

# **Обектно-ориентирано програмиране (записки)**

- **Марина Господинова**
- **Илиан Запрянов**

# Тема 08. Статични член-дани. Изключения.

## Static

- **Static локални променливи (статични променливи в тялото на функция)**
  - държи се в паметта на **глобалните/статичните променливи**
  - **static** променливите се създават в началото на програмата и се унищожават в края на програмата (а не в края на scope-а)
  - инициализира се **само веднъж** - при първото влизане в съответния scope и запазва стойността си дори след като излезе от scope-а

```
#include <iostream>

void increment()
{
    static int valueStatic = 0; //инициализира се само при първото викане
                                //на функцията [increment], след това единствено
                                //променя стойността (++value) и

    int valueAuto = 0;
    ++valueStatic;
    ++valueAuto;

    std::cout << valueStatic << " " << valueAuto << std::endl;
}

} //тук се освобождава паметта, заделена за [valueAuto]
  //(след края на scope-а)
```

```
int main()
{
    increment(); //първо викане на [increment]
    //01. [valueStatic] се инициализира
    //02. добавя се единица към [valueStatic]
    //03. [valueAuto] се инициализира
    //04. добавя се единица към [valueAuto]
    //=> valueStatic == 1 && valueAuto == 1

    increment(); //второ викане на [increment],
    //#[valueStatic] е вече инициализирана
    //(тъй като е static, няма да се инициализира
    //наново, както сме свикнали да очакваме)
    //=> стъпка [01] ще бъде пропусната =>
    //02. добавя се единица към [valueStatic]
    //03. [valueAuto] се инициализира НАНОВО
    //04. добавя се единица към [valueAuto]
    //=> valueStatic == 2 && valueAuto == 1

    increment(); //3 1

    //напомняме, [static] променливите се държат в паметта на глобалните/статичните променливи,
    //но това НЕ означава, че valueStatic е глобална променлива

    //valueStatic++; //![X] когато променлива е декларирана като static в рамките на функция,
    //видяхме, че запазва стойността си между различните извиквания на функцията,
    //(тоест не се инициализира наново на всяко извикване)

    //![NO] въпреки това, тя е достъпна само в блока, в който е декларирана
    //(в случая скоупа на функцията [increment])

    return 0;
}

//тук се освобождава паметта, заделена за [valueStatic]
//(след края на програмата)
```

## Димитриев

```
f()
{
    A obj;
    static B obj2; //static -> общ за всички
                    //извиквания на функцията
    C obj3;

    ...
    //логика
    ...
}

int main()
{
    f(); //ще извика конструкторите на [A], [B], [C],
    //тъй като са локални, паметта ще се освободи в края на scope-а
    //![НО] помним, че static променливите се изтриват в края на изпълнението на
    //програмата => ще се извикат само деструкторите на [A], [C]

    f(); //помним, че статичните данни се инициализират само първия път,
    //след което запазват старата си стойност на всяко извикване
    //=> на второто извикване на функцията, [B] вече съществува
    //=> ще извикаме само конструкторите на [A], [C] и съответно
    //техните деструктори в края на scope-а на функцията

    return 0;
} //крайт на програмата => тук ще се унищожи [B]
```

**Създаване на обекта: при първото извикване на функцията**

**Изтриване на обекта: при излизане от програмата**

## Static функции

- **видимост** - когато функция е декларирана като **static** (извън всички класове), това ограничава нейната видимост само до файла, в който е дефинирана. Това предотвратява функциите от други файлове да извикват тази функция, дори ако имат функция със същото име.

**Накратко:** обвързва се с **ЕДНА** компилационна единица (**.obj файл**) и не може да се използва от други файлове

Досега:

```
#include <iostream>
void plovdiv();
int main()
{
    plovdiv();
}
```

```
#include <iostream>
void plovdiv()
{
    std::cout << "Plovdiv e mnogohubav grad." << std::endl;
}
```

От разделната компилация и **линкинга** научихме, че по време на процеса на **линкинга** **03.cpp** ще открие дефиницията на функцията **plovdiv()**, която извиква във файла **02.cpp**. Тоест ще се отпечата съответния изход.

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "Plovdiv e mnogohubav grad." << std::endl;
6 }
7
```

Когато обаче кажем във файла **02.cpp**, че функцията е **static**, то тогава казваме, че тази функция може да се използва **CAMO** в този файл => Няма да бъде открита от файла **03.cpp** => **Linking** грешка, тоест не е открита дефиниция за тази функция

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "plovdiv 1" << std::endl;
6 }
7
```

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "plovdiv 2" << std::endl;
6 }
7
8 int main()
9 {
10
11     plovdiv();
12 }
```

За да оправим настъпилата **Linking** грешка, то трябва да напишем дефиниция за функцията **plovdiv()** и във файла **03.cpp**. След компилация ще се отпечата **“plovdiv 2”**, защото ще намери тази дефиниция. (вече казахме, че **static** функциите са достъпни **САМО** във файла, в който се намират)

Можем да забележим, че имаме две функции с **ЕДНАКВИ** имена и **ЕДНАКВИ** параметри, което не беше възможно досега, тоест можем да направим извода, че:

- **static** функциите ни помагат с **капсулатията** една функция да не може да бъде използвана от други файлове
- позволява ни декларацията на функция със **същото име и същите параметри** в друг файл (друг вариант е **namespace { f(); }**)

## Static член-променливи

01. не е обвързана с конкретен обект, а с целия клас
02. всички обекти от класа използват една и съща инстанция
03. инициализира се извън класа

```
#include <iostream>

class A
{
public:
    static int x;

public:
    A() = default;
};

int A::x = 10; //статичните член-данни на класовете се инициализират извън класа,
               // (в глобалното пространство на файла, където класът е дефиниран)
               //
               //това се дължи на факта, че статичните член-данни принадлежат на класа,
               //а не на специфичен обект на този клас
               //
               //по този начин, те съществуват независимо от всякали обекти, създадени от класа,
               // (те са външни) и са споделени от всички инстанции на класа.

int main()
{
    A obj1; //създаваме ДВА РАЗЛИЧНИ обекта
    A obj2;

    std::cout << obj1.x << " " << obj2.x << std::endl; //спрямо инициализацията -> 10 10

    obj1.x++; //увеличаваме член-данната [x] на [obj1] с 1

               //тъй като вече разбрахме, че статичните член-данни са свързани с КЛАСА
               //(тоест всички обекти от този клас, а НЕ СЪС СПЕЦИФИЧЕН такъв, то тогава следва,
               //че с увеличаването на член-данната [x] на [obj1], ще се увеличи
               //и член-данната [x] на [obj2], тъй като де факто [x] е обща член-данна
               // (споделена от всички инстанции на класа) за двата обекта

    std::cout << obj1.x << " " << obj2.x << std::endl; //11 11
}
```

## Димитриев

```
//04.h

#include <iostream>

class A
{
    int a = 0;
public:
    A() = default;
};

class B
{
    int b = 0;
public:
    B() = default;
};

class X
{
    A obj;
    static B obj2; // [01] глобален обект, скрит в клас X
                    // (енкапсулация в клас X)
                    // терминът енкапсулация в случая означава:
                    // скриване на вътрешните данни на обекта от външния свят

                    // [02] не се дефинира в конструктора, а ИЗВЪН него, тоест
                    // конструктора на [X] няма да извика конструктора на [B]
                    // деструктора на [X] няма да извика деструктора на [B].
                    // [B] се изтрива сам след изпълнението на програмата

                    // [03] не се обвързва с конкретен обект от този клас,
                    // а с всички обекти от този клас

                    // [04] статичните член-данни не са нито агрегация, нито композиция

public:
    X() = default;
};
```

```
#include "04.h";

B X::obj2; // дефинираме static член-данната
            // чрез конструктора на [B]

int main()
{
    // В X::obj2; // [X] обръщаме внимание, че статичните член-данни
                // са глобални => не можем да инициализираме
                // [obj2] в scope-а на main
}
```

## ![B] НЕ влияе на големината на обекта [X]

```
//04.h
#include <iostream>

class A
{
    int a = 0;
public:
    A() = default;
};

class B
{
    int b = 0;
public:
    B() = default;
};

class X //можем да кажем, че [X] играе ролята на namespace
{
    //за [obj2]

public:
    A obj;
    static B obj2;

public:
    X() = default;
};
```

```
//04.cpp
#include "04.h";

int main()
{
    X obj;

    //достъпване на статична член-данна обект

    obj.obj2; //можем, но не го правим

    X::obj2; //просто глобален обект (както при namespace)

    return 0;
}
```

## Static член-функция

01. не е обвързана с конкретен обект, а с целия клас
02. използва се за достъпване на статичните член-данни
03. няма указател **this**
04. не е нужен обект, за да се достъпи

```
#include <iostream>

class Test
{
public:
    int z = 5;
    //static int r = 10; // [X] не можем да инициализираме static член-данна в класа

    static const int P = 20; // ОСВЕН, ако не е константа
private:
    static int x; // статична член-данна
    int y;

public:

    int getXStatic()
    {
        return x;
    }

    int getY()
    {
        return y;
    }
```

```
static void f()
{
    x *= 2; // може да достъпва само статичните член-данни

    //y += 2;      // [X] НЕ можем да достъпваме НЕстатични член-данни
    //this->y;    // [X] в статичните функции нямаме указател [this]
    //getY();      // [X] НЕ можем да достъпим НЕстатични член-функции
    //this->getY(); // [X] в статичните функции нямаме указател [this]
    //z++;        // [X] НЕ можем да достъпваме НЕстатични член-данни

    //x += p; //можем да достъпим СТАТИЧНИТЕ член-данни [x], [p]
}

static void r()
{
    f(); // в СТАТИЧНИ член-функции можем да извикваме
          // единствено СТАТИЧНИ член-функции

    //g(); // [X] НЕ можем да извикваме НЕстатични член-функции

    std::cout << "r" << std::endl;
}

void g()
{
    r(); // в НЕстатични член-функции можем да извикваме
    f(); // както НЕстатични член-функции, така и СТАТИЧНИ член-функции
    k();
}

void k()
{
    std::cout << "k" << std::endl;
}
};

int Test::x = 3; // трябва да я инициализираме извън класа
```

```
int main()
{
    Test::f(); // не ни трябва обект, за да я извикаме

        // забелязваме, че [f] променя статичната член-данна [x] (x *= 2),
        // вече знаем, че статичните член-данни са общи за класа (всички обекти Test)

    Test a;      //=> новосъздадените обекти [a], [b] ще имат [x] == 3 * 2 == 6
    Test b;

    std::cout << b.getXStatic() << std::endl; //6

    a.f(); //макар и да фикаме [f] върху обекта [a], то това не променя факта,
           //че промяната на [x] за [a] е промяна на [x] за всички обекти от тип [Test]

    std::cout << b.getXStatic() << std::endl; // [x] == 6 * 2 == 12

}
```

## Димитриев

```
#include <iostream>

class Z
{
    int x;
    static int y;
public:
    void g() //НЕстатичните функции имат достъп
    {           //до всички член-данни и член-функции

        f();
        x;
        y;
    }

    static void f() //СТАТИЧНИТЕ член-функции имат достъп
    {               //до СТАТИЧНИ член-данни и член-функции

        //СТАТИЧНИТЕ функции са външни функции (инициализират се извън класа)
        //(подобно на friend)
        //тоест не ни трябва обект, за да я извикаме

        y;
        //x;   // [X]
        //g(); // [X]
    }
};

int main()
{
    Z::f(); //можем да викаме СТАТИЧНИ член-функции без да създаваме обект, т.е.
            //СТАТИЧНИТЕ член-функции са ГЛОБАЛНИ функции част от класа

    //g(); // [X] НЕ можем да викаме НЕстатични член-функции без да създаваме обект

    return 0;
}
```

def| **Статичен клас** - клас, който има само статични член-данни

# Изключения / Exceptions

**def| Exception** - сигнал, че има проблем

## Приложение

Изключенията в C++ помагат при неочеквани или грешни условия, които се появяват по време на изпълнението на програмата. Чрез тях позволяваме на програмата да обработва грешки по зададен от нас начин.

Ключови думи, които ще използваме:

**throw <обект>**: когато възникне проблем в програмата, може да се "хвърли" изключение с помощта на ключовата дума **throw**, последвана от **<обект>**. Този **<обект>** може да бъде от всякакъв тип, но обикновено е от класа **std::exception** (напр. можем да "хвърлим" като **<обект>** числото **38**, както може и да "хвърлим" **std::runtime\_error**).

**try**: след ключовата дума **try**, отваряме блок (**scope**), в който слагаме кода, който според нас е проблемен и за когото се прилагат изключенията. Тоест, в рамките на този **scope** се извършват операциите, които могат да хвърлят изключение.

**catch**: след блока **try** следват **един или повече** блокове **catch**, които улавят хвърлените изключения. Всеки блок **catch** обработва определен тип изключения. Кодът в блока **catch** определя как да се реши проблемът, предизвикан от изключението.

```
#include <iostream>

void error(const bool isValid)
{
    if (isValid)
    {
        std::cout << "no error" << std::endl;
        return;
    }
    else
    {
        //с ключовата дума [throw] сигнализираме, че
        //е възникнал някакъв проблем, а след него
        //"хвърляме" изключението, което искаме да обработим

        throw '2'; //char -> a = 30
        //throw "ASD"; //char масив -> a = 50
        //throw 412; // int -> [...] -> a = 60
    }
}
```

```

{
    int a = 0;

    try
    {
        //scope-a, в който слагаме кода,
        //в който може да възникне някакъв проблем

        error(true); //в първото извикване на функцията подаваме, че [isValid] е <true>
                    //=> всичко е наред, ще отпечата, че няма грешка и ще излезе от функцията

        error(false); //във второто извикване на функцията подаваме, че [isValid] е <false>
                    //=> нещо НЕ Е наред => ще "хвърлим" някаква грешка,
                    //за да сигнализираме, че нещо не е наред
    }

    //конструкцията, която обработва различните типове изключения
    //(влизаме тук, ако изобщо в [try] е хвърлено изключение

    catch (char ch) //ако обектът хвърлен след [throw] е от тип [char]
    {
        //помним синтаксиса throw <обект>
        a = 30;
    }
    catch (const char* str) //ако обектът хвърлен след [throw] е МАСИВ от тип [char]
    {
        a = 50;
    }
    catch (...) //с [...] се обозначава всичко, т.e int, char, bool, double, float...
                //дори ако е хвърлена инстанция на даден клас, но тъй като вече имаме
    {
        //#[catch] за тип [char] напр., теоретически никога няма да влезе в този при
        //хвърлен [char], така че може да се каже, че хваща всичко останало

        a = 60;
    }

    std::cout << a << std::endl;
    return 0;
}

```

## Приложение/Димитриев (github)

[https://github.com/Angeld55/Object-oriented\\_programming\\_FMI/tree/master/Week%2008#exception-handling](https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2008#exception-handling)

### Накратко

в **try** блока се намира проблемният код

ако се хвърли грешка, останалата част от **try** блока не се изпълнява

в **catch** блоковете грешката се обработва

може да има няколко **catch** блока, влизаме в този, който приема като параметър обекта, който е хвърлен (ако се хване грешката, проверката надолу спира)

**catch ( ... )** хваща всичко, ако до момента не е хванато - слага се накрая

## Особености при инстанции на клас (Димитриев лекция)

```
#include <iostream>

// в следващите примери, ще използваме следните класове [X], [A], [B], [C]
// с цел визуализация, тъй като на лекции ги ползвахме наготово

class X
{
public:
    X()
    {
        std::cout << "X()" << std::endl;
    }

    ~X()
    {
        std::cout << "~X()" << std::endl;
    }
};
```

```
class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }
    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};

class B
{
public:
    B()
    {
        std::cout << "B()" << std::endl;
    }
    ~B()
    {
        std::cout << "~B()" << std::endl;
    }
};

class C
{
public:
    C()
    {
        std::cout << "C()" << std::endl;
    }
    ~C()
    {
        std::cout << "~C()" << std::endl;
    }
};
```

```

void g()
{
    X obj; //X()

    throw 37; //хвърляме изключение (намерили сме проблем с кода)
        //функцията g() спира изпълнението си веднага при хвърлянето,
        //тоест НЯМА да създаде [obj2]
        //
        ^
A obj2; //<-----| //след спирането на изпълнението, g() започва да търси
//кой може да обработи грешката, като по пътя вика деструкторите
//на всички обекти, създадени във функцията => се вика деструктора на [X]

//грешката е прехвърляне на отговорност, тоест
//в момента стека ни приема следния вид

//g()
// | функцията g() хвърля изключение, което се прехвърля в f(),
// v тъй като g() е извикана в f()
//f()
// | програмата вижда, че f() не може да обработи изключението, хвърлено от g()
// v и прехвърля отговорността изключението да бъде обработено нататък (на main)
//main()

}

void f()
{
    A obj; //A()
    B obj2; //B()

    g(); //извикваме функцията g()

    //хвърлено е изключение (намерили сме проблем с кода)
    //от g() => функцията f() спира изпълнението си веднага при хвърлянето,
    //тоест НЯМА да създаде [obj3]
    //
    ^
C obj3; //<-----| //подобно на разгледаното във функцията g()
//след спирането на изпълнението, f() започва да търси
//кой може да обработи грешката, като по пътя вика деструкторите
//на всички обекти, създадени във функцията => се вика деструкторите на [A], [B]

//междувременно не намира начин да обработи грешката и прехвърля
//отговорността на main
}

```

```
113 int main()
114 {
115     f(); //извикваме функцията f()
116
117     //-[ЗАКЛЮЧЕНИЕ] Редът на действие е:
118     // f() -> A() -> B() -> g() -> X() -> throw 37 -> ~X() -> ~B() -> ~A() -> main()
119
120     return 0;
121 }
122 }
```

Тъй като в **main()** изключението не е обработено, сме длъжни да го направим, в противен случай кодът ни ще гръмне.

```
int main()
{
    try
    {
        //за примера ще използваме инстанции на класа [Y], [Z]
        //където класовете [Y], [Z] са идентични на останалите

        Y obj; //Y()

        f(); // --|   вече разглеждахме реда на действия в f()
        //
        //   тъй като f() не обработва изключението, хвърлено от g()
        //   то викането на f() ще хвърли изключение
        //
        //   проблемният код в [try] приключва изпълнението си,
        //   при първото хвърлено изключение, след което търси
        //   съответния [catch], за да го обработи
        //
        //   [!] Обръщаме внимание, че обектът от тип [Y] е създаден
        //   в тялото на [try] => ще се извика деструктора на [Y] след
        //   |-----|   като излезем от тялото на [try] и преди да влезем
        //   |-----|   в съответния [catch], който да обработи грешката
        //
        //предвид казано =>
        //#[obj2] няма да се
        //създаде
        Z obj2;           //|
        //|
    }                   //|
    catch (int x) // <-----|
    {
        std::cout << "error" << x;
    }

    return 0;
}
```



```
int main()
{
try
{
    //за примера ще използваме инстанции на класа [Y], [Z]
    //където класовете [Y], [Z] са идентични на останалите

    Y obj; //Y()

    f(); // --
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    // предвид казаното =>      | Допълнение [2]: ако имаме > 1 [catch], които могат да
    // [obj2] няма да се          | обработят грешката => ще се изпълни първият от тях
    // създаде                  | (този по-нагоре)
    Z obj2;                   ||
                           ||
    catch (int x) // <-----| макар и двата [catch] да хващат [int]
    {                         ||
                           ||
        std::cout << "first error" << x; //| ще се изпълни този, тъй като стои по-нагоре
    }                         ||
                           ||
    catch (...)
    {
        std::cout << "second error"; //| ред на търсене на [catch], който
    }                         // да обработи грешката ни

    return 0;
}
```

Всичко, казано дотук обобщава **Stack unwinding**, който има два случая:

- ако **НЯКОЙ** обработва грешката
- ако **НИКОЙ** не обработва грешката

(`std::terminate ->` незабавно прекратяване на изпълнението на програмата)

### Stack unwinding

- при хвърляне на грешка **изпълнението на функцията се прекратява**
- програмата проверява дали текущата функция или някоя от извикващите функции нагоре по стека може да се справи с изключението (т.е. дали има **try-catch блок**)
- ако бъде намерен съответстващ **блок за обработка** на изключение, изпълнението се прескача от момента, в който е хвърлено изключението, до началото на съответстващия блок за обработка

[ БЕЛЕЖКА]  
(за “прескача”)

```
try
{
    //за примера ще използваме инстанции на класа [Y], [Z]
    //където класовете [Y], [Z] са идентични на останалите

    Y obj; //Y()

    f(); // --
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //предвид казаното =>
    //[obj2] няма да се
    //създаде
    Z obj2;           //|
}                   //|
catch (char ch) // <-----|
{
    std::cout << "error" << ch;
}
```

- това изисква **stack unwinding** (премахване на **текущата функция** от стека на повикванията) толкова пъти, колкото е необходимо, за да може функцията, обработваща изключението, да бъде най-горе в call stack-a

**[БЕЛЕЖКА]:** g() хвърля изключение и ни е **текущата функция**, премахваме я и f() ни става **текущата функция**. Тя не може да обработи изключението, премахваме **текущата функция f()** и отиваме в **main**. **Main** може да обработи изключението.

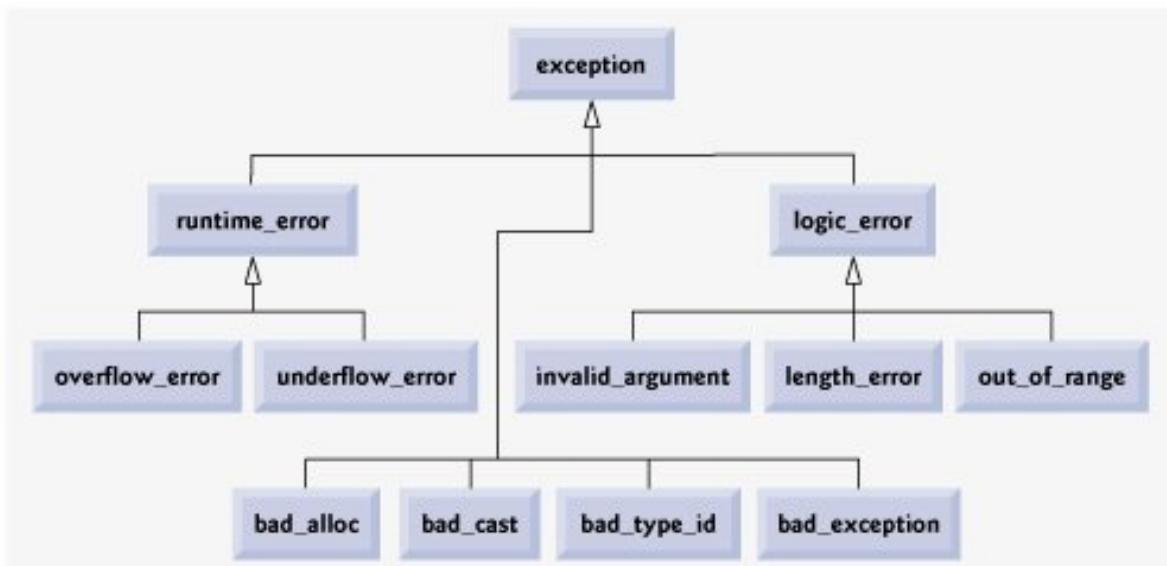
- когато текущата функция се премахне от call stack-a, всички успешно създадени локални променливи се унищожават както обикновено, но не се връща стойност

**[БЕЛЕЖКА]:** Викането на деструкторите от долу нагоре, което разгледахме (всички локални обекти/променливи се троят)

### Exceptions vs. assert

- **assert** - системни/наши грешки
- assert(bool) - ако е **true**, ще спре програмата, ако е **false**, просто ще продължи напред
- **exception** - идеята е да сигнализираме на външния свят, че нещо се е случило некоректно и да обработим грешката (напр.: входни данни)

### Видове грешки



**std::bad\_alloc** - грешка при заделяне на памет (оператора **new**)

**std::bad\_cast** - грешка при кастване (**преобразуване**)

**std::runtime\_error** - грешка по време на изпълнение

**std::logic\_error** - грешка, която наруши инвариантите на класа (условията, които трябва да са изпълнени) и може да бъде предотвратена

(напр.: рационално число число със знаменател 0)

**std::out\_of\_range** - грешка, наследник на **std::logic\_error** (когато например излезем от масива)

Когато подреждаме catch блоковете, класовете по-нагоре в юерархията трябва да са по-надолу, защото са по-общи и хващат повече видове грешки. Тоест:

```
int main()
{
    try
    {
        Y obj; //Y()
        f();
        Z obj2;
    } //най-конкретната
    catch (std::bad_alloc) //|
    { //|
        //|
    } //|
    catch (std::out_of_range) //out_of_range //|
    { //|
        //е наследник на //|
        //|
    } //|
        //стои по-нагоре //|
        //|
    catch (std::logic_error) //logic_error //|
    { //|
        //е наследник на //|
        //|
    } //|
        //стои по-нагоре //|
        //|
    //следните два [catch] пишем за всеки случай //|
    catch (std::exception) //|
    { //|
        //|
    } //|
    catch (...) //|
    { //|
        //|
    } //|
        //към най-лошата (най-общата)
        //|
        //помним, че [...] означава ВСИЧКО, т.е.
        //сърържа и std::exception и неговите наследници
    }
}
```

```
int main()
{
    try
    {
        Y obj; //YO
        f();
        Z obj2;
    }                                //най-конкретната
    catch (std::bad_alloc)           //|
    {                                //|
        //                                //|
    }                                //|
    catch (std::out_of_range)        //|
    {                                //|
        //                                //|
        //е наследник на          //|
        //                           //logic_error и      //|
        //стои по-нагоре          //|
    }                                //|
    catch (std::logic_error)         //|
    {                                //|
        //                                //|
        //е наследник на          //|
        //                           //std::exception и //|
        //стои по-нагоре          //|
    }                                //|
    //следните два [catch] пишем за всеки случай //|
    //|
    catch (...)                      //|
    {                                //|
        //                                //|
    }                                //|
    catch (std::exception)           //|
    {                                //|
        //                                //|
        //                                //|
    }                                //|
    //към най-лошата (най-общата)
}

//от предходния извод следва, че [...] не може да стои преди друг [catch],
//защото ще го "замаскира"
```

```
int main()
{
    try
    {
        throw std::exception("ABC");
    }

    catch (const std::exception& e) //разбираме се, да приемаме обекти от НЕпримитивен тип
                                    //като константна референция в [catch], за да не ги копирате
                                    std::cout << e.what() << std::endl; //ABC, чрез e.what() можем да изведем съдържанието на грешката

    catch (...)
    {
        //тук не можем да изведем съдържанието на грешката, тъй като [...]
        //може да приеме всичко и липсва параметър, който да улавя изключението
    }

    return 0;
}
```

## Деструктор/конструктор

### Деструктор

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

    ~A()
    {
        throw 37;
    }
};

void f()
{
    A obj;
    throw false; //хвърляме изключение [false] =>
                  //помним, че започват да се тряят всички създадени преди
                  //хвърлянето на изключението локални променливи =>
                  //ще се изтрие [obj] =>
                  //ще се извика деструктора на [A] в края на scope-a,
                  //който също хвърля изключение [37]
                  //=> имаме 2 хвърлени изключения и нито едното не е обработено
                  //=> програмата се терминира (std::terminate)
}
```

```
int main()
{
    try
    {
        //извикваме функцията f()
        f();
    }

    //не можем да обработим 2 изключения едновременно
    catch (int x)
    {
        //...
    }

    catch (bool is)
    {
        //...
    }

    return 0;
}
```

**Извод** - В **деструкторите НЕ хвърляме изключения, защото ако този деструктор бъде извикан от друго изключение, програмата се терминира.**

## Конструктор

```
#include <iostream>

class A
{
public:
    A()
    {
        throw 37; //хвърляме изключение [37]

        //при хвърляне на изключения в конструктора се
        //извикват деструкторите на всички НАПЪЛНО построени
        //обекти в конструктора

        //![!] не се извиква деструктора на [A]
    }

    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};

int main()
{
    try
    {
        A obj; //A()
    }
    catch (...)
    {
        std::cout << "here" << std::endl;
    }

    return 0;
}
```

```
//в следващите примери, ще използваме следните класове [A], [B], [C], [X]
class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};

class B
{
public:
    B()
    {
        std::cout << "B()" << std::endl;
        throw 37; //хвърляме изключение в конструктора на [B]
    }

    ~B()
    {
        std::cout << "~B()" << std::endl;
    }
};
```

```
32
33     \> class C
34     {
35         public:
36             C()
37             {
38                 std::cout << "C()" << std::endl;
39             }
40
41             ~C()
42             {
43                 std::cout << "~C()" << std::endl;
44             }
45     };
46
47     \> class X
48     {
49         A obj;
50         B obj2;
51         C obj3;
52     public:
53         X()
54         {
55             std::cout << "X()" << std::endl;
56         }
57
58         ~X()
59         {
60             std::cout << "~X()" << std::endl;
61         }
62     };
63
```

```
int main()
{
    X xObj;

    //създаваме обект от тип [X]
    //=> се извикват конструкторите в следния ред: [X] -> [A] -> [B] (както сме свикнали)
    //помним, че макар X() да се отпечатва последно, конструктора на [X] действително се извика първи
    //просто преди да стигнем до отпечатването викаме конструкторите на [A], [B]

    //когато стигнем до конструктора на [B] в него се хвърля изключение,
    //=> както казахме се извикват деструкторите на всички обекти създадени в конструктора
    //=> деструктора на [B] НЯМА да се извика, тъй като обекта НЕ е приключил създаването си успешно,
    //деструктора на [A] ще се извика, тъй като обекта е приключил създаването си успешно
    //деструктора на [X] няма да се извика, тъй като обекта НЕ е приключил създаването си успешно ([B] е хвърлил изключение)

    //=> A() B() ~A()

    return 0;
}
```

**Извод - В конструкторите МОЖЕМ да хвърляме изключения, но не трябва да останат незатворени външни ресурси (динамична памет)**

```
#include <iostream>

class A
{
    char* str;
public:
    A()
    {
        str = new char[10];

        throw 37; //вече разбрахме, че при хвърляне на
                   //изключение в конструктора,
                   //не се вика деструктора на този обект

                   //=> ~A() няма да се извика
                   //=> няма да се изчисти паметта,
                   // заделена за [str]

    }

    ~A()
    {
        delete[] str;
    }
};
```

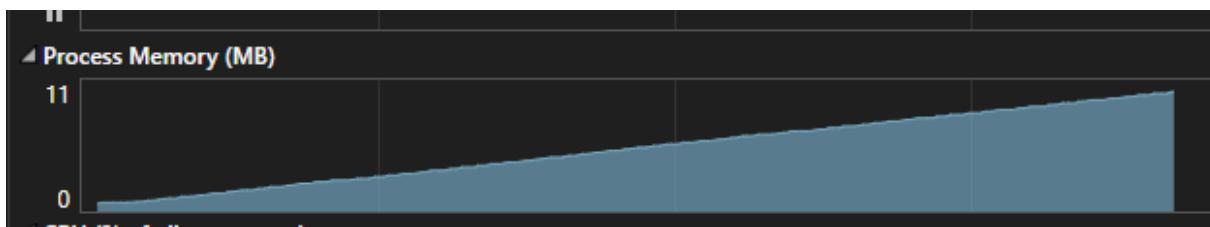
```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като не се вика деструктора на [obj]
        //ще заделяме постоянно нова памет за член-данната [str]
        //без да я освобождаваме => memory leak
        try
        {
            A obj;
        }
        catch (int x) //#[catch] 37
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```

Ако проследим, заделянето на памет, ще видим, че на всяка итерация, паметта необходима на програмата се увеличава постоянно, това означава, че някъде имаме **memory leak** и не сме освободили паметта, заделена за стари данни => графиката на заделената памет потвърждава твърдението, че не се извиква деструктора на **[A]**, откъдето пък следва, че паметта за **[str]** никога не се освобождава



```

#include <iostream>

class A
{
    char* str;

public:
    A()
    {
        str = new char[10];

        //тъй като примерът не е подходящ,
        //за да покажем, че трябва да изчистим паметта
        //сами, ще добавим някакво условие, което ако е изпълнено,
        //значи има проблем

        if (true) //някакво условие, което, ако е изпълнено
            //ще доведе до проблем
        {
            delete[] str;
            throw 37;
        }
    }

    ~A()
    {
        delete[] str;
    }
};

```



```
int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като не се вика деструктора на [obj] ,
        //НО сме изчистили паметта, заделена за външните ресурси,
        //то НЯМАЕ memory leak
        try
        {
            A obj;
        }
        catch (int x) //#[catch] 37
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}
```

Какво прави оператор **new** в следните два сценария:

- **Успешна алокация (успешно създаване на динамичен масив):**

Ако **new** успешно задели памет, вече имаме създаден динамичен масив, готов за използване за нашите цели

- **Неуспешна алокация (неуспешно създаване на динамичен масив):**

Ако **new** не успее да задели памет, то хвърля изключение **std::bad\_alloc**. Важно е да се отбележи, че в този случай **new** се погрижва за изчистването на паметта, която все пак е успяла да се задели.

```
#include <iostream>

class A
{
    char* str;
    char* strTwo;

public:
    A()
    {
        // [new] може да хвърли изключение
        str = new char[10];

        //ако втория масив [strTwo] хвърли изключение,
        //то паметта за него ще се изчисти сама
        //(!НО) няма кой да изчисти [str] => ще получим memory leak
        //ако се хвърли изключение при създаването на [strTwo]

        size_t size = -1; //максимална възможна стойност за size_t,
                           //която ще използваме, за да провокираме bad_alloc
                           //(системата няма достатъчно свободна памет, която да задели)

        strTwo = new char[size];
    }

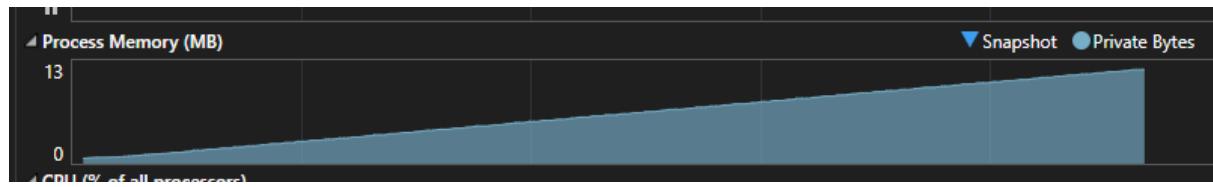
    ~A()
    {
        delete[] str;
        delete[] strTwo;
    }
};
```

```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като [strTwo] хвърля изключение, [new] се грижи за
        //освобождаването на паметта, свързана със [strTwo],
        //но нямаме какво да изчисти [str] => ИМАМЕ memory leak
        try
        {
            A obj;
        }
        catch (...)
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```



```
#include <iostream>

class A
{
    char* str;
    char* strTwo;

public:
    A()
    {
        str = new char[10];

        try
        {
            size_t size = -1;
            strTwo = new char[size];
        }

        catch (const std::bad_alloc& err)
        {
            delete[] str; //освобождаваме паметта,
                           //заделена за [str]

            throw; //хвърляме същата грешка
        }
    }

    ~A()
    {
        delete[] str;
        delete[] strTwo;
    }
};
```

```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като [strTwo] хвърля изключение, [new] се прижи за
        //освобождаването на паметта, свързана със [strTwo],
        //И ВЕЧЕ сме се погрижили за изчистването на [str]
        // => НЯМАМЕ memory leak

        try
        {
            A obj;
        }
        //хващаме хвърленото изключение от A()
        catch (...)
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```



Аналогично, за три масива, ако имаме три масива, заделени динамично, които е възможно да доведат до хвърлянето на изключение

01. Ако **[strOne]** хвърли изключение => **new** ще се погрижи за паметта и не трябва да правим нищо допълнително
02. Ако **[strTwo]** хвърли изключение => **new** ще се погрижи за паметта за **[strTwo]** и ще трябва да изчистим паметта за **[strOne]** сами
03. Ако **[strThree]** хвърли изключение => **new** ще се погрижи за паметта за **[strThree]** и ще трябва да изчистим паметта за **[strOne]** и **[strTwo]** сами

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};
```

```

class X
{
    A* ptr;
    char* arr;
public:
    X()
    {
        //01. заделя паметта за 10 A
        //02. вика 10 x A()
        //03. като някой хвърли изключение, вика деструкторите на миналите
        // Напр.: нека A[4] хвърли изключение
        // т.е. ~A[3], ~A[2], ~A[1], ~A[0]
        // (A[4] не се създава. защото е хвърлил изключение и
        //съответно не се вика деструктора му)
        //04. [new] се грижи в себе си да освободи паметта и осигурява,
        //че се изчиства паметта за незавършените обекти (тоест A[4], A[5] ... A[9])
        ptr = new A[10];

        //съответно деструктора ~X() не се извиква,
        //тъй като грешката е в конструктора X(),
        //чрез std::nothrow забраняваме на [new]
        //да хвърли изключение, ако създаването на [arr]
        //е НЕУСПЕШНО, вместо това, ако е НЕУСПЕШНО,
        // [arr] става [nullptr]
        arr = new (std::nothrow) char[30];
    }

    ~X()
    {
        delete[] ptr;
    }
};

```

```

int main()
{
    //
    //
    //
    //

    return 0;
}

```

# Нива на exception safety (Семинар)

## No throw guarantee (Гаранция за отсъствие на изключения)

Това е най-високото ниво на гаранция, при което се обещава, че дадена операция няма да хвърли изключение **ПРИ НИКАКВИ** обстоятелства. Тази гаранция често се осигурява при функции, които:

- използват само операции, които сами по себе си са no-throw.
- функции, които са деклариирани с `noexcept` (ключова дума, която гарантира, че функцията няма да върне изключение)
- поддържат състоянието непроменено при всякакви условия.

### No throw guarantee

Можем да гарантираме за операциите, че ще успеят дори в ситуация на грешка. Ако грешка се случи тя ще бъде обработена вътрешно и потребителите на кода няма да разберат за нея.

Пример за такива функции обикновено са `move` конструкторите. Функция, която не хвърля грешки, се декларира като `noexcept`.

```
int f() noexcept
{
    return 2 + 3;
}
```

Също функции като `size()`, `empty()`, `capacity()` и тн имат `no throw guarantee`.

Примери за всеки споменат вариант:

```
#include <iostream>

//f() е no-throw функция,
//тъй като всички операции в нея
//няма как да хвърлят грешка
int f()
{
    int x = 3;
    int y = 5;
    int z = x + y;
    int d = z * y;
    //...
    //...

    return 3 + 1;
}
```

```
//g() гарантира, че няма да бъде хвърлена грешка
//чрез ключовата дума noexcept
int g() noexcept
{
    int arr[30] = {};

    //не можем да имаме
    //int* arr = new int[30];
    //тъй като това може да хвърли изключение,
    //което нарушава [noexcept] подсказката
    //(можем но трябва да го направим с [try-catch],
    //за да обработим грешката и да я премахнем)
    //...
    //...

    return 31;
}
```

```
//r() гарантира, че няма да бъде хвърлена грешка
//тъй като в себе си улавя всички възможни такива
int r()
{
    int* arr;
    char* ptr;

    //обработваме всички възможни изключения
    try
    {
        arr = new int[30];
    }
    catch (...)
    {
        arr = nullptr;
    }

    try
    {
        ptr = new char[30];
    }
    catch (...)
    {
        delete[] arr;
    }

    //...
    //...

    return 32;
}
```

```
int main()
{
    f();
    g();
    r();

    return 0;
}
```

## Strong exception safety

### (Силна гаранция за безопасност при изключения)

Силната гаранция за безопасност при изключения означава, че операцията може да хвърли изключение, но ако това стане, състоянието на обекта ще остане непроменено.

```
void f(char*& str)
{
    //ако [new] хвърли изключение,
    //то [str] не се променя и остава
    //#[nullptr]
    try
    {
        size_t size = -1; //тъй като е size_t (unsigned),
                           //превърта от положителната страна

        str = new char[size]; //нямаме достатъчно памет
    }
    catch(...)
    {
        throw; //прехвърляме отговорността изключението
               //да се обработи на main()
    }
}

int main()
{
    char* str = nullptr;

    try
    {
        f(str); //въпреки, че [f] може да върне изключение,
                 //това изключение е обработено на по-високо ниво (в main)
                 //и няма да промени състоянието на програмата,
                 //(няма да се терминира)
                 //както и състоянието на [str] (ще си остане nullptr)
    }
    catch (...)
    {
        std::cout << !str << std::endl; //true (str е nullptr)
    }
}
```

## strong exception safety

Възможно е да се хвърли грешка, но дори да се случи обектът няма да промени състоянието си. Пример за такава функция е функцията `push_back` от `std::vector`. При добавяне на 10 елемента проваленото добавяне на единадесети няма да повлияе на добавените елементи.

## Basic exception guarantee (Основна гаранция за безопасност при изключение)

Ако функцията хвърли изключение програмата е във валидно състояние. Няма отечки на памет и всички инварианти на обектите са изпълнени. За разлика от **Strong exception safety**, тук не е гарантирано, че обекта няма да бъде променен, след хвърлянето на изключение

```
void f(char*& str)
{
    //ако [new] хвърли изключение,
    //то [str] винаги ще се промени
    try
    {
        size_t size = -1; //тъй като е size_t (unsigned),
                           //превърта от положителната страна

        str = new char[size]; //нямаме достатъчно памет
    }
    catch(...)
    {
        str = new char[5] {"asd"};
        throw; //прехвърляме отговорността изключението
               //да се обработи на main()
    }
}

int main()
{
    char* str = nullptr;

    try
    {
        f(str);
    }
    catch (...)
    {
        std::cout << !str << std::endl; //false, [str] е променен от f(),
                                         //тоест при хвърлено изключение,
                                         //функцията НЕ гарантира,
                                         //че [str] няма да бъде променен => basic
    }

    return 0;
}
```

## Basic exception guarantee

Ако функцията хвърли изключение програмата е във валидно състояние. Няма отечки на памет и всички инварианти на обектите са изпълнени.

## No exception guarantee (Няма гаранция за безопасност при изключения)

Не обещаваме нищо - ако програмата хвърли грешка можем да не сме във валидно състояние.

```
#include <iostream>

void f(char*& str)
{
    str = new char[5] {"ads"};

    size_t size = -1; //тъй като е size_t (unsigned),
                      //превърта от положителната страна

    char* strTwo = new char[size]; //нямаме достатъчно памет
}

int main()
{
    char* str = nullptr;
    f(str);

    //ще се хвърли изключение от [strTwo],
    //което не отработваме и програмата ще се терминира

    return 0;
}
```

```
#include <iostream>

void f(char*& str, size_t size)
{
    str = new char[size] {};
}

int main()
{
    //не обещаваме нищо за f()
    //можем да сме във валидно състояние,
    //можем и да не сме
    char* str = nullptr;

    f(str, 312); //валидно състояние
    f(str, -1); //невалидно състояние (хвърля се изключение)

    return 0;
}
```

## No exception guarantee

Не обещаваме нищо - ако програмата хвърли грешка можем да не сме във валидно състояние.