

Обектно-ориентирано програмиране (записки)

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

Тема 14. Type casting. SOLID principles. Design patterns

Type casting

def| когато искаме да преобразуваме от един тип в друг

```
int main()
{
    int x = 7;
    double y = x; //преобразуваме от int в double

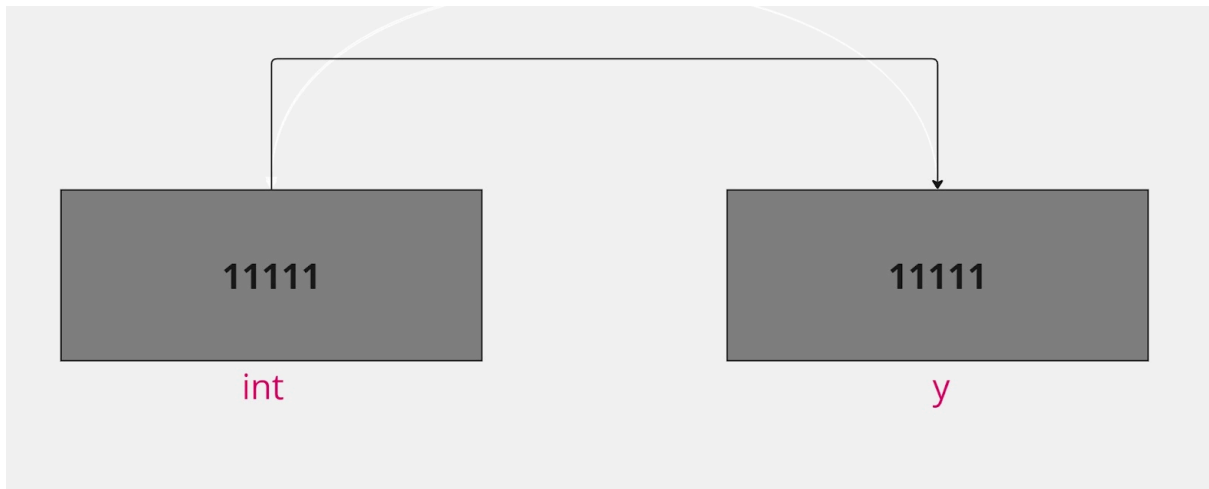
    return 0;
}
```



```
int main()
{
    int x = -1;
    unsigned y = x;

    std::cout << y << std::endl;
    //мук ще се отпечата MAX на min int
    //заради побитовите им стойности

    return 0;
}
```



```
#include <iostream>

class Base{};
class Der:public Base{};

int main()
{
    Der* d = new Der();
    Base* b = d; // става на Base* b = (Base*)
                //моем C-style cast (неявно кастване)

    return 0;
}
```

Static_cast

- шаблонна функция е
- използваме го само когато сме сигурни в типа, в който преобразуваме
- използваме го за преобразуване на примитивни типове
- използваме за upcasting (Der* -> Base*)
- не прави runtime check, тоест ако възникне грешка - crash/undefined behaviour
- използва се много в стари C библиотеки

```
class Base
{
public:
    int virtual getType();
};

class A:public Base
{
public:
    int getType() override { /* */ };
};

class B :public Base
{
public:
    int getType() override { /* */ };
};

void f(Base* ptr)
{
    if (ptr->getType() == 1)
    {
        A* obj = static_cast<A*>(ptr); //downcasting
        //в случая, сме сигурни, че обектът е от тип A
        //и можем да cast-нем наголу
    }

    /*...*/
}
```

```
int main()
{
    Base* ptr = new A();
    B* b = static_cast<B*>(ptr); //(!) това ще крашне или ще доведе до UB
    //нямаме проверка, която да види, че ptr НЕ сочи
    //към обект от тип B, а към обект от тип A)

    return 0;
}
```

```

void f(int x, void* ptr)
{
    int* data = static_cast<int*>(ptr); //за примитивни типове
}

int main()
{
    int x = 0;
    void* ptr = &x;
    f(x, ptr);

    return 0;
}

```

Dynamic_cast

- преобразуваме, но с runtime check (която става чрез виртуалната таблица, където се пази типа), т.е. ако работим с указатели и преобразуването е неуспешно, но dynamic_cast ще върне nullptr, но ако работим с референции, ще хвърли std::bad_cast
- не се използва за upcasting
- използва се основно за downcast, когато не знаем какъв е типа
- по-бавно от static_cast

```

class Base
{
    void virtual g(); //dynamic_cast може да downcast-ва единствено
                    //когато имаме поне една виртуална таблица
};

class A: public Base{};
class B: public Base{};

void f(Base* ptr)
{
    if (A* aPtr = dynamic_cast<A*>(ptr)) //ще се замести с aPtr и ще провери
    {                                     //дали е nullptr
        //
    }

    else if (B* bPtr = dynamic_cast<B*>(ptr)) //аналогично
    {
        //
    }
}

```

```

void g(Base& ref)
{
    A& refA = dynamic_cast<A&>(ref); //при неуспешен cast хвърля std::bad_cast
}

```

Const_cast

- **не го използваме**
- идеята е манипулиране на константност на обекти, с които работим чрез указатели и референции
- за да премахнем const трябва да сме сигурни, че при създаването си този обект не е бил const
- ако първоначално е бил деклариран като const => **Undefined Behaviour** (константите седят на друго място в паметта)

```

void f(const int* ptr) //забранено е да го променяме (const)
{
    int* n = const_cast<int*>(ptr); //махаме константността
    (*n)++; //вече не е const и можем да го променяме
}

int main()
{
    int x = 10; //първоначално не е const

    f(&x);

    std::cout << x << std::endl; //11

    return 0;
}

```

```

void f(const int* ptr)
{
    int* n = const_cast<int*>(ptr); //[!] UB, константите имат специално място в паметта
    //и не можем да ги местим от там
}

int main()
{
    const int x = 10; //[!] първоначално е const

    f(&x);

    return 0;
}

```

Reinterpret_cast

- използва се за преобразуването на pointer от даден тип към pointer от друг тип, дори типовете да не съвпадат (не прави проверка)
- реинтерпретация на памет - работи като Union
- използва се, когато искаме да работим с битовите (напр. двоични файлове)

```
class X
{
public:
    int a;
    int b;
};

class Y
{
public:
    char ch[5];
};

int main()
{
    X* ptrX = new X();

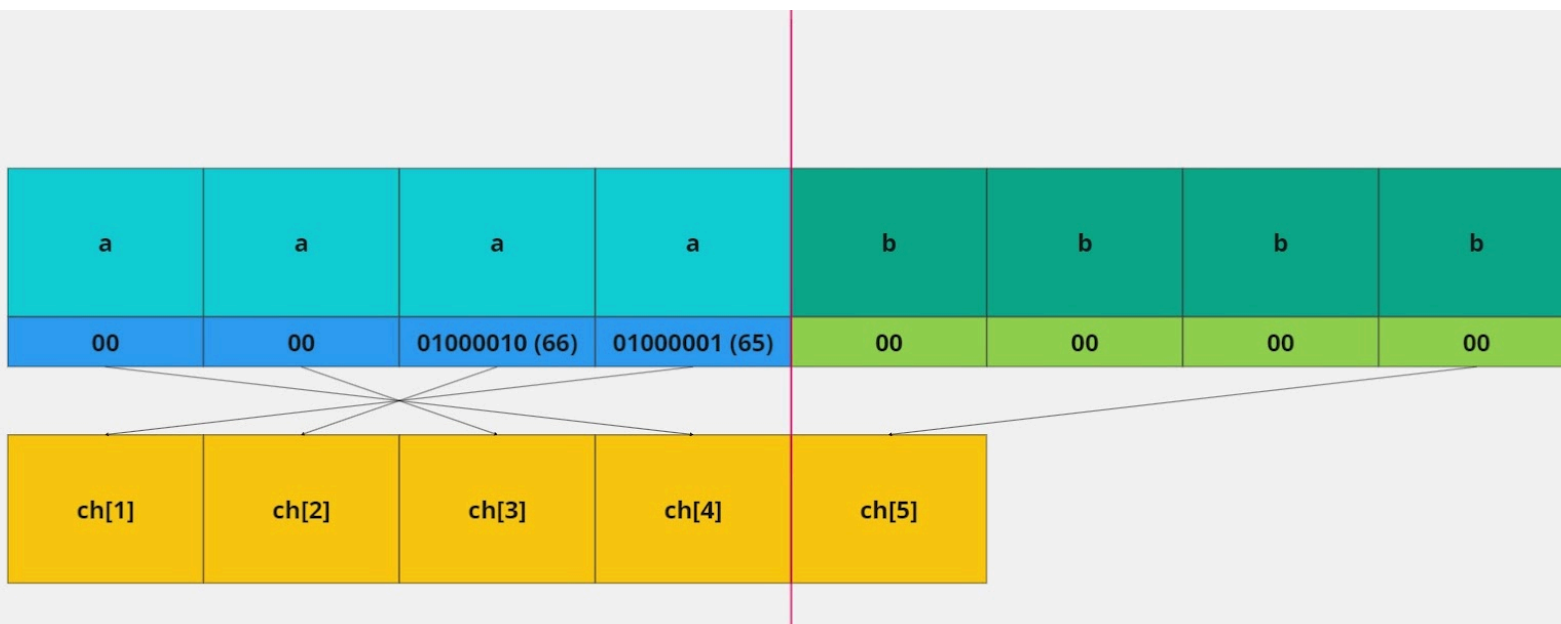
    ptrX->b = 4294967106; //11111111 11111111 11111111 01000010
    ptrX->a = 16961; //00000000 00000000 01000010 01000001

    Y* ptrY = reinterpret_cast<Y*>(ptrX);

    std::cout << ptrY->ch[4] << std::endl;
    std::cout << ptrY->ch << std::endl;

    return 0;
}
```

[!] Подобно на Union



```

#include <iostream>
#include <fstream>

int main()
{
    std::ofstream ofs("file.dat", std::ios::binary);

    int x = 5;
    ofs.write(reinterpret_cast<const char*>(&x), sizeof(x)); //вече така ще използваме .write()
                                                             // .read()
    return 0;
}

```

C-style cast

- изпълнява последователно: _

→ const - cast
 → static - cast
 → static - cast + const - cast
 → reinterpret - cast
 → reinterpret - cast + const - cast

[!] рано или късно
 някой от кастовите ще
 успее

```

#include <iostream>
#include <fstream>

class A
{
public:
    int a;
    int b;
};

class B: public A
{
public:
    char ch[5];
};

int main()
{
    A* ptr2 = new A();
    A* ptr = (B*)ptr2; //C-style cast

    return 0;
}

```


SOLID Principles

Single responsibility principle - 1 компонент има точно 1 отговорност

cohesion - доколко компонентите са свързани
(разделяме ги в класове по **strong cohesion**)

Open-closed principle - отворен за разширение, но затворен за модификация (класът да не се променя при добавянето на наследник)

Liskov substitution principle - трябва да използваме указатели/референции от базовия клас, без да се интересуваме към кой наследник е насочен

Interface segregation - потребителите не трябва да разчитат на интерфейс, който не използват (правим класове с точно и ясно предназначение)

Dependency Inversion principle - модулите от високо ниво не трябва да зависят от модулите на ниско ниво (класовете трябва да зависят от интерфейси и абстрактни класове, не от конкретни класове и функции)

Design Patterns

def| **Design Patterns**

- обобщени практики (добри)
- решения на често възникващи проблеми
(не специфичен код, а концепция за решение)

3 вида:

1. **creational patterns** - осигуряват създаването на обекти, като скриват логиката по тяхното създаване (factory)
2. **structural patterns** - начин за създаване на по-сложни обекти, използвайки инструменти като наследяване и композиция
3. **behavioral patterns** - комуникация между обектите (visitor)

Singleton

- creational pattern
- осигурява само една инстанция за даден клас, към която има глобален достъп (пр. StringPool)
- private конструктор и deleted copy constructor и operator=
- функция getInstance()

Плюсове

- имаме само една инстанция на класа
- имаме глобален достъп до нея
- обектът се инициализира при първото достъпване на обекта (lazy initialization)

Минуси

- може да доведе до многонишково програмиране
- много често се определя като антипатърн
 - > не може да се тества при използване на друг код
 - > обвързва се с конкретна инстанция

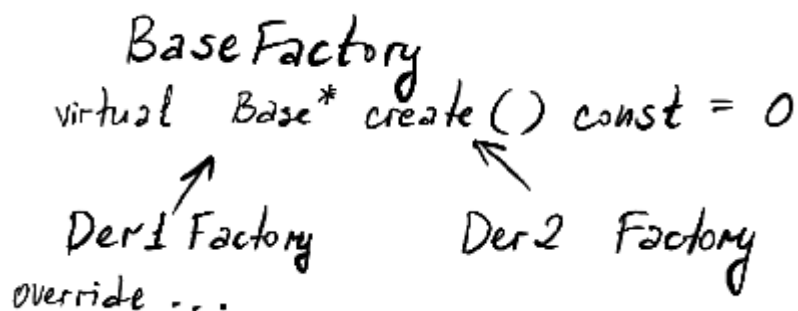
Factory

- creational pattern
- статична функция (може в клас), която на база подаден аргумент връща инстанция на клас

Factory Method

- creational pattern
- предоставя интерфейс с точно един create method
- всеки наследник на този базов клас презаписва имплементацията на create, в която разписва какъв обект да върне

BaseFactory
virtual Base* create() const = 0
Der1Factory Der2 Factory
override ...



Abstract Factory

- представлява интерфейс с по един create за всички обекти, които създава
- обектите са свързани логически

Prototype (clone)

- creational pattern
- създаване на копие на обект (от полиморфна йерархия) без да се интересуваме какъв е типът

Composite

- structural pattern
- композиране на обекти в дървовидна структура
- листа и междинни обекти
- BooleanExpression

Flyweight Pattern

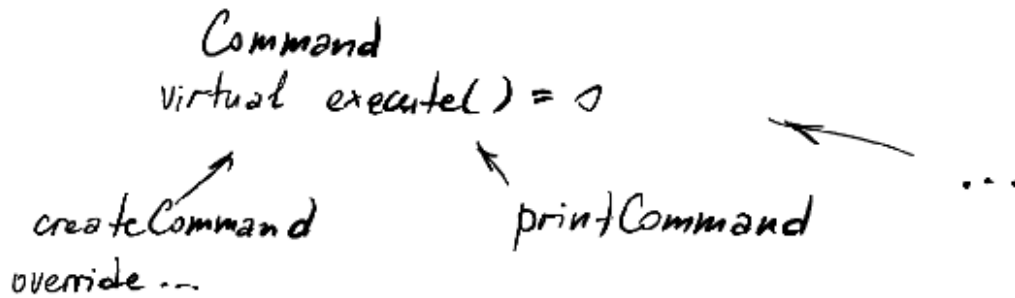
- structural pattern
- събира повече обекти в паметта като споделя общите им ресурси
- подобрява бързодействието, ако създаването е тежка операция
- StringPool

Iterator Pattern

- behavioral pattern
- начин за работа с колекция без да се интересуваме каква е тя
- има итератор - указател към конкретен елемент
- *it - връща елемент, op++, op--, op!=, op==
- колекциите трябва да имат следния интерфейс
 - > begin() - връща итератор към началото
 - > end() - връща итератор към края

Command Pattern

- behavioral
- програма, която получава заявки



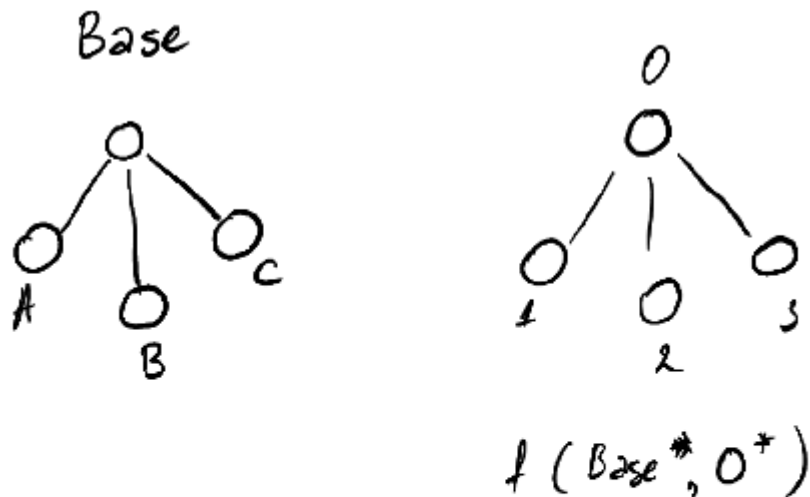
Handwritten code snippet:

```

Command * curr = CommandFactory (...)
curr -> execute()
  
```

Visitor Pattern

- когато си взаимодействат обекти от полиморфна йерархия (например член-функция на наследник, на която се подава обект от друг наследник)



разнозначно



Вика групата йерархия