C++

- Viacparadigmový vyšší programovací jazyk, nadstavba jazyka C. Moderná podoba je objektovo orientovaná, generický a s funkcionálnymi zložkami, obsahujúci aj možnosť low-level manipuláciou s pamäťou. Je skoro vždy implementovaný ako kompilovaný jazyk.
- Je orientovaný na systémové programovanie (ktoré sa na rozdiel od aplikačného programovania nepoužíva na tvorbu aplikácii pre užívateľa, no najmä tvorbu podporných programov pre iné softvéry)

Realizácia myšlienky OOP

- Základnou myšlienkou je, že C++ neobsahuje žiadne high-level dátové typy ani high-level primitívne operácie (string concat, matrix inversion). Všetky musia byť užívateľom definované ručne.
- Napriek tomu, užívateľom definované typy sa od built-in typov líšia iba v definícii, nie v spôsobe, akým sú použité
- Zložky, ktoré by spôsobili overhead, či už v pamäti alebo v run-time (ako napr. Housekeeping info) boli z jazyka vynechané, preto napr. Užívateľom definovaný objekt s 2-mi 16-bit premennými sa vojde do 32 bitov.

Trieda, člen triedy

- Trieda je užívateľom definovaný typ. Jej definícia obsahuje členov triedy, teda všetky premenné, konštanty, funkcie, metódy a procedúry, ktoré určujú vlastnosti a vykonávajú sa priamo nad daným objektom - inštanciou danej triedy.
- Trieda v C++ narozdiel od C# či Javy nepodporuje keyword static. Je možné vytvoriť triedu, v ktorá obsahuje výlučne statické členy, no kompilátor ju považuje za klasickú triedu, teda aj táto trieda sa dá inštancovať. V prípade statickej triedy je lepšie použiť namespace.
- Členy môžu byť takisto statické. Tieto sú volané pomocou celej triedy a nie sú závislé na jednotlivých inštanciách triedy.

Konštruktor

- Konštruktor je členská funkcia triedy, ktorá inicializuje objekt danej triedy. V C++ je automaticky zavolaný pri vytvorení novej inštancie.
    - Musí mať rovnaký názov ako trieda
    - Nemá výstupový typ
    - Je automaticky volaný pri tvorbe objektu
    - Pri vynechaní definície kompilátor vygeneruje defaultný konštruktor, kde všetky členské premenné nadobudnú defaultnú hodnotu pre ich dátový typ.
    - Môže byť overloadovaný.

Deštruktor

- Deštruktor je členská funkcia triedy, ktorá ničí a maže objekt.
    - Je volaný automaticky, keď je objekt OOS
        - Keď skončí funkcia, v ktorej je objekt vytvorený
        - Keď skončí program
        - Keď skončí blok, v ktorom je objekt
        - Keď je explicitne zavolaný deštruktor
    - Má názov rovnaký ako názov triedy prefixovaný tilde (~)
    - Nemá argumenty, ani výstupnú hodnotu

- Pri vynechaní definície je vytvorený kompilátorom automaticky.
- Musí byť napísaný explicitne, ak v objekte vystupujú napr. Nosné raw pointery, ktorých zmazaním tak zostane v pamäti neprístupný iniciovaný objekt.
- Môže (mal by) byť virtuálny v base class, ak obsahuje virtuálne členy.
- Nemôže byť overloadovaný (nemá ani prečo).

ochrana přístupu ke členům tříd

- Pri definícii triedy v *.h súbore je každý člen uložený pod blok public, private či protected, čím sa mu určí prístupnosť zvonka, pričom defaultný typ bloku je private.
  - Private: členy prístupné iba ostatným členom v tejto triede a vo friend triedach.
  - Protected: členy prístupné iba v danej triede, friend triedach a triedach, ktoré dedia danú triedu.
  - Public: členy voľne prístupné vrámci celého programu.
  - Friend: ak je v triede A zadeklarované "friend B", kde B je iná trieda, trieda B môže voľne pristupovať ku všetkým členom triedy A.
    - Tento vzťah nie je obojstranný (v triede B by musela byť trieda A deklarovaná ako friend explicitne).
    - Malo by byť používané zriedka, inak stráca zmysel encapsulation jednotlivých tried a celkové OOP.
    - Nie je dedené.
- Struct je narozdiel od niektorých iných jazykov identický s triedou s rozdielom, že defaultne sú členy public.

Viacnásobná dedičnosť

- Nastáva, keď jedna trieda dedí z viac ako jednej rodičovskej triedy
- Konštruktory sú volané v poradí, v akom daná trieda dedí rodičovské triedy (najprv rodičovské v poradí, potom samotný konštruktor dediacej triedy)
  - Class C : public B, public A
  - Zavolajú sa konštruktory v poradí B,A,C.
- **Diamond problem**
  - Nastáva, keď trieda dedí z viacerých nadtried, ktoré majú spoločnú base class
    - Class D : public C, public B
    - Class C : public A
    - Class B : public A
  - V takomto prípade je konštruktor base class (A) volaný 2-krát, rovnako ako deštruktor base class.
  - To takisto znamená, že daná trieda (D) má 2 kópie všetkých členov base class, čo vytvára nejasnosti.
  - Riešením takéhoto problému je keyword virtual, ktoré zaručí, že členovia nadradených tried sa vyskytnú iba raz.
    - Class D : public C, public B
    - Class C : public virtual A
    - Class B : public virtual A
  - ~~PS: buď som blbý alebo virtual inheritance nie je dobré na nič iné. Proste to rieši tento problém a basta, hlbší zmysel to nemá. Alebo som slabo hľadal.~~
  - ~~o~~ Virtuálna dedičnosť nie je nič viac ako to, že namiesto toho, že si dediaca trieda drží kópiu rodiča, drží si pointer na ňu. Potom keď D dedí C a B, dostane 2-krát pointer namiesto 2 kópii, čo si už vie ľahko porovnať.

- ~~-~~ Takisto, ak si dobre pamätám, napriek tomu, čo je napísané na wiki, quora, SO apod. nejde pri tomto ani tak o nezrovnalosti v metódach (B aj C overloaduje metódu Foo, D nie. Ktorú verziu potom dedí D.Foo()?), ako skôr v dátach. Disambiguation... výber konkrétnej verzie ľahko možno spraviť castom (static_cast<C>(d).Foo()).

Abstraktné triedy

- Používajú sa na vyjadrenie všeobecného konceptu a slúžia čisto ako základ dediacich tried
- Nemôže vytvoriť objekt
- Nestatické členy nesmú byť deklarované.
- Nemožno použiť ako typ parametra, návratový typ či ako cieľový typ konverzie.
- Môžu však byť použité pointery na abstraktnú triedu.
- Pre čisto virtuálnu členskú funkciu môže (a v prípade deštruktora musí) byť definovaná

Výnimky

- Základnou myšlienkou výnimiek je, že funkcia, ktorá nájde problém, s ktorým nevie pracovať, vráti (throw) výnimku dúfajúc, že nadradená funkcia ju dostante (catch) a bude vedieť, ako s ňou pracovať.
- Pri exception handling sa využíva tzv. Stack unwinding
    - Proces, ktorý odstraňuje z call stacku vstupné body funkcii.
    - Keď príde k výnimke, lineárne sa prehľadá call stack, nájde sa exception handler, a všetky vstupné body po ňom sú odstránené z call stacku.
    - Exception handler je na stack uložený pri otvorení try bloku, preto napr. Bez try v celom programe ľubovoľná výnimka ukončí celý program, keďže je zmazaný celý call stack.
- V C++ pri throw dostane handler kópiu výnimky.
- V takomto prípade môže funkcia spracúvajúca exception
    - Ukončiť chod programu
        - Vo väčšine prípadov najhoršie riešenie problému, keďže mnoho programov si nemôže dovoliť crashnúť. Moduly či knižnice s takouto funkciou sa teda stávajú veľakrát nepoužiteľné a nebezpečné.
    - Vrátiť hodnotu, ktorá reprezentuje error (nullptr, -1...)
        - Sú prípady, keď neexistuje hodnota, ktorá reprezentuje error. Navyše, ak takáto hodnota existuje, musí byť explicitne kontrolovaná, čo môže ľahko značne (zbytočne) zväčšiť program.
    - Vrátiť hodnotu, a ponechať program v chybnom stave
        - Môže sa stať, že si program nevšimne, že je v chybnom stave. Napr. v C mnoho funkcii v knižniciach nastavuje globálnu premennú “errno”, no mnoho ďalších funkcii túto hodnotu nekontroluje. Navyše, toto riešenie nefunguje v prípade concurrency (async, multithread...)
    - Zavolať funkciu, ktorá sa má o exception postarať
        - Keďže exception handling nemá výnimky priamo riešiť, táto možnosť je najlepšie riešenie, avšak mnohokrát má volaná funkcia možnosť riešiť výnimku iba predošlými tromi možnosťami.
- Jeden z prístupov je takisto pokračovať s istou hodnotu a dúfať, že to program prežije. V takom prípade sa často používa výpis errorov a logging, prípadne rôzne prompty na užívateľa, aby problém vyriešil sám, čo je ale poväčšinou zbytočné, ak sa problém nachádza v hĺbke programu a/alebo je užívateľ mimo z problému.
- Užívateľ môže definovať vlastné výnimky ako triedy, ktoré sa následne môžu dediť...

- Try-catch blok je riešený sekvenčne. Try sa vykoná úspešne alebo skončí pri prvej výnimke, ktorú následne porovná so všetkými catch postupne.
    - Class MathErr {}
    - Class ZeroDivErr : MathErr {}
    - Try { weirdly_suspicious_math_function(); }
    - Catch(ZeroDivErr) { cout >> “Division by zero ya troglodyte.”; }
    - Catch(Matherr) { cout >> “Some else math fuck-up, no zero division tho.”; }
    - Catch(...) { cout >> “Honestly anything could’ve gone wrong here, no math tho.”; }
- Catch sa teda vykoná, ak
    - Typ výnimky je zhodný s tou v argumente
    - Typ výnimky je potomok tej v argumente
    - Výnimkou je pointer, ako aj argument, a oba ukazuju na rovnaký typ výnimky
    - Argument je referencia výnimky a typy sa znova zhodujú
- Možno opakovane hádzať výnimku vyššie
    - Try {}
    - Catch(...) { throw; }
- Catch(...) zachytí akúkoľvek výnimku.
- Ak je zavolaný throw bez argumentu tam, kde nie je akú výnimku hodiť, zavolá sa terminate(), čo je handler, ktorý zavolá funkciu, ktorá je mu nastavená (defaultne abort(), ktorý ukončí aktuálny proces).
- Jedným z problémov, ktoré je dôležité riešiť exception handlingom je tvorba objektov “nebezpečných” tried.
    - Deštruktor je zavolaný na objekt iba v prípade, ak sa jeho konštruktor úspešne dokončil.

```
Class A{
        Int* p;
        Void init();
Public:
        A(int s) { p = new int[s]; init(); } // alokuje sa int, neošetrená výnimka na init()
        ~A() { delete p; }
}
```

    - Konštruktor nie je dokončený, deštruktor teda nie je zavolaný, vzniká memory leak.

Šablony

- “Nezávislé koncepty by mali byť reprezentované nezávisle a kombinované iba, ak je to nutné. Tam, kde je tento princíp porušený, možno buď “zabaliť” nesúvisiace koncepty dohdromady alebo vytvoriť zbytočné závislosti. Tak či onak, takto vzniká menej flexibilný set komponentov, z ktorých sa systém skladá. Šablony tak vytvárajú jednoduchý spôsob, ako reprezentovať širokú škálu všeobecných konceptov spolu so spôsobmi, ako ich navzájom kombinovať. Takto písané triedy a funkcie môžu následne konkurovať ručne písanému, špecifickejšiemu kódu ako v run-time, tak v pamäťovej efektivite.”
- Základný stavebný kameň genericity C++.
- Povoľuje použiť dátové typy ako parametre.

-

Preťažovanie (overloading) operátorov a metód

C vs. C++

- Keďže jazyk C++ bol postavený ako rozšírenie C, C++ vie skoro vždy rozbehať kód písaný v C, naopak nie.
- C++ podporuje OOP, kdežto C je čisto procedurálny jazyk.
- Aj vďaka náture OOP, C++ je schopné "ukryť" premenné do tried a povoliť k nim prístup len pomocou triedových funkcii. Premenné v C sú bez tejto možnosti vždy verejné, a teda vždy napadnuteľné iným softvérom.
- Navyše okrem faktu, že C++ podporuje iné ako primitívne dátové typy (ako C), má navyše PDT string a boolean (čo C nemá).
- C++ podporuje overloading (rovnaká funkcia pre iný vstup, eg. Add(float, float), Add(int, int)), C nic takového nemá.
- C++ podporuje default argumenty (pow(int num, int pow = 2)), C vie trt.
- C++ pracuje na viacerých úrovniach, C čisto na low-level
- Oba vyžadujú manuálne pracovanie s pamäťou
- Oba sú light-weight, i keď C je viac, oba sú kompilované
- Vďaka ich nízkoúrovňovému fungovaniu majú oba vysoký performance
- Pracujú na ľubovoľnej platforme
- C je najvhodnejšie na systémové programovanie a embedované zariadenia. C++ je vhodné na nízkoúrovňové ako aj vyššie softvéry, ako serverové aplikácie, networking, gaming, a drivery.
- V prípade programovania softvéru, kde záleží na každom bite, je vhodnejší C pre značne menší overhead (chipy)
- C, ako procedurálny jazyk, využíva top-down prístup. Naopak, C++ ako každý OOP jazyk využíva prevažne bottom-up prístup, kde sa začína triedami a modulmi, ktoré sú nakoniec spojené v main()
- Enum je v C definovateľný, no je typu integer, čo môže viesť k problémom. Navyše C neobsahuje typové zabezpečenie (my_enum.five == 5). V C++ je enum samostatný typ, teda na prácu s enumom ako s integerom je nutné použiť explicitnú konverziu.
- C neponúka žiadne exception handling. Ten musí byť spracovaný v dodatočných funkciách, narozdiel od exception v C++, ktorý stačí uložiť do try-catch bloku.
- Oba jazyky majú istý preprocesing. C využíva macro (#define obsahkruhu(r) = (PI*r*r)), C++ používa inline funkcie, ktoré sú v preprocesingu expandované priamo v kóde.
- C++ má namespace, C má trt.
- C++ má operator overloading, C má taktiež trt.
- Oba majú pointery, no C++ má referencie.
- Podobne ako v Pascale, v C treba deklarovať premenné vopred, na začiatku funkcie.

Vysvetlené pojmy:

- Koprogramy
    - narozdiel od metód, funkcii a procedúr môžu mať viacero vstupných bodov, byť pozastavené a neskôr obnovené v rôznych miestach. Uplatnenie majú najmä v multitaskingu, prúdoch (streams) a trubkách (pipes))
    - Je to akási obecnejšia verzia podprogramu. Podprogramy sú ukladané v stacku, kdežto životný cyklus koprogramu závisí od potreby a použitia.

    - Podprogram má jeden vstupný bod a jeden výstupný bod (return).
    - Koprogram môže vracať viackrát (yield). Yield vráti kontrolu volajúcemu koprogramu, podobne ako podprogramy, avšak pri opätovnom volaní koprogramu už nezačína od začiatku, ale pokračuje v behu (príkazom za yield, ktorým naposledy skončil).
- Discrete-event simulation (DES)
    - Základnou myšlienkou je, že zmeny v systéme sú iniciované eventami, ktoré sú spracovávané postupne a predpokladá sa, že medzi týmito spracovaniami nedošlo k žiadnej zmene v systéme.
    - Jeho alternatívou je napr. Fixed-increment time progression, kde je čas rozdelený na malé intervaly, v ktorých sa stav systému updatuje na základe eventov, ktoré spadajú do daného časového úseku.
- Viacparadigmový jazyk
    - Programovacia paradigma je základný programovací štýl, ktorý sa od ostatných líši v pojmoch a abstrakciách, ktoré tvoria jednotlivé prvky programu (objekty, funkcie, premenné, obmedzenia, atď.), a krokoch, z ktorých sa výpočet skladá
    - Viacparadigmový jazyk (ako napr. C++) je jazyk, ktorý zároveň podporuje viacero paradigiem (v prípade C++ ako procedurálne, tak aj objektové, prípadne ich kombináciou).
    - Iné paradigmy sú napr. Funkcionálne programovanie, procesne orientované programovanie (multithreading), štrukturálne programovanie (cascading, žiadne goto)

Blbosti, ktoré som si vygooglil (nie som blbý, len trošku ignorant):

- Argument vs. Parameter
    - Parametre sú použité v deklarácii a definícii funkcie/metódy/procedúry
        - void do_fun_stuff(int par1, float par2);
    - Argumenty sú použité pri volaní danej funkcie/metódy/procedúry
        - int arg1 = 2;
        - float arg2 = 5.5f;
        - do_fun_stuff(arg1, arg2);
- Metóda, Funkcia, Procedúra
    - Procedúra nemá návratovú hodnotu (void)
    - Funkcia vracia návratovú hodnotu
        - Pure Function je funkcia, ktorá pre rovnaký vstup vráti rovnaký výstup (nemá prístup ku globálnym premenný, randomu...)
    - Metóda je funkcia či procedúra, ktorá je výhradne v OOP, v konkrétnej triede.