II. Efektivita programu

V této části předmětu se budeme bavit o efektivitě programu. Cílem bude si vysvětlit, jak psát programy tak, aby svého výstupu dosáhly v minimálním čase, což lze říct i tak, že chceme, aby výsledku dosáhly s užitím minimálního množství strojových instrukcí. Jen okrajově se dotkneme volby samotného algoritmu. Předpokládejme spíše, že algoritmus jako takový už je dán a my musíme zvolit, jak ho budeme implementovat, aby jeho běh byl co nejkratší. Budeme se tedy rozhodovat, jaký vůbec použít programovací jazyk a jak v něm program zapsat tak, aby mohl být algoritmus rychle zpracován. K tomu budeme potřebovat určité povědomí o tom, jaká je architektura počítače a jakým způsobem je námi napsaný program převeden do stavu, aby mohl být spuštěn. Absolvování předmětů, ve kterých je probírána architektura počítače a kompilátory vyšších programovacích jazyků, by pro pochopení látky bylo ideální, ale není zcela nutné. To nejdůležitější pro správné pochopení si vysvětlíme. Pro začátek si pouze řekněme, že program zapsaný ve vyšším programovacím jazyku musí být buď kompilátorem přeložen do posloupnosti instrukcí procesoru, které jsou následně procesorem vykonávány a nebo musí na počítači běžet program zvaný interpreter, který algoritmus zapsaný ve vyšším programovacím jazyku interpretuje a vykonává. Ve druhém případě jde - mírně nepřesně řečeno - o průběžnou kompilaci a okamžité provedení přeložené části programu. Některé programovací jazyky jsou typicky kompilované (C/C++, C#, Java) a jiné se kompilovat nedají a jsou typicky interpretované (PHP, Javascript).

- **Efektivní programy x čitelné programy**

Obecně platí, že při psaní programů tak, aby mohly fungovat co nejefektivněji, musíme používat takové programátorské obraty, které nejsou úplně přehledné a tedy snadno čitelné. V takových situacích by měl programátor vždy zvážit, co je pro něho v danou chvíli důležitější. Zda má být program přehledný a dobře čitelný a nebo potřebuje klást důraz na výkon i na úkor čitelnosti

- **Výkonnost hardware v současnosti převyšuje požadavky běžného software -> při vývoji SW je proto potřeba spíše dbát na efektivitu práce (čitelnost programu)**

Protože pro běžné úlohy výkon současného hardware bohatě postačuje, je potřeba klást důraz spíše na to aby programy byly dobře čitelné a přehledné

- **Tlak na efektivitu u vysoce využívaných webových serveru a webových služeb**

V současnosti typickým příkladem, kdy je potřeba klást důraz na efektivitu programu, jsou webové servery poskytující webový obsah nebo data formou webových služeb, pokud se očekává jejich vysoké zatížení, tj. bude k nim přistupovat v krátkém časovém intervalu vysoké množství uživatelů. Tyto situace se samozřejmě částečně řeší i vyšším výkonem hardware, případně i dalšími technologiemi jako je rozložení výkonu mezi více serverů. Ne vždy je ale tato cesta možná, takže je potřeba se zabývat i tím, aby implementace zvoleného algoritmu byla co nejefektivnější.

- **Tlak na efektivitu programu u „malého" levného hardware (malé jednodeskové PC, jednočipové mikropočítače)**

Dalším poměrně typickým příkladem jsou implementace algoritmů, které mají být zpracovány na extrémně levném a tudíž pomalém hardware. Prakticky jakékoliv netriviální technické zařízení v sobě obsahuje procesor a jeho chování je řízeno pomocí software. Tady mám na mysli např. mikrovlnné trouby, pračky a jiné domácí přístroje, ale i třeba čtečky čipových karet, alarmy, elektroniku v autech apod. Vesměs se jedná o zařízení, které se vyrábějí v obrovských sériích a pokud by jejich elektronika měla významný podíl na výrobních nákladech, nebyly by dostatečně konkurenceschopné. Z toho důvodu se v nich používají jednočipové procesory mnohdy v ceně nižší než $1. Tyto levné procesory mají velmi malou paměť a jsou velmi pomalé, takže algoritmy musejí

být implementovány velmi pečlivě.

- **Tlak na efektivitu u programu zpracovávajícího v omezeném čase obrovské množství dat**

Další situací, kdy je potřeba algoritmy implementovat co nejefektivněji, je zpracování dat v omezeném čase. Zde jako příklady uveďme zpracování obrazových dat v reálném čase či komprese dat přenášených po datové lince. Velikost obrazových dat závisí na rozlišení jednotlivých snímků (počet pixelů horizontálně a vertikálně) a počtu snímků za vteřinu. Mají-li být taková data zpracovaná v reálném čase, musí být implementovaný algoritmus na daném hardware schopen zpracovat za časovou jednotku takový objem dat, který odpovídá počtu snímků za tuto časovou jednotku. Podobně u komprese dat přenášených po lince je tato komprese smysluplná pouze tehdy, pokud je komprese natolik rychlá, aby doba komprese plus doba přenosu byla kratší než doba přenosu původních dat bez komprese.

- **Znalost procesu kompilace a činnosti přeloženého programu napomáhá používání takových konstrukcí ve vyšším programovacím jazyce, které po přeložení pracují maximálně efektivně**

V extrémním případě lze algoritmus implementovat přímo ve strojovém kódu procesoru, na němž má být zpracováván. Takto implementovaný algoritmus bude zpravidla rychlejší než algoritmus zapsaný ve vyšším programovacím jazyku a přeložený kompilátorem do instrukcí procesoru. Nevýhodou ovšem je, že pokud by měl být provozován i na jiném procesoru, musel by být přepsán. Proto se o této variantě bavit nebudeme a zaměříme se výhradně na zápis algoritmu ve vyšším programovacím jazyce. Proto budeme potřebovat mít aspoň trochu představu, jak takový kompilátor pracuje, abychom se v případě nutnosti vyhnuli využití takových prostředků, které jsou sice přehledné a vedou k elegantním programátorským obratům, ale kompilátor nemá příliš prostoru je přeložit dostatečně efektivně.

8. *Optimalizace algoritmu*

Neefektivní programátorské obraty dokáže vyřešit kompilátor, ale v žádném případě ne všechny, někdy to při nejlepší snaze nejde.

Pokročilé kompilátory dokážou program do určité míry analyzovat hledat situace, které lze zkompilovat efektivněji (pozn.: někdy je nutno optimalizace explicitně povolit, např. u gcc - kompilátor jazyka C/C++). Typickým příkladem optimalizace může být např. když ve vyšším programovacím jazyku si složitější výpočet rozdělíme do několika částí, v nichž vznikne vždy mezivýsledek, který programátor uloží do nějaké proměnné. Pokud je tato proměnná použita jen jednou v rámci výpočtu následující části a nikdy více, nemusí kompilátor vůbec takovou proměnnou vytvořit a mezivýsledek si drží pouze v registru procesoru. Jsou ale situace, kdy bychom takovou “optimalizaci” měli udělat sami, např. v následujících případech:

- **vyloučení invariantního výrazu z cyklu**
  Pokud je v těle cyklu výpočet, který vychází čistě z dat, která se v jednotlivých iteracích cyklu nemění, musí být nutně výsledek výpočtu vždy stejný, a proto takový výpočet nemá žádné opodstatnění aby byl uvnitř cyklu. Pokud je takový výpočet dokonce “ukryt” ve funkci/metodě, která je z cyklu volána, hledá a řeší se tato situace o něco hůř, ale pokud by měla kritický vliv na rychlost zpracování, nezbývá než strukturu programu natolik předělat, aby se výpočet skutečně dostal mimo cyklus.
- **výpočet v getteru**
  Metody typu getter (zpravidla mají název get+název hodnoty), by opravdu měly jen vracet

hodnotu, která už je spočtena. Pokud tomu tak není, měly by se jmenovat jinak, aby programátor, který je používá, na první pohled viděl, že jejich zpracování může trvat déle. Pokud už výpočet v getteru potřebujeme, je vhodné si výsledek cachovat.

- **opakované výrazy, které by se daly při prvním výskytu uložit do proměnné**
  Pokud se ve složitějším výpočtu víckrát opakuje ten samý podvýpočet se stejnými daty, je potřeba ho provést pouze jednou, výsledek si uložit do proměnné a tu potom použít na všech místech, kde je to potřeba.
- **non-tail rekurze převést na tail rekurzi (pokud to lze)**
  Tail rekurze je taková rekurze, kdy rekurzivní volání je poslední činnost ve funkci. Kvalitní kompilátor dokáže u tail rekurze před dalším rekurzivním voláním na konci rekurzivní funkce uvolnit pamět zabranou volající funkcí, která je tak k dispozici pro volanou funkci.

Další příklady vylepšení:

```
    if (name_en != null && "cz".equals(country.getId()))
```

je lepší než

```
    if ("cz".equals(country.getId()) && name_en != null)
```

(první varianta je výhodnější, protože otestovat pointer na objekt s null je rozhodně rychlejší než porovnávat dva řetězce, zde dokonce se zavoláním funkce a případně dokonce s voláním getteru apod)

9. *Mechanismus přístupu k datům*

Každý program pracuje s daty. Konec konců i klasický první učebnicový program, který pouze vypíše "Hello world", pracuje s daty. Funkci, která realizuje vypsání textu, je potřeba předat pointer na řetězec, který je musíme chápat jako data. A dále řetězec samotný je posloupnost bajtů, tedy data. Dalo by se říci, že naprostá většina instrukcí, kterou procesor provádí, jsou nejrůznější manipulace s daty. Proto je potřeba se podívat na to, jak preložený program s daty pracuje a to zvlášť pro různé třídy dat (typy paměti) - lokální proměnné, parametry funkcí, statická data a data na haldě (častěji se i v češtině používá anglický pojem heap - jde o oblast paměti, která se přiděluje a uvolňuje na vyžádání programu a její nutnou součástí je poměrně komplikovaný mechanismus přidělování, uvolňování, evidování volné a obsazené paměti). Pokud budeme rozumět tomu, se kterými třídami dat je jednodušší manipulovat, můžeme je v místech programu, která jsou kritická pro dobu zpracování, preferovat před těmi komplikovanějšími. Při popisu práce programu s daty si budeme všímat tří základních situací:

- co všechno musí program udělat, aby pro daná data vyhradil prostor v paměti
- co musí udělat, aby tuto paměť uvolnil pro další využití v rámci tohoto programu
- jak data zpřístupní, tj. jak získá adresu těchto dat, aby mohl data odtud převzít a nebo je mohl na tuto adresu zapsat

Na tomto místě musíme udělat malou vsuvku pro ty, kteří příliš neznají architekturu procesorů. Procesor je součást počítače, který má přístup k veškeré paměti počítače a umí vykonávat instrukce (operace), které nějakým způsobem manipulují s daty v paměti i s daty uloženými přechodně přímo uvnitř procesoru (v tzv. registrech). Paměť je rozdělena na jednotky, každá má svou adresu. Velikost jednotky souvisí s šíří datové sběrnice, která je 8, 16, 32 nebo nejčastěji 64 bitů. V paměti počítače jsou umístěna zpracovávaná data a také je zde jako posloupnost bajtů uložen program, který je tvořen instrukcemi. Každá instrukce se skládá jednak z instrukčního kódu, který určuje, co konkrétně má procesor provést a dále z dat, které jsou buď konstanty, se kterými program pracuje a nebo pointery (adresy v paměti) někam do paměti, kde se nachází data ke zpracování. Některé procesory mají k dispozici i takové instrukce, kdy daný pointer je pouze relativní offset, který se přičte k pointeru (adrese) uložené v některém registru procesoru. Pro doplnění uvádím, že existují procesory, které "znají" velké množství instrukcí, mnohdy velmi sofistikovaných, takže k provedení určitého

algoritmu je potřeba menší počet instrukcí (tyto procesory se označují CISC) a dále procesory, které mají jen velmi málo jednoduchých instrukcí (označují se RISC), takže to, co procesor typu CISC zvládne jedinou instrukcí, se musí na procesoru typu RISC realizovat několika instrukcemi. Tato zdánlivá nevýhoda je ale plně vyvážena tím, že procesor typu RISC umí své instrukce provádět podstatně rychleji, takže ve výsledku není zdaleka jisté, která z obou architektur je výkonnější. Pro nás je ale podstatné, že procesor musí při vykonávání programu provést určité elementární operace a jestli jich dosáhne jednou nebo více instrukcemi je nepodstatné.

Jedním z důležitých registrů procesoru je PC (program counter) - registr, který obsahuje pointer do paměti (tedy adresu), ze které se vezme nejbližší zpracovávaná instrukce. Takže např. pokud program v rámci nějakého cyklu má provést skok na začátek cyklu (tj. přejít na další iteraci), provede se to velmi jednoduše, prostě se do PC nahraje adresa začátku cyklu (v praxi to někdy bývá realizováno jako přičtení nějakého záporného offsetu k aktuálnímu stavu registru PC, říká se tomu relativní skok - kompilátor v takovém případě neřeší přesnou cílovou adresu, ale pouze kolik instrukcí se má přeskočit, což je jednodušší).

Dalším důležitým registrem je SP (stack pointer). Ten souvisí s vyhrazenou částí paměti, ve které je umístěn zásobník, což je datová struktura, která umožňuje dvě základní operace: přidej na zásobník nová data a odeber ze zásobníku data, která jsou na něm nejkratší dobu. SP tedy ukazuje na tzv. vrchol zásobníku. Tam kam ukazuje se ukládají nová data a přitom se s uložením dat posune o adresu dál. Při odebírání dat se postupuje obráceně. Odebraná data procesor uloží do některého z dalších (tzv. univerzálních) registrů, které jsou v procesoru k dispozici. Zásobník je velmi důležitá struktura, protože při vyvolávání podprogramu/funkce/metody se na něj uloží návratová adresa, kam se má běh programu vrátit po ukončení podprogramu/funkce/metody. Procesor proto zpravidla disponuje instrukcemi, které odskok včetně uložení návratové adresy (což je původní obsah procesoru PC) realizují v jednom kroku a podobně i návrat. Programy, které procesor vykonává, často na zásobník ukládají i svá data, a proto jsou k dispozici i instrukce, které uložení a vyzvednutí hodnoty ze zásobníku realizují. Kompilátory často při volání funkce/metody uloží na zásobník i parametry, které se funkci/metodě předávají a vyvolaná funkce si na zásobník ukládá své lokální proměnné. Část paměti určená pro zásobník má svou předem danou velikost. Pokud program na zásobník uloží příliš mnoho dat, paměť se vyčerpá, což v praxi znamená, že registr SP překročí hranici vyhrazené paměti a dochází ke stavu, který se nazývá “stack overflow” - ne náhodou byl podle tohoto stavu pojmenován známý server. Za zmínku ještě stojí, že ne každý procesor má vestavěné mechanismy, které tento stav detekují, takže na starších a nebo i současných jednodušších procesorech pak dochází k nedefinovaným stavům, většinou zatuhnutí celého systému.

Posledním registrem, který lze najít v určité podobě v každém procesoru je tzv. akumulátor (v české literatuře často označován jako střadač). Ten je určen ke všem výpočtům i dalším operacím s daty. S tímto registrem jsou svázány instrukce, které vezmou data z paměti a uloží do tohoto registru nebo naopak vezmou obsah tohoto registru a uloží do paměti a dále instrukce, které provádějí jednoduché početní operace (sčítání, odečítání, násobení, dělení, porovnání a jiné) s tímto registrem, přičemž akumulátor je jedním operandem této operace a data v paměti (na konkrétní adrese která je součástí instrukce nebo je uložena v některém dalším registru) nebo konstanta (která je součástí instrukce) je operandem druhým. Tento registr bývá označování často zkratkami A nebo AX, ale lze najít i mnoho jiných. Některé procesory mívají akumulátorů víc. Za zmínku stojí, že pokud má procesor sečíst dvě čísla v paměti uložené na konkrétních adresách, musí provést program obsahující tyto instrukce:

- načtení hodnoty z paměti do A (adresa je součástí instrukce)
- přičtení hodnoty z paměti do A (adresa je součástí instrukce)
- uložení hodnota v A do paměti (adresa je součástí instrukce)

K dispozici bývají i další registry, které slouží buď pouze ke krátkodobému uložení dat a nebo jsou k nim k dispozici instrukce, které tyto registry využívají jako pointery do paměti.

Ještě malá poznámka: označení registrů A, PC a SP není ustálené a v podstatě každý výrobce procesoru si tyto názvy stanoví po svém. Zde v tomto textu budeme nadále používat toto označení.

Nyní se již vraťme k typům paměti a jak s nimi zachází kompilovaný program.

- **typy paměti používané programem pro ukládání dat (statická paměť, zásobník, halda)**
  - **lokální proměnné (zásobník), to samé platí i pro parametry funkcí/metod**
    adresa vrcholu zásobníku + offset
    Veškeré lokální proměnné jsou umístěny na zásobníku (určitá výjimka platí pro malé levné jednočipové procesory, ale tou se nebudeme podrobněji zabývat). Díky umístění na zásobníku je vyhrazení prostoru pro lokální proměnné při vstupu do

funkce/metody jednorázovou operací, při které se jednoduše posune ukazatel na vrchol zásobníku o takový počet bajtů, kolik je potřeba pro všechny lokální proměnné dané funkce/metody. Zde se zpravidla vyhradí prostor pro úplně všechny, tj. nejen ty, které se definují na začátku těla funkce/metody, ale i ve vnořených blocích (v cyklu apod.). Uvolnění paměti na konci provádění funkce/metody se provede obdobnou (opačnou) operací. Přístup k těmto datům se realizuje tak, že se musí spočítat adresa těchto dat a teprve potom se data mohou vyzvednout (převzít do registru procesoru) a nebo zapsat. Proto v následujícím textu si budeme vždy popisovat "zpřístupnění hodnoty", pod kterým budeme chápat výpočet adresy a nebudeme řešit, jestli se následně do paměti na vypočtené adrese hodnota uloží a nebo naopak se odtud převezme. Samotný výpočet adresy je u lokálních proměnných a parametrů funkcí jednoduchý, protože kompilátor při překladu přesně ví, co kde na zásobníku v rámci dané funkce/metody má, takže vezme jednoduše adresu vrcholu zásobníku (registr SP) a k němu přičte offset (počet bajtů), který je rovněž možné určit v době překladu.
Je důležité zmínit, že parametry funkcí/metod se umisťují také na zásobník. Volající funkce je tam sama umístí, pak dojde k vyvolání volané funkce (včetně uložení návratové hodnoty na zásobník) a ta si následně ještě na zásobníku vyhradí prostor pro své lokální proměnné. Dodejme, že aby toto mohlo fungovat, musí kompilátor při kompilaci volané i volající funkce znát počet parametrů a jejich typy. Je to složitější o to, že kompilace může probíhat nezávisle a to dokonce různými kompilátory z různých programovacích jazyků. Např. v jazyce C/C++ se toto řeší hlavičkovými soubory obsahujícími hlavičky funkcí/metod. V jiných jazycích se to řeší různě a můžete přemýšlet jak je to ve vašem oblíbeném programovacím jazyku. Zde samozřejmě mluvíme pouze o kompilovaných jazycích. U těch interpretovaných nemá smysl mluvit o efektivitě, protože se řada věcí řeší až za běhu.

Na tomto místě si řekněme, že pokud daný programovací jazyk disponuje možností, že velikost dat (počet bajtů, které v paměti zabírají) nelze v době překladu určit - tj. velikost a případně počet je znám až za běhu programu, nelze tento jednoduchý postup pro zpřístupnění dat použít, a proto program přeložený z takového programovacího jazyka už v principu nemůže být dostatečně efektivní a pro implementaci některých algoritmů se prostě nehodí. To samé platí i pro parametry funkcí. Pokud není pro každou funkci přesně specifikováno, jaké má parametry a při kompilaci je tato informace k dispozici, řeší se toto až za běhu a zpracování je nutně o to pomalejší. Určitou výjimku z tohoto pravidla tvoří v jazyce C funkce, s volitelnými parametry na konci (v hlavičce funkce se toto specifikuje pomocí tří teček), protože o umístění takových parametrů se postará volající funkce, která "ví" (ve skutečnosti to ví kompilátor), kolik jich předala (přesněji řečeno kolik bajtů předala), takže může i snadno po návratu paměť zabranou těmito parametry (na zásobníku samozřejmě) jednoduše uvolnit. O zpřístupnění hodnot těchto parametrů ve volané funkci se musí postarat algoritmus, který napíše programátor ve vyšším programovacím jazyku, ale to zpravidla není nic složitého - pouze vhodné operace s pointery.

Za zmínku stojí i to, že pokud programovací jazyk definuje iniciální hodnotu lokální proměnné (typicky že lokální proměnná obsahuje na začátku vždy nulu), musí kompilátor na začátek každé funkce/metody zařadit kód, který se o to postará, což může být na úkor efektivity, protože nulování může být zbytečné, pokud se tam při běhu programu následně dostanou jiné hodnoty.

- **globální proměnné (statická paměť)**
  adresa ve statické paměti
  Ne každý programovací jazyk disponuje možností definovat globální a statické proměnné. Např. v C/C++ to lze. Tyto proměnné vznikají (tj. prostor pro ně se alokuje a proměnné se inicializují počáteční hodnotou) v okamžiku spuštění programu, existují po celou dobu běhu a mají stále stejnou adresu v paměti a tuto adresu určí kompilátor. Proto operace s těmito daty jsou jednoduché, kompilátor zakompiluje instrukce, jejichž součástí je adresa dat. Oblast paměti, kam se globální a statické proměnné ukládají se nazývá statická paměť, ale u běžných procesorů jde jednoduše o část paměti vyhrazenou v rámci běhu daného programu k tomuto účelu podobně jako jiná část je vyhrazena pro zásobník a jiná pro heap (viz dále).

  existují programy pouze s glob. proměnnými (bez lokálních a bez parametrů funkcí)
  Jen malá poznámka na okraj: protože operace s globálními proměnnými jsou rychlé, tak se u algoritmů extrémně náročných na výkon přistupuje k tomu, že se nepoužívají parametry funkcí a veškerá data mezi volanou a volající funkcí se sdílí v globálních proměnných. To je samozřejmě obrovský prohřešek proti přehlednosti programu, ale to se zde chápe jako daň za vyšší výkon. Konec konců u některých extrémně levných malých osmibitových procesorů má zásobník doslova jen pár bajtů, takže kompilátor musí tak jako tak parametry funkci a lokální proměnné umisťovat do statické oblasti, což např. u jazyka C není s ohledem na rekurzi vůbec jednoduché. Kompilátory pro tyto procesory proto musí provádět poměrně komplikovanou analýzu zdrojových textů a vzájemného volání funkcí.

- **registrové proměnné (procesor)**
  Některé procesory mívají kromě střadače k dispozici i několik dalších registrů, které jsou autorovi programu plně k dispozici (my, protože nepíšeme program přímo v instrukčních kódech procesoru, ale ve vyšším programovacím jazyku, který je kompilátorem přeložen do instrukcí procesoru, bychom spíš měli říkat, že tyto registry jsou k dispozici kompilátoru). Konkrétně v jazyce C je možné nadefinovat, že programátor si přeje některou proměnnou umístit do registru procesoru. Kompilátor toto buď vyhodnotí jako reálné a vyhoví mu a nebo jako nereálné a v tom případě s proměnnou zachází jako s každou jinou. V každém případě ale pokud je proměnná registrová, nelze např. získat pointer na tuto proměnnou (protože pointery jsou vlastně adresy v paměti a takováto proměnná není umístěna v paměti). V jiných (pozdějších) jazycích tato možnost programátorovi dána není a je spíše na kompilátoru, aby v rámci svých vestavěných optimalizaci detekoval, která data bude nejvýhodnější umístit do registru procesoru. Jinými slovy registrovými proměnnými nemá smysl se zde zabývat a zmiňuji je zde jen pro úplnost

- **halda (heap) (spousta různých implementací, většinou pomocí zřetězených seznamů)**

  Halda je velice složitý mechanismus uložení dat, bez kterého se žádný moderní programovací jazyk neobejde. U haldy rozlišujeme dvě základní operace: alokace prostoru na haldě a uvolnění alokovaného prostoru. Tyto operace jsou časově velmi náročné, a proto pokud algoritmus dovolí zapsat program bez použití haldy (tj. data jen v lokálních a globálních proměnných), může být program podstatně efektivnější.

Mnohdy to ale nejde. K operacím s haldou slouží aparát, který je součástí přeloženého programu. Tyto operace jsou vyvolávány buď pomocí konstrukcí programovacího jazyka (new, delete apod.) a nebo voláním funkcí standardní knihovny (v jazyce C např. známé funkce `malloc()` a `free()`).

- **objekty na haldě jsou referencovány pointery**

  jakmile je na haldě vyhrazen prostor, je programu vrácen pointer na tento prostor a s ním potom přeložený program pracuje dál. To znamená, že to, co programátor ve vyšším programovacím jazyku vnímá jako proměnnou obsahující nějaká data, je ve skutečnosti proměnná obsahující pointer, přes který se k těmto datům přistupuje. To potom znamená, že každý přístup k datům je komplikovanější o jeden krok a to získání adresy z pointeru. Ale to je zpravidla jednoduchá instrukce procesoru, takže tento krok efektivitu programu příliš nesníží.

- **operace alokování prostoru (včetně vyhledání ideálního volného prostoru)**

  součástí haldy musí být určitá datová struktura, která popisuje volné a obsazené oblasti haldy, aby bylo možné jasně říct, která oblast je k dispozici a která ne. V okamžiku, kdy je aparát haldy požádán o vyhrazení prostoru dané velikosti, musí aparát projít tuto datovou strukturu a vybrat prostor a současně tuto strukturu upravit tak, aby bylo jasné, že vybraný prostor již je obsazen. Tato popisná datová struktura by se dala realizovat pomocí tabulky pointerů s příznaky volno/obsazeno, ale taková tabulka by měla konečnou velikost a to by mohl být problém. Proto se volí strategie, při které taková tabulka nevzniká přímo, ale každá alokovaná oblast se zvětší o pár bajtů, do kterých se umístí informace o velikosti a příznak volno/obsazeno a případně pointery na předchozí a následující oblast (služební údaje oblasti haldy).

  Důležité je také si uvědomit, že není vhodné při alokaci prostoru vybrat první volnou oblast větší než je požadovaný prostor, rozdělit ji na dvě části, první označit jako obsazenou a druhou jako volnou. Tím by vznikalo poměrně dost děr, které by sice v součtu tvořily velkou část haldy, ale dostatečně velký souvislý prostor by nemusel být k dispozici. Proto že volí strategie, že se alokovaný prostor neodebere z první volné oblasti, ale najde se nejmenší taková, která je větší než požadovaný prostor.

- **operace uvolnění prostoru (včetně scelování)**

  uvolnění prostoru neznamená jen změnu příznaku volno/obsazeno. Je potřeba též zkontrolovat jestli některá ze sousedních oblastí není také volná a pokud ano, tak ji propojit (sloučit, scelit) s uvolňovanou oblastí. To také stojí nějaký čas.

- **operace „setřepání“ - nelze v každém prog. Jazyce**

  pokud dojde k tzv. fragmentaci haldy, kdy je činností programu (neustálým alokováním a uvolňováním prostoru na haldě) dosaženo stavu, že je alokováno velké množství menších oblastí, mezi kterými je velké množství

volných oblastí, může nastat problém, že oblast větší velikosti již není možné alokovat, protože tak velká oblast již neexistuje, ale přitom v součtu je na haldě dostatek místa. Některé programovací jazyky tento problém řeší občasnou defragmentací (setřepáním) haldy, kdy se jednotlivé obsazené oblastí přesouvají (tj. dochází ke kopírování dat v paměti na jiné místo). Tato operace je samozřejmě časově velice náročná, ale hlavně u řady programovacích jazyků nemožná, protože při operaci se mění adresa každé přesunuté oblasti a přitom aparát haldy nemůže vědět, kde všude má program pointery na tyto oblasti uloženy, aby je v rámci setřepání aktualizoval. Takže např. v C/C++ se halda nesetřepává. Např. v Javě je mechanismus setřepání řešen podstatně složitější organizací haldy, která je rozdělena na dvě základní části (mladá a stará generace), tyto mají své další podoblasti (oblasti přežití a eden) a k přesunu dat mezi jednotlivými oblastmi zde skutečně dochází, viz dále.

- **některé jazyky hlídají, jestli je prostor na haldě referencován**

  je dobré vědět, že v jazycích, ve kterých se alokovaný prostor nemusí uvolňovat, musí být tato záležitost zajištěna jiným mechanismem.
  - **počítadlo referencí**
    Jednou z možností je, že každá oblast na haldě obsahuje ve svých služebních údajích ještě počítadlo referencí. Kompilátor musí zajistit, že jakmile je pointer umístěn do jiné proměnné nebo je předán jako parametr do funkce, musí se počítadlo zvýšit a jakmile proměnná nebo parametr funkce zanikají, musí se počítadlo snížit. Jakmile je počítadlo na nule, je oblast k uvolnění.
  - **mark and sweep**
    Aparát haldy v určitých okamžicích "zastaví" běh program, označí všechny objekty na haldě jako *nenavštívené*, poté prochází všechny proměnné od nejvyšší úrovně zásobníku a (rekurzí) následuje všechny pointery a prochází objekty, na které ukazují. Pokud se objekt nachází na haldě, označí jej jako *navštívený*. Poté, co projde všechny reference, tak smaže všechny *nenavštívené* objekty.
  - **kopírovací algoritmus**
    má rozdělenou haldu na dvě části, vždy plní jen jednu, kopíruje! referencované objekty do druhé části a upraví všechny reference na ně. Tím se současně řeší defragmentace, ale nevýhodou je větší paměťová náročnost a zpomalení způsobené kopírováním a podmínkou je možnost dostat se ke všem pointerům, které referencují objekty na haldě
  - **generační algoritmus**
    má rovněž haldu rozdělenou na dvě (a více) částí, objekty vytváří v části pro nové objekty a sleduje dobu existence každého z nich a po určité době je přesunuje do části pro staré objekty.

- **garbage collector (hledá nereferencované objekty na haldě, případně volá destruktor)**

  garbage collecor souvisí s předchozím mechanismem hlídání referencí oblastí na haldě. Jedná se o samostatný proces běžící ve vlastním vlákně (s nízkou prioritou), který vyhledává oblasti na haldě, které již nejsou referencovány,

případně volá destruktor, pokud se jedná o oblast, která byla alokována pro umístění instance nějaké třídy (a pokud programovací jazyk destruktory disponuje) a následně oblast nechá aparátem haldy uvolnit. Garbage collector tedy neuvolňuje alokované oblasti hned, ale později, ideálně např. když hlavní vlákno je kvůli vstupně/výstupní operaci pozastaveno.

- **některé jazyky se bez haldy neobejdou (Java): výkonnost**

  zatímco v C++ lze programovat tak, abychom se využití haldy vyhnuli (tj. všechny instance objektů jsou umístěny v lokálních a globálních proměnných, v Javě toto možné není. Pokud je instance třídy jen lokální, je v lokální proměnné umístěn jen pointer na instanci a tato instance sama je umístěna na haldě. Na druhé straně ovšem Java používá řadu pokročilých optimalizačních mechanismů, které tuto nevýhodu snižují, ale nikoli odstraňují. Další pokročilé mechanismy jsou potom v Javě EE. V každém případě ale je potřeba toto mít na vědomí při volbě programovacího jazyka.

- **složené datové typy**
  - **jedno- a vícerozměrné pole**
    velikost známá při překladu – umístění (zásobník x halda)
    adresa pole + index * velikost prvku
    adresa pole + index1 * velikost řádku + index2 * velikost prvku
    Přístup k jednotlivým prvkům pole v zásadě není nic složitého, pokud je v době překladu známa velikost jednotlivých prvků pole (tj. počet bajtů, které zabírá prvek pole). V případě vícerozměrných polí musí být znám počet prvků ve všech rozměrech kromě toho, který je indexován indexem na nejvyšší úrovni (tj. např. u dvourozměrného pole nemusí být znám počet řádků pole, ale musí být znám počet sloupců). V takovém případě může být pole uloženo celé v jednom souvislém bloku paměti postupně po jednotlivých indexech (tj. např. u dvourozměrného pole po řádcích). Adresa konkrétního prvku pole se pak získá tak, že se vezme adresa pole jako takového (ta se získá v závislosti na tom, jestli je pole lokální proměnnou nebo např. někde na haldě - viz dále). K této adrese se přičte offset (počet bajtů) v rámci tohoto pole podle vzorečku uvedeného výše.

    Např. pokud máme v lokální proměnné **a** dvourozměrné pole o 10 sloupcích čísel typu 32-bitový integer, dostaneme se k prvku a[5,8] takto:
    SP + <*offset na zásobníku*> + 5 * 10 * 4 + 8 * 4
    vše samozřejmě za předpokladu, že prvky pole jsou indexovány od 0.
    Zde v tomto příkladu vidíme, že vše za "SP + " jsou konstanty známé v době kompilace, takže kompilátor může veškeré násobení i sčítání provést už v době kompilace a zakompiluje pouze SP + <výsledek výpočtu>. Z toho plyne, že zpřístupnění prvku pole, jehož indexy jsou konstanty známé v době kompilace, je stejně náročné jako zpřístupnění obyčejné integerové lokální proměnné.
    Pokud by byla situace komplikovanější v tom, že oba indexy budou v lokálních proměnných **i** a **j**, musí se nejprve zakompilovat zpřístupnění jejich hodnot, tj. instrukce, kterými se získají postupně jejich adresy a následně instrukce, které získají hodnotu v paměti uloženou na této adrese. Zpřístupnění prvku pole a[i, j] se pak provede takto:
    SP + <*offset na zásobníku*> + <*hodnota proměnné i*> * 10 * 4 +
    + <*hodnota proměnné j*> * 4

Zde vidíme, že násobení 10 * 4 může ještě provést kompilátor, ale pro ostatní násobení a sčítání musí kompilátor zakompilovat instrukce, které tyto operace provedou až za běhu programu.

Obecně k polím bych dodal, že zpřístupnění hodnoty prvku pole je složitější než zpřístupnění hodnoty v samostatné proměnné, ale vidíme, že je to “jen pár” instrukcí. Samozřejmě, že u programu s extrémním nárokem na výpočet má smysl hledat invariantní kód v cyklu i v takových detailech, jako je prvek pole a pokud se v cyklu pracuje např. s a[ind1, ind2] a přitom ind1 a ind2 se v cyklu nemění, má smysl si před cyklem uložit tento prvek do vlastní lokální proměnné a v cyklu pak pracovat s ní. Toto samozřejmě zejména v případě, kdy nemáme dostatek důvěry k optimalizacím vestavěným do kompilátoru.

- **struktury, třídy, volání metod, virtuální metody**
  ***adresa struktury +** __offset__*
  Prakticky každý “rozumný” vyšší programovací jazyk umožňuje definovat datové struktury skládající se z jednotlivých prvků struktury. Každý prvek má na úrovni programovacího jazyka svůj název a svůj typ. Z typu kompilátor odvodí počet bajtů, který tento prvek zabírá v paměti. Jednotlivé prvky struktury jsou uloženy v paměti za sebou, někdy z důvodu určitých optimalizaci jsou mezi nimi ještě vložené nevyužité bajty, díky kterým každý prvek struktury začíná na adrese, která je např. dělitelná čtyřmi. Objektové jazyky ukládají instance tříd podobným způsobem jako popsané datové struktury, pouze k nim přidají určité další “služební informace” jako je odkaz na tabulku tzv. virtuálních metod (to je pojem z jazyka C++, nicméně např. Java chápe všechny metody tříd v tomto smyslu jako virtuální). Virtuální metody jsou metody, jejichž adresa není známa v době kompilace, protože pokud je instance třídy referencována pointerem (v javě vždy), je konkrétní třída této instance známa až za běhu, a proto instance musí obsahovat pointer na tabulku obsahující pointery na jednotlivé metody této třídy. Proto je vyvolání metody třídy nepatrně složitější než zavolání funkce, která není metodou žádné třídy (takové funkce java nezná). Musí se totiž v jednom kroku jít přes pointer na instanci třídy do této instance a zde vzít pointer na tabulku virtuálních metod a v ní si vzít adresu metody a tu teprve zavolat. V rámci tohoto volání se musí kromě parametrů této metody umístit na zásobník i hodnota “this”, tedy pointer na instanci.

  Nyní si popišme, jak se získá (zpřístupní) konkrétní prvek struktury. K tomu samozřejmě potřebujeme znát adresu, kde je v paměti uložena celá struktura (nebo instance třídy), což záleží na tom, jestli je tato struktura lokální proměnou nebo statickou proměnnou nebo je referencována pointerem (viz dále), což se provede přesně jak je popsáno na příslušném místě této kapitoly. Protože umístění jednotlivých prvků struktury určuje kompilátor, je v době kompilace znám offset každého jednotlivého prvku (offset je počet bajtů od začátku struktury k danému prvku). Kompilátor tedy zakompiluje pouhé přičtení tohoto offsetu k adrese struktury.
  Např. mějme strukturu uloženou v proměnné __**s**__ a obsahující mimo jiné prvek __**p**__, který se ve struktuře nachází 8 bajtů za jejím začátkem. Získání adresy s.p pak se provede tímto jednoduchým postupem:
  <*adresa struktury s*> + 8
  Nyní nechť je prvek __**p**__ polem typu integer (nechť je integer zde čtyřbajtový) a chceme získat hodnotu s.p[2]. Kompilátor zakompiluje:
  <*adresa struktury s*> + 8 + 2 * 4

Zde si opět můžeme všimnout, že vše za prvním sčítáním je konstanta, kterou může vypočítat kompilátor. Pochopitelně pouze za předpokladu, že index v poli je konstanta.
Pokud by dokonce proměnná <u>**s**</u> byla lokální, zpřístupnila by se hodnota s.p[2] takto:
SP + <*offset struktury s*> + 8 + 2 * 4
a zde vidíme, že vše za SP + je konstanta, kterou může vypočítat kompilátor a že tedy zpřístupnění hodnoty s.p[2] je stejně náročné jako zpřístupnění obyčejné lokální proměnné typu integer.

- **pointer**
Pointery jsou prakticky ve všech programovacích jazycích důležitým mechanismem. I když řada jazyků zdánlivě pointery nedisponuje (java), pracuje se v nich s pointery skrytě a to velmi často. Programátor dokáže poznat, kde všude kompilátor musí práci s pointerem zakompilovat. Zjednodušeně lze říci, že přes pointer se přistupuje k veškerým datům na haldě a dále datům, která jsou fyzicky umístěna jinde, než v proměnné, která tato data reprezentuje. V javě se jedná o všechny instance tříd. Ale i v jiných jazycích, jakmile máme jakýkoli datový objekt (instanci třídy, pole, strukturu apod.) umístěnou v jedné proměnné jejím přiřazením do jiné proměnné respektive předáním tohoto datového objektu do funkce/metody nedojde k duplikování těchto dat (tj. případná změna těchto dat je dostupná přes obě proměnné resp. ve volané i volající funkci/metodě), je jasné, že tato data jsou referencována pointery a kompilátor musí zakompilovat instrukce, které operace s pointery zajistí. Postup je ale velmi jednoduchý. Nejprve se musí získat adresa, kde je v paměti uložen pointer, následně odtud převzít hodnotu (ten pointer) a s takto získanou adresou dále pracovat. Jedná se tedy o jednoduchý krok navíc, který je u řady procesorů proveditelný jedinou instrukcí, ale tento krok znemožňuje v některých výše uvedených příkladech zjednodušení zpřístupnění dat tím, že se některé výpočty adres provedou již v době kompilace.
Mějme např. lokální proměnou <u>**s**</u>, která bude pointerem na strukturu s prvkem p, který jsme měli i ve výše uvedeném příkladu u struktur. Potom s->p[2] se zpřístupní takto: (použitá syntaxe je z C/C++, protože v javě by se použila tečka místo šipky)
[SP + <*offset struktury s*>] + 8 + 2 * 4
Zde hranatými závorkami zapisuji operaci “z adresy, která je v závorkách vyzvedni hodnotu a v dalších operacích ji považuj za adresu”.

- **speciální třídy (Vector, ArrayList, HashMap, Hashtable, String)**
pole polí, hashovací metoda (objekt -> index)

Zde máme na mysli třídy z jazyka Java, které slouží k uchování dat. Podobné třídy lze nalézt i v C++ a jiných objektových jazycích. Je potřeba si uvědomit, že tyto třídy slouží k uchování dat předem neznámého rozsahu, takže v principu nemohou být uloženy jinde, než na haldě (viz dále) a nemohou být tedy zcela efektivní. Rozhodně efektivnější práce je s polem, takže pokud to algoritmus dovolí, je vhodnější preferovat pole. Konec konců třídy, o kterých je řeč, používají samy pole jako prostředek k uložení dat, pouze jsou nějak komplikovaně organizovány (data jsou rozložena do více polí), aby umožnily dosáhnout požadované funkčnosti. Z výše uvedených tříd je samozřejmě nejsložitější hashovací mapa, ale přesto se její užití může vyplatit, protože rychlost zpřístupnění konkrétní hodnoty umístěné v hashovací mapě může být mnohonásobně vyšší než kdyby ta samá data byla uložena v nějakém poli, které by se muselo sekvenčně procházet, dokud by se hodnota nenašla.
Podrobnější popis těchto tříd už je mimo rámec předmětu, nám postačí, pokud máme

představu o tom, jak jsou realizovány a dáme to do souvislosti s operacemi na haldě.

- **typ množina**

  množina je užitečný datový typ, který usnadní implementaci řady algoritmů, ale má svá úskalí

  - **s výčtem prvků známým při kompilaci**

    pokud se jedná o množinu, jejíž výčet prvků je znám při komplilaci (viz set v Pascalu), pak je tato datová struktura vnitřně implementována pomocí bitové mapy, kde každý bit obsahuje informaci o existenci jemu přidělenému prvku množiny. Bitová mapa je samozřejmě realizována jako pole bajtů. Pokud tedy definiční obor množiny obsahuje n prvků, zabírá tato množina v paměti n / 8 bajtů a to může být potom velmi náročné, při operacích s takovou proměnnou, např. při uložení hodnoty typu množina z jedné proměnné do jiné nebo předání jako parametr do funkce se musí kopírovat značné množství bajtů. Naopak ale operace umístění prvku do množiny, odebrání z množiny nebo zjištění jeho existence v množině jsou velmi jednoduché operace. Z důvodu prostoru zabraného proměnnou typu set Pascal omezuje definiční obor množiny na 256 prvků, což znamená, že množina může zabírat v paměti 32 bajtů, ale i taj operace s takovou proměnnou mohou být časově náročnější.

  - **dynamická (HashMap)**

    tato implementace je k dispozici v řadě objektových programovacích jazyků. Na jedné straně nemá omezen definiční obor, může obsahovat např. stringy nebo jiné objekty, ale na druhé straně se neobejde bez užití haldy a operace s ní jsou časově podstatně náročnější. Dále k tomuto datovému typu viz výše.

10. Implementace programových struktur

- **mechanismus volání funkce**

Co se všechno děje v okamžiku, kdy jedna funkce (či metoda - pokud se bavíme v pojmech objektového programování) volá druhou, je důležité vědět. Tento mechanismus není složitý, aby nás nutil v rámci úspory paměti a času zpracování psát tělo volané funkce přímo do volající funkce, ale ani není zcela triviální, takže v případě extrémních nároků na dobu zpracování musíme přemýšlet, kde je volání funkce rozumné a kde se může vyplatit místo volání funkce opsat její tělo, což je samozřejmě v rozporu s požadavkem na správnou strukturovanost programu a jeho čitelnost a přehlednost. Zkrátka je potřeba mít představu “kolik to stojí” a vždy být schopen uvážit, jestli to “za to stojí”.

Neopominutelným aspektem je rekurzivní volání funkcí, kdy k uplatnění mechanismu volání funkce dochází ve velkém měřítku. Jsou samozřejmě algoritmy, které jsou na rekurzi

postaveny a jejich přepracování na nerekurzivní je problematické a které v praxi zpravidla ve své rekurzivní podobě zůstanou. Na druhé straně je řada algoritmů, které se dají jednoduše převést na cyklus, příkladem nechť je notoricky známý výpočet faktoriálu. U těchto algoritmů je použití rekurze zbytečný luxus.

Nyní konkrétně k samotnému mechanismu volání funkce. Nechť je náš program ve stavu, kdy jedna funkce (nazývejme ji volající funkce) zavolá druhou (tu nazývejme volaná funkce). Víme, že volající funkce musí mít možnost předat volané funkci data - parametry funkce a volající funkce musí mít možnost předat návratovou hodnotu. V některých programovacích jazycích pak volaná funkce může mít přístup k proměnným volající funkce.

Volající funkce musí nejprve připravit parametry pro volanou funkci. Ty umístí na zásobník. Případně může na zásobníku vytvořit prostor, kam volaná funkce později umístí návratovou hodnotu. Často se ale návratová hodnota předává v registru procesoru. Toto je věcí kompilátoru volající a volané funkce, aby mechanismus předání návratové hodnoty odpovídal. Následně dojde k vyvolání funkce, což zajistí instrukce procesoru (případně krátká sekvence instrukcí), která umístí na zásobník návratovou adresu (tj. adresu v přeloženém kódu volající funkce, kde byl běh této funkce voláním volané funkce přerušen) a do registru PC (pointer na právě prováděné instrukce programu) umístí adresu volané funkce. Tím dojde k jejímu vyvolání. Ta musí nejprve vyhradit prostor na zásobníku pro své lokální proměnné. Pokud je daný programovací jazyk navržen tak, aby byla v době kompilace známa velikost prostoru pro lokální proměnné, je to jen jednoduchá operace, u některých procesorů dokonce jediná instrukce - jednoduše se posune vrchol zásobníku o příslušný počet bajtů dále. Pokud tato velikost v době kompilace nemůže být známa, musí být zakompilován programový kód, který tuto velikost zjistí a zajistí alokaci prostoru pro lokální proměnné. To už je samozřejmě na úkor efektivity. Případně samozřejmě může být tento prostor alokován na haldě, ale to je taky značně na úkor efektivity. Poté může činnost volané funkce začít.

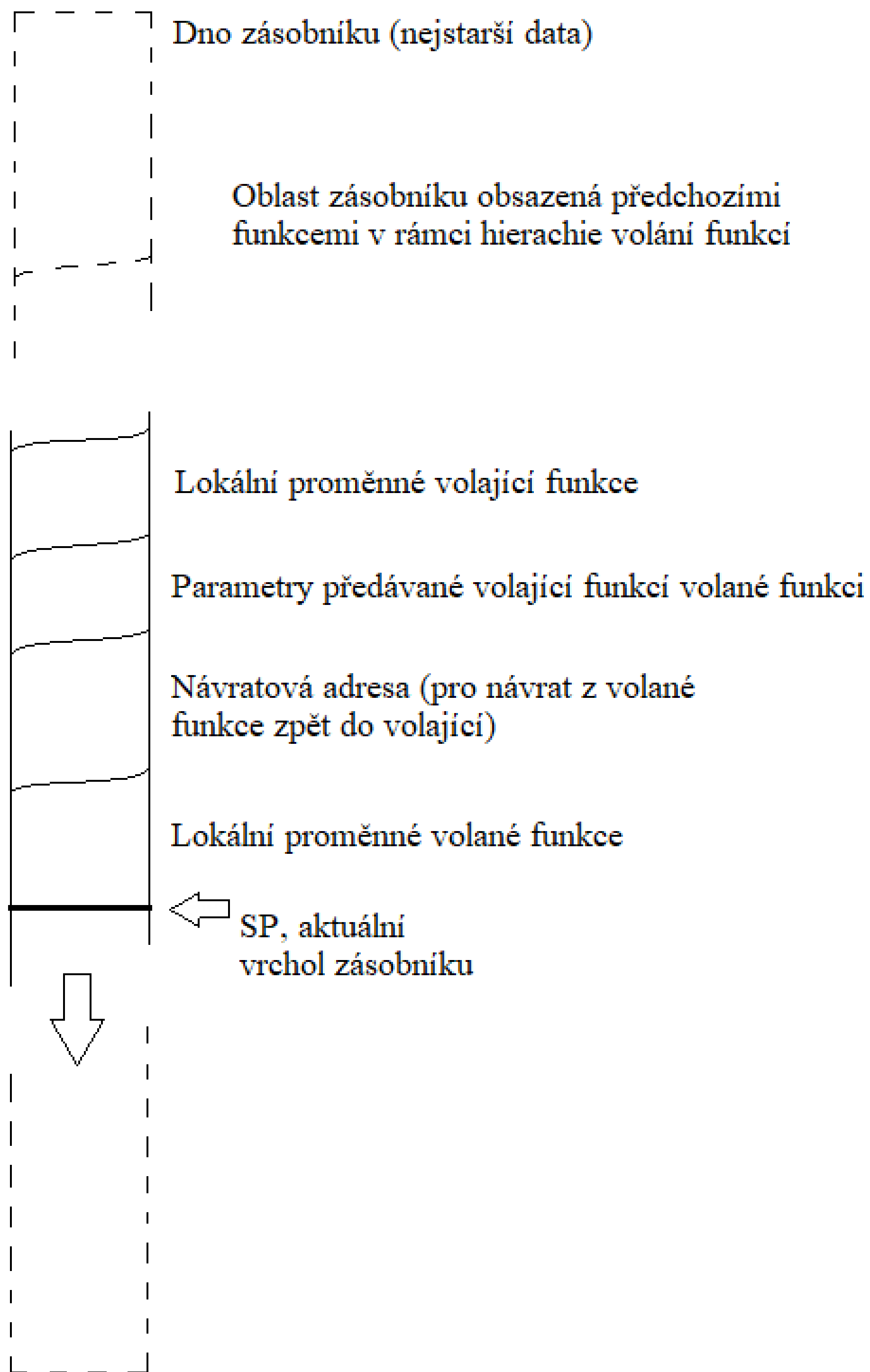
Volaná funkce přistupuje ke všem svým lokálním proměnným i parametrům tak, že vezme vrchol zásobníku (registr SP), přičte k němu určitý offset, který je znám v době kompilace a tím se dostane k požadované hodnotě. Podrobněji viz výše v odstavci o lokálních proměnných. Pokud programovací jazyk dovoluje volané funkci i přístup k lokálním proměnným z volající funkce, musí být kompilátor schopen zjistit příslušené offsety od vrcholu zásobníku, což předpokládá společnou kompilaci obou funkcí.

Poznámka: norma jazyka C99 zavádí možnost, aby lokální proměnná typu pole mohla mít velikost, která se určí až za běhu. Realizováno je to tak, že proměnná sama je realizována pointrem (a ten má velikost známou při kompilaci) a prostor se vyhradí též na zásobníku, ale až za všemi ostatními lokálními proměnnými (tj. stack pointer SP se posune jednou pro všechny lokální proměnné a pak ještě extra pro každé takové pole) a také se buď ke všem referencím lokálních proměnných musí přičítat další konstanta, která je ale známa až za běhu a nebo se lokální proměnné referencují nějakým dalším registrem procesoru, zatímco SP plní jen funkci ukazatele na vrchol zásobníku. Na tomto příkladu vidíme, jak se tvůrci programovacích jazyků snaží vyvážit možnosti, které programátorovi jazyk poskytne s tím, jak je to celé efektivní.

Ukončení volané funkce a návrat do volající je opačný proces. Pokud má volaná funkce předat zpět nějakou hodnotu, zanechá ji buď v některém registru procesoru a nebo ji umístí na zásobník do prostoru, který nachystala volající funkce. Volaná funkce nyní uvolní prostor pro lokální proměnné (pokud jsou na zásobníku, stačí jen posunout vrchol zásobníku zpět o

tolik bajtů, o kolik byl posunut při vstupu do funkce) a poté provede instrukci, která vezme návratovou adresu ze zásobníku a umístí ji do registru PC. Tím pokračuje v běhu volající funkce, která ještě musí ze zásobníku odstranit parametry předané volané funkci a převezme si návratovou hodnotu vrácenou volanou funkcí (buď ze zásobníku nebo z registru procesoru). Některé procesory mívají pro podporu tohoto procesu jedinou instrukci.

Následující obrázek ilustruje část zásobníku v rámci volání jedné funkce. Zásobník na tomto obrázku roste směrem dolů (což je i častý případ v praxi, kdy dno zásobníku je na vyšších adresách a vrchol zásobníku na nižších.



(Toto schéma je podrobněji popsáno v jiném dokumentu.)

- **for-cyklus**

Zde si uveďme, jakým způsobem zkompiluje kompilátor klasický for-cyklus, který se vyskytuje v programech velmi často.

```
for (I = 0; I < 10; I++) {opakovaný kód }

Zkompiluje se jako:
I = 0
začátek cyklu:
if (I >= 10) goto konec cyklu

opakovaný kód

I++
Goto začátek cyklu
Konec cyklu:
```

Z uvedeného je vidět, je klasický for-cyklus je implementován velmi jednoduše a přímočaře a nemá cenu přemýšlet nad tím, jestli je jeho užití dostatečně efektivní.

- **vícenásobné větvení (switch)**

Vícenásobné větvení už určitý prostor pro efektivnější implementaci přináší, ale pouze v případě, že jednotlivé větve jsou definovány jednoduchou hodnotou známou při kompilaci. Jakmile se jedná o stringy nebo hodnoty neznámé při kompilaci, nelze vícenásobné větvení implementovat promocí sekvence if - else if - else ... if - else. Ideální prostor pro optimalizaci na straně kompilátoru je, pokud datovým typem vícenásobného větvení je integer nebo enumerace (která stejně každý prvek nahrazuje interním číselným kódem. Pak mohou nastat tyto možnosti:

  - **po sobě jdoucí hodnoty: lookup table adres**

  pokud větve jsou uvozeny po sobě jdoucími hodnotami, ideálně bez mezer nebo jen malým množstvím malých mezer, může kompilátor zakompilovat do přeloženého programu tabulku adres jednotlivých větví a jednoduchý výpočet, který z hodnoty, podle které se provádí rozvětvení (hodnota ve switch), jednoduše vypočte pozici v tabulce a odtud vezme adresu a provede skok na tuto adresu

  - **jinak lookup table hodnot + adres**

  pokud je řada hodnot uvozující větve více nesouvislá, vypadá tabulka tak, že neobsahuje jen adresy, ale pro každou adresu i hodnotu. Kompilátor pak zakompiluje kód, který v cyklu tuto tabulku projde a najde v ní hodnotu, vezme adresu u ní uloženou a tam skočí

Poznámka: nezabývali jsme se obyčejným if - else příkazem, ale to proto, že na něm je nejsložitější kompilace výrazu s podmínkou. Následně skok do jedné větve nebo do druhé

plus přeskočení druhé větve na konci první větve jsou vždy jediná instrukce procesoru, která změní hodnotu PC registru.

11. Rozdíl v interpretovaných a překládaných jazycích

Předchozí dvě kapitoly souvisely víceméně s jazyky kompilovanými a zabývaly se hodně tím, jak kompilátor zdrojový text ve vyšším programovacím jazyku překládá. Důležitou součástí moderní informatiky jsou i programovací jazyky, které se nekompilují. To znamená, že ke spuštění je určen přímo zdrojový text ve vyšším programovacím jazyku. Procesor samozřejmě není schopen přímo zpracovávat takový program, takže musí být k dispozici ještě jeden program - interpretter, který program ve vyšším programovacím jazyku převezme a postupně jej interpretuje a provádí.

To přináší určité výhody interpretovaných jazyků, které si v dalším popíšeme, ale ty jsou vyváženy značnou neefektivitou, protože analýza zdrojového textu a další činnosti, které u kompilovaných jazyků provádí kompilátor, nemohou být provedeny předem. Při výběru programovacího jazyka a rozhodování mezi jazykem kompilovaným a interpretovaným je požadavek efektivitu hlavním rozhodovacím kriteriem.

Typickými příklady interpretovaných jazyků jsou Javascript, PHP, Python, Perl apod.

Pokud se chystám psát webovou aplikaci a vím, že s ní budou pracovat stovky uživatelů v jednom okamžiku, není PHP dobrá volba i přes své nesporné výhody včetně těch, že webhostingy umožňující provoz aplikací v PHP jsou velmi levné.

- **interpretované jazyky – pomalejší, typová volnost**

Výhodou interpretovaných jazyků je, že autor může šířit zdrojový text bez ohledu na to, na jakém stroji bude potom program běžet, což je u kompilovaných jazyků samozřejmě problém a program vždy může běžet pouze na těch procesorech, pro které byl zkompilován.

Interpretované jazyky umožňují použít programátorské praktiky, kdy části zdrojových textů vznikají až za běhu a nebo přinejmenším názvy tříd, atributů, proměnných či metod a funkcí se získají až v rámci činnosti programu. Není potřeba připomínat, že takové praktiky vedou ke značné nepřehlednosti programu a je nutno je velmi pečlivě komentovat.

Nové programátorské techniky přinesla např. v Javascriptu možnost předat zdrojový kód funkce jiné funkci jako parametr nebo jej uložit jako atribut do instance objektu. Tento zdrojový kód se totiž může odvolávat na proměnné z původní funkce. Tato výhoda bývá často též připisována interpretovaným jazykům, ale podstata leží jinde, protože v Javě nebo C# nebo i jiných jazycích jsou k dispozici lambda výrazy, se kterými lze dosáhnout téhož.

Podobně možnost odkazovat se na třídy, atributy či metody tříd jejich názvem vzniklým jako řetězec až za běhu je připisována interpretovaným jazykům a často se uvádí, že přeložený program si s sebou již tyto názvy nenese a není možné se přes název odkázat na metodu či atribut apod. To je víceméně pravda, ale např. Java tuto možnost má díky mechanismu, který se nazývá reflexe.

Interpretované jazyky často disponují určitou mírou typové volnosti. To znamená, že u proměnných nebo parametrů funkcí se nespecifikuje datový typ a co tam programátor uloží/předá, tak tam prostě je a interpretter si s tím musí nějak poradit. Tím ovšem až za běhu vznikají chyby, na které by u kompilovaných jazyků přišel již kompilátor. U nepořádně odladěných programů je tak uživatel vystavován chybovým hlášením, kterým nerozumí. Navíc typová volnost komplikuje přehlednost programu a určitý pořádek.

Pomalost interpretovaného programu je dána jednak tím, že v rámci interpretace se provádí činnosti, které by jinak provedl již kompilátor, ale podstatné je na tom zejména to, že tyto činnosti se provádějí opakovaně, pokud se nějaký kód provádí opakovaně, např. v cyklu, nebo je některá funkce volána víckrát.

- **překládané jazyky – rychlejší, typová omezení (v dobrém slova smyslu)**

Pokud shrneme rozdíly v překládaných a interpretovaných jazycích, tak přeložený program bude jednoznačně efektivnější ve zpracování, zatímco interpretované programy poskytují určité programátorské techniky, se kterými je potřeba zacházet v duchu hesla - dobrý sluha, ale zlý pán. Často kompilované jazyky vyžadují u proměnných či parametrů přesně specifikovat typ dat a pokud programátor chce předat jiný typ, musí explicitně uvést konverzi. Toto ale by mělo být považováno spíš za výhodu než za nevýhodu přinášející omezení a z něho vyplývající nutnost provést “cosi navíc”. V programu je totiž pořádek a snižuje se pravděpodobnost chyby, která se zjistí až za běhu.

Výše zmíněný mechanismus reflexe v Javě sice přináší do kompilovaného programu možnosti typické pro interpretované jazyky, ale přesto je potřeba s ním zacházet velmi opatrně, protože i reflexe jako taková je poměrně pomalá. Přímé vyvolání metody nějaké třídy je rychlá záležitost (bylo podrobně popsáno v předchozí kapitole), zatímco přes reflexi to znamená, že nejprve se musí získat objekt popisující třídu s interními údaji o tom, jak byla zkompilována a na kterých adresách jsou umístěny její metody a jaké se předávají parametry. Teprve poté lze metodu vyvolat. Zpracování je tak přibližně o jeden řád pomalejší. Pokud už se reflexe v Javě používá, je dobré přinejmenším třídy nebo jejich atributy zadané stringem zanalyzovat jedenkrát na začátku a informace si uložit než tak činit opakovaně.

- **příklady technologií na urychlení programů:**

na tomto místě poznamenejme, že Java byla původně považována za interpretovaný jazyk. Nicméně zavedením technologie JIT začala být Java označována za částečně kompilovaný jazyk a později už vypadlo i slovo “částečně”. Java totiž sice byla kompilována již v prvních svých verzích, ale vždy pouze do tzv. bajt-kódu, což je sice přeložený program do instrukcí procesoru, ale jednalo se o neexistující virtuální procesor, který byl implementován ve virtuálním stroji, který bajt-kód vykonával. To znamená, že k běhu programu v Javě musel být k dispozici program, ve kterém byl virtuální stroj implementován a bajt-kód se interpretoval.

- JIT (v Javě od 1.3 – cca 6x rychlejší, od 1.4 – cca 12x rychlejší, v .NET od počátku)

v Javě 1.3 byl zaveden mechanismus JIT (just in time kompilace). Ten spočívá v tom, že bajt-kód se v okamžiku spuštění programu kompiluje do instrukcí procesoru a následně už je

procesorem přímo vykonáván přeložený program. Takže k běhu programu je sice stále potřeba mít nainstalovaný určitý software, ale nejedná se již o interpretter bajt-kódu, ale kompilátor. Kompilací před spuštěním se sice nepatrně zpomalí start programu, ale následný běh je podstatně rychlejší. Uvádí se, že po vylepšeních v Javě 1.4 došlo až k dvanáctinásobnému zrychlení běhu oproti původní interpretaci bajt-kódu. Výhodou JIT kompilace je, že v době jejího provádění je již znám konkrétní procesor, na kterém kód poběží, takže je možné při kompilaci využít ty nejoptimálnější instrukce, kterou jsou na daném procesoru k dispozici.

Podobný mechanismus byl zaveden i v prostředí .NET Framework firmy Microsoft, kde kompilací vznikají tzv. assembly (sestavení) uložené v DLL souborech. Ty mohou být na úrovni operačního systému ukládány do GAC (global assembly cache). Assembly neobsahují spustitelný kód, ale CIL (původně MSIL), což je obdoba bajt-kódu v Javě. Pro spuštění je rovněž potřeba provést JIT kompilaci. Tu je v .NET Frameworku provést nejen až v okamžiku spuštění, ale předem (např. při instalaci nebo na pozadí v době, kdy procesor není vytížen). Přeložená assembly se ukládá do GAC. Výhodou je, že překlad se neprovede při každém spuštění, ale jednorázově. V tomto smyslu se jedná spíše o AOT, ale označení JIT zůstalo vžité zejména proto, že se zde neprovádějí optimalizace.

Tato technika se používá v omezené míře i u interpretovaných jazyků.

- AOT

Pro překlad JIT provedený ne v okamžiku spuštění, ale předem bez okamžité potřeby běhu, se později začal používat pojem AOT (ahead of time kompilace). Jeho výhoda spočívá v tom, že v době, kdy je AOT kompilace prováděna, je víc času na analýzu kódu a jeho optimalizace, zatímco u JIT se jedná o čistý převod původního kódu do instrukcí procesoru. Druhou výhodou je, že AOT kompilace se provede pouze jednou, zatímco JIT kompilace při každém spuštění.

Tímto pojmem je označován i překlad TypeScriptu do JavaScriptu. Nemusí tedy nutně jít o překlad do spustitelného programu.

- Enterprise Bean třídy v Java EE (problém značné režie související s vytvářením a zanikáním instancí objektů)

Bean třídy v Javě EE jsou dalším příkladem snahy o zrychlení běhu programů. Zde máme na mysli technologii EJB pooling, ve které jde o to, že instance některých tříd v programu často vznikají a zase zanikají, ale průběžně jich neexistuje příliš mnoho současně. V takovém případě může tzv. EJB kontejner na počátku vytvořit určitou množinu instancí, které následně “propůjčuje” programu, který si o ně “řekne” a později je zase vrátí. Vrácené instance zůstávají v paměti a čekají na další využití. Snižuje se tak potřeba alokovat a uvolňovat prostor na haldě.