Обектно-ориентирано програмиране
(Информатика, Информационни системи)

2021/ 22 ФМИ, СУ

До тук за класове…

Всеки нов клас, който се дефинира е нов тип в езика. Както за всеки друг тип, трябва да се въведат правила, по които да се работи с него. Една част от характеристиките на класа (и поведението) трябва да останат скрити за външния свят, като бъдат маркирани като `private`. Други, обратно, да бъдат достъпни за него (`public`). Какви са основните предимства на капсулирането?

Конструктори

Нека да е дефинирана структура, представяща рационално число. Нека рационалното число се характеризира с числител и знаменател – цели числа.

Когато всички член-данни на даден клас или структура са публични, обект от класа или променлива от типа на структурата могат да се създадат и инициализират с помощта на инициализатор (`list-initialization`).

```
struct Rational
{
	int numerator;
	int denominator;
};

int main()
{
	Rational r {3, 5};

	return 0;
}
```

Когато член-данните обаче са скрити (`private`) и не могат да бъдат достъпени пряко извън класа, инициализацията не може да се случи по същия начин.

Нека вместо структура да е дефиниран клас `Rational`.

`Rational.h`

```
#ifndef RATIONAL_H_INCLUDED

class Rational
{
	private:
		int numerator;
		int denominator;

	public:
		int getNumerator();
		int getDenominator();

		void setNumerator(int newNominator);
		void setDenominator(int newDenominator);

		void print();
};

#endif // !RATIONAL_H_INCLUDED
```

`main.cpp`

```
#include "Rational.h"
```

```
int main()
{
	Rational r;
	r.setDenominator(3);

	return 0;
}
```

В примера, памет за обекта `r` се заделя автоматично, когато се влезе в блока, в който обектът е дефиниран (в случая това е функцията `main()`). Тази памет е достатъчно, за да се съхранят числителят и знаменателят на рационалното число.

Какви обаче са стойностите на характеристиките на създадения обект? В общия случай – случайни. Стойностите на член-данните могат да бъдат променени с помощта на съответните мутатори (`set`-функции), ако в класа има дефинирани такива.

Какъв обаче е механизмът вече заделената памет за обекта `r` автоматично да се инициализира с някакви конкретни стойности?

Идея

Конструкторът е специална член-функция на класа, която се извиква автоматично при създаването на обект от този клас. Конструкторът се извиква *един-единствен път* при създаването на обекта.

Конструкторите се използват, за да инициализират всички член-данни на обекта с подходящи, валидни стойности, подадени от потребителя. Общо казано, конструкторите подготвят обекта за работа. Това може да означава и да се задели динамична памет, да се отвори файл, да се направи връзка с база от данни и др.

След приключването на конструктора, обектът трябва да бъде във *валидно* състояние.

Какво означава валидно състояние в случая с рационално число? Знаменателят трябва да бъде различен от 0.

Синтаксис

Две важни правила при дефинирането на конструктор:

1. Конструкторите носят същото име като класа.
2. Конструкторите не връщат стойност.

Пример:

```
public:
    Rational();
    Rational(int numerator, int denominator);
```

Възниква въпросът, ако не се връща стойност, как може да се съобщи за възникване на грешка по време на работата на конструктора? (по-късно)

Видове

Езикът позволява съществуването на функции с еднакви имена, стига броят и/ или типът на техните аргументи да се различават. Това важи и за конструкторите.

Всеки клас може да има повече от един конструктор. Конструкторите трябва да инициализират *всички* член-данни на обекта с *валидни* стойности. За да могат да се използват във външни функции, конструкторите трябва да бъдат публични.

Конструктор по подразбиране

Ако в класа не е дефиниран *никакъв* конструктор, то компилаторът генерира т. нар. *конструктор по подразбиране*.

В примера по-горе, в класа няма дефиниран конструктор. Въпреки това, може да се създаде обект, чиито характеристики ще имат случайни стойности.

Системният конструктор по подразбиране може да бъде предефиниран, ако в класа се дефинира конструктор *без параметри*.

```
public:
    Rational();

Rational::Rational()
{
    this->numerator = 0;
    this->denominator = 1;
}
```

Как се използва конструкторът по подразбиране?

НЕ използвайте:

```
Rational r();
```

Какво представлява тази конструкция? Декларация на функция с име `r`, която не получава никакви аргументи и връща обект от клас `Rational`.

Правилният начин е:

```
Rational r;
Rational r {};
```

И в двата случая се използва дефинираният конструктор по подразбиране.

Резултатът е различен обаче в случая, когато в класа няма дефинирани конструктори. Тогава `Rational r;` ще инициализира член-данните със случайни стойности, докато `Rational r {};` ще ги нулира.

Конструктор с параметри

В класа могат да бъдат дефинирани и други конструктори, които получават като аргументи конкретни стойности, с които да бъдат инициализирани член-данните на създавания обект.

В примера за рационално число може да се дефинира конструктор, който получава два параметъра, с които да бъдат инициализирани съответно и числителят, и знаменателят.

```
Rational::Rational(int numerator, int denominator)
{
    this->numerator = numerator;
    // без проверка за стойността, с която се инициализира знаменателят
    this->denominator = denominator;
}
```

Как се извиква?

```
Rational r(2, 3); или Rational r {2, 3};
```

Ако има дефиниран поне един конструктор, *не се генерира* системен конструктор по подразбиране.

Ако не съществува конструктор по подразбиране, опитът да се създаде обект с него ще доведе до грешка. В този случай, обект от класа може да се създаде само с помощта на дефинирания/ дефинираните конструктори.

Как цяло число да се представи като рационално? Може да се дефинира конструктор, който получава като аргумент едно цяло число, което служи за инициализиране на числителя, а знаменателят се инициализира с 1.

```
public:
    Rational(int numerator);

Rational::Rational(int numerator)
{
    this->numerator = numerator;
    this->denominator = 1;
}
```

Могат ли тези три конструктора да бъдат обединени по някакъв начин?

Функциите в езика C++ могат да имат параметри с подразбиращи се стойности, които да се използват, ако при извикването на функцията не бъде подадена стойност за съответния параметър.

Стойностите по подразбиране се задават в прототипа на функцията. Ако става въпрос за член-функция на даден клас и дефиницията на класа е разделена от дефиницията на член-функциите му, то мястото на подразбиращите се параметри е само в header файла с декларацията на функцията.

Правила при използването на параметри с подразбиращи се стойности във функции:

1. Ако се зададе стойност по подразбиране на един параметър на функция, всички параметри след него също трябва да имат стойност по подразбиране.
2. Ако при обръщение към функцията някой от параметрите с подразбиращи се стойности бъде пропуснат, то всички след него също трябва да бъдат пропуснати.

Конструкторите също могат да имат параметри със стойности по подразбиране.

```
Rational(int numerator = 0, int denominator = 1);
```

Какво дава тази дефиниция?

Така дефинираният конструктор с параметри с подразбиращи се стойности може да се използва като конструктор с два параметъра, конструктор с един параметър и дори като конструктор по подразбиране.

```
Rational r;         // създава се обекта (0, 1)
Rational r(2);      // създава се обекта (2, 1)
Rational r(2, 3);   // създава се обекта (2, 3)
```

Конструкторите не могат да бъдат използвани за повторна инициализация на вече съществуващ обект. Те се използват еднократно за инициализация на обекта при неговото създаване.

Копиращ конструктор

Инициализацията на нов обект от даден клас може да зависи от вече създаден обект от същия клас, като се използват символът за присвояване или ().

```
Rational r(2, 3);
Rational copy = r;        или с директна инициализация          Rational copy (r);
```

Обектът `copy` се инициализира от специален конструктор, който се нарича *конструктор за копиране*. Ако в даден клас няма явно дефиниран такъв, той се генерира системно, дори и в класа да има дефиниран друг конструктор с аргументи. Конструкторът копира едно към едно съдържанието на аргумента в обекта, който се създава.

В примерния клас `Rational` няма нужда изрично да се дефинира копиращ конструктор. Член-данните могат да бъдат копирани коректно от системния копиращ конструктор, който ще се генерира за класа.

Кога се извиква копиращ конструктор?

Ако формалният параметър на една функция е предаден по стойност, фактическият параметър се копира с помощта на конструктора за копиране.

```
void function(T param)
{ ... }
```

Ако резултатът от изпълнението на дадена функция е обект, който се връща по стойност (стига в класа да няма дефиниран move конструктор).

```
T function()
{
    ...
    return res;
}
```

Какъв е прототипът на копиращия конструктор?

Копиращият конструктор, който се генерира за класа `Rational` е публичен и изглежда така:

```
public:
    Rational(const Rational&);

Rational::Rational(const Rational& other)
{
    this->numerator = other.numerator;
    this->denominator = other.denominator;
}
```

Аргументът на копиращия конструктор не може да бъде предаден по стойност. Защо?

Когато предаването е по стойност, се извършва копиране на фактическия параметър. За да се осъществи това копиране е необходим същият копиращ конструктор, който се дефинира. Получава се една бездънна рекурсия… По тази причина аргументът задължително се предава по референция.

Най-често версията на копиращия конструктор, който се генерира или който се дефинира изрично, е с аргумент от тип `const <име_на_класа>&`.

Има ли случаи, в които няма да бъде генериран копиращ конструктор?

Системен копиращ конструктор няма да бъде генериран, ако някои от член-данните не могат да бъдат копирани (копирането е забранено, копиращите конструктори са недостъпни или има повече от един копиращ конструктор, но не може да се избере кой).

Още един пример, в който генерираният копиращ конструктор е достатъчен.

Нека е дефиниран клас Студент, както следва:

```
class Student
{
    ...
    private:
        char name[24];
        int fn;
        double avgGrades;
}
```

Паметта, която се заделя за един обект от класа `Student`, е достатъчна, за да се съхранят символен низ с най-много 23 символа, едно цяло число и едно реално число. Данните се съхраняват в самия обект. Системният копиращ конструктор ще копира информацията едно към едно от вече създаден обект – в новия, който се създава.

Кога изрично трябва да бъде дефиниран копиращ конструктор?

Копирането на член-данните на обект от даден клас не е тривиално, например ако обектът има член-данни, разположени в динамичната памет.

Ако дефиницията на класа `Student` е променена по следния начин:

```
class Student
{
    ...
    private:
        char* name;
        int fn;
        double avgGrades;
}
```

Нека името се съхранява в динамичната памет. Тогава в обекта се съхранява адресът на тази динамична памет.

Паметта, която се заделя за един обект от класа `Student`, е достатъчна, за да се съхранят един адрес, едно цяло и едно реално число. Системният копиращ конструктор ще копира информацията едно към едно от вече създаден обект – в новия, който се създава. Какво означава това?

В новия обект и обекта, от който се копират данните, ще бъде записан един и същи адрес. Двата обекта не са независими, сочат една и съща памет. Промяна в единия, означа промяна в другия. А ако единият излезе от област на видимост?

За да се осъществи правилно копирането, в класа трябва да се дефинира копиращ конструктор, който да се погрижи да копира всички член-данни от вече създадения обект. Дори и по принцип да се налага валидация, заради ограничения в допустимите стойности на член-данните, в копиращия конструктор проверките могат да се пропусната, защото се копира от обект, който е бил валидиран при неговото създаване.

```
Student::Student(const Student& other)
	: fn(other.fn), grade(other.grade)
{
	this->name = new char[strlen(other.name) + 1];
	strcpy(this->name, other.name);
}
```

Константни член-функции и член-данни. Инициализатори.

Константни член-функции

Както за всеки друг тип, така и при потребителските дефинирани типове може да бъде създаден константен обект. Кои от дефинираните в класа операции са позволени за един константен обект? Това са функции, които не променят състоянието на обекта? Как да бъдат разграничени тези функции? Функциите, които не променят състоянието на обекта трябва да бъдат маркирани като константни.

```
void print() const;
```

Ключовата дума `const` се добавя след затварящите скоби за аргументите на функцията. Така компилаторът може да проследи дали в рамките на функцията не се прави опит за промяна на обекта, върху който е приложена. Ако в константна функция се използват други член-функции, то те също трябва да бъдат константни.

Константни са обикновено селекторите в даден клас. Константна член-функция не може да върне неконстантна референция към член-данни на класа, поради това често се дефинират константна и неконстантна версия на една и съща член-функция.

Например в клас, описващ динамичен масив, може да съществуват две функции за достъп до конкретен елемент на масива по индекс.

```
int& Array::at(std::size_t index);

const int& Array::at(std::size_t index) const;
```

Много по-често от същински константни обекти се работи с референции към константни обекти. Ако е дефинирана функция, чиито аргумент е от вида `const <име_на_клас>&`, то върху този аргумент могат да бъдат приложени само константни функции.

Какъв е типът на указателя this в константна функция?

`const <име_на_клас> * const this;`

Константни член-данни

Какво се случва, ако в класа има член-данни, които са константни?

Константите могат да бъдат инициализирани един-единствен път, когато се декларират. Когато конструкторът започне работа, за член-данните на обекта вече е заделена памет и са инициализирани или със случайни стойности, или със стойности, подадени при тяхното деклариране. В тялото на конструктора се извършва промяна на стойностите им (чрез присвояване). За член-данните, които са константи това е проблем.

```
class Sample
```

```
{
    private:
        const int c;
        int var;
    public:
        ...
};

Sample::Sample(int c, int var)
{
    // преди да се влезе в тялото на конструктора,
    // константата с вече е инициализирана със случайна стойност
    // опитът да бъде променена тази стойност води до грешка
    this->c = c;
    this->var = var;
}
```

За да бъдат инициализирани константни член-данни, както и за референции, се използва т. нар. инициализиращ списък на конструктора. Конструкторите са единствените функции, които имат инициализиращ списък.

```
Sample::Sample(int c, int var)
             : c(c), var(var)
{ ... }
```

Инициализиращият списък се описва в дефиницията на конструктора, като след затварящите скоби на списъка с аргументи се поставя : и се изброяват член-данните, които трябва да се инициализират, като след идентификаторите на член-данните в скоби се посочват съответните стойности, с която да бъдат инициализирани. Инициализиращия списък се изпълнява преди да се влезе в тялото на конструктора.

В случая член-данните и параметрите на конструктора имат еднакви имена. Вътре в скобите е името на параметъра, с чиято стойност ще бъде инициализирана съответната характеристика. Идентификаторът на характеристиката е извън скобите.

В какъв ред се случва инициализацията? Независимо от реда, в който член-данните са изброени в инициализиращия списък, те се инициализират в реда, в който са декларирани в дефиницията на класа.

В инициализиращия списък могат да се инициализират онези член-данни, за които не се налага верификация на стойностите.

Деструктор

Деструкторът е специална член-функция на класа, която се извиква автоматично, когато обектът излезе от област на видимост. Ако е обект, създаден в динамичната памет, за него трябва да бъде извикан `delete`.

Идея

Целта на деструктора е да почисти след обекта. За класове, в които конструкторите само инициализират обикновени член-данни, не е необходимо да се дефинира деструктор. Такъв ще бъде генериран системно.

Ако при инициализирането на обекта се ангажира някакъв външен ресурс (заделя се динамична памет, отваря се файл или се създава връзка към база от данни), деструкторът е точното място, на което този ресурс може да бъде освободен. Това е последната функция, която се извиква преди обектът да бъде унищожен.

Синтаксис

Деструкторът носи името на класа с ~ отпред. Няма никакви аргументи и не връща никакъв резултат. За разлика от конструкторите, един клас може да има точно един деструктор.

```
Student::~Student()
{ delete [] this->name;}
```

Деструкторът може да извиква други член-функции в себе си, тъй като обектът бива унищожен чак след като деструкторът приключи.

Разлика между delete и delete[]

Нека е създаден динамичен обект от тип `Student`.

```
Student* ivan = new Student {"Ivan", 1, 4};
```

Когато бъде извикан операторът `delete`, се извиква деструкторът за създадения динамичен обект.

Ако е създаден масив от студенти:

```
Student* group5 = new Student[25];
```

Всеки един от обектите в масива ще бъде инициализиран с помощта на конструктор по подразбиране. Ако такъв не съществува в съответния клас, то опитът да се създаде масив предизвиква грешка.

Ако, за да се освободи динамичния масив, се извика операторът

```
delete group5;
```

ще бъде извикан деструктор само за първия обект от масива, което ще предизвика грешка по време на изпълнение или изтичане на памет. За да бъде извикан деструктор за всички обекти в масива, трябва да бъде използвана правилната форма на оператора `delete`:

```
delete [] group5;
```

Операция за присвояване

Идея

Когато обектите вече са създадени, те могат да бъдат променени с помощта на операцията за присвояване.

```
Rational r (1, 2);
Rational r1;

r1 = r;
```

Двата обекта вече са създадени. Първият с помощта на конструктор с параметри, а вторият с помощта на конструктора по подразбиране.

Внимание!

```
Rational r (1, 2);
Rational r1 = r;
```

Въпреки че отново се използва =, в този случай не става въпрос за присвояване. Обектът `r1` се създава сега. Това означава, че се извиква копиращ конструктор.

За да се изпълни копирането на данни от `r` в `r1`, се използва системният оператор за присвояване, който подобно на системният копиращ конструктор копира данните едно към едно.

Синтаксис

Копирането в случая на рационалните числа е тривиално, но това не винаги е така. Например в случая със студента, чието име се съхранява в динамичната памет. Какви проблеми могат да възникнат, ако се използва системната?

В този случай операцията за присвояване трябва да бъде дефинирана изрично. Тя има следният вид:

```
Student& Student::operator=(const Student& other)
{
	if (this != &other) // проверка за самоприсвояване
	{
		delete[] this->name;

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

		this->fn = other.fn;
		this->grade = other.grade;
	}

	return *this;
}
```

Резултатът от присвояването е променения текущ обект. Това е така, за да е възможно да се направи верижно прилагане на операцията за присвояване. Обектът, който се копира се предава по референция и е достъпен само за четене. Копират се всички член-данни. Не се прави валидация, защото копирането се извършва от вече създаден валиден обект.

Защо е нужна проверката за самоприсвояване?

Полезни връзки:

https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)

https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

https://en.wikipedia.org/wiki/Destructor_(computer_programming)

https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html

https://en.cppreference.com/w/cpp/language/member_functions#Special_member_functions