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 ...

Objektově orientované programování (OOP)

- vzniklo již v 70. letech
  - Smalltalk – čistý objektový jazyk
  - Simula67 – objektový jazyk pro simulace
- prudký rozvoj od 80. a v 90. letech
  - začlenění do jazyků C - vznik C++ (1983), Pascal, Visual Basic,
  - Java, C#
  - dnes také součástí jazyků php, Python

Struktura pro reprezentaci komplexních čísel – neobjektově

- soubor komplex.h

```
typedef struct
{
  float re,im; //reálná a imaginární část
} Komplex;

// funkce pro výpočet velikosti kompl. čísla
float velikost(Komplex &c);
```

Struktura pro reprezentaci komplexních čísel - neobjektově

- soubor komplex.c

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

Struktura pro reprezentaci komplexních čísel - neobjektově

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

  cout << "Velikost c1 je " << velikost(c1) << 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

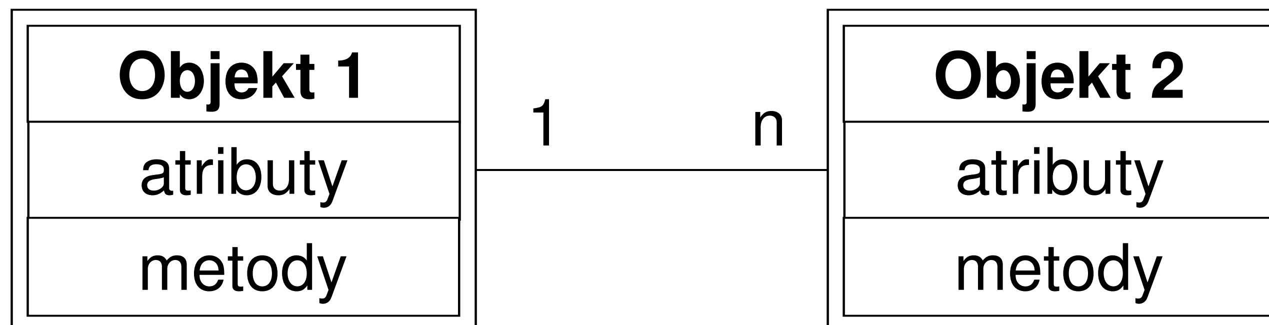
- atributy a metody jsou úzce svázány (syntakticky i sémanticky); objekt je zapouzdřením (encapsulation) atributů a metod
- objekty mezi sebou komunikují vzájemným voláním metod (říkáme též, že si předávají zprávy)

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
{ ... };
```

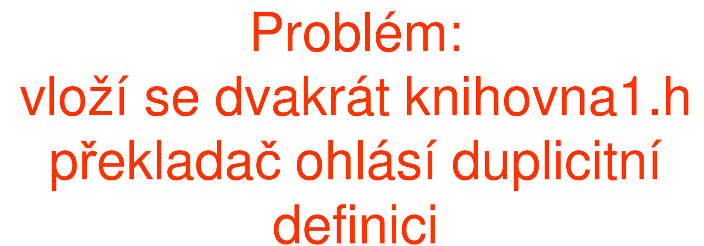
Soubor **knihovna3.h**

```
#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` ... (**strážný blok – guard block**)

Příklad:

Navrhneme třídu pro reprezentaci komplexních čísel tentokrát objektově.

- deklaraci třídy zapíšeme do souboru `komplex.h`

#ifndef KOMPLEXH
#define KOMPLEXH
class Komplex
{
float re,im; //reálná a imag. část
float velikost();
};
#endif
atributy
metoda

Implementace metod

- obvykle se píše odděleně od deklarace, do zvláštního souboru .cpp

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

- soubor `komplex.cpp` bude obsahovat:

```
#include "komplex.h”

float Komplex::velikost()
{
  return sqrt(re*re+im*im);
}
```

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).

Jak to použijeme v programu?

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

zde se vytvoří dvě proměnné – dva objekty c1 a c2 typu (třídy) Komplex

Zde překladač ohlásí chybu

zde se volá metoda velikost() nad objektem c1

Proč ohlásí překladač chybu?

- atributy a metody jsou automaticky tzv. *soukromé* (private), tj. lze s nimi manipulovat pouze uvnitř *členských funkcí* (metod třídy)
- musíme deklarovat atributy a metody, které chceme zpřístupnit (zveřejnit), jako *veřejné* (public)

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)

```
#ifndef KOMPLEXH
#define KOMPLEXH
class Komplex
{
 public:
  float re,im; //reálná a imag. část
  float velikost();
};
#endif
```

Jak to použijeme v programu?

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

Nyní již program přeložit půjde

Poznámky k řízení viditelnosti

- někdy je užitečné mít soukromé atributy a metody
  - např. když hodnota atributu vyjadřuje nějaký stav objektu a není žádoucí, aby programátor "z vnějšku" hodnotu měnil, ať úmyslně nebo omylem
    - implementujeme objektově množinu a jeden atribut je počet prvků množiny; je zřejmé, že hodnota tohoto atributu se má měnit pouze, když se volají metody vložení a vyjmutí prvku

- rovněž některé metody, které slouží jako pomocné, se deklarují jako soukromé
- seznam veřejným metod pak tvoří tzv. ***interface třídy (rozhraní)***
- je-li potřeba získat hodnotu soukromého atributu, nadefinuje se veřejná metoda s typickým názvem `get_`, resp. `read_`, `cti_`, např. **`float`** `get_img()` vracející hodnotu atributu

- je-li potřeba nastavit hodnotu soukromého atributu, nadefinuje se veřejná metoda s typickým názvem `set_`, resp. `write_, nastav_`, např.

```
void set_img(float imag)
```

  – deklarovat atributy jako soukromé je vhodné, když je omezena množina hodnot atributů; pak veřejná metoda `set_` kontroluje, zda nastavovaná hodnota není mimo rozsah

- komplex.h

```
#ifndef KOMPLEXH
#define KOMPLEXH
class Komplex
{ private:
   float re,im; //reálná a imag. část
  public:
   float get_real();
   float get_img();
   void set_real(float real);
   void set_img(float imag);
   float velikost();
};
#endif
```

- komplex.cpp

```
#include "komplex.h”

float Komplex::get_real()
{
  return re;
}

float Komplex::get_img()
{
  return im;
}
```

- komplex.cpp

```
#include "komplex.h"

void Komplex::set_real(float real)
{
  re = real;
}

void Komplex::set_img(float imag)
{
  im = imag;
}
```

```
float Komplex::velikost()
{
  return sqrt(re*re+im*im);
}
```

```
#include "komplex.h”
void main()
{
  Komplex c1,c2;
  c1.set_real(4); c1.set_img(3);
  c2.set_real(0); c2.set_img(0);
  cout << "Realna cast c1 je " <<
  c1.get_real() << endl;
  cout << "Velikost c1 je " << c1.velikost()
  << endl;
}
```

U této třídy nemá praktický smysl deklarovat atributy re, im jako soukromé

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

```
class Komplex
{
  float re,im; //reálná a imag. část
 public:
  void set_real(float real);
  void set_img(float imag);
  float get_real() { return re; };
  float get_img();
  float velikost();
};
```

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

```
class Komplex
{
   float re,im; //reálná a imag. část
 public:
   void set_real(float real);
   void set_img(float imag);
   inline float get_real() { return re; };
   float get_img();
   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 Komplex::set_real(float re)
{
  Komplex::re = re;
};
```

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

```
void Komplex::set_real(float re)
{
  this -> re = re;
};
```

- deklarace objektu:

```
Komplex c1,c2;
```

  - vyhradí se paměťový prostor pro dva objekty (dvě **instance třídy**), tj. pro dva atributy na každý objekt
    - počáteční hodnoty atributů nejsou definovány, jsou náhodné
  - 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`

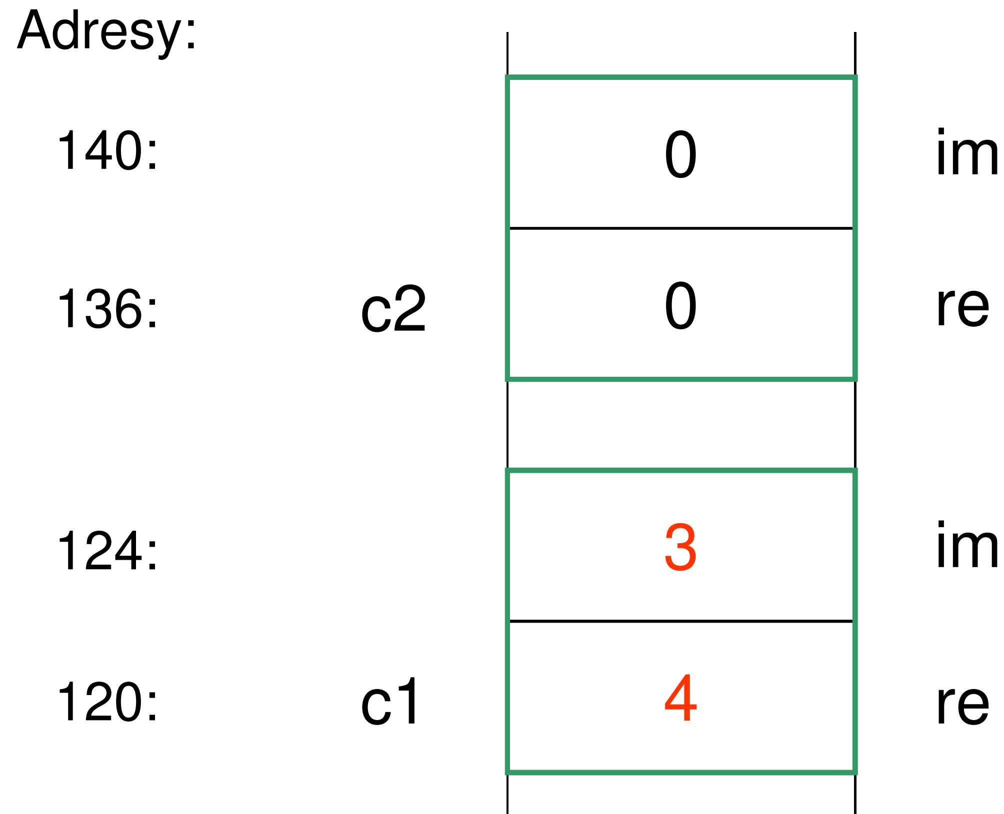
- hodnota ukazatele na „sebe sama“ je v metodě dostupná pomocí **`this`**

- po ukončení 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 pro ilustraci adresa 120, tato adresa je obsahem **`this`**

Přiřazování objektů

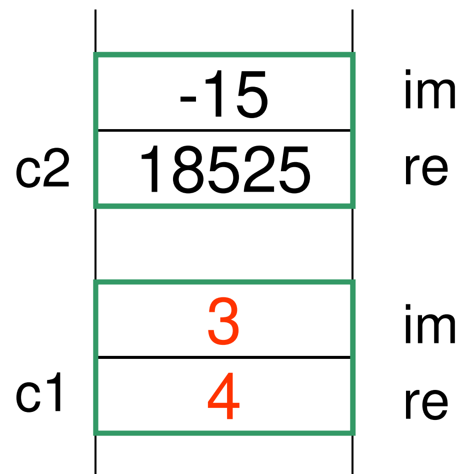
- objekty lze přiřazovat mezi sebou
- při přiřazení se provádí ***bitová kopie objektů***
  - kopírují se hodnoty atributů

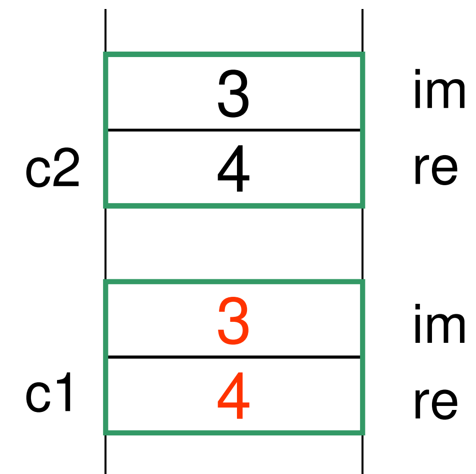
```
Komplex c1,c2;

c1.set_real(4);

c1.set_img(3);

c2 = c1;
```





Dynamická alokace objektu za běhu programu

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

```
Komplex *c3;
c3 = new TKomplex;
*c3.set_real(5.4);
c3 -> set_img(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, budeme pribírat později
- pomocí ukazatelů na objekty se realizují vazby mezi objekty - atributem je ukazatel na jiný objekt

Srovnání – ještě jednou neobjektově pomocí struktur

```
struct Komplex
{
float re,im; //reálná a imag. část
}
// vnější samostatná funkce
float velikost(Komplex &kc);
```

Neobjektově pomocí struktur

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

Neobjektově pomocí struktur

```
#include "komplex.h”
void main()
{
  Komplex c1,c2;
  c1.re = 4; c1.im = 3;
  c2.re = 0; b1.im = 0;
  cout << "Realna cast c1 je " << c1.re << 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?

```
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[index];
  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 += 20; delete [] pole; pole = pom;
  }
  pole[akt++] = prvek;
}
```

```
int 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 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 Bod
{
 public:
  double x,y;
  void nastav_polar(double vzdal, double uhel);
  //uhel je v radianech

  // nastaveni podle jineho bodu
  void nastav(TBod &bod);
  double vzdalenost(Bod &b);
  ...
  void tisk();
}
```