**BI-SPOL-21 Objektově orientovaného programování v C++, zapouzdření, dědičnost, atributy a metody, statické atributy a metody, virtuální metody, polymorfismus, abstraktní třídy, výjimky, šablony, přetěžování operátorů, mělká a hluboká kopie**

BI-PA2

**OOP v C++**

* určení entit, analýza jejich interakcí a implementace entit v podobě tříd a jejich metod
* zakládá se na představě reálného světa, kdy existují objekty s různými vlastnostmi
* objekty drží stav (dán atributy)
* interface: definuje chování objektu; množina metod, které objektu náleží
* třída: definuje interface objektu
* objekt: instance třídy

**Třída**

* třída je abstrakce entity reálného světa
* třída je popisem datového typu
  + jméno
  + data – členské proměnné
  + interface – členské funkce

**Objekt**

* instance třídy
* objekty jsou proměnné
  + každý objekt má třídu
  + v průběhu výpočtu jsou objekty vytvářeny a rušeny
  + obvykle vytváříme více objektů stejné třídy
* vytváření objektů
  + objekt je proměnná (instance třídy)
  + př vytvoření objektu je zavolán konstruktor
  + konstruktory mohou být přetíženy. Platí zde pravidla pro přetěžování funkcí
  + Konstruktor nelze volat explicitně na existující objekt
  + Destruktor je volán automaticky, když je objekt rušen
* Objekty alokované dynamicky se vytváří operátorem new – ten volá konstruktor
  + Dynamicky alokované objekty jsou rušeny použitím operátoru delete – volá destruktor (pro dynamicky alokovaných se pro přístup k metodám atd. používá -> (šipka), jinak se používá . (tečka)

**Const**

* Když je objekt označen jako const, jeho položky nemohou být modifikovány
* Const metody – těla const metod nemohou modifikovat položky (kontroluje to kompilátor)
* Nekonstantní metody nelze na konstantní objektu volat

**Instanční metody**

* Mohou být volány na existující objekt
* Mají přístup k instančním proměnným, a to jak přímo v instanci, nebo přes ukazatel this
* Mají přístup k třídním proměnným
* Mohou volat jiné instanční nebo třídní metody

**Statické (třídní) metody**

* Jsou podobné běžným funkcím
* Nemají žádnou implicitní instanci, proto ani implicitní členské proměnné, ani ukazatel this nejsou k dispozici
* Mají přístup k třídním proměnným a mohou volat jiné třídní metody

**Zapouzdření**

* omezení přístupu ke členům třídy
* *public* (dostupné komukoliv)
* *private* (přístup jen ve třídě samé)
* *protected* (přístup ve třídě samé a v jejích podtřídách - dědění)
* prostředek k tomu, aby šlo (cílem je vysokoúrovňové veřejné rozhraní):
  + nezávisle upravovat implementaci uvnitř třídy (např. zvolit efektivnější algoritmus, jinou reprezentaci dat)
  + kompletně nahradit třídu bez rozbití zbytku programu (pokud nová třída dodrží veřejné rozhraní)
  + opravit chyby uvnitř třídy bez rozbití zbytku programu
  + pracovat na vývoji tříd nezávisle

**Dědičnost**

* schopnost tříd deklarovat jinou třídu jako nadřazenou (rodiče) a dědit její chování
* potomci mohou rozšířit chování rodiče o další metody a atributy
* mohou metody rodiče přepsat
* lze vytvářet hierarchie tříd

**Atributy**

* proměnné, které si objekt uchovává pro následnou práci
* každá instance má vlastní instanční proměnné
* deklarované bez klíčového slova
* *statické (static)*: nejsou součástí objektu, ale třídy, tento atribut je sdílený napříč všemi instancemi třídy,

**Metody**

* mohou být volány na existující objekt
* mají přístup k instančním metodám a atributům
* *statické (static)*: podobné funkcím, nejsou součástí konkrétního objektu, jsou přístupné bez jakékoliv instance, mají přístup k třídním proměnným i metodám

**Virtuální metody**

* základ polymorfismu
* není závislá na typu ukazatele na objekt, rozpoznává deklarovanou instanci třídy a volá overridnutou metodu
* musí mít dynamickou vazbu, ta určuje volanou metodu až v době výpočtu, na základě objektů, které jsou zpracovávány
* např. je třída Person s metodu print(), která vypisuje "Jsem člověk". Od ní dědí Man a Woman, přepisují print(), aby vypisovala "Jsem muž" a "Jsem žena".

Pokud deklaruju Person \* m = new Man() a zavolám m->print(), tak se zavolá virtuální metoda třídy Man a vypíše "Jsem muž"

**Polymorfismus**

* musí mít dynamickou vazbu
* schopnost objektů se stejnou specifikací mít rozdílnou imlementaci
* objekty poskytují stejné rozhraní, ale jiné chování na základě vyděděné třídy (viz příklad výš)
* *Overriding*: předefinování již implementovaných parent metod na nové se zachováním rozhraní.
* *Parametrický polymorfismus*: volání metody se stejným názvem na základě poskytovaného počtu a typech parametrů.

**Abstraktní metoda**

* Metoda, která je deklarovaná, avšak její implementace chybí – je nahrazena konstrukcí = 0
* Definují interface třídy. Všechny podtřídy budou implementovat tento interface
* Jsou užitečné když:
  + Je známo, že všechny podtřídy jsou schopny provést implementaci abstraktní metody
  + Implementace v nadtřídě nedává smysl (např. některé důležité položky v nadtřídě nejsou)
  + Je požadován nějaký unifikovaný pohled v podtřídách (např. kolekce instancí v nějakém kontejneru)

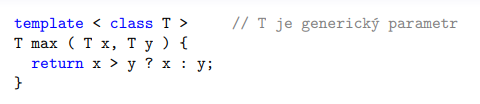
**Abstraktní třídy**

* třída, která obsahuje alespoň jednu abstraktní metodu
* využití při dědičnosti
* nelze takovou třídu deklarovat – nelze vytvořit objekt
* všechny podtřídy implementují tento interface

**Šablony (generické funkce/třídy)**

* parametrizovaná deklarace a definice funkce/třídy
* mohou parametrizovat datové typy svých parametrů nebo návratové hodnoty
* generický parametr (T) je datový typ, který je kompilátorem substituován, když vzniká nová instance generické třídy
* využití: pokud předem neznáme datový typ, případně chceme udělat funkce se stejným chováním pro více datových typů
* např. T max (T a, T b)
* implicitní deklarace max(1,2) → int max (int a, int b)
* explicitní deklarace max<char>(1, 2) → char max (char a, char b)

**Generické funkce (šablona)**

* parametrizovaná deklarace a definice funkce
* generický parametr (parametr šablony) je datový typ, který je kompilátorem substituován, když vzniká nová instance generické funkce
* generické funkce mohou parametrizovat datové typy svých parametrů nebo návratové hodnoty
* 
* Obsah obrázku text

  Popis byl vytvořen automaticky
* Explicitní instance:
* Obsah obrázku text, osoba, snímek obrazovky

  Popis byl vytvořen automaticky
* V posledním případě se definuje i typ návratové hodnoty
* Mohou se přetěžovat

**Generické třídy (šablona)**

* Parametrizovaná implementace třídy. Kompilátor odvozuje instanci generické třídy nahrazením generických parametrů skutečnými hodnotami
* Generické třídy jsou obvykle parametrizovány typovým(i) parametrem(y)
* Obsah obrázku text

  Popis byl vytvořen automaticky
* Na rozdíl od „obyčejných“ tříd jsou generické třídy deklarovány i implementovány v hlavičkových souborech
* Metody jsou obvykle implementovány jako inline metody
* Pokud jsou metody implementovány vně deklarace generické třídy, musí každá metoda začínat deklarací šablony template
* Obsah obrázku text

  Popis byl vytvořen automaticky

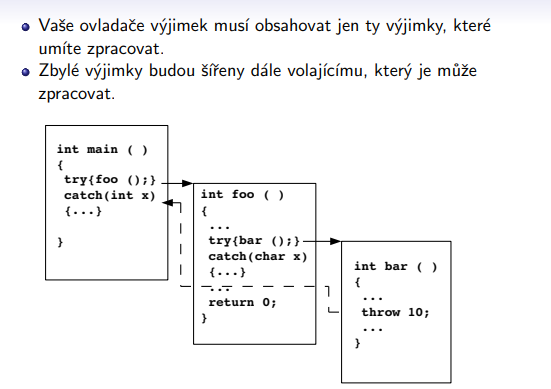
**Výjimky**

* neočekávaná situace, která přeruší příkazy
* **speciální chybový signál procházející řetězem volání od volaného k volajícímu**
* vyhazují se objekty typu Error, ty se musí zachytávat *try-catchem*, pokud se neodchytí, program spadne
* v případě odchycení jsme schopni opravit program, aby pokračoval v běhu
* např. vyhození výjimky dělení 0
* výhody:
  + standardní vlastnost jazyka
  + vnitřní funkce řetězu volání jsou informovány o výjimce a mohou uvolnit své alokované zdroje. Navíc toto je provedeno bez jakéhokoli dalšího programování – uvolnění provádíme v těle *catch { zde }*
  + různé chyby jsou odlišeny použitím různých typů výjimek
  + výjimka může doplnit další popis problému (např. jméno souboru). Navíc, různé výjimky mohou obsahovat různá data (různé druhy chyb mohou obsahovat různé datové struktur s popisem okolností chyb)
* Obsah obrázku text

  Popis byl vytvořen automaticky

Obsah obrázku text

Popis byl vytvořen automaticky



* Ovladač výjimek catch(…) je černá díra – zachytí jakoukoliv výjimku
* Příkaz throw bez parametrů pokračuje v šíření výjimky

**Friend funkce**

* funkce, kterým je udělen přístup k private a protected položkám třídy
* porušují zapouzdření
* užitečné při přetěžování operátorů

**Přetěžování operátorů**

* přetěžovány mohou být jen existující operátory (+, +=, < …)
* musí zůstat zachována arita (+= musí být binární)
* na základě přetěžování se dá definovat nové chování (např. můžu sečíst dvě instance třídy), nelze nahradit původní
* může se využívat *friend* funkcí. Ty jsou deklarovány ve třídě a samotná definice funkce je mimo třídu. Tato funkce má přístup k private a protected atributům a metodám. (např << pro ostream, aby se mohl výstup řetězit)

**Mělká a hluboká kopie**

* Nechť existuje instance objektu A , která má několik atributů staticky alokovaných na stacku (přímo data) a několik dynamicky alokovaných na heapu (ukazatelé).
* Při *mělké kopii* instance A se kopíruje objekt a jeho přímo uložené hodnoty (hodnoty proměnných), tzn. pro dynamicky alokované proměnné se kopíruje pouze ukazatel. Ve výsledku dva objekty dynamicky ukazují na stejnou paměť.
* Při *hluboké kopii* se např. v kopírovacím konstruktoru definuje i způsob kopírování dynamických proměnných. Dochází i ke kopii paměti na heapu, instance jsou pak nezávislé.
* ve zkratce: mělká kopie = kopie ukazatelů, sdílená data, hluboká kopie = kopie všeho, nezávislé

**Kopírující konstruktor vs operátor =**

* oba jsou použity pro vytvoření hluboké kopie (musí napsat programátor)
* operátor =
  + má na levé straně plně funkční objekt. Tudíž zdroje alokované pro cílový objekt musí být před kopírováním uvolněny
* kopírující konstruktor
  + nemá žádný objekt k modifikaci. Místo toho má cíl již rezervovanou (neinicializovanou) paměť, která musí být inicializována
* oba jsou těsně svázány s destruktorem – je-li důvod pro implementaci destruktoru, je také důvod pro implementaci kopírujícího konstruktoru a operátoru = (pokud kopírování není zakázáno)

**Počítání referencí**

* problém s mělkou kopií je ve sdílení zdrojů – můžeme uvolňovat již uvolněnou paměť
* řešením tohoto problému je počítání referencí
  + objekty s počítanými referencemi drží informaci o počtu jiných objektů referencujících stejný zdroj
    - nově vzniklé objekty, sdílející zdroj inkrementují čítač
    - rušené objekty dekrementují
    - rušený objekt, který vynuluje čítač referencí, fyzicky uvolní paměť

otázky:

* Vysvětlit koncept OOP, co je zapouzdřenost + výhody, co je dědičnost + výhody, co je virtuální (abstraktní) metoda a abstraktní třída + použití.