Objektově orientované programování

Úvod

Imperativní programovací styl

- klasický programovací styl používaný v době vzniku prvních vyšších programovacích jazyků
- těžiště programování je v *tvorbě algoritmů* nad *(relativně) jednoduchými daty* (pole, matice, texty, struktury, ...), data hrají podřadnou roli, odpovídají matematické formulaci (abstrakci) úlohy

- při tvorbě programů se často používá metoda dekompozice - rozklad problému na jednodušší činnosti (funkce) - návrh *shora - dolů*
- abstraktním popisem dílčích činností jsou procedury a funkce (***strukturované programování***)
- velmi vhodné pro řešení matematických úloh (k jejichž řešení byly počítače určeny především)

- při rozšiřování aplikační oblasti nasazení počítačů (modelování, simulace, databáze) přestávaly *klasické přístupy a zejména datové struktury* vyhovovat potřebám programování - neumožňovaly vhodně popsat objekty reálného světa a zachytit vztahy mezi nimi (generalizace, specializace)

Proto vznikl objektově orientovaný programovací styl ...

Struktura pro reprezentaci komplexních čísel - neobjektově

- soubor komplex.h

```
typedef struct
{
  float re,im; //reálná a imag. část
} TKomplex

// funkce pro výpočet velikosti
float velikost(TKomplex &kc);
```

Struktura pro reprezentaci komplexních čísel - neobjektově

- soubor komplex.c

```
float velikost(TKomplex &kc)
{
  return sqrt(kc.re*kc.re+kc.im*kc.im);
}
```

Struktura pro reprezentaci komplexních čísel - neobjektově

```
#include "komplex.h"
void main()
{
  TKomplex c1,c2;
  c1.re = 4; c1.im = 3;
  c2.re = 0; c1.im = 0;
  cout << "Realna cast c1 je " << c1.re << endl;
  cout << “Imag.  cast c1 je " << c1.im << endl;

  cout "Velikost c2 je “ << velikost(c2) << endl;
}
```

Objektově orientovaný programovací styl

Jedna z definic:

Objektově orientovaný programovací styl lze označit jako obecný postup analýzy, návrhu a implementace programu, založený na přímém modelování (programovém popisu) objektů z reálného světa aplikace (včetně jejich vazeb a interakce) ve světě počítače s využitím prostředků pro abstrakci a hierarchizaci popisu.

- modelem objektu z reálného světa je v aplikaci (programu) je datová struktura zvaná objekt
- objekt je charakterizován:
  - *informacemi o objektu* - jsou implementovány jako datové struktury (představme si proměnné) - v terminologii OOP se nazývají **atributy**
  - *typovými operacemi* prováděnými nad atributy (procedurami a funkcemi); nazývají se **metody**
  - někdy je implementována vlastní činnost (aktivita, „život“) objektu - thready v JAVě

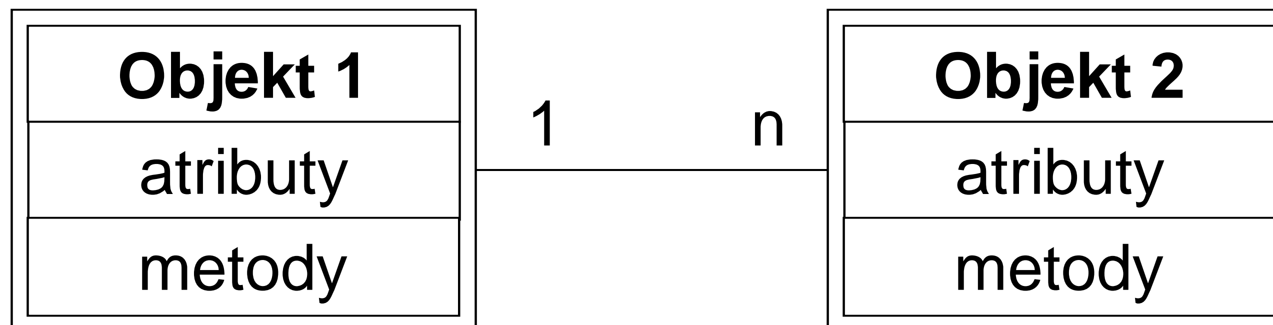
- atributy a metody jsou úzce svázány (syntakticky i sémanticky); objekt je zapouzdřením (encapsulation) atributů a metod
- objekty mezi sebou komunikují vzájemným voláním metod (říkáme též, že si předávají zprávy)

Objektově orientovaná analýza

- zjednodušeně: spočívá v hledání objektů v reálném světě a vazeb mezi nimi (viz systémová analýza)
- objekt: dvojice (**D,F**): **D** - atributy, **F** - metody
  - snažíme se identifikovat atributy, popíšeme je svým jménem a typem
  - metody popisujeme jménem, chování popisujeme v prvé fázi zpravidla slovně, později třeba konečným automatem

- měli bychom pamatovat i na výjimečné a chybové stavy

- výstupem objektově orientované analýzy je systémový popis pomocí ustálených diagramů (např. Codadova notace - viz ukázka, **UML**) nebo již definice objektů zapsaná v progr. jazyce



- více např. Richta: Softwarové inženýrství

Při objektově orientované analýze (a návrhu) jsou v popředí data, nikoliv algoritmy - ty navrhujeme až v poslední fázi

„Neptej se nejdříve, CO má program dělat, ale s ČÍM to má dělat!“

- pro usnadnění návrhu poskytují jazyky podporující OOP prostředky:
  - *dědičnost* - objekt (potomek) přebírá vlastnosti jiného objektu (od rodiče)
    - princip specializace
    - další vlastnosti potomka je možno dodefinovat nebo předefinovat
  - *polymorfismus* - „vícetvarost“ - stejná syntaktická podoba pro různé prvky - zajištěna mechanismem přetěžování funkcí (procedur) a operátorů
  - *genericita*

OOP v C++

- v C++ se zavádí nový datový typ pro popis objektů - třída (class)
- třídy deklarujeme zpravidla v hlavičkovém souboru `.h`, vlastní implementace je oddělená v souboru `.cpp`
  - není třeba deklarovat nový typ pomocí **`typedef`**, protože jména struktur, tříd,… v C++ patří do stejného prostoru jmen jako jiné proměnné

- deklarace třídy v hlavičkovém souboru:

```
class Jméno_třídy
{
   seznam atributů;
   seznam metod;
};
```

- abychom se vyhnuli potížím s vícenásobnými definicemi při vkládání hlavičkového souboru (některé překladače by mohly mít problémy), uzavíráme deklaraci do `#ifndef ... #endif`

Soubor knihovna1.h

```
class A
{ ... };
```

Soubor knihovna2.h

```
#include "knihovna1.h"
class B
{ ... };
```

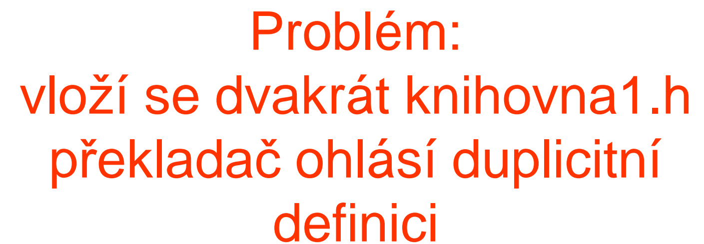
Soubor knihovna3.h

```
#include "knihovna1.h"
class C
{ ... };
```

Hlavní program

```
#include "knihovna2.h"
#include "knihovna3.h"


int main()
{
   ...
};
```



Soubor knihovna1.h správně:

```
#ifndef KNIHOVNA1H
#define KNIHOVNA1H
class A
{ ... };
#endif
```

- většina programátorských prostředí při vytvoření nového souboru typu .h automaticky vloží do textu `#ifndef` …

Příklad:

Navrhneme třídu pro reprezentaci komplexních čísel tentokrát objektově.

- deklaraci třídy zapíšeme do souboru `komplex.h`

```
#ifndef KOMPLEXH
#define KOMPLEXH
class TKomplex
{
 private:
  float re,im; //reálná a imag. část
 public:
  void nastav(float real, float imag);
  float vrat_real();
  float vrat_imag();
  float velikost();
};
#endif
```

atributy

metody

Řízení přístupu (viditelnosti) k atributům a metodám třídy:

`private`

- soukromý, s atributy (metodami) nelze manipulovat mimo členské funkce (metodám a funkcím ve třídě se také říká *členské funkce*)

`public`

- veřejný, manipulace je možná kdekoliv

`protected`

- podobné jako private, používá se při dědění; s atributy (metodami) je možné manipulovat pouze ve členských funkcích a v potomcích (což nejde u private)

- není-li přístup vymezen klíčovým slovem, je automaticky (u atributů i metod) **`private:`**

```
class TKomplex
{
  float re,im; //jsou private
 public:
  void nastav(float real, float imag);
  float vrat_real();
  float vrat_imag();
  float velikost();
};
```

Implementace metod

- obvykle se píše odděleně od deklarace, do zvláštního souboru

```
typ Jmeno_tridy::metoda(parametry)
{
  kód
}
```

- soubor `komplex.cpp` bude obsahovat:

```
#include "komplex.h"

void TKomplex::nastav(float real, float imag)
{
  re = real; im = imag;
}

float TKomplex::vrat_real()
{
  return re;
}
```

- nepsaným pravidlem je, že atributy jsou **`private`** (nebo **`protected`**);
- je-li potřeba nastavit hodnotu „zvnějšku“, nadefinuje se metoda s názvem `set_`, resp. `nastav_`, např.

```
nastav_img(float img)
```

- hodnota atributu se získá voláním metody   s názvem `get_`, resp. `vrat_`, např.

```
float vrat_real()
```

Deklarací třídy a zápisem implementace nevzniká žádný objekt!

To je pouze definice nového datového typu, tj. předpis pro překladač, jak mají konkrétní objekty daného typu vytvořit!

Objekt je tedy proměnná typu třídy (instance třídy).

- implementaci jednoduchých funkcí lze zapsat již při deklaraci (tzv. **inline** funkce), překlad inline však není zaručen

```
class TKomplex
{
  float re,im; //reálná a imag. část
 public:
  void nastav(float real, float imag);
  float vrat_real() { return re; };
  float vrat_imag();
  float velikost();
};
```

- klíčové slovo **inline** je možné uvést:

```
class TKomplex
{
  float re,im; //reálná a imag. část
 public:
  void nastav(float real, float imag);
  inline float vrat_real() { return re; };
  float vrat_imag();
  float velikost();
};
```

- kód inline funkce je přímo vložen do programu místo volání instrukcí call

- pokud nazveme parametr metody stejně jako atribut, dojde k zastínění atributu; v takovém případě se na atribut odkážeme pomocí názvu třídy:

```
void TKomplex::nastav(float re, float im)
{
  TKomplex::re = re;
  TKomplex::im = im;
};
```

- jiná možnost je pomocí klíčového slova **`this`**; **`this`** je ukazatel na „sebe sama“, tj. obsahuje adresu *konkrétního objektu*, nad kterým je metoda vyvolána

```
void TKomplex::nastav(float re, float im)
{
  this -> re = re;
  this -> im = im;
};
```

Konečně ...
Jak to použijeme v programu?

```
#include "komplex.h"
void main()
{
  TKomplex c1,c2;
  c1.nastav(4,3); c2.nastav(0,0);
  cout << "Realna cast cisla je " <<
          c1.vrat_real() << endl;
  cout "Velikost je " << c1.velikost() <<
  endl;
}
```

zde překladač vytvoří dva objekty c1 a c2 typu (třídy) TKomplex

zde se volá metoda velikost() nad objektem c1

- deklarace objektu:

```
TKomplex c1,c2;
```

  - v tomto momentě vyhradí překladač paměťový prostor pro dva objekty (dvě instance), tj. pro dva atributy na každý objekt
    - počáteční hodnoty atributů nejsou definovány
  - jinak řečeno, objekt je proměnná typu třída
- přístup k atributům a metodám
  - *tečkovou notací* jako u struktur, např.:

```
c1.velikost()
```

  - při volání metody je jí automaticky (skrytě) předán další parametr - ukazatel na konkrétní objekt, nad kterým je volána, zde `c1`

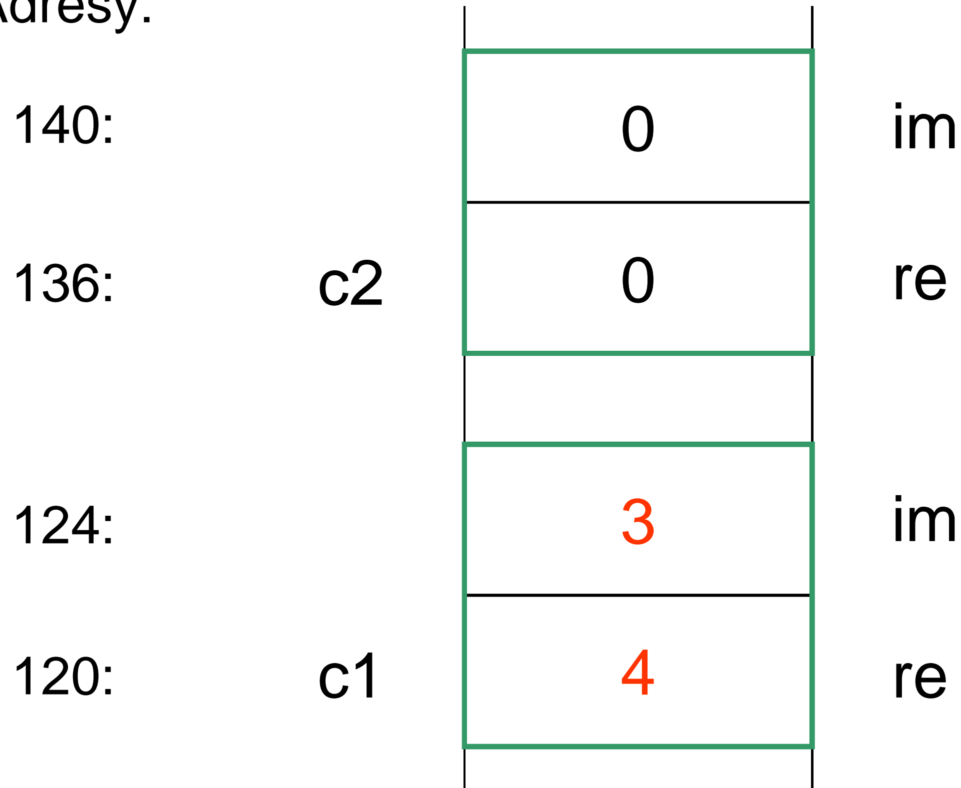
- hodnota ukazatele na „sebe sama“ je v metodě dostupná pomocí **`this`**

- po skončení programu, resp. při výstupu z bloku, jsou objekty `c1` a `c2` automaticky zrušeny (jsou ve třídě **`auto`**)

Poznámka:

- při deklaraci objektu je alokována paměť pouze pro atributy, metody jsou společné (kód je v paměti na jediném místě)!
- konkrétní objekt se také nazývá v terminologii OOP **instance třídy**

Objekty `c1, c2` v paměti

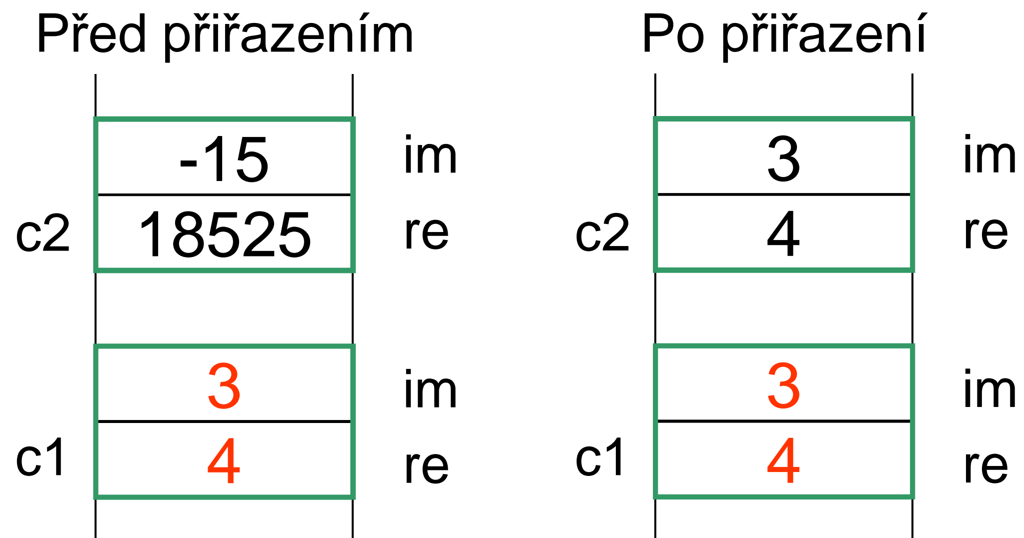


- při volání metody nad objektem, např. `c1.velikost()`, je metodě automaticky předána adresa objektu c1, tj. zde 120, je obsahem `this`

Přiřazování objektů

- objekty lze přiřazovat mezi sebou
- při přiřazení se provádí ***bitová kopie objektů***
  - kopírují se hodnoty atributů

```
TKomplex c1,c2;
c1.nastav(4,3);
c2 = c1;
```



Dynamická alokace objektu za běhu programu

- deklarujeme ukazatel na objekt, alokujeme pomocí **`new`**, k položkám přistupujeme pomocí `->`, dealokujeme funkcí **`delete`**

```
TKomplex *c3;
c3 = new TKomplex;
c3 -> nastav(5.2,4);
delete c3;
```

Poznámky:

- při dynamické alokaci se musí programátor postarat i o dealokaci
- alokovat a dealokovat objekty lze jen pomocí **`new`** a **`delete`** (jsou pro tento účel přetíženy)
  - při použití malloc některé věci nefungují, např. volání konstruktoru, viz další hodina
- pomocí ukazatelů na objekty se realizují vazby mezi objekty - atributem je ukazatel na jiný objekt

Poznámky k řízení viditelnosti

- pokud jsou atributy nebo metody deklarovány jako **`private`** (viz atributy `re`, `im` ve třídě TKomplex), pak u příkazů

```
c1.re = 3;
cout << c1.re;
```

ohlásí překladač chybu, že nelze přistupovat k soukromým položkám třídy, pokud se tyto příkazy vyskytnou jinde než v členských funkcích

Poznámky k řízení viditelnosti

- pokud bychom atributy `re`, `im` deklarovali jako veřejné, metody `nastav`, `vrat_real`, `vrat_imag` jsou de facto zbytečné
  - v této třídě asi není zvláštní důvod mít atributy soukromé, změna mimo členské funkce nevadí

```
class TKomplex
{
 public:
  float re,im;
  void nastav(float real, float imag);
  float vrat_real();
  float vrat_imag();
  float velikost();
};
```

Poznámky k řízení viditelnosti

```
#include "komplex.h”
void main()
{
  TKomplex c1,c2;
  // OK, pokud re a im jsou verejne
  c1.re=4; c1.im=3; c2.re=0; c2.im=0;
  cout << "Realna cast cisla je “ <<
          c1.vrat_real() << endl;
  cout "Velikost je “ << c1.velikost() <<
  endl;
}
```

Srovnání – ještě jednou neobjektově pomocí struktur

```
struct TKomplex
{
float re,im; //reálná a imag. část
}
// vnější samostatné funkce
float vrat_real(Tkomplex kc);
float vrat_imag(Tkomplex *kc);
float velikost(Tkomplex &kc);
```

Neobjektově pomocí struktur

```
 float vrat_real(Tkomplex kc)
{
  return kc.re;
};
float vrat_imag(Tkomplex *kc)
{
 return kc->re;
};
float velikost(Tkomplex &kc)
{
  return sqrt(kc.re*kc.re+kc.im*kc.im);
}
```

Neobjektově pomocí struktur

```
#include "komplex.h”
void main()
{
  TKomplex c1,c2;
  c1.re = 4; c1.im = 3;
  c2.re = 0; c1.im = 0;
  cout << "Realna cast c1 je " << c1.re << endl;
  cout << “Imag. cast c1 je " << vrat_imag(&c1) << endl;

  cout "Velikost c2 je “ << velikost(c2) << endl;
  //srovnejte: volani metody nad objektem: c2.velikost()
}
```

Rozdíl mezi `struct` a `class` v C++

- `struct` je v C++ rozšířena o objektové rysy, tj. třídu lze deklarovat pomocí `class` i pomocí struct (do struktury lze zahrnout metody)
- jediný rozdíl je v implicitní viditelnosti
  - pokud není viditelnost deklarována explicitně, tak u `class` je automaticky `private`, u `struct` je automaticky `public`

K čemu jsou dobré soukromé atributy?

- na minulé hodině:

```
int vloz(int &n, int &akt, int **pole,
  int prvek)
{
  …
}
```

- hlavní program

```
void main(void)
{
  int n=0;
  int akt=0;
  int pole = NULL;
  vloz(n,akt,*pole,3);
  /* toto může udělat programátor
 omylem */
  n = 10; akt = 50;
}
```

Co nám na tom vadí?

- počet prvků pole `n` a vlastní `pole` pro uložení dat není svázáno
- hodnotu `n` lze měnit kdykoliv (např. omylem při chybě programátora); správně by měla být měněna pouze při operaci `vloz`; jinak není zaručena konzistence dat (logická správnost)

```
class BezpPole
{
 private:
  int n,akt,*pole;
 public:
  void vloz(int prvek);
  int vrat_prvek(int index);
  int vrat_akt_pocet();
  void init();
};
```

```
void BezpPole::init()
{
 n = 0; akt = 0; pole = NULL;
}
int BezpPole::vrat_prvek(int index)
{
  if (index>=0 && index < akt)
    return pole[i];
  else
    return -1;
}
```

```
void BezpPole::vloz(int prvek)
{
   if (n==0)
  {
    pole = new int[20]; n = 20;
  }
  if (akt == n)
  { int *pom; pom = new int[n+20];
    memcpy(pom,pole,n*sizeof(int));
    n += KROK; delete [] pole; pole = pom;
  }
  pole[akt++] = prvek;
}
```

```
void BezpPole::vrat_akt_pocet()
{
  return akt;
}
```

```
void main()
{
  int i;
  BezpPole a;
  a.init(); // na init nesmíme zapomenout
  a.vloz(3); a.vloz(4); a.vloz(2);
  for(i=0;i<a.vrat_akt_pocet();i++)
    cout << a.vrat_prvek(i) << endl;
}
```

- jak zařídit, abychom nemuseli myslet na to, že nesmíme zapomenout zavolat na začátku init, si povíme příště – o tzv. konstruktorech

Úkol

Navrhněte a naimplementuje třídu, která uchovává informace o bodech v dvourozměrném prostoru. Vymyslete vhodné metody. Ověřte chování na jednoduchém programu

Atributy:

```
double x,y;
```

Metody:

nastavení hodnot podle parametrů x,y

nastavení hodnot podle polárních souřadnic

nastavení hodnot podle jiného objektu bod

výpočet vzdálenosti od počátku

výpočet vzdálenosti od jiného bodu

výpočet úhlu (polární souřadnice)

posuv bodu - podle hodnot dx, dy

tisk – vytiskne souřadnice ve tvaru [x,y]

```
class TBod
{
 private:
   double x,y;
public:
  void nastav(double px, double py);
  void nastav_polar(double vzdal, double uhel);
  //uhel je v radianech
  // nastaveni podle jineho bodu
  void nastav(TBod &bod);
  ...

}
```