Objektové programování v C

Marcel Baláš

22. července 2013

Outline

Outline

Základní pojmy

Objekt

”Černá skříňka”; má nějaké *vlastnosti* a nějaké *chování*; můžeme mu zaslat zprávu a počkat na odpověď (= zavolat jeho metodu).

Objektové programování

Volná síť komunikujících objektů.

Další pojmy

Třída

Druh objektu. (I třída může být objektem!)

Instance

Konkrétní *realizace* nějaké třídy.

Atribut

Vlastnost objektu. (Chápána šířeji i jako *součást* objektu.)

Example (atribut)

auto :: barva, velikost, volant, řadící páka

Metoda

Definuje chování objektu.

Další pojmy

Dědičnost

Typová hierarchie. Potomek dědí od předka všechny jeho atributy a chování (které však může pozměnit). Přidává nové atributy a/nebo chování.

Zapouzdření

Jinak řečeno ona ”černá skříňka”, neviditelnost atributů objektu navenek. Není možné měnit atributy objektu bez jeho vědomí.

Polymorfismus

Možnost zaslat různým objektům tutéž zprávu.[a]

[a] Obecně *nesouvisí* s dědičností!

Outline

Proč OOP?

- Objektový přístup jako analogie způsobu vytváření pojmů. Objekty v OOP modelují objekty z reálného světa.
- Pro člověka přirozený způsob analýzy a tvorby.
- Flexibilní spolupráce objektů ($\Rightarrow$ tvorba patternů, objektových hierarchií).
- Reuse!
- Rychlejší vývoj.
- Méně chyb, resp. jejich snadnější ladění.
- Jednodušší údržba a změny kódu.

Outline

S céčkem na OOP?

Proč C a nikoliv C++?

- Jednoduchost kódu, rychlost kódu, rychlost překladu, více platforem, menší binárka, nedostupnost C++ kompilátoru, ...
- C++ není snadný jazyk, není pouhou nadstavbou C, přidává vlastní ”pitfalls”.
- Rozhodnutí managementu.

Co by se nám v C hodilo z OOP?

- Dědičnost, polymorfismus, větší možnosti zapouzdření, přetížení argumentů, jmenné prostory?

S céčkem na OOP?

You can't eat a cake and have it too.

Neboli

Něco za něco.

- Psát a udržovat kompletní a univerzální dědičnost je bláznovství.[1]
- Omezené formy dědičnosti, polymorfismu či zapouzdření však lze dosáhnout poměrně snadno a elegantně! (viz ▸ Kompromisní řešení dědičnosti)

[1] Autor po jistou chvíli bláznem byl.

Outline

Třída

V céčku není jiné možnosti než implementovat třídu jako obyčejnou strukturu. (Ostatně v C++ tomu není jinak.)

```
typedef struct Empty
{
  char _empty;
} Empty;
```

Třída

Pojmenování a jmenné prostory

Z důvodu absence jmenných prostorů (balíčků) je v praxi u rozsáhlejších projektů vhodné použít aspoň jedno z následujících pravidel pro názvy tříd tak, aby nedocházelo k jejich konfliktům:

- přidat prefix související s projektem jako takovým – zamezí se konfliktům zejména s externími knihovnami,
- přidat prefix/suffix související s daným balíčkem (logickou doménou), kam patří – zamezí se konfliktům i mezi interními knihovnami.

Example

`PDMDataBuffer` **vs** `yOSDataBuffer`

`HwBuiltInTest` **vs** `BuiltInTestController`

Outline

Instance

Instance je reprezentována pomocí proměnné dané struktury, zpravidla ale ukazatelem na proměnnou.
Může být alokována na zásobníku, dynamicky na haldě, nebo jako `static`.[2]

```
Empty e1;
Empty * e2 = malloc(sizeof(Empty));
Empty * e4 = Empty_create(); // constructor!
static Empty e3;
```

[2] Později uvidíme, že implementace některých objektových vlastností si vyžádají jen a pouze ”dynamické” konstrukce.

Outline

Atributy

Atributy jsou reprezentovány pomocí položek struktury – proměnných, kterým se rovněž říká *ivars* / *properties*, a které v každé instanci nabývají konkrétních hodnot.
Často je zapisujeme s pomocí nějakého prefixu/postfixu, abychom je odlišili od ostatních proměnných:

```
typedef struct NonEmpty
{
  int _first;
  int mSecond; // _m_ember
  int third_; // Google-style
} NonEmpty;
```

Outline

Metody objektu

Metodou bude obyčejná céčková funkce, jejíž první argument bude odkazem na instanci (v OOP jazycích programátorovi skryto, ale dostupné přes speciální klíčové slovo `this`, `self`, ...).

Metody objektu

Ve struktuře

Metody můžeme psát jako funkční ukazatele přímo ve struktuře:

```
typedef struct Data
{
   int _value;
   int (* getValue)(Data * me);
   void (* setValue)(Data * me, int newValue);
}
static int getValue(Data * me) { return me->_value; }
static void setValue(Data * me) { me->_value = newValue;}
```

Výhody:

- Řeší problém absence jmenných prostorů – zamezuje konfliktům názvů metod.

Nevýhody:

- Vysoká spotřeba paměti.
- vystavení vnitřní struktury třídy v hlavičkovém souboru (anti-zapouzdření).

Metody objektu

Mimo strukturu

Lepší variantou je psaní metod odděleně od vnitřní struktury. Ovšem kvůli absenci jmenných prostorů je nutné používat jako prefix název třídy v názvu každé metody:

```
typedef struct Data
{
  int _value;
}
int Data_getValue(Data * me);
void Data_setValue(Data * me, int newValue);
```

Výhody:

- Metody neovlivňují velikost objektu.
- Lepší možnosti zapouzdření (viz dále).

Nevýhody:

- Manuální udržování jmenných prostorů.

Metody třídy

Metody třídy nemanipulují s atributy instancí, týkají se jen meta-instančních vlastností, určených pro třídu jako takovou, a tedy společných pro všechny její instance.

Example

nastavení úrovně debug výpisů, vyčtení verze třídy, ...

Díky tomu, že nemusíme přistupovat na atributy objektu, můžeme při psaní těchto metod vynechat odkaz na instanci:

```
static uint32 _dbgLevel = 0;
void Data_setDebugLevel(uint32 level)
{
  _dbgLevel = value;
}
```

Outline

Zapouzdření

Úvod

- Zapouzdření je způsob, jak realizovat onu ”černou skříňku”.
- Slouží zároveň k dosažení bezpečnějšího kódu.
- Konkrétní implementace OOP přinášejí spoustu úrovní zapouzdření: public, protected, protected internal, friend, private, ...
- V céčku lze s menším či větším úsilím (meta-jazykový přístup) dosáhnout pouze některých z nich.

Zapouzdření metod

Úrovně zapouzdření metod dosažitelných v C:

- Veřejná – uvedena v příslušném hlavičkovém souboru.
- Soukromá – uvedena pouze v implementačním souboru (nejlépe rovnou jako `static`).
- Přátelská/interní – přístupná přes speciální hlavičkový soubor (`DataPriv.h`) dostupný jen při kompilaci.
- Ostatní pouze meta-jazykovými přístupy – dohodou mezi programátory. :)

Zapouzdření atributů

Odlišení soukromých od veřejných atributů – umístění definice struktury do implementačního souboru:

```
// Data.h
typedef struct _Data Data;

// Data.c
struct _Data
{
  int _value; // private
};
```

Takto definovaná struktura se též nazývá *opaque type*.

Accesory a mutátory

- Někdy též nazývané gettery a settery.
- Přístupové metody pro *některé* z atributů objektu.
- Některé mohou být read-only – zveřejněn pouze accessor.
- Opatrně při výběru toho, co zveřejníme v hlavičkovém souboru.
  - Př. `Array` – pouze accessor pro počet prvků + indexový accessor a mutátor pro položky pole, vše ostatní (kapacita, samotný backing-store) by mělo zůstat skryto.

```
int Data_getValue(Data * me) // accessor
{
  return (me->_value);
}
void Data_setValue(Data * me, int newValue) // mutator
{
  me->_value = newValue;
}
```

Zapouzdření atributů

Vyhodnocení

Výhody:

- Přístupové metody jen pro některé atributy $\Rightarrow$ vyšší bezpečnost.
- Nedochází k narušení ABI (přidání/odebrání atributu, změna pořadí) $\Rightarrow$ nevyžaduje překompilování všech knihoven includujících hlavičkový soubor.
- Možnosti radikálních optimalizací skrytých uživateli a neohrožujících interface:
  - in-line alokace paměti,
  - dle potřeb se měnící backing-store – např. `Array` interně jako dequeue nebo dokonce (pro extra velké kapacity) jako tree.

Nevýhody:

- Kompilátor nezná velikost struktury $\Rightarrow$ nutnost dynamické alokace.
- Nemožnost přistupovat přímo na položky struktury (atributy).

Outline

Ukázka

Java

```
// Java:
public class Data {
  private int value; // private attribute

  public Data(int v) { this.value = v; } // constructor
  public int getValue() { return this.value; }
  public void setValue(int newValue) { this.value =
    newValue; }
  public static int getVersion() { return 0; } // class'
    method
};
public static int main(String[] args) {
  Data d = new Data(1);
  int x = d.getValue(); // x = 1
  d.setValue(2);
  int y = Data.getVersion();
}
```

Ukázka

C

```
// Data.h
typedef struct _Data Data;

// Data.c
struct _Data { int _value; };

Data * Data_create(int value) { /* later */ };
int Data_getValue(Data * me) { return me->_value; }
void Data_setValue(Data * me, int newValue) { me->_value
    = newValue; }
int Data_getVersion(void) { return 0; }

int main(void) {
  Data * d = Data_create(1);
  int x = Data_getValue(d); // x = 1
  int y;
  Data_setVaue(d, 2);
  y = Data_getVersion();
}
```

Outline

Životní cyklus objektu

- Každý objekt, konkrétně instance nějaké třídy, někdy vzniká a někdy zaniká.
- Objekt může být automaticky nahrán v době spuštění aplikace, může být uložen / vyčten do / ze souboru, nebo může být dynamicky vytvořen při běhu aplikace, tj. explicitně programátorem, programaticky.
- Objekt může přetrvat v paměti po celou dobu běhu aplikace, nebo může být opět dynamicky zrušen ještě před jejím ukončením.
- Pro vytvoření nové instance se používá *konstruktor*, pro zrušení instance se používá *destruktor*.
- Instancí třídy může být více, někdy je ale vhodné udělat z ní jedináčka (viz *singleton*).

Outline

Konstruktor a destruktor

Definice

K vytváření a rušení objektů se používají speciální metody, tzv. *konstruktory* a *destruktory*.

Konstruktor

Vytváří novou instanci. Může, ale nemusí alokovat paměť + inicializuje příslušné atributy.
$\Rightarrow$ Různé způsoby nastavení atributů = možnost více konstruktorů.

Destruktor

Ruší instanci – provede ”úklid” a příp. dealokuje jí zabrané místo v paměti.

Konstruktor a destruktor

Druhy konstruktorů

Defaultní konstruktor

Všechny atributy nastaví na výchozí hodnoty.

Úplný konstruktor

Nastaví všechny atributy na explicitně vyžádané hodnoty.

Mezi těmito dvěma póly může existovat celá kaskáda konstruktorů. Všechny neúplné konstruktory lze napsat tak, aby volaly úplný konstruktor – předají mu hodnoty atributů, které znají, a ty zbylé nastaví na výchozí hodnotu.

1-krokový konstruktor/destruktor

Třída nabízí metody `create` a `destroy`:

```
typedef struct Integer
{
  int _value;
} Integer;

// Default constructor
Integer * Integer_create(void)
{
  Integer * result = NULL;
  result = malloc(sizeof(Integer));
  if (result != NULL)
  {
    result->_value = 0; // 0 is a default value
  }
  return result;
}
```

1-krokový konstruktor/destruktor

```
// Destructor
void Integer_destroy(Integer * me)
{
   if (me != NULL)
   {
      me->_value = 0; // Not needed
      free(me);
   }
}
```

1-krokový konstruktor/destruktor

Varianta s úplným konstruktorem:

```
Integer * Integer_createWithValue(int v)
{
  Integer * result = NULL;
  result = malloc(sizeof(Integer));
  if (result != NULL)
  {
    result->_value = v;
  }
  return result;
}

// Default constructor
Integer * Integer_create(void)
{
  return Integer_createWithValue(0);
}
```

2-krokový konstruktor/destruktor

Třída poskytuje dvě metody jak pro konstrukci, tak pro destrukci – `allocate + init` a `deallocate + cleanup`:

```
void Integer_init(Integer * me, int v)
{
  if (me != NULL)
  {
    me->_value = v;
  }
}

Integer * Integer_allocate(void)
{
  Integer * result = malloc(sizeof(Integer));
  return result;
}
```

To samé, co platilo o kaskádovitosti `create` konstruktoru, platí zde pro metodu `init`.

2-krokový konstruktor/destruktor

```
void Integer_cleanup(Integer * me)
{
  if (me != NULL)
  {
    me->_value = 0;
  }
}

void Integer_deallocate(Integer * me)
{
  if (me != NULL)
  {
    free(me);
  }
}
```

2-krokový konstruktor/destruktor

2-krokový konstruktor/destruktor má tu výhodu, že pro alokaci instance může být využit prakticky jakýkoliv druh paměti.
(Ale pozor – jak již bylo řečeno – tento způsob vyžaduje, aby definice struktury byla uvedena již v hlavičkovém souboru!)

Example

V následující ukázce kódu bude `Zero` alokován v paměti programu, viditelný globálně, kdežto `One` bude alokován na stacku.

2-krokový konstruktor/destruktor

Příklad

```
// Integer.h
extern Integer Zero;

// Integer.c
Integer Zero;

void initializeZero(void)
{
  Integer_init(&Zero, 0);
}
void someFunction(void)
{
  Integer One;
  Integer_init(&One, 1);
  ...
  Integer_cleanup(&One);
}
```

Outline

Alokace na požádání

Problémy

Vytvoření zpravidla není problém.
Zrušení ovšem ano – objekt se musí dealokovat pouze a právě jednou!

- Nezavolání destruktoru $\Rightarrow$ memory leak.
- Zavolání destruktoru více jak jednou $\Rightarrow$ výjimka / error / pád aplikace / (nejhorší varianta) nepředvídatelný běh aplikace.

Nasdílení / přeposlání objektu vytváří otázku, kdo a kdy objekt zruší:

- Ten, kdo jej vytvořil – může ”umřít” dříve než jím vytvořený objekt!
- Poslední, kdo jej využívá. Jak to ale v asynchronním / multivláknovém prostředí zjistit? – obecně nesnadné (často nemožné, příp. velmi svazující) programaticky určit, kdo a kdy daný konkrétní objekt zruší.

GarbageCollector

- Runtime sám udržuje informace o alokovaných objektech.
- Sledují se všechny odkazy na daný objekt.
- Objekt, na nějž nejsou žádné odkazy, je následně runtimem dealokován.

Výhody:

- Vše automatizované.

Nevýhody:

- V čistém C dosti problematické.
- Nesnadné ladění – objekty se ruší ”někdy v budoucnu” (až se spustí uvolňovací proces GC).
- GC správce obvykle běží v samotném vlákně, užírá CPU i paměť samotné aplikaci.

Poloautomatický GC

- Založen na čítači referencí, nazvěme jej např. `refCount`.
- Po vytvoření je `refCount == 1`.
- Předání / nasdílení objektu $\Rightarrow$ příjemce si jej přivlastní zavoláním metody `retain`, jež zvýší čítač o 1.
- Jakmile stvořitel / příjemce nadále nechce s objektem pracovat, uvolní jej – zavolá metodu `release`, jež sníží čítač o 1.
- Až se čítač dostane na 0, automaticky se zavolá destruktor.

Pozor – v multivláknovém prostředí musí být změny `refCountu` atomické!

Poloautomatický GC

Nadefinujeme základní strukturu (třídu):

```
typedef struct BaseClass
{
  uint32 _refCount;
} BaseClass;
```

A vložíme ji jako *první* položku ve všech ostatních třídách:

```
typedef struct Data
{
  BaseClass _base;
  int _value;
} Data;
```

Poloautomatický GC

```
void retain(void * obj)
{
  BaseClass * bc = (BaseClass *) obj;
  atomic_increment(&(bc->_refCount));
}
void release(void * obj)
{
  BaseClass * bc = (BaseClass *) obj;
  int32 rc = atomic_decrement(&(bc->_refCount));
  if (rc == 0)
  {
    // Call destructor ... POLYMORPHIC!!
  }
}
```

Singleton

Definice

Singleton (pattern)

Jedna jediná možná instance třídy. Konstruktor vrací vždy jednu a tu samou instanci.

- V embedded světě velmi často využíván jako reprezentant těch vlastností daného systému, jejichž multiplicita je rovna 1.
- Po alokaci již většinou zůstává v paměti až do ukončení aplikace.
- Obecně nedostačující – objekty typu String, ByteBuffer, kontejnery používáme v mnoha instancích.

Example

V aplikaci zabývající se řízením auta budou jako singleton např. objekty *volant*, *motor*, *rychlost*.
V (křesťanské) aplikaci o rodině budou jako singleton objekty *manželka*, *manžel*.

Singleton

Ukázka

```
// Universe.c
typedef struct Universe
{
  TEverything _content;
} Universe;

static Universe * instance = NULL;

Universe * Universe_create(void)
{
  if (instance == NULL)
  {
    instance = malloc(sizeof(Universe));
    // init the instance
  }
  return instance;
}
```

Pozor – v multivláknovém prostředí musí být synchronizováno.

Object pool

Object pool (pattern)

Množina předem vytvořených objektů (instancí nějaké třídy) připravených k použití.

- Objekty jsou vytvářeny zpravidla pri initu aplikace, rušeny při jejím ukončení.
- Konstruktor = metoda pro výběr objektu z poolu. Destruktor = metoda pro vrácení objektu do poolu.
- Problém ”poslední ruší” nicméně *neřeší*, pouze převádí na jiný – kdo vrací objekt do poolu?

$\Rightarrow$ Dynamická alokace na požádání stále nejflexibilnější!

Memory pool

Nemáme-li k dispozici dynamickou paměť, příp. je její použití považováno za nebezpečný prvek (fragmentace paměti), lze využít například kombinaci *Static Allocation* a *Fixed Size Allocation* návrhových vzorů:

- Staticky alokovaná paměť rozdělena do několika poolů, každý tvořen několika bloky paměti jisté velikosti.
- Algoritmus přidělování paměti může být jednoduchý First-Fit -– hledá se nejmenší volný blok splňující požadovanou velikost.
- Funkce `malloc` a `free` jsou nahrazeny příslušnými metodami alokátoru pro zabrání a uvolnění bloku statické paměti.

Memory pool

Výhody:

- Umožňuje programátorovi psát kód, jako kdyby měl k dispozici dynamickou paměť.
- Nízká fragmentace paměti.
- Obvykle jednodušší, a tedy i rychlejší než komplexní správa dynamické paměti.

Nevýhody:

- ”Plýtvání” pamětí, neb rozdílné požadavky se musí namapovat na bloky fixní délky.
- Nutnost statické analýzy, kolik paměti a v jakém rozložení je pro danou aplikaci potřeba.

Outline

Měnitelné a neměnitelné objekty

Neměnitelný objekt

Objekt, jehož hodnoty atributů zůstanou po vytvoření nedotčené.

Výhody:

- Nižší spotřeba paměti – už při konstrukci známé kompletní vnitřní uspořádání $\Rightarrow$ je možné využít tzv. *inline alokace*.
- Lepší výkonnost
  - Vychází mimo jiné z předchozího bodu, neb neměnitelné objekty jsou šetrnější vůči cache.
  - Poměrně často používaná metoda `clone/copy` je zde extrémně jednoduchá – převede se na volání `retain`.
- Bezpečné vůči vláknům – není třeba synchronizace!
- Snadnější implementace.

Měnitelné a neměnitelné objekty

$\Rightarrow$ V praxi by se mohlo a mělo využívat neměnitelných objektů v mnohem větší míře, než se obvykle užívají!

Example

Prakticky každá ”value” třída a každý kontejner by měl mít svého neměnitelného sourozence:
`String`, `ByteBuffer`, `Number`, ale i `Array`, `Set`, `Dictionary`, ...

Inline alokace

Inline alokaci je více než vhodné použít v případech konstruktorů neměnitelných objektů obsahujících buffery nebo jiné objekty (o předem známé délce/velikosti)!

Example

```
String_createWithChars(char * str, size_t len);
ByteBuffer_createWithBytes(uint8 * bytes, size_t len);
Array_createWithCapacity(size_t capacity);
Set_createWithCapacity(size_t capacity);
...
```

Inline alokace

```
typedef struct String
{
  size_t _len;
  char * _str;
} String;
String * String_createWithChars(char * str, size_t len)
{
  String * res = NULL;
  res = malloc(sizeof(String) + len * sizeof(char));
  if (res != NULL)
  {
    res->_len = len;
    res->_str = (char *) res + sizeof(String); // !!!
  }
  return res;
}
```

Outline

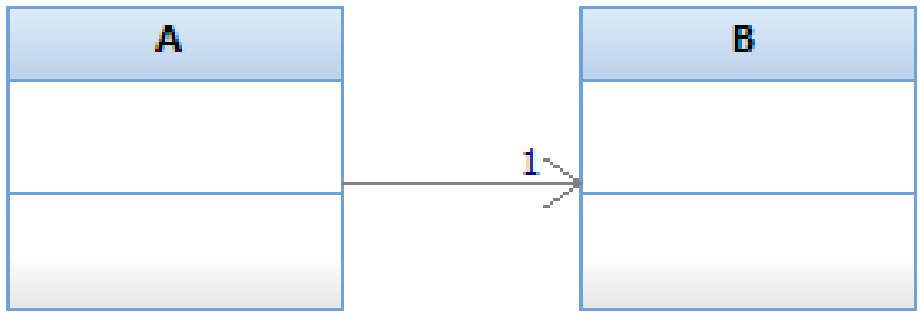
Asociace

Asociace

Vztah mezi dvěma a více instancemi.
Objekt A chce poslat zprávu objektu B ⇒ objekt A má asociaci na objekt B ⇒ objekt A obsahuje odkaz na objekt B.

Asociace může být jednosměrná nebo obousměrná. Má nějakou multiplicitu.

```
struct A
{
  B * _itsB;
};
```

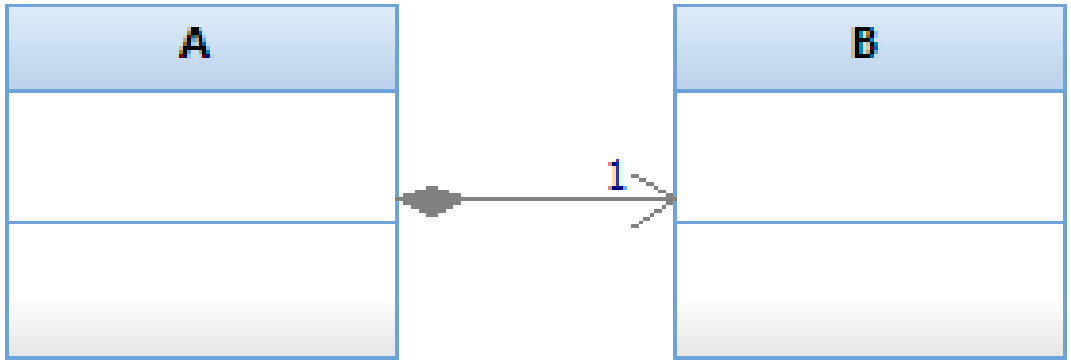


Outline

Kompozice

Kompozice

Vztah typu ”owns a”.
Objekt A je *výhradním* vlastníkem objektu B a nemůže bez něj existovat.
$\Rightarrow$ A je tudíž zodpovědný za vytvoření i zrušení B.



Example

Člověk – mozek

Kompozice

Konstruktor

```
A * A_create(void)
{
  A * result = NULL;
  result = malloc(sizeof(A));
  if (result != NULL)
  {
    result->_itsB = B_create(); // create my B
    if (result->_itsB == NULL)
    {
      A_destroy(result);
      result = NULL; // A cannot exist without B!
    }
  }
  return result;
}
```

Kompozice

Destruktor

```
void A_destroy(A * a)
{
  if (a != NULL)
  {
    if (a->_itsB != NULL)
    {
      B_destroy(a->_itsB);
    }
    free(a);
  }
}
```

Kompozice 2

V případě, kdy známe vnitřní strukturu třídy B (a můžeme tedy použít i 2-krokový konstruktor/destruktor), lze objekt B vnořit do A tzv. ”inline”:

```
struct A
{
  B _itsB;
};
```

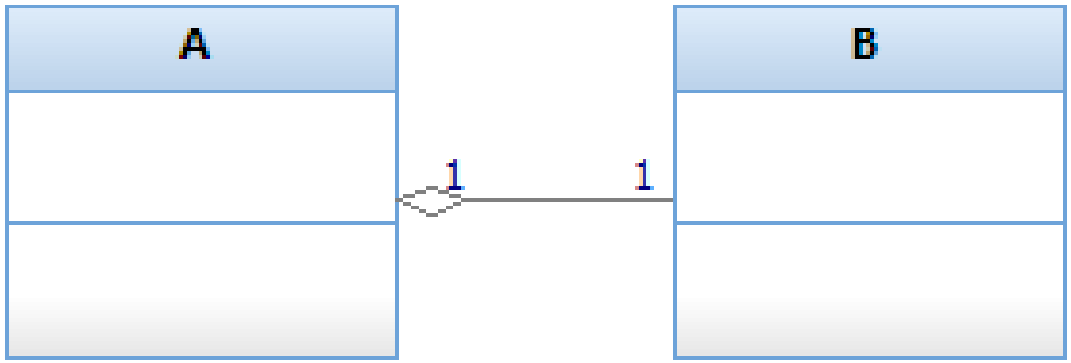
Outline

Agregace

Agregace

Vztah typu ”has a”.
A není výhradním vlastníkem B, může jej sdílet s jinými objekty. Sám se bez B obejde.

Někdy se též nazývá *sdílená asociace*. ⇒ Kdo vytváří/ruší objekt B?



Example

PC – tiskárna

Agregace

Často lze využít způsob ala *Builder* pattern:

```
typedef struct A
{
  B * _itsB;
} A;
typedef struct B
{
  A * _itsA;
} B;

// Called e.g. when application started.
void InitializeABPackage(void)
{
  A * objA = A_create();
  B * objB = B_create();
  A_setItsB(objA, objB); // setter for itsB
  B_setItsA(objB, objA); // setter for itsA
}
```

Agregace

V kombinaci s poloautomatickým GC, je možno využít již známého principu `retain/release` a kód doplnit:

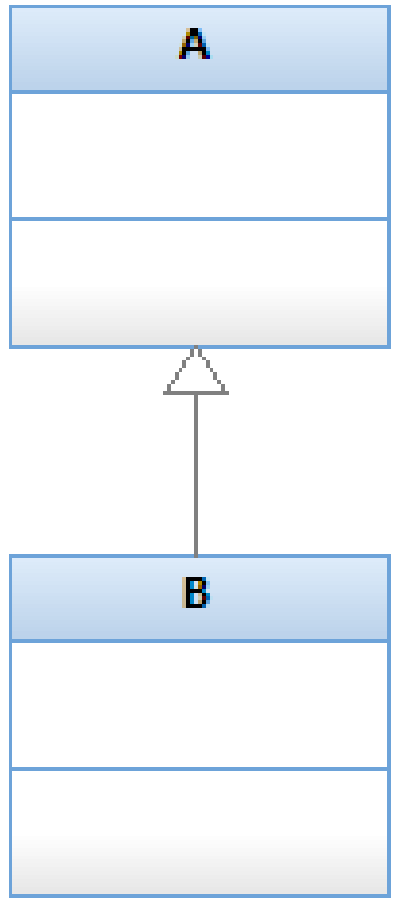
```
void A_setItsB(A * a, B * b)
{
  if (a->_itsB != NULL)
  {
    release(a->_itsB);
  }
  a->_itsB = retain(b);
} // ... same for B_setItsA()

void InitializeABPackage(void)
{
  A * objA = A_create();
  B * objB = B_create();
  A_setItsB(objA, objB);
  B_setItsA(objB, objA);
  release(objA);
  release(objB);
}
```

Outline

Dědičnost

Dědičnost

Vztah typu ”is a”.
B je *specializací* A, A je *generalizací* B. S každou instancí B můžeme zacházet zároveň jako s instancí A (naopak neplatí!).



Example

Auto (B) je zároveň dopravním prostředkem (A).
V praxi často vznikají problémy: kružnice (B) vs. elipsa (A), reálné (B) vs. komplexní číslo (A), ...

Pozor – dědičnost narušuje zapouzdření!

Implementace v C

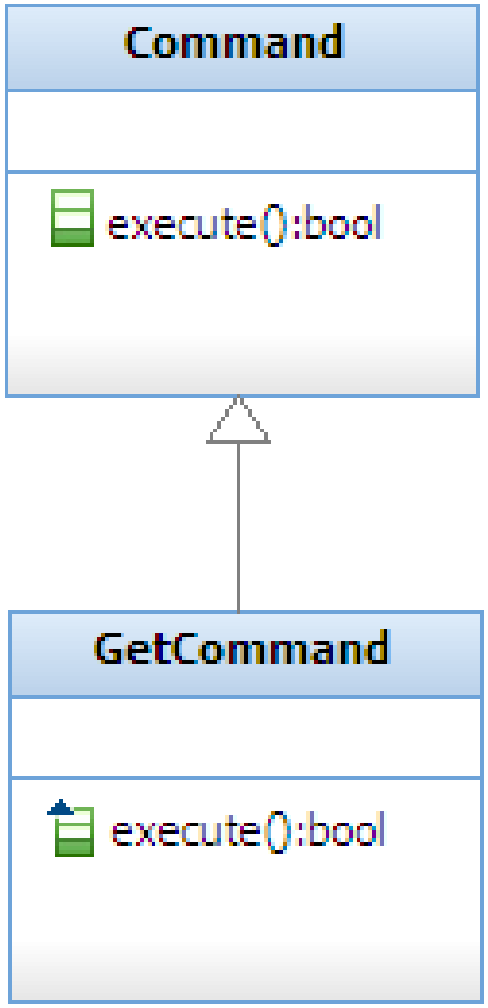
Postupně představím 3 způsoby implementace dědičnosti v C:

1. Naivní
   - Jednoduchá, přímočará implementace, avšak se spoustou nevýhod.
2. Sofistikovaný
   - Komplexní varianta dědičnosti ala C++, složitá na údržbu.
3. Kompromisní
   - Mírně omezená forma dědičnosti ala Java (bez interfaců), jednoduchá a flexibilní.

Outline

Naivní přístup

Mějme následující hierarchii:



Implementace

- Nadefinujeme si předka (s jednou jedinou virtuální metodou `execute`):

```
typedef struct Command
{
  void * _child; // pointer to my child
  bool (* _fpExecute)(void *);
} Command;
```

- Klíč k dědičnosti – kompozice! Nadefinujme potomky:

```
typedef struct GetCommand
{
  Command _super;
} GetCommand;
bool GetCommand_execute(void *);
```

Implementace

- V konstruktoru potomka musíme nastavit atributy předka (přes přístupové metody vhodnější a v případě, že `_super` je *pointer* na předka, dokonce nutné):

```
GetCommand * GetCommand_create(void)
{
  GetCommand * result = malloc(sizeof(GetCommand));
  if (result != NULL)
  {
    result->_super._child = result;
    result->_super._fpExecute = GetCommand_execute;
  }
  return result;
}

bool GetCommand_execute(void * me) { ... }
```

Implementace

- A konečně v předkovi napíšeme funkci `execute` následovně:

```
bool Command_execute(void * me)
{
  bool result = false;
  if ((me != NULL) &&
      (me->_child != NULL) &&
      (me->_fpExecute != NULL))
  {
    result = me->_fpExecute(me->_child);
  }
  return result;
}
```

Naivní přístup

Vyhodnocení

Výhody:

- Relativně jednoduché na implementaci.

Nevýhody:

- Virtuální metody jako funkční pointery $\Rightarrow$ vyšší spotřeba paměti (závislá od počtu instancí).
- Neflexibilní.

Outline

Sofistikovaný přístup

Motivace

Motivace pro následující koncept dědičnosti:

- Plnohodnotná vícenásobná dědičnost ala C++.
- Nízká spotřeba paměti.
- Stále relativně rychlé v runtimu.

Implementace

Každý objekt si udržuje ukazatel na ”třídní informace”:

```
typedef struct Command
{
  CommandClass * _isa;
}
```

Ta může vypadat např. takto:

```
typedef struct CommandClass
{
  CommandVtbl _vtbl;
  // any additional info e.g. wrapped into "MetaClass"
  // structure can follow (class' name, version, ...)
}
```

Implementace

Kde `CommandVtbl` je tabulka virtuálních metod dané třídy:

```
typedef struct VtblEntry
{
  void * _op;
  int _offset;
} VtblEntry;
typedef struct CommandVtbl
{
  VtblEntry _execute;
} CommandVtbl;
```

Implementace

Inicializace

Každý objekt je nutno řádně inicializovat:

```
static CommandClass isa =
{
  { // CommandVtbl
    { Command_execute_job, 0 }
  }
};
void Command_init(Command * me)
{
  Command_setItsClass(me, &isa);
}
void Command_setItsClass(Command * me, CommandClass *
    cls)
{
  me->_isa = cls;
}
```

Implementace

Meta info a VTBL

Pokud by instance třídy `Command` vůbec metodu `execute` neimplementovaly (tj. byla-li by čistě virtuální), pak by virtuální tabulka vypadala následovně:

```
static CommandClass isa =
{
   { // CommandVtbl
     { NULL, 0 }
   }
};
```

Implementace

Virtuální metoda

A konečně implementace virtuální metody bude vypadat takto[3]):

```
bool Command_execute(void * me)
{
  bool result = false;
  Command * _me = (Command *) me;
  CommandVtbl * vtbl = me->_vtbl;
  if (vtbl->_execute->_op != NULL)
  {
    void * meaddr = (void *)
      ((char *) me - vtbl->_execute->_offset);
    result = (* vtbl->_execute->_op)(meaddr);
  }
  return result;
}
```

[3] Většinu z toho lze úspěšně ”zamakrovat”.

Implementace

Poznámky

- Metody s příponou `_job` obsahují samotnou implementaci. Slouží i k přímému vyvolání metody předka (`super.execute()`).
- Offset je udržován pro každou metodu. To umožňuje podědit v potomkovi metodu kteréhokoliv předka v řetězci dědičnosti bez nutnosti ji přepisovat (psát pro ni její `_job` implementaci). Alternativně lze udržovat offset pro celou virtuální tabulku – pak je ale nutno všechny metody předka buď přepsat, nebo podědit, nic mezi.
- Definici struktury virtuální tabulky je vhodné mít jako `#define`, a při dědění pak v potomkovi definujeme jeho virtuální tabulku tak, že vložíme tento define předka a doplníme definem potomka. Velmi to usnadňuje makra při volání virtuálních metod. (Zde to vynecháme.)

Implementace

Potomek

Naimplementujme potomka:

```
typedef struct GetCommand
{
  GetCommandClass * _isa;
  Command _super; // parent
} GetCommand;

typedef struct GetCommandClass
{
  CommandClass _superClass;
  GetCommandVtbl _vtbl;
} GetCommandClass;

typedef struct GetCommandVtbl
{
  VtblEntry _execute; // parent's method
} GetCommandVtbl;
```

Implementace

Potomek

A zinicializujeme:

```
static GetCommandClass isa =
{
  { // CommandClass
    {
      { GetCommand_execute_job, offsetof(GetCommand,
   _super) }
    }
  },
  { // GetCommandVtbl
    { GetCommand_execute_job, 0 }
  }
};
void GetCommand_init(GetCommand * me)
{
  Command_init(&me->_super);
  GetCommand_setItsClass(me, &isa);
}
```

Implementace

Potomek

Nesmíme zapomenout přepsat třídní informace i svému předkovi!

```
void GetCommand_setItsClass(GetCommand * me,
    GetCommandClass * cls)
{
  me->_isa = cls;
  // We need to overwrite parent's class info:
  Command_setItsClass(&me->_super, &cls->_super);
}
```

Implementace

Potomek

Třída potomka a zejména jeho virtuální tabulka bude vypadat dle toho, zda chceme tu kterou metodu přepsat nebo jen podědit. Všimněte si, které metody se v tom kterém případě pro tu kterou virtuální tabulku (předka/potomka) volají a zejména s jakým offsetem!

1. Override – viz předchozí slide
2. Inherit

```
static GetCommandClass isa =
{
   { // CommandClass
     { Command_execute_job, 0 }
   },
   { // GetCommandVtbl
     { Command_execute_job, -offsetof(GetCommand,
    _super) }
   }
};
```

Implementace

Up- & Down-Cast

Upcast

Danou instanci chceme přetypovat na některého z předků.

Downcast

Danou instanci chceme přetypovat na potomka. (Je možné jen tehdy, jde-li skutečně o instanci třídy potomka či jeho potomků!)

Ani v jednom případě nelze použít obyčejné céčkové přetypování!
Proč?

Implementace

Up- & Down-Cast

- Zavedeme metodu `cast`, kterou každá třída *povinně*[4] přepíše.
- Dle jistého porovnávacího mechanismu nejdřív zkontroluje, zda je shodná s požadavanou třídou. Pokud ano, vrátí `me` objekt, s nímž byla volána, v opačném případě volá předkovu metodu a jako `me` předává pointer na předka.
- Porovnávací mechanismus může být různý – porovnání adresy proměnné obsahující meta-třídní informace, porovnání názvů tříd uložených v metatřídě (pomalé), ...

[4] Při zavolání této metody se tedy přejde na její implementaci ve třídě, jejíž instancí skutečně daný objekt je, a postupně se průchodem po předcích hledá ta správná třída.

Implementace

Up- & Down-Cast

```
// GetCommand.h
char GetCommandClassName[] = "GetCommand";
// GetCommand.c
void * GetCommand_cast_job(void * me, char * className)
{
  return (!strcmp(GetCommandClassName, className))
    ? me : Command_cast_job(&me->_super, className);
}

// Command.h
char CommandClassName[] = "Command";
//Command.c
void * Command_cast_job(void * me, char * className)
{
  return (!strcmp(CommandClassName, className))
    ? me : NULL;
}
```

Implementace

Up- & Down-Cast

Příklad volání:

```
GetCommand * A = GetCommand_create();
Command * B = Command_create();
Command * C = Command_cast(A, CommandClassName); /* OK */
GetCommand * D = Command_cast(B, GetCommandClassName);
    /* NOK */

Command_execute(A); /* FAILURE! */
Command_execute(Command_cast(A, CommandClassName)); /*
    OK */
Command_execute(&A->_super); /* OK, easy up-cast */
```

Sofistikovaný přístup

Vyhodnocení

Výhody:

- Plnohodnotná dědičnost s možností vícenásobné dědičnosti – několik `_super` předků a pro každého virtuální tabulku.
- Možnost některé metody přepsat, některé podědit.
- Podpora up- i down-castu. (ala `<dynamic_cast>` v C++)
- Relativně nízká spotřeba paměti.

Nevýhody:

- Nutnost explicitního castování.
- Občas nesnadné ladění (ta samá instance, jen jinak ”nacastovaná”, má rozdílnou adresu).
- U složitějších objektových modelů (vícenásobná dědičnost, delší řetěz předků) se stává poměrně složitá na údržbu. (Můžete si sami zkusit za DÚ.)

Outline

Kompromisní řešení

Upustí-li se od možnosti vícenásobné dědičnosti, lze implementovat poměrně jednoduchý koncept dědičnosti / polymorfních funkcí, jež oproti předchozímu řešení nabízí:

- Výrazně jednodušší vtbl bez offsetů.
- Up-/Down-cast céčkovým přetypováním.

Avšak! Může být problém u některých code-generating IDE.

Implementace

Bázová třída

Ukážeme si na jiné hierarchii. Je výhodné nadefinovat si základní třídu s několika důležitými metodami, aplikovatelnými na všechny objekty.

```
typedef struct Object
{
  void * _isa;
  uint32 _info; // can serve e.g. as a refCount
} Object;
#define OBJECT_CLASS \
  void * (* retain)(void * me); \
  void (* release)(void * me); \
  uint32 (* getRefCount)(void * me); \
  bool (* equal)(void * me, void * to); \
  uint32 (* getHashCode)(void * me); \
  String * (* getDescription)(void * me); //
typedef struct ObjectClass
{
  OBJECT_CLASS;
} ObjectClass;
```

Implementace

Bázová třída Object

Jako příklad uvedeme metodu `equal`:

```
bool Object_equal(void * me, void * to)
{
  Object * _me = (Object *) me;
  ObjectClass * isa = (ObjectClass *) _me->_isa;
  bool (* equal)(void *, void *);
  equal = (isa->equal != NULL) ? isa->equal :
    Object_equal_job;
  return equal(me, to);
}

bool Object_equal_job(void * me, void * to)
{
  return (me == to); // simple address comparison
}
```

Implementace

Potomek Command

```
typdef struct Command
{
  Object _super; // must be at first place in struct!!
  // other attributes
} Command;

#define COMMAND_CLASS \
  bool (* execute)(void *); //

typedef struct CommandClass
{
  OBJECT_CLASS;
  COMMAND_CLASS;
} CommandClass;
```

Implementace

Potomek Command

Jak na instanci `Command` zavolat metodu equal? Snadno!

```
Command * cmd1 = Command_create();
Command * cmd2 = Command_create();
Object_equal(cmd1, cmd2);
```

Jak vidno:

- ”Zero cost” up-/down-cast – uvědomte si, jak vypadá paměť zabraná instancí třídy `Command`. Díky tomu, že struktura předka je vždy na prvním místě potomka, při volání polymorfních funkcí není vůbec třeba objekty castovat, a uvnitř polymorfní funkce stačí pouhé céčkové přetypování (viz `equal`).
- Není nutné a dokonce ani žádoucí, aby metoda `equal` byla definována i pro `Command`. Vždy se volá metoda s prefixem té třídy, která ji definovala.

Kompromisní řešení

Vyhodnocení

Výhody:

- Velmi jednoduché řešení, jak na implementaci, tak na údržbu i testování (objekt má stále stejnou adresu, ať je nacastovaný na libovolného předka).
- Velmi malý overhead, CPU i paměti $\Rightarrow$ vhodné i pro embedded aplikace.
- Zero cost up-/down/cast.

Nevýhody:

- Předek musí být ve struktuře potomka vždy na prvním místě. – Některá IDE s tím nemusí být v souladu!
- Nemožnost vícenásobné dědičnosti, a to ani ve formě (bezatributových) interfaců ala Java. – Nutno zavést dynamické vtbl.

Námitky

Výkonnost

Je to pomalé!

- Volání polymorfních metod
  - = Jeden skok přes funkční pointer.
  - V praxi zpravidla nepředstavuje žádný zásadní problém.
  - Vembedded aplikacích mnohem důležitější stabilita výkonnosti, než její absolutní hodnota (tj. pokud možno žádné velké výkyvy pro libovolný vstup).
- Volání accessorů a mutátorů
  - = Volání funkce namísto okamžitého přístupu k proměnné.
  - Vykoupeno všemožnými optimalizacemi skrytými uživateli za univerzální API – bitová pole VS. několikero položek zvlášť, inline alokované atributy, ...

Námitky

Výkonnost

- ”S kanónem na vrabce”
  Neboli – objekty jsou zbytečně velké a komplikované, standardní céčkové služby jsou skvěle optimalizované.
  - V mnoha případech pravý opak pravdou!
  - `char *` VS. `String`
    dtto pro `uchar *` VS. `ByteBuffer`
  - `[]` VS. `Array`

⇒ **Není!**

Námitky

Chyby a testování

Náchylné k chybám a špatně se testuje!

- Co když zavolám metodu s jiným objektem? (`void * me`)
  - Stává se, ale zejména v případě vícenásobné dědičnosti a opomenutí castu.
  - V ostatních případech se snadno odladí pomocí polymorfní metody `getDescription`.
- Co když zavolám metodu s ”ne-objektem”? (`void * me`)
  - V praxi téměř vůbec nenastává.
- Zapouzdření znesnadňuje debugování.
  - Nahrazeno textovým popisem z metody `getDescription`.
- Dynamická alokace objektů je potenciálně nebezpečná.
  - Poloautomatický GC významně snižuje procento chyb. Metoda `getRefCount` pak výrazně usnadňuje ladění.
  - Není-li vůbec k dispozici, lze využít některého z Memory Allocation patternu.

Shrnutí

- Objektové programování v C *je možné*.
- Objektové programování v C *je relativně snadné*.
- Objektové programování v C *je výhodné*.

Další témata

Bude-li zájem, je možné pokračovat například v následujících tématech:

- Abstraktní třídy, clustery tříd.
- Alternativy pro dědění.
- Objektový návrh aplikace.
- Návrhové vzory (s příklady v C).

Otázky

Otázky?