Введение в (и заодно выведение из) классы и прочее ООП.

В написании сего опуса участвовали два студента мгту, один выпускник какого-то днепропетровского айтишного вуза, и один геймдев-самоучка. при написании было ПОТРАЧЕНО много нервных клеток, литров чая, и килобайтов логов жарких споров и дискуссий «доходчивость vs корректность». Сий текст правилен ровно настолько, насколько это возможно для объяснения классов тому, кто с ними никогда не сталкивался – его можно юзать при знакомстве с дотнетом, но не для написания про-хардкор-трустори лабораторных задач по ООП – используй свои знания с осторожностью, анон!

**Об азах**

Азом я буду считать *структуру*. На самом деле это не совсем корректно, потому что структуру в С++ надо считать частным случаем класса, но пока давайте считать что это наоборот класс эволюционировал из структуры (де факто, так оно и есть, если сравнивать с89 и цыплюс). Вобщем забудьте что я сказал что структура это частный случай класса, я этого не говорил, ага? Итак, всё растёт из структуры.

Дляначала немного оффтопа, чтобы понять что это такое и как используется.

Рассмотрим игруфку, в которой бегают цветные пони. Их куча, самых разных, но все они отталкиваются от одного набора исходных параметров – например от цвета, здоровья и скорости. В СИ мы бы завели структуру,в которой бы хранили базовые данные о пони.

struct pony

{

int hp;

float speed;

int color;

};

Структура – группа из нескольких переменных, идущих друг за другом в одном блоке памяти. Грубо говоря, переменная, внутри которой есть ещё переменные. Её можно объявить явно (она создастся на стеке) или через указатель (тогда её можно будет создать в динамической памяти). *//Про стек и динамику – это так, по требованию редакции и для общего развития.*

Явно – это pony art; дальнейший доступ осуществляется через art.hp. Через указатель – это pony \*art = new pony; доступ осуществляется через art->speed.

new – новая милота из цыплюса, которая автоматически выделяет ровно столько памяти сколько нужно под экземпляр, без какихлибо маллоков и сайзофов. Тем не менее, когда мы объявляем любую переменную через нью, потом её нужно будет **вручную** удалить из существования, во имя Бога-Императора и контрацепции от утечек памяти. Делается это через delete art.

На невысказанный вопрос «а нахрена нам извращаться с указателями, когда можно объявить структуру явно» - использование структур через указатели чрезвычайно полезно во всяких функциях. Легче передать в функцию какой-нибудь 4байтовый указатель, чем копировать 10кб данных структуры. Более того, временный объект, созданный без нью, на то и временный что уничтожится при выходе из {блока}, в котором был объявлен, в то время как созданный через new будет доступен к использованию вплоть до вызова делет.

Но возвращаясь к нашему лирическому оффтопу, его там совсем мало осталось.

Итак, у нас есть разноцветный табун поней, и мы захотели взять отдельный экземпляр из этого табуна, и изменить его переменную color. Для этого получилась бы функция, в которую мы бы передавали указатель на нужного коня и новый цвет

void SetPonyColor( pony \*target, int color )

{

target->color = color;

}

Вот базовый принцип работы со структурами с точки зрения структурного программирования. Прежде чем переходить к объектному, я сделаю ещё одно отступление, которое может весьма облегчить понимание.

**О неймспейсах**

namespace - это ещё одна милота из цыплюса, благодаря которой мы можем объявлять переменные (и функции (и вообще всё что угодно)) с одинаковым именем, и при этом всёравно понимать, какую мы имеем в виду.

namespace IU7

{

int Ivan;

double Maria;

void func()

{

printf("iu7");

}

}

namespace RK4

{

int Ivan;

int Maria;

void func()

{

printf("rk4");

}

}

В результате этого финта ушами мы по сути объявляем не double Maria и int Maria, а double IU7Maria и int RK4Maria. Но эта приставка перед именем, делающая наши глаза плакать, определяется внутри компилятора компилятором, а не в непосредственно именах нами. В дальнейшем доступ осуществляется через IU7::Maria и RK4::Ivan.

Также можно сказать "не хочу учиться, хочу жениться" и ~~отправиться в армию~~ ~~пойти на панель~~ написать using namespace IU7; что позволит обращаться не к IU7::Ivan а напрямую к Ivan. Более того, после этого даже можно будет написать ещё и using namespace RK4; однако тогда уже нельзя будет понять при вызове func() - вызываем ли мы печаталку иу7 или печаталку рк4? В этом случае нужно уже будет конкретизировать это через неймспейс::func(); аналогично и с иваном и марией.

Смысл юзинга в том, что неймспейс, который мы юзинг, будет первостепенным источником для «поиска» требуемой переменной - Ivan будет искаться в первую очередь среди переменных неймспейса.

Алсо можно объявлять неймспейсы внутри неймспейсов внутри неймспейсов, ехал гитлер через гитлер,

namespace NAME1

{

int A;

namespace NAME2

{

int A;

}

namespace NAME3

{

float A;

}

}

и иметь доступ через NAME1::NAME2::A или NAME1::A. Будь осторожен, анон - если ты ляпнешь using namespace NAME1; using namespace NAME2; и попробуешь обратиться напрямую к А, терпение богоподобного цыплюса иссякнет и тебя нещадно ткнут носом в то что ты пытаешься погнаться за двумя зайцами. *//как правило это заканчивается тем что зайцы обегают дерево с разных сторон, а погонщик въябывается в дерево и остаётся наедине со своими шишками*

**О классах и публичности, приватности, защищенности**

В достаточно далёком приближении, на класс можно смотреть как на помесь неймспейса и структуры. От неймспейса в нём – синтаксис связанный с ::, от структуры – наполнение переменными и прочим добром.

Можно посмотреть на приведённый выше пример об игрушке про коней с точки зрения ООП, и подумать – а давайте функции, которые влияют на состояния отдельных объектов, привяжем к самим объектам? Как только мы об этом подумали, появилась концепция такой вещи как class, этакой структуры 2.0, где помимо переменных (полей класса) определены ещё и функции (методы класса).

class pony

{

public:

int hp;

float speed;

color32 GetMane()

{

return manecolor;

}

protected:

color32 manecolor;

color32 coatcolor;

private:

pony \*specialsomepony;

};

pony dream;

pony \*daisy = new pony;

Для начала, всё что есть в классе, и поля (переменные) и методы (функции), делятся на публичные, защищённые и приватные. Сие есмь камень преткновения, анон, перечитай эту фразу ещё пару раз, чтобы проникнуться всей глубиной этой идеи. *//второй камень преткновения – объявление экземпляра класса через нью, об этом позже но пока можно морально подготовиться*

Публичное - то кчему мы можем получить доступ напрямую, откуда угодно, через dream.hp или daisy->speed. При этом public: необходимо писать почти всегда, потому что по умолчанию все считается приватным.

Приватное - то кчему получить доступ можно **только из самого класса**, протектное - из самого класса или наследующих его классов (об этом ниже). Умные люди называют принцип деления класса на приватные и публичные части «инкапсуляцией».

Цвет гривы manecolor объявлен как протектное поле - это значит что я не могу просто так взять и написать где-нибудь color32 cvet = dream.manecolor; (заметьте, я использую **точку** – потому что я *не использовал* new при создании экземпляра dream). Но вот меня критически припёрло получить этот цвет гривы – поэтому я добавлю классу пони публичный метод GetMane() и получу интересующее значение через daisy->GetMane(); (заметьте, я использую **стрелочку** – потому что я*использовал* new при создании экземпляра daisy). Также, поскольку я пишу  
**daisy->**GetMane(); то я тонко намекаю методу GetMane(), что он должен взять именно экземляр daisy, и вернуть мне именно его поле ->manecolor – поэтому я не передаю никаких указателей на свою переменную daisy, как мне пришлось бы делать при работе со структурой. А вот для цвета шкурки писать функцию мне западло, поэтому coatcolor я вообще получить не смогу, пичальбеда.  
Надеюсь, вы все там прониклись всей глубиной идеи о публичных и непубличных полях и методах? Первое и главное что нужно знать в начале работы с классами – никогда не забывай писать public: если хочешь чтобы с твоим классом можно было работать. Я это даже подчеркну, да. Иди перечитай ту фразу про камень преткновения ещё раз :3

Также, допускается вариант неполного объявления: class gryphon; без непосредственно определения того, из чего он состоит. Здесь проводится аналогия с прототипированием функций – когда мы вначале пишем int func(); чтобы мы могли использовать функцию, и только где-то в конце кода пишем уже int func() { return 1; }

А вообще, тут нужно развести немного лирики.

Может сложиться впечатление, что классы были созданы ради того, чтобы жидоевреи из крупных корпораций могли объявлять внутри них священные приватные и протектные переменные, к которым простым юзорам путь закрыт, и толсто троллить оных катавасией из poeben::priebenciya::nihuevina::setvalue(100500); вместо value=100500. Сие есть не истина.

Поскольку практически все библиотеки идут вместе с .h файлами, и именно в хедерах описываются классы, то наглому быдлокодеру совершенно не составит труда своими кривыми копытами заменить все protected: и private: на public:, поэтому для «защиты» эта штука не работает.

Для чего эта штука действительно работает, так это для показания разборчивому небыдлокодеру, что вот этот метод («интерфейс») юзать нужно и можно, а вот в эти десять - лучше не лезть, хоть они и вызываются в том же интерфейсе. Потому что от их неправильного вызова магия программирования может перестать работать

*//Это не значит, что ты, анон можешь менять протектность на публичность безвозбранно – за это анально карают кожаными плётками и ссылают в сибирь, а в сибири с распростёртыми объятиями жду я, вооружённый ещё большим числом кожаных плёток и желанием наставлять всё и вся на путь истиный.*

**О наследовании и публичности, приватности, защищенности**



Наследование, inheritance, deriving, называйте как хотите. Если у меня есть класс пони задаваемый ^ характеристиками, а я хочу описать класс пегаса у которого будет ещё размах крыльев и коэффициент обтекаемости? Вместо того чтобы заново переписывать все понячьи методы и поля я делаю магический пас, и деривлю пегаса из пони,

class pegasus : public pony

{

public:

int wingsize;

/\*void ship(pony \*nyasha)

{

specialsomepony = nyasha;

}\*/

private:

float flightcoef;

public: //дада, разделы доступа могут повторяться и идти в любом порядке

int feathers;

};

pegasus\* art = new pegasus;

Экземпляр art будет обладать как всеми характеристиками пони, так и личными качествами пегаса. Получить доступ можно как к пегасьему art->wingsize так и к понячьему art->speed. К art->manecolor и art->flightcoef получить доступ, логично, нельзя.

Однако! В дело вступает ещё один аспект!

Поскольку в **классе pony** поле specialsomepony было объявлено как **приватное**, то из **класса pegasus** к нему вообще нет доступа. Совсем. В принципе. Даже через art->ship(daisy); несмотря на то что это метод класса, который (как считалось выше) умеет работать с недоступными извне полями. По той простой причине, что в Pegasus НЕТ поля specialsomepony. Пока ты ищешь выше фрагмент про *«протектные поля, доступные из наследных классов, и приватные, недоступные из наследных»*, пегас печально вздыхает и форевералонит.

Если бы мы объявили specialsomepony как протектное поле, то тогда мы всёравно не могли бы ->specialsomepony= , зато уже могли бы ->ship().

Помимо всего прочего, наследование может происходить не только от одного класса, но аж от нескольких, например class changeling : public pegasus, protected unicorn; при этом **все**, чем могут похвастаться единороги, **включая публичные поля и методы**, в экземпляре changeling chrysalis; нельзя будет юзать напрямую. Таким образом,

при публичном наследовании класса Б из класса А: публичные поля и методы А останутся публичными в Б, протектные – останутся протектными.  
При протектном наследовании: публичные и протектные поля и методы А станут протектными в Б.

При приватном же наследовании, все публичные и протектные поля и методы класса А станут приватными в классе Б. Пожалуйста, не путайте это с тем фактом, что приватное поле класса А будет отсутствовать в классе Б – такая путаница может увести в эпическое зависание головного мозга на полчаса, как у автора.

Также можно упомянуть, что множественное наследование на практике используется не слишком часто, а многими языками так и вовсе не поддерживается. Цыплюс фтв.

**Об отличительной особенности методов**

Небольшое отступление от цветных коней, ради углубления в сущность классов. Где-то там выше я ляпнул что-то про «намекаю методу» и «не передаю указателей», при работе с экземпляром. Каюсь. Грешен. Соврал.

class Ctest

{

public:

int a;

void SetValue(int b)

{

a = b;

}

};

void SetValuePointer(Ctest \*subject, int c)

{

subject->a = c;

}

Ctest \*exemplar = new Ctest;

exemplar->SetValue(3);

SetValuePointer(exemplar, 3);

Результатом выполнения обеих функций станет то, что exemplar.a == 3. Казалось бы, подъёба нет, но подъёб есть.

На самом деле ОБЕ функции принимают первым аргументом **указатель на Ctest**. Подъёб в том, что при вызове  
exemplar->SetValue мы, как бы, **неявно** передаем в метод SetValue указатель на экземпляр. А метод, в свою очередь, работает не с a = b; а с очень даже this->a = b;. Зыс, «это» – кейворд, который толсто указывет, что при exemplar->SetValue() работаем мы с полем ИМЕННО ЭТОГО экземпляра класса, вот конкретного, который exemplar. Поэтому можно написать

class Ctest

{

public:

int a;

void SetValue(int b)

{

this->a = b;

}

};

– результат будет таким же.

Уточню – зыс доступен **только** в пределах методов класса, и олицетворяет собой этот самый скрытый параметр экземпляра, который неизбежно присутствует в любом методе. Кроме статических. О коих ниже.

**О статических методах и полях**

Вернёмся к цветным коням.

class unicorn : public pony

{

public:

int hornsize;

**static** int GetCount()

{

return horny\_count;

}

private:

**static** int horny\_count;

};

class alicorn : public pegasus, unicorn

{

private:

color32 crowncolor;

public:

**static** alicorn\* MakeAlicorn(unicorn \*target)

{

alicorn \*princess = new alicorn;

princess->hornsize = target->hornsize;

return princess;

}

};

unicorn \*twilight = new unicorn;

alicorn \*celestia = new alicorn;

alicorn \*marysue;

\*умилительно посмотрел на написанный код\*\*вспомнил что это методичка а не развлекаловка в визуалстудии\*

Итак, в оном коде трижды встречается кейворд static, я его на всякий случай даже жирным выделил. И если со статическим полем на первый взгляд всё более-менее понятно, то вот статический метод – это что-то новенькое, его и рассмотрим.

По факту, статический метод – это глобальная функция в неймспейсе класса (с поправкой на приватность и протектность). Она связана с данным классом, может работать с его приватными и протектными данными, но при этом *не получает адрес экземпляра*, из которого была вызвана. И так как в неё не передаётся this, то мы не можем работать с экземпляром напрямую – для этого нам понадобится вручную передавать указатель на то, с чем мы будем работать.

Если я напишу celestia->MakeAlicorn(twilight); что от этого, у селестии что ли рог увеличится на 20 см бесплатно без смс и регистрации? Канеш нед, вместо этого глупый арт создаст где-то в памяти новый экземпляр аликорна (при этом не присвоив его адрес какому-нибудь указателю – атата утечка памяти), у которого будет рог такой же длины как у твайлайт. Напишем marysue = celestia->MakeAlicorn(twilight); память протекать не будет, но вопрос остаётся открытым – а причём здесь селестия?

Вот тут-то и возникает прелесть работы со статическими методами – нам не важен экземпляр, из которого мы будем их вызывать, нам важен только результат! Поэтому мы с чистой совестью можем ляпнуть  
marysue = alicorn::MakeAlicorn(twilight); а селестию отправить в заслуженный отпуск на недельку.

Вспомним теперь, что у нас есть ещё и статическое поле horny\_count, определяющее поголовье рогатой живности в эквестрии. Собственно, статическое поле делает ровно то же самое что и статическая переменная – инициализируется единожды, а потом используется одно на все экземпляры, по аналогии со статметодами через alicorn::horny\_count.

Статическое поле, объявленное в классе **публичным**, можно интерпретировать как глобальную переменную, объявленную в неймспейсе alicorn.

Уже должно быть ясно, что получить доступ из какой-нибудь своей функции к alicorn::horny\_count я не смогу, потому что в классе единорога (из которого наследуется аликорн) она объявлена приватной. Но даже если я её там сделаю публичной, через **alicorn**::horny\_count она всё равно будет недоступна! По той простой причине, что  
class alicorn : public pegasus, unicorn наследуется из единорога ПРИВАТНО, ибо коварный я написал просто unicorn вместо public unicorn. Соответственно, alicorn::GetCount() я не смогу использовать по той же причине – аликорн попросту не обладает этим методом. Да-да, я всё хожу и хожу вокруг глубины идеи о публичности и приватности. Теперь мы наглядно видим, что если класс Б приватно наследуется из класса А, то никакие поля и методы класса А не будут доступны при работе с классом Б извне. А вот обратиться к GetCount() в пределах MakeAlicorn() можно.

Статические поля, особенно счётчики, может захотеться как-нибудь инициализировать перед запуском программы, например нулём. Сделать это можно через обычное unicorn::horny\_count=0; в начале мейн-функции. Однако если статическое поле является протектным или приватным – то понадобится писать  
**int** unicorn::horny\_count=0; будто бы мы объявляем глобальную переменную, с ОБЯЗАТЕЛЬНЫМ указанием типа поля – в противном случае компилятор ругнётся на ошибку доступа.

**О конструкции и деструкции, а заодно перегрузках**

Ещё один вкусный факт о классах, как будто всего ^ было мало.

Помните, как после создания структуры, мы долго и мучительно непосредственно в коде инициализировали её какими-нибудь стартовыми значениями, чаще всего нулями? Это делалось для того, чтобы если ВДРУГ произойдёт обращение к полю структуры до того, как мы «сознательно» инициализируем его «рабочим» значением, то наша программа не вышла бы на околоземную орбиту со словами «память не может быть read/write». Си, а тем более цыплюс – не дельфи, его задача – просто выделить програмисту память, а что там програмист будет делать с этой памятью – дело десятое. Если ты, анон, не проводил инициализацию (особенно указателей) – забудь о гордой должности «программист», постригись в монахи и сошли себя в сибирь. Я буду ждать тебя там, чтобы сказать «I know that feel, bro».

Класс – родственник структуры, поэтому возникает та же самая проблема инициализации, и даже более того. ООП в принципе построено на объектах, поэтому в коде их может фигурировать великое множество. Каждый объект нужно проинициализировать чем-нибудь. А если в классе 100500 полей? А если создаётся овер 9000 экземпляров этого класса? Писать 9000 раз по 100500 инициализаций как-то лениво, поэтому возникает логичное решение сделать 100500 инициализаций один раз в функции, а потом 9000 вызвать эту функцию. Но 9000 раз вызывать функцию тоже лениво, и вот тут-то в дело и вступают конструкторы. Прокрутимся чуть выше и переопределим класс единорога следующим образом

class unicorn : public pony

{

public:

int hornsize;

static int GetCount()

{

return horny\_count;

}

**unicorn()**

{

hornsize = 17;

horny\_count++;

}

**~unicorn();**

private:

static int horny\_count;

};

int unicorn::horny\_count(0);

unicorn::**~unicorn**()

{

horny\_count--;

};

int main() {

printf("%d\n", unicorn::GetCount());

unicorn \*rarity = new unicorn;

printf("%d\n", unicorn::GetCount());

delete rarity;

printf("%d\n", unicorn::GetCount());

return 0;

}

В глаза просто обязаны броситься два метода у которых имена совпадают с именем класса, а у второго ещё и какая-то загогулина в начале приписана. Знакомься, анон, unicorn() - это конструктор, а ~unicorn() - деструктор, твои лучшие друзья при работе с экземплярами классов.

Если в классе описана функция конструктора, то как только мы создаём экземпляр этого класса – **автоматически** происходит вызов конструктора, в нашем случае: rarity->unicorn(); - причем вручную эту функцию вызывать нельзя, табу. Аналогично, как только мы удаляем экземпляр класса, происходит **автоматический** вызов деструктора. В данной конфигурации, при запуске программы происходит инициализация статического поля horny\_count класса unicorn, после чего каждый новый созданный единорог увеличивает этот счётчик, а каждый удалённый – уменьшает; а заодно появляется с дефолтным значением рога.

Также можно обратить внимание на то, что деструктор я только объявил внутри класса, а определил - уже за его пределами, не забыв указать неймспейс, в котором я это делаю. По хорошему, именно так и надо поступать – в классе оставлять только объявление метода, а определение его давать где-то снаружи, например в соответствующем .cpp (сам класс определяется в .h). И да – конструкторы и деструкторы *не имеют возвращаемых типов*, це не опечатка.

Важное замечание: когда класс содержит какие-нибудь строки, или другие структуры\классы (ввиде указателей), или вообще указатели под которые где-то выделялась память, то опять-таки во имя Бога-Императора и предохранения, то, на что указывают эти указатели, нужно также очищать. Иными словами, делет удаляет ровно то, что непосредственно содержится в данной области – инты, флоаты и т.п., а также УКАЗАТЕЛИ на участки памяти, но **не очищает** эти участки памяти. Для этого-то и нужны деструкторы - чтобы описывать все дополнительные действия, необходимые для удаления экземпляра класса, в его деструкторе, и не париться с ручной очисткой каждый раз перед удалением.

Подводя итог можно сказать что new и delete это, по сути malloc() и free(), проапгрейженные для работы с классами. Они автоматически определяют размер блока памяти, который нужно создать\удалить, а заодно и вызывают конструкторы\деструкторы класса.

Также! На примере конструкторов, я рассмотрю ещё одну полезную вещицу из цыплюса, называемую перегрузкой (overloading).

class zebra

{

const int magikz;

public:

zebra(): magikz(0)

{

printf("<standart constructor %d>\n", magikz);

}

zebra(int val): magikz(val)

{

printf("<nonstandart constructor! %d>\n", magikz);

}

};

zebra \*zecora = new zebra;

zebra \*xenith = new zebra(7);

Тут я опять рушу все шаблоны мироздания и в одном неймспейсе (неймспейс класса, ну вы помните от чего я отталкивался выше (*//ган, сраный критик, перестань агриться на моё мировоззрение*)) объявляю две одноимённых функции, которые принимают разные значения и выполняют разные действия. Тут-то и вступает в действие перегрузка – цыплюс сам определяет, к какой именно функции мы обращаемся (с параметрами, или другими параметрами, или вобще без параметров), и вызывает именно её. Поэтому создание зекоры выведет надпись «стандартный конструктор 0», а создание ксенит – «нестандартный конструктор 7».

А ещё будет полезным обратить внимание на написание : magikz(0). Это стандартный способ инициализациии **константного** поля. Мы не сможем инициализировать его в теле конструктора через magikz = 0; компилятор посчитает, чтомы пытаемся *изменить константу*, а не инициализировать её.

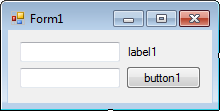
Разумеется перегрузку можно использовать не только в конструкторах, но и вообще для любых функций – главное чтобы эти функции различались списком передаваемых параметров. Это бывает чрезвычайно удобно, когда есть несколько функций, работающих с разными вещами, но выполняющих одно и тоже – например сравнение больших чисел, или экземпляров класса, или структур.

**О дотнете и проблемах с ним связанных**

А теперь я поговорю о том, что собственно и сподвигло меня на написание всего сего трактата – о .NET Framework, который является стандартным комплектом для разработки гуйовых приложений в визуал студии. Знакомство с этим чудом инженерной мысли на первых порах может вызвать разрыв пердака и выход хозяина этого пердака на околоземную орбиту, особенно если он до этого и знать не знал о каком-то там объектном программировании.

Дотнет (. == dot, +NET) построен на объектах целиком и полностью, именно поэтому я и расписывал всё ^ перед тем как вскользь поговорить о самом дотнете. Вскользь – потому что его изучение это совершенно отдельная песня, заниматься этим нужно самостоятельно, по книжкам или методом реверс инжинеринга из чего-нибудь уже готового, например из уроков с olocoder.ru (раздел windows forms), или моих лаб, периодически спрашивая у умных людей «wtf am I doing».

Я не собираюсь останавливаться на том, на каких невидимых маленьких гномиках работает разобранный мною пример, и тем более не берусь описывать стандартные классы дотнета. Я просто постараюсь максимально доходчиво объяснить что в нём происходит, и почему мы совершаем именно такие телодвижения руками чтобы сотворилось то или иное волшебное действо. Итак поехали.



Формочка Form1, на ней сверху вниз объявлены textBox1, textBox2, label1, button1. Задача – сделать так, чтобы при запуске программы в лабел1 вывелось заданное внутри программы дабловое число, в текстбокс1 нельзя было вводить ничего кроме цифр, а также по одному разу букву . и букву е; а при нажатии баттон1 высветилось сообщение содержащее текст из текстбокс2

Открываем кодовую составляющую (пкм по форме – View Code) формочки и зависаем накуй от всей той белиберды что там написано. Развисаем обратно, слушаем и запоминаем.

В самом верху, namespace <имяпроектаздесь> { - всё связанное с моим графоном будет объявлено в неймспейсе соответствующем тому, как я назвал проект.

using namespace System; using namespace System::Windows::Forms; ... - набор неймспейсов, которые дают нам возможность писать чуть меньше букв при работе с функциями и методами объектов дотнета. Систем – в нём описан строковый тип String^. Систем:Виндовс:Формс – в нём описаны всяческие оконные компоненты, в том числе всплывающее окошко с текстами.

public ref class Form1 : public System::Windows::Forms::Form – всё происходит в рамках класса Form1, наследующегося из «системного» класса виндовз-формочек. Как эти виндовз-формочки устроены – дело десятое и нас не интересующее.

**private:** System::Windows::Forms::TextBox^ textBox1; - извне доступа к компонентам формы нет! В функции в другом .cpp ты никак не сможешь использовать Form1->label1->Text! Ты обязан будешь передать этот текст как параметр в функцию!

void InitializeComponent(void) – функция, производящая инициализацию компонентов формы именами, размерами, цветами, текстами и т.д.

#pragma endregion - волшебное слово™, после которого заканчивается сгенерированный визуалом код, и начинается место где нам можно писать свой. Например после него можно написать public: и объявить «типа глобальные» переменные, которые будут использоваться во многих компонентах формы. То что написано между #pragma region и #pragma endregion – лучше не трогать без лишней на то надобности, пусть оно работает как работает.

Возвращаемся обратно к функции void InitializeComponent, и прокручиваем её до того момента где this->label1->Text = L"label1". Это дефолтное присвоение текста лабелу. ТЕКСТА! Текста а не ИМЕНИ! Их ни в коем случае нельзя путать!

Что же здесь происходит – в экземпляре зыс, то есть в формочке, берётся поле label1, т.е. компонент, и его полю Text присваивается стартовое значение. Всё просто. Меняем это присвоение на

= String::Format("chislo {0}, chislo {1}! {0}", 12345, 3.1415);

Здесь мы в классе String (объявленном в неймспейсе System, не забудь), вызываем статический метод Format(), который принимает строку, и переменное число параметров. Формат здесь работает по тому же принципу что и классический printf() – мы передаем строку, а затем переменные, которые нужно распечатать. При этом нам не нужно гадать, какой же спецификатор для печати использовать - %d, %f, %c или что-нибудь ещё – нам нужно просто указать в фигурных скобках номер переменной (начиная с нуля) из последующего списка. При этом первый и последний {0} выше заменятся на одну и ту же 12345.

Возвращаемся обратно на формочку. Выделяем верхний textBox1, в разделе его Properties переключаемся со свойств на события (рыжая молния), дважды жмякаем на KeyPress. Возникает функция private: System::Void textBox1\_KeyPress – приватная, к ней нет доступа извне класса формы!

Как мы можем заметить, метод – войдовый, ничего не возвращает, зато принимает два параметра, sender – экземпляр класса Object, который определён в неймспейсе System; и e – экземпляр класса KeyPressEventArgs, определённый вообще где-то в анальных далях. Е содержит различные данные касательно характера произошедшего события, а именно – нажатой кнопки. Сендер – указывает на объект, из которого, грубо говоря, мы вызвали эту функцию (в данном случае будет передан указатель на textBox1), и иногда может пригодиться.

Теперь, мы записываем в тело нашего новоиспечённого метода такой код:

bool EFound = (textBox1->Text->IndexOf('e') != -1);

bool PFound = (textBox1->Text->IndexOf('.') != -1);

if (e->KeyChar == '.' && PFound)

{

e->Handled = true;

return;

}

if (e->KeyChar == 'e' && EFound)

{

e->Handled = true;

return;

}

if (e->KeyChar == (char)Keys::Back || e->KeyChar == 'e' || e->KeyChar == '.')

return;

if (!Char::IsDigit(e->KeyChar))

{

e->Handled = true;

return;

}

Возможно, это не самый правильный вариант, но он достаточно нагляден. Что же здесь происходит?

* при нажатии клавиши на клавиатуре внутри текстбокса,
* мы определяем, встречаются ли в тексте буквы 'е' и '.' (в поле текстбокса, обращаемся к полю Text, являющемуся экземпляром класса String, и вызываем метод IndexOf() этого класса)
* если была нажата буква '.' (в классе характеристик поле KeyChar отвечает за нажатую кнопку), и при этом она уже встречалась в тексте, то мы передаём списку характеристик информацию о том, что мы обработали нажатие (устанавливаем поле Handled в истину – класс характеристик понимающе кивнёт, и не даст записать букву в текстбокс), и выходим прочь из функции.
* если была нажата буква 'е' – то же самое
* если была нажата кнопка стирания (определённая в неймспейсе Keys как Back – но каким-то левым способом, поэтому вначале мы приведём этот Back к типу char), или кнопка 'е', или кнопка '.' – то тоже вернёмся из функции, потому что эти кнопки нажимать разрешено.
* но! Если мы прошли все эти проверки, и нажатая кнопка не является цифрой (в классе Char есть статический метод IsDigit() для проверки этого) – то мы снова не дадим написать букву в текстбокс и вылетим из функции.

Просто? На мой взгляд - вполне, особенно если не бездумно копипиздить код, а пытаться понять что именно он означает.

Теперь прокручиваемся в самый верх, где юзинг неймспейсы, и после последнего из них (System::Drawing) пишем функцию

void DebugPrint(String^ text)

{

MessageBox::Show(text);

}

В неё мы передаём строку, в виде экземпляра класса String, объявленного в неймспейсе System, не устаю напоминать. Но также я сакцентирую внимание на том, что мы пишем String**^** – крышка это здешний аналог указателя. То есть при вызове функции мы будем использовать не String stroka; а String^ stroka = gcnew String; поскольку дотнет не разрешает явное создание экземпляров классов – только через gcnew, так за выделенной памятью начинает следить сборщик мусора. // *Зря я это сказал, вон у кого-то в аудитории мозг взорвался...*

Далее, мы вызываем Show() – статический метод класса MessageBox, в который передаём нашу строку – результатом станет выпрыгивание на экран мессаджбокса с данной надписью.

Наконец мы вновь возвращаемся на нашу формочку, выделяем кнопочку button1, и аналогичным образиком создаём для неё событийный методчик Click. Появляется функция System::Void button1\_Click, опять-таки принимающая аргументами указатель на то, откуда была вызвана (в нашем случае – на кнопку) и список всяких характеристик.

В неё мы записываем такой код:

MessageBox::Show("вызываем печать...", "Информация");

DebugPrint(textBox2->Text);

Прошу заметить, что в дебагпринте я передавал в Show одну строку, а здесь две. Мы наблюдаем перегрузку функций в действии – метод шоу может принимать как (строку1), так и (строку1, строку2), и даже (строку1, строку2, наборкнопок). Строка1 будет использоваться в теле мессаджбокса, строка2 – в его заголовке, а наборкнопок определит, какие кнопки будут доступны на мессаджбоксе (ОК, Да, Нет, Отмена).

Всё, можно компилировать и запускать. Если всё сделано правильно – то выполняться будут ровно те действия, которые я описал справа от картинки с формочкой. Если неправильно – прочитай методичку ещё раз, методичка сама себя не прочитает. Остальная работа с формами происходит по аналогии с дельфи, в сопровождении какой-нибудь книжки по всему этому, или уроков, или хотя бы справочной системы MSDN. Я кончел.