

Objektově orientované programování

Filip Rosa

2025

Obsah

Obsah	1
1 Modularita	4
1.1 Co je hlavním motivem pro vývoj programovacího paradigmatu od imperativního k objektovému?	4
1.2 Co je imperativní programování?	4
1.3 Co je modulární programování?	4
1.4 Jaké jsou hlavní faktory kvality software?	4
1.5 Co je pochopitelnost modulu? Uveďte příklad.	4
1.6 Co je samostatnost modulu? Uveďte příklad.	4
1.7 Co je kombinovatelnost modulu? Uveďte příklad.	5
1.8 Co je zapouzdření modulu? Uveďte příklad.	5
1.9 Co je explicitní rozhraní modulu? Uveďte příklad.	5
1.10 Co je syntaktická podpora modularity?	5
1.11 Co je pět kritérií pro dobrou modularitu?	5
1.12 Co se rozumí pěti pravidly zajišťující dobrou modularitu?	5
1.13 Popište jednotlivá kritéria dobré modularity. Uveďte příklady.	6
1.14 Popište jednotlivá pravidla pro dobrou modularitu. Uveďte příklady.	6
1.15 K čemu je konstruktor? Uveďte příklad.	7
1.16 K čemu je destruktore, kdy ho potřebujeme a kdy ne? Uveďte příklad.	7
2 Třídy a objekty	8
2.1 Co je hlavními příčinami potřeby změn software?	8
2.2 Jaké jsou hlavní faktory ovlivňující objektovou orientovanost?	8
2.3 Vysvětlete, co rozumíme pojmy objektově orientovaná metoda (přístup) a jazyk.	8
2.4 Vysvětlete, co rozumíme podporou objektově orientované implementace.	9
2.5 Vysvětlete, co rozumíme podporou opakované použitelnosti.	9
2.6 Vysvětlete pojmy třída a objekt a použijte správnou terminologii.	9
2.7 Zdůrazněte vlastnosti třídy z pohledu modularity.	9
2.8 Vysvětlete princip zapouzdření v OOP.	9
2.9 Vysvětlete princip zasílání zpráv.	9
2.10 Vysvětlete principy deklarace a definice jednoduché třídy v C++.	9
3 Návrh programu I	11
3.1 Vysvětlete, jak vznikají objekty třídy, pojem konstruktor a principy práce s ním v C++.	11
3.2 Vysvětlete, jak zanikají objekty třídy, pojem destruktore a principy práce s ním v C++.	11
3.3 Vysvětlete rozdíl mezi statickou a dynamickou deklarací objektů v C++.	11
3.4 Jak se dá postupovat, pokud chceme v zadání programu nalézt třídy, jejich metody a datové členy?	11
3.5 Kdy a proč potřebujeme použít více konstruktorů jedné třídy?	11
3.6 Kdy potřebujeme deklarovat a definovat destruktore?	11
3.7 Co jsou výchozí konstruktory a destruktory a k čemu je potřebujeme?	11
3.8 Jaké typy metod obvykle musíme deklarovat a definovat?	12

3.9	Co jsou objektové kompozice a k čemu jsou dobré?	12
4	Objektová dekompozice a třída jako objekt	13
4.1	Jaký je rozdíl mezi funkční a objektovou dekompozicí programu?	13
4.2	Proč preferujeme objektovou dekompozici a jaké jsou hlavní problémy funkční dekompozice?	13
4.3	Za jakých podmínek můžeme považovat třídu za objekt a jak to implementovat v C++?	13
4.4	Vysvětlíte rozdíl mezi členskými položkami třídy a instance a popište jejich dostupnost.	13
4.5	Jak můžeme v C++ důsledně odlišovat práci s členskými položkami tříd a instancí?	13
4.6	Potřebuje třída v roli objektu konstruktor resp. destruktory a proč? . . .	14
5	Úvod do dědičnosti	15
5.1	Které dva klíčové požadavky řešíme pomocí dědičnosti?	15
5.2	Jaké návrhové požadavky máme na použití tříd (co s nimi můžeme dělat)?	15
5.3	Jaký je rozdíl mezi dědičností a skládáním? Co mají společného?	15
5.4	V jakých rolích vystupují třídy v dědičnosti? Použijte správnou terminologii.	15
5.5	Vysvětlíte v jakém obecném vztahu je třída, ze které se dědí, se třídou, která dědí.	15
5.6	Co všechno se dědí, co ne a proč?	15
5.7	Co rozumíme jednoduchou dědičností a jak s tím souvisí hierarchie tříd v dědičnosti?	15
5.8	Co je Liskovův substituční princip a jak se projevuje v dědičnosti?	15
5.9	V jakém pořadí se volají a vykonávají konstruktory při použití dědičnosti?	16
6	Dědičnost – změna chování	17
6.1	Co rozumíme paradoxem specializace a rozšíření?	17
6.2	Uveďte správné a špatné příklady vztahu "generalizace-specializace". . .	17
6.3	Co rozumíme v dědičnosti změnou chování?	17
6.4	Co rozumíme přetížením? Jedná se o rozšíření nebo změnu chování? . . .	17
6.5	Uveďte různé typy přetížení.	17
6.6	Co rozumíme překrytím? Jedná se o rozšíření nebo změnu chování? . . .	17
6.7	Jaký princip porušujeme, použijeme-li „protected“ a proč?	18
6.8	Jaký problém přináší potřeba změny chování v dědičnosti?	18
6.9	Popište, jak se prakticky projevuje různá míra přístupu k položkám třídy.	18
6.10	Jak se použití „protected“ projeví ve vztahu předka a potomka?	18
7	Dědičnost - polymorfismus	19
7.1	Jaký je rozdíl mezi shadowing a overriding překrytím? Uveďte příklady. .	19
7.2	Co rozumíme polymorfismem a s čím to souvisí?	19
7.3	Co rozumíme polymorfním přiřazením?	19
7.4	Co je časná vazba? Uveďte příklady.	19
7.5	Co je pozdní vazba? Uveďte příklady.	19
7.6	Popište, co je virtuální metoda a její vlastnosti.	19
7.7	Popište, co je tabulka virtuálních metod a jak funguje.	19
7.8	Může být konstruktor virtuální? A proč?	19
7.9	Může být destruktory virtuální? A proč?	20

7.10	Kdy mluvíme v C++ o polymorfismu a jak se to projeví v návrhu? . . .	20
7.11	Co je polymorfní datová struktura a k čemu ji využíváme?	20
7.12	Kdy potřebujeme virtuální destruktory? S čím to souvisí?	20

1 Modularita

1.1 Co je hlavním motivem pro vývoj programovacího paradigmatu od imperativního k objektovému?

Hlavním motivem pro prechod od imperatívneho programovania k objektovo orientovanému programovaniu je zjednodušenie riadenia komplexity pri vývoji a údržbe veľkých softvérových systémov. Tento prechod priniesol niekoľko kľúčových výhod, ktoré sa týkali najmä organizácie kódu, opakovateľnosti, rozšíriteľnosti a správy stavu aplikácií.

1.2 Co je imperativní programování?

Poznáme ho z bežne používaných jazykov. Môžeme ho rozdeliť na dva druhy:

- **procedurálne programovanie** - postupnosť krokov, ktorými meníme stav premenných programu
- **štruktúrované programovanie** - sekvencie, iterácie, vetvenie, skoky, abstrakcia

1.3 Co je modulární programování?

Je to návrh zhora-dolu. Rozdeľuje program na nezávislé, zameniteľné moduly, ktoré zaisťujú jednotlivé drobné funkčnosti. Modul obsahuje všetko potrebné pre zaistenie funkčnosti (dáta a algoritmy).

1.4 Jaké jsou hlavní faktory kvality software?

- vnútorné sú schované pred používateľom (AKO)
- vonkajšie popisujú správanie navonok - správnosť, robustnosť, rýchlosť, rozšíriteľnosť... (ČO)
- musíme byť schopní merať kvalitu software

1.5 Co je pochopitelnost modulu? Uved'te příklad.

Modul by mal vykonávať jednu jasne definovanú a pochopiteľnú úlohu, prípadne niekoľko málo jasne definovaných úloh.

Napr. modul pre prácu s dátumami funkciou `is_leap_year` jednoducho zisťuje, či je rok priestupný, a `days_in_month` vracia počet dní v danom mesiaci.

1.6 Co je samostatnost modulu? Uved'te příklad.

Každý modul musí byť relatívne samostatný a mal by mať čo najmenej väzieb na ostatné moduly. Nebolo by vhodné, aby všetky moduly programu boli navzájom prepojené a na seba závislé.

Napr. modul pre základné operácie s textom. Vykonáva dve nezávislé operácie: konverziu textu na veľké písmená a počítanie samohlások. Nevyžaduje žiadne externé knižnice alebo konfigurácie. Môže byť použitý samostatne v akomkoľvek projekte.

1.7 Co je kombinovatelnost modulu? Uved'te příklad.

Moduly musia byť navzájom kombinovateľné. Musí byť možné modul vziať a použiť v inom kontexte alebo v inom projekte.

Napr. máme dva moduly: jeden na spracovanie textu a druhý na spracovanie čísiel. Oba moduly budú kombinovateľné v rámci väčšej aplikácie, kde budú spolupracovať. Každý z nich vykonáva jasnú úlohu a má jednoduché rozhranie.

1.8 Co je zapouzdření modulu? Uved'te příklad.

Moduly musia mať právo na isté súkromie: je prípustné a žiadúce, aby všetky informácie, ktoré nie sú potrebné pre klientov modulu zostali skryté vnútri modulu. Väčšina funkcionality modulu je skrytá a len malá časť je viditeľná zvonku. Skrytej časti vravíme **implementácia** modulu a verejnej časti **rozhranie** modulu.

Napr. Máme triedu Car, ktorá reprezentuje automobil. Tento automobil má súkromné údaje, ako je aktuálny počet paliva v nádrži. Chceme, aby tieto údaje boli prístupné iba prostredníctvom metód triedy a neboli priamo meniteľné z vonkajšieho kódu.

1.9 Co je explicitní rozhraní modulu? Uved'te příklad.

Z deklarácie modulu musí byť všeobecne zrejmé, aké predpoklady pre vykonávanie svojej úlohy potrebuje.

Napr. Vytvárame modul na spracovanie jednoduchých matematických operácií. Tento modul bude mať jasne definované funkcie pre sčítateľ, odčítateľ, vynásobiť a vydeliť čísla, ktoré tvoria jeho explicitné rozhranie.

1.10 Co je syntaktická podpora modularity?

Moduly počítačového programu musia byť jasne vymedzené syntaktickými jednotkami programu. Zo zápisu programovacieho jazyka musí byť zrejmé, kde končí a začína zápis modulu.

1.11 Co je pět kritérií pro dobrou modularitu?

- dekomponovateľnosť
- kombinovateľnosť
- pochopiteľnosť
- kontinuita
- ochrana

1.12 Co se rozumí pěti pravidly zajišťující dobrou modularitu?

- priame mapovanie
- pár rozhraní
- malé rozhrania

- explicitné rozhrania
- skrývanie informácií

1.13 Popíšte jednotlivá kritéria dobrej modularity. Uveďte príklady.

- dekomponovateľnosť - schopnosť rozložiť veľký a komplexný systém na menšie, samostatné komponenty. Predstavme si, že vyvíjame webovú aplikáciu pre správu objednávok. Celkový systém môžeme rozdeliť na menšie moduly, ako sú: modul pre správu používateľov: Registrácia, prihlásenie, správa profilov. Modul pre spracovanie objednávok: Vytváranie objednávok, platby, správa stavu objednávky. Modul pre spracovanie platieb.
- kombinovateľnosť - schopnosť spájať rôzne moduly alebo komponenty do väčších celkov, ktoré spolupracujú, aby vykonávali zložitejšie úlohy. Zoberme si modul na spracovanie platieb v online obchode. Tento modul môže byť kombinovaný s rôznymi ďalšími modulmi ako: Modul pre spracovanie objednávok: Keď používateľ vytvorí objednávku, modul pre spracovanie objednávok využije modul na spracovanie platieb na dokončenie transakcie. Modul pre notifikácie.
- pochopiteľnosť - systém alebo modul je ľahko čitateľný a zrozumiteľný pre vývojárov alebo iných používateľov systému - premenné sú popisné, názov funkcie je jasný, kód je čitateľný
- kontinuita - systém alebo modul by mal fungovať bez prerušenia aj pri zmenách alebo aktualizáciách. Ak aktualizujeme databázový systém, musíme zabezpečiť, že staršie verzie systému budú fungovať aj po implementácii novej verzie.
- ochrana - zabezpečenie systému pred neautorizovaným prístupom, chybami alebo nepredvídanými situáciami. - autentifikácia, ochrana údajov, validácia vstupov

1.14 Popíšte jednotlivá pravidla pro dobrou modularitu. Uveďte príklady.

- priame mapovanie - existuje jednoznačný a priamy vzťah medzi internými komponentmi alebo dátovými štruktúrami a rozhraním modulu. Predstavme si triedu na reprezentovanie bankového účtu. Ak by sme použili priame mapovanie, poskytl by sme metódy, ktoré priamo manipulujú s internými hodnotami (ako je zostatok).
- pár rozhraní - modul poskytuje dve alebo viac rozhraní, ktoré sa vzájomne dopĺňajú a používajú sa spoločne. Predstavme si systém na správu súborov, ktorý poskytuje rozhranie na čítanie a zapisovanie do súborov, ale každé z týchto rozhraní môže byť prispôbené pre rôznych používateľov (napr. pre čitateľov a zapisovačov).
- malé rozhrania - rozhranie modulu je minimalizované na čo najmenší počet metód alebo funkcií, ktoré vykonávajú jasne definovanú a konkrétnu úlohu. Predstavme si modul na validáciu používateľských údajov. Tento modul môže mať malé a špecifické rozhranie. Modul poskytuje len dve metódy, ktoré vykonávajú veľmi špecifické úlohy – validáciu e-mailu a validáciu hesla.

- explicitné rozhrania - rozhranie modulu je jasne definované a dokumentované. Rozhranie modulu je explicitné, pretože používateľ má jasne definované metódy na pridávanie položiek a získanie celkovej ceny. Modifikácia interných dát (napríklad priamy prístup k zoznamu položiek) nie je povolená.
- skrývanie informácií - interné detaily implementácie sú skryté pred používateľmi modulu. predstavme si triedu na správu bankového účtu, ktorá skrýva interný stav (ako je zostatok) a poskytuje len verejné metódy na interakciu.

1.15 K čemu je konštruktor? Uveďte príklad.

Konštruktor inicializuje dáta objektu hodnotami parametrov v konštruktori (naplní pamäť dátami).

1.16 K čemu je deštruktor, kedy ho potrebujeme a kedy ne? Uveďte príklad.

Deštruktor odstráni v pamäti dáta objektu (čistí pamäť). Nie je potrebný pokiaľ sú dáta objektu statické.

2 Třídy a objekty

2.1 Co je hlavními příčinami potřeby změn software?

1. Technologický pokrok
2. Zmeny v užívateľských požiadavkách
3. Bezpečnosť
4. Chyby a problémy
5. Zmeny v legislatíve
6. Zastaralosť softwaru
7. Zlepšenie užívateľského zážitku
8. Prechod na cloud alebo inú architektúru
9. Konkurenčný tlak

2.2 Jaké jsou hlavní faktory ovlivňující objektovou orientovanost?

- metóda a jazyk
- implementácia a prostredie
- knižnice

2.3 Vysvětlete, co rozumíme pojmy objektově orientovaná metoda (přístup) a jazyk.

Nejde len o programovací jazyk a spôsob jeho použitia, ide aj o spôsob uvažovania a vyjadrovania a taktiež o záznamy v textovej alebo grafickej forme.

- trieda
- trieda ako modul
- trieda ako typ
- zasielanie správ
- skrývanie informácií
- statická kontrola typov
- dedičnosť, redefinícia, polymorfizmus, dynamická väzba
- generickosť
- správa pamäti a garbage collector

2.4 Vysvětlete, co rozumíme podporou objektově orientované implementace

Je to podpora vývoje. Zahřňa vlastnosti a efektivitu nástrojů pre vývoj, nástroje pre podporu nasadenia nových verzí a nástroje pre podporu dokumentování.

2.5 Vysvětlete, co rozumíme podporou opakované použitelnosti.

Znamená to navrhování a vytváření komponent, modulů, kódu alebo systému takým způsobem, aby boli ľahko použitelné v různých kontextech alebo projektech bez nutnosti ich opětovného vytváření alebo prepisování.

Příklady:

- Knižnice a frameworky
- Kódové šablóny
- Cloudové služby a API

2.6 Vysvětlete pojmy třída a objekt a použijte správnou terminologii.

Trieda je část software, která popisuje abstraktný datový typ a jeho implementáciu.

Objekt je abstraktný datový typ so spoločným chovaním reprezentovaným zoznamom operácií, ktoré vie objekt vykonávať.

2.7 Zdůrazněte vlastnosti třídy z pohledu modularity.

Triedy nepopisují len typy objektů, ale musia byť zároveň modulárními jednotkami. V čisto OOP by nemali být jiné samostatné jednotky než triedy.

2.8 Vysvětlete princip zapouzdření v OOP.

Zapůzdření (enkapsulácia) obmedzuje přístup k určitým částiam objektu a poskytuje kontrolu nad tým, ako sú dáta a funkcie v objekte využívané.

Cieľom je ochrana dát a zjednodušenie správy komplexivity. Pre interakciu s dátami objektu a jeho ovládanie sa používajú metódy **Getter** a **Setter**.

2.9 Vysvětlete princip zasílání zpráv.

Tento princip slůží na komunikáciu medzi objektami aplikace. Namísto toho, aby objekty přímo přistupovali k internému stavu jiných objektů alebo přímo volali ich metódy, komunikují medzi sebou prostřednictvím správ.

Správy představují požadavky na vykonanie nejakej akcie, ktorú objekt vykonaná na základe prijatej správy. Tieto správy môžu být volania metód, žádosti o změnu stavu objektu alebo žádost o dáta.

2.10 Vysvětlete principy deklarace a definice jednoduché třídy v C++.

Deklarácia je proces, pri ktorom definujeme štruktúru triedy, teda aké vlastnosti a chovanie trieda bude mať. `class MyClass public: // Specifikuje, že členy jsou veřejné`

```
(přístupné zvenčí) MyClass(); // Konstruktor void display(); // Veřejná metoda private:  
// Specifikuje, že členy jsou soukromé (nepřístupné zvenčí) int x; // Soukromý atribut ;  
Definícia je proces, pri ktorom konkrétne určujeme ako budú metódy triedy implementované.
```

3 Návrh programu I

3.1 Vysvětlete, jak vznikají objekty třídy, pojem konstruktor a principy práce s ním v C++.

Objekt je instanciou třídy. Je reprezentací nějaké entity, která má stav reprezentovaný datami a chování reprezentované metodami.

Vzniká konstruktorem, který inicializuje objekty. Nemá návratovou hodnotu, pokud není deklarovaný, vytvoří se automaticky. Při statické deklaraci se volá automaticky, jinak je nutné použít **new**. Může ich být více, ale musí se lišit počtem nebo typem parametrů.

3.2 Vysvětlete, jak zanikají objekty třídy, pojem destruktorka a principy práce s ním v C++.

Objekty zanikají destruktorem. Ten slouží pro dealokování dynamicky vytvořené paměti. Nemá návratovou hodnotu. Pokud není definovaný, vytvoří se automaticky. Při statické deklaraci se volá automaticky, jinak s použitím **delete**.

3.3 Vysvětlete rozdíl mezi statickou a dynamickou deklarací objektů v C++.

Statická deklarace: objekt je na zásobníku, jeho životnost je automatická a řízená blokem, v kterém byl vytvořený.

Dynamická deklarace: objekt je na halde, jeho životnost je řízená manuálně pomocí **new** a **delete**.

3.4 Jak se dá postupovat, pokud chceme v zadání programu nalézt třídy, jejich metody a datové členy?

- **třídy** - často opakující se podstatné mená
- **metódy** - slovesá
- **dáta** - podstatné mená

3.5 Kdy a proč potřebujeme použít více konstruktorů jedné třídy?

V případě, že potřebujeme více instancí dané třídy s různým počtem nebo typem parametrů.

3.6 Kdy potřebujeme deklarovat a definovat destruktorka?

Keď sú dáta objektu dynamické.

3.7 Co jsou výchozí konstruktory a destruktory a k čemu je potřebujeme?

Východzí konstruktorka je automaticky definovaný kompilátorem, pokud není explicitně deklarovaný. Slouží k vytvoření objektu s východzími hodnotami.

Východzí deštruktor je automaticky definovaný kompilátorom, pokiaľ nie je explicitne deklarovaný. Slúži k zničeniu objektu bez špecifickej činnosti.

3.8 Jaké typy metod obvykle musíme deklarovať a definovať?

- konštruktor a deštruktor
- metódy poskytujúce informácie o stave objektu (getter)
- metódy meniace stav objektu (setter)

3.9 Co jsou objektové kompozice a k čemu jsou dobré?

Objekt môže byť súčasťou iného objektu a stáva sa jeho dátovou položkou.

Vznikajú tak komponované objekty s presne definovanými kompetenciami. Tie však môžu byť realizované prostredníctvom iterácie objektov, z ktorých sú komponované.

4 Objektová dekompozice a třída jako objekt

4.1 Jaký je rozdíl mezi funkční a objektovou dekompozicí programu?

Funkčná dekompozícia sa zameriava na rozdelenie programu do samostatných funkcií alebo procedúr, ktoré vykonávajú konkrétne úlohy.

Objektová kompozícia je charakteristická pre OOP. Program je v nej rozdelený objekty, ktoré sú inštanciami tried.

4.2 Proč preferujeme objektovou dekompozici a jaké jsou hlavní problémy funkční dekompozice?

Objektová dekompozícia nám umožňuje používať lepšie modelovať reálne objekty a ich interakcie. Môžeme používať enkapsuláciu, dedičnosť, polymorfizmus a umožňuje nám lepšiu rozšíriteľnosť.

Problémy funkčnej dekompozície sú zlá rozšíriteľnosť, nedostatok štruktúr pre zložitejšie systémy, obmedzená reprezentácia zložitých entít, horšie testovanie a debugging.

4.3 Za jakých podmínek můžeme považovat třídu za objekt a jak to implementovat v C++?

Trieda sa stáva objektom ak:

- obsahuje dáta, ktoré reprezentujú vlastnosti objektu
- má metódy
- má konštruktor
- môže mať prístupové modifikátory
- môže využívať dedičnosť a polymorfizmus

4.4 Vysvětlete rozdíl mezi členskými položkami třídy a instance a popište jejich dostupnost.

Členské položky triedy sú spojené s triedou ako celkom, môžeme k nim pristupovať bez potreby vytvárať inštanciu triedy, ale pristupujeme k nim prostredníctvom samostatnej triedy. Tieto položky sú zdieľané medzi všetkými inštanciami triedy.

Členské položky inštancie sú spojené s konkrétnymi inštanciami triedy. Každá inšancia má svoju vlastnú kópiu týchto atribútov a metód. K položkám pristupujeme pomocou objektu.

4.5 Jak můžeme v C++ důsledně odlišovat práci s členskými položkami tříd a instancí?

K **členským položkám triedy** pristupujeme pomocou názvu triedy - `ClassName::staticMember`.

K **členským položkám inštancie** pristupujeme prostredníctvom konkrétnej inštancie objektu - `object.member`.

4.6 Potřebuje třída v roli objektu konstruktor resp. destruktory a proč?

Áno, potrebuje. Konštruktor zaistí, že objekt je správne inicializovaný a pripravený k používaniu. Deštruktor zaistuje správne uvoľnenie prostriedkov, aby sa predišlo únikom pamäti. Inak by to mohlo viesť k nesprávnemu chovaniu aplikácie.

5 Úvod do dědičnosti

5.1 Které dva klíčové požadavky řešíme pomocí dědičnosti?

Znovu-použitelnost a rozšířitelnost.

5.2 Jaké návrhové požadavky máme na použití tříd (co s nimi můžeme dělat)?

Kombinovanie s inými triedami, skladanie, rozšírenie o nové chovanie, pozmenenie existujúceho chovania.

5.3 Jaký je rozdíl mezi dědičností a skládáním? Co mají společného?

Skládáním docielime to, že objekt jednej triedy je zložený z objektov inej triedy. Jedná sa o vzťah „MÁ“.

Dedičnosťou docielime to, že nová trieda je rozšírením alebo špeciálnym prípadom existujúcej triedy. Jedná sa o vzťah „JE“.

5.4 V jakých rolích vystupují třídy v dědičnosti? Použijte správnou terminologii.

Predok - potomok, priamy predok - potomok. Rodič - dcéra (syn). Nadradená trieda - pod trieda.

5.5 Vysvětlete v jakém obecném vztahu je třída, ze které se dědí, se třídou, která dědí.

Trieda z ktorej sa dedí je rodičovská trieda. Potomok zdedí všetky metódy rodičovskej triedy. Môže dané chovanie rozšíriť alebo pozmeniť. Nemôže sa ho zbaviť.

5.6 Co všechno se dědí, co ne a proč?

Dedí sa všetko bez výnimky, rovnako aj miera skrývania informácie. Pretože predok definuje spoločné chovanie všetkých svojich potomkov. Potomkovia môžu toto chovanie rozšíriť či pozmeniť. Nemôžu sa ho zbaviť.

5.7 Co rozumíme jednoduchou dědičností a jak s tím souvisí hierarchie tříd v dědičnosti?

Každý potomok má práve jedného priameho predka. Predok môže mať viac priamych potomkov. Hierarchiou je strom.

5.8 Co je Liskové substituční princip a jak se projevuje v dědičnosti?

Potomok môže vždy zastúpiť predka a to preto, že majú spoločné chovanie. Opačne to neplatí.

5.9 V jakém pořadí se volají a vykonávají konstruktory při použití dědičnosti?

1. Volanie konštruktora objektu.
2. Volanie konštruktora priameho predka.
3. Vykonanie konštruktora priameho predka.
4. Vykonanie konštruktora objektu.

6 Dědičnost – změna chování

6.1 Co rozumíme paradoxem specializace a rozšíření?

Vztah dedičnosti je vztahom obecný - speciální. Potomok je teda špeciálnym prípadom predka. Paradoxné je, že pri rozšírení dochádza k tomu, že potomok vie viac ako ktorýkoľvek jeho predok. A teda čím bohatšie chovanie uvažujeme, tým menej tried ho poskytuje.

6.2 Uveďte správné a špatné příklady vztahu "generalizace-specializace".

Zlý příklad - vztah bodu a kružnice. Mohli by sme potrebovať rozšíriť bod o prácu s polomerom, t.j. pridať nové chovanie. Avšak potreba rozšírenia sama o sebe nie je dostatočujúca pre použitie dedičnosti. Nie je splnená potreba špecializácie (kružnica nie je špeciálnym prípadom bodu).

Dobrý příklad - vozidlo je všeobecná kategória reprezentujúca prostriedky na prepravu, takže bicykel môžeme považovať za špeciálny typ vozidla - má konkrétne vlastnosti.

6.3 Co rozumíme v dědičnosti změnou chování?

Pokiaľ je chovanie deklarované v predkovi, môžeme ho v potomkovi deklarovať znovu. Existuje potom viac metód rovnakého mena. Deklarované chovanie potom musíme v potomkovi implementovať.

6.4 Co rozumíme přetížením? Jedná se o rozšíření nebo změnu chování?

Preťaženie je situácia, keď daná metóda má rovnaké meno ale má iné parametre, iné typy parametrov, inú návratovú hodnotu. Preťaženie nie je zmena chovania, napriek tomu, že má rovnaké meno. Ide teda o rozšírenie chovania.

6.5 Uveďte různé typy přetížení.

- meno metódy zostáva rovnaké
- iný počet parametrov
- iné dátové typy parametrov
- iná návratová hodnota (nie v C++)

Tieto typy je možné kombinovať.

6.6 Co rozumíme překrytím? Jedná se o rozšíření nebo změnu chování?

Prekrytie je situácia, kedy metóda potomka má rovnakú deklaráciu ako metóda predka. Potomok dedí aj metódy predka. Má teda dve metódy s rovnakou deklaráciou. Je to zmena chovania. Typickým príkladom použitia sú konštruktory.

6.7 Jaký princip porušujeme, použijeme-li „protected“ a proč?

Porušujeme princip zapúzdrenia. Pri zmene chovania môže vzniknúť potreba pracovať so súkromnou časťou predka.

6.8 Jaký problém přináší potřeba změny chování v dědičnosti?

Prekrytie metód, polymorfizmus, zhoršenie čitateľnosti

6.9 Popište, jak se prakticky projevuje různá míra přístupu k položkám třídy.

- **public** - položka triedy je dostupná odkiaľkoľvek
- **private** - položka triedy je prístupná iba v rámci triedy, v ktorej bola definovaná
- **protected** - položka je prístupná v triedach, ktoré dedia danú triedu, ale nie úplne voľne pre akýkoľvek kód

6.10 Jak se použití „protected“ projeví ve vztahu předka a potomka?

Prejaví sa to tak, že protected položky sú prístupné v podtriedach, aj keď sú inak skryté pred zvyškom programu. Potomok má priamy prístup k protected metódam a atribútom. Môže protected metódy prekryť a zmeniť ich chovanie. Pracuje s položkami akoby to boli jeho vlastné.

7 Dědičnost - polymorfismus

7.1 Jaký je rozdíl mezi shadowing a overriding překrytím? Uveďte příklady.

Shadowing je statické překrytí, kdy nová instancie potomka zatieni metódu predka. Čiastočné chovanie objektu teda odpovedá trieda, v roli ktorej vystupuje.

Overriding je dynamické překrytí, kdy sa vždy použije metóda potomka, pokiaľ ju má implementovanú. Čiastočné chovanie objektu teda odpovedá triede, jej je tento objekt inštanciou.

7.2 Co rozumíme polymorfismem a s čím to souvisí?

Polymorfizmus je schopnosť objektov vystupovať v rôznych roliach a podľa toho sa aj chovať. Objekt kombinuje svoje chovanie s chovaním predka. Súvisí to so substitučným princípom, teda zastupiteľnosťou predka potomkom.

7.3 Co rozumíme polymorfním přiřazením?

Zdroj priradenia je iného typu než cieľ priradenia.

7.4 Co je časná vazba? Uveďte příklady.

Prekladač za normálnych okolností využíva to, že pri volaní metódy vyhodnocuje typ inštanciu už v dobe prekladu. V metóde Withdraw sa zavolá metóda CanWithdraw predka.

7.5 Co je pozdní vazba? Uveďte příklady.

Prekladač vyhodnocuje typ inštanciu až v okamžiku volania. Použijeme virtual.

7.6 Popište, co je virtuální metoda a její vlastnosti.

Používa sa, pokiaľ chceme prenechať rozhodnutie, ktorá prekrytá metóda bude volaná až v priebehu programu (overriding). Dávame prekladaču najavo, že chceme využívať neskorú väzbu. Jedenkrát označená metóda ako virtuálna, zostane virtuálna vo všetkých potomkoch.

7.7 Popište, co je tabulka virtuálních metod a jak funguje.

Akonáhle niektorú metódu definujeme ako virtuálnu, prekladač pridá k triede „neviditeľný pointer“ ktorý ukazuje do VMT. Pre každú triedu, ktorá má aspoň jednu virtuálnu metódu, prekladač vytvorí vlastnú VMT. Tabuľka obsahuje pointery na virtuálne metódy. Je spoločná pre všetky inštancie danej triedy.

7.8 Může být konstruktor virtuální? A proč?

Nemôže. Pred jeho volaním ešte nie je vytvorený odkaz do VMT.

7.9 Môže byť deštruktor virtuálny? A prečo?

Áno. Zabezpečí, že pri vymazávaní objektu sa najprv zavolá deštruktor podtriedy, a až potom deštruktor základnej triedy.

7.10 Kedy mluvíme v C++ o polymorfizmu a jak se to projeví v návrhu?

Je spojená s dedičnosťou. Musia byť použité virtuálne metódy (overriding). Stále ide o zastúpiteľnosť predka potomkom.

7.11 Co je polymorfní datová struktura a k čemu ji využíváme?

Je to štruktúra, ktorá obsahuje objekty rôznych tried (pole, zoznam, ...). Po takto uložených objektoch môžeme volať iba spoločné metódy predka. Pre volanie ostatných metód objektu je nutné pretypovávať.

7.12 Kedy potřebujeme virtuální deštruktor? S čím to souvisí?

Potrebujeme ho, pokiaľ začneme v programe využívať polymorfizmus. Zabezpečí, že pri vymazávaní objektu sa najprv zavolá deštruktor podtriedy, a až potom deštruktor základnej triedy.