Učební texty k státní bakalářské zkoušce
Obecná informatika
Programovací jazyky

8. června 2011

6 Programovací jazyky

Požadavky

- Principy implementace procedurálních programovacích jazyků, oddělený překlad, sestavení.
- Objektově orientované programování.
- Neprocedurální programování, logické programování.
- Generické programování.

6.1 Principy implementace procedurálních a objektové orientovaných programovacích jazyků, oddělený překlad, sestavení

Principy implementace proceduralnich programovacich jazyku je v zasade jednoducha vec. Je treba vedet, jak funguji zakladni veci uvnitr. Ze program ma ctyri kusy pameti (code, data, stack a heap) a co ktera dela. Jak probiha volani procedur, kam se pri nem co schova. Jak funguje rekurze, kam se ukladaji lokalni promenne a kam globalni, ... atd.

Strukturované programování

Počítačový program je nějakým způsobem zaznamenaný postup počítačových operací, který speciálním způsobem popisuje praktickou realizaci zadané úlohy (tedy algoritmus výpočtu). Program z *procedurálního* úhlu pohledu je vlastně přesná specifikace všech kroků, které musí počítač vykonat, aby došel k cíli, a jejich pořadí. Pro určování pořadí kroků se používají základní operace *řízení toku* – skoky, podmínky, cykly apod.

Jedním z důležitých konceptů procedurálního programování je *strukturované programování* – jeho idea je založena na rodělení programu na *procedury* (rutiny, podrutiny, metody, funkce), které samy obsahují výčet výpočetních kroků k vykonání, mohou být ale spouštěny opakovaně a z libovolného místa v programu. Jejich výhodou je mnohem názornější pohled na strukturu programu a snazší udržování kódu, než v případě použití jen nejjednoduššího řízení toku (tedy hlavně skoků, které by se ve strukturovaném programování správně používat neměly).

Datové a řídící struktury vyšších programovacích jazyků a jejich implementace

Řízení toku

V informatice se *tokem řízení* rozumí pořadí ve kterém se provádějí jednotlivé příkazy programu. V imperativních (=procedurálních) programovacích jazycích se *příkazy pro řízení toku* rozumí příkazy, které dokážou změnit pořadí provádění příkazů na jiné než přirozené (to v jakém jsou zapsány). Některé funkcionální jazyky také obsahují takové příkazy (které již nepatří do funkcionálního programování), ale potom už se jim většinou neříká příkazy pro řízení toku.

Druhy příkazů pro řízení toku se mezi jazyky liší, ale přibližně mohou být rozděleny do následujících kategorií.

- Pokračování na jiném místě programu (skok).
- Vykonání skupiny příkazů pouze pokud je splněna určitá podmínka (if-then-else)
- Nevykonání skupiny příkazů nebo jejich opakování do té doby než je splněna určitá podmínka (smyčka/cyklus). Cykly mohou být s podmínkou na začátku, na konci, uprostřed, nekonečné, s daným počtem opakování.
- Vykonání skupiny příkazů, které se nacházejí na jiném místě kódů, a následné (volitelné) vrácení řízení toku zpět (subroutines, coroutines, and continuations).
- Zastavení/ukončení programu.

Přerušení a signály jsou nízkoúrovňový mechanizmus, který může změnit tok řízení podobně jako se to dělá u podprogramů, ale většinou je vyvolán vnější událostí a není tedy použit ve smyslu řízení toku programu, jak bylo popsáno.

Výjimky

Výjimky jsou speciálním příkazem řízení toku, vyskytujícím se v některých vyšších programovacích jazycích. Základní myšlenkou je, že program může na nějakém místě vyhodit výjimku (příkaz `throw`), což způsobí, že provádění programu se zastaví a buď pokračuje tam, kde je výjimka ošetřena (tzv. `catch` blok), nebo pokud takové místo není nalezeno, program skončí s chybou. Během hledání místa ošetření je datová hodnota výjimky uložena stranou a pak může být použita.

Při hledání místa ošetření výjimky (`try`-bloku, následovaného catch-blokem se správným datovým typem výjimky) se postupuje zpět po zásobníku volání funkcí, tato technika se nazývá stack unwinding (odvíjení zásobníku). V některých jazycích (Java) lze definovat i akci, která se provede v každém případě, i pokud nastane výjimka, ještě před odvíjením zásobníku – `finally` blok.

Volací konvence

Při volání procedur a funkcí je nejdůležitější zásobník. Ukládá se na něj

- kam se vrátit po volání
- argumenty funkce (v překladem definovaném pořadí – nutné mít ve všech modulech stejné; většinou se liší v závislosti na programovacím jazyku)
- návratová hodnota funkce
- ukazatel na sémanticky nadřazenou funkci (Pascal)

Dohromady všem těmto datům se někdy říká *aktivační záznam* procedury. Po skončení funkce je nutné zásobník opět uklidit (vymazat zbytečná uložená data, většinou jen zůstává návratová hodnota) a která část programu to dělá (volaná nebo volající procedura), závisí opět na překladači a konvenci jazyka.

Volací konvence dvou nejtypičtějších jazyků:

- *Pascal*
  uklízí volaná funkce, argumenty se ukládají na zásobník zleva doprava (nejlevejší nejdřív, tj. nejhlouběji)
- *C*
  uklízí funkce volající, argumenty se ukládají zprava doleva (tj. nejlevější je na vrcholu zásobníku. Je to kvůli funkcím s proměnným počtem parametrů. Volaná funkce musí podle prvního argumentu poznat, jaký je skutečný počet argumentů. Kdyby byl první argument někde hluboko v zásobníku, tak ví prd.)

Organizace paměti

Paměť procesu (spuštěného programu) lze rozdělit do několika částí:

- *kód programu (kódový segment)*
  vytvořen při překladu, součást spustitelného souboru, neměnný a má pevnou délku; obvykle bývá chráněn proti zápisu
- *statická data (datový segment)*
  data programu, jejichž velikost je známa již při překladu a jejichž pozice se během programu nemění (je připraven kompilátorem a jeho formát je taktéž zadrátovaný ve spustitelném souboru, u inicializovaných statických dat je tam celý uložený); v jazyce C jde o globální proměnné a lokální data deklarovaná jako `static`, konstanty

- *halda (heap segment)*
  vytvářen startovacím modulem (C Runtime library), ukládají se sem dynamicky vznikající objekty (`malloc, new`) – neinicializovaná data, i seznam volného místa
- *volná paměť*
  postupně jí zaplňuje z jedné strany zásobník a z druhé halda
- *zásobník (stack segment)*
  informace o volání procedur (aktivační záznamy) — návratové adresy, parametry a návratové hodnoty (nejsou-li předávány v registrech), některé jazyky (Pascal, C) používají i pro úschovu lokálních dat. Typicky roste zásobník proti haldě (od konce paměti k nižším adresám).

Poznámka *(Vnořené funkce)*
V Pascalu mohou být funkce definované uvnitř jiné funkce. Ta vnitřní potřebuje přistupovat k proměnným té vnější. Proměnné jsou sice na zásobníku, ale pouhý odkaz na volající funkci nestačí, protože se vnořená funkce může volat rekurzivně. Proto je na zásobníku ukazatel na funkci sémanticky nadřazenou.

Alokace místa pro různé typy proměnných

- Dynamicky alokované proměnné (přes pointer) se alokují na haldě. Opakovanou alokací a dealokací paměťových bloků různé velikosti vznikají v haldě díry (střídavé úseky volného a naalokovaného místa). Existuje několik strategií pro vyhledání volného bloku požadované velikosti (first-fit, next-fit, buddy systém) a udržení informací o volném místě, které jsou většinou implementovány v knihovních funkcích jazyka (C, Pascal).
- Lokální proměnné se ukládají na zásobník, po skončení funkce, které přísluší, jsou zase odstraněny.
- Globální a statické se ukládají do segmentu pro statická data. Tady se díry tvořit nebudou, protože tyhle proměnné vznikají na začátku a zanikají na konci programu (takže se formát segmentu nemění).

Oddělený překlad, sestavení

Struktura programu
Program se skládá z *modulů*:

- Překládány samostatně kompilátorem
- Spojovány linkerem

Modul z pohledu programátora

- Soubor s příponou .cpp (.c)

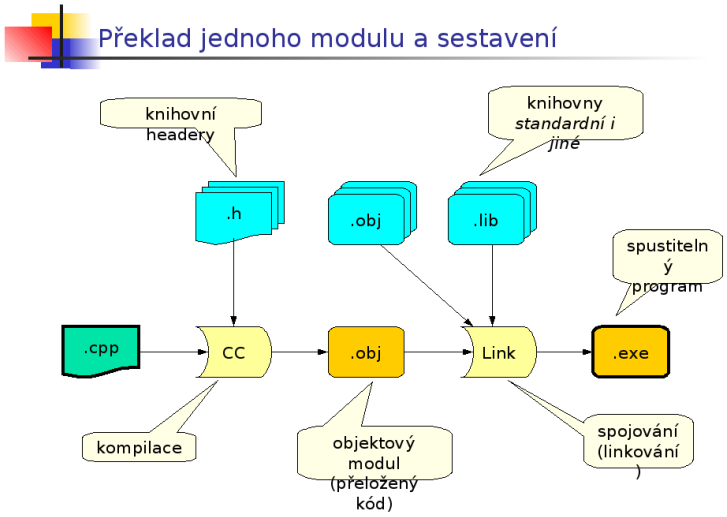
Hlavičkové soubory

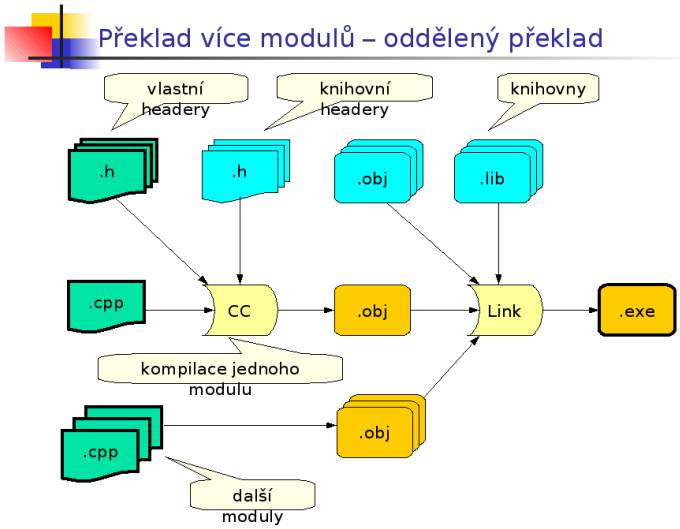
- Soubory s příponou .h
- Deklarují (a někdy i definují) identifikátory používané ve více modulech
- Vkládány do modulů direktivou include
  - Direktivu zpracovává preprocesor čistě textově
  - Preprocesor je integrován v kompilátoru jako první fáze překladu

Modul z pohledu kompilátoru

- Samostatná jednotka překladu
- Výsledek práce preprocesoru

Oddělený překlad



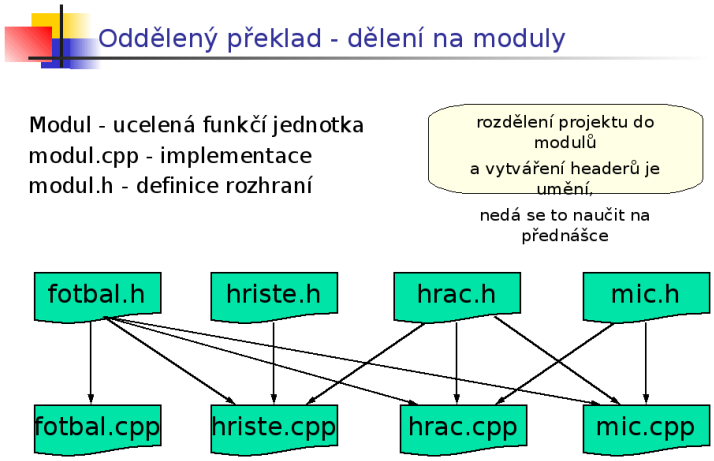


Smysl odděleného překladu modulů je urychlení celkového překladu – nepřekládat to, co se od minula nezměnilo. Oddělený překlad dnes díky automatizaci makefily (viz níže) a integrovanými prostředími není téměř pro programátora vidět.

...na tomto slide je vhodné si ujasnit, jak funguje statické a dynamické linkování (jak a kde a kdy se opravují adresy objektů atd...):

- *Statické linkování*
  Po odděleném překladu jednotlivé object moduly ještě neobsahují přímo adresy všech funkcí a externích identifikátorů, jen odkazy na ně. Linker se postará o jejich spojení dohromady. Je nutné, aby jména byla unikátní, takže u přetížených a virtuálních funkcí, jako je v C++, musí bý jména zpotvořena tak, aby ukazovala i třídu, namespace, parametry a jejich typy. To má na starosti compiler a říká se tomu *name mangling*.
- *Dynamické linkování*
  Nastává po volání operačního systému – zavedení dynamické knihovny do paměti. Jsou dvě možnosti jeho provedení, první je právě při zavádění knihovny, kdy se odkazy na všechny

funkce (a mezi nimi navzájem) naplní správnými hodnotami (podle bázové adresy, na kterou se knihovna do paměti nahraje). Druhá možnost je použití dvou pointerů při volání funkcí z knihovny – to se vytvoří tabulka skutečných adres, na kterou se z knihovny ukazuje. První možnost trvá déle při zavádění knihovny, druhá je zase pomalejší při provádění, ale umožňuje kód knihovny beze změn sdílet více procesy.



Linker je program, který prijímá jeden alebo více objektů generovaných kompilátorem a složí je v jeden spustitelný program.

Objektový kód, nebo objektový soubor je reprezentace kódu, který kompilátor nebo assembler vytvoří zpracováním zdrojového kódu. Objektové soubory obsahují kompaktní kód, často nazývaný binárky :-) Linker se typicky používá na vytvoření spustitelnýho souboru nebo knihovny spojením (slinkováním) objektových souborů. Základní částí objektového souboru je strojový kód (kód přimo vykonávaný CPU počítače).

Makefile

Smyslem programu *make* je řízení překladu a linkování. Popis závislostí jednotlivých modulů a hlavičkových souborů je definován v 1 textovém souboru – *Makefile* (tj. které soubory je nutné mít aktuální/vytvořené pro překlad kterého souboru). Make vždy po změně souboru přeloží jen to, co na něm závisí. Formát souboru make:

```
targets: files;
        commands; #comment; line-begin\
        line contd.;
```

Targets – cíle činností / cílové soubory, možno definovat vic, při spuštění make bez parametrů se bere první; univ. nástroj (nejen pro překlad C/C++). Lze definovat i vlastní makra (příkazem `<název makra> = <string>`) a pak je používat (`${makro}`).

6.2 Objektově orientované programování[1]

[1] podle zkoušení Burešem a Tůmou

Účel objektového porgramování

In the 1960s, language design was often based on textbook examples of programs, which were generally small (due to the size of a textbook); however, when programs become very large, the focus changes. In small programs, the most common statement is generally the assignment statement; however, in large programs (over 10,000 lines), the most common statement is typically the procedure-call to a subprogram. Ensuring parameters are correctly passed to the correct subprogram becomes a major issue.

Many small programs can be handled by coding a hierarchy of structures; however, in large programs, the organization is more a network of structures, and insistence on hierarchical structuring for data and procedures can produce cumbersome code with large amounts of "tramp data" to handle various options across the entire program.

Although structuring a program into a hierarchy might help to clarify some times of software, even for some special types of large programs, a small change, such as requesting a user-chosen new option (text font-color) could cause a massive ripple-effect with changing multiple subprograms to propagate the new data into the program's hierarchy. The object-oriented approach is allegedly more flexible, by separating a program into a network of subsystems, with each controlling their own data, algorithms, or devices across the entire program, but only accessible by first specifying named access to the subsystem object-class, not just by accidentally coding a similar global variable name. Rather than relying on a structured-programming hierarchy chart, object-oriented programming needs a call-reference index to trace which subsystems or classes are accessed from other locations.

Definice *(Objektově orientované programování)*

Objektově orientované programování (zkracováno na OOP, z anglického Object-oriented programming) je metodika vývoje softwaru. Dá se na něj nahlédnout jako na kolekci spolupracujících objektů – v protikladu k tradičnímu pohledu, kdy se za program považuje sled instrukcí pro počítač. V OOP je každý objekt schopný přijímat zprávy, zpracovávat data a posílat zprávy jiným objektům. Na každý objekt se tak dá nahlížet jako na nezávislý malý stroj s vlastní rolí a zodpovědností. Zjednodušeně řečeno jde o dotažení konceptu *data + algoritmy = program.* Data tvoří s kódem, který je spravuje, jeden celek. Je založeno na následujících myšlenkách, koncepci:

Objekty

Jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty si pamatují svůj stav a navenek poskytují operace (přístupné jako metody pro volání).

Implementace objektů Z hlediska jazyka není velký rozdíl mezi složenými datovými typy a třídami. Deklarace třídy obsahuje, stejně jako u složeného dat. typu, datové položky. Navíc ale obsahuje i deklarace funkcí (metod), které s nimi pracují. Některé funkce mohou mít speciální vlastnosti – statické, virtuální, konstruktory, destruktory. Navíc většina jazyků přidává možnost označení kterýchkoliv položek jako veřejné nebo privátní. Třídy mohou někdy (C++, Java) obsahovat i vnořené datové typy (výčty, ...) a dokonce vnořené třídy.

Za běhu je jedna instance třídy – objekt reprezentována v paměti pomocí:

- datových položek (stejně jako složený datový typ),
- skrytých pomocných položek umožňujících funkci virtuálních metod, výjimek, RTTI a dědičnosti (identifikace typu / jeho velikosti apod.)

Abstrakce

Programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

Zapouzdření

Zaručuje, že objekt nemůže přímo přistupovat k vnitřnostem jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

Jake ma vyhody OOP proti pouze proceduralnimu programovani - Zacal jsem zapouzdrenim, kde me vcelku potrapil pure C jazykem jehoz moduly, .h soubory, pouziti staticu, ifdefu a dalsich konstruktu muze slouzit jako slusne zapouzdreni - ukolem bylo rict na konkretnim jazyce co prinasi OOP navic (napr. vetsi granularita oddeleni (Java a viditelnost v Packagi, protected, final...))

Skládání

Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

Třída

Třída definuje abstraktní vlastnosti nějakého objektu, včetně obsažených dat (atributy, pole (fields) a vlastnosti (properties)) a věcí, které může dělat (chování, metody a schopnosti (features)). Například třída *Dog* by obsahovala věci spoločné pro všechny psy - např. atributy rasa, barva srsti a schopnosti štěkat.

Třídy poskytují v objektově-orientovaném programu modularitu a strukturu. Třída by typicky měla být rozpoznatelná i ne-programátorovi, který se ale v dané doméně problémů orientuje – tzn. že charakteristiky třídy by měly dávat v kontextu smysl. Podobně i kód třídy by měl být relativně samostané ("self-contained"). Vlastnosti a metódy tříd se dohromady nazývají i *members*.

Dědičnost

Objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd). Umožňuje zacházet s množinou tříd, jako by byly všechny reprezentovány tím samým objektem. Například známá hierarchie: grafický objekt, bod, kružnice. Navíc je to prostředek pro úsporu práce při kódování.

Implementace dědičnosti v C++: Je-li třída B (přímým či nepřímým) potomkem třídy A, pak paměťová reprezentace objektu typu B obsahuje část, která má stejný tvar jako paměťová reprezentace samostatného objektu typu A. Z každého ukazatele na typ B je možno odvodit ukazatel na část typu A – tato konverze je implicitní, tj. není třeba ji explicitně uvádět ve zdrojovém kódu. Tato konverze může (obvykle pouze při násobné dědičnosti) vyžadovat jednoduchý výpočet (přičtení posunutí).

Z ukazatele na typ A je možno odvodit ukazatel na typ B, jen pokud konkrétní objekt, do kterého ukazuje ukazatel na typ A, je typu B (nebo jeho potomka). Zodpovědnost za ověření skutečného typu objektu má programátor a tuto konverzi je třeba explicitně vynutit přetypováním. Může to znamenat odečtení posunutí v paměti.

-jak budou v paměti vypadat objekty třídy která z nějaké dědí a jak funguje přetypování pointerů mezi nimi (rodičovská třída že je v paměti nahoře, při vícenásobné dědičnosti že je to složitější),

-Vysvetleni proc muze byt vicenasobna dedicnost (alias diamant) zlo. (peklo delaji napr. polozky).

-private + public dedicnost

Polymorfismus

: odkazovaný objekt se chová podle toho, jaký je jeho skutečný typ. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší. V praxi se tato vlastnost projevuje např. tak, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy (třídy, která přímo či nepřímo z této třídy dědí), která se může chovat jinak, než by se chovala instance rodičovské třídy, ovšem v rámci mantinelů, daných popisem rozhraní.

-Konkretni ukazka polymorfismu, dedicnosti a zapouzdreni NA KONKRETNI NAPSANE TRIDE (no matter jaky jazyk, bez ohledu na syntakticke chyby). příklady v nějakém jazyce

Virtuální metody

V objektově orientovaném programování se pod pojmeme virtuální funkce nebo virtuální metoda myslí funkce/metoda (v kódu označená klíčovým slovem – typicky virtual), jejíž chování je určeno definicí v posledním objektu (ve směru dědičnosti od rodičů k potomkům), který definici funkce s danou signaturou obsahuje. Tento koncept je důležitou součástí polymorfismu v objektově orientovaném programování.

Koncept virtuálních funkcí řeší následující problém:

Když v OOP dědí nějaká třída od jiné, tak instance poděděné třídy může být přetypována na libovolného ze svých předků. Pokud však poděděná třída předefinovala poděděnou metody, nebude při jejím přetypování na předka jasné, kterou z definicí metody použít.

Rozhodnutí otázky, kterou funkci zavolat, přináší právě rozdělení metod na virtuální a nevirtuální. Když je funkce označena jako virtuální, je použita definice funkce z poděděné třídy (pokud existuje). V opačném případě je použita definice ze třídy na kterou je instance třídy právě přetypována.

Pozdní vazba (late binding; virtual call): Je-li metoda nějaké třídy virtuální či čistě virtuální, pak všechny metody se stejným jménem, počtem a typy parametrů deklarované v potomcích třídy jsou považovány za různé implementace téže funkce. Která implementace se vybere, tedy které tělo bude zavoláno, se rozhoduje až za běhu programu podle skutečného typu celého objektu. Použije se tělo z posledního potomka, který definuje tuto funkci a je součástí celého objektu. Pozdní vazba má smysl pouze u vyvolání na objektu určeném odkazem.

Pozdní vazba je implementačně umožněná skrytým pointerem na *tabulku virtuálních funkcí (VMT)* uvnitř každého objektu. Existuje pro každou třídu jedna. Při dědičnosti zůstává v celém objektu odkaz jeden (v C++ jich může být při výsenásobné dědičnosti i více), ale (i pro nejvnitřnější bázovou třídu) odkazuje na tabulku odvozené třídy. V tabulce musí být proto pointery na funkce, deklarované už u bázové třídy, umístěny na začátku (aby bylo možné volat funkce bázové třídy mezi sebou bez změny kódu).

Kontruktory