Teorie programování

Petr Horáček

Obsah

1 **Úvod** 1

2 **Programovací jazyk** 2

2.1 Programovací paradigma 4

2.1.1 Imperativní programování 6

2.1.2 Deklarativní programování 7

2.2 Druhy programovacích jazyků 8

2.2.1 Vyšší a nižší programovací jazyky 9

2.2.2 Interpretované a kompilované programovací jazyky 10

2.3 Vývoj programování programovacích jazyků 10

2.3.1 Programovací jazyky 1. generace 11

2.3.2 Programovací jazyky 2. generace 12

2.3.3 Programovací jazyky 3. generace 13

2.3.4 Programovací jazyky 3 $\frac{1}{2}$. generace 14

2.3.5 Programovací jazyky 4. generace 14

2.3.6 Programovací jazyky 5. generace 15

2.4 Druhy programů 15
2.4.1 Firmware 16
2.4.2 Ovladač periferií 16
2.4.3 Operační systém 17
2.4.4 Démon 17
2.4.5 Aplikační program 18
2.4.6 Skript 18
2.5 Faktory ovlivňující kvalitu software 19
2.5.1 Entropie softwaru 20
2.6 Lexikální prvky programovacích jazyků 21
2.6.1 Identifikátory 22
2.6.2 Operátory 23
2.6.3 Klíčová slova 23
2.6.4 Literály 24
2.6.5 Odkazy 24
2.6.6 Pomocné symboly 24
2.7 Struktura programu 24
3 **Principy programování** 26

3.1 Stav 26
3.2 Vrstva 27
4 **Proces překladu zdrojových kódů** 29
4.1 Preprocesing zdrojových kódů 31
4.2 Komentáře 32
4.2.1 Typy komentářů 32
4.3 Programová makra 33
4.3.1 Makro bez parametrů 34
4.3.2 Makra s parametry 35
4.4 Podmíněný překlad zdrojových kódů 36
4.5 Vkládání hlavičkových souborů 36
4.6 Nedostupný kód 38
4.7 Mrtvý kód 38
4.8 Logický řádek vs fyzický řádek 38
5 **Knihovny** 41
5.1 Interface 41
6 **Data a literály** 42
6.1 Literály 42

6.2 Celočíselné literály 42
6.3 Literály vyjadřující čísla s pohyblivou řádovou čárkou
43
6.4 Znakové literály 45
6.5 Escape sekvence 45
6.6 Řetězcové literály 46
7 **Proměnné** 47
7.1 Jednotky pro měření paměti 48
7.1.1 Byte 49
7.1.2 Word 50
7.1.3 Nibble 50
7.2 Datové typy 50
7.3 Runtime type information 52
7.4 Datová struktura 52
7.5 Datový typ void 53
7.6 Deklarace a definice proměnné 53
7.7 Operátory a operandy 54
7.8 Signet a unsigned proměnné 55

7.8.1 Přetečení proměnné 55
7.9 Konstantní proměnné 57
7.10 Rozsah platnosti proměnných 57
7.11 Globální proměnné 58
7.12 Typová konverze 59
7.12.1 Implicitní datová konverze 59
7.12.2 Explicitní datová konverze 60
7.13 Alokace paměti 60
7.13.1 Statická alokace paměti 60
7.13.2 Dynamická alokace paměti 61
7.14 Paměťové třídy 62
7.14.1 Paměťová třída auto 62
7.14.2 Paměťová třída extern 63
7.14.3 Paměťová třída static 63
7.14.4 Paměťová třída registr 64
7.15 Konstanty 64
8 **Příkazy a výrazy** 65
8.1 Aritmetické operátory 65

8.2 Relační operátory 66
8.3 Bitové operátory 67
8.4 Příkazy 67
9 **Ukazatelé** 69
9.1 Adresy v paměti 69
9.2 Ukazatel na typ void 70
9.3 Ukazatelé jako argumenty podprogramů 70
9.4 Aritmetika ukazatelů 71
9.5 Ukazatelé na ukazatele 71
9.6 Ukazatelé na podprogramy 72
9.7 Argumenty příkazového řádku 72
10 **Řetězce** 73
11 **Binární a textové soubory** 75
11.1 Binární soubor 75
11.2 Textový soubor 76
12 **Procesy a podprogramy** 78
12.1 Vykonávání programového kódu 78
12.2 Popis výkonu podprogramů 79

12.3 Druhy podprogramů 80
12.3.1 Funkce 81
12.3.2 Metoda 81
12.3.3 Procedura 82
12.4 Předávání hodnot do podprogramu 82
12.5 Strukturování operační paměti 84
12.6 Segment text 84
12.7 Segment data a bss 85
12.8 Segment heap 85
12.9 Segment stack 86
13 **Správa verzí** 88
13.1 Verze 88
14 **Dokumentace softwarového projektu** 91
14.1 Životní cyklus projektu 91
14.2 Dokumentace architektury 93
14.3 Dynamické generování dokumentace zdrojových kódů
93
14.3.1 Značka pro nadpis 95

14.3.2 Značka pro třídu 95
14.3.3 Značka pro funkce a metody 96
14.3.4 Značka pro proměnnou 97
14.3.5 Značka pro struktury 98
14.3.6 Značka pro konstantu 98
14.3.7 Značka pro výčtový typ 100
14.4 Soubory README 100
14.4.1 Anatomie souborů README 101
15 **Zálohování** 104
15.1 Zásady zálohování 104
15.2 Zálohovací software 105
16 **Strukturování softwarového projektu** 106
16.1 Konvence psaní identifikátorů 106
16.2 Přehlednost zdrojových kódů 107
16.3 Odsazování 108
16.4 Správné komentování 109
16.5 Fragmentace zdrojového kódu 111
16.6 V jednoduchosti je síla 111

16.7 Žádný opakující se kód 112
16.8 Kompromisy 112
16.9 Defragmentace projektu 113
16.10 Žádné kopírování zdrojového kódu 114
16.11 Chybové hlášky 114
16.12 Nešetřit bílými znaky 115
16.13 Plánovat a myslet dopředu 115
16.14 Předvídat co by se mohlo stát 116
16.15 Znát cíle projektu 116
16.16 Pozor na úniky paměti 117
16.17 Názvosloví 118
16.18 Konzistence identifikátorů 118

Kapitola 1
Úvod

Obecné programování definují obecně platné zákonitosti používané v oboru programování. Každý programovací jazyk disponuje určitými schopnostmi, možnosti a zaměřením na určitou problematiku, ale většina z nich se řídí z větší části stejnými pravidly vycházející z logických úsudků zakladatelů oboru programování.

Oddělení teorie programování od syntaktických pravidel konkrétního programovacího jazyka umožňuje zaměřit se pouze na daný programovací jazyk, protože teorie je pořád stejná.

Teorie je důležitá stejně jako praxe. V případě, že člověk zná teorii a nezná praxi, je to jako by neuměl nic. Disponuje sumou znalostí, které ale není schopen využít. Na druhou stranu bez teorie jak dané věci fungujíc je získat praxi jen velice obtížné. Proto teorie a praxe jde ruku v ruce. Důležité je každou teoretickou znalost prověřit v praxi, protože jen praxe úplně ukáže chování daného systému.

Kapitola 2
Programovací jazyk

Programování, neboli psaní počítačových programů, znamená tvoření instrukcí, které je počítač schopen po hardwarové stránce interpretovat. Tyto instrukce jsou ve formě binárního kódu, 1 a 0, které je jednoduché hardwarové realizovat jako stav zapnuto a vypnuto. Jednotlivé kombinace těchto stavů identifikují danou instrukci. Logický obvod, který je schopen vykonávat různé operace na základě vstupních instrukcí se nazývá programovatelný počítač, který je možné naprogramovat (nastavit), aby vykonával libovolný algoritmus. Jedná se tedy o univerzální obvod, který je schopen řešit a vykonávat různé problémy. Zařízení, které vykonává daný algoritmus se nazývá **procesor**. Motivací k tvorbě programovatelných procesorů byla cena jednoúčelových obvodů (automaty), které byly navrženy, aby vykonávali pouze jeden požadovaný algoritmus. Výhodou toho byl velice rychlé vykonávání algoritmu, ale nevýhodou vysoká cena celkového návrhu a výroba obvodu (většinou se jednalo pouze o malé výrobní série pro konkrétní použití). Obvod, který lze naprogramovat lze použít v mnoha různých aplikacích a proto bylo cenově výhodné vytvořit

velké série programovatelných procesoru, které se pouze naprogramovaly pro daný účel.

Člověk, který vytváří algoritmus pro elektronický procesor se nazývá **programátor**. Programování jednotlivých instrukcí pomocí kombinací nul a jedniček je velice zdlouhavé a u rozsáhlejších projektů velice nepřehledné. Z tohoto důvodu se vyvinuly programovací jazyky, které jednotlivé příkazy definované jako nuly a jedničky slovně pojmenovaly a zjednodušili tak jejich používání.

Zvláštním případem programovacího jazyka je tzv. **pseudokód**, který vlastně tak docela není programovací jazyk. Jedná se o univerzální jazyk pro popis algoritmů, který připomíná programovací jazyk vyšší úrovně. Pseudokód umožňuje překlenout rozdíly mezi různými programovacími jazyky tak aby daný algoritmus pochopil každý programátor, který rozumí nějakému typu vysokoúrovňového jazyka. Pseudokód není nijak standardizovaný vyjadřovací jazyk a proto může každý programátor používat vlastní verzi pseudokódu na kterou je zvyklý. Většinou ale využívají podobná syntaktická pravidla a klíčová slova pro vyjádření algoritmu a proto je snadno pochopitelný. Využívá pouze základní příkazy nutné pro zápis algoritmů a nezabývá se hardwarovými prostředky. Pro je vhodné jej využívat pro příklady zdrojových kódů v různých publikacích zabývajících se obecnými metodami

programování.

Programovací jazyk se sada syntaktických a sémantických (význam jednotlivých slov) pravidel, které umožňují vyjádřit a zaznamenat algoritmus, který může být vykonáván na počítači. Zápis algoritmu ve zvoleném programovacím jazyce se nazývá **program**. Programovací jazyk je komunikačním nástrojem mezi programátorem, který v programovacím jazyce formuluje postup řešení daného problému, a počítačem, který program interpretuje technickými prostředky. Programovací jazyky vznikly pro ulehčení, zrychlení a zpřehlednění rozsáhlých softwarových projektů. Při vývoji informatiky a softwarového inženýrství prošly programovací jazyky několika stádii vývoje. V každém stádiu programovací jazyky využívaly určité styly strukturování celého projektu, které umožňovali vytvářet stále rychleji a rozsáhlejší softwarový projekt. Těmto stylům se říká **programovací paradigma**.

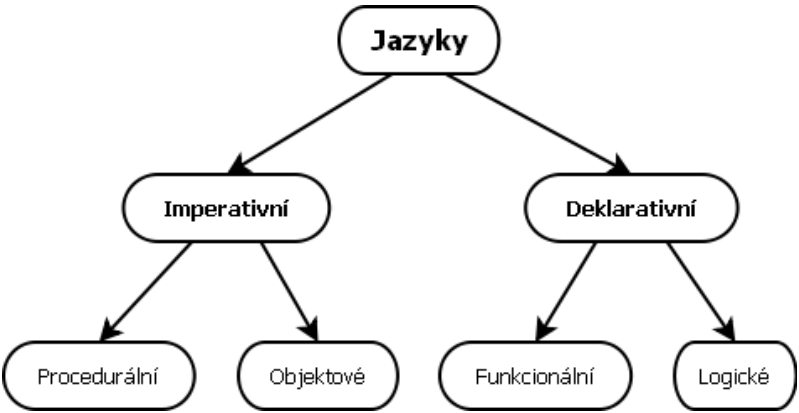
2.1 Programovací paradigma

Programovací paradigma je programovací styl, jakým je strukturován zdrojový kód a celý softwarový projekt. Existuje několik základních programovacích paradigmat. Důležité je, že každé programovací paradigma se hodí na řešení jiného typu problému. Jeden programovací jazyk může podporovat (a v praxi tomu tak bývá) více než

jedno programovací paradigma. To je z důvodu, že každé programovací paradigma přináší určité možnosti, které jiné programovací paradigma neposkytuje. Zároveň však může přinášet určitá omezení která jiné programovací paradigma odstraňuje. V případě spojení více programovacích paradigmat v jednom programovacím jazyce je v mnoha případech možné využívat výhody z více programovacích paradigmat a docílit tak jednoduššího a přehlednějšího a efektivnějšího strukturování (řešení) výsledného projektu. Programovací jazyk, který podporuje více než jedno programovací paradigma se nazývá **Multiparadigmatický programovací jazyk**. Cílem je vytvořit takové programové paradigma, který by umožnilo rychlé, přehledné programování rozsáhlých softwarových projektů.

Programovací paradigmata lze rozdělit do dvou skupin:

- Imperativní
  - -Procedurální
  - -Objektové, (Strukturované)
- Deklarativní
  - -Funkcionální
  - -Logické



2.1.1 Imperativní programování

Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit. Program je sadou proměnných, jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav. Imperativní přístup je blízký i obyčejnému člověku. Například kuchařské recepty či návody k montáži zakoupených výrobků jsou také příkazy krok za krokem a každý příkaz je v závislosti na podmínkách svázán s určitým stavem jídla či kompletnosti výrobku. Hardware pak stejně jako vařící/montující člověk vykonává příkaz za příkazem, přesně jak to odpovídá imperativnímu přístupu.

Pouze jednotlivé příkazy jsou instrukce strojového kódu, k jehož vykonávání je hardware navržen, a aktuální stav vyjadřuje obsah paměti. Vyšší imperativní jazyky pak používají proměnné a komplexnější příkazy (tedy výrazy a funkce), ovšem stále vyznávají to samé, imperativní paradigma.

Procedurální programování člení daný kód do podprogramů, které umožňují recyklovat (znovu použít) již napsaný kód v různých částech. programu. Tím je výrazně zrychlen vývoj programů. Dále je možné celý program rozdělit do jednodušších logických celků, které je jednoduší vytvořit.

Objektové (strukturální) programování do procesu vývoje zavádí možnost jednoduché tvorby složitých datových struktur - třídy, struktury. Objektově orientované paradigma se tak snaží přiblížit reálnému světu. Je snahou popsat nějaké vlastnosti (proměnné) a funkce (podprogramy) reálného objektu. Struktura je druh lineární datové struktury, který je základem objektového programování. Základem strukturovaného programování je vytváření komplexních datových struktur skládající se z více proměnných různého datového typu.

2.1.2 Deklarativní programování

Deklarativní programování je založeno na myšlence

programování aplikací pomocí definic **co se má udělat** a ne **jak se to má udělat**. Výsledný algoritmus pak vytvoří daný překladač programovacího jazyka. Výhodou toho je, že je možné se vyhnout zbytečným chybám při programování, ale překladač ne vždy může zvolit optimální algoritmus.

Funkcionální programování je založeno na postupném vykonávání funkci, které ale nejsou chápány jako programové funkce, ale jako matematické funkce. Funkce v tomto případě bývají aplikované na výsledky jiných funkcí (vnořené funkce). Nevýhodou tohoto programování je menší přehlednost u rozsáhlejších projektů. Využití bylo původně při programování inteligentních programů.

Logické programování je založeno na principech matematické logiky. Program je reprezentován množinou faktů a pravidel překladač se je snaží dokázat. Použití je v programování inteligentních programů, pro výuku a výzkumu.

2.2 Druhy programovacích jazyků

Programovací jazyky lze dělit podle různých kritérií do několika skupin. Podle úrovně abstrakce nad hardwarem je možné programovací jazyky dělit na vysokoúrovňové a nízkoúrovňové. Podle způsobu vykonávání pak na kompilované a interpretované.

2.2.1 Vyšší a nižší programovací jazyky

Nižší programovací jazyky jsou jazyky primitivní, jejichž instrukce (víceméně

přesně) odpovídají příkazům procesoru. To znamená, že procesor bude vykonávat ty instrukce, které programátor napíše. Jsou závislé na svém procesoru a nepřenositelné na jiný (nepříbuzný) procesor. Výhodou je, že programátor má takto přístup i k funkcím počítače, které by měl ve vyšším programovacím jazyce jen těžko a nebo úplně nedosažitelné. Do této skupiny patří **jazyk symbolických instrukcí** a **strojový kód**. Zvláštním typem nižšího jazyka je tzv. **autokód**, který spojuje prvky nižších a vyšších jazyků. Vznikl rozšířením jazyka symbolických instrukcí o jednoduché příkazy pro často používané skupiny instrukcí.

Vyšší (problémově orientované) **programovací jazyky** jsou podstatně srozumitelnější, struktura jejich zdrojových kódů je logická, nejsou závislé na strojových principech počítače. Jsou navrženy na rychlý vývoj rozsáhlejších softwarových projektů. V případě vysokoúrovňových programovacích jazyků zdrojový kód disponuje určitou mírou abstrakce nad hardwarovými prostředky procesoru. To umožňuje zanedbat některé hardwarové detaily a zaměřit se na samotné sestavování algoritmů.

2.2.2 Interpretované a kompilované programovací

jazyky

Interpretované jazyky jsou překládány až za běhu programu. Jsou pomalejší, ale nemají tak velké formální požadavky (není potřeba inicializovat proměnnou, její datový typ se může za běhu měnit, ukazatele jsou zbytečné). Překládají se interpretrem, ten instrukce zároveň při překladu provádí. Hlavní nevýhodou těchto jazyků je, že se musejí vždy spouštět v interpretru. Výhodou je, že interpretované programy jsou multiplatformní. To znamená, že pro spuštění na různých platformách stačí naprogramovat interpret a v něm lze pak spouštět všechny interpretované programy v daném programovacím jazyce.

Kompilované jazyky jsou celé přeloženy do strojového kódu a až poté mohou být spuštěny. Jsou rychlejší, mají vyšší nároky na formální správnost kódu a jsou závislé na hardwaru. Překládají se kompilátorem, výsledkem překladu je nějaký druh spustitelného souboru. Patří sem většina klasických programovacích jazyků.

Teoreticky může mít jeden programovací jazyk verzi interpretovanou i kompilovanou.

2.3 Vývoj programování programovacích jazyků

Jako první programovatelné stroje se označují mechanické stroje tvořené soukolími a mechanickými převody.

Pro tyto stroje se později začaly vytvářet jakési matrice, které stroji říkaly co má v danou dobu dělat (jaký převod má nastavit, které kolo se má pootočit o danou vzdálenost, ...). Tyto stroje později začaly nabírat elektromechanický a elektrotechnický charakter a začaly být tvořeny elektronkami, děrnými štítky, ... Později začaly být tvořeny polovodičovými součástmi, což umožnilo rapidní zmenšení rozměrů a spotřeby energie a zároveň zvýšení výpočetního výkonu. S konstrukcí počítačů se zároveň vyvíjel způsob jejich programování. Vývoj programovacích jazyků je členěn do generací, které jsou charakteristické možnostmi výpočetních zařízení, využitím v dané době a programovacími paradigmaty.

2.3.1 Programovací jazyky 1. generace

V první generaci byly vytvářeni jednoúčelové relativně jednoduché programy pro účely použití v matematice a fyzice (inženýrských prací). Programovacím jazykem byl přímo strojový kód pro daný počítač. Jednotlivé příkazy (instrukce) byly do paměti zapisování v binární podobě, ale z důvodu lepší čitelnosti se programu se používal pro zápis osmičkové nebo šestnáctkové podobě. Takové programy byly zpravidla rychlé a efektivní, ale byl problém při hledání chyb a úpravách kódu. V případě, že bylo potřeba do existujícího kódu vložit nový úsek programu bylo nutné vytvořit posun zbytku programu a pozměnit všechny

adresy skoku. Navíc každý program měl na různých počítačích jiný kód, protože každý počítač používal vlastní sadu příkazů.

2.3.2 Programovací jazyky 2. generace

Programování ve strojovém kódu počítače se časem ukázalo jako zdlouhavé, neefektivní a pro větší projekty nepřehledné a nevhodné. Proto byl vytvořen jazyk symbolických instrukci a překladový program nazývaný Assembler. Každá instrukce daného počítač je reprezentována symbolickým názvem. To přineslo výrazné zpřehlednění zápisů počítačových programů. Symbolické názvy proměnných a adresových míst odbouralo nutnost přepisovat adresy při pozdějších úpravách programu. Přetrvávající problém byl však závislost zápisu daného programu na použitém počítači a programátor zároveň musel být odborníkem na použitý hardware.

Jisté zjednodušení přinesli tzv **autokódy** rozšíření jazyka symbolických instrukcí o symbolické zápisy složitější povely (skupiny instrukcí). Díky tomu bylo možné předem vytvořit nejpoužívanější skupiny instrukcí (procedury, ...).

Jazyk symbolických instrukcí je v současné době využíván jen zřídka (při tvorbě některých částí operačního systému, části programu kde je potřeba rychlost, ...), ale programátorům poskytuje představu o fungování

strojových instrukcí.

2.3.3 Programovací jazyky 3. generace

Programovací jazyky 3. generace byly nejspíš nejvýznamnější změnou pro rozvoj softwarového inženýrství. Díky těmto jazykům došlo k vytvoření **strojově nezávislých** programovacích jazyků podporující metody **strukturovaného programování**. Díky tomu došlo k rozčlenění celého programu na autonomní (nezávislé na zbytku programu) logické části - moduly.

Vznik jazyků 3. generace byl umožněn pokroky na poli teorie formálních jazyků. K většině těchto jazyků existuje přesná norma, takže tentýž program vypadá stejně na jakémkoliv počítači - přenositelnost jazyka. To byl nečekaný ale prospěšný vedlejší produkt jazyků vyšší úrovně. Jeho převedení do strojového kódu zajišťuje překladač.

Typické příklady jazyků 3. generace je FORTRAN (formula translation), který byl orientován především na matematické výpočty. Dále to je významný jazyk C, Basic, Pascal, ...

Jazyky této generace došlo k rozdělení podle paradigmat na imperativní a funkcionální.

2.3.4 Programovací jazyky 3 $\frac{1}{2}$. generace

Takto bývá označována rodina **objektově orientovaných jazyků**. Jedná se o určité rozšíření (mezikrok) programovacích jazyků 3. generace. Objektově orientované programovací jazyky se vyznačují snahou o napodobení reálných objektů, které jsou definovány určitými vlastnostmi (proměnnými) a funkčností (funkce). To výrazně zrychlilo vývoj softwaru díky možnosti znovu použít (a modifikovat) již vytvořený kód. Typickými zástupci objektově orientovaných jazyků je C++, C#, Java, ...

2.3.5 Programovací jazyky 4. generace

V 80. letech se začaly objevovat prostředky (spíše než jazyky), které místo vypisování jednotlivých příkazů do příkazové řádky dovolují komunikovat s počítačem pomocí obrázkových prostředků - nabídek, dialogů, obrázků, ikon (GUI) označující data nebo programy, které je možné pomocí myši přesouvat, kopírovat, označovat a podobně.

Velmi často toto prostředí 4. generace pak generuje nějaký kód jazyka 3. generace, který pomocí příkazů popisuje názorně specifikované akce. Uživatel tohoto prostředí tedy vůbec nemusí umět programovat, pouze interaktivně vytváří požadovaný výsledek co možná vizualizovaným způsobem. Typickým je princip WYSIWYG (What You See Is What You Get - co vidíš to dostaneš).

Určitou kombinací jazyků 4. generace a 3. generace je spojení grafického nástroje pro tvorbu grafického uživatelského prostředí programu 4. generace a v jazyce 3. generace je jednotlivým grafickým prvkům programu přiřazena určitá funkcionalita. Díky tomu je urychlena část tvorby grafické nadstavby programu a je možné se zaměřit na detaily programu.

2.3.6 Programovací jazyky 5. generace

V předchozí generaci musí programátor, byt' maximálně jednoduchým a názorným způsobem, specifikovat posloupnost akcí, které mají být provedeny, aby se dosáhlo požadovaného výsledku. Naproti tom prostředky 5. generace nabízejí nalezení postupu vedoucího k požadovanému cíli samotným počítačem. Jedná se o neprocedurální programovací jazyky - deklarativní paradigma. Typickým zástupci této generace programovacích jazyků je jazyk Prolog. Lips, ...

2.4 Druhy programů

Algoritmus, který je vyjádřen pomocí libovolného programovacího jazyka a vykonáván na libovolné počítačové platformě se nazývá počítačový program. Jedná se o posloupnost instrukcí procesoru, které vykonávají daný algoritmus. Počítačový program je konečný produkt softwarového

inženýrství.

Z hlediska účelu je možné rozlišovat několik druhů počítačových programů:

- Firmware
- Ovladač periferií
- Operační systém
- Démon
- Aplikační program
- Skript

2.4.1 Firmware

Firmware je základní softwarová výbava programovatelných obvodů. Jedná se o nízkoúrovňový program, který přímo ovládá jednotlivé periferie. V případě firmwaru mohou nastat dvě situace. Dané programovatelné zařízení je plně autonomní a ke svému chodu nevyžaduje žádný nadřízený systém, takový případ je nazýván jako **vestavěný** (embedded) **systém**. V opačném případě firmware obsahuje komunikační rozhraní, které slouží k ovládání daných periferií nějakým nadřazeným systémem. Tím je většinou ovladač periferie.

2.4.2 Ovladač periferií

Ovladač periferie je program, který slouží k ovládání periferních zařízení počítače. V kontextu operačního systému se jedná o tlumočníka mezi komunikačním rozhraním firmwaru zařízení a komunikačním rozhraní operačního systému definované abstraktní vrstvou hardware (viz. abstraktní vrstva: operační systémy), která definuje komunikační pro komunikaci s periferiemi. To umožňuje pro libovolnou hardwarovou sestavu používat pouze jeden stejný operační systém se standardizovanými příkazy, aniž by věděl jak s danými periferiemi zacházet. Příkladem může být ovladač grafické karty. V jiném smyslu může jít o ovládací aplikační program, který komunikuje s danou periferií, například souborový správce, který komunikuje se souborovým systémem dané paměti.

2.4.3 Operační systém

Operační systém je základní softwarové vybavení počítačů, které slouží k paralelnímu zpracování dat. Operační systém má na starosti správu hardwarových prostředků, správu paralelně spuštěných programů a komunikační rozhraní mezi počítačem a jeho uživatelem. Operační systém je velice složitý program, který většinou sestává z více částí, které mají za úkol rovnoměrně rozdělovat hardwarové prostředky mezi aplikační programy a démony.

2.4.4 Démon

Démon nebo také **služba** je typ programu, který se neřadí ani mezi ovladače ani mezi aplikační software. Jedná se o dlouhodobě spuštěný program (typicky je spuštěn spolu s operačním systémem.), který v nečinnosti čeká až nastane nějaká událost, kterou má na starosti obsloužit. Běžný uživatel se programem typu démon normálně nesetká, protože většinou nemá uživatelské ovládací rozhraní a nebo jej poskytuje jiný program, který musí být za tímto účelem spuštěn. Typickým příkladem programu typu démon je nějaký typ serverového programu jako http server nebo tiskový server a jiné.

2.4.5 Aplikační program

Aplikační program je programu, který je spouštěn samotným uživatelem a který obsahuje nějaké komunikační rozhraní (textové, grafické, ...) pro komunikaci s uživatelem. Aplikační programy umožňují vykonávat na počítači předem definované algoritmy snadno využívat jeho hardwarové prostředky. Jedná se o typ programů, které jsou zaměřeny na používání samotnými uživateli a proto jsou k tomu patřičně upraveny (komunikační rozhraní, zvolená funkcionalita, možnost přistupovat k souborům v paměti počítače, ...).

2.4.6 Skript

Skripty jsou konfigurační soubory, které v operačním systému slouží k vykonání nějakých akcí buď při spuštění systému a nebo v případě, že nastane nějaká událost. Na rozdíl od démonů, nejsou dlouhodobě spuštěny. Skripty jsou po vykonáni své předem definovaného úkolu opět ukončeny. Jedná se o **interpretované programy**, které vyžadují interpretační nástroje. Skripty v operačním systému vytvářejí jednoduchý nástroj pro jeho konfiguraci a pokročilé nastavení. Typickým příkladem využití skriptů je spuštění určitých aplikačních programů po spuštění operačního systému nebo po přihlášení daného uživatele.

2.5 Faktory ovlivňující kvalitu software

Kvalitu software ovlivňují především dva faktory: *vnější* a *vnitřní*. Vnější faktory jsou viditelné přímo uživateli, vnitřní faktory jsou viditelné pouze počítačovým profesionálům (programátorů daného software, počítačovým analytikům, ...). Jako příklad vnějších faktorů kvality je možné uvést:

- *Správnost* - schopnost programu přesně vykonávat svou úlohu jak byla zadána v požadavcích
- *Robustnost* - schopnost programu reagovat na abnormální podmínky
- *Další faktory* - rychlost, efektivnost, rozšířitelnost, kom-

patibilita, ...

Podstatný je zejména vztah správnosti a robustnosti programu. Správný program vyjadřuje schopnost správně pracovat v situaci popsané v zadání. Naproti tomu robustní program vyjadřuje schopnost ignorovat nastalé nesprávné situace, aby nezpůsobily katastrofu programu, ale musí zároveň správně reagovat na zadané situace ve specifikaci. Robustnost je tedy nadmnožinou správnost. Robustnost lze mnohem obtížněji zajistit než samotnou správnost programu.

2.5.1 Entropie softwaru

Entropie je pojem z termodynamiky, který označuje míru neuspořádání (chaosu) v (fyzikálním) systému. **Entropie softwaru** označuje chaos a neuspořádání ve struktuře zdrojových kódů, souborů a dalších materiálech v systému softwarového projektu.

Pokud se entropie softwarového projetu zvětšuje (prohlubuje) říká se že softwarový projekt **"uhnívá"**. Toto uhnívání může mít za následek neúspěch softwarového projektu.

Náchylnost softwarového projektu k prohlubování entropie závisí na mnoha faktorech jako je například rozsah softwarového projektu, jeho podstata a cíle, použité

technologie, vlastnosti softwarového týmu a jiné. Všechny tyto vlastnosti mají ale společný následek. Jak softwarový projekt roste dochází k větší neuspořádanost a snižování přehlednosti jehož následkem je neúmyslný vznik chyb, které jsou nazývány **softwarové díry**, nebo také **softwarové bugy** (pojem z problematiky testování software). Tyto softwarové díry jsou následkem nesprávně nebo nevhodně navržené části softwaru, vlivem špatné přehlednosti projektu, nebo z jiných důvodů. Tyto chyby mohou být nenápadné, ale mohou také výrazně ovlivňovat chod a stabilitu softwaru jako celku.

K omezení entropie je nutné tyto díry v softwaru najít a opravit dříve než se jich nahromadí tolik, že dojde k ohrožení stability systému, kdy jejich odstranění již nemusí být triviální záležitost.

2.6 Lexikální prvky programovacích jazyků

Každý programovací jazyk se skládá ze základní sady slov (slovník programovacího jazyka). Každé slovo má svůj sémantický význam a vztahují se na něj určitá syntaktická pravidla. Z toho vyplývá, že programovací jazyk lze pokládat za **formální jazyk**.

Syntaxe programovacího jazyka definuje formální strukturu programovacího jazyka. Určuje způsob jak vzájemně kombinovat jednotlivé lexikální elementy progra-

movacího jazyka. Sémantika určuje význam jednotlivých lexikálních prvků programovacího jazyka. To znamená, že syntaxe a sémantika jsou vzájemně propojeny.

Základní slovník programovacího jazyka je navržen tak, aby byl vhodný pro určité aplikace, na které je daný programovací jazyk určen. Zaměření programovacího jazyka může být například nízkoúrovňové programování programů optimalizovaných na výkon a úsporu paměti (obecně hardwarových prostředků), nebo na rychlý vývoj programů pro osobní počítače kde díky výkonu a možnostem počítače není nutné šetřit hardwarové prostředky.

Slova ze slovníku každého programovacího jazyka lze rozdělit do několika lexikálních skupin, ale ne každý programovací jazyk muže obsahovat slova ze všech skupin:

- identifikátory
- operátory
- klíčová slova
- literály
- odkazy
- pomocné symboly

2.6.1 Identifikátory

Identifikátory jsou slovní pojmenování objektů ve zdrojovém kódu. Identifikátor může pojmenovávat paměťovou adresu proměnné, nebo další příkazy, aliasy. Dalo by se říci, že základní slovník programovacího jazyka je tvořen identifikátory, které identifikují základní funkční prvky programovacího jazyka (klíčová slova, operátory, ...). Typickým příkladem je pojmenování proměnné nebo podprogramu, které se při překladu do strojového kódu nahradí příslušnou paměťovou adresou.

2.6.2 Operátory

Operátory jsou základní aritmetické a logické operace se kterými umí daný procesor pracovat. Tyto operace jsou v programovacím jazyce definovány pomocí znakových popřípadě slovních identifikátorů s určitými syntaktickými pravidly použití.

2.6.3 Klíčová slova

Klíčová slova mají ve slovníku programovacího jazyka zvláštní sémantický význam. Klíčová slova jsou speciální identifikátory, které jsou rezervovány pro použití jako součást daného programovacího jazyka. Tyto rezervované identifikátory klíčových slov nemohou být použity k žádnému jinému účelu. Příkladem může být if, else, this, ...

2.6.4 Literály

Literály jsou znakové elementy, které umožňují ve zdrojovém kódu vložit nějaká data a pracovat s nimi.

2.6.5 Odkazy

Odkazy jsou speciální prvky, které umožňují skoky ve struktuře strojového kódu programu. Jedná se o speciální případ slovního identifikátoru, který pojmenovává paměťovou adresu, na kterou se má v případě skoku program přemístit. Jedná se výhradně o tzv. **návěští**, která jsou vstupem do příkazu pro skok. Určitým typem odkazu jsou také identifikátory podprogramů, které po svém zavolání přeskočí do jiné části programu.

2.6.6 Pomocné symboly

Pomocné symboly jsou znaky, které mají určitý význam při strukturování zdrojového kódu, ale neovlivňují samotný programovaný algoritmus. Pomocnými symboly jsou například definice komentářů, symbol pro ukončení příkazů (většinou středník), závorky, ...

2.7 Struktura programu

Zdrojový kód programu napsaný v daném programovacím jazyce se vyznačuje určitou vnitřní strukturou

(uspořádáním), která je dána programovacím paradigmatem, které daný programovací jazyk využívá. Cílem strukturování zdrojových kódu je rozdělit rozsáhlý program na menší snáze pochopitelné, otestovatelné a odladitelné logické části. Tyto části se obecně nazývají **moduly**.

Modul by měl v programu vykonávat jednu přesně definovanou a jasně pochopitelnou úlohu, popřípadě několik přesně definovaných úloh. Jen tehdy má dělení na moduly smysl.

Každý modul by měl mít co nejméně vazeb na ostatní moduly v programu. V případě velkého propojení modulu s ostatními moduly by nebylo jednoduché jednotlivé moduly otestovat, pochopit nebo přenést do jiného projektu.

Každý modul obsahuje rozhraní a tělo. Rozhraní udává jaké prostředky (proměnné, datové struktury, ...) modul nabízí k použití ostatním modulům. Tělo pak obsahuje implementační detaily a mělo by zůstat ostatním modulům skryto. Nejjednodušším modulem je procedura nebo funkce, jejímž rozhraním jsou její parametry. Procedura by tedy neměla komunikovat se zbytkem programu jinak než svými parametry.

Kapitola 3
Principy programování

Principy programování umožňují definovat základní zákonitosti, postupy a techniky, které jsou využívání při navrhování architektury sotware.

3.1 Stav

Stav jsou hodnoty, které popisují daný objekt. Tímto objektem může být cokoliv. Například hodinky, počítačový program nebo i člověk. Stavem hodinek je poloha ručiček ukazující čas, stavem programu jsou hodnoty vnitřních proměnných a stavem člověka je nezpočet údajů, které popsují funkci a polohu všechn vnitřních orgánů. Stav může být závislí na čase. T znamená, že stav může být **proměnný v čase**. Stav nějakého objektu nelze zjišťovat sledováním pouze určitých hodnot, které budou porovnávány s jinými hodnotami v jiném čase. Tím by se získal nesprávný údaj o stavu daného objektu.

Stavv je základní prvek při budování software. Vnitřní stav programu určuje jak se má v danou chvíli chovat. Díky vnitřnímu stavu programu lze definovat různé pracovní režimi programu.

Stav je základem stavového automatu, který jehož činnost plně závisí na aktuálním vnitřním stavu. Vnitřním stavem stavového automatu jsou definovány také hodnoty vstupů, které jsou vnitřně tvořeny proměnnými.

3.2 Vrstva

Vrstva je způsob členění činností do logických celků, které umožňují seskupovat související činnosti a tak zpřehlednit výslený algoritmus (program). Účelem vrstvy je také izolovat určitou hladinu dějů tak, aby jejich složitost zůstala skryta ostatním částem programu.

Každá vrstva poskytuje komunikační rozhraní, prostřednicvím kterého komunikuje s ostatními vrstvami programu. Vrstvy mohou být v programu organizovány několika způsoby:

- Hierarchické uspořádání - vrstvy jsou děleny do nižžších a vyšších vrstev přičemž vyšší vrstvy stojí na nižších vrstvách, to znamená, že využívají funkčnosti nižších vrstev. Jedná se o systém klient-serverS. Nikdy nemůže nastat, aby vrstva nižší vrstva používala služby vrstvy vyšší. Zároveň by neměl nastat případ, kdy vyšší vrstva využívá služby vrstvy o více než jednu úroveň nižší.
- Paralelní uspořádání - vrstvy jsou na stejnolehlé úrovni.

Jedná se o nezávislé funkční celky, které se vzájemně neovlivňují a pracují nezávisle na sobě.

- Hybridní uspořádání - v hybridním uspořádání se nacházejí jak v hierarchickém uspořádání tak v paralelním uspořádání.

V případě, že program není rozdělen do vrtevnaté struktury vznikne z programu nepřehledná zněť funkcí, ve které je těžké se orientovat. Každá vrstva řeší svou problematiku, kterou tak nemusí zatěžovat ostatní vrstvy, které předpokládají že nižší vrstva jejíž služby využívá pracuje bez chyb přesně podle poředpokladů. Nadřazená vrstva se tak může zabývat svými problémy.

Další výhoda členění softwarového projektu do vrstev je možnost, aby na projektu pracovalo více programátorů, kteří se nemusejí zajímat o to co programují ostatní.

V neposlední řadě umožňuje používání vrstev jistou dynamiku projektu. Každá vrstva definuje své komunikační rozhraní, které vrstva vyšší využívá. Díky tomu je možné rozšířit, nebo upravit funkčnost programu pouhým nahrazením jedné vrstvy vrstvou jinou (rozšířenou, opravenou, pro jiné zařízení, ...) se stejným programovým rozhraním, ale jinou funkčností.

Kapitola 4
Proces překladu zdrojových kódů

Překladač je speciální program, který má za úkol přeložit program z jednoho jazyka do jazyka jiného. Nejčastěji se jedná o překlad programu z nějakého programovacího jazyka vyšší úrovně do strojového jazyka počítače. Další možností je překlad mezi dvěma různými programovacími jazyky vyšší úrovně a nebo ze strojového jazyka do jazyka symbolických adres nebo programovacího jazyka vyšší úrovně (reverzní inženýrství, analýza spustitelného programu).

Existují dva přístupy k překladu zdrojového kódu: *kompilační* a *interpretační*.

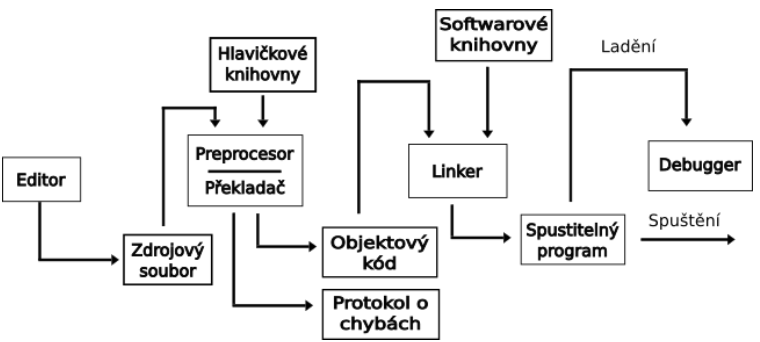
• **Kompilátory** jsou překladače, které překládají zdrojový kód do strojového jazyka počítače. Překlad předchází spuštění takového programu.

• **Interprety** jsou překladače, které obcházejí přímí překlad do strojového kódu počítače. Jednotlivé výrazy a příkazy se ze zdrojového kódu programu překládají postupně při

jejich interpretaci. Spustitelný program jako celek v tomto případě nevzniká. Překlad probíhá souběžně s během programu.

Vstupem procesu kompilace je programátorem napsaný zdrojový kód uložený ve zdrojovém souboru. Jako první se ke zpracování dostává preprocesor. **Preprocesor** je softwarová část kompilátoru, která je spuštěna v první části překladu. Přitom dojde k předzpracován kódu, například k odstranění komentářů, netisknutelných znaků, dosazení konstant, správné vložení dodatečných knihoven s před-připravenými podprogramy, ..., aby měl překladač snadnější práci. Výsledkem práce preprocesoru je opět textový soubor. Z preprocesoru vstupuje upravený kód do samotného překladače (Compiler). **Překladač** též **kompilátor** nebo anglicky Compiler je softwarový nástroj používaný programátory pro překlad programu napsaný ve vyšším programovacím jazyce do strojového jazyka procesoru. Procesor počítače ve zkutečnosti danému programovacímu jazyku nerozumí, protože rozumí pouze programům zapsaných v takzvaném strojovém kódu. **Strojový kód** se skládá z jedniček a nul, které říkají logickým obvodům procesoru co mají v danou chvíli dělat. Výsledkem kompilátoru je objektový soubor, který ještě není spustitelný. Překladač překládá zdrojový soubor do relativního (objektového) kódu. Relativní kód je téměř

hotový program, ale adresy funkcí a proměnných ve strojovém kódu ještě nejsou známy - jsou relativní. Vedlejším produktem překladu je protokol o překladu, který uchovává informace o chybách nalezených při překladu(vytváří se na speciální přání uživatele pro účely ladění a hledání chyb). Poslední na řadu přichází **linker** neboli **sestavovací program**, což je program, který provede sestavení objektového souboru obsahující objektový kód a knihovní funkce na spustitelný soubor. Relativním adresám přidělí absolutní adresy, propojí program s externími softwarovými knihovnami, ...



4.1 Preprocesing zdrojových kódů

Preprocesing zdrojových kódů umožňuje do zdrojových kódů vkládat dodatečný kód, který ale není obsažen ve výsledném strojovém kódu programu. **Preprocesing** nebo také **předzpracování** zdrojového kódu je proces kdy jsou ze zdrojového kódu vyjmuty přebytečné bílé znaky, komentáře a rozbalují se makra. To umožňuje lépe strukturovat zdrojový kód a tím zvýšit jeho přehlednost a čitelnost.

O preprocesing se stará část kompilačního programu, která se nazývá **preprocesor**. Ta připraví zdrojový kód k procesu kompilace.

4.2 Komentáře

Komentář je text, který se nachází ve zdrojovém souboru vedle zdrojového kódu, který je ve fázi překladu překladačem zcela ignorován (preprocesor je ze zdrojového kódu zcela vypustí a do překladače jde již čistý zdrojový kód). Komentáře slouží k popisu činnosti složitějších programových konstrukcí uvnitř programu, jejichž význam nemusí být na první pohled jasný. Komentáře výrazně zvyšují čitelnost zdrojového kódů.

4.2.1 Typy komentářů

Rozlišují se dva druhy komentářů - řádkový komentář a blokový (víceřádkový) komentář . Řádkové komentáře jsou krátké komentáře s působností pouze na jednom jediném

řádku. Slouží ke krátkým vysvětlivkám a poznámkám. Řádkové komentáře jsou většinou definovány pomocí dvou po sobě jdoucích lomítek:

```
//řádkový komentář
```

Blokový komentář definuje rozsáhlejší typ komentáře s působností dle aktuální potřeby. Slouží k popisu funkčnosti podprogramů, vstupů, výstupů, ... Blokové komentáře jsou definovány mezi uvozovací /* a ukončovací */ sekvencí blokových komentářů:

```
/*blokový komentář*/
```

4.3 Programová makra

Makro je pojmenovaná část zdrojového kódu. Jinými slovy je to identifikátor, který pod sebou uchovává část kódu podobně jako funkce, který může být kdykoliv zavolán. Tento kód se při kompilaci v preprocesoru nakopíruje do míst kam odkazuje identifikátor makra. Vložení kódu makra do zdrojového kódu programu je také označováno jako **expanze** nebo **substituce makra**. Makra jsou texty na které lze nahlížet jako na zdrojové kódy, ale jako se zdrojovými kódy se s nimi pracuje až, po zpracování preproce-

sorem. Funguje to tak, že při každém výskytu makra ve zdrojovém kódu je identifikátor makra nahrazen ve zdrojovém kódu programu textem makra.

4.3.1 Makro bez parametrů

Makra bez parametrů se používají pro definici konstant, kde je použito nějaký řetězec nebo hodnota. Výhodou je oproti klasickým konstantám, že není zbytečně zabíráno místo v paměti při deklaraci. Konstanta pomocí makra funguje stejně jako by se daná hodnota napsala do kódu ručně ale s výhodou, že při potřebě změny její hodnoty ji není třeba měnit všude, ale pouze na jednom místě. Makro bez parametrů se typicky definuje pomocí direktivy pro preprocesor (zpráva o vytvoření makra), identifikátorem makra a její hodnotou. Definice makra může vypadat takto:

```
makro IDENTIFIKATOR_MARKA hodnota
```

Makro může být definováno v jakékoliv části programu, ale s ohledem na přehlednost kódu se vždy definují na začátku kódu. Platnost makra začíná od místa jeho definice až do konce programu.

Pro psaní maker bez parametrů platí pravidla:

• Jména konstant se píší velkým písmenem, aby se odlišily od proměnných.

• Za definicí konstanty by měl následovat komentář popisující význam dané konstanty

• Pokud je hodnota konstanty delší než řádka, musí být na konci každé řádky znak zpětného lomítka \, který se do makra nerozvine, ale slouží jako pomocný znak (escape).

• Nové konstanty mohou využívat již dříve definované konstanty

4.3.2 Makra s parametry

Makra s parametry fungují jako funkce, které se ale v programu nevolají ale preprocesor kód makra vloží přímo do volaného místa v kódu. Tím pádem nedochází ke skokům do podprogramu a program je tím pádem rychlejší. To má význam například při programování jednoduchých mikrokontrolérů, kde záleží na efektivním využití jeho výkonu. Definice makra s parametry může vypadat takto:

```
makro IDENTIFIKATOR_MAKRA(parametry) kód makra
```

Mezi jménem a kulatými závorkami s parametry nesmí být mezera jinak by byla brána jako hodnota makra a ne jeho parametry. Jména maker s parametry se na rozdíl

od maker bez parametrů píší malými písmeny. Volání je prováděno výpisem jména makra s kulatými závorkami ve kterých se nacházejí vstupní hodnoty.

4.4 Podmíněný překlad zdrojových kódů

Preprocesor často disponuje také direktivami ovliňující viditelnost určitých částí zdrojového kódu. Pomocí podmíněných výrazů pro preprocesor lze uzavřít určité části zdrojového kódu do bloků, které se při kompilaci přeloží pouze za splnění určitých podmínek. To je často využíváno při ladění programu, protože při identifikaci problémů v kódu je občas třeba některé části kódu vypustit, aby se zjistilo jak se program bude chovat. Dále je možné využívat podmínění překlad při dynamické konfiguraci zdrojových kódů, kdy jsou za určitých podmínek některé části skryty a zároveň jiné části zviditelněny. Tím je možné prohodit hardwarově závislé části programu a jednoduše portovat daný program na jinou platformu.

4.5 Vkládání hlavičkových souborů

Preprocesor se také stará o vkládání hlavičkových souborů do zdrojových kódů. Do zdrojového kódu lze obecně vložit obsah jakéhokoli textového souboru, ale standardně se používá soubor s určitým formátem, aby bylo na první pohled jasné co je obsahem daného souboru.

Hlavičkové soubory jsou soubory obsahující již hotové a odladěné části kódu, které se dají opakovaně používat. Je to dobrý mechanizmus jak udržovat velký program v čitelné podobě. Celý zdrojový kód programu se rozbije na menší související díly, které je snazší odladit a kontrolovat. Takovému přístupu se říká **vrstvení kódu**.

Problém u hlavičkových souborů může nastat v případě, kdy jsou vícenásobně (přímo nebo nepřímo skrze jiný hlavičkový soubor) vloženy do stejného souboru. Překladač se poté musí potýkat s problémem, že po slinkování zdrojových souborů se ve výsledném kódu vyskytuje kód z prvního hlavičkového souboru dvakrát. Tento problém se řeší pomocí podmíněného překladu. Celý kód v hlavičkovém souboru se uloží do obdobného bloku:

```
ifndef JEMENO_SOUBORU

makro JMENO_SOUBORU

    kód hlavičkového souboru

endif
```

Text hlavičkového souboru se do míst vkládání vloží

pouze v případě, že nebylo vytvořeno dané makro. Toto makro je ale vytvořeno až za kódem podmíněného překladu. Tím pádem se do daného místa vloží text hlavičkového souboru pouze jednou, protože tím je vytvořeno dané makro které následně brání dalšímu vkládání.

4.6 Nedostupný kód

Nedostupný kód) je část programového kódu, která není nikdy vykonána, protože neexistuje žádná cesta, která by k ní vedla ze zbytku programu. Obecně se jedná o nežádoucí chybu v programovém kódu. Nedostpný kód způsobuje redundanci programového kódu, která způsobuje snížení jeho přehlednosti. Některé překladače jsou schopné odhalit nedostupný kód a nezačlenit jej do výsledného spustitelného programu, ale v případě, že překladač nezachytí nedostupný kód, způsobuje zvýšení velikosti výsledného programu jehož následkem je potřeba více paměti, cashování instrukcí v CPU a tím snížení výkonosti daného programu.

Nedostupný kód může mít mnoho příčin,

4.7 Mrtvý kód

Mrtvý kód (dead code)

4.8 Logický řádek vs fyzický řádek

V prostředí zdrojových kódu se rozlišují dva typy řádků zdrojového kódu. Prvním typem je **fyzický řádek** zdrojového kódu. Jedná o klasický řádek textu který je ukončen ukončovacím znakem.

Druhým typem řádku je **logický programový** řádek zdrojového kódu. Logický programový řádek je dán typem použitého programovacího jazyka. Každý příkaz v daném programovacím jazyce by měl být umístěn na samostatný řádek. Každý řádek, který pojímá jeden příkaz daného programového jazyka se pak nazývá logický programový řádek zdrojového kódu.

Na první pohled není mezi fyzickým a logickým programovým řádkem žádný rozdíl, ale u zdrojových kódů platí, že jeden fyzický řádek může obsahovat více logických řádků. Dále se ve zdrojovém kódu nachází velké množství prázdných řádků a řádků s komentáři, které výrazně zvyšují čitelnost a přehlednost zdrojového kódu. To v praxi znamená že na jeden fyzický řádek je umístěno více příkazů, které by správně měly být umístěny na samostatný fyzický řádek. V extrémních případech (pokud to překladač daného programovacího jazyka umožňuje) může být zdrojový kód napsán pouze na jeden jediný fyzický řádek. Výsledný kód je ale velice nepřehledný a nepřidáší žádné přínosy pro výsledný program. Výsledkem je, že ve zdrojovém kódu může být (a v praxi tomu tak bývá) je jiný počet fyzických řádků

než logických.

Logické a fyzické programové řádky se využívají při analýze zdrojových kódu určování jejich zkutečného rozsahu a odhadování náročnosti.

Kapitola 5
Knihovny

5.1 Interface

Kapitola 6
Data a literály

Pro mnoho lidí se zdají termíny data a informace jedno a to samé. To je ale omyl. Data jsou nejmenším množstvím informace. Například celé jméno člověka je informace, ale nejsou to data, protože celé jméno lze rozdělit na křestní jméno a příjmení. Křestní jméno a příjmení jsou data protože již nejdou dále rozložit na další smysluplné informace.

6.1 Literály

Data jsou čísla nebo znaky, které jsou v programu vyjadřovány pomocí tzv. literálů - explicitně zapsané hodnoty. Na literál lze pohlížet jako na číslo nebo znak, který je vepsán do zdrojového kódu. Například písmeno "J" je literární znak. Rozlišují se 4 základní typy literálů:

- Celé číslo
- Číslo s pohyblivou (plovoucí) řádovou čárkou
- Znak
- Řetězce

6.2 Celočíselné literály

Celé číslo je číslo, které nemá desetinnou část, zapsané přímo v programu. Celé číslo lze zapsat formou dvojkové, desítkové, osmičkové nebo šestnáctkové soustavy. Při použití desítkové soustavy se píší čísla do programu běžným způsobem. Příklad celočíselného literálu zapsaného v desítkové soustavě je číslo 5. Způsoby zápisu celočíselných literálů do zdrojových kódů v ostatních číselných soustavách závisí na konvencích daného programového jazyka. Většinou se ale před zápisem čísla v dané číselné soustavě nachází určitá předpona identifikující danou číselnou soustavu například 0b110101 je číslo zapsané v binární soustavě a vyjadřující číslo 53 v desítkové soustavě.

6.3 Literály vyjadřující čísla s pohyblivou řádovou čárkou

Pohyblivou řádovou čárkou nebo plovoucí řádovou čárkou se rozumí způsob reprezentace čísel, která by byla moc malá nebo velká pro vyjádření v pevné řádové čárce. Čísla jsou obecně uložena jako určité množství platných číslic vynásobený exponentem. Čísla, která mohou být v pohyblivé řádové čárce vyjádřena přesně, jsou ve tvaru:

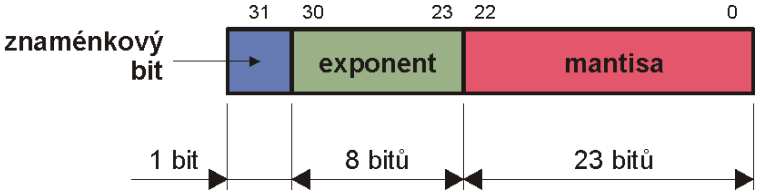
$$mantisa \times základ^{exponent}$$

Základem exponentu bývá většinou číslo 2, 10 nebo 16

(což odpovídá dvojkové, desítkové a šestnáctkové soustavě). Název pohyblivá respektive plovoucí vznikl z toho, že se desetinná čárka pohybuje - je umístěna kdekoliv relativně k platným číslicím. Tato pozice je interně uložena separátně. Výhodou reprezentace s plovoucí místo s pevnou desetinnou čárkou (popř. integery) je mnohem širší oblast hodnot. Formát plovoucí desetinné čárky vyžaduje o trochu více paměti (k zakódování pozice desetinné čárky), proto při uložení ve stejném prostoru mají čísla s plovoucí desetinnou čárkou menší rozsah, ale větší přesnost než čísla s čárkou pevnou. Rychlost operací prováděných s čísly s plovoucí desetinnou čárkou je důležitým měřítkem rychlosti počítačů v mnoha oblastech. Měří se v jednotce FLOPS (operace s plov. des. čárkou za vteřinu). Příklad vyjádření desetinné aproximace čísla pí:

$$31415 \times 10^{-4} = 31425 \times 0,0001 = 3,1415$$

V paměti jsou reálná čísla uložena jako jedno velké bitové pole rozdělené po 8 bitech:



6.4 Znakové literály

Znak je písmeno, číslice, interpunkční zamínko nebo jakýkoliv jiný znak definovaný v sadě Unicode. Ale je rozdíl mezi číslicí vyjadřující znak a číslicí vyjadřující hodnotu. Příklad: Ulice Nádražní 123. Věta obsahuje číslo chápané jako znak ale výraz 10+5 je chápáno jako číselná hodnota. Znaky se do programu zapisují tak, že se umístí mezi dva apostrofy. Příklad: 'L'

Znaková sada nebo také kódová stránka je kód, který každému znaku (abecedy) přiřazuje určité číslo (bajt, sekvenci elektrických pulzů ap.). Výraz sada odráží to, že obsahuje kód pro určitou množinu znaků (např. obsahuje latinku bez nebo s určitými diakritickými znaménky), kód všech existujících znaků zahrnuje standard Unicode. Převod textu do posloupnosti (sekvence) čísel a zpět slouží pro ukládání textu v počítači, jeho přenos telekomunikačními sítěmi apod.

Unicode je tabulka znaků všech existujících abaced, která v současnosti obsahuje více než 110 000 znaků. Unicode umožňuje pracovat se znaky všech písem i různými jinými symboly stejným způsobem, takže mohou být využívány současně.

Ne všechny znaky se dají při výpisu vytisknout na obrazovku. Těmto znakům se říká netisknutelné znaky a pomocí nich se zadávají do programu speciální funkční znaky. Netisknutelným znakem jsou například apostrofy, kterými se tisknou znakové literály. Při výpisu se nezobrazují, ale kompilátor podle nich zjistí že má vytisknou znak který se nachází uvnitř. Netisknutelné znaky představují prvních 32 znaků znakové sady unicode. Netisknutelné znaky se do programu zapisují pomocí znaku escape, za kterým následuje netisknutelný znak. Ve většině programovacích jazyků se jako znak escape používá zpětné lomítko " \". Například:

```
'\n'
```

Výsledkem bude vytištění znaku odřádkování (text se odřádkuje jako při stisknutí klávesy Enter).

6.6 Řetězcové literály

Řetězec je posloupnost znaků například: Karel. Řetězcový literál se do programu zapisuje tak, že se řada znaků tvořící textový řetězec umístí na jednom řádku do uvozovek. Příklad zápisu řetězcového literálu:

```
"Příklad řetězcového literálu."
```

Kapitola 7
Proměnné

Operační paměť je elektricky závislá paměť s rychlým přístupem, do které procesor může přistupovat přímo. Paměť je tvořena paměťovými buňkami které mohou nést základní informaci bit. Bit definuje dva stavy logická 1 nebo logická 0. V hardwarovém pojetí se jedná o stavy zapnuto a vypnuto. Protože je v praxi potřeba zaznamenat více stavů než jen vypnuto nebo zapnuto, jsou jednotlivé paměťové buňky shlukovány do skupin po 8 bitech. Každým 8 bitům se bajt. Díky tomu je možné vytvořit více kombinací stavů než jen zapnuto nebo vypnuto. S 8 bity je možné vytvořit 256 kombinací, co dává možnost uložit hodnotu 0-255. Každý bajt, respektive osmice bitů v paměti je identifikována pomocí jedinečné paměťové adresy, pomocí které je možné k nim přistupovat, číst a měnit jejich hodnotu.

Proměnné jsou základem každého programovacího jazyka. Jedná o místo v paměti identifikované paměťovou adresou. Pro usnadnění orientace ve zdrojovém kódu je toto paměťové místo pojmenováno jednoznačným slovním identifikátorem - **identifikátor proměnné**, který představuje nějaké symbolické jméno související s účelem existence dané

proměnné. Pro přístup k tomuto paměťovému místu se přistupuje pomocí identifikátoru proměnné. To výrazně zjednodušuje práci s pamětí a zpřehledňuje zdrojový kód.

Proměnné slouží k ukládání dat v programu, například výsledky a mezivýsledky výpočtů, informace o různých entitách, řetězce, ... Většina programovacích jazyků je tzv. *case sensitive*, to znamená, že rozlišuje velká a malá písmena. Z toho vyplývá, že názvy *PROMENA* a *promena* jsou dva různé slovní identifikátory a nelze je zaměňovat (mohou v programu vzájemně koexistovat). Délka identifikátoru proměnné může být maximálně 31 znaků, což bohatě stačí na libovolný název. Prvním znakem identifikátoru nesmí být podtržítko, ale jinak se v názvu identifikátoru smí vyskytovat spolu s libovolnou kombinací písmen a čísel. Název identifikátoru také nesmí kolidovat s názvy již existujících příkazů (if, return, ...).

7.1 Jednotky pro měření paměti

Pro měření kapacity a velikosti paměti se používá jednotka **bit**. Bit je základní a nejmenší jednotka dat. Jedná se o pojem, který pochází z teorie informace, která se zabývá matematickým popisem informací. Bity se značí malým písmenem *b*, například *8b*, ale běžně se užívá i celo-slovní název *bit*, například *8 bit*. Bit může nabývat pouze jedné ze dvou hodnot, nejběžnější znázornění těchto hodnot je 0 a

1.

Obě hodnoty mohou také být interpretovány jako logické hodnoty (true / false, ano / ne), stavy aktivace (zapnuto / vypnuto), nebo jakýkoli jiný dvouhodnotový atribut. Pro zaznamenání složitějších dat se používá kombinace více bitu s určitým způsobem kódování dat.

Bit se v souvislosti s měřením množství pamětí používá nejčastěji při měření **velikosti uložených dat**, které jsou v daném okamžiku uloženy na konkrétním paměťovém médiu. Dále jako **kapacita paměti** daného paměťového média, tedy jaké množství informací je možné na dané paměťové médium uložit. Pokud je kapacita paměti podělena časem je získána **přenosová rychlost**, která udává kolik bitu je možné přenést přes dané přenosové médium za jednotku času. Přenosová rychlost se udává v jednotkách *bit/s* (také jako bfs z anglického *bit peer second*).

7.1.1 Byte

Byte nebo česky **Bajt** (méně často slabika) je označení pro jednotku pro měření množství dat. Nejčastěji se jedná o osmici bitů, které tak umožňují uchovat 256 různých kombinací stavů jednotlivých bitů. Jeden byte tak může uchovávat nejrůznější informace, například celé číslo z intervalu 0 až 255, nebo přirozené číslo z intervalu -128 až 127, jeden znak, programovou instrukci nebo cokoli jiného. Jeden byte

je obvykle nejmenší množstvím dat se kterým dokáže přímo pracovat. K tomu jsou již přizpůsobena paměťová média, která zpravidla neumožňují přístup k jednotlivým bitů, ale pouze k jednotlivým bytům, nebo jejich násobcích.

U jiných počítačů může jeden byte označovat i jiný počet bitů (jedná se o historický pozůstatek z prvních stolních počítačů), proto může být jeden byte označován termínem **oktet**.

Byte se značí velkým písmenem *B* (na rozdíl od bitu, který se značí malým písmen b) například *10B*.

7.1.2 Word

7.1.3 Nibble

7.2 Datové typy

V programech je většinou potřeba pracovat s čísly, která jsou větší než hodnota 255. To přináší nutnost sdělit počítači kolik paměti je potřeba použít, aby se do dané proměnné vešel požadovaný rozsah hodnot. **Abstraktní datový typ** je klíčovým pojmem při vytváření proměnných. Každá proměnná je kromě svého identifikátoru ještě určena datovým typem, který definuje množství přidělené paměti a operace, které je možné s touto proměnnou vykonávat. Datový typ je soubor syntaktických a sémantických

pravidel. Existuje velmi úzká souvislost mezi datovým typem a datovou strukturou. Každá datová struktura má datový typ, ale ne každý datový typ je strukturou. Datový typ slouží k definování datových struktur, jsou jejich základem, ale abstraktní datové typy nejsou datové struktury. Datové typy lze rozdělit na **abstraktní datové typy** a **uživatelsky definované datové typy**. Abstraktní datové typy jsou základní datové typy, kterými disponuje daný programovací jazyk. Pomocí abstraktních datových typů jsou vytvářeny uživatelsky definované datové typy.

Abstraktní datové typy jsou:

- **char** - 8-bitová proměnná pro uložení jednoho znaku
- **short** - 16-bitová proměnná pro uložení celého čísla
- **int** - 32-bitová proměnná pro uložení celého čísla
- **long** - 32-bitová proměnná pro uložení celého čísla
- **long long** - 64-bitová proměnná pro uložení celého čísla
- **float** - 32-bitová proměnná pro uložení reálného čísla
- **double** - 64-bitová proměnná pro uložení reálného čísla
- **long double** - 96-bitová proměnná pro uložení reálného čísla

Uživatelsky definované datové typy jsou vytvářeny spo-

jením více abstraktních datových typů a nebo jiných uživatelsky definovaných datových typů.

Rozlišují se dva druhy programovacích jazyků - typové programovací jazyky a typově dynamické programovací jazyky. V případě typového programovacího jazyka má každá proměnná pevně určený datový typ a nemůže v ní být uložena hodnota jiná než odpovídající tomuto datovému typu. Datový typ je obor hodnot, kterých může proměnná nabývat. Typově dynamické programovací jazyky naproti tomu umožňují definovat proměnnou bez udání jejího datového typu a teprve po inicializaci dané proměnné je podle typu uložených dat rozhodnuto o výsledném datovém typu proměnné. Typově dynamické programovací jazyky jsou především skriptovací a interpretované programovací jazyky.

7.3 Runtime type information

Runtime type information (RTTI), česky *informace o typu za běhu* je metoda, která je v softwarovém inženýrství používána pro odhalení datového typu objektu za běhu programu.

7.4 Datová struktura

Datová struktura udává jak jsou data organizovány v paměti počítače (struktura dat v paměti). Data v paměti

mohou být uložena podle určitého vzoru, který je udán právě datovou strukturou. Obvykle se datová struktura skládá z několika (alespoň dvou) proměnných abstraktního datového typu, ale může se jednat také o jiné uživatelsky definované proměnné tvořící jinou datovou strukturu. Díky tomu se mohou jednoduše vytvářet složitější datové struktury spojováním jednodušších. Datová struktura udává jak jsou proměnné abstraktních datových typů uloženy v daném úseku paměti. Datová struktura se pojí s pojmem uživatelsky definované datové typy.

7.5 Datový typ void

Typ **void** není vlastně tak docela datový typ. Jedná se o neúplný datový typ jehož definice (přidělení paměti) nemůže být dokončeno. Typ void je možné použít jako návratový typ funkce, který nevrací žádnou hodnotu. V takovém případě se nejedná o funkci ale o proceduru (funkce bez návratové hodnoty). Dále se používá u ukazatelů kdy není předem známé na jaký typ bude ukazatel ukazovat. Ukazatel na typ void může ukazovat na jakýkoliv datový typ na který je následně přetypován.

7.6 Deklarace a definice proměnné

Při vytváření proměnných v programu jsou rozlišovány dva pojmy - deklarace a definice. **Deklarace proměnné**

znamená, že překladači jsou předány datové typy a identifikátory nových proměnných. Samotné proměnné však prozatím nejsou vytvořeny, to znamená že prozatím není pro proměnnou alokována žádná paměť, ale pouze je oznámen jejich budoucí vznik.

Definice proměnné znamená, že deklarace proměnné je doplněna ještě o inicializaci. Inicializace znamená, že do proměnné je načtena inicializační (počáteční) hodnota a přitom je proměnné vymezen (alokován) prostor paměti. Proměnná musí být definovaná před prvním voláním v programu (proměnné není ještě přidělena paměť), jinak překladač zahlásí chybu. Celý postup lze provést bud ve dvou nebo v jenom kroku (nejprve deklarace a později definice a nebo deklarace a současně definice).

7.7 Operátory a operandy

Při programování se rozlišují dva pojmy spojené s aritmetickými a logickými operacemi s proměnnými. Pro práci s proměnnými se používají tzv. **operátory**. To jsou znaménka, která definují nějakou logickou nebo aritmetickou operaci. **Operandy** jsou čísla, se kterými se v programu pracuje prostřednictvím operátorů. Operátory a operandy společně v programu vytvářejí **matematické** nebo **logické výrazy**.

7.8 Signet a unsigned proměnné

V proměnných jsou ukládána čísla, u kterých msb (most significant bit) reprezentuje znaménko čísla (doplňkový kód), taková proměnná se nazývá **signed (znaménková) proměnná**. Ale tím že je jeden bit odebrán pro reprezentaci znaménka se také snižuje maximální hodnota, kterou lze do proměnné uložit. Například u 8-bitové proměnné je na uložení hodnot použito pouze 7 bitů. Protože $2^7 = 128$, lze tak do takové proměnné uložit hodnoty v intervalu -128 *až* 127.

Unsigned (bez znaménkové) proměnné nepoužívají msb pro reprezentaci znaménka a proto se předpokládá že jsou to vždy kladné hodnoty. Díky tomu pak proměnná využívá všechny bity pro uložení své hodnoty. Proto v případě 8 bitové proměnné protože platí $2^8 = 256$, lze uložit hodnoty v intervalu 0 *až* 255.

7.8.1 Přetečení proměnné

K přetečení proměnné dochází v případě, že je do ní uložena hodnota, která je větší než, je maximum proměnné, dané množstvím alokované paměti. Toto maximum je při stejném množství alokované paměti různě u proměnných signed a unsigned, kvůli msb, který je užíván k definici znaménka u signed proměnných. U 8-bitové proměnné je pak místo $2^8 - 1 = 255$ (počítá se od nuly) u 8-bitové

znaménkové proměnné je možné uložit pouze $2^7 - 1 = 127$

Tyto proměnné signed a unsigned se také chovají různě v případě, že nastane přetečení. Unsigned proměnné nemohou uchovávat zápornou hodnotu, proto v případě, že dojde k přetečení je v proměnné uložena hodnota o kterou byla maximální hodnota proměnné překročena:

$$\left[\begin{array}{cccccccccc} 8 & 7(msb) & 6 & 5 & 4 & 3 & 2 & 1 & 0 & hodnota \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 255 \\ & + & & & & & & & 1 & \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & + & & & & & & & 1 & \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right].$$

Signed proměnné se při přetečení chovají úplně jinak a proto je potřeba s tím při používání znaménkových proměnných počítat. Pro reprezentaci záporné hodnoty se v počítačích používá jiný způsob kódování čísel - **dvojkový doplněk**, který využívá msb k reprezentaci znaménka. Pokud je msb rovno 0, číslo uložené v proměnné je kladné, pokud je msb rovno 1, číslo v proměnné je záporné. Tím že číslo 127 (mezní hodnota 8-bitové signed proměnné) přeteče, je do proměnné uložena hodnota -128. Protože 128 v binárním kódu je zároveň -128 v kódu dvojkového doplňku, je při přetečení hodnota 8-bitové proměnné počítačem interpretována ne jako hodnotu 128, ale jako -128. Při přetečení

je do proměnné uložena hodnota o kterou byla maximální hodnota proměnné překročena, ke které je přičtena hodnota -128:

$$\left[\begin{array}{cccccccccc} 7(msb) & 6 & 5 & 4 & 3 & 2 & 1 & 0 & hodnota \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 127 \\ + & & & & & & & 1 & \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -128 \\ + & & & & & & & 1 & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -127 \end{array}\right].$$

7.9 Konstantní proměnné

V případě, že se daná hodnota proměnné nesmí v průběhu vykonávání programu měnit a je třeba ji před touto eventualitou ochránit je možné ji definovat jako konstantu. Jedná se o spojení hodnoty dané proměnné se jménem. To umožňuje lépe udržet přehled ve zdrojovém kódu. Jedná se o alternativu k programovým makrumů. Některé programovací jazyky ale nedisponují preprocesingem a není tak možné makra definovat. Konstantní proměnné jsou tak jediná možnost jak spojit jméno s danou hodnotou.

7.10 Rozsah platnosti proměnných

Bloková struktura je označení pro zdrojový kód programu, který je rozčleněn do samostatných **bloků kódu** (úseků) používaných ve strukturovaném programování. Blok kódu umožňuje v programování specifikovat část kódu programu, která se vůči okolí chová tak, jako by byla jedním příkazem. Každý blok je definován uvozovacím a ukončovacím příkazem. Proměnné, které jsou definované (deklarované) uvnitř jakéhokoli bloku kódu se nazývají **lokální proměnné**. Blok kódu definuje platnost daného příkazu, ale i rozsah platnosti proměnné deklarované uvnitř bloku kódu od místa její definice (deklarace) do konce daného bloku (nebo do jejího zrušení). Počítač rezervuje paměť pro proměnou pouze po tu dobu dokud program zůstává v rozsahu platnosti.

7.11 Globální proměnné

Vedle lokální proměnné se v programu rozlišují také globální proměnné. Proměnné definované vně jakéhokoliv bloku kódu se nazývají **globální proměnné** a mají platnost (jsou dostupné) v rámci celého zdrojového kódu (v závislosti na nastavených přístupových právech). Pokud je ve funkci lokální proměnná se stejným názvem jako globální proměnná, lokální proměnná skryje (překryje) globální a je využívána lokální proměnná - lokální proměnná má vyšší prioritu než globální. Lokální proměnné je třeba v procesu definice inicializovat na počáteční hodnotu (to zajistí

alogkaci paměti). V případě, že globální proměnné nejsou při při deklaraci inicializovány na počáteční hodnotu, jsou automaticky inicializovány na hodnotu 0.

7.12 Typová konverze

Pod pojmem typová konverze se míní převod proměnné určitého datového typu na jiný datový typ. V kontextu programovacích jazyků se rozlišují dva druhy typové konverze:

- Implicitní - samovolná, automatická
- Explicitní - vynucená

7.12.1 Implicitní datová konverze

Implicitní datová konverze se vyskytuje především při vyhodnocování aritmetických výrazů, které obsahují operandy (proměnné) různých datových typů. V případě že se dané aritmetické operace zúčastní dva operandy s různými datovými typy, konvertuje se typ operandu s nižší prioritou na typ s prioritou vyšší:

char $\rightarrow$ int

short int $\rightarrow$ int

int $\rightarrow$ unsigned int

unsigned int $\rightarrow$ long

long → unsigned log

unsigned long → double

float → double

double → long double

long double → nejvyšší priorita

7.12.2 Explicitní datová konverze

Explicitní datová konverze se též nazývá **přetypování**, které znamená, že výraz nebo proměnná je v čase překladu konvertována na požadovaný typ. Přetypování je cílená změna datového typu proměnné nebo výrazu na jiný datový typ.

7.13 Alokace paměti

Každá proměnná musí mít během své existence přidělen paměťový prostor, který odpovídá datovému typu proměnné. Identifikátor, jméno proměnné je symbolická adresa tohoto paměťového prostoru. Identifikátor je při kompilaci přeložen na paměťovou adresu v binární podobě, která je vložena do všech míst v programu, kde je daný identifikátor použit. Akce, která vyhrazuje paměťový prostor se nazývá **alokace**.

7.13.1 Statická alokace paměti

U statické alokace paměti je předem daná paměť, kterou nelze překročit. Statická alokace je prováděna loaderem, což je systémový nástroj (zavaděč), který umožňuje zavést programy do paměti počítače. Loader přečte datové typy všech proměnných v programu a přidělí jim potřebnou paměť. Velikost této přidělené paměti je statická, to znamená, že se v průběhu vykonávání programu již nemění a není přidávána ani odebírána další paměť.

7.13.2 Dynamická alokace paměti

V případě dynamické alokace paměti si program (programátor) sám určí kolik paměti právě potřebuje a to dynamicky až za běhu programu. V případě že dynamicky alokovanou paměť daný program již nepotřebuje může ji (a měl by) vrátit, aby mohla být recyklována a znovu použita pro v jiné části programu nebo v případě multitaskingového systému jiným programem. Při této alokaci ale nevzniká žádný identifikátor a výsledkem příkazu na alokaci je vrácení paměťové adresy do které se přistupuje pomocí ukazatelů (pointerů).

Únik paměti anglicky memory leak je situace kdy počítačový program dynamicky alokuje paměť a není schopen ji vrátit poté co ji už nepotřebuje. Negativní vliv se zvyšuje, pokud dochází k opakovanému úniku paměti, což může způsobit zřetelné zpomalení počítače nebo dokonce až vyčer-

pání veškeré dostupné paměti (OOM - out of memory), které může nakonec způsobit násilné ukončení programů nebo i fatální selhání počítače.

Při programování jsou úniky paměti poměrně běžnou chybou, obzvlášť v případě programovacích jazyků, neobsahujících zabudovanou automatickou správu paměti, jako jsou například C a C++. Programátor musí v těchto jazycích zajistit uvolňování veškeré alokované paměti vlastním kódem, a proto k chybám a opomenutím dochází.

7.14 Paměťové třídy

Paměťové třídy určují ve které části paměti bude proměnná kompilátorem umístěna a kde v programu bude viditelná. Rozlišují se:

- auto
- extern
- static
- registr

7.14.1 Paměťová třída auto

Třída auto je implicitní paměťová třída pro lokální proměnné. Proměnná typu auto začíná na začátku funkce a končí s výstupem z funkce. Proměnná paměťové třídy

auto si nepřenáší datovou část mezi funkcemi, protože je alokována v zásobníkové části paměti.

7.14.2 Paměťová třída extern

Třída extern je implicitní paměťová třída pro globální proměnné. Tyto proměnné jsou uloženy v datové části paměti RAM. Tyto proměnné jsou používány při odděleném překladu kdy dva nebo více souborů sdílejí stejnou proměnnou.

7.14.3 Paměťová třída static

Proměnné paměťové třídy static jsou uloženy v datové části paměti RAM a u jejich definice musí být vždy přítomno klíčové slovo static. Statické proměnné mají dvě oblastí použití:

• Pro globální proměnné, nebo funkce, které jsou ale viditelné pouze v souboru ve kterém jsou definované. Pokud je proměnná nebo funkce v jednom souboru definovaná jako static a v druhém (se stejným jménem) jako extern, překladač nahlásí chybu, protože deklarace static nezmění paměťovou třídu dané proměnné a překladač si bude myslet, že se jedná o pokus přístupu ke statické proměnné v jiném souboru než byla definovaná.

• Pro lokální proměnné které si ponechají svoji hodnotu i

mezi jednotlivými voláními dané funkce. Tato proměnná je opět přístupná pouze v této funkci a existují od prvního volání dané funkce do ukončení programu.

7.14.4 Paměťová třída registr

Některé programovací jazyky umožňují uložení některé z proměnných místo do operační paměti přímo do registru procesoru. To má za výhodu rychlejší přístup k uloženým datům. Pomocí klíčového slova registr je se proměnná naváže na registr, který je určen překladačem. Tato proměnná ale vznikne pouze pokud je některý z registrů procesoru volný. Pokud žádný registr volný není vznikne obyčejná proměnná. Tato proměnná se používá pouze u lokálních proměnných.

7.15 Konstanty

Kapitola 8
Příkazy a výrazy

Výraz je v kontextu programování spojením alespoň jednoho operátoru a několika operandů (podle arity dané operace). U programovacích jazyků se nejčastěji rozlišují:

- Aritmetické operátory
- Relační operátory
- Bitové (logické) operátory
- Ostatní

8.1 Aritmetické operátory

Aritmetické operátory slouží k vykonávání aritmetických operací mezi operandy a tvoří základ pro vykonávání složitějších algoritmů. Mezi aritmetické operátory patří:

- součet
- rozdíl
- součin
- podíl
- modulo

- inkrementace
- dekrementace

Mezi zajímavější aritmetické operátory patří operandy modulo, inkrementace a dekrementace. Operace **modulo** vrací jako výsledek zbytek po dělení. Operace inkrementace a dekrementace jsou unární operace, to znamená, že pracují pouze s jedním operandem. Operace **inkrementace** přičte hodnotu jedna k danému operandu a operace **dekrementace** naopak odečte hodnotu jedna od daného operandu. To se velice hodí na řízení cyklů a procházení prvků polí.

8.2 Relační operátory

Relační operátory jsou operátory, které říkají jaký je vztah mezi dvěma hodnotami. Relační operátory se používají v podmíněných výrazech, které umožňují rozhodnout co program bude dále dělat. Všechny relační operátory vracejí jako výsledek hodnotu 0 (true) nebo 1 (false). V případě pozitivního výsledku je vrácena hodnota 1 a v případě negativního výsledku hodnota 0. Mezi nejčastější relační operátory patří:

- větší než - $x > y$
- menší než - $x < y$
- je rovno - $x = y$

- není rovno - $x \neq y$
- větší nebo rovno - $x \geq y$
- menší nebo rovno - $x \leq y$

8.3 Bitové operátory

Bitové operátory umožňují přímo manipulovat s bity tvořící binární čísla (data). To je často využíváno především při nízkoúrovňovém programování kdy je třeba manipulovat s hodnotami na vstupně-výstupních pinech, konfiguračních registrech procesoru, ... Nejčastěji používané bitové operace jsou:

- Bitový (logický) součin - AND
- Bitový (logický) součet - OR
- Bitový (logický) exkluzivní součet - XOR
- Bitová negace
- Logická negace - NOT
- Bitový posun doprava
- Bitový posun doleva

8.4 Příkazy

Příkaz je nejmenší samostatný prvek programu v

imperativních programovacích jazycích vyjadřující nějakou činnost, která má být provedena. Na klíčová slova, operátory a operandy lze nahlížet jako na slova, kterým Kompilátor rozumí. Výrazy pak představují způsob jak tato slova organizovat a předávat tak procesoru určité požadavky, které má vykonat. Ale dříve než je požadavek splněn je třeba jej umístit do příkazu. Příkaz je věta skládající se z výrazů dávající procesoru instrukce. Příkladem příkazu, který se vyskytuje v každém programovacím jazyce je příkaz na přiřazení hodnoty do proměnné, nebo deklarace/definice proměnné.

Kapitola 9
Ukazatelé

Pointery, nebo také **ukazatelé** jsou speciální celočíselné proměnné, které v sobě uchovávají paměťovou adresu nějaké proměnné (objektu), nebo podprogramu. Ukazatelé s sebou zároveň nesou informaci o datovém typu proměnné, která se na této adrese nachází. To je z důvodu využití aritmetiky ukazatelů, která umožňuje pohodlněji pracovat s pamětí uloženou v ukazateli. Ukazatel je tedy vždy svázán s nějakým datovým typem (doménový typ). Potom je řeč o *ukazateli na daný typ*. Doménový typ ukazatele musí odpovídat zkutečnému typu objektu, na který ukazatel ukazuje, jinak by program nesprávně interpretoval hodnoty, které jsou v něm uloženy.

9.1 Adresy v paměti

Procesor disponuje sérií speciálních registrů (ukazovací registry), které umožňují uchovávat adresu paměti a skrze tuto adresu manipulovat s daty uloženými na dané adrese.

Paměť počítače je rozdělena do sekvenčně číslovaných paměťových buněk. Každá proměnná je uložená na jedinečném místě v paměti, které se označuje její adresou.

Různé počítače tuto paměť číslují podle různých složitých schémat. Obvykle není třeba tyto informace znát, protože se o tyto věci stará překladač. Pokud je ale třeba tuto adresu zjistit je možné použít **referenční operátor**, který umožňuje získat adresu daného objektu.

V opačném případě může být někdy potřeba přistupovat k paměti na kterou daný ukazatel ukazuje. K tomu slouží **dereferenční operátor** nebo také operátor nepřímého přístupu. Ukazatel poskytuje nepřímí přístup k hodnotě proměnné, jejíž adresu uchovává. Pokud by nebyl použit operátor nepřímého přístupu nebylo by přistupováno k hodnotám proměnné, ale k adrese uložené v tomto ukazateli.

9.2 Ukazatel na typ void

Ukazatel na typ void slouží pro postupné využívání adres proměnných různých typů. To znamená že ukazatel na typ void ukazuje na jakýkoliv datový typ. Protože ukazatelé jsou druhem proměnných je možné je stejným (nebo podobným) způsobem přetypovat na jiný doménový typ. Práce s těmito ukazateli je stejná jako s klasickými ukazateli, ale s rozdílem, že při použití operátoru nepřímého přístupu je nutné ukazatele přetypovat na správný doménový typ.

9.3 Ukazatelé jako argumenty podprogramů

Při předávání argumentů funkcím hodnotou uloženou v nějaké proměnné, není uvnitř funkce manipulováno s hodnotou dané proměnné, ale pouze s hodnotou, která byla do podprogramu "nakopírována". Ale pokud jsou data do funkce předány jako adresa ukazatele je možné pomocí operátoru nepřímého přístupu přímo ovlivňovat hodnotu uloženou v dané proměnné.

Uvnitř zásobníkového rámce daného podprogramu je sice opět vytvořena lokální proměnná, ale typu ukazatel, která v sobě uchovává paměťovou adresu dané proměnné. Tento ukazatel je opět zrušen spolu s ostatními lokálními proměnnými při opouštění těla podprogramu. Při volání podprogramu je třeba jako argumenty podprogramu předat adresy proměnných pomocí referenčního operátoru.

Předávání ukazatele jako argument podprogramu se také velmi často využívá pro rozšíření možností návratové hodnoty podprogramu. Díky ukazatelům je možné, aby jeden podprogram vracel více než jeden výsledek. Ukazatelé na návratové hodnoty podprogramu jsou do podprogramu vloženy jako argumenty a běžný výstup podprogramu je možné využít k výpisu chybových zpráv.

9.4 Aritmetika ukazatelů

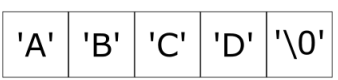
9.5 Ukazatelé na ukazatele

9.6 Ukazatelé na podprogramy

9.7 Argumenty příkazového řádku

Kapitola 10
Řetězce

Řetězec (string) je speciální použití jednorozměrného pole datového typu char (znakové pole). Každý prvek pole obsahuje právě jeden znak. Některé programovací jazyky vyšší úrovně zaobalují pole typu řetězec do tříd, které poskytují širokou škálu funkcí/metod, které výrazně zjednodušují a urychlují práci s řetězci a snižují ryziko chyb. Na druhou stranu mohou výsledný program mírně spomalovat, nebo nemusejí umožňovat nízkoúrovňovou práci s nad řetězcovým polem.



Na konci každého řetězce se nachází znaménko '\0'. Jedná se o řídící znak z escape sekvence, který má v ASCII hodnotu nula. Tento znak se používá jako ukončovací znak, který umožňuje najít konec řetězce v paměti u dynamicky vytvářených řetězců, u kterých nelze předem převídat počet znaků. Kdyby řetězec nebyl ukončen *'ukončovací nulou'*,

nebylo by možné určit kde daný řetězec končí a algoritmy, které pracují s řetězci by mohly zasahovat do paměti mimo řetězcové pole. Z toho vyplývají některé zkutečnosti:

- Díky tomu může mít řetězec libovolnou (statickou) délku omezenou pouze velikostí paměti.
- Při alokaci paměti pro řetězec je důsledkem toho potřeba alokovat o jeden bajt paměti více pro znak '\0'.
- V případě, že na konec řetězce není vložen ukončovací znak '\0' je za řetězce považován zbytek paměti až do místa kde se v paměti objeví ukončovací znak nula. To vede k chybě v programu obzvlášť v případě, že je do této paměti zapisováno.

Kapitola 11
Binární a textové soubory

V kontextu operačního systému, respektive souborového systému jsou rozlišovány dva druhy souborů:

- Binární soubor
- Textový soubor

11.1 Binární soubor

Binární soubor je soubor uložený na disku počítače, který obsahuje čísla v binární soustavě, která předem určeným způsobem reprezentují uložené informace - zvuk, video, spustitelný program ale i text. Při čtení binárního souboru je třeba vědět jak uložená data interpretovat.

Aby bylo možné identifikovat typ uložených dat a způsob jakým s nimi zacházet, binární soubory často obsahují hlavičky (záhlaví), která uložená data popisuje (metadata).

Za hlavičkou souboru se nacházejí samotná data binárního souboru. Jejich organizace je závislá na tom jak je příslušný program zapisuje (čte).

Při zobrazování binárního souboru v nějakém textovém editoru je každých 8 bitů interpretováno jako samostatný znak. V textovém editoru bude pravděpodobně obsah jako řada nesmysluplných znaků.

Výhodou binárních souborů je rychlá práce s nimi (nic se neformátuje), zabírají méně místa v paměti počítače (pro textovou reprezentaci čísla je třeba použít pro každý číselný řád jeden byte). Jedinou nevýhodou binárního souboru je jeho čitelnost běžnými textovými editory.

11.2 Textový soubor

Textový soubor je opakem binárního souboru. Textový soubor je soubor uložených dat v paměti počítače, který obsahuje pouze textová data. Textová data jsou složena výhradně z tisknutelných znaků. Interpretace jednotlivých znaků závisí pouze na zvoleném kódování daného textového editoru.

Kromě tisknutelných znaků může textový soubor obsahovat ještě tzv. **bílé znaky**. V běžném textovém souboru se nacházejí pouze tři bílé znaky:

- mezera
- tabulátor

- odřádkování

Textový soubor není prostou lineární posloupností znaků, ale **je členěn do jednotlivých řádků** (odstavců). Řádky mohou mít proměnný počet znaků a každý řádek je ukončen znakem konce řádku.

Vnitřně je textový soubor reprezentován binárními daty → textový soubor je formátovaný binární soubor. Při vytváření/zápisu/čtení jsou použity funkce, které binární data, především číselné hodnoty formátují do příslušné podoby znakové sady.

Kapitola 12
Procesy a podprogramy

Proces je v počítači spuštěný program zavedený do operační paměti. V operační paměti je rozdělen do několika logických částí (celků), které obsahují určité části programu. Jeden program respektive proces může být rozdělen na více podprogramů. Podprogram je část programu na kterou je v programu odkazováno. To umožňuje zkrátit a zpřehlednit zdrojový kód programu a velikost výsledného spustitelného kódu.

12.1 Vykonávání programového kódu

V procesoru je program reprezentován jako série po sobě jdoucích strojových instrukcí reprezentující základní operace, které je procesor schopen vykonat. Cyklus vykonávání strojového kódu je většinou podobný a skládá se z několika kroků:

- Řídící jednotka procesoru načte instrukci v paměti na kterou ukazuje registr IP (ukazatel instrukce)
- Hodnota v registru IP se zvýší o hodnotu vyjadřující bajtovou hodnotu instrukcí - počet bajtů které zabírá jedna

instrukce

- Vykoná se instrukce přečtená v prvním kroku
- Celý cyklus se opakuje

V případě že při vykonávání instrukcí nastane skok v programu procesor jednoduše přejde do prvního bodu a přepíše hodnotu v registru *IP* a dál pokračuje ve vykonávání programu na jiném místě paměti.

12.2 Popis výkonu podprogramů

Zvláštním případem jsou podprogramy, které se vykonávají trochu jiným způsobem:

- Hodnota adresy v ukazateli instrukce se zvýší o jedničku na další instrukci, která následuje za voláním funkce (aby při návratu z funkce instrukční čítač neukazoval opět na adresu volání funkce, musí ukazovat na další instrukci v pořadí). Tato adresa je uložena do zásobníkového rámce v zásobníku a stane se návratovou adresou při navracení řízení programu
- V zásobníku se vytvoří prostor pro návratový typ, který byl deklarovaný
- První adresa volané funkce, která je uložená ve speciální

části paměti vyhrazené pro tento účel se načte do ukazatele instrukce, takže další provedená instrukce bude ve volané funkci

• Všechny argumenty funkce se uloží do zásobníku spolu se všemi lokálními proměnnými. Když je funkce připravena vrátit řízení programu umístí návratovou hodnotu do oblasti zásobníku určené pro návratovou hodnotu. Pak se ze zásobníku vymažou všechny lokální proměnné a argumenty funkce. Návratová hodnota se přiřadí volajícímu elementu a odebere se ze zásobníku. Dále program pokračuje tam, kde skončil i s hodnotou získanou z funkce

12.3 Druhy podprogramů

V kontextu softwarového vývoje je rozlišováno několik druhů podprogramů, podle jejich významu a charakteristických vlastností:

- Funkce
- Metoda
- Rutina

Vzájemně jsou si všechny tři pojmy velmi podobné. Označují nějakou **pojmenovanou** skupinu příkazů tvořící

část programu, kterou lze opakovaně volat z jiných částí programu. Každý pojem vznikl v různých programových paradigmatech a jedná se o dědictví, které přejaly novější programovací jazyky.

12.3.1 Funkce

Funkce je základním type podprogramu, která na vstupu může, ale nemusí přijímat libovolné množství vstupních argumentů a na svém konci může, ale nemusí vrátit nějakou návratovou hodnotu jako výsledek své činnosti. To může být buď nějaká vypočítaná hodnota, nebo zpráva o chybě při výkonu funkce.

12.3.2 Metoda

Metoda je pojem z objektově orientovaného programování. Jedná se o běžnou funkci, která je ale součástí nějaké programové třídy. To vnitřně znamená, že musí vždy přijímat jako parametr odkaz (ukazatel) na objekt dané třídy, aby mohla přistupovat k jejím datům (atributům). Výsledkem je, že metoda je funkce, která je svázána s daným datovým typem a nelze ji použít na jinou třídu. To se v OOP označuje jako zapouzdření, které svazuje data funkce dohromady, aby na danou funkci nemohly být použity nesouvisející data stejného typu. V objektově orientovaných jazycích se to většinou děje na pozadí a kompilátor tento

parametr doplní automaticky při překladu.

12.3.3 Procedura

Procedura označuje druh podprogramu, který se podobá běžné funkci. Jedná se o speciální případe funkce, ale narozdíl od funkce nevrací žádnou návratovou hodnotu a nemusí mít ani žádné vstupní argumenty. V programu jsou procedůry volány jako příkazy, které něco vykonají, například něco zapnou, vypnou nebo nastaví. Typickým příkladem jsou řídící procedury, které mohou skrývat funkcionalitu pro ovládání nějakého hardwaru. Ve všech zbývajících ohledech je ale procedůra stejná jako funkce.

12.4 Předávání hodnot do podprogramu

V souvislosti s funkcemi (respektive metodami či procedůrami) se mluví o **parametrech funkce** a **argumentech funkce**. Tyto dva pojmy se v praxi velmi často zaměňují. Argument funkce označuje deklaraci vstupních hodnot, které daná funkce při svém volání přijímá. Jedná se pouze o definici, ale ne o zkutečná data! Naproti tomu parametr funkce označuje reálnou hodnotu, která je předávána funkci při svém volání. Parametr funkce musí být stejného datového typu jako argument funkce, jinak by došlo k chybě při jejím volání.

S parametry a argumenty funkce souvisí způsob

předávání dat do podprogramu. Existují dva způsoby, kterým lze do podprogramu vložit vstupní hodnoty při jeho volání:

- Předávání hodnotou
- Předávání odkazem

Předávání vstupních paramterů hodnotou znamená, že dojde k fyzickému zkopírovnání hodnot předaných na vstup funkce do proměnných deklarovaných v její hlavičce. Jedná se o základní metodu předávání vstupních dat, které ale má několik zásadních nevýhod. Při vkládání rozsáhlých dat dochází k jejich úplnému zkopírování, které zabírá relativně velké množství času. To má za následek snížení výkonu programu. Předávaná data jsou navíc v paměti reprezentovány dvakrát (původní data a data uvnitř podprogramu) a dochází tak k duplicitě dat a plýtvání pamětí. Předávání dat hodnotou je také limitující v tom, že jakákoli operace se s předanými daty uvnitř funkce provede je pouze lokální a v předané proměnné se neprojeví.

Předávání vstupních hodnot pomocí odkazu respektive paměťové adresy je rozšíření způsobu předávání hodnot hodnotou. Místo hodnoty se ale předá paměťová adresa na které jsou data uložena. Díky tomu nezáleží jak velká data

jsou do podprogramu předávána, protože paměťová adresa má vždy fixní velikost. Tím odpadá zdlouhavé kopírování a duplicita dat v paměti. Protože se uvnitř podprogramu pracuje s paměťovou adresou je možné výsledky ukládat přímo do proměnné předané na vstupu. Výsledky práce s proměnnou již nemusejí být pouze lokální.

12.5 Strukturování operační paměti

Zkompilovaný program je rozdělena do několika segmentů: text, data, bss, heap a stack. Každý segment je po spušťění a zavedení do operační paměti uložen v určité části paměti, která je vymezena pro jistý účel.

12.6 Segment text

Segment text nebo také segment kódu slouží pro uložení strojového kódu programu. Vykonávání kódu tohoto segmentu je díky podprogramům a řídícím strukturám nelineární. Jakmile je program spuštěn nastaví se hodnota registru ukazatele instrukce na adresu první instrukce v tomto segmentu. V segmentu text je oprávnění pro zápis zakázáno, protože tento segment se nepoužívá pro zápis hodnot proměnných, ale pouze strojových instrukcí. Tím je zamezeno, aby se modifikoval kód daného programu. Jedná se o bezpečnostní mechanismus procesoru (operačního systému), který tak chrání daný program. Při

pokusu o zápis do segmentu text je uživateli oznámena chyba a daný program je násilně ukončen. Další důležitá vlastnost tohoto segmentu je, že když se do něj nezapisují žádná data má pevnou velikost, která je nastavena při zavádění programu do paměti a v průběhu vykonávání daného procesu je neměnná.

12.7 Segment data a bss

Do segmentu data a segmentu bss se ukládají globální a statické proměnné programu. Do segmentu data se ukládají inicializované globální a statické proměnné a do segmentu bss se ukládají jejich neinicializované protějšky (například proměnná int i; se uloží do segmentu bss). Tyto segmenty mají stejně jako segment text pevnou velikost, protože počet globálních proměnných je v programu dán při zavádění programu do paměti (jejich počet je statický).

12.8 Segment heap

Segment heap nebo také halda je segment programu, který může programátor přímo ovládat. Bloky paměti v tomto segmentu se dají dynamicky za běhu alokovat a použít přesně podle potřeby. Rysem tohoto segmentu je, že nemá pevnou velikost a může se libovolně (v závislosti na možnostech hardwaru) zvětšovat nebo zmenšovat podle toho kolik paměti je v programu potřeba. Aby nedošlo

ke konfliktu paměťových adres je paměť v segmentu heap přidělována pomocí alokačních a dealokačních algoritmů, které rezervují potřebné místo pro použití v programu, popřípadě ji opět navrátí k opětovnému použití v jiném procesu. Halda narůstá směrem dolů k vyšším paměťovým adresám.

12.9 Segment stack

Segment stack nebo také zásobník má také proměnlivou velikost a používá se pro dočasné uložení lokálních proměnných podprogramů a kontextu procesoru při jejich volání. Instrukce tvořící daný podprogram jsou uloženy na určité adrese v segmentu text, ale lokální proměnné, které obsahuje jsou ukládány do segmentu stack. Kromě proměnných se při volání podprogramů do zásobníku ukládá návratová adresa volání podprogramu (následující instrukce za voláním podprogramu). Tyto informace se do zásobníku uloží do jistého druhu záznamu, který se nazývá **zásobníkový rámec**. Zásobník obsahuje mnoho takovýchto zásobníkových rámců. Zásobník roste směrem nahoru k nižším paměťovým adresám. Každý zásobníkový rámec obsahuje parametry podprogramu, jeho lokální proměnné a dva ukazatele, které umožní aby se uložená data nakopírovala opět zpátky na své místo. Ukazatel na uložený rámec SFP - Saved Frame Pointer (aby se zjistilo kde je uložen zásobníkový rámec nadřazeného

podprogramu, tato hodnota je uložena do registru BP ukazatel na vrchol zásobníku) a návratová adresa (pro návrat do míst volání podprogramu).

Kapitola 13
Správa verzí

Verzování je způsob uchovávání historie provedených změn obecně u jakékoli digitální informace. V případě vývoje softwaru je evidováno kdo kdy a jakým způsobem upravil dané řádky zdrojového kódu v jednotlivých verzích ve stádiu vývoje softwarového projektu. To slouží nejenom k úplnému přehledu všech změn, ale také možnost vidět přesný stav sledovaných dat kdykoliv v minulosti a také vrátit se k předchozí verzi daného programu v případě, že během dalšího vývoje dojde k chybám. Každé změně provedené ve zdrojových kódech je přidělováno unikátní číslo, označované většinou jako **číslo revize**.

13.1 Verze

Verze je číselné nebo jmenné označení verze produktu, softwarového projektu. Název nebo číslo verze se obvykle přidávají za název. Číslování obvykle vyjadřuje vývoj produktu. Nejobvyklejším způsobem jsou označovány verze číslem, které je složeno ze dvou nebo tří částí. První je tzv. **major verze** (hlavní číslo verze, když nastala změna, která není zpětně kompatibilní s ostatními (API)), za tečkou následuje **minor verze** (vedlejší číslo verze, když

se přidá funkcionalita se zachováním zpětné kompatibility) a třetí číslo označuje **číslo revize** (záplata neboli patch, opravila chyba a zůstala kompatibilita). Číslo major musí být větší než nula, může být rovno nule jen v případě vývoje první verze daného programu, která ještě není vydána. Čísla se s postupujícím vývojem navyšují, obvykle s ohledem na rozsah provedených změn. Podle čísla verze tak může kdokoliv poznat, která verze je starší a která novější. Zatímco změna nejnižšího čísla (nejvíce vpravo) obvykle označuje *nevýznamné změny*, popřípadě izolované opravy programátorských chyb, označují změny čísla v levé části *zásadnější změny* Typickým příkladem čísla verze je: NázevProgramu 2.3.11. Jakmile se vydá očíslovaná verze programu, **nesmí** se tento program měnit a každá další úprava nebo oprava je vydána pod novou verzí. MAJOR verze s hodnotou 0 (0.y.z.) je určena pro počáteční vývoj. Cokoliv se může změnit a API v tomto formátu by NEMĚLO být považováno za stabilní. Verze 1.0.0 definuje veřejně vydané API. Způsob, jakým se dále navyšuje číslo verze je ovlivněné tímto API a jeho změnami. Nejjednodušším způsobem je začít vývoj na verzi 0.1.0. a potom zvyšovat MINOR verzi při každém dalším vydání softwaru.

Některá stádia vývoje softwaru se mohou označovat dalšími přívlastky, např. alfaverze, betaverze, stable, pre-release (zkratka pre), release candidate (zkratka RC) aj.

Například: NázevProgramu 2.3.11-beta

Alfa verze je verze softwaru, která je zpravidla poskytována pouze v rámci společnosti, která tento software vyvíjí. Jedná se o produkt, který většinou obsahuje všechny důležité funkce, avšak také spoustu chyb. Tato verze je proto testována pouze vývojáři, kteří vědí, jak tento software pracuje. V tomto stádiu vývoje se nalezne a odstraní nejvíce chyb. Jakmile jsou vážné chyby odstraněny, bývá často vydávána Beta verze.

Beta verze je softwarový produkt, na kterém je již opravena většina chyb, nicméně je pořád nestabilní a na jeho chování se nedá spolehnout. Beta verze programu může být dostupná zdarma z různých zdrojů pro účely testování širokou veřejností v rámci tzv. **betatestu** (a to i komerční produkty). Beta verze softwarového projektu většinou odesílá shromažďovaná data pro účely analýzy při zjišťování dalších možných chyb. To, co vznikne z betaverze po odstranění chyb, se nazývá Release Candidate.

Release candidate (většinou se používá zkratka RC) je testovací verze připravovaného programu. Jedná se o kandidáta na konečnou, finální verzi. Ke zkratce RC se přidává i číslice. Vyjít tedy může RC1, RC2, RC3, atd.

Kapitola 14
Dokumentace softwarového projektu

Dokumentace softwarového projektu je důležitá část celého projektu, protože

umožňuje udržet v projektu přehled, naplánovat jej, aby byl minimalizován čas potřebný k jeho realizace, minimalizovat potřebné zdroje na realizaci a umožňuje získat přehled pro jiné vývojáře, kteří v budoucnu mohou daný projekt dále upravovat.

Dokumentace se skládá z několika částí:

- Dokumentace architektury
- Popis realizace projektu
- Zpráva z realizace
- Dokumentace zdrojových kódů

14.1 Životní cyklus projektu

Každý projekt by měl projít určitými stádii realizace. Někdy je možné některý z kroků vynechat, jindy je zase jeho vynechání cestou k problémům při realizaci. V případě, že

celý projekt není dobře promyšlen a naplánován je možné se při jeho realizaci zamotat a výsledkem toho bude nutnost přepracovat část již vytvořeného zdrojového kódu. Projekt by tedy měl projít těmito stádii:

• **Zadání úkolu** - jedná se o výchozí bod, na základě kterého se odvíjí zbytek projektu.

• **Analýza projektu** - jedná se o jednu z nejnáročnějších činností před samotným programováním. je třeba dobře celý projekt promyslet, znát veškeré jeho složky tak, aby při pozdější činnosti nemohlo dojít k nějaké nečekané situaci. Hlavně je důležité v tomto bodě ujasnit veškeré vlastnosti a funkce, kterými má aplikace disponovat. Dobrý programátor již při analýze vidí části kódu, který bude zanedlouho sepisovat. Analýzou je třeba se zabývat opravdu pečlivě.

• **Programování** - realizace projektu podle předem vytvořeného plánu.

• **Ladění** - jedná se o činnost, při které je projekt již hotový a jsou zjišťovány spíše už jen závady a možná vylepšení. S tím úzce souvisí testování.

• **Testování** - jedná se o sadu činností, které mají za cíl najít bugy, které mohou ovlivnit činnost celého programu. Ty občas není možné snadno objevit, ale při používání v praxi je zde určitá procentuální šance, že nastane - to je nepřípustné.

• **Završení projektu** - dokončení projektové dokumentace popřípadě uživatelského manuálu a finální vydání celého programu do plného provozu.

14.2 Dokumentace architektury

14.3 Dynamické generování dokumentace zdrojových kódů

Jednou z možností jak rychle a snadno vytvořit dokumentaci zdrojových kódů je používat v komentářích určité formátování, díky kterému je možné celý zdrojový kód vložit do speciálního programu, který vyjme text z komentářů s tímto formátováním a sestaví z nich tak výslednou dokumentaci. Formátování použité v komentářích je zároveň srozumitelné i člověku a díky tomu jsou splněny hned dva kroky při vývoji programu.

Existuje více programů které umožňují generovat dokumentaci z komentářů zdrojových kódu. Příkladem je QDoc, Doxygen, naturaldoc, ... Zároveň také existuje mnoho stylů formátování komentářů tak, aby z nich bylo možné vygenerovat dokumentaci. Daný program může také podporovat více stylů formátování, ale v praxi je nejlepší použít takový styl, který je zároveň dobře čitelný pro člověka.

Při vytváření komentářů, ze kterých bude následně generována dokumentace se rozlišují různé části zdrojového

kódu, které jsou identifikovány určitou řetězcovou značkou. Tyto značky v komentářích by se měly vkládat pouze do hlavičkových souborů s deklaracemi funkcí, proměnných, konstant, ... Výjimku tvoří soubor s hlavní částí programu (main). Další výjimkou je potřeba do dokumentace zkopírovat část zdrojového kódu. V takovém případě se do zdrojového kódu vloží pomocí formátu:

```
/* start:návěští

nějaký kód

end*/
```

Návěští funguje jako identifikátor kam se má daný kód vložit do komentáře v hlavičkovém souboru.

Formátovací značky je možné vkládat do řádkových i víceřádkových komentářů, ale na začátku každého řádkového komentáře musí být definována nějaká formátovací značka. Z toho vyplývá, že řádkové komentáře se hodí pouze na krátké úseky dokumentace. Důležité je, že v případě, že se v daném komentáři nachází nějaká formátovací značka, považuje se za její parametry všechen zbylý text dokud program nenarazí na další formátovací značku nebo na konec komentáře.

Každá značka má syntaxi:

```
značka:  parametr
```

14.3.1 Značka pro nadpis

Nejvýše položená značka je značka pro nadpis. Nadpis by se měl nacházet na začátku každého hlavičkového souboru, aby v dokumentaci bylo dobře rozlišitelné jednotlivé části programu. V případě, že na začátku hlavičkového souboru nebude tato značka uvedena, budou popisy jednotlivých třídy, funkcí, ... na stejné logické úrovni a bude obtížnější určit v jakém hlavičkovém souboru se nacházejí. Za názvem nadpisu by se měl nacházet popis jeho účelu (knihovna, vrstva programu, ...), volitelně také autor, datum poslední editace, licence, verze, ... Ta je definována pomocí:

```
/* section:  název

text v kapitole

*/
```

14.3.2 Značka pro třídu

Značka pro třídu definuje podsekci nadpisu dokumentace. Třída většinou zapouzdřuje nějaké metody, proměnné, a jiné. Vytváří tak komplexní datovou strukturu s nějakým účelem, který je třeba popsat. Konvencí je vkládat komentář popisující třídu před její hlavičku deklarace Formátovací značka je definována:

```
/* class:  název

popis třídy

*/
```

14.3.3 Značka pro funkce a metody

Protože funkce a metody jsou ve své podstatě téměř to samé je pro ně použita stejná formátovací značka. Tato značka je:

```
/*

function:  název

popis funkce

*/
```

Funkce a metody jsou typické tím, že mohou mít vstupní parametry a návratovou hodnotu. Aby byla hlavička (deklarace) funkce v dokumentaci kompletní je třeba udat jaké má parametry. Každý parametr funkce se definuje pomocí značky:

```
/*

parametr:  datový_typ název - popis

*/
```

U tohoto příkazu je třeba vždy odřádkovat popis proměnné od její deklarace, protože znak odřádkování v programu funguje jako oddělovací znak. Návratová hodnota je u funkce pouze jedna a je zadána pomocí značky:

```
/*

return:  datový_typ - popis

*/
```

V případě, že u funkce není uvedena návratová hodnota, je automaticky předpokládán návratový typ *void*.

14.3.4 Značka pro proměnnou

Proměnná je důležitou součástí všech algoritmů, protože slouží jako úložiště dat v paměti a prostředek pro práci s ní. Rozlišují se globální proměnné a lokální proměnné. Globální proměnné se nacházejí jako první v dokumentaci před funkcemi nebo třídami. Lokální proměnné se nacházejí v části popisující danou funkci nebo třídu (atributy třídy). Značky pro proměnnou je:

```
/*

variable:  datový_typ název
```

```
popis

*/
```

14.3.5 Značka pro struktury

Účel jednotlivých prvků struktury je třeba v programu vysvětlit pro názornost jiným programátorům. Struktura se skládá ze sady proměnných různého datového typu identifikované pod jedním identifikátorem respektive jednou paměťovou adresou. Značka pro uvedení názvu struktury je:

```
/*

structure:  název

popis

*/
```

Pro popis prvků struktury slouží běžná značka pro proměnné. V tomto případě ale také slouží k sestavení deklarace struktury.

14.3.6 Značka pro konstantu

V programu je možné dokumentovat buď každou konstantu zvlášť a nebo sadu souvisejících konstant. Značka

pro dokumentování samostatné konstanty:

```
/*

constant:  NÁZEV KONSTANTY

popis

*/
```

Značka pro dokumentování celé sady souvisejících konstant je:

```
/*

constants:  NÁZEV SKUPINY

popis skupiny

NÁZEV KONSTANTY - popis

NÁZEV KONSTANTY - popis

...

*/
```

U názvů jednotlivých konstant není nutné vkládat jejich popis, ale je nutné dodržovat, že každý popis je na jednom řádku.

14.3.7 Značka pro výčtový typ

Výčtový typ je sada konstant, která je identifikována pomocí jednoho identifikátoru. Značka pro dokumentaci výčtového typu je:

```
/*

enum:  název výčtového typu

popis

NÁZEV KONSTANTY - popis

NÁZEV KONSTANTY - popis

...

*/
```

14.4 Soubory README

Soubory README (čti-mě) jsou speciální soubory, obsahující dokumentaci měnších projektů, které nevyžadují podrobný záznam o vedení a vývoji. Soubory README se používají v případě malých projektů jako je softwarová knihovna, utility a podobně.

Soubory README jsou určeny pro tři kategorie lidí:

• **Uživatelé** - popis rozhraní, kterým je mohou daný produkt používat, popřípadě postup instalace, překladu, ...

• **Kolegové** - vyžadují ne jen popis rozhraní jakým se daný produkt používá, ale také vyžadují popis jakým vnitřně fungují, aby se mohli zapojit do dalšího vývoje.

• **Samotný tvůrce** - lidský mozek přehlcený informacemi po delší době vypustí některé detaily o projektu, které ale mohou být klíčové pro následný vývoj nových rozšíření.

Z tohoto důvodu je nutné u každého projektu vést alespoň základní typ dokumentace, který zajistí rychlé získání potřebných informací.

14.4.1 Anatomie souborů README

V případě souborů README, ale také v případě komplexnějších dokumentací platí, že neexistuje žádný přesně daný standart, nebo popis jak má taková dokumentace vypadat. To je především z toho důvodu, že každý projekt je jiný a vyžaduje jiný způsob (přístup) dokumentace (co může být u jednoho nezbytné může být u jiného triviální). Existuje pouze sada doporučení jak co by měla taková dokumentace obsahovat.

Je důležité, soubory README (ale obecně i rozsáhlejší dokumentace) jsou psány pro ostaní lidi. První co se chtějí

při otevření souboru README dozvědět, zda se jedná o dokumentaci k danému projektu a čeho se daný projekt týká.

Jako první by se tedy v souboru README mělo vyskytovat titulek s názvem projektu pod kterým bude stručný popis cílů, popstupů, ... projektu. Popis by ale neměl být zbytečně rozsáhlý, aby uživatele zbytečně nepletl.

Pro snazší orientaci by dokumentace měla obsahovat obsah, který informuje co se v dokumentaci nachází a v jakém pořadí.

Následuje výpis nezbytných informací, které uživatel potřebuje pro pochopení činnosti projektu. To mohou být reference na dodatečné zdroje, výpis použitých knihoven, ale také výpis známých chyb (bugů), které se prozatím v projektů vyskytují, popis instalace popřípadě překlad zdrojových kódů projektu.

Dokumentace by se neměla změnit v odbornou publikaci vysvětlující do podrobna veškeré pojmy. Dokukumentace by se měla psát s zaměřením na určitou skupinu uživatelů, u kterých se předpokládá určitá úroveň znalostí. Dokumentace by tedy neměla obsahovat informace, které již uživatel zná. Díky tomu je možné napsat méně rozsálý popis daného projektu. V dokumentaci by se tedy měly objevit odpovědi na otázky:

Co je třeba udělat, aby mohl být zdrojový kód projektu použitelný (spustitelný).

Je třeba se na tento problém podívat z perspektivi někoho kdo s daným projektem nikdy před tím nepracoval a neví co má dělat.

To se nejčastěji nachází v samostatné části, která se nazývá instalace (installation sectio) nebo začínáme (getting started).

Další užitečná informace pro uživatele jsou práva podle kterých mohou zdrojový kód používat. V základu si tvůrce projektu drží plná vlastnická práva definující i jeho následné použití. I v případě, že kód projektu není otevřen pro použití ostatními je vhodné (z důvodů možných nedorozumnění) to zdůraznit vypsáním licence a práv, pod která daný projekt spadá. V případě, že je projekt vytvořen pro spolupráci s ostatními uživateli (open source) je nutné tuto zkutečnost zdůraznit, ne jenom licencí ale také záměrem jakým má být na projektu spolupracováno, aby potenciální uživatelé chtěli používat a rozvíjet zdrojový kód daného projektu.

V další sekci je možné vypsat odpovědi na nejčastěji kladné otázky ohledně daného projektu (FAQ - frequently asked questions).

Kapitola 15
Zálohování

Zálohování je proces tvorby záložního zdroje dat pro případ ztráty hlavního zdroje dat. Ke ztrátě hlavního zdroje dat může dojít z nezpočtu příčin a jejich obnova nebo znovu získání nemůsí být jednoduché, levné nebo v nejhorším případě možné. V kontextu softwarového vývoje je dalším dobrým důvodem případ, kdy je při tvořně zdrojového kódu do projektu zanesena chyba (poškození dat) a její lokalizace a oprava by byla časově náročná. V takovém případě se sáhne po starší ale stále aktuální verzi projektu, která následně bude sloužit jako hlavní zdroj dat.

15.1 Zásady zálohování

Při zálohování hraje roli několik základních otázek:

- Kdy zálohovat
- Kam zálohovat
- Co zálohovat

15.2 Zálohovací software

Kapitola 16
Strukturování softwarového projektu

Při vytváření složitějších aplikací (programů) se při neopatrnosti může stát, že se v určitou chvíli zdrojové kódy stanou nepřehledné a sám programátor se v nich začne ztrácet. V ideálním případě by ale zdrojové kódy měly být srozumitelné ne jen pro počítač (překladač) a tvůrce programu, ale i pro ostatní programátory, které je budou číst (popřípadě upravovat). K tomu aby zdrojové kódy byly přehledné a srozumitelné se využívají určité postupy a formátování.

16.1 Konvence psaní identifikátorů

Dnes nejpoužívanější způsoby zápisu víceslovných názvů elementů zdrojových kódů jsou:

• Oddělování slov podtržítky - Jednotlivá slova v názvu jsou místo mezery oddělena pomocí podtržítka: `nazev_promenne`.

• Oddělování velkým písmenem - Každé slovo v názvu začíná velkým písmenem a tak umožňuje optické rozdělení jednotlivých slov: `nazevPromenne`.

Obě formy zápisu jsou dobře čitelné a konvence pro používání oddělování jednotlivých slov názvů programových elementů je volitelná. V případě psaní víceslovných názvů maker je možné použít pouze metodu oddělování slov podtržítkem, protože konvence říká, že makra bez parametrů se píší velkými písmeny:

```
TOTO_JE_MAKRO
```

16.2 Přehlednost zdrojových kódů

Pro snadnější a přehlednější tvorbu zdrojového kódu a k nim vytvořených komentářů je nutné striktně dodržovat pravidlo, že na každém řádku se smí vyskytovat pouze jeden jediný příkaz. I když programovací jazyky umožňují psát zdrojový kód jako jeden dlouhý řádek, výsledný zdrojový kód není přehledný. Cílem při psaní zdrojových kódů je umožnit pozdější úpravy ve zdrojovém kódu, které rozšíří, optimalizují nebo jinak upraví funkci daného programu. Z tohoto důvodu je nutné vytvářet přehledný zdrojový kód. Konvence psaní pouze jednoho příkazu na řádek umožňuje také jednodušší komentování výsledných kódů a tím i snazší čitelnost při jejich pozdějším čtení a úpravě.

Dalším pravidlem při psaní zdrojových kódů je udržovat délku jednoho řádku s ohledem na šířku obrazovky. Pokud je řádka příliš dlouhá, je třeba ji na logickém místě

rozdělit (zalomit).

16.3 Odsazování

Odsazování jednotlivých elementů zdrojového kódu je velice výrazný grafický prvek, který umožňuje vytvářet čitelnější (čistší) zápis (podobně jako mezery mezi slovy v běžném textu) a proto by měl být použit v každém zdrojovém kódu. Je důležité, že bez odsazení částí zdrojového kódu daný zápis algoritmu bude fungovat (kompilátor jej přečte a zpracuje), ale čitelnost rozsáhlejšího kódu bude výrazně nižší. Pro odsazování se používají buď znaky mezery, nebo tabulátoru.

Nejčastěji se příkazy odsazují v bloku kódu. Blok kódu se může vyskytovat u mnoha programových elementů, například tělo funkce, větvící příkazy, ... Důležité je, že by se příkazy měly vyskytovat až za úrovní znaků (příkazů), které blok kódu definují:

```
funkce
{
    /* odsazení */
    příkazy
    {
```

```
        další příkazy
    }
}
```

Toto odsazení se nejčastěji dělá pomocí tabulátoru, ale je možné jej definovat pomocí několika mezer vysázených vedle sebe. V případě, že se bude vyskytovat blok kódu v bloku kódu, pak dojde k odsazení o další úroveň. Takovéto vnoření bloků kódu může mít libovolné množství úrovní a v každé úrovni dojde k dalšímu odsazení.

Další případ odsazování je v případě několika příkazů zapsaných na jednom řádku. Například v hlavičce funkce nebo jiných programových strukturách. Zde se používají výhradně mezery, protože tabulátor vytváří příliš velkou mezeru, která může mít opačný efekt:

```
funkce (parametr1, parametr2);
```

Odsazování slov na jednom řádku je v podstatě stejné jako sázení běžného textu (za každým slovem, znakem, následuje oddělovací mezera).

16.4 Správné komentování

Komentář se vkládá před nebo za část zdrojového kódu (např. příkaz nebo název podprogramu), jehož činnost

nebo význam je potřeba vysvětlit. Používání komentáře není povinné, umožňuje však napomoci k pochopení činnosti programu jako celku. Komentář nemá popisovat, co program dělá (protože to vyplývá ze zápisu programu), ale proč to dělá.

Komentují se významná místa ve zdrojovém kódu, definice/deklarace proměnných a jejich význam, deklarace metod/funkcí a změny v kódu

Je dobré na začátku programu/hlavičkového souboru uvést v komentářích název a funkci programu/hlavičkového souboru, autora, verzi a popřípadě další užitečné informace.

V případě, že je třeba dočasně zakrývat část zdrojových kódů, například při jeho ladění a testování, je možné buď využít podmíněného překladu a nebo kombinace řádkových a víceřádkových komentářů:

```
/*

nějaký kód

//*/
```

Sekvence `//*/` funguje stejně jako obyčejný ukončovací značka víceřádkového komentáře - jedná se o komentář v komentáry. Ale v případě, že je k uvozovací značce víceřádkového komentáře připsáno ještě jedno lomítko: `//*` dojde

k jeho zakritií a řádkový komentář zároveň zakryje také ukončovací příkaz. Tak lze rychle přepínat mezi viditelností a neviditelnéstí daného úseku kódu.

16.5 Fragmentace zdrojového kódu

Aby bylo jednodušší zapsat složitý a rozsáhlý algoritmu je vhodné ho fragmentovat, rozdělit na jednodušší části, které je možné jednodušeji zapsat a zvlášť odladit. Pokud budou správně fungovat jednotlivé části algoritmu, bude pravděpodobně správně fungovat i jako celek.

16.6 V jednoduchosti je síla

Programátor by se neměl bát složitých věcí, ale měl by k nim přistupovat jednoduše. To znamená, že návrh daného programu by neměl obsahovat nic co není nezbytně nutné. Toto pravidlo se nzívá KIS - Keep It Semple (udržuj to jednoduché). Dodatečné vlastnosti je možné doplnit po naprogramování a odladění základní struktury programu. Dále by se programátor měl snažit, aby výsledný zdrojový kód byl co nejkratší, ale nikdy na úkor jeho přehlednosti. Zkracování zdrojového kódu by mělo probíhat tak, že se navrhne co možná nejúspornější algoritmus, který zároveň poskytuje optimální výkon a funkčnost, odkazování na stejnou část programu z více míst, ...

Dobrým nástrojem, který umožňuje redukovat

množství zdrojového kódu je zobecňování. To znamená, že se navrhne určitá část algoritmu, která je univerzálně použitelná na více datových struktur, čímž odpadá potřeba vytvářet další části zdrojového kódu.

Dobré je před zahájením programování odhadnout z daného návrhu kolik logických řádků by mohlo stačit na naprogramování daného software a snažit se tento počet nepřesáhnout.

16.7 Žádný opakující se kód

Ve zdrojovém kódu by se neměly objevovat dvě stejné části kódu. V daném případě je možné danou část vložit do podprogramu (makra s parametry) a případně pouze vkládat parametry, které ovlivňují vykonávání dané části a pouze ji s určitými parametry volat na daných místech. To je vhodné z několika důvodů. Šetří se tak místo - zdrojový kód zabírá méně logických řádků. Programování bude rychlejší když je třeba se starat pouze o malou část zdrojového kódu.

16.8 Kompromisy

Vytvořit dokonalý software bez chyb je velmi těžký úkol (neexistuje bezchybný software, pouze software, který je dostatečně funkční). K tomu přispívá neustále rostoucí nároky a rozsah softwarových projektů a zkracující se

termíny realizací. Často je nutné dělat určité kompromisy mezi kvalitou a kvantitou softwaru. Tím je myšleno zda je vhodnější vytvořit neodladěný a nestabilní software, který v sobě zahrnuje všechny plánovanou funkcionalitu nebo stabilní odladěný software, který prozatím obsahuje pouze základní funkčnost, kterou bude do budoucna nutné dodělat. Z pohledu dobré pověsti je vhodnější vytvořit odladěný a funkční program, který obsahuje pouze základní požadovanou funkčnost, než nestabilní program obsahující vše co bylo požadováno.

K tomu je třeba hned v počátcích plánování softwarového projektu určit která část softwarového projektu má vyšší prioritu než jiná a podle toho se na ně v průběhu realizace zaměřit.

16.9 Defragmentace projektu

Projekt je dobré rozdělit (defragmentovat) na více menčích částí, které je jednodušší splnit. Pro přehled je dobrévypsat seznam dílčích bodů přiřadit jim určité priority a postupně odškrtávat splněné. Díky tomu je dobře patrné co vše je třeba ještě udělat a tím lépe odhadnout čas, který je třeba na dokončení projektu. Tento seznam může být spolu s termíny splněný součástí výstupní dokumentace, části zprávy realizace.

16.10 Žádné kopírování zdrojového kódu

Stejné části zdrojového kódu se nikdy nesmí kopírovat na různá místa. Částečně to souvisí s tím, že se nesmí žádná část zdrojového kódu opakovat To je důležité z několika důvodů. V případě, že se v dané části bude nacházet chyba, bude nakopírována na všechna ostatní místa ve zdrojovém kódu. Z toho vyplývá, že je potřeba tyto chyba opravovat všude kam byl daný kód nakopírován. V případě, že se daný kód nachází pouze v jednom podprogramu, stačí když se chyba opraví pouze v daném podprogramu není třeba ji opravovat již nikde jinde.

16.11 Chybové hlášky

V případě že určitý podprogram vrací informace o úspěšném dokončení své činnosti, nemusí v některých případech statči pouhá logická hodnota *true* nebo *false*. Pokud je uvnitř podprogramu více částí, které se mohou při svém zpravocání pokazit, je dobré jako návratovou hodnotu vracet číselnou hodnotu, kde hodnota 0 znamená, že podprogram byl proveden bez chyb a hodnota větší než nula slouží jako identifikátor části ve které nastal nějaký problém. Tyto chybové kódy je dobé zapsat do komentáře na začátek podprogramu. Díky tomu je možné zjistit ne jen že v daném podprogramu nastala chyba, ale také je možné identifikovat k jaké chybě při zpracování došlo.

16.12 Nešetřit bílými znaky

Bílé znaky jsou v programovacích jazycích využívány pro zvýšení přehlednosti zápisu. Je možné jich do programového kódu zapsat vedle sebe (na legitimní místa) libovolné množství, protože preprocesor kompilátoru je při kompilaci ze zdrojového kódů odstraní. Bílé znaky jako jsou mezery a odsazení tabulátoru jsou používány například v zápisech příkazů, výrazech, pro odsazení řádků v blocích kódu a jiné. Ve výsledku toto strukturování výrazně zvýší čitelnost daného zápisu.

16.13 Plánovat a myslet dopředu

Před zahájením programování je třeba promyslet jak bude daný programu fungovat jako celek. Není nutně třeba naplánovat vše do posledního detailu. Je třeba pouze vědět jak bude daný program fungovat, rozdělit si ho na jednodušší logické celky a jaké podprogramy je třeba v každé části definovat, aby vše bylo přehledné a funkční (aby nebylo v průběhu programování potřeba část programu předělat). Není třeba se dopředu zabývat implementací daného podprogramu, protože v praxi není možné vždy dopředu předvídat zda dané řešení bude na 100% fungovat. Funkční implementace a ladění je pak záležitostí samotné realizace programu.

16.14 Předvídat co by se mohlo stát

Při plánování nějakého algoritmu, který pracuje s daty je třeba myslet dopředu, přehrávat si v hlavě co se s těmito daty bude dít a najít rizika, která by mohla narušit celý algoritmus. Každý algoritmus pracuje bez chybně pokud není narušen nějakou nesprávnou činností. To ale uživatel nebo jiná část programu, který tento algoritmus používá nemusí vědět a je třeba zabránit, tomu aby tyto zakázané operace, nebo hodnoty narušili činnost daného algoritmu. Typickým případem je dělení. V případě, že se dělí kladnými nebo zápornými hodnotami je vše v pořádku. V případě ale že se dělí nulou celý algoritmus dělení se zhroutí. Aby nemohlo zpravidla dojít k zadání zakázaných hodnot, nebo nebyla vykonána zakázaná operace s daty je třeba daný algoritmus proti tomu **ošetřit**. Ošetřením je myšleno zabránění, aby došlo k vykonání zakázané operace, nebo zadání zakázané hodnoty do proměnné.

16.15 Znát cíle projektu

Při samotném programování je nutné do detailu znát cíle softwarového projektu, aby nebylo nutné v průběhu realizace přeplánovat celou architekturu, popřípadě aby nevznikaly logické chyby (díry) ve výsledné architektuře. V ideálním případě by programátor měl znát na spaměť každou část svého projektu (nebo alespoň těch částí, které

má na starosti). Jedině tak je minimalizováno riziko tvorby logických chyb při realizaci (přehled v projektu).

16.16 Pozor na úniky paměti

Při programování je časté, že při dynamické alokaci paměti programátoři zapomenou alokovanou paměť navrátit poté co již není potřeba. To má za následek únik paměti a snížení výkonu počítače. To je při tvorbě kvalitního software nežádoucí. Proto je třeba se řídit určitými zásadami, díky kterým je možné se úniku paměti vyhnout. Obecně je třeba ihned po zapsání příkazu pro alokaci paměti napsat současně také příkaz pro uvolnění dané paměti. Díky tomu není možné v jiné části programu zapomenout na její uvolnění. Protože poté co je paměť uvolněna ukazatel stále obsahuje danou paměťovou adresu proto je velice vhodné po uvolnění paměti nastavit hodnotu ukazatele na nulu. Tím je možné se vyhnout přístupu do cizí části paměti a násilnému ukončení celého programu.

Nelze obecně říci kde by měly být umístěny příkazy na uvolnění paměti, ale existují vhodná místa kde je vhodné je umístit. V případě globálních dynamicky alokovaných proměnných, které se alokují pouze jednou je příkaz pro uvolnění umístěn na konec celého programu. To aby se příkaz pro uvolnění paměti provedl až ve chvíli kdy je jisté, že již nebude potřeba. V případě globálních proměnných

uvnitř třídy jsou příkazy pro alokaci paměti umístěny do konstruktoru třídy a příkazy pro uvolnění paměti do destruktoru třídy. V případě, že se jedná o dynamicky alokovanou lokální proměnnou je jejich alokace umístěna na začátek daného bloku (nebo do míst kde je jejich alokace zapotřebí) a jejich uvolnění na konec bloku.

16.17 Názvosloví

V případě pojmenování jakékoli části zdrojových kódů ať už proměnné, funkce nebo makra, platí že jeho jméno by mělo vystihovat účel daného elementu.

V případě pojmenování jakýchkoli programových elementů, které něco uchovávají - proměnné, třídy, ... se používá pro jejich pojmenování **podstatná jména**. Pokud je třída abstraktní, je doporučeno tento fakt zdůraznit v jejím názvu pomocí předpony *„Abstrakt“*. Příklad pojmenování proměnné: `počet`, `hodnota`, ... Funkce, popřípadě metody (obecně podprogramy nebo pojmenované části kódu) naopak mají za úkol něco vykonávat a proto se pro jejich pojmenování používá **sloveso**. Příklad názvu funkce: `NastavHodnotu`, `Načti`, ...

16.18 Konzistence identifikátorů

Pro jména identifikátorů by měl být použit pouze jeden jazyk (čeština, angličtina, ...). Z funkčního hlediska se

nejedná o chybu, ale v případě prezentace to nedělá dobrý dojem. Jména identifikátorů by také neměla být zbytečně dlouhá, kvůli zapamatování v programu a kvůli výsledným délkám programových řádků při výskytu několika identifikátorů v jednom příkazu.