельзя, но можно вводить данные в текстБокс1 второй оконной формы и нажимать кнопку button1 второй оконной формы. Так можно делать много раз:

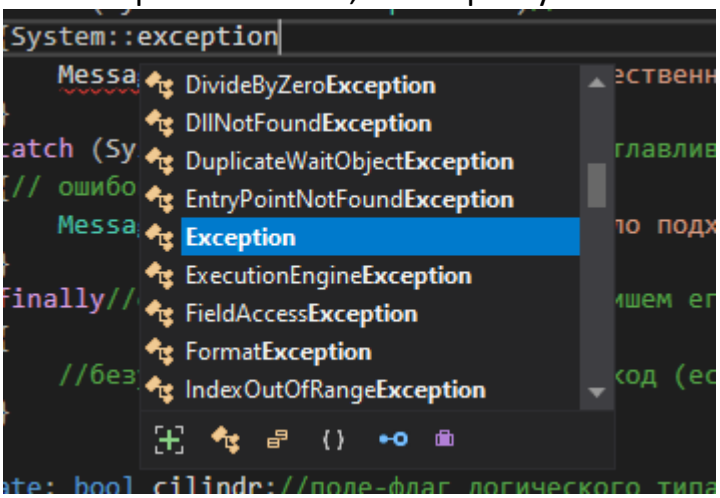



Form1
button1
Мама мыла раму.
Form2
The sun is shining now.
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку сим
Form1
button1
The sun is shining now.
Form2
The sun is shining now.
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку сим
Form1
button1
The sun is shining now.
Form2
0123456789
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку сим
::Form2(MyDelegate^ sender)//инициализация конструктора для объе
Form1
button1
0123456789
Form2
0123456789
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку си
::Form2(MyDelegate^ sender)//инициализация конструктора для объе
Form1
button1
0123456789
Form2
Привет.
Пока.
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку си
Form1
button1
Привет.
Пока.
Form2
Привет.
Пока.
button1
eleg(textBox1->Text);//вызываем делегат и передаем ему строку си

11. Таким образом, с помощью делегатов и расположения кода по разным группам файлов можно создавать оконные Windows Forms приложения, в которых получится многократно передавать данные из второго окна в первое, причем второе окно при этом может как оставаться открытым, так и закрываться пользователем.

Задание 5.

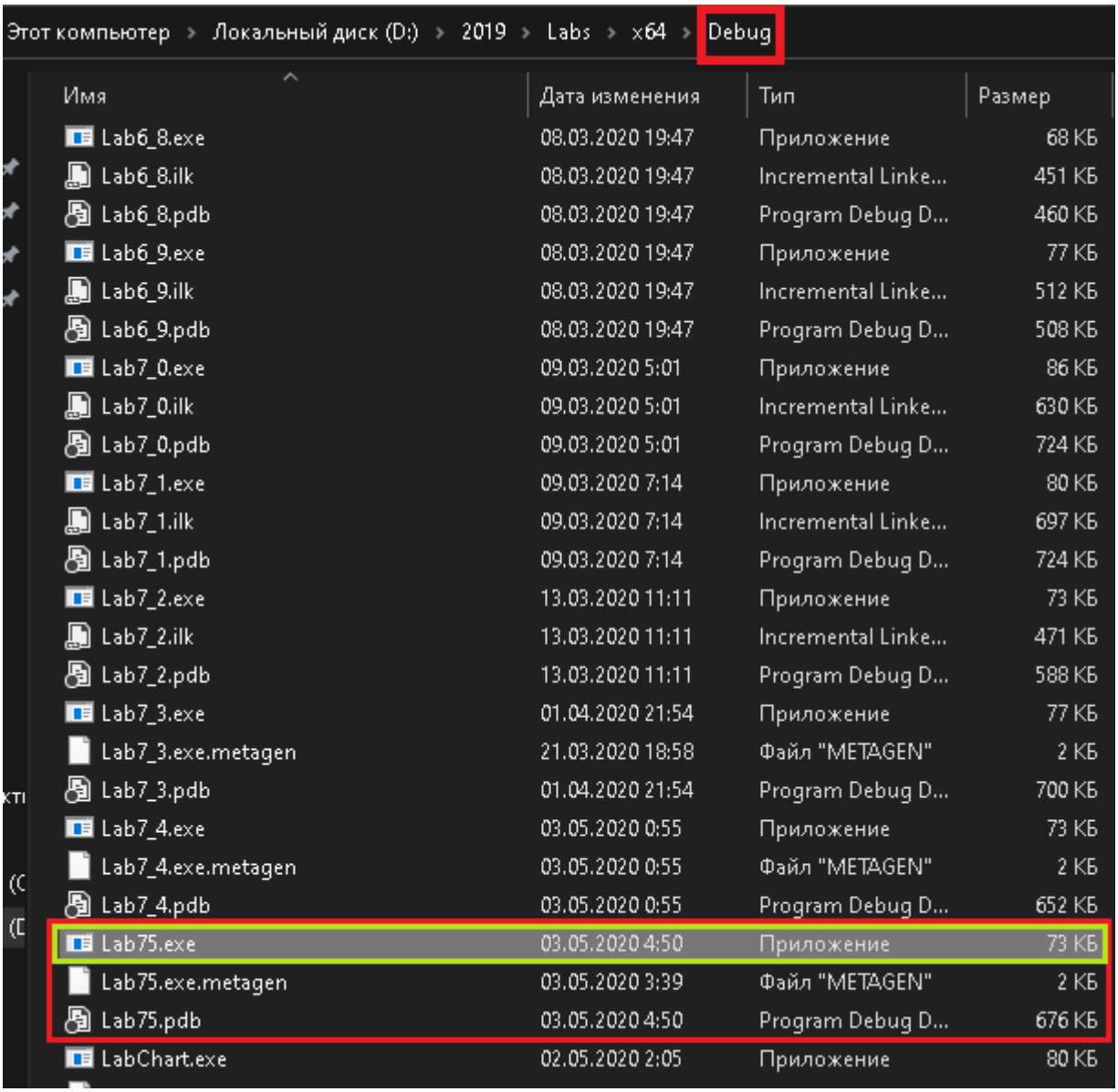
Допишите тела методов обработчиков нажатия на кнопки button обеих оконных форм и метода GetData() с помощью блоков try – catch или try – catch – finally для отлова в них ошибок нескольких типов, последний из которых «универсальный» System::Exception^ e. Все исключительные ситуации в VC++ имеют в своем названии слово Exception, причем оно находится в конце названия исключительной ситуации, например, DivideByZeroException, NullReferenceException, FormatException, OverflowException и так далее, поэтому для получения списка имеющихся типов классов Exception надо набрать System::Exception и во всплывающем списке исключений вы увидите готовые классы исключений, настроенные на обозначение ошибок разных типов, о которых указывают их англоязычные названия:



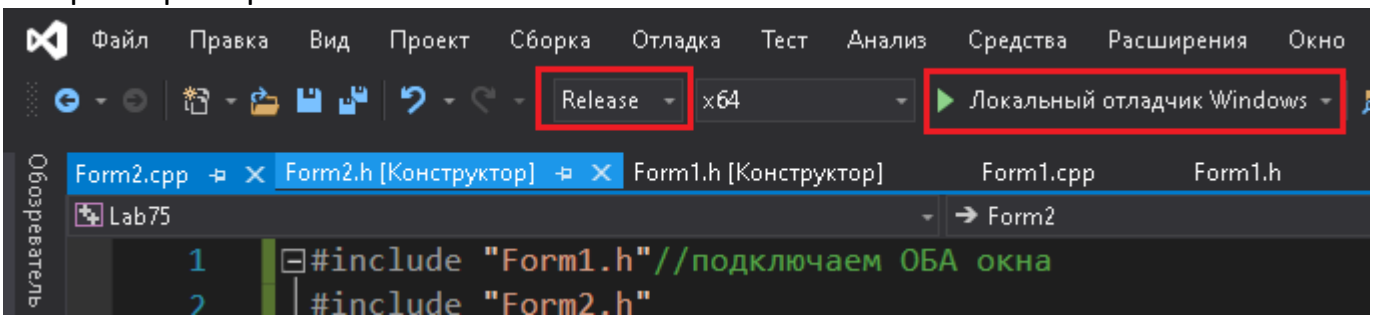
Задание 6. Проделайте нижеследующие действия

Создание и различия Debug и Releaze-сборок проектов приложений

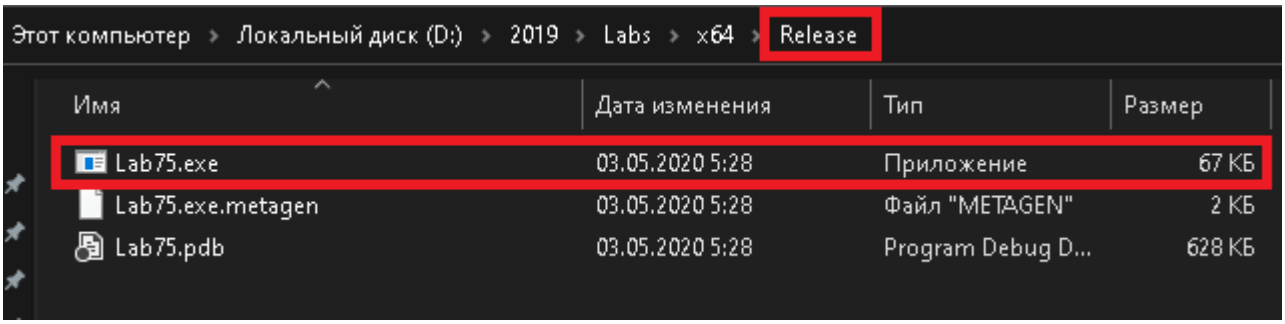
12*. Для курсовых проектов можно после исправления всех ошибок в программе сделать финальную Releaze-сборку проекта приложения, но учтите, что при дальнейшем возникновении необходимости отладки приложения, нужно вернуть Debug-сборку. За счет хранения вспомогательной информации, необходимой для обеспечения отладки, Debug-сборки имеют больший размер, чем Releaze-сборки. Размер исполнимого файла для приложения Lab75.exe составляет 73 КилоБайта.



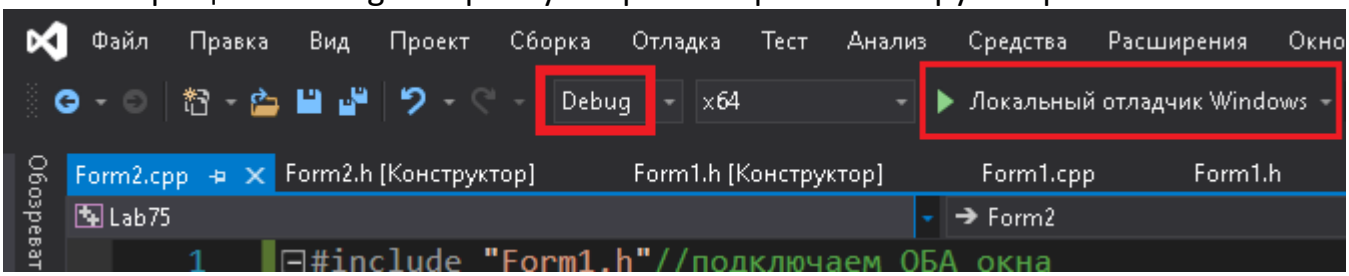
13*. Поменяем тип сборки на Releaze, выставив соответствующую настройку в проекте и пересобрав проект:



14*. Ищем папку Releaze и смотрим размер нашего итогового приложения. Размер уменьшился за счет удаления ВижуалСтудией вспомогательной информации из исполнимого файла, но при этом стала недоступной отладка приложения, проведение замеров и метрик его работы:



15*. Возвращаем Debug-настройку сборки и перекомпилируем проект:



6. Ответить на контрольные вопросы.

4. Контрольные вопросы

1. Можно ли обрабатывать исключительные ситуации, не используя try, throw, catch, и если да, то зачем нужны последние?

2. Как реализуется обработка исключений?

3. Какие сообщения и в какой последовательности будут выведены на монитор при a=1 и a=0 и почему?

```
class Alpha
{
	public:
		Alpha()
		{
			cout << "Constructor A." << endl;
		}
		~Alpha()
		{
			cout << "Destructor A." << endl;
		}
};
void myfunc()
{
	try
	{
		Alpha alf;
		int a;
		cin >> a;
		if( a )
```

```
        {
            throw 1;
        }
        cout << “End of function.\n”;
    }
    catch(int)
    {
        cout << "Exception: Handle of Integer." << endl;
    }
    cout<<"Continue."<<endl;
}//конец тела функции myfunc()
int main()
{
    myfunc();
    system(“pause”);
    return 0;
}
```

Литература

Страуструп, Б. Язык программирования C++ / Б. Страуструп. – СПб. : БИНОМ, 2011.

Павловская, Т. А. C++. Объектно-ориентированное программирование: практикум / Павловская, Т. А., Щупак. – СПб. : Питер, 2011.

Преподаватель Белокопыцкая Ю.А.

Рассмотрено на заседании
цикловой комиссии ПОИТ № 10
Протокол № ____ от «___» _____ 2017 года
Председатель ЦК_____