Objektově orientované programování

Dědičnost - polymorfismus

2023/24

Osnova přednášky

- Vztah přetížení, překrytí a *protected* přístupu.
- Typy překrytí, polymorfismus.
- Příklad.

Přetížení, překrytí, *protected*

Přetížení x překrytí

- Přetížením řešíme „doplnění“ chování. Jedná se o rozšíření, byť má metoda stejné jméno.

- Překrytím řešíme opravdovou změnu chování.

- *Polymorfismus* je ještě něco navíc…

Problém změny chování

- Mnohdy potřebujeme přistupovat k detailům implementace.
- Detaily implementace by ale měly být skryty.
- Dá se přistoupit k soukromým položkám předka (předků)?

Přístup ke stavu a chování

	public	private	protected
klient	x	-	-
třída	x	x	x
potomek	x	-	x

Protected přístup

- Přístup k detailům implementace můžeme řešit využitím *„protected“*.
- Je to ale správné?
- Nebo je to špatně? *A proč?*

```
class Account
{
private:
    int number;
    double interestRate;

    Client *owner;

protected:
    double balance;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);

    bool CanWithdraw(double a);
    bool Withdraw(double a);
};
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

Použití *protected*…

- …porušuje zapouzdření

- Důsledky:
  - Rozhodneme-li se změnit implementaci předka, může se to dotknout implementace potomka.

  - Potomek se stává implementačně závislým na předkovi (a platí to i naopak).

Co je polymorfismus?

Polymorfismus

- *Polymorfismus* je schopnost objektu vystupovat v různých rolích…

- …a podle toho se chovat.
  - **Kombinuje** své chování s chováním předka, jinak se o skutečný polymorfismus nejedná…

- Souvisí to se substitučním principem, tedy se zastupitelností předka potomkem.

Polymorfní přiřazení

- Zdroj přiřazení je jiného typu než cíl přiřazení.

```
Client *o = new Client(0, "Smith");
CreditAccount *ca = new CreditAccount(1, o, 1000);
```

```
Account *a = ca;
```

Vlastnost polymorfního přiřazení

Feature Call rule

In a feature call $x.f$, where the type of x is based on a class C, feature f must be defined in one of the ancestors of C.

Překrytí x polymorfismus

- Zajišťuje nám „obyčejné překrytí“ to, že se jedná polymorfismus?

- **NE!**

- Proč?

- Protože potomek se při překrytí chová v roli předka přesně jako tento předek (chování se tedy **nekombinuje**).

Bez *„protected“*?

- Jak jinak získat přístup k soukromým položkám předka?

- Když jsou tyto položky pro potomka díky *„private“* skryty…

Co vlastně chceme?
Vraťme se o krok zpět…

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);

    bool CanWithdraw(double a);
};
```

```
bool Account::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

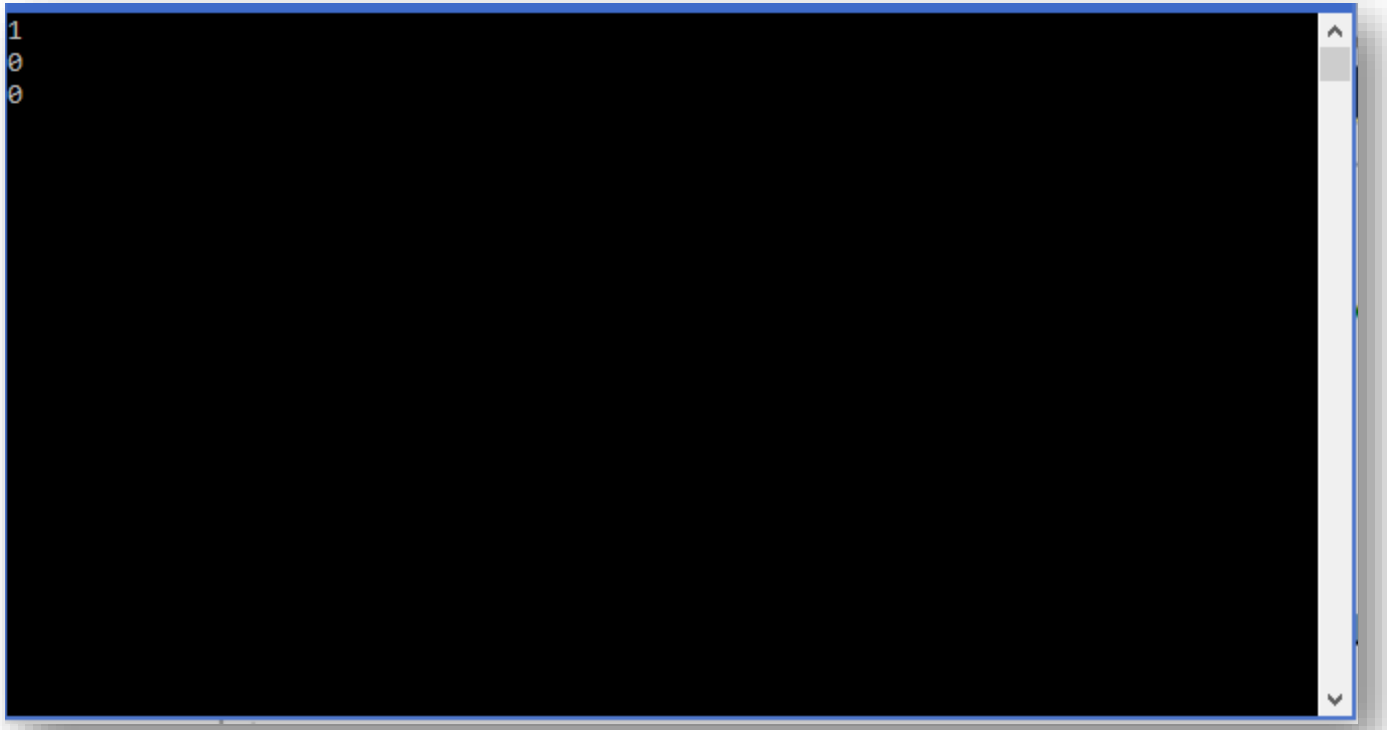
```
bool Account::CanWithdraw(double a)
{
    return (this->balance >= a);
}
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

Funguje to?

```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    cout << ca->CanWithdraw(1000) << endl;

    Account *a = ca;

    cout << a->CanWithdraw(1000) << endl;
    cout << ca->Withdraw(1000) << endl;

    a = nullptr;
    delete ca;

    getchar();
    return 0;
}
```

Ne tak, jak bychom očekávali…



Časná vazba (*early binding*)

- Překladač za normálních okolností využívá tzv. časnou vazbu (*early binding*), která při volání metody vyhodnocuje typ instance již v době překladu.

- V metodě *Withdraw* se zavolá metoda *CanWithdraw* předka.

Pozdní vazba (*late binding*)

- Potřebujeme zjistit, kdo metodu žádá, ale až v okamžiku volání.
- To v našem případě nelze, protože je použita časná vazba.
- K tomu slouží pozdní vazba (*late binding*).

Shadowing x Overriding

- *Shadowing (method hiding).* Jde o statické překrytí, kdy nová metoda potomka „zastíní“ metodu předka
  - Dílčí chování objektu tedy odpovídá třídě, v jejíž roli vystupuje.
- *Overriding*. Jde o dynamické překrytí, kdy se vždy (i v roli předka) použije metoda potomka, pokud ji má implementovanou.
  - Dílčí chování objektu tedy odpovídá třídě, jejíž je tento objekt instancí.

Příklad

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    virtual bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```


1
1
1

Virtuální metody

- Chceme-li se přenechat rozhodnutí, která překrytá metoda bude volána, až v průběhu programu (*overriding*), musíme metodu označit klíčovým slovem *virtual*.

- Dáváme překladači najevo, že si přejeme využít dynamickou nebo také pozdní vazbu (*late binding*).

- Jednou označená metoda jako **virtuální zůstává virtuální** ve všech potomcích!!!

Tabulka virtuálních metod (VMT)

- Jakmile některou metodu definujeme jako virtuální, překladač přidá ke třídě „neviditelný ukazatel“, který ukazuje do speciální tabulky nazvané tabulka virtuálních metod (*VMT*).
- Pro každou třídu, která má alespoň jednu virtuální metodu, překladač vytvoří tabulku virtuálních metod.
- Tabulka obsahuje ukazatele na virtuální metody.
- Tabulka je společná pro všechny instance dané třídy.

Virtuální konstruktor?

- **NE!**
- Před jejich voláním není ještě vytvořen odkaz do *VMT*.
- Uvnitř konstruktorů sice můžeme volat virtuální metody, ale ty se budou chovat nevirtuálně.

Virtuální destruktor?

- **ANO!**

```
CreditAccount *ca = new CreditAccount(1, 0, 1000);
Account *a = ca;
delete a;
```

- Který destruktor se v případě, že není virtuální, zavolá? Je to správně? A proč? A který destruktor se zavolá, pokud virtuální je?

Polymorfismus

- Polymorfismus je spojen s dědičností.
- Nemá smysl mluvit o polymorfismu, pokud nepoužijeme virtuální metody (*overriding*).
- Jde stále o zastupitelnost předka potomkem.

Virtuální metody

- Potomek využívá virtuální metodu v různých kontextech:
  - V případech, kdy je tato virtuální metoda použita v těle kterékoli metody kteréhokoli předka.
  - Na rozdíl od obyčejného překrytí (*shadowing*) i v případě polymorfního přiřazení.

Polymorfní datové struktury

- Struktura která obsahuje objekty různých tříd.
  - Např. pole, seznam,…, který je typu „Předek“ (ukazatel)

- Po takto uložených objektech můžeme požadovat (volat) pouze společné metody předka.

- Jak volat ostatní metody objektu vráceného v typu předka?
  - Je nutno přetypovat – je to jedno z omezení polymorfismu.

Úkoly na cvičení

- Implementuje příklady z přednášky, zaměřte se na využití virtuální metody a pochopení, jak funguje při polymorfním přiřazení.

- Navrhněte a implementuje jednoduchou dědičnou hierarchii geometrických objektů, které budou mít společné virtuální metody „Obsah“ a „Obvod“. Využijte polymorfní datovou strukturu (např. pole ukazatelů) a rozeberte chování při využití substitučního principu (zejména při srovnání s obyčejným překrytím).

Kontrolní otázky

- Jaký je rozdíl mezi *shadowing* a *overriding* překrytím? Uveďte příklady
- Co rozumíme polymorfismem a s čím to souvisí?
- Co rozumíme polymorfním přiřazením?
- Co je časná vazba? Uveďte příklady.
- Co je pozdní vazba? Uveďte příklady.
- Popište, co je virtuální metoda a její vlastnosti.
- Popište, co je tabulka virtuálních metod a jak funguje.
- Může být konstruktor virtuální? A proč?
- Může být destruktor virtuální? A proč?
- Kdy mluvíme v C++ o polymorfismu a jak se to projeví v návrhu?
- Co je polymorfní datová struktura a k čemu ji využíváme?
- Kdy potřebujeme virtuální destruktor? S čím to souvisí?

Ke studiu

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [467-472]