Vývoj programátorských štruktúr

Andrej Lúčny

Katedra aplikovanej informatiky FMFI UK

a MicroStep-MIS

andy@microstep-mis.com

http://www.microstep-mis.com/~andy

Vývoj programátorských štruktúr

- **Neštruktúrované programovanie**
  - základné typy
- **Štruktúrované programovanie**
  - záznamy a množiny
- **Objektovo-orientované programovanie**
  - triedno-inštančný model
  - polymorfizmus, dynamická väzba
  - aktorový model, súbežné OOP
- **(Zovšeobecné) agentovo-orientované programovanie**
  - agenty, silné a slabé
- Sprievodné javy
  - šetrenie s pamäťou a likvidácia odpadu
  - menné priestory
  - ošetrovanie výnimiek
  - aspektovo orientované programovanie

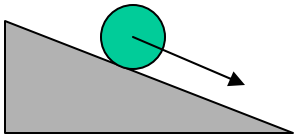
Načo štruktúry pri programovaní potrebujeme ?

- **Modelovanie reálneho sveta v počítači**
- (Vytváranie virtuálnych svetov v počítači)

Ako môžeme niečo reálne preniesť do počítača?

Neštruktúrované programovanie

- do počítača prenášame len jednotlivé parametre entity, ktoré môžeme merať v reálnom svete
- premenné, hlavný program, podprogramy
- základné typy premenných: integer, real, double, boolean, void, complex, ...
- polia, 1,2,3-trojrozmerné polia
- dynamika: dĺžka polí, inak statické štruktúry



Neštruktúrované programovanie

```
#include <math.h>

int main() {
   double x=0.0, y=5.0, fi=0.56;
   int t;
   for (t=0; t<10; t++) {
       x += cos(fi);
       y += sin(fi);
   }
}
```

Prototypy

- Implementácia na inom mieste než použitie

```
extern void posun (
   double *a, double *b,
   double fi
);
int main() {
   double x=0.0, y=5.0;
   double fi=0.56;
   int t;
   for (t=0; t<10; t++)
     posun (&x,&y,fi);
}
```

```
void posun (
   double *a, double *b,
   double fi
)
{
   *a += cos(fi);
   *b += sin(fi);
}
```

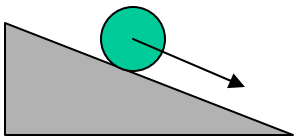
Deskriptory

- z knižnice ktorá pracuje pre jednu entitu urobíme takú ktorá pracuje pre mnoho jej klonov

```
#define m 100
double x[m],y[m];

void posun(int i, double fi) {
   int t;
   for (t=0; t<10; t++) {
       x[i] += cos(fi);
       y[i] += sin(fi);
   }
}
```

Štruktúrované programovanie

- do počítača prenášame pasívnu entitu
- jedna štruktúra môže mať viac parametrov
- chápe sa ako dodefinovaný zložený typ
- štruktúra typu záznam (record, struct)
- varianty (case of, union)
- skladanie štruktúr
- dynamika: statické aj dynamické štruktúry



Štruktúrované programovanie

```
#include <math.h>
typedef struct gulka {
   double x;
   double y;
} GULKA;

void posun(
   GULKA *gul,
   double fi
){
   gul->x += cos(fi);
   gul->y += sin(fi);
}
```

```
int main() {
   GULKA g, *h;
   double alpha = 0.56;
   int t;
   g.x = 0.0; g.y = 5.0;
   h = (GULKA*)
       malloc(sizeof(GULKA));
   h->x = 0.0; h->y = 5.0;
   for (t=0; t<10; t++) {
       posun(&g,alpha);
       posun(h,alpha);
   }
   free(h);
}
```

Opaque pointery (neprehliadné smerníky)

- **zakrytie implementačných detailov v knižnici (enkapsulácia)**

```
int main() {
  GULKA *h;
  double alpha = 0.56;
  int t;
  h = (GULKA*)
      malloc(sizeof(GULKA));
  h->x = 0.0; h->y = 5.0;
  for (t=0; t<10; t++)
      posun(h,alpha);
  free(h);
}
```

```
int main() {
  GULKA *h;
  double alpha = 0.56;
  int t;
  h = open_gulka();

  set_gulka(h,0.0,5.0);
  for (t=0; t<10; t++)
      move_gulka(h,alpha);
  close_gulka(h);
}
```

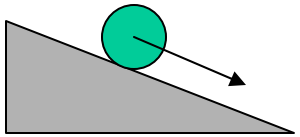
Pridávanie kódu k premenným

```
#include <math.h>

typedef struct gulka {
   double x;
   double y;
   void (*p)(GULKA *g);
} GULKA;

void posun30(
   GULKA *g
){
   g->x += cos(0.56);
   g->y += sin(0.56);
}
```

```
void posun45(
   GULKA *g
){
   g->x += cos(0.78);
   g->y += sin(0.78);
}

int main() {
   GULKA g =
       {1.0, 5.0, posun30};
   int t;
   for (t=0; t<10; t++) {
       g.p(&g);
   }
}
```

Objektovo orientované programovanie

- do počítača prenášame reaktívnu entitu, obsahujúcu nielen dáta, ale aj kód, ktorý s nimi manipuluje, keď ho vyvoláme
- jedna štruktúra môže mať viac atribútov a metód
- triedno-inštančný prístup
- dynamika: hlavne dynamické štruktúry, statické napr. atribúty triedy



OOP

```
class Gulka {
   private:
      double x;
      double y;
   public:
      Gulka(double _x, double_y);
      void posun(double fi);
}

void Gulka::posun(
   double fi
){
   x += cos(fi);
   y += sin(fi);
}
```

```
Gulka::Gulka (double _x,
   double_y) {
   x = _x;
   y = _y;
}
int main() {
   Gulka *g =
      new Gulka(1.0, 5.0);
   Gulka *h =
      new Gulka(0.0,5.0);
   double alpha = 0.56;
   int t;
   for (t=0; t<10; t++) {
       g->posun(alpha);
       h->posun(alpha);
   }
   delete g; delete h;
}
```

Triedno-inštančný prístup

- každý objekt vzniká ako inštancia triedy
- trieda definuje jeho jeho atribúty a metódy
- zložený typ je obohatený na triedu
- štruktúra je obohatená na objekt
- z premenných sa stávajú atribúty
- z funkcií a procedúr metódy

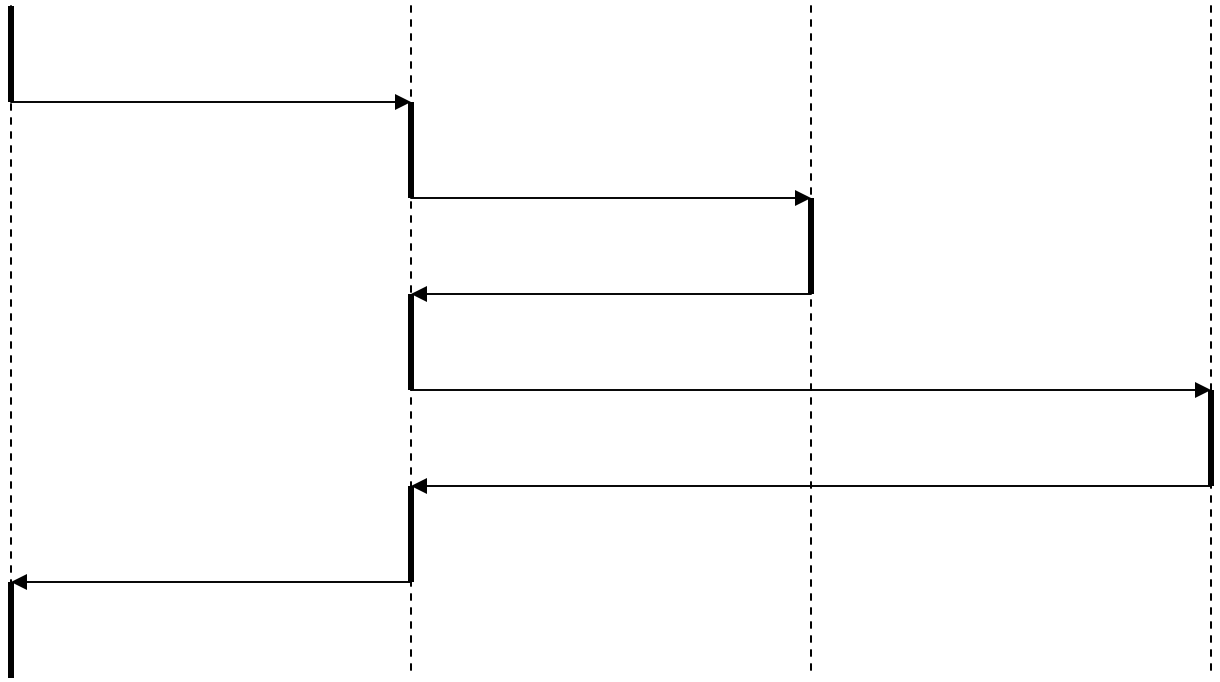
Konštruktory a deštruktory

- Konštruktor – náhrada za malloc() a úvodné priradenia
- Deštruktor – náhrada za free()

```
Gulka::Gulka (
   double _x,
   double_y)
{
   x = _x;
   y = _y;
}

Gulka::~Gulka ()
{
   std.cout << x << ″,″ <<
   y << ″ has finished″;
}
```

Riadenie v klasickom modeli

- riadenie sa odovzdáva zavolaním metódy iného objektu a vracia odovzdaním návratovej hodnoty z nej



Preťaženie (overloading)

- zdieľanie mena metód viacerými metódami ktoré sa odlišujú počtom a typom parametrov

```
void Gulka::posun (double fi)
{
   x += cos(fi);
   y += sin(fi);
}

void Gulka::posun ()
{
   posun(0.0);
}
```

Dedenie (inheritance)

- vzniká analógiou zo skladania typov
- umožňuje aby sme jednu triedu definovali ako špeciálnejší prípad inej

```
class ValivaVec {
  private:
    double x;
    double y;
  public:
    ValivaVec (
      double _x,
      double_y
    );  …
}
```

```
class Gulka : ValivaVec {
  public:
    Gulka (
      double _x,
      double _y
    ); …
}
```

Prekrytie (overriding)

- pritom pri definícii metód použiť zdedené metódy (scoping), ale môžeme ich aj prekryť

```
ValivaVec :: ValivaVec (
  double _x, double_y
){
  x = _x; y = _y;
}
```

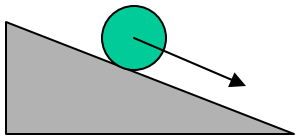
```
Gulka :: Gulka (
  double _x, double_y
){
  ValivaVec::ValivaVec(
    _x, _y
  );
}
```

```
void ValivaVec :: vypis ()
{
  std.cout << ″ValivaVec″;
}
```

```
void Gulka :: vypis ()
{
  std.cout << ″Gulka″;
}
```

Interface

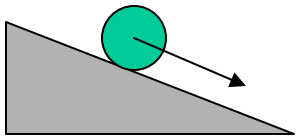
- vyvinul sa z prototypov
- abstraktná trieda
- nedefinuje atribúty, len metódy a tým definuje len ako sa volajú, a nie ako sú implementované
- možnosť definovať tvar objektov, ktoré už potrebujeme použiť, ale ešte nie sú vytvorené
- lebo vytvoriť ich má vytvoriť až užívateľ nami navrhovanej mašinérie



Interface

```
class Valenie {
  private:
    ValivaVec *v;
  public:
    Valenie (ValivaVec *v);
    void run (int n);
}
Valenie::Valenie (ValivaVec *v){
  this->v = v;
}
Valenie::run (int n) {
  int t; for (t=0; t<n; t++)
    v->posun();
}
```

```
class ValivaVec {
  private:
    double x;
    double y;
  public:
    ValivaVec (
      double _x,
      double_y
    );
    virtual void posun (
      double fi
    );
}
```



Interface

```
class Gulka : ValivaVec {
}

void Gulka::posun (double fi)
{
   x += cos(fi);
   y += sin(fi);
}

int main () {
   Gulka *g = new Gulka(1,1);
   Valenie *v = new Valenie(g);
   Valenie.run(0.56);
}
```

Najprv jeden vývojár implementuje Valenie, ale nevie ešte čo sa bude valiť, tak valí ValivaVec

Oveľa neskôr druhý implementuje ako sa valí guľka a použije prácu prvého vývojára

Polymorfizmus

- jeden objekt môže byť vnímaný ako inštancia viacerých tried, napr. `Gulka` môže byť vnímaná aj ako `Gulka`, aj ako `ValivaVec`
- Keď potom voláme určitú všeobecnú metódu ako je `posun(double fi)` , raz sa volá kód pre `Gulka` inokedy pre `Hranol` alebo `Valcek`. Táto schopnosť sa nazýva dynamická väzba

```
int main () {
  ValivaVec *a[2] = {
    new Gulka(1,1), new Hranol(2,2,0.5,0.8) };
  for (int i=0; i<2; i++) a[i]->posun(0.56);
}
```

Propagácia udalostí

- jeden objekt udalosť generuje druhý ju prijíma
- modely: signal-slot, action-listener (event-driven)

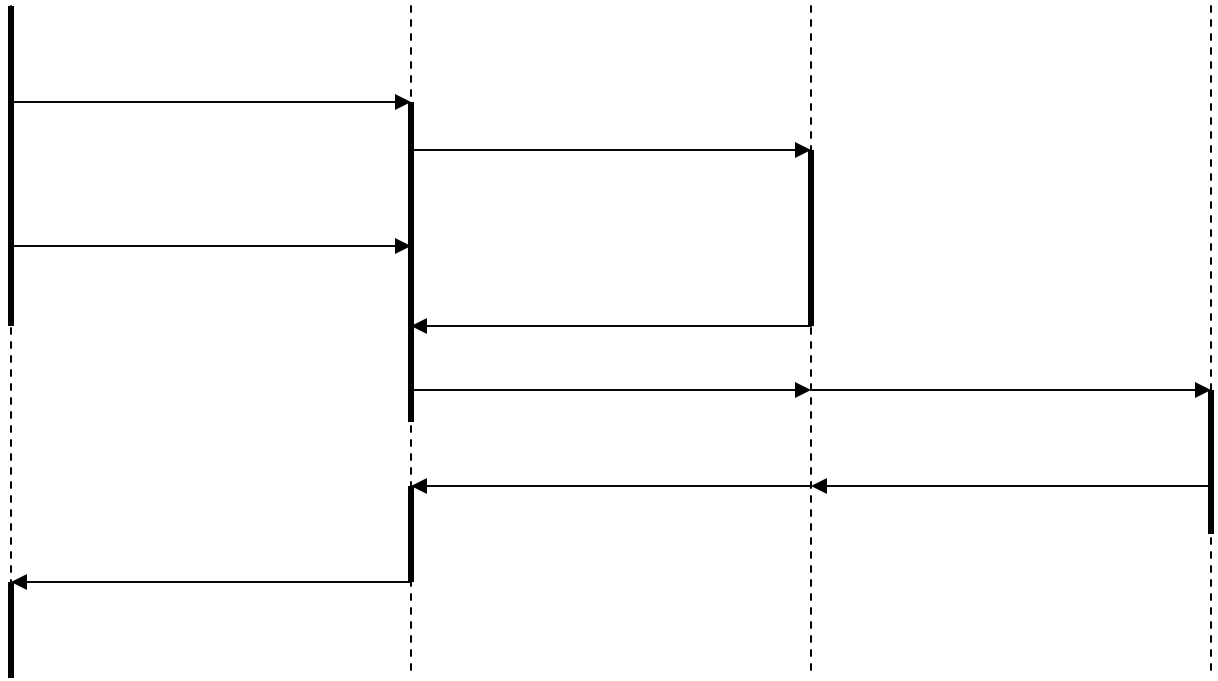
```
void Gulka::addListener (GulkaListener *listener) {
  listeners.push_back(listener); cnt++;
}

void Gulka::posun (double fi) {
  x += cos(fi);
  y += sin(fi);
  for (int i=0; i<cnt; i++)
    listeners[i].actionPerformed();
}
```

Objektovo založené programovanie

- OOP mieša prácu so základnými typmi a objektami
- výhoda: používame čo sa kde hodí
- nevýhoda: nemôžeme volať metódy na premenné základného typu
- alternatívny prístup: 2 + 2 znamená 2.+(2)
  t.j. že na objekte 2 voláme metódu + s argumentom ktorým je objekt 2 a výsledkom volania je objekt 4
- C++ je OOP, zatiaľ čo napr. Smalltalk je OBP

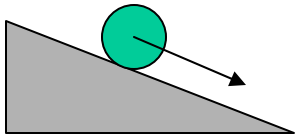
Aktory

- Alternatívny model výpočtu v OOP
- miesto synchrónneho volania metód sa vykonáva asynchrónne posielanie správ
- miesto dedičnosti sa používa delegovanie



(Zovšeobecnené) Agentovo orientované programovanie

- do počítača prenášame proaktívnu entitu, obsahujúcu nielen dáta a kód, ale aj vlastným vláknom (program counter)
- synchronizáciu zabezpečuje len komunikačný mechanizmus.
- priama komunikácia je veľmi podobná aktorom, nepriama môže byť na rozdiel od aktorov stratová



Agenty

```
class Gulka : Agent {
  private:
    double x;
    double y;
    void posun (double fi);
  public:
    Gulka(double _x,double_y);
    void handleEvent();
}
Gulka::Gulka (double _x,
  double_y) {
  x = _x;
  y = _y;
  attachTrigger(˝fi˝);
}
```

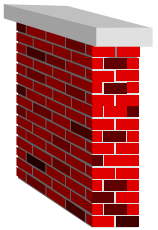
```
Gulka::handleEvent() {
  alpha =
    getSpace().get(˝fi˝);
  this->posun(alpha);
}

int main() {
  Agent gulka =
    new Gulka(1.0,5.0);
  Space space =
    Space.getInstance();
  int t;
  for (t=0; t<10; t++)
    Space.put(˝fi˝,0.56);
  gulka.stop();
}
```

Alternatívna propagácia udalostí

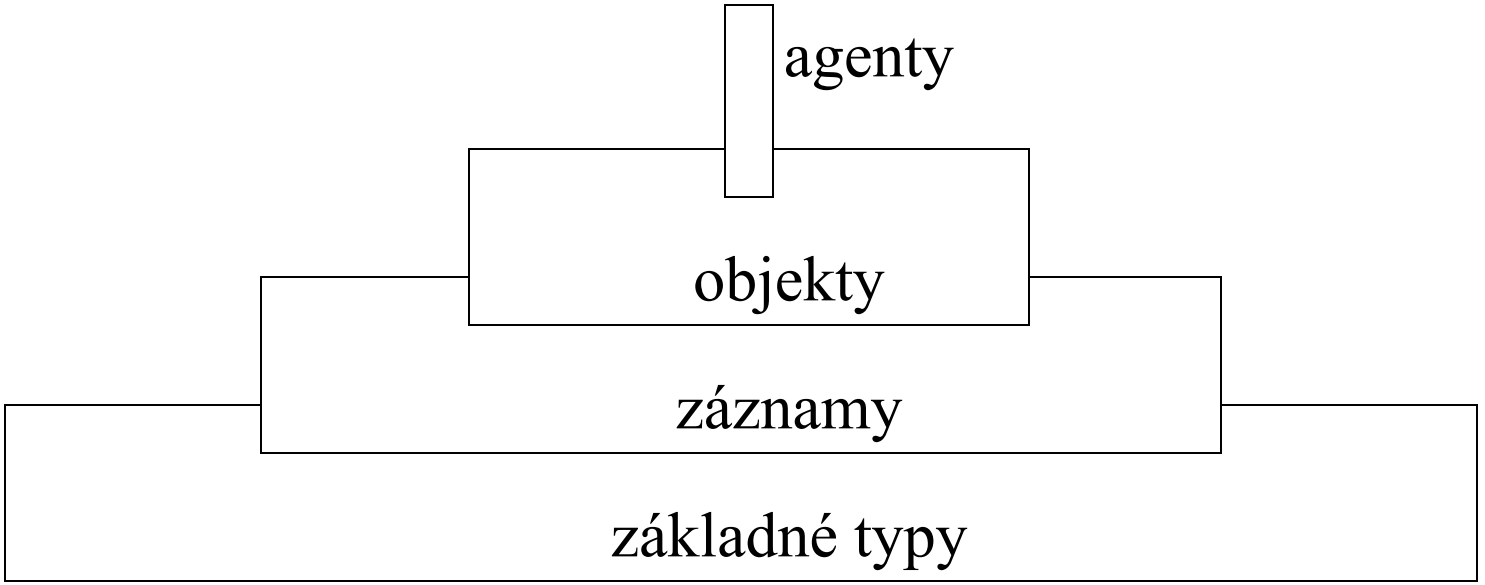
- jeden objekt udalosť generuje druhý prijíma len zobudenie, ale čo sa stalo si zistí sám
- tým sa veľa actionPerformed() transformuje na jeden kód, ktorú nie je event-driven

```
boolean Gulka::zrazka () {
  while (p=get(˝position.*˝))
    if (zrazka(p,this->p))
      return true;
}

void Gulka::posun (double fi) {
  p->x += cos(fi);
  p->y += sin(fi);
  put(˝position˝+id,p);
}

Gulka::Gulka (...) {
  attachTrigger(˝position.*˝);
}
```

Vývoj aktivity štruktúr

pasívna entita		**záznam, (record, struct)**
reaktívna entita		**objekt**
proaktívna entita		**agent**

Jeden typ štruktúr nevytesňuje iný úplne



Sprievodné javy vývoja štruktúr

Nešetrí sa pamäťou	garbidge collection
Nešetrí sa výkonom	vlákna, procesy, časovače, komponenty, decentralizácia
Šetrí sa programátorská práca	výnimky, aspektovo orientované prog., dev. tooly, vizuálne prog.

Balíkovanie

- knižnice sú tak bohaté, že sa už nedá spoliehať sa všetky názvy typov líšia
- preto majú typy štruktúr dlhé mená ako `fyzika.mechanika.valenie.Gulka`
- programy potom sprehladnuju importy či menné priestory (namespace), umožňujúce používať na určitom mieste kódu iba označenie `Gulka`

Schémy (Template)

- do jazyka sa zavádzajú rôzne prvky „druhého rádu“ umožňujúce písať jediný kód pre viac typov súčasne

```
template <class myType> myType GetMax (
   myType a,
   myType b
)
{
   return (a > b ? a : b);
}
```

Reflečný model

- Schopnosť štruktúry podať informáciu o svojom zložení počas behu programu, t.j. napr. vygenerovať kód z ktorého bola skompilovaná
- C++ nemá reflekčný model, ale Java áno
- pokiaľ nie je r.m. k dispozícii, musí sa nahradzovať súborom, ktorý štruktúru popisuje (napr. IDL)

Mobilné štruktúry

- štruktúra ktorá sa vie prešťahovať z jedného procesu (virtuálnej mašiny, počítača) do druhého
- Marshalling – premena štruktúry na postupnosť bytov
- Demarshalling – premena postupnosti bytov na štruktúru
- = serializovateľnosť

Verzionovanie

- S potrebou distribuovaných systémov z ktorých každý môže byť skompilovaný s inou verziou štruktúry alebo inou štuktúrou toho istého mena, prichádza potreba tieto (mobilné) štruktúry rozlíšiť
- riešenie: UUID, GUID = číslo generované generátorom, ktorý po mnoho ďalších tisícročí zopakuje dve čísla s pravdepodobnosťou blízkou nule.

```
public class Gulka {
   static final long serialVersionID = 667788901L;
   ...
}
```

Interface druhého rádu

- beans, widgets, ...
- implementujú metódy určitého dohodnutého tvaru mien (prefixy, suffixy) avšak nie dohodnuté mená
- tým je určené, ktoré metódy treba zavolať za určitým účelom (napr. nastavenia hodnoty pri bean-e alebo zobrazenia pre widget-y)
- podpora vizuálneho, distribuovaného alebo komponentového programovania

Ďakujem za pozornosť !

Vývoj programátorských štruktúr

Andrej Lúčny

KAI FMFI Bratislava & MicroStep-MIS

andy@microstep-mis.com

http://www.microstep-mis.com/~andy