Обектно-ориентирано програмиране
(Информатика, Информационни системи)

2021/ 22 ФМИ, СУ

Вграждане на обекти

Клас може да съдържа като член-данни обекти от други, вече дефинирани, класове. По подразбиране, когато се създава обект от външния клас, се извиква конструктор по подразбиране за вградените обекти.

Нека е дефиниран клас Окръжност, който описва окръжност с даден център и радиус. Центърът на окръжността е обект от клас Точка.

```
class Point
{
	public:
		Point(double = 0, double = 0);
		...
	private:
		double x, y;
};

class Circle
{
	public:
		Circle(): radius(0) {}
		Circle(const Point&, double);
		...
	private:
		Point center;
		double radius;
};

Circle::Circle(const Point& c, double r)
	: radius(r)
{
	center = c;
}

int main()
{
	Circle c;
	Circle c1(p, 1);

	return 0;
}
```

Ред на изпълнение на конструкторите

При създаването на обект от даден клас, преди да се влезе в тялото на конструктора на този клас, се преминава през конструкторите на всички член-данни, които са обекти от други класове. Ако явно не са посочени други конструктори (чрез инициализиращия списък) се използват тези по подразбиране. Спазва се редът, в който член-данните са декларирани.

Ако не съществува конструктор по подразбиране в класовете, от които са член-данните, и няма дефинирани други конструктори, ще бъдат генерирани системни подразбиращи се конструктори. Ако има дефинирани конструктори с параметри (без това да включва параметри със стойности по подразбиране), трябва да бъдат използвани те, като инициализацията се извършва в инициализиращия списък на конструктора на външния клас.

В примера, при създаването на обект c, преди да се влезе в тялото на конструктора на `Circle`, се търси конструктор по подразбиране в класа `Point`, който трябва да инициализира center. В класа

`Point` има конструктор, чиито параметри са с подразбиращи се стойности. Той изпълнява ролята и на подразбиращ се конструктор. Преди да се влезе в тялото на конструктора на класа `Circle` се извиква този конструктор.

Когато се създава обекта `c1`, преди да се влезе в тялото на конструктора с параметри на класа `Circle`, се извиква конструкторът на класа `Point`, който инициализира центъра с подразбиращи се стойности. Търси се именно конструкторът по подразбиране на `Point`, защото липсва инициализация на center в инициализиращия списък на конструктора на класа `Circle`. След това радиусът се инициализира с `r`. В тялото на конструктора на класа `Circle` обектът `center` вече е създаден и се изпълнява операция по присвояване, която променя обекта, като копира подадения аргумент.

Ако в конструктора в класа `Point` нямаше стойности по подразбиране, то задължително в инициализиращия списък на конструктора на `Circle` трябваше да има обръщение към него. *Например:*

```
Circle(): center(0, 0), radius(0) {}
```

За да се спести инициализирането с подразбиращи се стойности, последвано от присвояване, обектът `center` може директно да бъде инициализиран с подадения аргумент, като се използва инициализиращия списък на конструктора.

```
Circle::Circle(const Point& c, double r)
       : center(c), radius(r)
{}
```

Редът, в който се инициализират член-данните, е редът, в който те са декларирани в класа, а не този в инициализиращия списък.

В инициализиращия списък на конструктора на класа `Circle`, центърът на окръжността се инициализира директно с желаната стойност в `c`, като се използва конструкторът за копиране, генериран за класа `Point`.

Инициализиращият списък дава възможност член-данните да бъдат инициализирани директно, вместо първо да получат случайни или подразбиращи се стойности, които да бъдат променени впоследствие. Това е също единственият начин да бъдат инициализирани член-данни като константи и референции, които трябва да получат стойност еднократно при тяхното дефиниране.

За всички членове деструкторите се извикват в обратен ред на реда на конструкторите.

Още за функциите, които се генерират системно

Компилаторът може да генерира служебно редица функции като подразбиращ се конструктор, копиращ конструктор, оператор за присвояване, деструктор.

От друга страна, генерирането на тези функции може да бъде забранено. Как?

Подразбиращ се конструктор

Системен подразбиращ се конструктор се генерира стига в класа да няма дефиниран друг конструктор и когато всички член-данни, обекти от друг клас, могат да се инициализират от конструктор по подразбиране.

Нека е даден класът A, за който не е дефиниран конструктор по подразбиране.

```
class A
{
    int varA;
    public:
        A(int a) { varA = a; }
};

class B
```

```
{
    private:
        A a; // B съдържа a сред член-данни си

    public:
        …
};

int main()
{
    B b;
    return 0;
}
```

За класа A не се генерира автоматично конструктор по подразбиране, защото в класа има дефиниран конструктор с параметри. За класа B не може да бъде генериран конструктор по подразбиране, защото неговите член-данни не могат да бъдат инициализирани с конструктор по подразбиране.

Дори и в класа да има други конструктори, системен подразбиращ се конструктор може да бъде генериран, ако се използва `= default`.

```
A() = default;
```

Дори и за член-данните да не са зададени инициализатори, така генерираният конструктор по подразбиране ще ги нулира. Ако за тях са зададени инициализатори при тяхната декларация, ще бъдат използвани те.

```
class A
{
    int varA{ 3 };
    public:
        A() = default;
        A(int a) { varA = a; }
};
```

Липсата на конструктор по подразбиране води до факта, че не може да бъде дефиниран статичен масив от обекти от съответния клас, освен ако елементите на масива не се инициализират директно с инициализиращ списък:

```
int main()
{
    A array[5]; // изисква конструктор по подразбиране
    A array[5] = { A(1), A(2), 3, 4, A(5) };
    ...
}
```

За `a[2]` неявно се извиква конструкторът с единствен аргумент и `varA` се инициализира с 3.

Всеки конструктор с един аргумент може да се използва за неявно преобразуване, стига да не е дефиниран като `explicit`. При липса на подразбиращ се конструктор за даден клас не може да се създаде динамичен масив от обекти от този клас.

Копиращ конструктор

Конструктор за копиране може да бъде генериран винаги, независимо дали в класа има дефиниран друг конструктор (подразбиращ се или с аргументи).

Системният копиращ конструктор копира едно към едно данните от обекта, подаден като аргумент, използвайки техните копиращи конструктори, ако са обекти или побитово копиране на паметта, ако са скаларни (базови) типове или масиви от такива. Ако трябва да бъде забранен копиращия конструктор (или друга член-функция на класа), то функцията може да се декларира в `private` частта на класа (и да не се дефинира, за да се подсигури, че няма да бъде използвана неволно) или да се използва конструкцията `= delete`.

Нека в класа A копиращият конструктор е забранен. Нека класът B има сред член-данните си обект от класа A. За класа B не може да бъде генериран копиращ конструктор, след като няма как да бъдат копирани всички негови член-данни.

```
class A
{
    …
    public:
        A(const A& other) = delete;
};

class B
{
    private:
        A a; // B съдържа a сред член-данни си

    public:
        …
};

int main()
{
    B b;
    B copy (b); // За B не може да бъде генериран копиращ конструктор
    return 0;
}
```

Ако за класа A е дефиниран копиращ конструктор от вида `A(A&)`, то за B ще бъде генериран копиращ конструктор от вида `B(B&)`.

Един клас може да има повече от един копиращ конструктор. Дори и да има дефиниран копиращ конструктор, може да се предизвика генерирането на системен с конструкцията `= default`, стига да бъде с различен прототип от съществуващия.

Деструктор

Системен деструктор се генерира за всеки клас, за който не е дефиниран изрично. В случая, в който конструкторът не заделя никакви външни ресурси, генерираният системен деструктор е достатъчен.

Деструкторът не трябва да бъде забраняван!

Когато даден обект излезе от област на видимост, той трябва да бъде унищожен, а за целта се извиква деструктор. Може ли да се създаде обект, ако няма как да бъде унищожен? НЕ.

Забраняването на деструктор означава, че обекти от съответния клас не могат да бъдат създавани автоматично. Могат да съществуват обекти в динамичната памет, но забраната на деструктор ще възпрепятства извикването на `delete` и освобождаването на заетата памет.

Ако деструкторът е дефиниран, но в `private` частта, достъп до него имат само други член-функции на класа (или приятели).

Оператор за присвояване

Могат да се генерират различни форми на оператора за присвояване в зависимост от контекста. Копирането с помощта на системния оператор за присвояване е едно към едно.

Има случаи, в които оператор за присвояване не може да бъде генериран. Например, ако в класа има константни член-данни или член-данни, които са референции, т.е. член-данни, за които не може да бъде приложено присвояване.

Ако операторът за присвояване трябва да бъде забранен, може да се декларира в `private` частта на класа или да се използва `= delete`.

Контрол на генерирането на член-функции

В обобщение, генерирането на определена член-функция може да се забрани по два начина:

- функцията може да бъде декларирана, но нейната дефиниция да бъде пропусната;
- с помощта на конструкцията `= delete`.

Каква е разликата?

Когато функцията е забранена с конструкцията `= delete`, всяко нейно извикване ще предизвика грешка по време на компилация.

Когато функцията е декларирана, системна член-функция със същото предназначение не се генерира. Функцията може да бъде извикана, ако е декларирана в `public` частта. Това няма да предизвика грешка по време на компилация. Грешката ще бъде от `linker`-а, който няма да успее да открие дефиницията на функция. Затова в този случай се предпочита декларацията да бъде в `private` частта и да се избегне външен достъп до функция, която няма да бъде дефинирана.

Изключения

В една програма могат да възникнат различни типове грешки:

- синтактични грешки, които се откриват по време на компилация;
- семантични грешки, които се откриват по време на изпълнение на програмата.

Често срещани проблеми, които трябва да могат да бъдат предвидени и може да се реагира на тях, са например:

- параметри с невалидни стойности, подадени на дадена функция;
- възникване на грешка по време на изпълнение на функция, в следствие на която функцията връща определен резултат;
- грешен формат на потребителски вход на дадена програма.

Какво може да се направи?

Ако има ограничение върху стойностите на входните аргументи на дадена функция, в началото на функцията те трябва да се проверят. Например проверка за `nullptr` във функция, която намира дължината на символен низ.

Или функция, която намира индекса на даден елемент в масив, може да върне -1, ако елементът не е част от масива. След приключването на функцията, резултатът трябва да се провери.

```
int findFirstElement(const int* array, size_t size, int elem)
{
    for (std::size_t i = 0; i < size; ++i)
    {
        if (array[i] == elem)
            return i;
    }

    // в случай, че елементът не е открит
    return -1;
}
```

Потребителският вход трябва да се валидира спрямо конкретен формат, който се изисква от програмата. Например, когато от стандартния вход трябва да се прочете цяло число в някакъв интервал, добре е да се провери първо дали изобщо е прочетено число (състоянието на потока), а след това дали числото е в определени граници.

Какво трябва да се направи, ако функцията трябва да върне код за грешка, но и резултат?

```
double divide(int x, int y)
{
    return static_cast<double>(x) / y;
}
```

Какво ще се случи, ако потребителят въведе 0 като стойност за y?

```
// да се провери резултата от функцията, преди да се използва стойността на result
bool divide(int x, int y, double& result)
{
    if (0 == y)
        return false;

    result = static_cast<double>(x) / y;
    return true;
}
```

Този подход, обаче може да бъде неудобен и да направи кодът по-труден за четене, особено когато подобна ситуация би настъпила относително рядко.

Всички тези начини обаче не са подходящи, ако става въпрос за конструктор. Задачата на всеки конструктор е да създаде валиден обект. В допълнение, всяка публична член-функция, която се изпълнява върху обект, предполага начално валидно състояние и трябва да остави обекта отново във валидно състояние.

Конструкторът не връща стойност, той не може да върне код за грешка. Как да се реагира в случай на проблем или на невалидни входни данни? Дори да се добави аргумент, който да указва дали всичко е преминало успешно, в крайна сметка обект ще бъде конструиран, дори той да не е валиден.

До сега в конструкторите на класовете с подобни проблеми се действа по няколко начина:

1) Обектът се маркира като невалиден. Това означава, че при всяка операция (член-функция на класа), която се прилага над обект от този клас, трябва да се проверява дали се работи с валиден обект.

   Например, ако трябва да се конструира рационално число, а за знаменател се подаде 0.

```
Rational::Rational(int numer, int denom)
   : numerator(numer), denominator(denom)
{}

bool Rational::isValid() const
{ return denominator != 0; }
```

   main.cpp

```
Rational rat(1, 0);
if( rat.isValid() ) …
```

2) Обектът се поправя локално, подменя се с валиден.

```
Rational::Rational(int numer, int denom)
   : numerator(numer)
{
   if(0 == denom)
          denom = 1;

   denominator = denom;
}
```

В допълнение, ако в мутаторите се получат невалидни данни, текущият обект може изобщо да не бъде променен.

```
void Rational::setDenominator(int denom)
{
      if(0 == denom) return;
      denominator = denom;
}
```

Дали това е добре? Не. Проблемът се „смита под килима“. Не се сигнализира за него.

Има ли друг механизъм за справяне в такива ситуации? Да.

Изключенията са механизъм за сигнализация за неочаквана ситуация, различна от очаквания, основен ход на събитията. Идеята е да се подаде съобщение за проблем, възникнал по време на изпълнение на програмата и някой, който може да реши проблема, да поеме отговорност да възстанови нормалния ход на нещата. С изключенията в езика са свързани три ключови думи: `throw`, `try` и `catch`.

Хвърляне на изключение

За да бъде сигнализирано за възникването на определена неочаквана ситуация, различна от това, която програмата може да обработи, се използва следната конструкция:

```
throw <израз>;
```

`<израз>` е съобщението към външния свят и може да бъде от произволен тип. Обикновено това е код на грешка (константа от изброен тип), символен низ с описание на грешката, може да бъде дори обект от потребителски дефиниран тип.

Например:

```
throw “Index out of range!”;
throw INVALID_INDEX;
throw InvalidStudentsIDException();
```

Прихващане на изключения

Ключовата дума `try` се използва, за да дефинира блок от операции, който ще бъде наблюдаван за възникване на изключения. Всяка операция в рамките на блока се проверява. Ако възникне изключение, то се прихваща само, ако `try` блокът е последван от `catch` блок с тип, съответстващ на възникналото изключение. Всеки `try` блок трябва да има поне един `catch` блок, а може и да е последвано от няколко такива.

Прости примери за възникване и прихващане на изключения:

```
std::cout << "Point 1: Before the try block..." << std::endl;
try
{
    std::cout << "Point 2: Inside the try block, before the throw..." << std::endl;
    throw "Error"; // изключението е от тип символен низ
    std::cout << "Point 3: Inside the try block, after the throw..." << std::endl;
}
catch (const char* ex) // това, което е обявено като тип в catch блока трябва да съвпада по тип
{
    std::cerr << "Exception caught: " << ex << std::endl;
}
std::cout << "Point 4: After the try block, after the throw..." << std::endl;
```

При възникване на изключение, изпълнението на програмата се прехвърля към най-близкия `try` блок, който го обхваща. Изключението се прихваща от съответния `catch` блок, който е от същия тип, като типа на изключението (или от неговия базов тип). След обработване на изключението в `catch` блока, изпълнението се възстановява от позицията след `catch` блока. На екрана се извеждат `Point 1, 2 и 4`.

Какво може да се случи в един `catch` блок?
Най-често се съобщава на потребителя за възникналата грешка, като грешката се записва във файл, възможно е да се освобождават някакви заети ресурси. В `catch` блока може да се хвърли друг вид изключение, да се промени съобщението към външния свят.

Логическа грешка е един и същ тип изключение да се обработва от два различни `catch` блока.

```
void throwException()
{
    int choice;
    std::cout
        << "What do you want to throw?\n"
```

```
        << "  [1] char\n"
        << "  [2] int\n"
        << "  [3] double\n"
        << "  [4] std::exception object\n"
        << "  [5] std::out_of_range object\n"
        << "Your choice: ";

    std::cin >> choice;

    switch (choice)
    {
        case 1: throw '!';
        case 2: throw 5;
        case 3: throw (double)5.5;
        case 4: throw std::exception("Something happened!");
        case 5: throw std::out_of_range("Something is out of range!");

        default:
            throw std::exception("A number between 1 and 5 was expected!");
    }

    std::cout << "Will this line ever be executed? Never!\n";
}

// външна функция, която обработва само два вида изключения
void caller()
{
    try
    {
        // дори и функцията да хвърли изключение от тип char,
        // няма да бъде направено преобразуване до int

        // не се осъществява неявно преобразуване за примитивните типове данни
        // единствено, ако имаме йерархия от изключения,
        // изключение от производен тип може да бъде прихванато от catch блок,
        // в който е указан базовият тип
        // ако в catch блока трябва да бъдат изброени и производните типове и базовия тип,
        // базовият тип трябва да бъде накрая
        throwException();
    }
    catch (int ex)
    {
        std::cout << "Caught an int: " << ex << std::endl;
    }
    catch (double ex)
    {
        std::cout << "Caught a double: " << ex << std::endl;
    }
}

int main()
{
    try
    {
        // кои от изключенията са обработени в caller?
        caller();
    }
    // ако изключението е обект, то трябва да бъде прихванато по псевдоним,
    // за да се избегне излишното му копиране
    catch (const std::out_of_range& ex)
    {
        std::cout << "Caught a std::out_of_range: " << ex.what() << std::endl;
    }
    // std::out_of_range преди std::exception, защото е производен клас
```

```
    // в противен случай изключение от тип std::out_of_range ще бъде прихващането
    // в catch блока на std::exception
    catch (const std::exception& ex)
    {
        std::cout << "Caught a std::exception: " << ex.what() << std::endl;
    }
    // специален тип catch блок, който се използва за прихващане на всякакъв вид изключения,
    // ако се използва трябва да стои най-отдолу, като последен блок
    catch (...)
    {
        std::cout << "Caught something the rest have failed to catch... but what is it?\n";
    }
}
```

След затварящата скоба на `catch` блока, всички локални променливи излизат от област на видимост, включително и параметъра на блока. След обработване на изключението, изпълнението продължава от първата операция след `try` блока, освен ако в catch блока не бъде хвърлено друго изключение или да се хвърли наново същото.

Ако в тялото на `try` блока не възникне изключение, всички `catch` блокове се игнорират и изпълнението продължава от първата операция след `try` блока.

Ако в тялото на `try` блока възникне изключение, което няма съответстващ `catch` блок или възникне изключение извън `try` блок, съдържащата го функция (т.е. функцията, в която е възникнало изключението) прекъсва, при което всички локални променливи се освобождават и програмата се опитва да намери `try` блок в извикващата функция. Извършва се изкачване по стека (`stack unwinding`)...

Всички ресурси, които се използват трябва да бъдат ангажирани в конструктори на обекти, за да могат да бъдат освободени при извикването на съответните деструктори.

Кои от изключенията се обработват във функцията `caller` и кои ще бъдат прехвърлени към `main`?

Ако изключението е от тип `int` или `double`, то ще бъде обработено във функцията `caller`. Ако обаче изключението е от друг тип, то ще бъде прехвърлено към функцията `main`.

Изключения в член-функции на класове и при предефиниране на операции

Изключения в конструкторите

Ако конструктор не успее по някаква причина да създаде валиден обект, например при подадени некоректни стойности за член-данните, може да се хвърли изключение, с което да се сигнализира, че обектът не е създаден успешно.

```
Rational::Rational(int numer, int denom)
	: numerator(numer)
{
	if(0 == denom)
		throw “Invalid denominator!”;

	denominator = denom;
}
```

В този случай, всички член-данни, които вече са били създадени и инициализирани, преди възникването на грешката, се освобождават. Деструкторът на класа обаче не се извиква.

И понеже деструктор не се извиква, не може да се разчита на него да освободи външни ресурси, ангажирани преди възникването на грешката. Освобождаването на тези ресурси трябва да се осъществи преди хвърлянето на изключението. Ако в класа има вградени обекти от други класове, за тях ще бъдат извикани деструктори.

```
class Item
{
	public:
```

```
            Item(const char* id, const char* name, double price);
            Item(const Item& other);
            Item& operator=(const Item& other);
            ~Item();
            ...
      private:
            char id[6];
            char* name;
            double price;
};

Item::Item(const char* newCode, const char* newName, double newPrice)
{
      if (newName == nullptr) throw "Invalid name!";

      // ако паметта за името е заделена първо
      this->name = new char[strlen(newName) + 1];
      strcpy(this->name, newName);

      if (!isValid(newCode))
      {
            // трябва да се освободи преди да се хвърли изключение
            delete [] this->name;
            throw "Invalid code!";
      }

      strcpy(this->id, newCode);

      if (newPrice < 0)
      {
            delete [] this->name;
            throw "Invalid price!";
      }

      this->price = newPrice;
}

// може да се реализира така
Item::Item(const char* newCode, const char* newName, double newPrice)
{
      if (!isValid(newCode))
            throw "Invalid code!";

      strcpy(this->id, newCode);

      if (newPrice < 0)
            throw "Invalid price!";

      this->price = newPrice;

      // може името да се обработи последно
      if (newName == nullptr) throw "Invalid name!";

      this->name = new char[strlen(newName) + 1];
      strcpy(this->name, newName);

}

// ако се използват мутатори, които хвърлят изключения
Item::Item(const char* newCode, const char* newName, double newPrice)
    : name(nullptr)
// трябва да се зададе стойност, в случай че възникне изключение преди да е заделена паметта
{
      try
```

```
	{
		this->setID(newCode);
		this->setName(newName);
		this->setPrice(newPrice);
	}
	// трябва да е ясно какви изключения могат да възникнат и да бъдат прихванати
	catch (const char* ex)
	{
		delete[] this->name;
		// хвърля се същото изключение, което е прихванато
		throw;
	}
}

void Item::setID(const char* newCode)
{
	if (!isValid(newCode))
		throw "Invalid code!";

	strcpy(this->id, newCode);
}

void Item::setName(const char* newName)
{
	if (newName == nullptr)
		throw "Invalid name!";

	size_t nameLen = strlen(newName);
	if (nameLen == 0)
		throw "Empty name!";

	char* temp = new char[nameLen + 1];
	// ако има проблем при заделянето на памет, текущият обект няма да бъде променен
	strcpy(temp, newName);

	delete[] this->name;
	this->name = temp;
}

void Item::setPrice(double newPrice)
{
	if (newPrice < 0)
		throw "Invalid price!";

	this->price = newPrice;
}
```

Нека е даден следния пример:

```
Item* items = new Item [5];
```

Какво ще се случи, ако възникне проблем при създаването на 3-тия обект?

Паметта за масива е заделена. Създават се успешно обект [0] и [1]. При създаването на обект [2] възниква изключение. Обекти [3] и [4] не се създават изобщо.

След възникването на изключението, обекти [0] и [1] се освобождават, за тях се извикват деструктори, след което се освобождава заделената за масива памет.

Изключения в деструкторите

Изключенията не трябва да напускат деструкторите. Ако възникне изключение в деструктора, то трябва да се прихване и да се обработи, ако е възможно да бъдат освободени всички заделени външни ресурси. Проблемът при възникване на изключение в деструктор е, че при превъртането на стека при обработка на изключение се извикват деструкторите на всички локални обекти. Ако от такъв деструктор възникне изключение, то компилаторът не е в състояние да обработи новото

изключение успоредно с текущото и възниква конфликт. В крайна сметка, програмата ще бъде прекъсната веднага без възможност за обработка.

Изключения в операторни функции

Например `operator[]` в клас IntArray, представящ динамичен масив от цели числа.

```
int& IntArray::operator[](const int index)
{
    return m_data[index];
}

int& IntArray::operator[](const int index)
{
    assert(index >= 0 && index < getLength());
    return m_data[index];
}
```

Тъй като операторите имат точно определен брой и тип на аргументите си и на резултата, който връщат, няма голяма свобода при евентуално възникване на грешка. Не може да се върне код за грешка или пък булева стойност, която да подскаже дали функцията е завършила успешно.

`Assert` може да се използва само в debug режим. Ако условието е истина, изпълнението на програмата ще продължи. В противен случай, програмата ще бъде прекъсната.

```
int& IntArray::operator[](const int index)
{
    if (index < 0 || index >= getLength())
        throw “Invalid index”;

    return m_data[index];
}
```

Нива на сигурност при изключенията

Съществуват няколко различни нива на сигурност при възникване на изключение.

No-throw guarantee

Операциите винаги са успешни. Дори и да възникне някакво изключение, то се обработва вътрешно и външният свят не разбира за него. Пример за такава функция е преместващият конструктор или деструктора.

Силна сигурност (strong exception safety guarantee)

Операциите могат да пропаднат, но ако дадена операция пропадне, то няма да има странични ефекти от нея и обектът остава в оригиналното си състояние. Понякога се постига по-трудно.

Слаба сигурност (weak exception safety guarantee)

Частично изпълнение на операциите, което може да доведе до странични ефекти. Обектът е във валидно състояние, което може да е различно от първоначалното. Няма изтичане на ресурси.

В операцията за присвояване редът, в който се извършват операциите, има значение. Копирането се извършва от вече създаден валиден обект. Това означава, че не могат да възникнат изключения във функциите `setID` и `setPrice`.

Примери:

Ако възникне изключение при заделянето на памет за `name`, останалите член-данни на обекта вече ще бъдат променени. Каква е гаранцията?

```
Item& Item::operator=(const Item& other)
{
	if (this != &other)
	{
		this->setID(other.id);
```

```
            this->setPrice(other.price);
            this->setName(other.name);
      }

      return *this;
}
```

Ако възникне изключение при заделянето на памет за name тук, член-данните на обекта няма да бъдат променени. Каква е гаранцията?

```
Item& Item::operator=(const Item& other)
{
      if (this != &other)
      {
            this->setName(other.name);
            this->setID(other.id);
            this->setPrice(other.price);
      }

      return *this;
}
```

No exception safety

Няма никаква гаранция за състоянието на обектите и операциите.

Други проблеми, които могат да възникнат при обработване на изключения:

```
try
{
    openFile(filename);
    writeFile(filename, data); // ако възникне изключение тук, файлът няма да се затвори
    closeFile(filename);
}
catch (FileException& exception)
{
    cerr << "Failed to write to file: " << exception.what() << '\n';
}
```

Файлът е отворен, но при неговото обработване възниква някаква грешка. Процесът продължава с `catch` блока, за да обработи изключението и файлът остава незатворен.

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException& exception)
{
    // първо да се затвори файла
    closeFile(filename);
    // после да се съобщи за грешката
    cerr << "Failed to write to file: " << exception.what() << '\n';
}
```

Пример със заделяне на динамична памет:

```
try
{
    // me е дефинирано локално за try блока,
    // няма да бъде достъпно в catch блока
    Student* me = new Student("Me", 18, 42901);
    processStudent(me);
    delete me;
}
catch (StudentException& exception)
```

```
{
    cerr << "Failed to process student: " << exception.what() << '\n';
}
```

Какво може да се направи?

```
Student* me = nullptr;
try
{
    me = new Student("Me", 18, 42901);
    processStudent(me);
    delete me;
}
catch (StudentException& exception)
{
    delete me;
    cerr << "Failed to process student: " << exception.what() << '\n';
}
```

Изключенията увеличават размера на изпълнимите файлове, може да има и малко забавяне при изпълнението на програмата, поради допълнителните проверки, които трябва да бъдат направени. Всъщност най-голямата цена, която се плаща е превъртането на стека при възникване на изключение.

Статични член-данни и член-функции

Когато се създава обект от даден клас, този обект разполага със свое собствено копие на член-данните. Когато обаче член-данните са декларирани като статични (с ключовата дума `static`), те се поделят от всички обекти на класа.

Особености на статичните член-данни:

1. Статичните член-данни се декларират в класа. Преди да бъдат използвани, те трябва да бъдат дефинирани и инициализирани. Освен в някои случаи, това се случва извън декларацията на класа.
2. Памет за статичните член-данни се заделя не върху стека, а в областта за статични данни.
3. Памет за статичните член-данни се заделя еднократно. Всички обекти от класа имат достъп до нея.
4. Статичните член-данни се създават, когато се стартира програмата и се унищожават, когато програмата приключи. Тъй като те съществуват преди изобщо да има създадени обекти от класа, те се свързват със самия клас, а не с някой конкретен обект. Най-добре е достъпът до тях да се осъществява през класа (с оператор за принадлежност ::), а не през обект.

Нека е даден клас, който следи броя на живите си обекти. Всеки обект има свой уникален идентификатор. Броят на живите обекти не може да бъде деклариран като обикновени член-данни. Той трябва да се променя (достъпва) при създаването на обект от съответния конструктор, както и при унищожаването на обект от деструктора на класа. Всеки обект може да променя стойността му и тази промяна да е видима за останалите обекти.

```
class ObjectCounter
{
public:
    // Брой обекти, живи в момента.
    static int getCurrentCount();

public:
    // Подразбиращ се и копиращ конструктор. Променят броя живи обекти.
    // Задават уникален идентификатор
    ObjectCounter();
    ObjectCounter(const ObjectCounter&);

    // Деструктор - намалява броя живи обекти
    ~ObjectCounter();
```

```
    // При присвояване трябва да запазваме уникалното ID
    ObjectCounter& operator=(const ObjectCounter& rhs);

    int getObjID() const;

private:
    static int count;
    static int nextID;
    static const int idStep = 1;

private:
    int id;

private:
    // Помощна функция за пресмятане на следващо ID на обект
    static int getNextID();
};
```

В `private` частта на класа са декларирани две статични член-данни `count` и `nextID`, които описват съответно броя на живите обекти и поредния идентификатор на обект.

Статичните член-данни се декларират в класа. Независимо в коя част на класа са декларирани (`public, private, protected`), те се дефинират явно извън класа (подобно на глобални променливи), като дефиницията може да е съпътствана с инициализация. Ако не е посочена стойност, те се инициализират с 0.

Ако класът е дефиниран в `header` (`.h/.hpp`) файл, то статичните член-данни се дефинират в съответния `.cpp` файл, заедно с дефинициите на член-функциите на класа. Ако статичните член-данни се дефинират в `header` файл, включването на този header файл в няколко source файла ще доведе до многократна им дефиниция и съответна грешка при свързване на програмата (linker error).

Ако дефинициите на класа и неговите член-функции не са в отделни файлове, то статичните член-данни се дефинират непосредствено след дефиницията на класа.

Ключовата дума `static` се използва само при декларирането на член-данните.

```
int ObjectCounter::countObjects = 0;

int ObjectCounter::nextID = 0;
```

Ако статичните член-данни са константи от интегрален тип (целочислен тип, `char`, `bool`), те могат да бъдат инициализирани директно с тяхната декларация. В този случай дефиниция извън класа не е необходима. Такъв пример е `idStep` от класа `ObjectCounter`, описващ стъпката при генерирането на `ID`-та за обектите.

Статичните член-данни на класа са достъпни във всички член-функции на класа.

```
ObjectCounter::ObjectCounter()
{
	++countObjects;
	this->id = ObjectCounter::generateID();
}
```

Ако статични член-данни са декларирани в `private` частта на класа, до тях няма външен достъп. За да бъде осигурен този достъп не е необходимо да се използва обект на класа, те могат да съществуват и без да е създаден такъв. Достъпът може да се осъществи през самия клас и дефинирани в него *статични член-функции*.

В класовете могат да бъдат дефинирани и статични член-функции. Ключовата дума `static` отново се използва само при декларацията на член-функцията.

Особености на статичните член-функции

1. Тъй като статичните член-функции могат да бъдат извикани през самия клас, дори без да съществуват обекти от този клас, то в тях няма дефиниран указател `this`.
2. След като не се изпълняват върху конкретен текущ обект, то в тях не могат да се използват други член-данни на класа, които не са статични. В статичните член-функции могат да се използват само статични член-данни.
3. Статичните член-функции не могат да бъдат константни. Причината отново е, че не се изпълняват върху конкретен текущ обект.
4. Препоръчително е статичните член-функции (както и статичните член-данни) да бъдат използвани само с пълното име на класа, а не през обект.

В класа `ObjectCounter` статичните член-функции са две, дефинирани в различни секции на класа:

```
static int getCurrentCount();
static int getNextID();
```

При дефинирането на статичната член-функция ключовата дума `static` се пропуска:

```
int ObjectCounter:: getNextID()
{
    nextID += idStep;
    return nextID;
}
```

Допълнителни материали:

- https://en.wikipedia.org/wiki/Constructor_%28object-oriented_programming%29
- https://en.wikipedia.org/wiki/Destructor_%28computer_programming%29
- http://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm
- https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html
- http://www.learncpp.com/cpp-tutorial/8-7-destructors/
- https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Resource_Acquisition_Is_Initialization
- http://en.cppreference.com/w/cpp/language/member_functions#Special_member_functions
- http://www.cplusplus.com/doc/tutorial/exceptions/
- https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm
- http://www.acodersjourney.com/2016/08/top-15-c-exception-handling-mistakes-avoid/