Obecné otázky

1. Vývoj programovacích jazyků (FORTRAN, Algol 60, PL/I, Algol 68, Pascal, Simula 67, Ada, C a C++,Java, APL, LISP, PROLOG)

Fortran

- navrhla firma IBM
- pro vědecké výpočty a numerické aplikace
- stále udržovaný viz poslední standardy (Fortran 2018, Fortran 2008)
- autoři měli strach, že to nikdo nebude používat pokud to nebude tak rychlé jako assembly language → mnoho optimalizací pro kompilátor
- první verze chtěli precizně zarovnaný zdrojový kód jako např. python
- dá se považovat za první “vysokoúrovňový programovací jazyk”

Algol 60

- dá se považovat za 2. po Fortranu
- mnoho průlomových novinek
    - **scope**
    - rekurzivní volání
    - boolean hodnoty
    - cykly jejichž podmínkou byl nějaký boolean namísto iterování podle proměnné
    - více rozměrné pole
    - vícenásobné přiřazení a := b := c
- Navrhnut mezinárodním výborem informatiků jako univerzální jazyk pro psaní algoritmů, aniž bychom museli řešit problémy související s konkrétním hw

IBM PL/I

- PL/I - programming language one
- velmi složité parsování jazyka pro kompilátory
- jednoduchý na naučení
- efektivní (je vysokoúrovňový celkem)
- poprvé se zde objevil koncept **výjimek**
- univerzální jazyk vhodný zejména pro hromadné zpracování dat např:
    - výpočty mezd
    - podniková administrativa
    - finanční výpočty (uměl fixní i pohyblivou aritmetiku)
    - zpracování textových i bitových řetězců (i různě složitě strukturovaných)
- v té době byla část programovací komunity u Fortranu nebo Cobolu (akademici, business)
- PL/I si dával za cíl sjednotit komunitu univerzálním jazykem

Algol 68

- nástupce Algolu 60
- navržen s cílem:
    - rozšířit možné aplikace Algolu
    - víc rigorózně definovat syntaxi a sémantiku
- spousta konceptů se později objevila v budoucích programovacích jazycích (vlastně byla znovuobjevena)
- kritizován některými pro ztrátu jednoduchosti
- V SSSR měl velký dopad místní programování

Pascal

- navrh Niklaus Wirth na ETH v Curychu
- navržen jako jazyk na výuku programování
- Turbo Pascal
    - slavná knihová Borland licence za 49 USD
- původně trpěl některými nedostatky, které vznikly tím, že to měl být jazyk na výuku, postupem času většinou vyřešeny
    - např. původně se logické formule vyhodnocovali kompletně
- z Pascalu vzniklo Delphi
- Myslím, že jsem si Pascalu užil dost

Simula 67

- první objektově orientovaný jazyk
- odvozen z Algolu
- navržen pro simulování systémů hromadné obsluhy
- prakticky se nepoužíval mimo akademické prostředí
- avšak měl silný dopad na vývoj objektově orientovaných jazyků (C++)
- garbage collector

Ada

- na objednávka od amerického ministerstva obrany
- v novějších verzích přidána podpora objektově orientovaného programování

Cobol

- ve spolupráci výrobců počítačů, uživatelů a amerického ministerstva obrany
- požadavky:
    - sestavení programů v minimálním čase s minimálním programovacím úsilím
    - zápis programů v jazyce blízkém angličtině
    - snadný převod programů na nové typy počítačů
- programy opravdu vypadaly jako anglické věty

APL

- hlavní typ vícerozměrné pole

- používala se obrovská spousta speciálních symbolů
- nutnost vlastnit speciální klávesnici

Lisp
- spíše rodina programovacích jazyků (Lispů je hodně)
- akademický jazyk
- hlavně obor umělá inteligence

Prolog
- vyjádření faktů a vztahů mezi nimi
- extrémní důraz na potlačení imperativní složky
- pouze se popisuje cíl výpočtu
- založen na predikátové logice 1. řádu (Hornovy klauzule)
- počítačová lingvistika, umělá inteligence
- unifikace, rekurze, backtracking

C ⊂ C++ (skoro)

C++ - vybraný jazyk, nebudu zde rozepisovat

Jáva
- syntaxe z C++
- některé funkce dokonce ze SmallTalku
- princip WORA
    - write once run anywhere
- velmi oblíbený a používaný na business aplikace
- garbage collector (ne příliš dobrý)

2. Programovací styly

- proceduální (imperativní)
- strukturované
- objektově orientované

Úvod

Styl programování je vlastně způsob, jakým nad programem uvažujete, když program tvoříte. Obecně se dá prohlásit, že se usadily strukturované objektově orientované programovací jazyky, které se používájí v komerční sféře. Funkcionální programování je víceméně doména akademické sféry.

Většina jazyků, které se používají v praxi jsou spíše imperativní (proceduální), tzn. programátor popisuje průběh výpočtu programu narozdíl od deklarativního programování,

kde popisujeme jaký výsledek chceme a jak se k němu program dostane nám je úplně jedno.

Strukturované programování

Snad nejúspěšnější jazyk všech dob, jazyk C, byl následovníkem nám už neznámého jazyka B. Proto jeho název.
Se strukturovanými jazyky přišel nový způsob přemýšlení nad programy. Strukturované jazyky se nazývají strukturovanými proto, že přinášely možnost definování a používání složitějších celků, struktur. A to hned z několika pohledů. Např. datové struktury umožnily vytvořit jeden identifikátor pro celou skupinu proměnných.

V každém případě přineslo strukturované programování do programování základní pořádek. Funkce jsou dodnes úžasným nástrojem k izolaci komplexit (složitostí) a k přepoužívání kódu (o čemž bude řeč v jiném článku). Každá funkce by měla obsahovat své “tajemství”, něco, co ukrývá před ostatním světem a co zná jen ona sama, ať už je to způsob, jakým něco provádí, nebo třeba zdroje, odkud bere nějaká data (princip tzv. information hiding).
I když volání funkcí představuje z hlediska procesoru zbytečně prováděné akce (především ve formě ukládání a odebírání dat na/ze zásobníku), je smysl funkcí bezpochyby naprostým ospravedlněním.

Objektově orientované programování

Objektový způsob myšlení v programování se prosadil vlastně nedlouho po principech strukturálního programování (Simula byla objektově orientovaná). Objektové programování vhodně doplňuje struktury o další strukturální úroveň, o objekty.
Objekt je to, co si pod tím slovem každý z nás představí. Jsou to hodinky, televize nebo člověk. Je to ale také vítr, rychlost nebo barva. I myšlenka, sekunda, obchodní transakce nebo léčení jsou objekty. Při objektovém myšlení se prakticky vynalézavosti meze nekladou, co vše lze označit jako objekt.
Základní myšlenkou objektově orientovaného přístupu je, že všechny funkce nebo proměnné jsou vlastnostmi ne systému nebo programu, ale jednotlivých objektů, které v něm vystupují a které si v něm nadefinujeme. Funkce i data tedy neplavou volně v programu, jako tomu je v případě ryze strukturálního programování, ale jsou vázány pouze na objekty a bez nich nemají smysl. V čistě objektových jazycích jako C# nebo Java tedy nelze mít funkci mimo objekt, v jazycích, které umožňují mít paralelně strukturální i objektový přístup (jako např. C++), toto možné je.

Objektové myšlení daleko víc odpovídá myšlení lidí. Když chceme znát čas, řeknou nám to hodinky. Když chceme poslouchat hudbu, vyluzuje se z rádia. Kromě principu skrývání informací (information hiding), což je ono “tajemství”, které objekt skrývá před světem (jinak také tzv. zapouzdřuje), což se objevilo už s příchodem strukturálního programování a funkcí, je zde několik úplně nových principů.

3. Strukturované, modulární, objektové programování, event-driven

Modulární programování

Modulární programování je technika návrhu softwaru, která zdůrazňuje rozdělení funkčnosti programu na nezávislé, zaměnitelné moduly, z nichž každý obsahuje vše nezbytné pro jediný aspekt požadované funkcionality. Koncepčně moduly představují oddělení odpovědností a zlepšují udržovatelnost softwaru explicitním vyjádřením logických hranic mezi komponenty. Při vytváření většího množství softwarových projektů přinášejí koncept znovupoužitelnosti, umožňující moduly vytvořené v jednom projektu používat i v projektech jiných.

event-driven programování

Obvykle souvisí s tvorbou aplikací s uživatelským rozhraním.

- tok programu řízen událostmi
- události nastávají obvykle určitou uživatelskou akcí - klik či pohyb myši, stisk tlačítka…
- událostmi řízené aplikace musí být většinou programovány jako vícevláknové (i když spouštění vláken obvykle explicitně programovat nemusíme).

Programování řízené událostmi je druh programování kdy běh programu je řízen událostmi. Jedná se o dominantní styl programování, který je využíván hlavně pro programování aplikací
s GUI.

V aplikaci řízené událostmi je obvykle hlavní smyčka, která naslouchá, jestli nenastala nějaká akce, z které by měla vzniknout událost. Pokud ano, tak zavolá callback funkci události, která by měla být ohlášena. V embedded systémech se podobného cíle dobereme pomocí hardwarových interruptů.

Programování řízené událostmi je možné v jakémkoliv programovacím jazyce, ačkoliv je obvykle tento úkol je obvykle jednodušší v jazycích, které disponují vysokoúrovňovými abstrakcemi.

4. Logické a funkcionální programování

Logické programování

Logické programování je druh programování (programovací paradigma) vycházející z matematické logiky.

Logický program je množina vět (v prologu ukončena tečkou) faktů
a pravidel.

Stylově se tedy jedná o programování deklarativní a nikoliv imperativní, programátora nezajímá jak se k výsledku (prolog) dobere.
Funkcionální programování

Výpočet je vlastně vyhodnocování matematických funkcí. Funkcionální programování má své kořeny v lambda-kalkulu, formálním systému vyvinutém v 30. letech k vyšetřování definicí funkcí, jejich aplikace a rekurze. Mnoho funkcionálních programovacích jazyků může být považováno za rozšíření lambda kalkulu.

V praxi je rozdíl mezi matematickou funkcí a představou funkce použité v imperativním programování. Imperativní funkce mohou mít vedlejší účinky, které mění stav programu. Stejné volání může vést k různým návratovým hodnotám v závislosti na stavu vykonávaného programu. Oproti tomu ve funkcionálním kódu návratové hodnoty funkcí záleží pouze na argumentech funkce, a tudíž dvě volání téže funkce se stejnými argumenty vrací vždy stejnou hodnotu. Eliminace vedlejších účinků může zjednodušit analýzu a pochopení chodu programu, což je jednou z klíčových motivací pro vývoj funkčního programování.

5. Datové typy a datové abstrakce, generické typy

Datový typ

- druh nebo význam hodnot, kterých smí nabývat proměnná
- určen oborem hodnot a podporovanými operacemi
    - [0,2^31], <,>,+,-,*,/…

Součástí programovacího jazyka je definice základních datových typů. Pomocí těchto základních typů může ve většině jazyků programátor tvořit nové složené typy.

- ordinální datové typy
    - tvoří lineárně uspořádanou množinu (každý prvek má předchůdce i následníka)
    - u posledního prvku vždy dochází k přetečení
- logické hodnoty

    - boolean (true, false)
    - obvykle false == 0, true != 0
- znak
    - char
    - původně 1 byte (ASCII)
    - dnes UTF-něco, tedy často více bytů…
- neordinální datové typy
    - reálná čísla
- prázdný datový typ (specialita C)

Složené datové typy
- pole (array)
- textový řetězec (jakási kolekce znaků)
- kolekce (seznam)
- záznam (record, struct)

Generický datový typ reprezentuje množinu datových typů.

V C++ unikátní systém šablon, které si instancijují compile-time.
V ostatních jazycích run-time generika. Je možné třeba garantovat aby povolený datový typ implementoval nějaký interface…

ADT (abstraktní datový typ)
- matematický model pro datový typ, který je definovaný z pohledu uživatele
- Tj. přípustné hodnoty, přípustné operace s hodnotami a jejich následné chování
- velmi podobné například konceptu algebraických struktur v matematice

Rozdíl mezi ADT a datovou strukturou:
- ADT je z pohledu uživatele
- ADT je teoretický koncept, který je využíván při analýze a návrhu (algoritmů, datových struktur…)
- datová struktura je konkrétní reprezentace dat z pohledu “implementátora”

6. Řídící struktury programovacích jazyků

- konstrukce využívaná pro zápis programů
- rozhodují o dalším provádění programu: větví jeho běh, vytváří cykly nebo jinak mění běh programu
- typy řídících příkladů
    - posloupnost příkazů provádějících se postupně jeden po druhém
    - větvění (podmíněný příkaz)
        - v závislosti na splnění podmínky se určitý příkaz (skupina příkazů) buď provede nebo neprovede (přeskočí)

    - cyklus
        - v závislosti na splnění podmínky se určitá část programu vykoná vícekrát

Větvení

Podmíněný příkaz a podmíněná konstrukce jsou prostředky programovacího jazyka, které umožňují rozdílné chování programu,
v závislosti na specifikované logické podmínce, která je vyhodnocena jako pravda, či nepravda.

- IF..GOTO Forma vyskytující se v nestrukturovaných programovacích jazycích. Napodobuje typickou strojovou instrukci GOTO, která umožňuje skok na určitý řádek kódu.
  IF..THEN..(ENDIF). Pokud je podmínka v části IF vyhodnocena jako pravda, je vykonán kód specifikovaný v části THEN. V případě, že je podmínka vyhodnocena jako nepravda, je kód specifikovaný v části THEN vynechán a program pokračuje dále za částí ENDIF.

- IF..THEN..ELSE..(ENDIF) Oproti předchozímu výrazu, je v případě vyhodnocení podmínky v části IF jako nepravda vykonán kód specifikovaný v části ELSE.
- Podmíněné příkazy mohou být a také velmi často bývají částí jiných podmíněných příkazů. Některé jazyky umožňují sloučit ELSE a IF do ELSEIF.

Příkaz switch (v některých jazycích uveden jako case) porovnává předanou hodnotu s předem specifikovanými konstantami. V případě shody předané hodnoty s definovanou konstantou, vykoná příkaz, nebo příkazy, které jsou definovány za ní. Obvykle tato konstrukce také obsahuje možnost, pro případ, že by shoda nalezena nebyla, nejčastěji ELSE, OTHERWISE nebo default. V dynamických jazycích nemusí být případ pro porovnání omezen pouze na konstanty a může být rozšířen do vzorového porovnávání, jako například v shell skriptech, kde regulární výraz '*)' označuje jakýkoli řetězec.

Cyklus
- opakované provádění posloupnosti příkazů
- typy
    - nekonečný (embedded zařízení)
    - for
    - do-while
    - while-do
- k násilnému “vyskočení” z cyklu je obvykle možné použít příkaz **break** (exit)
- k násilnému “skočení” k další iteraci je obvykle možné použít příkaz **continue** (next)

7. Struktura programovacího jazyka, proměnné, jejich hodnoty, typy, definiční oblast, životní cyklus Překlad a interpretace programovacího jazyka, separátní překlad

Proměnná

- koncept pocházející z matematiky zastupující nějaký objekt
- vznikl pro řešení algebraických rovnic
- V imperativním programování je proměnná „úložiště“ informace (tedy vyhrazené místo v paměti - v některých jazycích se ovšem v průběhu výpočtu může místo, kde je proměnná uložena, měnit).
- Každá proměnná má nějaký typ
    - v beztypových (slabě typovaných) jazycích můžeme toto úložiště kdykoliv přepsat něčím jiného typu
    - v silně typovaných má proměnná asociovaný typ (jiný typ do ní prostě nedáme)
- základní typy
    - znak (char)
    - textový řetězec (string)
    - pole (obvykle říkáme čeho)
    - struktura (objekt)
    - ukazatel (často říkáme na co)
    - celé číslo
        - podle fixního rozsahu (short, int, long…)
    - reálné číslo
        - float, double, decimal
- mezi některými typy proměnných je možné převádět, ať už implicitně nebo explicitně
- v matematice proměnné zpravidla jednopísmenkové, v programování libovolné výstižné jméno nejlépe
- v deklarativním programování (není zde obvykle přiřazovací příkaz) je koncept proměnné v programování bližší vztah ke konceptu proměnné v matematice

Oblast, kde proměnná žije = scope

- Scope = oblast (dnes obvykle {}), kde je platná asociace jména k entitě
    - Příklad tzv. statického (lexical) scope
    - např. asociuji jméno x k číslu 1
    - {
    - int x = 1;
    - }
- ve zbylé části programu už můžeme definovat jinou proměnnou x, tato již neexistuje
- definice scopu vznikla v 60. letech, s tímto konceptem přišel Algol 60, doteď se ta definice nezměnila
- Lexical (static)
    - v závislosti na pozici ve zdrojovém kódu
    - vidíme, že scope této funkce je mezi { }
- existují i jiné typy scopů, např. dynamický scope

    - existence proměnné závisí na stavu programu

V některých programovacích jazycích (např. Pascal) si nemůžeme proměnnou vytvořit jen tak někde.

- v procedurách a funkcích je speciální definiční blok na začátku

V některých jazycích není možné vytvořit proměnnou rovnou s přiřazenou hodnotou (staré C)

Běžně dnes je možné vytvářet proměnné s přiřazením, popř. vytvářet a inicializovat proměnné v for cyklu, if statementu…

Překladač (kompilátor)

- softwarový nástroj
- překlad zdrojového kódu zapsaného v programovacím jazyce do nějakého výstupního kódu
- buďto přímo do strojového kódu pro danou architekturu
- nebo do nějakého mezi-kódu, kterému rozumí nějaký interpret, přes který jde program spustit
- jako první “překladač” se považuje A-0 systém napsaný pro počítač UNIVAC (nebyl to tak úplně překladač jak ho známe dnes, ale spíš jakýsi předchůdce)
- první skutečný kompletní překlač - jak ho známe dnes byl pro Fortran
- obvykle “překladače” z assembleru do strojového kódu nejsou považovány za překladač

Dneska však není obvyklý princip překladu programů takový, že by kompilátor dostal veškerý zdrojový kód a vyplivl výstup.

Zdrojový kód programu bývá obvykle rozdělen do více jednotek, u velkých programů se může jednat o tisíce souborů se zdrojovými texty. Důvodů pro takovou fragmentaci zdrojového kódu programu je mnoho, např. snadnější údržba zdrojových kódů, možnost znovu-využívání jednotek, nižší nároky na kompilátor atp.

Proto kompilátor překládá jednotky zdrojového kódu do tzv. objektového kódu. Objektový kód jednotky je v podstatě již strojovým kódem cílové architektury, obsahuje však dodatečné informace umožňující přemístění návěstí a proměnných přeložené jednotky a informace o vazbách na externí proměnné a na externí kód. Z jednotlivě přeložených objektových kódů a z funkcí obsažených v knihovnách pak linker sestaví výsledný spustitelný kód.

Překladač může dostat kromě zdrojáku i příkazy pro svou činnost prostřednictvím options, například o požadované míře optimalizace nebo o povolených rozšířeních syntaxe, použitých v konkrétním zdrojáku. Taktéž zdroják může obsahovat příkazy nebo informace pro kompilátor, známé jako pragmy. A výstupem nemusí být jen strojový kód, ale i dodatečné informace pro debugger apod.

8. Podprogramy a předávání parametrů

Funkce + procedury = metody = podprogramy

- funkce vrací návratovou hodnotu, procedura ne
- název metody se používá hlavně v OOP (kde třída má metody)

Používání procedur je důležitým nástrojem strukturovaného programování a umožňuje zavádět do programů vyšší míru abstrakce.

Používání podprogramů je při programování natolik mocný nástroj, že jej umožňuje naprostá většina programovacích jazyků. Používání podprogramů může vést ke snížení nákladů na údržbu rozsáhlých projektů a zároveň zvyšovat jejich kvalitu a spolehlivost. Podprogramy jsou často sdružovány do knihoven, které se zaměřují na určitou oblast (například práce s grafikou, zvukem, šifrování a podobně). Knihovny usnadňují sdílení a prodej kódu. Objektově orientované programování přidružilo podprogramy k datům (tj. metody jsou součástí objektů nebo tříd).

Podprogram může mít parametry (také označované za „argumenty“ jako v matematice), tedy při volání zadávané vstupní hodnoty podprogramu, které udávají, s jakými hodnotami má pracovat. Podprogram může vracet návratovou hodnotu.

Důvody použití podprogramů (částečně již řečeno)
- rozklad složitých problémů na jednodušší, nebo v případě rekurze menší
- odstranění opakování kódu v programu, a díky parametrům jeho zobecnění
- umožňuje znovupoužití v jiných programech, obvykle formou modulů nebo knihoven
- rozvržení projektu mezi více programátorů
- odstínění detailů implementace od konkrétního použití podprogramu

Pascal, Fortran a Ada rozlišují mezi procedurama a funkcema.

Výsledek a chování programové funkce na rozdíl od funkce v matematice se chová odlišně. To spočívá ve dvou projevech:

- nemusí záviset jen na jeho parametrech a při volání se stejnými parametry může podprogram vracet jiné návratové hodnoty, v závislost na aktuálním stavu programu.
- obdobně návratová hodnota zdaleka nemusí být jeho jediným výsledkem, ale má i vedlejší účinek, může měnit i jiné hodnoty, než jen vrácené do výrazu, ve kterém byla funkce volána, např. funkce může cachovat

Silně typované vs slabě typované jazyky
- silně typované kontrolují typy parametrů a návratových hodnot
- u slabě typovaných si musí dávat pozor programátor

Některé jazyky povolují proměnný počet vstupních parametrů podprogramu (variadic template v C++, params v C#,...)

Předávání parametrů

- velmi základní rozdělení (v C++ je to mnohem komplikovanější)
- předpokládejme, že metoda čeká vstupní parametry na volacím zásobníku
- by value
    - zkopíruje hodnotu parametru na volací zásobník
- by reference
    - zkopíruje adresu paměti, kde se nachází proměnná na volací zásobník

Rekurzivní funkce

- funkce, která ve svém těle volá sebe sama
- zpravidla krátký kód
- P(konstanta) = ručně vyřešíme triviální případ
- P(n) = něco + P(m), kde m < n

Přetížené funkce

Přetížení funkce (anglicky overloading) znamená deklarovat více funkcí pod stejným názvem lišících se ve struktuře seznamu parametrů (počet, datový typ). Při volání funkce překladač analyzuje parametry a podle toho určí odpovídající funkci. Přetížení se týká i návratové hodnoty, překladač analyzuje typ požadované návratové hodnoty na levé straně přiřazovacího operátoru = a podle toho vybere příslušnou funkci.

9. Zpracování výjimek

Výjimečná situace, která může nastat za běhu programu, případně ve volané metodě nebo funkci. Obecně jsou jako výjimky označovány objekty nesoucí informace o chybovém stavu. V případě objektově orientovaného programování jsou jako výjimky označovány objekty nesoucí tyto informace.

Vyhození výjimky je způsob signalizace, že probíhající část zpracovávaného kódu nemohla být provedena normálně. Důvodem může být např. nedostupnost použitého zdroje (neexistující soubor, chyba média, nedostatek paměti) nebo chybný vstupní argument (zadaná hodnota se nenachází v definičním oboru použité funkce).

V některých případech výjimek je také možno chod programu obnovit. Zde vždy závisí na konkrétním použitém mechanismu ošetření výjimky a také na druhu výjimky. Příkladem relativně snadno obnovitelného chybového stavu je dělení desetinného čísla (float) nulou. Naopak velmi obtížně se ošetřují stavy s nedostatkem paměti (out-of-memory).

Využití výjimek a jejich zpracování není pro všechny programovací jazyky vždy stejné a přístup k nim se liší i mezi jednotlivými programátory. Typickým přístupem prosazujícím použití výjimek je tzv. Defenzivní programování, které se vyznačuje snahou o ošetření všech neočekávaných stavů.

Blok s výjimkou obvykle začíná klauzulí „try“ nebo „begin“, jednotlivé chybové stavy jsou poté označeny bloky „catch“ nebo „except“. Celý blok zpracování výjimky je ukončen sekcí „finally“ nebo „ensure“, která slouží pro uvolnění systémových zdrojů použitých při zpracování výjimky.