Interní dokument FIT VUT v Brně

Nedělejte zbytečné chyby
při programování v C

příručka pro studenty
základních kurzů programování

Poslední revize 21. října 2011

Autor: Ing. David Martinek
Ústav Inteligentních Systémů
Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Lektoři: doc. Ing. Zdeňka Rábová, CSc., prof. Ing. Jan Maxmilián Honzík, CSc., Dr. Ing. Petr Peringer

Dokument byl vysázen programem LaTeX.

Obsah

Kapitola 1

Úvod

Tato příručka obsahuje přehled a rozbor nejčastějších chyb, které se vyskytují při řešení studentských projektů v jazyce C. Každý problém obsahuje hodnocení jeho závažnosti a návrh správného řešení.

Příručka je určena hlavně začínajícím programátorům, ale i mírně nebo středně pokročilé programátory může upozornit na některé špatné programátorské návyky, které působí potíže zejména při práci v týmu a při pozdějších úpravách programů. Příručka však není učebnicí, takže pokud máte v plánu učit se podle ní programovat, můžete být zklamaní.

Pro úspěšné programování je více než co jiného nezbytná praktická zkušenost. Věřím však, že tato příručka může proces získávání programátorských zkušeností výrazně urychlit.

1.1 Jak číst tuto příručku

Každá sekce se skládá ze dvou částí – obecného úvodu, který se snaží danou oblast stručně vysvětlit, a ze seznamu nejčastějších chyb, které jsem vypozoroval při opravování projektů i při vlastní programátorské práci.

Nedá se říci, že byste v této příručce měli číst od začátku do konce. Mnohem užitečnější bude, když si ji prolistujete těsně před tím, než začnete řešit projekt a projdete si oddíly, které se k řešenému projektu blíže vztahují (například práce se soubory).

Na začátku následující kapitoly jsou soustředěny obecnější programátorské problémy týkající se všech programů, které budete ve své programátorské praxi řešit. Bude nejlepší, když začnete zde. Poté následuje soupis konkrétnějších problémů.

Na konci příručky najdete seznam užitečné literatury i množství odkazů na internetové stránky.

1.2 Typografické konvence

V textu se vyskytují ukázky kódu. Černým textem na šedém pozadí je kód, ze kterého si můžete brát příklad:

```
TData *prvek = malloc(sizeof(TData));
if (prvek == NULL) chyba(CHYBA_PAMETI);
prvek->data = 24;
...
free(prvek);
```

Červeným písmem na světlejším šedém pozadí jsou psány odstrašující ukázky, tzv. antiidiomy. Podobných konstrukcí se rozhodně vyvarujte!

```
int *prvek;
*prvek = 10;
free(prvek);
```

Klíčová slova jako **`if`**, **`while`**, **`typedef`** jsou v textu jasně odlišena od názvů podprogramů jako `printf`, `sin`, `isCorrect`.

U každého problému najdete hodnocení jeho závažnosti. Každý problém je ohodnocen jednou až pěti hvězdičkami. Nejméně závažné problémy mají jednu hvězdičku a ty nejzávažnější jich mohou získat až pět. Mezi počtem hvězdiček a počtem záporných bodů v projektech je přímá úměra.

U každého hodnocení se nachází i krátké upozornění, jaké následky může tato chyba mít:

Hodnocení:	***** Velmi zákeřná chyba! Během ladění a testování se vůbec nemusí projevit.

V textu najdete i tipy a rady, jak se vyvarovat často opakovaných chyb. Tipy v této příručce vyjadřují mé osobní zvyklosti. Můžete se jimi inspirovat, ale není třeba se jimi doslovně řídit. Rady vycházejí z mých osobních zkušenosti i z obecně vžitých řešení problémů. Měli byste se jimi řídit.

Tip: Naučte se psát všemi deseti. Vlastní zápis programů tím velice urychlíte.

Rada: Řiďte se radami v této příručce. Ušetříte si spoustu času a starostí.

Kapitola 2

Stylistické chyby

Pokud se chcete programováním živit, je velmi pravděpodobné, že budete pracovat v týmu. I když prozatím pravděpodobně píšete programy sami, je dobré si hned zpočátku osvojit alespoň základní zásady týmové spolupráce.

Jedním ze základních požadavků při týmové spolupráci je kompatibilita. Pokud se programátoři v týmu nejsou schopni dohodnout, jak bude vypadat jejich kód, potom spolu nebudou schopni nic vytvořit. Když programátor napíše určitý kód, měl by počítat s tím, že ho časem bude používat někdo jiný.

Občas se programátorům zdá, že požadavky na přehledné odsazování bloků programu a důsledné komentování jednotlivých částí jsou zbytečným obtěžováním. Není tomu tak. Až budete nuceni pracovat s cizím kódem, určitě oceníte, když bude napsán srozumitelně.

2.1 Málo výstižné identifikátory

Hodnocení:	*** Chyba způsobuje problémy při údržbě a při týmové spolupráci.

Používání názvů funkcí jako `f1`, `f2`, konstant `S1, ..., S8` nebo proměnných jako `a`, `b`, `c`, `d`, je kontraproduktivní. Názvy proměnných, konstant a podprogramů musí být dostatečně výstižné, aby nebylo potřeba příliš často pátrat v dokumentaci, k čemu vlastně slouží ta funkce `f7`.

Je dobré používat víceslovní názvy, ale nesmí se to přehánět. Jednopísmenové identifikátory je možné používat pouze pro indexy (`i`, `j`, `k`) nebo tam, kde je to vžité – zejména v matematických algoritmech (`e`, `a`, `b`, `c`), případně pro parametry velmi krátkých funkcí. Čísla se v identifikátorech používají jen výjimečně, pokud to má smysl. Pro zápis identifikátorů jsou vžité tyto konvence:

- Víceslovní názvy se zapisují takto:
  `vypisNapovedu`, `nasobeniMatic`, `otestujAOtevri`, ...
  Druhým způsobem je použití podtržítek:
  `Vypis_Napovedu`, `Nasobeni_Matic`, `Otestuj_A_Otevri`, ...
  Osobně si myslím, že první způsob je lepší, protože znak podtržítka je na klávesnici na pozici, která se špatně používá (navíc se tato poloha liší při českém a anglickém rozložení kláves).

- Konstanty se obvykle píší velkými písmeny, aby je bylo možné na první pohled odlišit od proměnných a názvů funkcí. Slova lze oddělovat podtržítkem nebo je nechat bez podtržítka: `MAX_RADKU, PI, LIDIVSYSTEMU`

- Názvy statických typů začínají prefixem T (jako type): `TSeznamLidi`, `TFronta`. Další možností je používat příponu: `wchar_t`, `size_t`. Tento způsob se používá v systémových knihovnách. Použitím prvního způsobu je možné odlišit vlastní datové typy od systémových.

- Názvy dynamických typů (ukazatele) je v jazyce C zvykem zapisovat přímo takto: `TSeznam *`, `TFronta *`.

Rada: V programu není dobré kombinovat české a anglické identifikátory[1]. Vyberte si jeden jazyk a důsledně jej používejte. Pokud se někdy budete programováním živit, pravděpodobně vás zaměstnavatel nebo jiné okolnosti donutí psát vše anglicky, takže je dobré si na to od počátku zvyknout.

2.2 Nepřehledné nebo neúsporné odsazování

Hodnocení:	**
	Chyba způsobuje problémy při údržbě a při týmové spolupráci.

Jazyk C je strukturovaný jazyk. To znamená, že umožňuje organizovat jak data (pole, struktury), tak kód do složitějších struktur (bloky, funkce). Bloky kódu tvoří logické celky a měly by jít v programu lehce odlišit – nejlépe pomocí odsazování. Pokud tyto bloky nejsou na první pohled v kódu patrné, ztrácí se jedna z hlavních výhod vyšších programovacích jazyků – přehlednost.

Odsazování by mělo být v celém zdrojovém souboru jednotné. Pokud jsou každý cyklus či funkce odsazovány jinak, je velmi únavné se tímto kódem probírat[2].

Pokud s programováním začínáte, je dobré si hned ze začátku vytvořit určitý styl odsazování. Během let se ustálilo několik základních paradigmat, které je vhodné dodržovat.

- Odsazovat o 2 až 3 znaky[3]. Méně než 2 znaky je málo a více než 3 zase příliš plýtvá místem.

- Není dobré používat tabulátor[4], protože u kolegy pak může formátování vypadat úplně jinak.

- Blok, který tvoří tělo příkazu **`if`**, **`for`**, atd., by měl být umístěn tak, aby bylo na první pohled vidět, ke kterému příkazu patří.

- Oddělovat kusy kódu, které spolu těsně nesouvisejí, prázdnými řádky. Takto se oddělují zejména funkce, ale také cykly, inicializace proměnných, a podobně.

Tato pravidla neplatí jenom pro jazyk C, ale i pro většinu dnes používaných programovacích jazyků. Následuje ukázka jednoho z používaných stylů odsazování. Všimněte si, že podle odsazení je na první pohled patrné zanoření jednotlivých bloků.

[1] V této příručce budu v zájmu lepší srozumitelnosti používat české názvy.
[2] Často to také může být známkou opisování.
[3] Někdy se uvádí až 8 znaků, ale mně se to zdá příliš mnoho.
[4] Výjimku tvoří soubor Makefile, kde má tabulátor svůj význam, tam jej nejde nahradit mezerami.

```
/**
 * Tato funkce slouží jako ukázka správného odsazení.
 * Jde o naprosto umělou funkci, která nedělá nic
 * rozumného!
 * @param param1 Dolní mez cyklu.
 * @param param2 Horní mez cyklu.
 */
void ukazOdsazeni(int param1, int param2)
{
  int index = 10;
  printf("Cyklus while\n");
  while (index > 0)
  { // vhodné místo pro komentář
    printf("%d ", index);
    index--;
    if (index == 5)
    {
      printf("\n----------\n");
    }
  }
  printf("\nCyklus for\n");
  for (int i = param1; i <= param2; i++)
  { // cyklí od param1 do param2
    printf("%d ", i);
    if (i == ((param1 + param2)/2)
    {
      printf("\n----------\n");
    }
  }
}  //ukazOdsazeni
```

Tip: Nevymýšlejte svůj speciální styl odsazování. Mezi programátory v jazyce C se během let od jeho vzniku ustálily asi tři styly odsazování. Jestliže si zvyknete na svůj vlastní styl, bude pro vás nepohodlné pracovat s cizím kódem, který bude téměř jistě odsazován jinak. Dále tím velmi ztížíte práci svým kolegům, kteří budou muset s vaším kódem dále pracovat.

Na internetu najdete plno stránek, které se věnují stylům odsazování v jazyce C, viz [cst04]. Zvolte si jeden z používaných stylů a držte se ho.

Tip: Protože jsou programátoři jenom lidé, a co člověk to jiný názor, vznikly už v dobách počítačového pravěku programy, které umí zdrojový text zformátovat podle zvoleného stylu. Existují snad pro každý počítačový jazyk (Java, C++, Pascal/Delphi, ...). Tyto programy se liší možnostmi nastavení a komfortem použití – některé se spouštějí z příkazového řádku, jiné je zase možné integrovat do nejrůznějších vývojových prostředí (ta modernější prostředí už je mají integrována od výrobce). Na internetu jich lze najít nepřeberné množství – stačí v libovolném vyhledávači zadat klíčová slova *indent* nebo *refactoring* a samozřejmě název jazyka.

Tip: V Linuxu existuje program *indent* [ind04], který je velmi konfigurovatelný. Vřele doporučuji jej používat. Tento program je možné získat i ve verzi pro Windows.

2.3 Chybí komentáře na správných místech

Hodnocení:	** Chyba způsobuje problémy při údržbě a při týmové spolupráci.

Komentář slouží k rychlejší orientaci a snazšímu pochopení zdrojového textu. Je zbytečné komentovat každý řádek kódu – to se hodí jenom pro popis velmi složitých a z hlediska řešení klíčových algoritmů. Zato je důležité komentovat každý podprogram. Před každou hlavičkou funkce by měl být komentář, který stručně vysvětlí, co podprogram dělá. Dále je vhodné popsat parametry, které podprogram používá.

Komentáře jsou důležité pro spolupracovníky, kteří budou s kódem dále pracovat. Stejně důležité jsou však i pro vás, jako autora. Pokud se po delší době rozhodnete, že svůj program rozšíříte, bude to mnohem jednodušší, pokud jste věnovali čas psaní komentářů. Než se pokoušet zprovoznit nekomentovaný program, bývá často rychlejší napsat ho celý znovu.

Obzvláště důležité je psát komentáře v programových modulech či knihovnách. Je velmi dobrým zvykem komentovat vše v hlavičkovém souboru a potom tyto komentáře překopírovat i do samotného zdrojového kódu.

V dobře napsaném programu by měly být komentovány:

zdrojové soubory – každý soubor by měl obsahovat strukturovanou hlavičku s podpisem autora, přiřazením souboru ke konkrétnímu projektu, popisem problému, který se v tomto souboru řeší, a datem vytvoření (většinou stačí rok); u složitějších projektů je pak dobré uvádět sem i číslo verze;

hlavičkové soubory – platí pro ně totéž, co pro zdrojové soubory, ale silněji; hlavičkový soubor musí být komentován zvlášť pečlivě, protože je určen zejména pro programátory, kteří chtějí vaše funkce používat; zdrojové soubory se často nezveřejňují, hlavičkové soubory téměř vždy;

datové typy – zde by mělo být popsáno, jaká data jsou tímto typem popisována, a k čemu je to dobré;

podprogramy – popis vlastní činnosti by měl vypadat podobně jako v nápovědě k systémovým funkcím (viz nápověda ke standardní knihovně jazyka C, Kylixu, ...); dále by zde měl být popsán každý parametr, který se podprogramu předává nebo jehož pomocí se vracejí hodnoty zpět;

velmi složité algoritmy – důležité je pochopení jejich funkce; v některých knihovnách není výjimkou, když komentář zabírá několikanásobně více prostoru než samotný algoritmus.

Zde je ukázka vzorově komentovaného podprogramu:

```
/**
 * Sečte dvě matice (mPrva, mDruha) a výsledek uloží do
 * parametru mVysledek. Pokud má první matice rozměr P krát
 * Q, pak druhá musí mít také rozměr P král Q a výsledkem bude
 * matice stejných rozměrů. Všechny matice se předávají odkazem
 * z důvodu efektivity, modifikován bude pouze mVysledek.
 * @param mPrva První matice o rozměrech P x Q
 * @param mDruha Druhá matice o rozměrech P x Q
 * @param mVysledek Výsledná matice (P x Q)
 */
void nasobeniMatic(const TMatice *mPrva, const TMatice *mDruha,
         TMatice *mVysledek);
```

Tip: V této ukázce je současně demonstrováno další využití komentářů. Zvláštní způsob psaní hvězdiček a klíčová slova `@param` slouží pro program *doxygen* [dox04], který umí z komentářů vytvořit hypertextový dokument. Tímto způsobem můžete velmi pohodlně vytvořit programátorskou dokumentaci ke svému kódu. Uvedená syntaxe komentářů je převzata z jazyka Java, kde podobnou funkci jako *doxygen* zastává program *javadoc*. Pokud plánujete, že se někdy naučíte jazyk Java, ušetříte si práci, když tento styl komentování kódu budete používat už nyní.

2.4 Řádky delší než 80 znaků

Hodnocení:	** Nepřehledné. Může zakrývat chyby.

Požadavek na maximálně 80 znaků na řádek se může zdát v dnešní době zbytečný. Pochází z dob, kdy se vše dělalo převážně v textovém režimu, který více znaků na řádek nenabízel[5].

Ovšem i v okenním systému málokdy používáte okna roztažená přes celou obrazovku. Rolování textu do stran nejenom velmi zdržuje, ale také snižuje čitelnost programu. Pokud text pokračuje za hranici okna, špatně se v něm hledají chyby.

Rada: Pište kód, který je v souboru orientován spíše vertikálně než horizontálně. Raději ať má více řádků, než aby byl špatně čitelný. Nespoléhejte na to, že všichni vaši spolupracovníci budou mít stejně velký monitor jako vy.

2.5 Horizontálně orientovaný kód

Hodnocení:	* Způsobuje potíže při ladění.

Jazyk C umožňuje velmi kompaktní zápis. Někdy to však svádí ke snaze mít program vtěsnaný na co nejmenší počet řádků. To je naprosto nesmyslné, protože se snižuje čitelnost a vznikají problémy při ladění, neboť se ztrácí možnost jemnějšího krokování.

[5] V dnešní době to není pravda, protože existuje množství terminálů, které nabízejí téměř libovolné rozměry textového okna.

Následující způsob zápisu kódu je nevhodný, protože při ladění neumožňuje vložit zarážku (breakpoint) na každé volání funkce zvlášť:

```
if (jeVPoradku(vstup)) { vypis(vstup); } else { osetri(vstup); }
```

Šetřit místem ve zdrojových souborech v dnešní době nemá velký smysl. Dobře zformátovaný kód je nejen přehlednější, ale zároveň usnadňuje hledání chyb. Ideální je, pokud je každý příkaz a každý výraz na samostatném řádku:

```
if (jeVPoradku(vstup))
{ // vše o.k.
  vypis(vstup);
}
else
{ // něco je špatně
  osetri(vstup);
}
```

Z příkladu je také vidět, že kód orientovaný vertikálně se mnohem lépe komentuje a je přehlednější. Není potřeba se příliš namáhat, abychom pochopili, co tento kód znamená.

Tip: Všimněte si polohy komentářů v jednotlivých větvích. Je velmi užitečné komentovat tímto způsobem složitější podmínky. Výrazně to urychluje jejich pochopení.

Kapitola 3

Obvyklé programátorské chyby

Každý člověk dělá chyby. Je dobré se s tímto faktem smířit a chovat se podle toho. Pokud s programováním začínáte a máte pocit, že chybujete zbytečně často, nedělejte si s tím zatím těžkou hlavu. Až získáte více zkušeností, vypěstujete si sami návyky, jak nejhorším chybám předcházet.

Jazyk C je jazyk, který používá minimum základních datových typů a v porovnání například s Pascalem provádí slabší typovou kontrolu. Nepříjemným důsledkem pro začínajícího programátora je skutečnost, že musí sám ošetřovat rozsahy hodnot svých proměnných. Dále je nepříjemné, že některé konstrukce jsou z pohledu jazyka legálním kódem, ačkoli jsou naprosto chybné. Překladač je klidně přeloží, ale výsledný program pak bude fungovat jinak, než si programátor původně představoval. Je velmi důležité sledovat všechna varování, která překladač vypisuje. Je nutné si uvědomit, že to není chyba jazyka, ale jeho vlastnost. Dává mu to velkou sílu zvláště při práci s hardwarem na nízké úrovni. Na druhou stranu to klade větší nároky na pozornost programátora[1].

Pro začátečníka v jazyce C je nejvýhodnější naučit se množinu bezpečných konstrukcí (idiomů) a tu potom používat. Zvýšíte tím pravděpodobnost, že váš program bude fungovat podle vašich představ. Další výhodou je, že takový kód je snáze pochopitelný i pro ostatní programátory. Je samozřejmě dobré znát co nejvíce schopností jazyka, ale používejte je až když je budete znát naprosto dokonale, a když si budete jisti, že je to užitečné.

Onu potřebnou množinu základních idiomů se dozvíte především na přednáškách nebo je najdete v učebnicích ([Her03, Kad02, Töp95]). Některé ukázky správných řešení najdete i v následujícím textu, ale nejsou uspořádány tak, aby se podle nich dalo učit. Následující kapitoly jsou zaměřeny spíše na protipříklady – antiidiomy, tj. příklady, které byste používat raději neměli.

3.1 Používání goto

Hodnocení:	**** Velmi závažná chyba! Snižuje čitelnost kódu. Často způsobuje nepředvídatelné chování.

Příkaz `goto` není ve vyšších programovacích jazycích v 99,9% potřeba. Protože zde mluvím o strukturovaném programování, vždy je k dispozici dostatek prostředků, jejichž prostřednictvím je možné se tomuto příkazu vyhnout. Předně to jsou cykly a podmíněný příkaz. Další náhradou za `goto` jsou podprogramy. Pokud je potřeba

[1] Říká se, že jazyk C je jako ostře nabitá zbraň. Pokud s ní umíte zacházet, je možné, že občas přinesete domů něco k večeři. Pokud ne, hrozí vám, že se střelíte do nohy.

přerušit cyklus uprostřed jeho těla, je možné to udělat příkazem **`if`** a příkazy **`break`** či **`continue`**.

Příkaz goto má své opodstatnění pouze v případě, kdy je potřeba vyskočit z příliš hluboké hierarchie zanořených cyklů[2]. Jindy ne! Používání tohoto příkazu nepřináší žádné zrychlení kódu a navíc je to extrémně nepřehledné.

Například v Javě příkaz **`goto`** vůbec není. Místo něj je zde modifikovaný příkaz **`break`**, který se umí vrátit na návěští definované před cyklem. Jde tedy přesně o tentýž případ jako v minulém odstavci.

Zákeřnost příkazu **`goto`** spočívá v tom, že v jazyce C je možné skočit téměř kamkoli[3]. Představte si případ, kdy se pomocí **`goto`** skáče dovnitř cyklu. V takovém případě bude mít program předem těžce definovatelné chování.

Používání příkazu **`goto`** často svědčí o nepochopení významu podmínek v příkazech cyklu. Následující příklad je chybný:

```
while (index > 0)
{
  if (index == 1000)
    goto konec;
  ...
}
konec:
```

Mnohem lepší je použít o málo složitější podmínku cyklu:

```
while (index > 0 && index < 1000)
{
  ...
}
```

Všimněte si, že význam celého cyklu je v tomto případě mnohem snáze pochopitelný. Podmínka ukončení cyklu není schovaná někde uprostřed jeho těla, jako v předešlém případě.

U nezanořených cyklů je možné použít pro ukončení cyklu příkaz **`break`**:

```
int c;
while ((c = fgetc(f)) != '\n')
{
  if (c == EOF)
  { // EOF místo konce řádku
    zpracujChybu(CHYBA_EOF);
    break;
  }
  ...
}
```

Tip: Používejte takto násilné přerušení cyklu jen pro ošetření výjimečných situací. V tomto případě cyklus načítá ze souboru řádek textu. Pokud soubor končí dříve než se v něm vyskytne znak `'\n'`, jde o výjimečnou událost.

[2]Příliš hlubokou hierarchií cyklů mám na mysli hloubku 4 a více. V jiných případech považuji použití **`goto`** za hrubou chybu.

[3]Nedá se skočit z jedné funkce do druhé.

3.2 Program není dostatečně obecný

Hodnocení:	*** – **** Při budoucích úpravách očekávejte problémy. Neobecná řešení jsou ve svých důsledcích pracnější než obecná řešení, i když to na první pohled tak nevypadá.

Program je málo obecný tehdy, když není strukturovaný. Bez podprogramů se složitější problémy řeší velice obtížně. Ovšem ani s podprogramy není zdaleka vyhráno. Začátečníci často používají podprogramy tímto způsobem:

```
uvod();
nactiData();
prevedData();
odectiData();
vypisVysledek();
```

Problém je v tom, že se zde hodnoty předávají přes globální proměnné. Nikdy to tak nedělejte! Takové podprogramy jsou prakticky nepoužitelné mimo aktuální projekt. V tomto případě je dokonce nejde zavolat na jiném místě programu. Podrobněji je tento problém probrán v kap. 4.2 na str. 23.

3.3 Program je zbytečně neefektivní

Hodnocení:	*** – **** Některé neefektivní algoritmy nejde zrychlit ani rychlejším procesorem.

Problémy s efektivitou se často vyskytují v případě, kdy má program zpracovávat soubory, ale také pole, textové řetězce, seznamy, atd. Soubory mohou být obecně velmi velké (desítky MB). Pokud jsou programy neefektivně navrženy (na krátkých souborech se to neprojeví), nezřídka při zpracování rozsáhlejších dat zhavarují. V nejhorších případech pak tato data i poškodí.

Proto je potřeba, aby se programátor vždy před vlastním návrhem algoritmu zamyslel. Je nutné analyzovat všechny možné vstupy, jimž může být program vystaven. Pak je teprve možné navrhnout efektivní algoritmus.

Neefektivní bývají zejména algoritmy, které mají za úkol procházet delší datové struktury jako pole, soubory, textové řetězce nebo lineární seznamy. Většinu těchto struktur lze zpracovat v jednom průchodu. Víceprůchodová řešení mohou dokonce o řád (i více) zhoršit asymptotickou časovou složitost algoritmu.

Prohlédněte si pozorně následující algoritmus. Jde o častou chybu při zpracovávání textových řetězců (vyskytuje se však u všech lineárních datových struktur).

```
for (int i = 0; i < strlen(text); i++)
{
  zpracujZnak(text[i]);
}
```

Tento algoritmus vypadá na první pohled bez problémů. V podstatě má však kvadratickou časovou složitost. Problém je v podmínce cyklu, která se vykonává v každém kroku. Funkce `strlen` má sama lineární časovou složitost, protože při počítání prvků musí projít všemi prvky řetězce.

Předpokládejme, že řetězec `text` má 100 prvků. Funkce `strlen` musí po zavolání vykonat nejméně 100 kroků. Cyklus v tomto případě musí v každém svém kroku jednou zavolat funkci `strlen` a ještě musí zpracovat jeden znak. Pokud budeme brát samotné zpracování znaku jako jeden krok, celý algoritmus vykoná 100 krát 100 + 1, tedy 10100 kroků. Deset tisíc kroků na sto prvků!

Zde je lepší verze:

```
int delka = strlen(text);
for (int i = 0; i < delka; i++)
{
  zpracujZnak(text[i]);
}
```

V tomto případě se během řešení celého algoritmu vykoná 200 kroků. Prvních 100 kroků vykoná jediné volání funkce `strlen` a dalších 100 kroků samotný algoritmus. Stále ovšem jde o dvouprůchodové řešení. Pokud by se tímto způsobem zpracovával dlouhý soubor, stále by to bylo citelně neefektivní.

Celé řešení jde napsat ještě efektivněji, když se celý řetězec projde pouze jednou:

```
char znak;
int i = 0;
while ((znak = text[i]) != '\0')
{
  zpracujZnak(znak);
  i++;
}
```

Tento příklad lze pokládat za idiom pro průchod všech typů lineárních struktur, u nichž předem není známa jejich délka (textový řetězec, soubor, lineární seznam). V podmínce se řeší získání další hodnoty a současně se zde testuje výskyt koncového prvku.

3.4 Příliš „zadrátované“ řešení

Hodnocení:	***–**** Problémy do budoucna. Čím složitější zásah musíte při opravě udělat, tím hrozí větší nebezpečí zavlečení nových chyb.

Zadrátovaným řešením[4] problému mám na mysli takový program, který se těžko modifikuje, a potom také takový program, který značně omezuje uživatele (neplést s jednoúčelovým programem typu `dir`!). Těžko se modifikují takové programy, které jsou složeny z málo obecných podprogramů, nebo ty, které vůbec podprogramy nepoužívají. Podrobněji budou tyto případy rozebrány v další kapitole.

Příkladem programu, který omezuje uživatele, je například takový program, který ukládá svůj výstup do souboru `C:\data\vystup.txt` a nedá uživateli možnost to změnit. Ještě horší je, že se program spoléhá na konkrétní nastavení okolního systému. Kdo zaručí, že program bude mít právo zápisu do tohoto souboru? Program se nemůže spoléhat na to, že na disku bude adresář `data`, dokonce ani na to, že existuje nějaký disk `C:` – většina operačních systémů ostatně s disky tímto způsobem vůbec nepracuje (viz Unix, MacOS, PalmOS, …).

Dále zvažte použitelnost funkce, která umí vypočítat součin matic, ale zvládne to pouze s maticemi rozměru 10×10. Taková funkce asi není dobrou vizitkou jejího autora.

[4] Tento výraz pochází z dob prvních počítačů, které se programovaly pomocí drátových propojek, a změna algoritmu byla u nich velmi pracná.

Tyto příklady jsou samozřejmě ilustrativní a lze je zobecnit i na jiné problémy. Uvedená chyba má často spojitost s používáním „magických“ čísel (viz kap. 3.9 na str. 16).

Programy by měly uživateli poskytovat dostatek volnosti k práci. Příliš natvrdo nastavené parametry značně devalvují hodnotu programu.

Rada: Jazyk C nabízí dostatečné prostředky pro zjednodušení a zobecnění programů – funkce, strukturované datové typy, moduly, aj. Využívejte je.

3.5 Neinicializované proměnné

Hodnocení:	***
	Způsobuje zákeřné chyby zvláště v případě ukazatelů.

V jazyce C není nikdy zaručeno, že lokální proměnná bude mít po definici bez inicializace nějakou konkrétní hodnotu. U lokálních proměnných podprogramů mohou být hodnoty neinicializovaných proměnných náhodné. Tyto proměnné totiž vznikají pouze vyhrazením prostoru na zásobníku. Zde je ukázka zákeřné chyby:

```
int sum(int max)
{
  int i, sum; // chyba! neinicializované proměnné
  while (i <= max)
  {
    sum += i;
  }
  return sum;
}
```

Zákeřnost této chyby spočívá v tom, že během ladění se velice často zdá, že taková proměnná má pokaždé nějakou stabilní hodnotu, často hodnotu nula. Programátor by mohl velice lehce nabýt dojmu, že je vše v pořádku. Při reálném používání ovšem mohou proměnné `i` a `sum` v okamžiku své definice nabýt libovolné hodnoty a důsledkem budou naprosto nesmyslné výsledky. Toto chování se vyskytuje hlavně v systémech, které nepoužívají multitasking[5] (DOS), ale může se vyskytnout i u multitaskových systémů, zvláště pokud běží s dostatkem paměti RAM a systém není příliš vytížený.

U ukazatelů mívá tato chyba ještě fatálnější následky – přístup do nealokované paměti. Přístup do takové paměti v chráněném režimu způsobí ve většině operačních systémů havárii programu. Náhodný ukazatel se však také může trefit někam do prostoru, ke kterému má program dostatečná přístupová práva a potom může dojít k poškození dat. Taková chyba se pak obtížně hledá, protože se projeví v naprosto neočekávaném místě. Více se dozvíte v kap. 5.3 na str. 33.

Rada: Pokud definujete proměnnou, snažte se ji inicializovat nějakou bezpečnou hodnotou. Zvláště se to týká ukazatelů. Vyhnete se tím obtížně odhalitelným chybám.

Novější programovací jazyky deklaraci neinicializované proměnné považují přímo za syntaktickou chybu (viz Java). Moderní překladače jazyka C v některých případech vypisují varování, ale neumí odhalit vše.

[5]Multitasking znamená přepínání procesů. Multitaskový operační systém umožňuje zdánlivě paralelní běh více programů tím, že jim cyklicky poskytuje procesorový čas.

3.6 Použití nesprávného cyklu

Hodnocení:	** – *** Občas může způsobit nechtěné chování programu.

Zvláště začínající programátor často nevidí žádný rozdíl mezi cyklem typu **`while`** a **`do-while`**. Někteří programátoři se zase naučí používat jediný typ cyklu a používají ho za všech okolností. Oba typy cyklů lze samozřejmě vzájemně převést, ale pouze za cenu zmenšení přehlednosti kódu.

Je důležité si uvědomit, že cyklus **`while`** **nemusí** proběhnout ani jednou. Naproti tomu cyklus **`do-while`** **musí** proběhnout alespoň jednou. Pokud si tento rozdíl neuvědomíte, může to vést k chybám v programu.

Často také dělá problémy uvědomit si, jak má být zapsaná podmínka cyklu a kdy se vyhodnotí. V jazyce C pro oba výše uvedené cykly platí, že tělo cyklu se opakovaně provádí, pokud je výsledek podmínky **`true`** (nenulová hodnota). Jakmile je výsledkem podmínky **`false`** (hodnota 0), pokračuje se kódem za tělem cyklu. U cyklu **`while`** se podmínka vyhodnotí před započetím cyklu, zatímco u cyklu **`do-while`** se vyhodnocení provede po provedení těla cyklu.

Pozor na zápisy tohoto typu:

```
do
{
  if (index >= 10) break;
  index++;
} while (index < 10);
```

Příkaz **`if`** v tomto případě slouží jako podmínka cyklu. Toto je typický příklad chybné volby cyklu. V tomto případě je správným řešením použití cyklu **`while`**:

```
while (index < 10)
{
  index++;
}
```

Třetím typem je cyklus **`for`**. V jazyce C jde o velmi univerzální typ cyklu. Více než jinde zde ovšem platí, že přílišné experimentování vede spíše k chybám a k méně pochopitelnému kódu.

Tip: Raději cyklus **`for`** používejte jen k tomu, k čemu je primárně určen – k cyklení s předem známým počtem kroků. Například pro průchod polem používejte tento idiom:

```
for (int i = 0; i < delkaPole; i++)
{
  pole[i] = ...;
}
```

Rada: Podle nové normy (ISO C99) je možné definovat proměnnou cyklu (`i`) v jeho těle[6]. Tato lokální proměnná za tělem cyklu zaniká, takže je možné znovu definovat proměnnou stejného jména. Pokud máte v programu více cyklů, použitím této vlastnosti se vyhnete vymýšlení stále nových jmen proměnných a také zabráníte nechtěnému použití staré hodnoty této proměnné. Tuto vlastnost rozhodně využívejte. Jde o rys jazyka převzatý z C++, takže pokud si ho zvyknete používat ušetříte si práci, až se budete učit C++.

[6] Obecně lze definovat proměnnou, která je lokální v rámci svého bloku.

3.7 Upřednostňování podružných problémů

Hodnocení:	** – ***
	Tento způsob práce je velmi neefektivní a znesnadňuje pozdější úpravy programu.

Někteří programátoři začínají řešit úlohy od méně významných problémů směrem k těm závažnějším. Proto se nejdříve zabývají programováním různých oken, menu, barevných nápisů, a podobně. Na vlastní řešení problému už jim pak nezbývá čas.

Dalším nešvarem pak je, že vlastní řešení problému „utopí" v záplavě zbytečného kódu. Takové programy se pak velmi těžko spravují. Pokud je výkonný algoritmus obalen množstvím jiného kódu (vykreslování *teploměru*, různá počítadla, apod.), velmi to ztěžuje ladění a jenom těžko ho lze použít v jiném projektu.

Rada: Pokud máte potřebu vaše programy nějakým způsobem vyzdobit, pak je rozdělte na více modulů. Tím oddělíte podstatný kód od kódu implementujícího uživatelské rozhraní (vstup/výstup, ...). Navíc potom můžete skutečně užitečný kód použít i v jiných projektech. Dbejte na to, aby funkce prováděly pouze to nejnutnější (viz kap. 4.7 na str. 27).

Tip: Uvědomte si, že uživatel neocení pěkně vypadající program, pokud bude fungovat chybně (viz odstrašující strategie některých softwarových firem v 90. letech minulého století).

3.8 Spoléhání na konkrétní velikost datových typů

Hodnocení:	**–***
	Program bude těžko přenositelný na jiné platformy. Za pět let nemusí fungovat vůbec.

V jazyce C je normou definován rozměr datového typu `char` (1 byte). U jiných typů jsou dány pouze přibližné vztahy mezi jejich velikostmi. Pokud je potřeba vědět, kolik prostoru skutečně nějaká proměnná nebo datový typ zabírá, slouží k tomuto účelu operátor **`sizeof`**.

Na tomto místě se zmíním ještě o strukturách. V jazyce C není zaručeno, že součet velikostí jejich prvků je roven velikosti struktury. Jazyk C totiž jednotlivé složky struktury v paměti zarovnává tak, aby se na dané architektuře dobře adresovaly a byl k nim co nejrychlejší přístup. S tímto chováním je nutné počítat zejména při alokacích rozsáhlých datových struktur (pole, seznamy), ale i při ukládání do souboru. Tam navíc ještě hrozí problémy s kompatibilitou mezi platformami. Například binární soubor uložený na platformě PowerPC nebude použitelný na platformě x86, protože oba používají jiný druh řazení bytů ve slově (little-endian, big-endian)[7].

Rada: Pokud se budete spoléhat na to, že datový typ `int` zabírá 4B, může váš program s příchodem nových 64 bitových procesorů přestat fungovat a bude v něm potřeba provádět rozsáhlé úpravy. Ušetříte si mnoho starostí, když budete důsledně používat operátor **`sizeof`**.

[7] To samozřejmě neplatí za předpokladu, že programátor tuto vlastnost bere v potaz už při návrhu programu.

3.9 Používání „magických“ čísel

Hodnocení:	** Problémy do budoucna. Snižuje to čitelnost kódu, který se potom špatně opravuje.

Jde o poměrně častou chybu začínajících programátorů. „Magické“ číslo je taková číselná konstanta, která je v programu použita, aniž by byla deklarována jako pojmenovaná konstanta. Častým zdrojem „magických“ čísel bývají specifikace chybových stavů programu. Předpokládejme funkci, která vypisuje chybové hlášení podle zadaného kódu chyby. Tuto funkci je možné (ale ne účelné) volat takto:

```
vypisChybu(4569);
```

Jak je vidět, takové číslo vůbec nic nevypovídá o svém významu. Pro tyto účely existují v jazyce C různé druhy konstant a zejména pak výčtový datový typ (`enum`). Existuje konvence, že konstanty se pojmenovávají velkými písmeny. Je možné je specifikovat na úplném začátku programu, ale často se specifikují i před deklarací funkce, která je používá, aby je bylo možné rychle najít a modifikovat.

V jazyce C existují tři způsoby, jak vytvořit konstantu. Prvním způsobem je deklarovat symbolickou konstantu:

```
#define SPATNY_FORMAT 4569 // pozor! tady není ani =, ani ;
...
vypisChybu(SPATNY_FORMAT);
```

V tomto případě ovšem konstanta nemá specifikován žádný datový typ!

Dále je možné definovat konstantní proměnnou, která se chová jako kterákoli jiná proměnná s deklarovaným datovým typem, ale překladač hlídá, zda se do ní program nepokouší něco zapsat.

```
const int SPATNY_FORMAT = 4569;
```

Třetím typem konstant jsou hodnoty výčtového datového typu `enum`.

```
enum chybovehodnoty
{
  CHYBA_OK,  // implicitně hodnota 0
  CHYBA_SPATNY_FORMAT = 4569,
  CHYBA_VADNY_UZIVATEL
};
```

Tento typ se hodí zejména pro vytváření množin konstantních hodnot s podobným významem, jako jsou třeba chybové kódy. Dobrým zvykem je pojmenovávat konstanty z jednoho výčtu stejnou předponou.

Pozor! Výčet má některá významná omezení! Každá konstanta výčtu je typu `int`. Není možné žádným způsobem otestovat, k jakému výčtu daná konstanta patří. Je zbytečné vytvářet proměnné typu `enum`, půjde do nich stejně přiřadit jakákoli hodnota typu `int`. Překladač neprovádí kontrolu, zda byla přiřazena hodnota odpovídající některé z výčtových konstant.

Velmi užitečné je používat pojmenované konstanty pro meze polí. V tomto případě ovšem nejde použít konstantní proměnnou. Důvodem je skutečnost, že nejde o řádnou konstantu, ale o proměnnou (i když konstantní) a překladač pro deklaraci pole potřebuje konstantu[8].

[8] Až se zase někdy bude upravovat standard jazyka, snad se dočkáme logičtější specifikace typových

```
#define MAXP 20
Tpolozka pole[MAXP];
for (int i = 0; i < MAXTP; i++)
{
  pole[i] = ...;
}
```

Pokud se v budoucnu rozhodnete změnit rozměry pole, lze to provést malou úpravou konstanty `MAXP`. Kdyby se v tomto případě používaly „magické“ konstanty, bylo by potřeba tuto změnu udělat na několika místech v souboru. Pokud se něco takového dělá v knihovně, hrozí nebezpečí, že přestanou fungovat všechny programy, které tento modul používají.

Konstanty jsou velmi užitečné. Na druhou stranu není dobré to s nimi přehánět. Hodnoty jako 0, 1, 2, které se používají pro inkrementaci proměnných nebo pro práci s indexy pole, je většinou neúčelné definovat jako pojmenované konstanty.

Rada: Totéž, co pro „magická“ čísla, platí i pro textové řetězce, i když v menší míře. Doporučuji ukládat jako konstanty veškeré názvy cest a souborů, které jsou definované přímo v kódu. Pokud dodržíte onu konvenci o umísťování konstant na začátek souboru, usnadní se případná budoucí modifikace. Upozorňuji ovšem, že názvy cest a souborů by měly mít v tomto případě význam implicitní hodnoty. Uživatel by měl mít možnost cestu i název souboru změnit (viz kap. 3.4 na str. 12).

Vůbec nejlepší cestou je načítat hodnoty konstant z konfiguračního souboru, ale to už vyžaduje přece jenom větší znalost programování.

3.10 Použití pole namísto struktury

Hodnocení:	** Při budoucích úpravách vznikají problémy.

Datový typ struktura (`struct`) slouží ke sdružování proměnných, které patří logicky k sobě. Například pokud je potřeba vytvořit datový typ reprezentující čas, je nejlepší to udělat takto:

```
typedef tcas
{
  int hodin;
  int minut;
  int sekund;
} TCas;
...
TCas cas;
cas.minut = 10;
```

Použití struktury je mnohem užitečnější než použití pole pro tentýž problém. Příklad s časem by mohl svádět k definici tohoto typu pole:

```
enum slozkycasu { HODIN, MINUT, SEKUND };
int cas[3];
...
datum[MINUT] = 10;
```

konstant. V C++ jsou konstanty navrženy lépe.

Tento příklad se na první pohled jeví rozumně. Dokonce se některé operace mohou zdát jednodušší. Problém však nastane, když vznikne potřeba modifikovat program tak, aby uchovával současně další údaje, například datum a časovou zónu. Změna struktury je velmi jednoduchá:

```
typedef tcas
{
  int hodin, minut, sekund;
  int den, mesic, rok;
  char *zona;
} TCas;

TCas cas;
cas.zona = "Europe/Prag";
```

Podobná úprava ve verzi s polem by šla provést velice obtížně. Nejenom že by to vyžadovalo změnu datového typu nebo definici typu nového, ale navíc by bylo potřeba podstatně předělat všechny doposud vytvořené algoritmy.

Rada: Při analýze problému vždy přemýšlejte, co potřebujete. Pole je sekvenční blok paměti, který se dá indexovat. To u struktury nejde. Struktura zase může obsahovat složky různých datových typů.

Pokud nevíte pro co se rozhodnout, měli byste si uvědomit, že struktura je heterogenní datový typ. Měla by tedy obsahovat položky, které jsou vzájemně odlišné svou podstatou, často popisují nějakou vlastnost objektu (barva, hmotnost, jako vlastnosti slepice). Do pole zase patří prvky, které jsou svou podstatou shodné, pouze se liší hodnotou (slepice, jako obyvatelé kurníku). Neměli byste se příliš ohlížet na použité datové typy. Den, měsíc a rok, jakožto části konkrétního data patří do záznamu, i když pro uložení používají stejný datový typ.

3.11 Použití mnoha proměnných místo struktury

Hodnocení:	** Při budoucích úpravách vznikají problémy. Hrozí riziko zavlečení dalších chyb. Známka amatérismu.

Podobný problém jako v předchozí sekci. Strukturu lze s výhodou využít pro předávání většího množství proměnných pomocí parametrů funkcí. Samozřejmě je vhodné, aby tyto proměnné k sobě nějakým logickým způsobem patřily.

Programy s velkým množstvím proměnných jsou velmi nepřehledné. Někteří začátečníci se snaží problém řešit pomocí polí. Z následujícího odstrašujícího příkladu si rozhodně vzor neberte:

```
char jmeno[10][10];
char prijmeni[10][10];
char narozeni[10][10];

void pracujSLidmi(char jmeno[10][10],
    char prijmeni[10][10], char narozeni[10][10]);
```

Řešení, které využívá vhodně strukturovaná data, je mnohem flexibilnější:

```
#define VALLEN 10;
#define POCET_LIDI 10;
typedef char TValue[VALLEN];
typedef struct osoba
{
  TValue jmeno, prijmeni, narozeni;
} TOsoba;
typedef TOsoba TLide[POCET_LIDI];

void pracujSLidmi(TLide lide);
```

3.12 Podceňování implicitních konverzí

Hodnocení:	** Způsobuje chyby v matematických výrazech.

Jazyk C používá implicitní konverze ve větší míře než jiné jazyky. To znamená, že ve výrazech mění datový typ podvýrazů podle potřeby i bez výslovného uživatelova schválení. Tyto konverze se dějí podle přesně daných pravidel a předpokládá se, že si jich je programátor vědom. To je ovšem velmi optimistické tvrzení. Často se totiž stává, že programátor na nějaké pravidlo zapomene[9] (a nemusí jít o začátečníka).

Příkladem chyby vzniklé v důsledku implicitních konverzí může být tato funkce:

```
float procentoChytrych(int celkem, int chytrych)
{
  return chytrych*100/celkem;
}
..
double chytrych = procentoChytrych(110, 50);
```

Po vykonání bude v proměnné `chytrych` hodnota 45.0 namísto správné hodnoty 45.45454545. Chyba je způsobena tím, že funkce `procentoChytrych` má celočíselné parametry a konstanta 100 je zapsána jako celočíselná hodnota. Překladač v tomto případě zjistí, že ve výrazu jsou samé celočíselné hodnoty, proto zvolí celočíselné dělení. Teprve po celočíselném výpočtu přetypuje výsledek na `float`.

V tomto případě by stačilo zapsat konstantu ne jako 100, ale v její desetinné podobě 100.0. Obecně je ale nebezpečné míchat ve výrazech celočíselné a reálné hodnoty. Pokud to ovšem smysl má, podobně jako v této funkci, je vhodné používat raději explicitní konverze, abyste měli výraz pod kontrolou:

```
float procentoChytrych(int celkem, int chytrych)
{
  return (float)chytrych*100.0/(float)celkem;
}
..
double chytrych = procentoChytrych(110, 50);
```

Dalo by se namítnout, že by stačilo změnit datový typ parametrů funkce z `int` na `float`, což by také tento problém vyřešilo. Například v tomto případě by to sice fungovalo, ale pak by někdo mohl použít tuto funkci s necelými argumenty a funkce

[9]Pokud si vzpomínáte, na začátku kapitoly jsem se zmiňoval, že jazyk C klade větší nároky na pozornost programátora.

by byla používána jinak než bylo zamýšleno[10]. Parametrem funkce může být ovšem také struktura, jejíž složka je použita v nebezpečném výrazu. V takovém případě je prakticky nemožné problém vyřešit jednoduchou změnou typu parametru.

3.13 Vynechaný středník

Hodnocení:	*–**
	Při překladu způsobuje chyby jinde, než byste čekali.

Jde o jednu z nejčastějších chyb v jazyce C. Vzhledem k vlastnostem jazyka C není překladač vždy schopen tuto chybu přesně lokalizovat a může se stát, že ohlásí chybu o značný počet řádků dále, nebo dokonce v jiném souboru (to když středník zapomenete v hlavičkovém souboru). Pokud tedy překladač začne hlásit nějakou naprosto nepochopitelnou chybu, ve většině případů je na vině vynechaný středník (`;`).

Pokud přecházíte z Pascalu, je nutné si uvědomit, že v jazyce C má středník trochu jinou funkci. V jazyce C středník ukončuje příkazy[11]. To znamená, že středník nelze nikde vynechat (ani před koncem bloku). Středník ukončuje každý příkaz každou deklaraci i definici. Výjimkami jsou příkaz bloku a definice funkcí, kde se středník nepíše ani za hlavičkou funkce (narozdíl od deklarace funkčního prototypu) ani za jejím tělem. Naopak pokud definujete proměnnou, musíte ji ukončit středníkem, i když se při inicializaci používají složené závorky – ty zde nehrají roli složeného příkazu, ale inicializátoru.

3.14 Nesprávné používání logických výrazů

Hodnocení:	**
	V jazyce C může způsobit neočekávané chyby.

V jazyce C je podle nové normy možné používat datový typ `bool`[12]. Tento datový typ má pouze dvě hodnoty – `true` a `false`. Je však nutné si uvědomit, že nejde o logický datový typ jako v Pascalu. V jazyce C je typ `bool` kompatibilní s datovým typem `int`. Výsledkem veškerých logických operací je také hodnota typu `int`. Konstanta `false` má hodnotu 0 a `true` má hodnotu 1. Stejných hodnot nabývají také veškeré logické operace. Je však třeba vědět, že při vyhodnocování podmínek v cyklech nebo u příkazu **`if`** se za logicky pravdivou považuje kterákoli **nenulová** hodnota.

Například následující zápis je naprosto legální, ale zamlžuje podstatu problému:

```
if (pocetLidi) {...}
```

Pokud proměnná `pocetLidi` obsahuje nenulovou hodnotu, tělo příkazu se vykoná. Tento způsob zápisu je ovšem málo čitelný. Pokud proměnná neobsahuje skutečně logickou hodnotu (a v tom případě by se měla vhodně jmenovat), je lepší zapisovat podmínku takto:

```
if (pocetLidi > 0) {}
```

Další častou chybou je zneužívání příkazu **`if`** pro přiřazení logické hodnoty do proměnné.

[10] Což je vždy nebezpečné.
[11] Zatímco v Pascalu středník slouží jako oddělovač příkazů, takže ho lze před koncem bloku vynechat.
[12] Pokud jej chcete používat, musíte ovšem nejprve vložit hlavičkový soubor `<stdbool.h>`

```
if (znak == '\027')
{
  jeKonec = true
}
else
{
  jeKonec = false;
}
```

Tuto konstrukci lze mnohem efektivněji přepsat takto:

```
jeKonec = znak == '\027';
```

nebo ještě přehledněji takto:

```
jeKonec = (znak == '\027');
```

Občas se přiřazení logické konstanty do proměnné nelze vyhnout. Je to zejména v případech, kdy logická hodnota vyplývá z delšího úseku kódu jako výsledek výpočtu algoritmu. Zde uvedený příklad ovšem tento případ nepředstavuje.

Další chybou je porovnávání logických proměnných s logickými konstantami `true` a `false` v podmíněných příkazech.

```
if ((lidi <= 10 && volnychMist > 800) == true)
  dejSlevu();
```

Touto konstrukcí na sebe programátor prozrazuje, že neví, jak se v jazyce C provádí vyhodnocování logických výrazů. Porovnávání s logickými konstantami je ošidné, protože v jazyce C není každá pravda rovna `true`:

```
int lidiVSystemu = 10;
if (lidiVSystemu)
{ /* někdo tu je */ }
if (lidiVSystemu == true)
{ /* tohle se provede jen když lidiVSystemu == 1! */ }
```

Správná verze:

```
if (lidiVSystemu > 0)
{ /* někdo tu je */ }
```

Rada: Číselné hodnoty v podmínkách vždy porovnávejte s číselnou konstantou, i kdyby to měla být jednička či nula. Na první pohled je pak zřejmé, že se porovnává číselná hodnota. Naopak logické hodnoty je zbytečné porovnávat s logickými konstantami `true` a `false`.

3.15 Záměna porovnání a přiřazení

Hodnocení:	*
	Jde o chybu, kterou není vždy snadné odhalit.

Tato chyba vzniká často jako překlep. Operátor přiřazení (=) si s operátorem porovnávání (==) pletou začínající uživatelé jazyka C, kteří dříve programovali v Pascalu.

```
if (a = b)
{ /* tento kód se vykoná, když má b nenulovou hodnotu */ }
```

```
if (a == b)
{ /* tohle se vykoná, jen když jsou hodnoty a a b stejné */ }
```

Existují ovšem případy, kdy je použití přiřazení v podmínce žádoucí, například při čtení po znacích ze souboru. Všimněte si, že v tom případě je na výsledek přiřazení aplikován porovnávací operátor.

```
int c;
while ((c = fgetc(f)) != EOF)
{ // vypíše soubor na stdout
  putchar(c);
}
```

V případech, kdy se v podmínce vyskytuje přiřazení, jehož hodnota není v podmínce dále použita, vypisuje překladač varování. Není toho ale schopen vždy. U předešlého příkladu je například poměrně běžný tento překlep, který zcela změní význam podmínky, ale překladač ho není schopen odhalit:

```
int c;
while ((c = fgetc(f) != EOF))
{ // závorka je na špatném místě
  putchar(c);
}
```

Kapitola 4

Podprogramy

Podprogramy jsou základní stavební jednotkou strukturovaného programu. Umožňují řešit složité problémy rozkladem na několik problémů jednodušších. Pokud je navíc podprogram vytvořen tak, aby byl co nejobecnější, lze jej použít opakovaně. Z takových podprogramů lze skládat celé knihovny, které jsou využitelné ve více programech.

4.1 Program bez podprogramů?!

Hodnocení:	*****
	Bez využití podprogramů se složitější problémy stávají ještě složitějšími.

Programovací jazyk C slouží pro psaní **strukturovaných** programů. Podprogramy (funkce) společně se složitějšími datovými typy (pole, struktury) umožňují vytvářet obecné, ale hlavně znovupoužitelné části kódu. Podprogramy pomáhají zpřehlednit zápis programu a ulehčují ladění. Každá funkce může (ale také nemusí) mít parametry a návratovou hodnotu. Díky tomu je možné vytvářet obecnější podprogramy, které řeší celou třídu podobných problémů. Programování se potom tak trochu podobá skládání kostek Lega.

Pro podrobnější informace se podívejte do jakékoli učebnice programování.

4.2 Použití globální proměnné v podprogramech

Hodnocení:	*****
	Způsobuje velmi zákeřné chyby. Snižuje přehlednost a použitelnost podprogramů. Rozhodně se tomu vyhněte!

Používání globálních proměnných v podprogramech je velmi **nebezpečná a nepřehledná** programátorská konstrukce. Chyby, které v důsledku jejich použití vznikají, se velmi těžko hledají.

Je chybou používat globální proměnné pro výměnu hodnot mezi podpbrogramy. K tomuto účelu slouží parametry funkcí. Může se zdát, že se zbytečně zavádějí další proměnné, ale věřte, že to zbytečné není. Například v matematice nikdo nepředpokládá, že při počítání funkce $\sin a$, kam dosadíme za a hodnotu 0.5, se změní b ve výrazu $\sin a+b$. Při programování je to stejné. Takové neviditelné činnosti podprogramu se říká vedlejší efekt (side-effect). Funkce či procedura nesmí měnit žádná data, která nejsou předána přes vstupní/výstupní rozhraní (parametry).

Velké problémy mohou nastat při používání globálních pomocných proměnných. V tomto případě může dojít k nechtěné výměně dat mezi podprogramy. V následující ukázce je vidět co se stane, když jedna funkce do takové proměnné ukládá mezivýsledek a ve druhé funkci je pak tato hodnota nechtěně použita. Takové chyby se velmi těžko hledají a často se nějakou dobu (většinou po dobu testování) vůbec neprojeví.

```
int tmp;

void vymen(int *a, int *b)
{
  // tmp zde může obsahovat jakoukoli hodnotu
  tmp = *a; // !!!
  *a = *b;
  *b = tmp;
}

int suma(int pole[10])
{
  // tmp zde může obsahovat jakoukoli hodnotu
  for (int i = 0; i < 10; i++)
  {
    tmp += pole[i]; // !!!
  }
  return tmp;
}
```

Tento příklad je sice velmi průhledný, ale kdyby obě funkce byly delší a řešily se v nich složitější problémy, nebylo by snadné chybu identifikovat. Taková chyba může snadno vzniknout omylem při úpravách programu.

Rada: Pokud použijete v podprogramu globální proměnnou, většinou tím výrazně omezíte jeho použitelnost. Naproti tomu, pokud použijete místo globálních proměnných parametry, vytvoříte podprogram, který je do jisté míry nezávislý na svém okolí. Takový podprogram můžete volat na různých místech programu s různými hodnotami parametrů. Případně ho můžete použít při řešení dalších projektů.

Na druhou stranu je poctivé říci, že se globální proměnné občas používají i v systémových knihovnách. Jde většinou o speciální případy[1]. Rozumné využití globálních proměnných spadá mezi pokročilé konstrukce, které najdou uplatnění buď v nízkoúrovňových aplikacích nebo ve vícevláknových či paralelních programech. V běžných programech většinou není globálních proměnných potřeba vůbec.

Rada: Pokud přemýšlíte o použití globální proměnné uvnitř funkce, zvažte, zda to opravdu přinese podstatnou výhodu a zda problém nejde řešit čistším způsobem. Pokud přece jen dospějete k závěru, že je použití globální proměnné nevyhnutelné, věnujte zvýšené úsilí dokumentaci. Popište všechny funkce, které s globální proměnnou pracují a vysvětlete, proč je problém řešen tímto způsobem.

[1] Nebo o pozůstatky starých knihoven, které vzhledem k požadavkům na kompatibilitu přežily do dnešních dob

4.3 Parametr místo lokální proměnné

Hodnocení:	**** Jde o naprosté nepochopení práce s proměnnými.

Pokud je potřeba zavést pomocnou proměnnou pro výpočet v podprogramu, nedeklarujeme ji jako parametr, ale jako lokální proměnnou[2]. Parametry slouží k **parametrizaci** podprogramu. Přes parametry komunikuje podprogram s okolím. Lokální proměnné naproti tomu slouží pouze pro vnitřní potřebu podprogramu. Tyto proměnné nejsou vně podprogramu viditelné, takže se ve dvou různých podprogramech mohou vyskytovat lokální proměnné stejného jména.

Něco takového je naprosto nepřípustné:

```
int vypocet(int delka, int mezisoucet)
{
  mezisoucet = delka*delka + 7;
  return mezisoucet - delka;
}
```

Správná konstrukce má tvar[3]:

```
int vypocet(int delka);
{
  int mezisoucet = delka*delka + 7;
  return mezisoucet - delka;
}
```

4.4 Podprogramy nejsou řešeny obecně

Hodnocení:	*** – **** Pozdější úpravy kódu jsou velmi pracné. Kód není použitelný v jiných projektech. Amatérismus.

Málo zkušení programátoři občas nejsou schopni problémy dostatečně zobecnit, takže píší zbytečný kód. Poměrně často se vyskytuje problém s velmi podobnými podprogramy:

```
void zpracujDatum1(TDatum datum1);
{ /* kód */ }
void zpracujDatum2(TDatum datum2);
{ /*až na maličkosti stejný kód jako v prvním případě*/ }
...
  // v hlavním programu
  zpracujDatum1(datum1);
  zpracujDatum2(datum2);
```

Pouze zdánlivé řešení tohoto problému (stejně špatné):

[2]Tato chyba je sice kuriozitou, ale setkal jsem se s ní.
[3]Jde samozřejmě o ilustrativní příklad. Ve skutečnosti zde žádný mezisoučet není potřeba

```
void zpracujData(TDatum data[2])
{
  for (int i = 0; i < 2; i++
  {
    //zpracování
  }
}
...
  // v hlavním programu
  zpracujData(data);
```

Podobné chyby se vyskytují v množství nejrůznějších variant (podobně se dají zneužít příkazy **`switch`** a **`if`**). Druhý případ je nevhodný, protože vyžaduje, aby byly proměnné zpracovávány po dvojicích. Co když v budoucnu bude potřeba zpracovávat lichý počet proměnných?

Správné řešení problému:

```
void zpracujDatum(TDatum datum)
{/* ... */}
...
  // v hlavním programu
  zpracujDatum(datum1);
  zpracujDatum(datum2);
  // zpracujDatum(datum3); když bude potřeba
```

Výjimečné situace lze podle podstaty řešeného problému ošetřit buďto v podprogramu nebo v kódu, kde se tento podprogram používá.

Rada: Má-li být program obecný, neznamená to, že by měl být složitý. Při programování se naopak cení elegance a jednoduchost. Čím složitější podprogram vytvoříte, tím větší je pravděpodobnost, že v něm budou chyby. Pište malé, jednoduché, snadno pochopitelné podprogramy a teprve z nich skládejte řešení složitých problémů.

Tip: V této souvislosti je vhodné zmínit, že velice důležitá je také volba výstižných identifikátorů. U krátkých funkcí je snazší najít výstižný identifikátor než u funkcí, které řeší mnoho dalších problémů. Pište tedy podprogramy tak krátké, aby je bylo snadné výstižně a krátce pojmenovat.

4.5 Chybné použití rekurze

Hodnocení:	*** – ****
	Vyčerpání paměti má za následek havárii aplikace.

Rekurzivní volání podprogramu znamená, že funkci voláte v jejím vlastním těle. Tato programátorská technika může výrazně ulehčit řešení některých problémů, ale má svá úskalí. Pokud není rekurze omezena dostatečně robustní podmínkou, bude se funkce neustále zanořovat a program skončí s chybou přetečení zásobníku.

Vzhledem k tomu, že rekurzi lze převést na iteraci, každou rekurzivní funkci lze převést pomocí cyklu na nerekurzivní řešení. Na druhou stranu je potřeba říci, že to není vždy jednoduché. Nejsnadněji se přepisují takové rekurzivní podprogramy, které volají samy sebe hned na začátku nebo až úplně na konci svého těla.

Nerekurzivní řešení problému bývá často bezpečnější a efektivnější, než jeho rekurzivní varianta.

4.6 Rozkopírovaný kód

Hodnocení:	** – *** Pozdější úpravy kódu jsou velmi pracné. Kopírováním se množí chyby.

Vyplatí se věnovat čas vytváření obecných podprogramů namísto psaní podobných funkcí. Představte si program, který má řešit osmisměrku. Má tedy procházet maticí v osmi směrech a hledat jednotlivá slova. Následující řešení není příliš dobré, protože všechny funkce budou pravděpodobně obsahovat velmi podobný kód:

```
void zlevaDoprava(TMatice *osmismerka);
void zpravaDoleva(TMatice *osmismerka);
void shoraDolu(TMatice *osmismerka);
...
```

Mnohem lepším řešením je napsat jednu obecnou proceduru, která bude osmisměrkou procházet v obecném směru daném směrovým vektorem, přičemž složky vektoru se využijí pro indexování matice s osmisměrkou:

```
void pruchod(TMatice *osmismerka, const TVektor *smer);
```

Rada: Pokud se v programu vyskytuje na více místech zhruba stejný kód, stává se opravování chyb velmi složitým. Když zjistíte, že je chyba na jednom místě, musíte vyhledat všechna podobná místa v programu a opravit je stejným způsobem. Při tom hrozí velké nebezpečí, že na něco zapomenete. V obecném podprogramu stačí chybu opravit jen jednou. Vždy se snažte o co nejobecnější řešení problémů. Vytváření obecných řešení je možná zpočátku pracnější, ale v budoucnu si tím ušetříte spoustu starostí.

4.7 Podprogram dělá více, než by měl

Hodnocení:	*** Snižuje to obecnou použitelnost podprogramu.

Tento problém je zvláště citelný při programování modulů. Uvažujme například funkci, která vkládá prvek do matice (nebo složitější struktury). Potom tato funkce musí vykonat právě tuto činnost a žádnou jinou. Je naprosto nepřípustné, aby něco vypisovala na obrazovku nebo se dokonce na něco tázala uživatele. Podprogramy v modulech musí být napsány tak, aby je bylo možné použít v co nejširším spektru programů. Pokud by zmíněná funkce komunikovala s uživatelem, nebylo by ji možné použít například v grafické aplikaci.

Rada: Pište co nejjednodušší podprogramy. V krátkém podprogramu se lépe hledají chyby. Pokud programujete složitější algoritmus, je lepší jej poskládat z jednodušších podprogramů.

Rada: Zajistěte, aby vaše podprogramy vykonávaly jenom ty nejnutnější operace. Pokud chcete uživateli vaší knihovny nabídnout nějakou funkčnost navíc, napište další funkci. Ponechte uživateli možnost volby mezi verzí funkce se základní funkčností a verzí s přidanou hodnotou.

4.8 Podprogramy nekontrolují své parametry

Hodnocení:	*** Nebezpečný zlozvyk, zvláště ve spojení s ukazateli.

Každý podprogram by měl být co nejvíce nezávislý na svém okolí. To znamená, že by se měl zachovat korektně i v případech, kdy mu uživatel předá chybná data. V pokročilejších jazycích pro řešení těchto situací existuje mechanismus výjimek. V jazyce C nic takového není[4]. To však neznamená, že by programátor měl na ošetřování těchto stavů rezignovat.

K problémům dochází zejména při předávání aritmetických hodnot a ukazatelů. V následujícím příkladě mohou nastat problémy:

```
void praceNaPoli(int index, int pole[])
{
  pole[index] = ...
  ...
}
```

Snadno se může stát, že někdo předá parametr `index` mimo rozsah pole. S ohledem na efektivitu je vhodné si zvyknout na používání makra `assert()`[5]. Parametrem je logický výraz, který má význam předpokladu. Pokud předpoklad neplatí, program se ukončí a vypíše se chybové hlášení, na kterém řádku došlo k porušení předpokladu. Tímto způsobem se dají zjišťovat nebezpečné chyby během ladění:

```
void praceNaPoli(int index, int pole[])
{
  assert(index >= 0 && index < MAXIMUM);
  pole[index] = ...
  ...
}
```

Pokud není možné spolehlivě zajistit, aby během vykonávání programu byla hodnota parametru `index` ve správných mezích, je lepší testovat jeho hodnotu pomocí příkazu **if** a vracet chybový kód. Chybový kód se často vrací jako návratová hodnota funkce, například takto:

```
int praceNaPoli(int index, int pole[])
{
  if (index < 0 || index >= MAXIMUM)
    return CHYBA_MEZE;
  pole[index] = ...
  ...
}
...
chybovyKod = praceNaPoli(i, pole);
```

Další typickou oblastí, kde neošetření parametrů působí vážné potíže, jsou ukazatele. Tento případ je rozebrán v kap. 5.4 na str. 35.

---

[4] Výjimky jsou až v C++.

[5] Je nutné vložit hlavičkový soubor `<assert.h>`. Po odladění a dostatečném otestování stačí do kódu vložit řádek `#define NDEBUG`. Bližše viz `man assert`.

4.9 Zkrácené vyhodnocování logických výrazů

Hodnocení:	** – *** Program se nemusí vždy chovat tak, jak bylo zamýšleno.

V jazyce C je dáno normou, že se bude používat zkrácené vyhodnocování logických výrazů. V praxi to znamená, že se vyhodnocování logického výrazu ukončí v okamžiku, kdy je jasné, jaká bude jeho výsledná hodnota. V důsledku toho není zaručeno, že se vždy vyhodnotí všechny části logického výrazu. Pokud logický výraz obsahuje volání funkcí, není zaručeno, že budou vykonány za všech okolností. Této vlastnosti se často využívá, ale programátor si musí být vědom, jak tento mechanismus funguje.

```
if (list1 != NULL && list2 != NULL && !jsouStejne(list1, list2))
```

Tento příklad je často používaný idiom. K zavolání funkce `jsouStejne` nedojde, pokud bude některý z ukazatelů `list1`, `list2` obsahovat hodnotu `NULL`.

Rada: Nikdy nevolejte funkci uprostřed logického výrazu, pokud chcete zaručit její zavolání (například proto, že logická hodnota je jenom jedním z výsledků). V takovém případě raději uložte výsledek volání funkce do pomocné proměnné a pracujte až s ní.

4.10 Používání podprogramů ve výrazech

Hodnocení:	** – *** Program se nemusí vždy chovat tak, jak bylo zamýšleno.

Jazyk C nezaručuje přesné pořadí vyhodnocení jednotlivých složek některých výrazů[6]. Překladač může změnit pořadí vyhodnocování operandů v rámci optimalizací, pokud dojde k závěru, že to nebude mít vliv na výsledek. Z pohledu samotného výrazu je to naprosto bezpečné, ale problémy mohou nastat, pokud se ve výrazu vyskytne více funkcí s vedlejšími efekty.

Rada: V okamžiku, kdy je nutné zajistit přesné pořadí volání funkcí ve výrazu, je lepší použít pomocné proměnné. Tento příklad ukazuje, jak jsou funkce s vedlejším efektem nebezpečné.

4.11 Chybné používání parametrů v podprogramech

Hodnocení:	** Může způsobovat problémy s efektivitou a občas i další chyby.

V jazyce C existují dva typy předávání parametrů. Základním typem je **předávání hodnotou**. Deklarace funkce s tímto typem parametru má tvar:

```
void necoUdelej(int parametr);
```

Hodnota parametru předávaného hodnotou se při zavolání **zkopíruje** na zásobník vyhrazený podprogramu. Tuto hodnotou je možné v podprogramu modifikovat, aniž by se to projevilo vně podprogramu. Z toho vyplývá, že při volání podprogramu lze jako parametr použít i konstanty.

[6] Přesné pořadí vyhodnocení operandů je dáno pouze u operátorů `&&`, `||` a u operátoru čárka

Druhým typem předávání parametrů je **předávání odkazem**. V jazyce C se to řeší pomocí ukazatelů předávaných hodnotou:

```
void necoUdelej(int *parametr)
{
  *parametr = *parametr * 10;
}
```

Na zásobník podprogramu se tentokrát nekopíruje hodnota parametru, ale ukazatel na něj. Pokud podprogram modifikuje hodnotu odkazovanou tímto ukazatelem, projeví se to i vně podprogramu. Jde totiž o paměť, která leží mimo paměťový prostor podprogramu a po jeho ukončení nezaniká[7]. Tento typ parametru se používá pro vracení výsledků podprogramu. V tomto případě je možné při volání použít jako parametr výhradně ukazatele na proměnné (do konstanty by nešlo zapisovat). Při volání je nutné zajistit, aby se skutečně předal ukazatel:

```
int x = 10;
necoUdelej(&x); // adresa proměnné x
```

Pozor při práci s poli! Specifikum jazyka C (a C++) spočívá v tom, že ačkoli to syntakticky vypadá, že se pole předává hodnotou, vždy se ve skutečnosti předává ukazatel (ale při volání se nepíše &). Pole ve skutečnosti nelze předat hodnotou[8], pouze odkazem (pole se nikdy nebude kopírovat na zásobník).

Jméno parametru platí (je viditelné) jenom v rámci svého podprogramu. Proto je možné mít více podprogramů se stejně pojmenovanými parametry. Totéž platí i pro lokální proměnné.

Rada: Záměna jednoho způsobu předávání parametru za druhý může vést k problémům. Parametr předávaný odkazem používejte pouze tehdy, když jeho prostřednictvím chcete vracet výsledek. V ostatních případech je lepší používat předávání hodnotou. Výjimku z tohoto pravidla můžete udělat, pokud pracujete s rozsáhlými strukturami. V tomto případě by se při předávání hodnotou muselo na zásobník kopírovat velké množství dat, takže je lepší používat předávání odkazem[9].

4.12 Procedury versus funkce

Hodnocení:	* – **
	Může vést k méně přehlednému kódu.

Funkce je podprogram, který vrací hodnotu. V jazyce C může funkce vracet hodnotu jakéhokoliv datového typu, na který lze aplikovat přiřazovací operátor. Prakticky jde tedy o všechny základní datové typy (včetně struktury), kromě polí.

Procedura je taková funkce, která má návratový typ `void`. I taková funkce ovšem může vracet hodnoty pomocí parametrů předávaných odkazem. Občas působí problémy se rozhodnout, zda vracet hodnotu přes parametr nebo jako výsledek funkce.

Použití procedury v následujícím případě je chybné:

```
void vratMocninu(float x, float *mocnina);
```

Naproti tomu funkci je možné použít přímo ve výrazu:

[7] Zjednodušeno pro lepší čitelnost.

[8] To platí pro jazyky C a C++. V Pascalu je možné předat pole i hodnotou, ale vzhledem k neefektivitě se to nepoužívá.

[9] V dřívějších verzích jazyka vůbec nebylo možné předávat struktury hodnotou, pouze odkazem. I dnes se struktury předávají hodnotou jen výjimečně.

```
float vratMocninu(float x);
...
koren1 = (vratMocninu(b) + D)/(2*a);
```

Tip: Proceduru využívejte jenom pro podprogramy, které žádnou hodnotu nevracejí, nebo pokud potřebujete vracet hodnoty složitějších typů či větší počet parametrů (v takovém případě je dobré zvážit použití struktury). Pokud by měla procedura vracet jediný parametr jednoduchého typu, je lepší ji vytvořit jako funkci. Funkce jsou také vhodné pro případy, kdy je nutné vracet chybový stav podprogramu.

Kapitola 5

Ukazatele a pole

Ukazatele bývají kamenem úrazu nejen pro začátečníky. Jde o největší zdroj chyb v programátorské praxi. Je tedy vhodné jim věnovat zvýšenou pozornost.

Problémy při práci s poli jsou v jazyce C velice úzce spojené s ukazateli, ať už jde o alokaci nebo o indexování za hranicí pole.

Tip: Pokud nemáte dobrou prostorovou představivost, zkuste si dynamické struktury kreslit na papír.

5.1 Segmentation fault

Hodnocení:	***** Chyba při práci s pamětí je vždy závažná. U primitivních operačních systémů může poškodit celý systém.

Všechny dále zmíněné chyby způsobí v chráněném režimu procesoru ukončení programu[1]. V chráněném režimu má aplikace k dispozici teoreticky veškerou dostupnou paměť systému. V tomto režimu je paměťový prostor programu chráněn před ostatními aplikacemi. Pokud se program pokusí pracovat s cizí pamětí (tj. s pamětí, kterou si sám nealokoval), operační systém jej ukončí. Hlídá se jak zápis, tak i čtení z nealokované paměti.

Standardně se při paměťové chybě vypíše pouze chybové hlášení bez bližší specifikace. V Linuxu (pokud je to povoleno) se při takové události generuje soubor `core`. Ten obsahuje přesnou podobu paměti přidělené programu v okamžiku pádu, takže je možné ji analyzovat pomocí ladícího programu.

Problém je, že na havárii programu při této chybě se nedá spoléhat. Standardní knihovna jazyka C při alokaci většinou přiděluje o něco více paměti, než je třeba[2]. To přináší urychlení, když program potřebuje rozšířit doposud alokovanou paměť, ale zároveň to poněkud ukrývá chyby, které vznikají při indexování polí.

Tip: V prostředí Linuxu existuje program *valgrind* [val04], který dokáže část takových chyb detekovat. Pro Windows existují podobné programy, ale nejsou volně šiřitelné.

[1] V unixu se navíc vypíše chybová zpráva „Segmentation fault“ a pokud to v systému není zakázáno, vygeneruje se soubor `core`.

[2] Alokuje se o něco více z důvodu zarovnávání. Přístup k nezarovnané paměti by byl na moderních procesorech velmi neefektivní. Skutečná velikost alokované paměti je zarovnaná na vhodný násobek mocniny dvou.

Rada: V reálném režimu procesoru paměť nijak chráněna není, takže chybný přístup do paměti může způsobit poškození nejen vašeho programu či dat, ale i cizích programů či celého operačního systému. Naštěstí se v dnešní době s reálným režimem běžně nesetkáte (DOS už dnes používá málokdo). Můžete na něj ovšem narazit, pokud byste byli nuceni programovat nějaké průmyslové aplikace, jednočipy a podobně. Pokud budete nuceni pracovat v takovém prostředí, věnujte testování mnohem více času než jinde.

5.2 Program neuvolňuje dynamicky alokovaná data

Hodnocení:	***** Zákeřná chyba, která neohrožuje pouze vaši, ale i všechny okolní aplikace.

Paměť žádného systému není neomezená, proto je třeba veškerou dynamicky alokovanou paměť uvolnit, když už ji nepotřebujete. Pro uvolňování dynamicky alokované paměti se používá funkce `free`[3]. Neuvolňování paměti je zvláště nebezpečné u programů, které běží dlouho. U nich hrozí nebezpečí, že časem spotřebují veškerou dostupnou paměť a způsobí zhroucení systému. Neohrožují tedy jenom sebe, ale i všechny ostatní programy a jejich data.

Uvolňování paměti může činit potíže zvláště u dynamických datových struktur jako jsou spojové seznamy, stromy a podobně. V těchto případech je vhodné si ověřit, že jste všechnu paměť skutečně uvolnili. K tomuto účelu je možné opět využít program *valgrind* [val04]. Existují i knihovny, které realizují vlastní způsob alokování paměti, který se hodí lépe k ladícím účelům (*dmalloc* [dma04], *ccmalloc* [ccm04], *Electric Fence* [ele04]).

Rada: Vždy věnujte velkou pozornost tomu, zda správně a včas uvolňujete vše, co alokujete. Nenechávejte uvolňování nepotřebné paměti až na konec programu. Nepotřebnou paměť uvolňujte hned, jak je to možné.

5.3 Ukazatele odkazují do nealokované paměti

Hodnocení:	**** V DOSu dokonce může poškodit operační systém. Jinde způsobí havárii aplikace.

Když se nějakým způsobem do proměnné typu ukazatel dostane odkaz do nealokované paměti, nastávají problémy. Pokud program do těchto míst něco zapíše, může to vést na některých systémech (DOS) až k pádu celého systému. U systémů pracujících v chráněném režimu procesoru program skončí s chybou segmentation fault (viz kap. 5.1 na str. 32).

Chyba vzniká často nedůslednou inicializací ukazatelů. Tomu je možné předcházet včasnou inicializací ukazatele buďto voláním `malloc` nebo přiřazením `NULL`.

[3]Ke každému volání `malloc` patří jedno volání `free`.

Někdy ovšem bývají začátečníci příliš pečliví a s inicializací to přehánějí:

```
TData *prvek = malloc(sizeof(TData));
prvek = NULL; //tady rozhodně NE!
prvek->data = ...;

TData *druhy; //tady chybí alokace nebo inicializace
druhy->data = ...;
```

V prvním případě se nejprve alokuje paměť pro `prvek`, ale vzápětí se tato paměť znepřístupní přiřazením `NULL`. Ve druhém případě nebyl ukazatel inicializován vůbec.

Nelegální ukazatel se může do proměnné dostat i tehdy, jestliže více ukazatelů odkazuje na stejnou paměť:

```
TData *pomocny = prvni;
...
free(prvni); // pomocny nyní odkazuje do nealokované paměti
```

Tento typ chyby se často objevuje při práci s dynamickými strukturami jako je seznam či strom. Je možné ji detekovat pomocí programu *valgrind* [val04].

Chyba se obzvláště často vyskytuje v podprogramech. Pokud je parametrem podprogramu ukazatel, předání ukazatele do nealokované paměti způsobí havárii. Tuto situaci naneštěstí nelze detekovat přímo, ale pouze s pomocí testovacích programů jako *valgrind*.

V některých případech může být použití hodnoty `NULL` legální z podstaty řešeného problému:

```
void uvolniPrvek(TPrvek *prvek);
{
  if (prvek == NULL) return; //nic se neděje
  uvolniCast(prvek->cast);
  ...
  free(prvek);
}
```

Když má ukazatel v parametru význam předávání odkazem, je vhodné testovat hodnotu ukazatele pomocí makra `assert`. Pokud v takovém případě dojde k předání `NULL`, jde téměř vždy o programátorskou chybu, protože se dá předpokládat, že běžným chodem programu k takovému stavu nemůže dojít:

```
TErrCode readFile(FILE *f, TData *data);
{
  assert(f != NULL);
  assert(data != NULL);
  ...
}
```

Rada: Zvykněte si definovat lokální proměnné (nejen ukazatele) až těsně před místem, kde jsou potřeba. Pokud je proměnná definována příliš daleko od svého prvního použití, hrozí nebezpečí, že ji omylem použijete dříve, než jste zamýšleli. To může znít poněkud nelogicky, ale může se to lehce stát v okamžiku, kdy program upravujete po nějaké delší době od jeho vytvoření. Potom snadno vícenásobné použití proměnné přehlédnete. Definování lokálních proměnných těsně před použitím by se mělo stát jedním z vašich sebeobranných reflexů. Vyhnete se tím budoucím problémům.

5.4 Porovnávání ukazatelů, které odkazují do nealokované paměti

Hodnocení:	**** Zákeřná chyba, kterou nemusíte odhalit ani laděním.

Jde v podstatě o stejný problém jako v předchozí sekci. Tato chyba je stejně závažná jako zákeřná. Její zákeřnost spočívá v tom, že se v reálném režimu často neprojeví ani při ladění. V chráněném režimu takový program může skončit s chybou segmentation fault, ale není to zaručeno (viz kap. 5.1 na str. 32). Prohlédněte si následující příklad:

```
/* Porovná dva seznamy. */
bool stejneSeznamy(TSeznam *seznam1, TSeznam *seznam2)
{
  bool stejne = true;
  while (seznam1->dalsi != NULL && seznam2->dalsi != NULL
         && stejne)
  {
    stejne = seznam1->data == seznam2->data;
  }
  return stejne;
}
```

Problém je v podmínce cyklu **while**. Pokud bude mít při volání funkce jeden z parametrů hodnotu `NULL`, dojde při vyhodnocování podmínky k použití adresy z nealokovaného prostoru (`NULL->dalsi`).

V reálném režimu procesoru se to vůbec nemusí projevit. Dokonce se může zdát, že program funguje. Obecně může takový odkaz obsahovat náhodnou hodnotu. V chráněném režimu dojde k havárii programu. Podmínka by v tomto příkladě měla mít tvar:

```
while (seznam1 != NULL && seznam2 != NULL && stejne)
```

5.5 Indexace za hranicí pole

Hodnocení:	**** Zákeřná chyba, která se těžko odhaluje a nese s sebou vážná bezpečnostní rizika.

Jazyk C žádným způsobem nehlídá překročení meze polí[4]. Ponechává to plně na zodpovědnosti programátora.

U dynamických polí odpovídá indexování mimo rozsah pole přístupu do nealokované paměti se všemi důsledky (viz předchozí oddíly). Lze ji tedy i stejným způsobem detekovat. U lokálních polí je problém o to závažnější, že neexistuje obecný způsob, jak tuto chybu odhalit. Lokální pole se alokuje na zásobníku, který je společný pro všechny podprogramy. Indexací mimo meze lokálního pole se program pohybuje v paměti, ke které má dostatečná práva, takže na to nezareaguje ani systém ochrany paměti.

Pokud podprogram omylem modifikuje data za hranicí lokálního pole, může modifikovat hodnoty nejenom svých lokálních proměnných, ale i hodnoty lokálních proměnných jiných podprogramů, protože všechny jsou alokovány ve stejném zásobníku.

[4] Překladač není principiálně schopen tyto kontroly generovat. Je to dáno definicí jazyka C, respektive úzkým vztahem polí a ukazatelů.

```
void test(void)
{
  int a = 12;
  int pole[8];
  int b = 34;
  pole[-1] = 45; // !!!
}
```

V tomto případě velice pravděpodobně dojde k nechtěné modifikaci proměnné `b`. Indexování za maximální hranicí by mohla poškodit lokální proměnné podprogramu, který by funkci `test` zavolal.

Této chybě se v literatuře říká přetečení vyrovnávací paměti (anglicky *buffer overflow*) a je nechvalně známá bezpečnostními riziky, které přináší. Pokud program chybným způsobem načítá vstupní hodnoty, umožňuje tato chyba modifikovat vnitřní proměnné programu vhodně zvolenými vstupními daty. To je zvláště nebezpečné například u programů, které ověřují hesla nebo počítají peníze.

Tip: Pro ladicí účely používejte dynamická pole. Chybnou indexaci potom můžete odhalit pomocí programu *valgrind*. Po otestování se můžete k lokálním polím vrátit. Lokální pole jsou o něco efektivnější než dynamická, zvažte však, zda jejich použití stojí za zvýšené bezpečnostní riziko.

5.6 Spoléhání na konkrétní velikost datových typů při alokaci

Hodnocení:	**–*** Program bude obtížně přenositelný na jiné platformy. Za pět let nemusí fungovat vůbec.

Jde o podobný problém jako v kap. 3.8 na str. 15. Spoléhání na konkrétní velikost datových typů je chybou. Obzvláště to platí při alokování paměti:

```
int *pole = malloc(10*4); // !!!
```

Velikost datových typů se může lišit na různých platformách. V tomto případě by program sice na 32 bitové platformě x86 fungoval správně, ale nemusí to být pravda za několik let, kdy se budou používat 64 bitové procesory.

Mnohem spolehlivější je používat operátor **`sizeof`**, který vrací velikost datového typu nebo proměnné v bytech.

```
int *pole = malloc(10*sizeof(int));
```

5.7 Při alokování se nedetekuje chyba

Hodnocení:	**–*** Častá chyba, která ve výjimečných případech může způsobit havárii operačního systému.

Funkce `malloc` alokuje požadované množství paměti. Pokud při alokaci dojde k chybě, vrátí místo ukazatele `NULL`. Korektní program by měl po každé alokaci kontrolovat, zda vše proběhlo v pořádku a nějak se vyrovnat chybou, která vznikla pravděpodobně nedostatkem paměti.

Řešením, které je doporučováno dokonce i v manuálu knihovny `libc`, je vytvoření pomocné funkce, která bude fungovat jako `malloc`, ale v okamžiku nedostatku paměti vypíše chybové hlášení a program ukočí. V tomto případě se spoléhá na to, že operační systém sám uvolní paměť, kterou si program naalokoval. Vzhledem k nepřítomnosti mechanismu výjimek v jazyce C by obecné řešení pro uvolňování paměti za těchto okolností bylo příliš složité (záleží na složitosti programu, někdy to nemusí být velký problém):

```
void *xmalloc (size_t size)
{
  register void *ptr = malloc (size);
  if (ptr == NULL)
  { // tady se můžete pokusit paměť uvolnit
    fatal (”virtual memory exhausted”);
    // častější je ale ukončení programu
    // exit(EXIT_FAILURE);
  }
  return ptr;
}
```

Kapitola 6

Soubory

Práce se soubory také patří mezi oblasti, ve kterých se často chybuje. Vzhledem k tomu, že přístup k souboru může být značně pomalý, vznikají problémy s efektivitou. Naštěstí není příliš těžké se těmto problémům vyhnout.

6.1 Proč se vyhýbat funkci `gets`

Hodnocení:	***** Toto je nebezpečná funkce. Nikdy ji nepoužívejte!

Funkce `gets` slouží pro načítání po řádcích ze standardního vstupu `stdin`. Ukládá načtená data do uživatelsky alokovaného pole, ale žádným způsobem nehlídá, zda nedochází k překročení mezí. Zůstala po prvních verzích standardní knihovny jazyka C. Pravděpodobně byla vytvořena v době, kdy se zdálo, že všechny terminály mají pouze 80 znaků na řádek a nebylo známo přesměrovávání souborů[1].

Namísto funkce `gets` je bezpečnější používat funkci `fgets`, která umí číst ze souboru a má další parametr, kterým se nastavuje maximální počet znaků, které se přečtou. Pokud je potřeba tímto způsobem číst ze standardního vstupu, je nutné ji předat místo souboru proměnnou `stdin`, což je systémová proměnná odkazující na standardní vstupní proud (stream).

Rada: Ne všechny funkce ve standardní knihovně jsou bezpečné. Vždy sledujte aktuální dokumentaci k funkcím, které používáte. Některé standardní funkce jsou označeny jako zastaralé (nebo zavržené, anglicky deprecated). K těmto funkcím vždy existují novější, bezpečnější alternativy. Zastaralé funkce v knihovně zůstávají pouze proto, aby byla zachována kompatibilita se staršími verzemi knihovny. Nikdy je nepoužívejte.

6.2 Není ošetřeno otevírání souborů

Hodnocení:	**** Program se dá označit za nedodělek. Vede to k tomu, že je program nepoužitelný.

Při pokusu o otevření souboru pomocí funkce `fopen` může obecně dojít k chybám. Soubor nemusí vůbec existovat, případně nemusí mít nastavena dostatečná přístupová práva. V tomto případě funkce `fopen` namísto ukazatele na datový typ `FILE` vrací

[1]Nejenom začátečníci dělají chyby. S podobně vadnými funkcemi se ovšem lze setkat u většiny déle používaných knihoven (nejen v jazyce C). Vznikají jako přirozený důsledek vývoje.

NULL. Je nutné vždy otestovat, zda nedošlo k chybě. Následující příklad je chybný:

```
FILE *f = fopen(cesta, "r");
...
int c = fgetc(f);
```

Správná verze:

```
FILE *f = fopen(cesta, "r");
if (f == NULL)
{
  fprintf(stderr, "Soubor %s nejde otevřít.", cesta);
  // ukončení aplikace nebo jiné ošetření situace
}
```

6.3 Program neuzavírá otevřený soubor

Hodnocení:	*** Může vést k poškození nebo zbytečnému blokování přístupu k souboru.

Standardní knihovna jazyka C používá pro přístup k souborům vlastní vyrovnávací paměť. Otevřené soubory se automaticky korektně uzavřou na konci funkce `main`, pokud v programu nedojde k chybě. V okamžiku, kdy program skončí s chybou (například voláním `exit`), není zaručeno, že budou otevřené soubory korektně zapsány na disk. Z toho vyplývá, že programátor by měl zajistit korektní uzavření souboru pomocí volání funkce `fclose` ihned poté, co s ním program přestane pracovat.

Spoléhat na automatické uzavření souboru je nebezpečné ještě z jednoho důvodu. Pokud program běží dlouho, zbytečně by blokoval přístup k souboru, se kterým už nepracuje[2].

Rada: Ke každému volání `fopen` patří jedno volání `fclose`. Soubor se z nejrůznějších příčin nemusí povést uzavřít (například porucha disku, přerušení síťového spojení, ...). Slušný program potom testuje návratový kód funkce `fclose`, aby zjistil, jestli se uzavření povedlo. Pro přesnější určení typu chyby je možné použít funkci `ferror`, která vrací chybový kód posledně provedené operace nad souborem.

6.4 Několikanásobný průchod celým souborem

Hodnocení:	*** Neefektivní. Při větších souborech nepoužitelné.

Většina problémů se dá vyřešit jediným přečtením souboru. Pevný disk je ta nejpomalejší paměť v celém počítači. Pokud vezmeme v úvahu, že soubor může být velmi velký (stovky MB), každé přečtení souboru je potenciálně velice časově náročnou operací.

Častou chybou začínajících programátorů je při prvním průchodu spočítat řádky souboru a při druhém průchodu teprve načítat data. Tento způsob práce je naprosto

[2] Dlouhý běh programu nemusí být vždy plánovaným chováním. Pokud se program dostane v důsledku chyby do nekonečného cyklu, může blokováním přístupu k souborům značně omezovat jiné programy.

zbytečný (viz idiom v 3.3 na str. 12) a navíc jej nelze použít při práci se standardním vstupem (`stdin`).

Rada: Pokud čtete soubor několikrát, aniž by to bylo nutné, vypovídá to o vaší programátorské neschopnosti nalézt efektivní algoritmus. Vždy se snažte o co nejúspornější jednoprůchodové řešení.

6.5 Načítání souborů po znacích (char)

Hodnocení:	***
	Vede k nekonečným cyklům při pokusu o čtení za koncem souboru.

Funkce `fgetc`, která v jazyce C slouží pro čtení ze souboru po znacích, nevrací datový typ `char`, ale trochu překvapivě `int`. Z toho důvodu je následující kód chybný:

```
char c; // tady je zakopaný pes!
while ((c = fgetc(f)) != EOF)
{
  // zpracuj znak c
}
```

Tento kód buď nikdy neskončí, nebo může skončit předčasně. Funkce `fgetc` (ale i getchar) vrací hodnotu konstanty `EOF`, což je typicky -1. Při pokusu o uložení této hodnoty do proměnné typu `char`, dojde k automatické konverzi a hodnota -1 pak bude splývat[3] s hodnotou znaku s kódem 255. Pokud se v souboru nevyskytuje znak s kódem 255, algoritmus uvízne v nekonečném cyklu. Pokud se v souboru znak s tímto kódem nachází, dojde k předčasnému ukončení cyklu.

Pro čtení souboru po znacích existuje v jazyce C tento idiom:

```
int c; // teď je to správně
while ((c = fgetc(f)) != EOF)
{
  // zpracuj znak c
}
```

Skutečnost, že funkce `fgetc` vrací `int`, má tedy svůj dobrý důvod. Umožňuje jí to vracet všech 256 možných znaků a zároveň detekovat konec souboru. Je často používanou konstrukcí, že funkce vrací hodnotu většího datového typu než je potřeba, aby mohla zároveň vracet chybový kód.

6.6 Spoléhání na test existence souboru

Hodnocení:	**–***
	Není zaručeno, že tato konstrukce bude funkční. Programátor by si toho měl být vědom.

Poměrně častým omylem při práci se soubory je následující konstrukce, kdy se programátor snaží ušetřit si práci s testováním při otevírání souborů:

[3] Pokud by EOF měla jinou hodnotu, splývala by po konverzi s jiným znakem.

```
bool existujeSoubor(const char *jmeno);
{
  FILE *f fopen(jmeno, "r");
  bool bezChyby = (f != NULL);
  if (!bezChyby)
    fclose(f);
  return bezChyby;
}
...
if (existujeSoubor(jmenoSouboru))
{
  FILE *f = fopen(jmenoSouboru, "r");//!! už nemusí existovat
  int data = fgetc(f);
  ...
}
```

Funkce `existujeSoubor` se může na první pohled jevit jako dobrý nápad. Nicméně je tato funkce naprosto zbytečná. Problém spočívá v tom, že u moderního operačního systému nikdo nemůže zaručit, že mezi voláním této funkce a pokusem o otevření souboru bude soubor pořád beze změny existovat. Je to důsledek použití multitaskingu. Kterýkoli jiný program mohl během té chvíle začít s uvedeným souborem pracovat nebo jej smazat.

6.7 Zbytečné zavírání a otevírání souborů

Hodnocení:	** Neefektivní. Občas může vést k havárii a ztrátě dat.

Pokud řešení problému nutně vyžaduje několikanásobný průchod souborem, je zbytečné a nebezpečné jej po každém přečtení uzavírat voláním `fclose` a pak jej znovu otevírat pomocí funkce `fopen`. Pro nastavení čtecí hlavy na začátek souboru slouží funkce `rewind`. Pokud je potřeba začít číst uprostřed souboru, používají se k tomu funkce `fseek` a `fsetpos`.

Zejména začátečníci chybně předpokládají, že pokud je soubor uzavřen a vzápětí znovu otevřen, tak není potřeba testovat chybový stav při druhém otevření. Předpokládají totiž, že soubor musí být pořád na disku. To ovšem není zaručeno – může dojít k přerušení síťového spojení nebo jiný program může soubor modifikovat nebo smazat.

Tip: Pokud otevíráte soubor pouze pro čtení, je možné jej otevřít několikrát do více proměnných. Každá proměnná potom má jakoby svou vlastní čtecí hlavu. To znamená, že při čtení z různých proměnných typu `FILE *` asociovaných se stejným souborem, můžete zároveň číst na různých místech tohoto souboru. Podobným způsobem je možné otevřít soubor pro zápis, ale potom je potřeba počítat s problémy, které z toho vyplývají.

Kapitola 7

Problémy s ovládáním programů

V této části se budu zabývat etiketou. Programy často komunikují s uživatelem, proto by se měly držet určitých společenských pravidel. Tento požadavek není až tak samoúčelný, jak by se mohlo na první pohled zdát. Způsob, jakým program komunikuje s uživatelem, mnohé napoví o jeho autorovi. Program, který je neovladatelný nebo nepohodlný, nikdo používat nebude.

7.1 Program nedetekuje chybový stav

Hodnocení:	*** – **** Velmi závažná chyba. Znamená, že autor zanedbal testování. Program je nedodělek a byl uveřejněn předčasně.

Jedním z nejzávažnějších nedostatků je neošetření chyby, pro kterou program skončí (například segmentation fault). Nejhorší na tom je skutečnost, že uživatel může přijít o data. Dále v paměti mohou zůstat neuvolněné bloky paměti a může dojít k poškození otevřených souborů. Neošetřené chyby nese každý uživatel velmi nelibě (jistě máte vlastní zkušenosti s chybujícími programy).

Rada: Testování aplikace by mělo zabrat nejméně tolik času jako vlastní programování, spíše by ale mělo trvat déle. Časem zjistíte, že aktivní programátor stráví více než dvě třetiny času testováním a opravováním chyb[1].

7.2 Program neošetřuje chybný vstup od uživatele

Hodnocení:	*** – **** Velmi závažná chyba. Znamená, že jste jako autor zanedbali testování. Chyba může znamenat bezpečnostní riziko.

Nezkušení programátoři často předpokládají, že uživatel zadá programu vždy správné údaje. Na to se však nedá nikdy spoléhat. Uživatel může zadat nesprávné údaje například omylem (překlep, atd.) Program, který skončí chybou po nesprávném zadání údajů nelze používat.

Tato chyba často souvisí s chybou přetečení vyrovnávací paměti (buffer overflow, viz kap. 5.5 na str. 35). U některých aplikací může vést až k získání neoprávněného přístupu (do systému, databáze, kamkoli) nebo k jinému útoku.

[1] Což rozhodně nemusí být tak nudná práce, jak to na první pohled vypadá.

Rada: Vždy důkladně analyzujte, co všechno může uživatel vašemu programu zadat. Když mu zabráníte v zadávání chybných údajů, zvýší se tím o něco bezpečnost aplikace[2]. Ovšem mnohem lepší než implementace jednoúčelových omezení je vytváření dostatečně robustních podprogramů.

7.3 Přehnaná komunikace s uživatelem

Hodnocení:	** – *** Velmi snižuje použitelnost programu.

Nyní budeme mluvit o nevizuálních aplikacích spouštěných z příkazové řádky. Programování okenních aplikací se začátečníků netýká.

Zvláště začátečníci mají často pocit, že program musí vést s uživatelem dialog. Neustále se uživatele vyptávají na nejrůznější parametry, v extrémních případech po něm požadují i potvrzení, zda mohou ukončit svou činnost. Na první pohled to vypadá jako rozumný přístup, ale ve skutečnosti je to jeden z nejhorších způsobů komunikace s uživatelem vůbec. Program by se do komunikace s uživatelem měl pouštět pouze v nejkrajnějších případech. Ideální je, když program po uživateli sám nic nechce a nechá se uživatelem kompletně ovládat – tedy uživatel má ovládat program, neměl by být sám ovládán programem.

Určitě byste měli zvážit, zda je interaktivní ovládání pro danou úlohu vůbec vhodné. Mnohdy je mnohem efektivnější několik malých, jednoduchých a jednoúčelových programů, než jedna velká aplikace.

Rada: Neustálá komunikace má jednu ohromnou nevýhodu – ovládání takového programu velmi zdržuje. Takový program se nedá spouštět automaticky pomocí skriptu (dávkového souboru). Uvažte, jak těžkopádné by bylo používání DOSového příkazu DIR, kdyby někdo napsal jeho ovládání tímto způsobem:

```
c:/> dir
Vítejte!
Zadejte cestu k adresáři, který chcete vypsat: c:/dokumenty
Výpis bude moc dlouhý, mám jej odstránkovat? [A, n]:
...
```

U jednoduchých programů je mnohem efektivnější vytvořit ovládání „zvnějšku“, pomocí přepínačů z příkazové řádky. Samotný běh programu potom není potřeba přerušovat. Navíc lze tímto způsobem udělat poměrně jednoduše mnohem širší škálu nastavení programu, než by bylo únosné prvně zmíněným způsobem. Pomocí přepínačů se dá elegantně řešit testování programu.

V jazyce C jsou argumenty příkazové řádky těsně svázány s parametry funkce `main`:

```
int main(int argc, char *argv[])
{ ... }
```

Parametr `argc` znamená počet argumentů detekovaných na příkazové řádce, a parametr `argv` je pole textových řetězců s jednotlivými argumenty. První argument (s indexem 0, `argv[0]`) vždy obsahuje jméno vlastního programu (včetně cesty v adresářové struktuře použité při spouštění – tato cesta může být buď relativní nebo absolutní). Argumenty od indexu 1 výše jsou konkrétní argumenty, které zadal uživatel.

[2] Ale zkušený narušitel si najde skulinu jinde.

7.4 Program má nejasné nebo chybné požadavky na uživatele

Hodnocení:	*** Pro uživatele velmi nepříjemné. Může vést ke ztrátě dat.

Dalším důvodem k zatracení programového díla jsou nejasná nebo chybná hlášení programu. Pokud pomineme překlepy (které ale něco o autorovi vypovídají), nejhorší jsou chybné požadavky. Když program napíše *Zadejte jméno souboru.* a přitom očekává, že zadáte adresář, je to velmi matoucí.

Dalším prohřeškem jsou nepřesné nebo nejasné požadavky. Příkladem může být následující věta z nápovědy k programu: *Za parametrem -i následuje číslo v intervalu* $(0, 256)$*.* Program přitom očekává číslo větší než 0 a menší než 256. Málokterý uživatel bude předpokládat, že autor měl na mysli otevřený interval.

Rada: Vyhýbání podobným chybám se podobá chůzi v minovém poli. Uživatel si pak určitě rozmyslí, zda takový program chce používat. Snažte se, aby program komunikoval s uživatelem přesně a srozumitelně.

7.5 Problémy s představováním

Hodnocení:	* – ** Může omezovat použitelnost programu.

Program, který se nepředstaví, je podezřelý. Na druhou stranu není rozumné vypisovat hlavičku programu společně s výslednými daty. Důvodem jsou opět dávkové soubory. Často je v nich potřeba přesměrovat výstup jednoho programu na vstup jiného. V takových případech je nutné zajistit jednotný formát výstupu a jakékoli informace navíc velmi ztěžují situaci.

Tip: Řešení tohoto problému je jednoduché. Implementujte přepínač *-h*. Po jeho aktivaci program nebude dělat nic jiného, než výpis hlavičky, která bude obsahovat název a popis programu, jméno autora, rok vytvoření a popis jednotlivých přepínačů. Jde o velmi rozšířenou konvenci, takže nemusíte mít strach, že si uživatel s takovým programem nebude vědět rady.

7.6 Spoléhání na terminál velký 80x25 znaků

Hodnocení:	* Nepříjemné pro uživatele novějších terminálů, u kterých lze měnit velikost.

Pokud vytváříte interaktivní program, neměla by data přetékat za konec obrazovky. To je sice rozumný požadavek, ale v současné době se realizuje poměrně těžko. Uživatel má totiž k dispozici terminály, u kterých si může velikost měnit téměř libovolně. Nelze se tedy jednoduše spolehnout na to, že když program vypadá dobře na terminálu 80x25 znaků, bude stejně dobře vypadat i jinde.

Pokud nepíšete interaktivní program, ale pouze vypisujete dlouhá data na standardní výstup, neměli byste si s odstránkováním dělat těžkou hlavu. Pokud si uživatel chce takto dlouhá data přečíst, může si kdykoli výstup přesměrovat do souboru a prohlédnout si je v textovém editoru. Další možností je pak použití programu `more` (nebo `less` v Linuxu):

```
c:/>dlouhyvypis.exe | more
```

Tip: Pokud přesto chcete v textovém režimu vytvářet interaktivní programy, použijte knihovnu `ncurses` nebo nějakou jinou, která si umí poradit s různě velkým terminálem.

7.7 Program čeká na akci uživatele, aniž by to dal najevo

Hodnocení:	* Nepříjemné, uživatel si může myslet, že se program zasekl.

Pro uživatele může být nepříjemné, když se program najednou zastaví a nic nedělá. Pokud program čeká na stisk nějaké klávesy, měl by o tom uživatele informovat. Pokud něco dlouho počítá, měl by o tom také podat zprávu (například může počítat procenta). Jinak může uživatel nabýt dojmu, že program zhavaroval a vypne ho nějakým hrubým způsobem (CTRL+Break, reset, vyhozením z okna…).

Rada: Vůbec se čekání na stisk klávesy vyhněte. Pouze amatéři píší do svých programů hlášení typu *Pro ukončení programu stiskněte Enter.*

7.8 Výpis netisknutelných znaků na obrazovku

Hodnocení:	* Když aplikace začne pískat a „vypisovat smetí“, je vidět, že autor na něco zapomněl.

Pokud tisknete na textovou obrazovku znaky, měli byste vědět, že znaky v ASCII tabulce s hodnotou menší než 32, jsou tzv. řídící znaky. Jsou zde znaky jako Escape, Backspace, zvonek, CR, LF, a podobně. Tyto znaky mohou při výpisu dělat různé potíže – mohou nepřípustným způsobem přemísťovat kurzor, mazat znaky, pípat. Často se těmto znakům říká *netisknutelné* znaky. Proto by se je program neměl pokoušet vypisovat přímo. Když už je to nutné, může je vypisovat například jejich číselným kódem (např. \027). Něco jiného je to samozřejmě v případě, kdy je zapisujete do binárního souboru.

Kapitola 8

Dokumentace

Dokumentace je důležitou a nedílnou součástí každého inženýrského díla. Vypovídá jak o kvalitě samotného díla, tak i o jeho autorovi (autorech). Měla by být především technicky přesná, korektní a úplná. Je naprosto nepřípustné, aby technická dokumentace obsahovala lživé nebo nepřesné údaje.

8.1 Pravopisné chyby

Hodnocení:	*** – **** Snižují kvalitu textu. Čtenář získá pochyby o kvalitě popisovaného produktu.

Pravopisné chyby nejsou hodny studenta vysoké školy. Při psaní delšího textu nelze spoléhat pouze na automatický korektor pravopisu (OpenOffice, Word, a jiné). Automatický korektor není schopen odhalit různé gramatické závislosti jako je skloňování, mužský a ženský rod a podobně, protože nerozezná význam sdělení.

Tip: Pokud vám čeština (slovenština, angličtina) činí velké problémy, nechte si svou dokumentaci opravit někým jazykově lépe vybaveným. Přistupujte k psaní dokumentace, jako byste pomocí ní chtěli získat zákazníka, nebo jako byste ji chtěli předložit při přijímacím pohovoru u zaměstnavatele jako ukázku svých schopností.

Tip: Jazykové schopnosti každého člověka jsou úměrné počtu a kvalitě knih, které přečetl. Chcete-li se umět dobře vyjadřovat, čtěte knihy psané kvalitní, krásnou češtinou (Čapek, Peroutka, Hrabal, ...).

8.2 Dlouhá souvětí a slohové chyby

Hodnocení:	*** Velmi únavné, čtenář z toho nebude mít dobrý pocit.

Dokumentace by měla dodržovat všechny slohové zásady jako kterýkoliv jiný psaný text. Jednou z nejčastějších chyb jsou příliš dlouhá a *šroubovaná* souvětí. Aby byl text srozumitelný, měl by obsahovat souvětí průměrně o dvou větách. Samozřejmě se tohoto pravidla nelze držet vždy, ale věta která tvoří celý odstavec, je většinou stěží pochopitelná.

Další chybou je časté používání stejných výrazů. Problém je, že v technickém textu se tomu často vyhnout nelze.

Tip: Pokud musíte často používat stejný výraz, je vhodné jej označit zkratkou. Například místo neustálého opakování
...genetický algoritmus...genetickém algoritmu...genetického algoritmu...
je lepší zavést při prvním použití zkratku a tu pak dále používat
...genetický algoritmus (GA)...GA...GA...

Zkratek však nesmí být příliš mnoho, protože by se text stal nesrozumitelným.

8.3 Členění na kapitoly

Hodnocení:	***
	Technický dokument bez kapitol není použitelný.

Technický text není román. Často se předpokládá, že čtenář bude číst zprostředka, pokud potřebuje pouze konkrétní informaci. Členění na kapitoly slouží k nasměrování čtenáře, proto by názvy kapitol měly co nejpřesněji (a co nejstručněji) vyjadřovat obsah textu pod nimi. Pokud například kapitola *Popis řešení* popisuje návod k použití, je to špatně.

8.4 Chybí popis principu řešení

Hodnocení:	***
	Popis principu řešení je jádrem dokumentace.

Nestačí, že princip řešení vyplývá z popisu implementace. Dokumentace k programovému projektu vyžaduje strukturovaný formát. Vůbec nevadí, když se v ní některé informace opakují vícekrát.

Zde je příklad, který popisuje princip řešení algoritmu semínkového vyplňování oblastí v grafických aplikacích: *Vstupem algoritmu je bod, který leží uprostřed vyplňované oblasti – semínko, barva, kterou se má oblast vyplnit, a nakonec barva hraniční oblasti. V zadaném bodě algoritmus začíná. Nejprve je obarven tento bod a potom se testují všechny okolní body, zda nejsou obarveny hraniční barvou. Pokud algoritmus narazí na bod, který není hraniční, postupuje stejně jako s prvním bodem – obarví jej. Pokud se algoritmus dostal k bodu, který je hraniční, nedělá nic a posune se na další bod v okolí na nějž aplikuje tentýž postup. Jelikož je algoritmus rekurzivní, skončí v posledním bodě z okolí semínka*[1].

Všimněte si, že princip řešení popisuje algoritmus obecně, kdežto když se popisuje vlastní implementace už se hovoří o konkrétních datových typech, podprogramech, modulech atd.

8.5 Použití „humoru“

Hodnocení:	***
	Rádoby vtipné vsuvky mohou navodit dojem, že řešeným problémem nebo čtenářem pohrdáte.

Technický text by měl obsahovat věcné, přesné a korektní formulace. Vtipné poznámky se do technického textu, jakožto žánru, příliš nehodí. Odvádějí pozornost od

[1] Toto je pouze příklad dokumentace. Ve skutečnosti se zde popisuje ta nejméně efektivní varianta semínkového vyplňování oblastí.

podstaty problému a mohou ve čtenáři navodit pocit, že jím nebo řešeným problémem pohrdáte.

Rada: V žádném případě vás nechci nutit být smrtelně vážní. Měli byste ovšem zvážit, zda je vhodné být přehnaně vtipný v technickém textu. Velmi tím šetřete a pokud o sobě víte, že občas neznáte míru, tak se vtipným poznámkám úplně vyhněte! V okamžiku, kdy vaše dokumentace není po obsahové stránce excelentní, je použití rádoby vtipných vsuvek naprosto nevhodné.

Holý, věcný text při čtení zpravidla nikomu nevadí, „vtipné poznámky" vadit mohou. Ne každý je musí pochopit a ne každý musí mít při čtení technické dokumentace náladu na vtipkování.

8.6 Emocionálně zabarvená sdělení

Hodnocení:	*** Obtěžují čtenáře a snižují hodnotu celého projektu.

Emocionálně zabarvená prohlášení velmi snižují hodnotu celého popisovaného díla. Technický text nemá vyvolávat emoce. Pokud se o to snaží, vyvolá ve čtenáři spíše znechucení než nadšení. Emocionálně zabarvená prohlášení čtenáři znemožňují objektivně ohodnotit kvalitu popisovaného díla.

Rada: Vyvarujte se prohlášení typu *Programování této úlohy se mi líbilo* nebo *Vytvoření tohoto programu mi nečinilo žádné problémy*, případně *Jakožto začínajícího programátora mě lepší řešení nenapadlo.*

Pokud se vychloubáte, že vám řešení nečinilo problémy, vystavujete se nebezpečí, že to bude působit směšně, zvláště pokud máte v programu chyby. Na druhou stranu, různé výmluvy že jste začínající programátor, také nepůsobí dobře. Zbytečně tím upozorňujete na své nedostatky a čtenář bude na vašem díle podvědomě hledat chyby, které tam třeba ani nemáte.

8.7 Chybí popis zadání

Hodnocení:	** Čtenář by měl vědět, o čem čte.

Z dokumentace musí být na první pohled patrné, co popisuje. Pokud není zadání příliš dlouhé, můžete je do úvodu dokumentace opsat celé. Pokud je zadání delší, je nutné z něj vybrat ty nejdůležitější body, aby bylo čtenáři jasné, co je předmětem řešení projektu, aby si to mohl porovnat s popisovaným řešením. Pokud jsou součástí zadání konkrétní technické specifikace, rozhodně se musí objevit i v dokumentaci.

8.8 Chybí návod k obsluze

Hodnocení:	** I k ledničce dostanete návod k použití.

Návod k obsluze je nezbytnou součástí dokumentace. Čtenář z něj získá představu o fungování programu, i když si jej zrovna nemůže sám vyzkoušet.

Tip: Nenechte uživatele tápat. To, že se vám zdá používání vašeho programu intuitivní, ještě neznamená, že tento názor budou sdílet ostatní. U grafických aplikací můžete sejmout obrázek okna (screenshot) a umístit jej do dokumentu. Obrázek často řekne více než několik odstavců textu.

8.9 Chybí konkrétní testovací hodnoty

Hodnocení:	* Snažte se být konkrétní a technicky přesní.

Pokud dokumentace obsahuje specifikaci testů, měla by obsahovat ukázky konkrétních vstupních hodnot a příslušný výsledek zpracovaný programem. V případě, že program zpracovává velké množství dat, je vhodné popsat ty stavy, u kterých se dají očekávat problémy. Pokud program zpracovává obrázky, je možné je do dokumentu vložit jako ukázku.

Tip: Čtenář nezíská představu o tom, jak program funguje, když mu řeknete, že si ho má vyzkoušet s daty ze souboru `proj1.dat`.

8.10 Exotický nebo nepřenosný formát

Hodnocení:	* Zbytečně odrazuje potenciální zájemce o popisovaný produkt.

V současnosti existují desítky běžně používaných publikačních systémů. Mnoho z nich je komerčních. Není možné předpokládat, že všichni čtenáři na svém počítači přečtou dokument v jakémkoli formátu. Nativní formáty jednotlivých editorů (např. doc ve Wordu, či sxw v OpenOffice) slouží především pro tisk. Tyto formáty nejsou určeny pro výměnu dokumentů (i když výrobci tvrdí opak). Často se u nich stává, že se formátování dokumentu poruší po přenosu na jiný počítač (nebo jinou verzi editoru).

Pro bezpečný a bezproblémový přenos dokumentů je lepší používat formáty, které nejsou vázány na konkrétní editor nebo prohlížeč. Z tohoto pohledu se jako nejméně problémové jeví použití formátů HTML, PDF nebo čistého postscriptu (ps).

Rada: Pokud chcete mít jistotu, že váš dokument budou schopni přečíst všichni uživatelé, použijte HTML. Není potřeba psát značky ručně, všechny běžně dostupné kancelářské balíky (MS Office, OpenOffice) jsou schopny uložit dokument jako HTML. Pokud chcete zajistit, aby uživatel po vytištění obdržel dokument přesně v takovém tvaru, v jakém jste to zamýšleli, použijte PDF nebo PostScript. Z běžně dostupných programů poskytuje největší kontrolu nad textem TEX, LATEX(nebo pdfLATEX). Pomocí těchto systémů je možné vygenerovat dokument jak ve formátu PDF, tak v PS nebo v HTML.

Formáty HTML a PDF jsou dnes na celém světě velmi rozšířené. Nemusíte mít žádné obavy, že by je někdo nemohl na svém počítači přečíst. Oba umožňují použití hypertextových odkazů a jsou tedy vhodné pro šíření elektronickou formou. Formát PDF navíc zaručuje vysokou věrnost a zachování sazby dokumentů po vytištění.

Kapitola 9

Na závěr

Věřím, že tento přehled převážně začátečnických omylů a jejich řešení, bude užitečný. Je ovšem potřeba, abyste si popisované problémy sami promysleli a odzkoušeli si správná řešení. Pouhé čtení příruček má při získávání programátorských dovedností omezenou účinnost, protože praktická zkušenost je nenahraditelná.

Nakonec mi nezbývá, než vám popřát hodně úspěchů při řešení všech projektů, které jsou před vámi a co nejméně neodhalených chyb.

David Martinek, 21. října 2011.

Literatura

[ccm04] ccmalloc home page.
http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/, 2004.

Knihovna pro práci s pamětí, která umožňuje hledat paměťové úniky (memory leaks).

[cfa04] C FAQ Index. http://www.faqs.org/faqs/C-faq/, 2004.

Často kladené otázky o jazyce C. Ačkoli je zde popisována starší verze jazyka C, jsou tyto stránky plné užitečných informací.

[cst04] C Style.
http://www.teamten.com/lawrence/style/,
http://www.doc.ic.ac.uk/lab/secondyear/cstyle/cstyle.html,
http://www.psgd.org/paul/docs/cstyle/cstyle.htm,
http://www.chris-lott.org/resources/cstyle/, 2004.

Stránky zabývající se vhodným stylem odsazování v jazyce C.

[dma04] Dmalloc – Debug Malloc Home Page. http://dmalloc.com/, 2004.

Dmalloc je knihovna, která nahrazuje standardní funkce pro práci s pamětí a poskytuje tak možnost lepšího ladění při alokaci/dealokaci.

[dox04] Doxygen. http://www.doxygen.org, 2004.

Doxygen je nástroj pro generování programátorské dokumentace z komentářů ve zdrojovém textu.

[ele04] Electric Fence. http://perens.com/FreeSoftware/, 2004.

Knihovna pro práci s pamětí, která umožňuje hledat paměťové úniky (memory leaks).

[gli04] GNU C Library. http://www.gnu.org/software/libc/libc.html, 2004.

GNU implementace standardní knihovny jazyka C. Stránky obsahují rozsáhlý manuál k systémovým funkcím. Knihovna je součástí téměř všech distribucí Linuxu, takže manuál je dostupný v podobě info stránek (`info libc`).

[Her03] Pavel Herout. *Začínáme v jazyce C*, volume III. upravené vydání. Nakladatelství Kopp, České Budějovice, 2003.

Tato kniha je jednou z nejlepších učebnic jazyka C. Oceníte ji, i když přecházíte z Pascalu.

[ind04] Indent. http://www.gnu.org/software/indent/indent.html, 2004.

Program pro automatické odsazování zdrojového textu podle vybraného stylu.

[Kad02] Václav Kadlec. *Učíme se programovat v jazyce C.* Computer Press, Praha, 2002.

Zabývá se nejenom programováním v jazyce C, ale obsahuje i úvod pro čtenáře, kteří nikdy neprogramovali.

[Pat02] Ron Patton. *Testování softwaru*. Computer Press, Praha, 2002.

Zabývá se základními principy testování a návrhem testovacích aplikací. Velice užitečné, pokud to s programováním myslíte vážně.

[Töp95] Pavel Töpfer. *Algoritmy a programovací techniky*. PROMETHEUS, Praha, 1995.

Učebnice základů programování. Příklady jsou sice psány v Pascalu, ale samotné algoritmy jsou na konkrétním programovacím jazyce nezávislé. Pokud budete znát základní algoritmy, ušetříte si mnoho práce a námahy.

[val04] Valgrind – článek na Root.cz. `http://www.root.cz/clanek/1635`, 2004.

Český návod k programu `valgrind`, který umožňuje hledat paměťové úniky, přístupy do nealokované paměti, používání neinicializovaných proměnných a další chyby při práci s pamětí.