Objektové programování

- přináší nové možnosti a styl programování
- vytváří nový datový typ, který „umí“ vše co standardní datové typy + to co ho naučíme
- překladač se k tomuto typu chová stejně jako k typům standardním
- vystavěn na složeném datovém typu (struct)
- spojení dat a funkcí/metod pro práci s nimi
- ochrana dat (přístupová práva)
- výsledný mechanizmus (spojení dat, metod a práv) nazýváme zapouzdřením
- zlepšuje "kulturu" programování – automatická inicializace (konstruktor), ukončení života proměnné (destruktor), chráněná data …

- nejblíže k ní má knihovní celek z jazyka C – rozhraní, data, kód
- výhody: tvorba knihoven, sdílení kódu, údržba programu

Třída - základ objektového programování

- možnost vytvořit nový složený typ (odvozena od struct)
- třída class respektuje nové přístupy, ale zůstává i struct (zpětně kompatibilní vlastnosti)

- třída je obdobně jako struktura dána **popisem** vlastností a velikosti nového typu (v hlavičkovém souboru).
- konkrétní realizace funkcí/metod třídy je ve zdrojovém souboru

- prvky/proměnné třídy se vytváří až při definici objektu/proměnné
- definujeme-li proměnnou daného typu, je jí rezervováno místo v paměti. U třídy nehovoříme o proměnné ale spíše o objektu, nebo o instanci
- zavádí se mechanizmy automatické inicializace proměnných při vzniku (konstruktory)
- velikost objektu (třídy, struktury) může být větší než součet velikostí jejích proměnných (přidány skryté složky, aplikováno paměťové zarovnání složek)

- funkce přístupné/nabídnuté k použití uživateli tvoří rozhraní třídy (přes které se s ní komunikuje)

- standardní typy mají i operátory. Jelikož se objekt chová jako nový typ podobný standardnímu, je možné pro něj nadefinovat i operátory (mající podobné chování jako u standardního typu)

Objektové programování – znovupoužití kódu

- C++ umožňuje znovupoužití kódu, tvorbu společných rozhraní (šablony, dědění, polymorfizmus…)

šablony (generické programování, templates)
– naprosto stejný kód pro různé typy,
- napsáno pouze jednou
- lze i pro neobjektové funkce

dědění (specializace, inheritance)
- odvození třídy z již existující třídy
- nová třída má vše co původní + drobné změny a rozšíření = specializace předka. Postupujeme od obecného ke konkrétnímu

polymorfizmus (mnohotvarost)
- dědění, které využívá tzv. virtuálních metod
- tvoření skupin tříd, které mají společné rozhraní – jde k nim přistupovat stejně, i když jsou to různé třídy
- při volání metody se vyhledá metoda pro typ aktuálního prvku – lze tedy dát do společné skupiny prvky různých tříd (se společnou bází – rozhraním)
- ekvivalentní předávání zpráv objektům – každý objekt umí přijmout zprávu

Rozbor úlohy pro objektové programování

- podobný jako u „normálního“ programování
- stanovení logických celků a jejich vazeb. Definice entit majících v úloze svoje role (například: zákazník (nakupuje, platí, objednává…), prodavač (vystavuje zboží, přijímá objednávku, ověřuje platbu, vydává zboží…)
- stanovení objektů a jejich rozhraní

- formulace (definice) problému – slovní popis
- vznik (inicializace) a zánik (zrušení) objektu
- rozbor problému – vstupní a výstupní data, operace s daty
- návrh dat (vnitřní datové struktury)
- návrh metod (vstupy, výstupy, "výpočty"/operace, vzájemné volání, rozhraní, předávaná data)
- testování (modelové případy, hraniční případy, vadné stavy …)

Rozbor problému – koncepce programu

- konzultace možných řešení, koncepce
- možnost znovupoužití kódu – stávající řešení nebo nové řešení; šablona, dědění (vztah „je“), prvek jiné třídy (vztah „má“)
- rozhodneme, zda je možné použít stávající třídu, zda je možné upravit stávající třídu (dědění), zda vytvoříme více tříd (buď výsledná třída bude obsahovat jinou třídu jako členská data, nebo vytvoříme hierarchii – připravíme základ, ze kterého se bude dědit – všichni potomci budou mít shodné vlastnosti). (Objekt je prvkem a objekt dědí z …) – relace má (jako prvek) a je (potomkem-typem)
- pohled uživatele (interface), pohled programátora (implementace)
- použití výjimek

Formulace problému – konkrétnější specifikace prvků programu

- co má třída dělat – obecně
- určení požadované přesnosti pro vnitřní data
- jak vzniká (->konstruktory)
- jak zaniká (->destruktor)
- jak nastavujeme a vyčítáme hodnoty (gettery a settery – data jsou soukromá/nedosažitelná pro uživatele)
- jak pracujeme s hodnotami (->metody a operátory)
- vstup a výstup

Návrh datové struktury

- zvolí se data (proměnné a jejich typ), které bude obsahovat, může to být i jiná třída
- během dalšího návrhu nebo až při další práci se může ukázat jako nevyhovující
- data jsou (většinou) skrytá

Navrhněte datovou strukturu (členské proměnné/data) pro třídu komplexních čísel.

Pro třídu komplexních čísel se nabízí dvě realizace dat:

- Reálná a imaginární složka
- Amplituda a fáze (délka a úhel, …)

První verze je výhodná pro operace jako sčítání, druhá pro násobení.
Budeme více násobit nebo sčítat? Obecně nelze říci -> reprezentace jsou rovnocenné.

Dále ve třídě/objektu můžeme mít datový člen pro signalizaci chybového stavu – minulý výpočet se nezdařil (dělení nulou …)

Návrh metod

- metoda – funkce ve třídě pro práci s daty třídy
- metody vzniku a zániku = konstruktor a destruktor
- metody pro vstup a čtení dat = gettery a settery
- metody pro práci s objektem
- operátory
- vstupy a výstupy
- metody vzniklé implicitně (ošetřit dynamická data)
- vnitřní realizace (implementace) metod - zde se (hlavně) zužitkuje C – algoritmy
- to jak je třída napsaná (jak vypadá ona a metody uvnitř) nazýváme implementací

Navrhněte metodu/funkci, která vrátí reálnou část komplexního čísla (pro obě reprezentace dat)

```
double Real()
{
  return iReal;
}

double Real()
{
  return iAmpl * cos( iUhel );
}
```

V případě, že uživatel nepracuje přímo s datovými členy třídy (iReal, iAmpl, iUhel), potom při změně (implementace/realizace) vnitřních parametrů uživatel rozdíl nepozná, protože s třídou komunikuje přes metody/funkce, které jsou veřejně přístupné a které tvoří rozhraní/interface mezi daty třídy a uživatelem.
Dojde ke změně časové a výpočetní náročnosti (zde související i s použitím knihovní funkce cos).

Testování

- na správnost funkce
- kombinace volání
- práce s pamětí (dynamická data)
- vznik a zánik objektů (počet vzniků = počet zániků)
- testovací soubory pro automatické kontroly při změnách kódu

Příklad:
Napište testovací soubor tester.bat pro testování programu progzk.exe tak, že mu předložíte vstupní soubor a jeho výstup porovnáte s výstupem očekávaným. V případě chyby vytiskněte hlášení na konzolu

Část testovacího souboru (**tester.bat** - Windows) pro jedny vstupní parametry

```
progzk.exe  <input1.dat   >output1.dat
fc output1.dat vzor1.dat
if ERRORLEVEL == 1 echo "Program s input1 vratil chybu"
```

nebo pro UNIX/LINUX vložte následující obsah do souboru: **tester.sh**

```
#!/bin/sh
progzk.exe <input1.dat >output1.dat
diff output1.dat vzor1.dat
if [ $? -ne 0 ] ; then
 echo "Program s input1 vratil chybu."
fi
```

Nastavte soubor jako spustitelný…

```
chmod u+x tester.sh
```

A spusťte soubor **tester.sh** z lokálního adresáře

```
./tester.sh
```

Test Driven Development

- unit testing – (postupný) cyklus psaní kódu, ladění, testování správnosti
- testování je součástí vývoje: nadefinuji činnost, napíši test, tvořím kód. Napomáhá k tomu, že v kódu nejsou zbytečnosti - jen to co se testuje, testuje se to co je v definici.
- red (chyba v testu)/green (test v pořádku)/refaktor (vylepši kvalitu) – (bez kódu v testované funkci) test nefunguje/napiš to aby to nějak fungovalo/ předělej to aby to fungovalo lépe (rychleji, přesněji, okomentuj, pojmenuj vhodně proměnné, ...)
- napomáhá vývoji rozhraní - promýšlím parametry již při psaní testu=volání
- kód se přidává, jen když neprojde test. Pokud není kód, mělo by vrátit chybu.
- Preferovat malé kroky přidávání kódu (testů)
- neduplikovat kód (pokud něco píši podruhé, tak to buď dělám zbytečně/navíc,nebo to dám do funkce - jinak musím řešit případné chyby vícekrát)
- neduplikovat testy; testy začnou jednoduchými příklady a jdou ke složitějším konstrukcím

https://mva.microsoft.com/en-us/training-courses/testdriven-development-16458?l=iCWHq6vXC_2706218965

Základní pojmy Objektového programování (shrnutí):

Třída (Class) - Nový datový celek (datová abstrakce) obsahující: data (složky / dříve atributy) a operace (metody), přístupová práva

Instance (Instance) - realizace (výskyt / exemplář) proměnné daného typu.
Objekt (Object) - instance třídy (proměnná typu třída).

(Členská) data (Member data / member variable) - data / proměnné definované uvnitř třídy. Často se používá i pojem složka. Dříve atribut (dnes má jiný význam).
Metoda (Member function / method) - funkce definovaná uvnitř třídy. Má vždy přístup k datům třídy a je určena pro práci s nimi.

Implementace (Implementation) - Těla funkcí a metod (tj. kód definovaný uvnitř funkce / metody).

Zapouzdření (Encapsulation) – shrnutí logicky souvisejících (součástí programu) do jednoho celku (zde data, metody, přístupová práva) – nového datového typu třída. Ve volnější pojetí lze používat i pro funkce a proměnné definované v rámci jednoho .c souboru. Někdy je tímto termínem označováno skrytí přímého přístupu k datům a některým metodám třídy (private members). Volně dostupné vlastnosti se označují jako veřejné (public members).

Rozhraní (Interface) - Seznam metod, které jsou ve třídě definovány jako veřejné a tvoří tak rozhraní mezi vnitřkem a vnějškem třídy.

Životní cyklus objektu (Object live cycle) **-** Objekt jako každá proměnná má definováno místo vzniku (definice/inicializace) a místo zániku.

Konstruktor / Destruktor (Constructor / Destructor / c'tor / d'tor) - metody které jsou v životě objektu volány jako první resp. jako poslední. Slouží k inicializaci datových členů objektu resp. k jejich de inicializaci (navrácení alokovaných zdrojů – paměť, soubory…).

Operátor (Operator) - Metoda umožňující zkrácený zápis svého volání pomocí existujících symbolů. (součet +, podíl /, apod.)

Dědičnost (Inheritance) - Znovupoužití kódu jedné třídy (předka) k vytvoření kódu třídy nové (potomka). Nová třída (potomek) získá všechny vlastnosti (data, metody) z potomka a může definovat libovolnou novou vlastnost.

Mnohotvarost (polymorphism) - Třídy se stejným rozhraním, a různou implementací, jednotný přístup k instancím. Mechanismus umožňující správné volání metod potomka z metod předka.

Pojem třídy a struktury v C++ (o – Objektová vlastnost)

- struct a class jsou složené datové typy – vychází ze struct jazyka C
- pomocí struct a class se definuje nový datový typ (má „stejné“ vlastnosti a možnosti co do použití jako standardní typy – int, double …
- struct v C++ je rozšířena o (objektové vlastnosti) možnost přidat metody a přístupová práva (a dále dědit do dalších potomků)
- struct a class v C++ se liší pouze minimálně (leč v důležité věci)
- nebude-li dále řečeno jinak, platí uvedené pro struct i class
- třída tvoří jmenný prostor

- v jazyce C struct obsahuje pouze data
- v jazyce C++ obsahuje data/členy, metody (funkce pracující s třídou)
- v jazyce C++ lze nastavit přístupová práva pro data a metody (lze je používat pouze „uvnitř“ třídy, nebo je může používat i „uživatel“ třídy) – tzv. vnitřní a vnější rozhraní
- v C++ platí, že „data jsou to nejcennější, co máme“ a proto jsou v class data implicitně „schována“ – private
- struktura kvůli zpětné kompatibilitě s jazykem C musí implicitně umožňovat uživatelský přístup k datům - public