Объектно-ориентированное программирование с использованием языка C++

ШАГ компьютерная АКАДЕМИЯ

Урок №14

Объектно-ориентированное программирование с использованием языка C++

Содержание

Введение в обработку исключительных ситуаций

В самом начале нашего знакомства с программированием, мы говорили о том, что бывает два основных вида ошибок: ошибки на этапе компиляции и ошибки на этапе выполнения. Напомним, что первый вид подразумевает ошибку синтаксиса, которую, к счастью, не пропустит компилятор. Второй же тип ошибки, является следствием неправильной работы программы. Именно об этом типе ошибок мы и поговорим в этом уроке.

Вполне естественно, что при написании программы, мы стараемся предусмотреть возможные ситуации, при которых, во время выполнения, программа отработает некоректно. При этом, наверняка, вы используете один из стандартных, общепринятых вариантов, например:

1. ***Функция, в которой может произойти ошибка, сама её отслеживает, прекращает работу и выдаёт на экран сообщение.*** Недостаток этого метода заключается в том, что программист, использующий функцию, при желании обработать ошибку самостоятельно и совершить в этом случае какое-либо свое действие, не сможет этого сделать. Просто потому, что функция может лежать в библиотеке и доступа к ней программист не имеет. Вследствие чего, нет возможности избавиться, например, от стандартного сообщения об ошибке.

2. ***Функция, в которой может произойти ошибка, прекращает работу и возвращает код ошибки на место своего вызова в основной программе.*** Определенно, данный способ является решением задачи. Однако, если функция может сгенерировать несколько разных ошибок в разных ситуациях, то соответственно создателю необходимо придумать некоторое количество различных кодов ошибок и описать их в документации, что тоже не очень удобно.

Что же делать, скажете вы?! Неужели у всех методов "отлова ошибки" одни сплошные недостатки?! Конечно же это не так. Язык C++ предоставляет нам с вами отличное средство, которое поможет плавно уйти от недостатков, описанных выше и решить проблему с ошибками на этапе выполнения раз и навсегда. Это средство носит название — обработка исключений. Итак:

Исключение (exception) — исключительная ситуация, генерируемая программой, например, в момент возникновения ошибки.

Как вы в последствии убедитесь, механизм обработки исключений является прямым устранением всех вышеуказанных недостатков. То есть, программист, который использует функцию, реализующую его, сам определяет каким образом обработать ошибку. Далее в уроке мы рассмотрим ключевые слова используемые для реализации данного механизма.

Примечание: Следует отметить, что механизм обработки исключений может быть использован не только при обработке ошибочных ситуаций, но и в тех случаях, когда программисту необходимо выделить в программе какой-либо блок кода.

Реализация механизма. Ключевые слова try, catch, throw

Итак, для организации механизма обработки исключений в C++ используются три ключевых слова:

- **try** (*контролировать*) — с помощью фигурных скобок отделяет определенную область кода, в которой может быть сгенерированно исключение;
- **throw** (*бросать*) — оператор, который в момент возникновения исключительной ситуации генерирует исключение и выводит программу из "критического" блока;
- **catch** (*поймать*) — оператор, который позволяет "поймать" конкретное исключение и проанализировав его выдать нужную реакцию.

Синтаксис организации механизма:

```
try{
    блок кода;
    ...
    throw выражение_определенного_типа;
    ...
}

catch(тип_исключения имя)
{
    блок_анализа;
}
```

Анализ синтаксиса

1. В одном блоке try может быть несколько throw, возбуждающих различные исключения.
2. Когда срабатывает throw, то с помощью выражения, находящегося в нём в памяти образуется объект, который имеет тип совпадающий с типом выражения. После его формирования throw передает управление объектом за пределы блока try. Где объект попадет в соответствующий блок catch, который также имеет схожий тип данных. Именно в catch и происходит последующий анализ.
3. Блоков catch, также как и throw может быть несколько, однако их количество не обязательно должно совпадать, то есть никак не зависит друг от друга.
4. Один catch должен отличаться от другого по типу данных, то есть не может быть двух catch с одинаковым типом.

Для того, чтобы вам было легче понять данный материал, все последующие особенности мы будем рассматривать на простых примерах.

Пример 1. *Простой пример на исключение.*

```
# include <iostream>
using namespace std;

void main(){
    //Данная фраза появится на экране
    //сразу же при запуске программы
    cout<<"\nStart!!!\n";
    //вход в исключительный блок
    try{
        //выполнение показа на экран
        cout<<"\nBefore!\n";

        //прерывание блока try
        //возбуждение исключения типа int
        //выход из try
        throw 100;

        //данная фраза не появится
        //на экране никогда
        cout<<"\nAfter!\n";
    }

    //блок для отлова исключения типа int
    //значение 100 попадёт в g
    catch(int g){

        //результат анализа исключения
        cout<<"\nException!!!!\n";
    }

    //показ завершающей строки на экран
    cout<<"\nBye!!!\n";
}


_____________________________

Результат работы программы:
_____________________________
Start!!!

Before!

Exception!!!!

Bye!!!
```

Пример 2. *Исключение деления на нуль.*

```
# include <iostream>
using namespace std;

void main(){

    //вход в исключительный блок
    try{
        //создание переменных, запрос
        //и ввод данных с клавиатуры
        float a,b;

        cout<<"\nPut digit a:\n";
        cin>>a;

        cout<<"\nPut digit b:\n";
        cin>>b;

        //проверка делителя на нуль
        if(b==0){

            //если делитель нуль – возбуждение
            //исключения типа float
            //выход из try
            throw b;

        }

        //иначе блок успешно завершен
        cout<<"\nResult = "<<(a/b)<<"\n\n";
    }

    //сюда throw передаёт значение типа float
    catch(float g){

        //анализ и сообщение об ошибке
        cout<<"\nError – Divide by "<<g<<"\n\n";
    }
}
```


Пример 3. *Пример на несколько исключений.*

```
# include <iostream>
using namespace std;

void main(){

    //блок try способный возбудить
    //три исключения
    try{

        //указатель и размер
        //динамического массива
        int*ptr=0;
        int size;

        //ввод размера
        cout<<"\nPut size:\n";
        cin>>size;

        //если размер выходит за пределы
        //заданно го диапазона
        if(size<1||size>500)
            //возбуждаем исключение типа char*
            //блок try обрывается
            throw "\n\nErr Size!!!\n\n";

        //иначе создаем массив
        ptr=new int [size];

        //проверяем, выделилась ли память
        if(!ptr)
            //если нет возбуждаем
            //исключение типа char*
            //блок try обрывается
            throw "\n\nErr Memory!!!\n\n";
```

```
        //иначе создаем тестовую переменную a
        //осуществляем ввод данных с
        //клавиатуры
        int a;
        cout<<"\nPut digit a:\n";
        cin>>a;

        if(a==0)
            //если a равно 0 возбуждаем
            //исключение типа int
            //блок try обрывается
            throw a;

    }

    //отлов всех исключений типа int
    catch(int s){
        cout<<"\nError - A = "<<s<<"\n\n";
    }

    //отлов всех ислючений типа char*
    catch(char*s){
        cout<<s;
    }
}
```

Пример 4. *Универсальный catch.*

Если срабатывает throw с типом данных не имеющим эквивалента в catch, то программа будет автоматически закрыта. При этом произойдёт ошибка и (при нажатии кнопки ПРОПУСТИТЬ) на экране отобразится надпись:

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

Чтобы избежать данной ситуации обычно используют универсальный catch. Синтаксис его таков **catch(...){анализ;}**. Обращаем Ваше внимание на то, что данная конструкция должна быть самой последний среди всех перечисленных catch!!! Если для какого-то исключения не будет найден соответствующий catch, выполнится универсальная конструкция.

```
# include <iostream>
using namespace std;

void main(){

    //вход в блок
    try{
        //объявление и инициализация
        //переменной
        int a;
        cout<<"\nPut digit a:\n";
        cin>>a;

        //если переменная равна нулю
        if(a==0)
            //генерация исключений
            //типа char*
            throw "URRRRRRA!!!";
    }
    //универсальный catch
    catch(...){
        cout<<"\nSome Error!!!!\n\n";
    }
}
```

Пример 5. *Генерация исключения внутри функции. (Вся обработка находится внутри функции).*

```
# include <iostream>
using namespace std;

void Some(){

    //старый добрый анализ деления на нуль
    int a;
    int b;

    try{

        cout<<"\nPut digit a:\n";
        cin>>a;
        cout<<"\nPut digit b:\n";
        cin>>b;

        //если делитель равен 0
        //генерируем исключение
        if(b==0)
            throw "\tZerro!!!\n";

    }
    //отлов исключения и анализ
    catch(char*s){
        cout<<"\n Error!!!!"<<s<<"\n\n";
    }
}

void main(){

    cout<<"\nFirst!!!\n";
    //вызов функции, содержащей исключение
    Some();
    cout<<"\nSecond!!!\n";
}
```


Пример 6. *Генерация исключения внутри функции.*
(Внутри функции находится только возбуждение исключения).

```
# include <iostream>
using namespace std;

void Test(int t){

    //вход в функцию
    cout<<"\nInside!!!\n";

    if(t==2){
        //возбуждение исключения
        throw "\nError – t is 2\n";
    }

    else if(t==3){
        //возбуждение исключения
        throw "\nError – t is 3\n";
    }
}

void main(){
    //вызов функции заключается в тело try
    try{
        //вызовы
        Test(4);
        Test(2);
    }

    //отлов исключений
    catch(char*s){
        cout<<"\n\n"<<s<<"\n\n";
    }
}
```

Пример 7. *Повторная генерация исключения.*

Существует возможность внутри catch сгенерировать исключение еще раз:

```
# include <iostream>
using namespace std;

void Test(){
    try{
        //возбуждение исключения
        //первый раз
        throw "\nHello!!!\n";
    }
    //вход в анализ
    catch(char*s){

        cout<<"\n\nException!!!\n\n";

        //передача исключения следующему catch
        throw;
    }
}

void main(){
    cout<<"\nStart\n";
    try{
        //вызов функции с повторным исключением
        //заключается в блок try
        Test();
    }
    //отлов объекта, посланного повторно
    catch(char*p){
      cout<<p;
    }
}
```

Здесь мы постарались охватить все особенности работы с исключениями. Надеемся, что вы прониклись необходимостью их использования.

Пространства имен и их использование

"Представьте, что Вы отправились в неизведанные земли и в пути приобрели последователей. Последователи устали и решили основать новый город на берегах великой реки. Но Вы не знаете как назвать поселение, поскольку не знакомы с другими городами этой страны и не желаете выбирать уже существующее имя. Очевидное решение — провозгласить новое государство, чтобы имя нового поселения было гарантированно уникальным в его пределах." — этот пример является в специальной литературе классическим для наглядного разъяснения темы данного раздела урока. Дело в том, что в программировании точно также можно давать разным объектам одинаковые имена, разделяя их так называемыми пространствами имен. Познакомимся с определением:

Пространство имён (*namespace*) — в программировании именованная, либо неименованная область определения переменных, типов, констант. Пространства имён используются для отграничения набора данных и функций в один блок, чтобы исключить конфликты с другими наборами функций. Иначе говоря, пространства имен используются для разбиения области видимости на несколько зон.

Объявление пространства имен

Синтаксис объявления пространства имен чем-то напоминает синтаксис объявления класса. Условно — это обозначение имени для области видимости, в которую будут входить компоненты пространства:

```
namespace имя
{
    перечень_данных;
}
```

Применение пространства имен

Два одинаковых идентификатора в одной области видимости существовать не могут. Например, в одной программе невозможно создать две функции fire() с одинаковыми прототипами, одна из которых зажигала бы факел, а другая производила стрельбу из пистолета, кроме как с помощью пространства имен.

Пространства имен позволяют разместить две функции в различных областях видимости, не создавая при этом проблем. Кроме того, можно разместить все функции, связанные с определённым направлением, в отдельном пространстве имен. Например, функция fire(), зажигающая факел, попадет в пространство имен exploration (исследование) наряду с другими функциями исследования, а функция fire(), ответственная за стрельбу из пистолета, попадёт в пространство имен combat (боевое).

```
namespace combat{
    void fire(){
            cout<<"Vistrel";
        }
    }

    namespace exploration{

        void fire(){
            cout<<"Ogon`";
        }

    }
```


Для осуществления доступа к компонентам пространства имен используется оператора разрешения области видимости. Синтаксис доступа выглядит так:

```
#include <iostream>
using namespace std;

namespace combat{
    void fire(){
        cout<<"Vistrel";
    }
}

namespace exploration{

    void fire(){
        cout<<"Ogon`";
    }
}

void main(){

    combat::fire();
    exploration::fire();
}
```

Глобальное пространство имен

Глобальное пространство имен — это название области видимости самого высокого уровня, то есть той, где существуют глобальные переменные.

Чтобы обратиться к члену глобального пространства, часто приходится пользоваться оператором разрешения области видимости (::). Это необходимо, если существуют совпадающие имена в локальном и глобальном пространствах, поскольку по умолчанию всегда выбирается переменная с наименьшей областью видимости. Синтаксис обращения к члену глобального пространства следующий:

```
::глобальный_член;
```

Повторные объявления пространств имен

Если два пространства имен имеют одинаковые имена, второе считается логическим продолжением первого. И компилятор считает идентичными конструкции следующего вида:

```
namespace x
{
    func1(){}
}

namespace x
{
    func2{}
}
```



```
и

namespace x
{
    func1(){}
    func2(){}
}
```

C++ помещает оба пространства в одну область видимости при компиляции программы, это означает, что в дублирующихся пространствах имен не могут существовать члены с одинаковыми именами.

Прямой доступ к пространствам имен

Если Вы уверены, что пространство не содержит дублирующих членов, то можете произвести слияние его с глобальным пространством имен. Благодаря этому исчезает необходимость постоянно набирать идентификатор пространства имен и оператор разрешения области видимости. Существует два метода прямого доступа к пространству имен:

1. Объявление using.
2. Директива using.

Объявление using

Объявление using показывает, что работа, будет производиться с определенным членом области видимости более низкого уровня. В результате отпадает необходимость в явном указании области видимости. Синтаксис объявления using следующий:

```
using имя_пространства::член;

Пример использования:

#include <iostream>
using namespace std;
namespace dragon
{
    int gold=50;
}

void main()
{
    using dragon::gold;
    cout<<gold;
}
```

Директива using

Директива using показывает, что работа будет производиться со всеми членами конкретного пространства имен. Директива работает также, как и объявление, с той лишь разницей, что обеспечивает прямой доступ ко всем членам пространства. Воспользовавшись ею, вы избавляетесь от необходимости уточнять имена членов этого пространства до конца программы. Перед её применением следует убедиться, что во включаемом пространстве имен отсутствуют идентификаторы, дублирующие идентификаторы глобального пространства. Синтаксис директивы using таков:

```
using namespace имя_пространства;
```


С примером использования этой конструкции вы уже сталкивались, он присутствует во всех написанных вами программах:

```
using namespace std;
```

Создание безымянных пространств имен

Давайте теперь разберёмся, каким образом можно гарантировать корректность имени, присвоенного пространству, ведь может получиться, что неким пространствам имён были присвоены одинаковые имена. Как избежать подобного конфликта, спросите Вы?! Как это ни странно — создав пространство имен без имени!!! Это связано с тем, что если создать безымянное пространство, C++ автоматически даст ему уникальное имя, которое мы с Вами, естественно, не увидим. Однако, то что имя будет уникальным — факт неоспоримый. Синтаксис объявления безымянного пространства имен следующий:

```
namespace
{
    члены;
}
```

В такой ситуации C++ позволяет обращаться к компонентам безымянного пространства через операцию :: без указания имени. Вам нет необходимости объявлять это пространство (да и возможности нет, так как имени Вы не знаете). Неявное объявление using namespace добавляется автоматически после объявления самого пространства имен. Так что, в итоге, код построится таким образом:

```
namespace a //предположительно имя полученное
            //автоматически (неявно)
{
    члены;
}
using namespace a; //автоматически прописавшееся
                   //объявление
```

Рассмотрим пример работы с безымянным пространством имен:

```
#include <iostream>
using namespace std;

namespace
{
    void func(){cout<<"::func"<<"\n";}
}

void main()
{
    ::func();
}
```

В заключение, следует отметить, что безымянные пространства носят локальный характер и могут быть использованы только в том файле, в котором они объявлены.

Экзаменационные задания

Для реализации всех заданий использовать ООП.

1. Реализация карточной игры — "Покер".
2. Написать программу, которая разгадывает японский кроссворд под названием "Судоку". Пользователь вводит начальные значения уже существующие в кроссворде, который необходимо разгадать, и программа выдает на экран готовый результат.
3. Написать программу, реализующую электронный органайзер. Реализовать возможности добавления, удаления, редактирования и хранения данных. Предусмотреть обработку всех возможных ошибок.