

Kniha velmi těsně navazuje na **Učebnici jazyka C** od stejného autora a stejného nakladatelství. Obsahuje řadu informací, návodů a zkušeností, které ocení nejspíše programátoři, kteří jazyk C používají již víceméně suverénně.

Stěžejní částí knihy tvoří detailní rozbor všech možných činností se soubory a pak také popisy standardních knihoven. Proto je vhodná pro ty, programátory, kteří občas narazí na něco, co by potřebovali vědět, ne jen tušit, případně nemají čas bádát nad něčím, co je již prozkoumáno a popsáno.

Ke knížce si můžete zdarma stáhnout asi 350 kompletních odladěných programů. Jedná se o zdrojové soubory všech příkladů z knihy.

Autor (*1961) pracuje na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Zabývá se výukou softwarově orientovaných předmětů. Značné praktické zkušenosti s jazykem C získal jako spoluvůrce několika velkých projektů. Je autorem nebo spoluautorem několika skript a řady odborných knih, jejichž obliba je připočítána i tomu, že je v nich uvedeno něco víc, než bývá v manuálech – osobní praktické zkušenosti.



nakladatelství

Šumavská 3, 370 01 České Budějovice

Tel./fax: 386 460 474

e-mail: knihy@kopp.cz; Internet: www.kopp.cz

ISBN 978-80-7232-329-6



9 788072 323296

Pavel Herout

Učebnice jazyka C – 2. díl



UČEBNICE JAZYKA

C

2.díl



Pavel Herout



Daně, Kateřině a Tomášovi

Učebnice jazyka C

2. díl

Pavel Herout

 nakladatelství
České Budějovice, 2007

Microsoft je registrovaná ochranná známka *Microsoft Corporation*.

UNIX je registrovaná ochranná známka *AT&T*.

Jiné případné názvy mohou být ochranné známky
nebo registrované obchodní známky svých případných vlastníků.

Lektoři: Ing. Přemysl Brada, Ph.D. a Ing. Petr Grillinger

© Doc. Ing. Pavel Herout, Ph.D., 2004–2007

ISBN 978-80-7232-329-6

Obsah druhého dílu

Pozor:

- Číslování knihy je společné pro první i druhý díl Učebnice jazyka C (podrobnosti viz v předmluvě). To tedy znamená, že druhý díl začíná kapitolou 17 a stranou 272. Všechny strany obsahu a předmluvy jsou číslovány římskými číslicemi.

Předmluva k aktuálnímu vydání	vi
17 Komplexní pohled na souborový vstup a výstup	272
17.1 Rozdíly mezi binárním a textovým souborem obecně	273
17.1.1 Binární soubory	273
17.1.2 Textové soubory	274
17.2 Rozdíly mezi binárním a textovým režimem otevírání souboru	274
17.3 Otevření souboru	277
17.3.1 Různé způsoby otevírání souboru	278
17.4 Uzavření souboru	281
17.5 Ošetření chyb	283
17.6 Čtení a zápis dat	285
17.6.1 Formátované vstupy a výstupy	286
17.6.2 Neformátované vstupy a výstupy	286
Zpracování jednoho znaku v jeden okamžik	287
Zpracování celé řádky najednou	290
Zpracování celého bloku dat najednou	292
17.7 Bufferování	293
17.7.1 Základní informace	294
17.7.2 Možnosti bufferování poskytnuté ANSI C	295
17.7.3 Funkce pro práci s bufferem	295
Funkce <code>setbuf()</code>	295
Funkce <code>setvbuf()</code>	296
17.7.4 Řešení problému nebufferovaného <code>stdin</code>	297
17.8 Funkce pro zápis bufferu na disk	299
17.8.1 Funkce <code>fflush()</code>	299

17.9	Přímý přístup do souboru	300
17.9.1	Funkce <code>fseek()</code> – posun v souboru	300
17.9.2	Funkce <code>ftell()</code> – zjištění pozice v souboru	300
17.9.3	Typické použití přímého přístupu	302
17.10	Zbývající užitečné funkce ze <code><stdio.h></code>	306
17.10.1	Funkce <code>freopen()</code> – přesměrování proudu	306
17.10.2	Funkce <code>rename()</code> – přejmenování souboru	308
17.10.3	Funkce <code>remove()</code> – zrušení souboru	308
17.10.4	Funkce <code>tmpfile()</code> – pomocný soubor	308
17.10.5	Funkce <code>tmpnam()</code> – jméno pomocného souboru	308
17.11	Ukázka použití funkcí z UNIXové knihovny	309
17.12	Zjištění informací o položkách v adresáři	311
17.12.1	Prostředí UNIXu	311
	Zjištění pouze jmen položek	312
	Zjištění typu a stavu položek	313
17.12.2	Prostředí Windows	316
18	Návaznost překladače ANSI C na okolní prostředí	319
18.1	Standardní hlavičkové soubory	319
18.2	Rezervovaná jména	321
18.3	Standardní hlavičkové soubory bez funkčních prototypů	322
18.3.1	Soubor <code><errno.h></code>	322
18.3.2	Soubor <code><limits.h></code>	323
18.3.3	Soubor <code><float.h></code>	323
18.3.4	Soubor <code><stddef.h></code>	324
18.4	Předdefinovaná makra preprocesoru	325
18.4.1	Makro <code>_STDC_</code>	325
18.4.2	Makro <code>_FILE_</code>	325
18.4.3	Makro <code>_LINE_</code>	325
18.4.4	Makro <code>_DATE_</code>	325
18.4.5	Makro <code>_TIME_</code>	326
18.5	Externí proměnné	326
18.5.1	Proměnná <code>errno</code> z <code><errno.h></code>	326
18.5.2	Proměnná <code>sys_errlist</code> ze <code><stdlib.h></code>	327
18.5.3	Proměnná <code>sys_nerr</code> ze <code><stdlib.h></code>	327
18.5.4	Proměnná <code>environ</code> ze <code><stdlib.h></code>	328
18.5.5	Proměnná <code>_iob</code> ze <code><stdio.h></code>	328
19	Popis funkcí a maker ze standardních knihoven	329
19.1	<code><assert.h></code> – makro používané při ladění	329
19.2	<code><ctype.h></code> – práce se znaky	329

19.3	<code><math.h></code> – matematické funkce v přesnosti <code>double</code>	330
19.4	<code><locale.h></code> – přizpůsobení C národnímu prostředí	331
19.5	<code><setjmp.h></code> – umožnění běžně nedovolených skoků	332
19.6	<code><signal.h></code> – zpracování signálů	334
19.7	<code><stdarg.h></code> – práce s proměnným počtem parametrů	337
19.8	<code><stdio.h></code> – funkce pro vstup a výstup	337
19.9	<code><stdlib.h></code> – obecně užitečné funkce	339
19.9.1	Konverze řetězců na čísla	339
	Funkce pro jednoduchý převod desítkových čísel	339
	Funkce pro náročný převod čísel	339
19.9.2	Generátor pseudonáhodných čísel	340
19.9.3	Funkce pracující s dynamickou pamětí	341
19.9.4	Funkce pro spolupráci s operačním systémem	341
	Funkce <code>abort()</code>	341
	Funkce <code>atexit()</code>	341
	Funkce <code>exit()</code>	342
	Funkce <code>system()</code>	343
19.9.5	Funkce pro hledání a řazení	343
	Řazení algoritmem <i>quick-sort</i>	343
	Binární vyhledávání v seřazeném poli	344
19.10	<code><string.h></code> – zpracování řetězců	348
19.10.1	Funkce <code>strspn()</code>	348
19.10.2	Funkce <code>strcspn()</code>	349
19.10.3	Funkce <code>strpbrk()</code>	349
19.10.4	Funkce <code>strtok()</code>	349
19.10.5	Funkce <code>memchr()</code>	350
19.10.6	Funkce <code>memcmp()</code>	350
19.10.7	Funkce <code>memcpy()</code>	351
19.10.8	Funkce <code>memmove()</code>	351
19.10.9	Funkce <code>memset()</code>	351
19.11	<code><time.h></code> – práce s datem a časem	353
19.11.1	Funkce <code>clock()</code>	353
19.11.2	Měření doby běhu programu	354
19.11.3	Funkce <code>time()</code>	356
19.11.4	Funkce <code>difftime()</code>	356
19.11.5	Funkce <code>localtime()</code>	356
19.11.6	Funkce <code>asctime()</code>	357
19.11.7	Funkce <code>ctime()</code>	357
19.11.8	Funkce <code>strftime()</code>	358
19.11.9	Funkce <code>mktime()</code>	359

20	Funkce s proměnným počtem parametrů	362
20.1	Úvodní informace	362
20.2	Základní princip	363
20.3	Standardní hlavičkový soubor <code><stdarg.h></code>	365
20.4	Funkce <code>vprintf()</code> , <code>vfprintf()</code> , <code>vsprintf()</code>	366
20.5	Praktické příklady použití FPPP	367
20.5.1	Typické použití <code>vprintf()</code> , <code>vfprintf()</code> , <code>vsprintf()</code>	367
20.5.2	Přesměrování výstupu za běhu programu	368
20.5.3	Seznam parametrů je ukončený domluvenou hodnotou	370
20.5.4	Počet parametrů je předán v pevném parametru	372
20.5.5	Odstrašující případ zcela netypického použití	373
21	Ladění v jazyce C	376
21.1	Ladicí výpisy na <code>stderr</code>	378
21.2	Využití podmíněného překladu pomocí příkazů C-preprocesoru	378
21.2.1	Využití existence symbolické konstanty	379
21.2.2	Využití hodnoty symbolické konstanty	380
21.3	Self-testy modulů nebo funkcí	383
21.4	Využití makra ze souboru <code><assert.h></code>	385
21.4.1	Praktické použití	386
21.4.2	Zrušení aserce	389
21.4.3	Princip aserce	389
21.4.4	Chybné použití aserce	390
21.5	Prostředky pro ladění dynamických procesů	392
22	Minimální znalosti při programování pod UNIXem	396
22.1	Editor	396
22.2	Překladač a sestavovací program	397
22.3	Kontrola zdrojových textů v C	399
22.4	Odděleně překládané moduly	401
22.5	Ladicí prostředky	404
23	Změny v C podle nového standardu ISO	406
23.1	Stručný přehled norem a nejdůležitějších změn	406
23.2	Milé drobnosti	407
23.2.1	Komentáře do konce řádky	407
23.2.2	Definice kdekoliv v kódu	408
23.2.3	Co bylo zrušeno	408

23.3	Celočíselné datové typy	409
23.3.1	Logické hodnoty	409
23.3.2	Typ <code>long long int</code>	410
23.3.3	Celočíselné typy s přesně danou šířkou	410
23.3.4	Nové formátové specifikace	412
23.4	Okrajové hodnoty reálných čísel	413
23.4.1	Nekonečno	413
23.4.2	NaN	413
23.4.3	Denormální čísla	413
23.4.4	Podpora okrajových hodnot reálných čísel	413
23.5	Pole	414
23.5.1	Inicializace polí	414
23.5.2	Proměnná délka pole	415
23.5.3	Minimální velikost pole jako parametr funkce	416
23.5.4	Složené literály	416
23.6	Funkce	416
23.6.1	Identifikátor funkce	417
23.6.2	Funkce mohou být vnořené	418
23.6.3	Modifikátor <code>inline</code>	419
23.7	Vícebajtové znaky	420
23.7.1	Kódování a definice proměnných	421
23.7.2	Práce s jednotlivými znaky	421
23.7.3	Práce s řetězci	422
23.7.4	Vstupy a výstupy	423
24	Desatero příkázání pro programátory v C	425
	Literatura	430
	Rejstřík	431

Předmluva k aktuálnímu vydání

Po jedenácti letech od prvního vydání prvního dílu Učebnice jazyka C a po devíti letech od prvního vydání druhého dílu bylo po jednání s nakladatelem rozhodnuto, že oba díly projdou důkladnou revizí obsahu. Cílem bylo aktualizovat obě knihy tak, aby zahrnovaly aktuální poznatky, včetně věcí, které jsem až postupem času během výuky a nejrozličnějších školení tohoto jazyka začal považovat za důležité.

Aktualizace prvního dílu Učebnice jazyka C proběhla víceméně bezproblémově. U druhého dílu (tj. této knihy) však byla situace zcela odlišná. Tento díl více než polovinou svého rozsahu popisoval návaznost jazyka C na – v roce 1994 nejrozšířenější operační systém – MS-DOS. Po všeobecném přechodu na operační systémy typu Windows tyto informace samozřejmě zcela zastaraly, narozdíl od informací uvedených v první části druhého dílu. Tím vznikla krajně nevhodná situace, kdy si čtenář, který potřeboval informace uvedené v druhém dílu, musel koupit knihu, jejíž zhruba polovina mu již k ničemu nebyla.

Proto jsem všechny kapitoly týkající se práce s MS-DOSEm vypustil a ostatní důkladně zrevidoval. Navíc jsem přidal jednu kapitolu, ve které jsem se pokusil popsat alespoň část novinek, které přináší dosud poslední norma jazyka C.

Druhý díl by vám měl poskytnout ucelený objem speciálních, méně využívaných, ale důležitých informací o jazyce C, vhodný pro středně pokročilé programátory.

Původním záměrem bylo sloučit oba dva díly do jednoho a tak vytvořit učebnici, která pokryje celou problematiku jazyka C. Postupem času a po mnoha jednáních s nakladatelstvím však byla tato myšlenka opuštěna. Vedly k tomu následující důvody:

- První díl je svým rozsahem „rozumně“ velká kniha, která pro běžnou výuku postačí. Pro nemalou část studentů, kteří se potřebují jazyk C naučit, ale nepočítají s tím, že by v něm intenzivněji programovali, představuje dostatečné množství informace.
- Ve druhém dílu jsou informace pro ty, kteří budou jazyk C intenzivněji používat, buď při své práci nebo při dalším studiu. Tito lidé postupem

času s velkou pravděpodobností využijí většinu zde uvedených informací. Tyto informace však dříve zmíněné skupině studentů mohou z prvního pohledu připadat jako zbytečné a bylo by zbytečné je nutit, aby za tyto informace jednak zbytečně platili a jednak zbytečně „tahali“ mnohem objemnější knihu.

Závěrečné rozhodnutí tedy bylo ponechat oba dva díly s tím, že ten druhý si koupí až vážný zájemce o jazyk C.

Protože však oba dva díly spolu úzce souvisejí, došlo k jedné podstatné změně. Kapitoly obou dílů jsou spojitě číslovány, takže první díl končí kapitolou číslo 16 a druhý díl začíná kapitolou číslo 17. Vzájemná provázanost je umocněna i pokračujícím číslováním stran, kdy druhý díl začíná poněkud netradičně stranou číslo 272 a obsah a předmluva jsou číslovány římskými číslicemi.

Průběžné číslování umožňuje vytvořit jednotný obsah, rejstřík a významně zjednodušit křížové odkazy, kterých je v obou dílech velmi mnoho. Oba dva díly tak obsahují úplný rejstřík, samozřejmě vždy s informací, že v příslušném díle jsou dosažitelné jen určité strany. Na toto je třeba pamatovat i v případě křížových odkazů, kdy se ve druhém díle občas setkáte s odkazem na zde neexistující stranu. Znamená to samozřejmě odkaz na existující stranu, ale v prvním díle.

Čtenář vlastnící oba díly tak má možnost pracovat jakoby s jednou knihou, pouze z technických důvodů rozdělenou do dvou svazků.

Další systémovou změnou, kterou jako čtenáři jistě oceníte, je, že zdrojové texty všech zde zmiňovaných programů již nejsou šířeny na disketách, ale můžete si je kdykoliv a zdarma stáhnout z www.kopp.cz. Pokud to učíte, věnujte, prosím, pozornost souboru `READ.ME`, který je nedílnou součástí balíku programů.

Rád bych poděkoval Ing. Přemku Bradovi, MSc., Ing. Petru Grillingerovi a Ing. Miroslavu Viriusovi, CSc. za jejich přínosné poznámky k revidovaným a doplňovaným částem. Výrazně tak přispěli k jejich zkvalitnění.

Pokud se stane, že přes všechnu jejich i moji snahu objevíte nějakou chybu, napište mi prosím na herout@kiv.zcu.cz.

Pavel Herout

17 Komplexní pohled na souborový vstup a výstup

Standardní knihovna ANSI C pro vstupy a výstupy byla vytvářena ze dvou zdrojů:

- 1) bufferované (viz str. 294) vstupy a výstupy z K&R knihovny,
 - 2) nebufferované vstupy a výstupy z knihovny UNIX System V,
- kdy nebufferované vstupy a výstupy ANSI C pozměňuje na bufferované, se zachováním většiny jejich dalších výhod.

Mnohé implementace ovšem zachovávají i UNIXovou vstupně-výstupní (též „V/V“) knihovnu jako své rozšíření. Ta je nejčastěji popsána v souborech `<io.h>` a `<fcntl.h>` (viz str. 309).

Při zachování možnosti používat obě knihovny je přístup k souboru možný dvěma – z prvního pohledu nepřilíš rozdílnými – způsoby:

- 1) Funkce z ANSI C pracují se souborem pouze pomocí *proudu* (*stream*). Ten je představován ukazatelem na strukturu typu `FILE`, definovanou v `<stdio.h>`¹.
- 2) V rozšířené C knihovně existují kromě těchto funkcí navíc funkce, které pracují se souborem pomocí *handle* nebo *filehandle*, což je hodnota typu `int`. Česká terminologie je: *symbolické číslo souboru*, *číslo kanálu*, *logické číslo kanálu*, *logické číslo zařízení*.

Poznámky:

- Toto základní rozlišení bylo na úvod pro lepší pochopení vztahů a pojmů. Protože tato část pojednává o ANSI C, budou další informace výhradně o bufferovaných vstupech a výstupech, přístupných pomocí ukazatele na `FILE`.
- Všechny potřebné informace pro práci s těmito soubory jsou uvedeny v souboru `<stdio.h>`.
- Ukázka práce s funkcemi z UNIXové knihovny bude uvedena na str. 309.

¹ Viz též str. 277 a str. 328.

17.1 Rozdíly mezi binárním a textovým souborem obecně

17.1.1 Binární soubory

Binární soubory nejsnáze poznáme podle toho, že při jejich otevření libovolným textovým editorem nevidíme žádnou smysluplnou informaci (jako je tomu u textových souborů), ale „chaotickou zmrč“ dat. Toto je samozřejmě jejich nevýhoda, protože binární soubor může být vnitřně libovolným způsobem organizován a při zběžném pohledu na jeho obsah tuto organizaci nelze určit, tj. nevíme, co máme vlastně číst.

Další nevýhodou je, že pomocí běžně používaných editorů nejdou binární soubory nijak vytvářet či opravovat.

Přesto se binární soubory v profesionálních programech využívají poměrně často. Nejvýhodnější je jejich použití pro ukládání rozměrných dat – velkých polí, struktur, atd. U textového souboru je totiž pro uchování stejného množství informace potřeba mnohem více prostoru. Například číslo 65535 zabere v textovém souboru prostor pěti bajtů, zatímco v binárním souboru třeba jen 2 bajty².

Druhou výhodou binárních souborů je, že se s nimi pracuje mnohem rychleji než s textovými soubory. Důvody jsou dva – jednak jsou binární soubory pro stejný objem informace často kratší a jednak při zápisu čísla do textového souboru je nutné provést jeho konverzi z vnitřní reprezentace čísla v počítači na textovou podobu, což je časově náročné³. Tyto konverze v binárních souborech odpadají, protože se do nich zapisuje přímo obsah paměti po bajtech.

Pro pouhé uložení znaků nemají příliš význam, protože znak v textovém souboru zabírá jeden bajt stejně jako v binárním.

Poznámka:

- Binární soubory musejí mít pod různými operačními systémy naprosto stejný obsah, kdežto textové soubory ne – vysvětlení viz dále.

Pozor:

- Mají-li binární soubory stejný obsah pod různými operačními systémy, neznamená to ještě, že jsou tyto datové soubory 100% přenositelné, budeme-

² Záleží na vnitřní implementaci čísla – zde pro `sizeof(int) == 2`.

³ U čtení dochází také ke konverzi, ale k obrácené – z textu na číslo.

li je číst stejným programem. Zde sehrává svoji roli rozdílná délka základních (zejména celočíselných) datových typů. Zapišeme-li pod Windows do binárního souboru deset čísel typu `int`, bude mít tento soubor pravděpodobně délku 20 bajtů, protože zde `sizeof(int) == 2`. Pokusíme-li se tento soubor přečíst pod UNIXem, podaří se nám pravděpodobně naplnit jen pět proměnných typu `int`, ovšem zcela odlišnými hodnotami, protože je zde často `sizeof(int) == 4`.

Z výše uvedeného vyplývá, že 100% přenositelné jsou jen binární soubory obsahující výhradně data typu `char`.⁴

17.1.2 Textové soubory

Textové soubory mají jednu velkou výhodu – je možné si jejich obsah kdykoliv prohlédnout, vytvořit nebo opravit běžným textovým editorem.

Z toho také plyne další výhoda použitelná pro jejich zpracování – i když neznáme jejich vnitřní organizaci, prostým pohledem na obsah textového souboru lze určit, že některé údaje jsou celočíselné, jiné reálné apod. To tedy znamená, že i bez detailní znalosti textového souboru jej lze správně přečíst.

Navíc se můžeme spolehnout na to, že textový soubor je definován jako soubor organizovaný po řádcích, kde je každá řádka z hlediska jazyka C ukončena znakem `'\n'` – ASCII kód 0x0A.

Tato organizace platí v UNIXu, ale při pokusu o hexadecimální výpis textového souboru ve Windows lehce zjistíme, že každému znaku `'\n'` předchází znak `'\r'` – ASCII kód 0x0D.

Toto je zřejmá nevýhoda textových souborů, protože při jejich přenosu mezi různými platformami musíme dodržovat určité postupy, aby byl textový soubor na jiné platformě správně čitelný. Typicky při přenosu textových souborů z UNIXu do Windows musíme zvolit takový režim přenosu (ASCII), který zajišťuje dodatečné vložení znaku `'\r'` před každý znak `'\n'`. Naopak při přenosu z Windows do UNIXu stejný režim způsobí vymazání všech znaků `'\r'`.

Poznámka:

- Tato fakta platí i bez jakékoliv souvislosti s jazykem C!

⁴ S dosti velkou pravděpodobností by měla být 100% přenositelnost i pro reálné typy `float`, `double` a `long double`, protože většinou dodržují doporučenou konvenci o délce a způsobu uložení čísla. Jediné, kde zde může nastat problém, je pořadí jednotlivých bajtů v souboru.

17.2 Rozdíly mezi binárním a textovým režimem otevírání souboru v C

Hned zpočátku je nutné jednoznačně říci, že stav, kdy v souborovém systému existují textové a binární soubory, musí jazyk C respektovat, protože je to daný fakt. Slovo „respektovat“ znamená, že s nimi musí umět pracovat, tj. číst je a/nebo je vytvářet.

Základní problém, který je nutné si uvědomit, je, že zatímco my rozlišujeme textové a binární soubory, ANSI C rozlišuje textový a binární **režim otevíření** souboru. Tento režim je určen druhým parametrem funkce `fopen()`.

Rozdíl mezi „souborem“ a „režimem otevíření souboru“ může vypadat jako pouhá hra se slůvky. A také to je pravda, pokud textové soubory budeme otevírat v textovém režimu a binární soubory v binárním. Je třeba si ale uvědomit, že nám nic nebrání v tom, otevřít textový soubor v binárním režimu nebo binární v textovém. To první má někdy praktické opodstatnění, např. při komprimování (zazipování), to druhé je v naprosté většině případů chyba.

Poznámka:

- Rozlišení „binární režim otevíření“ a „textový režim otevíření“ nemá v prostředí UNIXu opodstatnění, protože tam není mezi těmito dvěma režimy z hlediska způsobu zpracování rozdíl. To ovšem neznamená, že UNIX nezná pojem binární a textový soubor, jen se k nim chová stejně.

U souborů otevřených v binárním režimu je stanoveno, že knihovní funkce neprovádějí žádnou dodatečnou akci s přečtenými nebo zapisovanými bajty. To znamená, že každý bajt souboru představuje pouze sám sebe a jakékoliv výpisové nebo čtecí funkce ke čtenému nebo zapisovanému bajtu nic nepřidávají ani neubírají.

U souborů otevřených v textovém režimu je ale dilema, jak zacházet s odlišnými konci řádek. Když by totiž byl vytvářen program pro UNIX, znamenal by konec řádky práci pouze se znakem `'\n'`. Pak by ale program napsaný pod Windows musel s koncem řádky zacházet jinak, protože ten vyžaduje dvojici znaků `'\r'\n'`.

Naštěstí je tento problém vyřešen, protože když otevřeme soubor v textovém režimu, budou čtecí funkce automaticky každý znak `'\r'` bez náhrady ubírat (požírat) a naopak, při zápisu bude před každý zapisovaný znak `'\n'` automaticky přidán znak `'\r'`.

Poznámka:

- Zde se jaksi automaticky předpokládá, že v textovém režimu otevřeme pouze textový soubor. Otevřeme-li tentýž soubor v binárním režimu, pak tytéž čtecí nebo zapisovací funkce nebudou provádět navíc vůbec nic.

Pozor:

- Někdo by si mohl říci, že s binárním souborem pracuje jen pomocí funkcí `fread()` a `fwrite()` (podrobně viz str. 292), které pracují s blokem bajtů a nějaké dotatečné akce se znakem `'\r'` se jej tedy netýkají. Není to ale pravda! Pokud budou tyto funkce pracovat se souborem otevřeným v textovém režimu, budou i ony přidávat (`fwrite()`) nebo ubírat (`fread()`) tento znak.

Z následující tabulky je patrné, jaký rozdíl je ve Windows při práci v binárním a textovém režimu otevření.

- Při zápisu v textovém režimu je automaticky před každý znak `'\n'` (0x0A) vložen znak `'\r'` (0x0D).
- Naopak, při čtení jsou všechny znaky `'\r'` vyřazovány, bohužel nehledě na to, zda jsou následovány znakem `'\n'` či nikoliv. To ovšem záleží také na typu překladače, kdy některé vynechávají skutečně jen znaky `'\r'` bezprostředně před `'\n'`.

zapisovaná data:	61 0A 62 0D 63 0A 0D 64
1) zápis při otevření v režimu "w"	
skutečný obsah souboru:	61 0D 0A 62 0D 63 0D 0A 0D 64
vypsáno při otevření v režimu:	
a) "r":	61 0A 62 63 0A 64
b) "rb":	61 0D 0A 62 0D 63 0D 0A 0D 64
2) zápis při otevření v režimu "wb"	
skutečný obsah souboru:	61 0A 62 0D 63 0A 0D 64
vypsáno při otevření v režimu:	
a) "r":	61 0A 62 63 0A 64
b) "rb":	61 0A 62 0D 63 0A 0D 64

Poznámky:

- Otevření: `"w"`, `"r"`, `"wb"`, `"rb"`
představuje jednotlivé režimy funkce `fopen()` – podrobně viz str. 278.
- Pro tyto pokusy byl použit následující program:

```
#include <stdio.h>
int main(void)
{
```

```
FILE *fr, *fw;
char rezim[3];
int c;

printf("Zadej rezim zapisu: ");
scanf("%2s", rezim);
fw = fopen("A.DAT", rezim);
putc('a', fw); putc('\n', fw); putc('b', fw); putc('\r', fw);
putc('c', fw); putc('\n', fw); putc('\r', fw); putc('d', fw);
fclose(fw);

printf("Zadej rezim cteni: ");
scanf("%2s", rezim);
fr = fopen("A.DAT", rezim);
while ((c = getc(fr)) != EOF)
    printf("%02X ", c);
fclose(fr);
return 0;
}
```

17.3 Otevření souboru

ANSI C nedělá rozdíly mezi zařízením⁵ a souborem – k oběma se přistupuje pomocí proudu (*stream*).

Abychom mohli se souborem nebo zařízením pracovat, je nutné spojit jeho jméno s proudem – to znamená otevřít soubor nebo zařízení.

Poznámky:

- Otvírat není nutné jen `stdin`, `stdout` a `stderr`, které jsou otevřeny v **textovém režimu** automaticky před spuštěním programu. Proud `stdin`, `stdout` a `stderr` lze ale jednoduše uzavřít (viz str. 281) a tato operace je v naprosté většině případů nevratná, tj. opětovné otevření není možné.
- Před otevřením souboru musíme definovat ukazatel na strukturu typu `FILE`. Ta je definovaná ve `<stdio.h>` a její obsah je implementačně závislý. Obecně obsahuje informace o konkrétním souboru, jako jsou ukazatel na aktuální pozici v souboru, přístupový mód (*read*, *write*), příznak konce souboru, ...

⁵ Například obrazovka nebo tiskárna.

Vlastní otevření se pak provede voláním funkce `fopen()`⁶, jejímž parametrem je jméno konkrétního souboru na disku. Díky němu `fopen()` získá potřebné údaje pro aktualizaci položek struktury `FILE`, na kterou nakonec vrácí ukazatel. Se souborem pak dále pracujeme jen pomocí tohoto ukazatele čtení dat, zápis dat, uzavření, ...

Pomocí jedné z položek struktury `FILE` – ukazatele na aktuální pozici kurzoru v souboru – systém sám ví, odkud má číst další data, nebo kam je má zapisovat, a automaticky tento ukazatel zvětšuje.

My tento ukazatel můžeme měnit. To lze provést buď přímo, tím, že ze `<stdio.h>` zjistíme jeho jméno. To se ale skutečně nedoporučuje. Druhá, mnohem bezpečnější a používanější možnost, je použít knihovní funkci `fseek()` (viz str. 300).

Pozor:

- Nedá se zaměňovat ukazatel na typ (`FILE *`), který ukazuje na celou strukturu, a ukazatel na aktuální pozici kurzoru v souboru, což je jedna položka této struktury. Inkrementací ukazatele na `FILE` se přesuneme na další otevřený soubor v pořadí odkazovaný z pole těchto struktur.

17.3.1 Různé způsoby otevírání souboru

O tom, zda se bude se souborem zacházet jako s textovým nebo binárním (bez ohledu na jeho skutečný obsah), rozhodneme při jeho otevření pomocí druhého parametru funkce `fopen()`.

```
FILE *fopen(char *jmeno_souboru, char *rezim);
```

Pokud se soubor nepodaří otevřít⁷, vrací `fopen()` hodnotu `NULL`. V případě úspěchu vrací ukazatel na strukturu `FILE`, která je naplněna aktuálními daty otevíraného souboru.

Druhý parametr `fopen()` – `rezim` – může nabýt celkem 12 hodnot – šesti pro textové soubory a šesti pro binární soubory. Všechna dvanáct způsobů se od sebe odlišuje.

Režim pro binární soubory se liší od režimu textových souborů jen přidáním znakem `'b'`, např.:

- `"r"` – je textový režim otevření,
- `"rb"` – je binární režim otevření.

⁶ Nejčastěji, ale není to podmínka – viz str. 306.

⁷ Neexistuje, je již příliš mnoho otevřených souborů, je již otevřen, ...

Základní režimy jsou:

- `"r"` Otevření **existujícího** souboru pro čtení od začátku souboru. Pokud soubor neexistuje, je hlášena chyba – vrácen `NULL`.
- `"w"` Vytvoření **nového** souboru pro zápis od začátku souboru. Pokud soubor již existuje, je zkrácen na nulovou délku – vymazán.
- `"a"` Otevření souboru pro **připisování** na jeho konec. Pokud soubor neexistuje, je vytvořen.

Rozšířené režimy jsou:

- `"r+"` Otevření **existujícího** souboru pro čtení a zápis. Neexistuje-li soubor, je hlášena chyba – vrácen `NULL`. V případě úspěšného otevření je ukazatel aktuální pozice v souboru nastaven na začátek souboru.
- `"w+"` Vytvoření **nového** souboru pro čtení a zápis. Pokud již soubor existuje, je zkrácen na nulovou délku – vymazán.
- `"a+"` Otevření existujícího souboru pro čtení a připisování na konec. Pokud soubor neexistuje, je vytvořen. Pozor na skutečnost, že číst se dá v souboru odkudkoliv, ale zapisovat pouze na konec souboru.

Možnosti jednotlivých režimů souhrnně ukazuje tabulka:

požadavek / režim otevření	"r"	"w"	"a"	"r+"	"w+"	"a+"
soubor musí existovat	*			*		
existující soubor bude vymazán		*			*	
existující soubor bude rozšiřován			*			*
neexistující soubor bude založen		*	*		*	*
data lze odkudkoliv číst	*		*	*	*	*
data lze kamkoliv zapisovat		*		*	*	
data lze zapisovat pouze na konec			*			*

Poznámky k tabulce:

- Přidáme-li ke každé položce *režimů otevření souboru* znak `'b'`, pak bude tabulka platit i pro binární soubory.
- Výrazy „odkudkoliv“ a „kamkoliv“ znamenají, že před požadovanou akcí je nutno použít funkci `fseek()`. Bez jejího použití se čte/zapisuje na aktuální pozici v souboru.

Pozor:

Pracujeme-li s režimem "a", pak je nutné si zapamatovat, že na konec souboru se zapisuje vždy, i když nastavíme ukazatel na aktuální pozici v souboru jinak, např. na začátek. To bude demonstrováno následujícím příkladem, kdy se pokusíme text „TOTO JE PRVNI TEST“ v souboru POKUS.TXT přepsat textem „toto je druhý test“.⁸ Když po skončení programu soubor POKUS.TXT vypíšeme, zjistíme, že jeho obsah je:

TOTO JE PRVNI TESTtoto je druhý test

```
#include <stdio.h>
int main(void)
{
    FILE *fw;

    fw = fopen("POKUS.TXT", "w");
    fprintf(fw, "TOTO JE PRVNI TEST");
    fclose(fw);

    fw = fopen("POKUS.TXT", "a");
    fseek(fw, OL, SEEK_SET); /* nastavení na začátek souboru */
    printf("Pozice: %ld\n", ftell(fw));
    fprintf(fw, "toto je druhý test");
    fclose(fw);
    return 0;
}
```

Závěrečné poznámky k otevírání souborů:

- Všimněte si, že pro přístup k souboru definujeme ukazatel na typ FILE: FILE *f; ne proměnnou typu FILE: FILE f;
- Pole⁹ proměnných typu FILE je již definováno systémem a jeho deklarace je v <stdio.h>, např. jako: extern FILE _iob[];
- Při otevírání souborů se tedy dynamicky nealokuje další prvek tohoto pole, ale přiřadí se nějaký – nejčastěji následující – volný.
- Protože je to pole statické, je zřejmé, že počet současně otevřených souborů je omezen jeho velikostí.
- Informace o maximálním počtu současně otevřených souborů je také v <stdio.h>. Má název FOPEN_MAX a hodnotu často 20.

⁸ Popisy funkcí fseek() a ftell() viz na str. 300 a na str. 301.

⁹ V podstatě je nám jedno, zda je to pole, či jiná datová organizace. Podstatné je, že strukturu typu FILE nealokujeme my. Viz též str. 328.

17.4 Uzavření souboru

Pro tuto akci se používá funkce: int fclose(FILE *fp);

Pokud se soubor nepodaří uzavřít¹⁰ vrací fclose() hodnotu EOF. V případě úspěchu vrací nulovou hodnotu.

Poté, co soubor uzavřeme, je možné používat ukazatel fp pro otevření a následnou práci s jiným souborem.

Základní zásada, kterou bychom měli neustále dodržovat, je, že zavíráme soubory okamžitě poté, co jsme s nimi přestali pracovat.

Před tím, než fclose() uvolní alokovanou položku v poli struktur typu FILE, zapíše obsah bufferu¹¹ do souboru (*flushing*).

ANSI C zaručuje, že se při správném ukončení programu všechny neuzavřené soubory automaticky uzavřou včetně vyprázdnění bufferů. Mohlo by se tedy zdát, že uzavírání souborů je činnost víceméně zbytečná, což je ovšem velmi nebezpečná představa.

Při havárii programu, kdy program není ukončen return ve funkci main() nebo voláním funkce exit()¹², není explicitně určeno, co se stane s neuzavřenými soubory. Některé systémy soubory uzavírají včetně *flushingu* a jiné je pouze uzavírají, čímž se obsah bufferů ztratí. Může se dokonce stát, že neuzavřené soubory zůstanou v nekonzistentním stavu, tzn. z pohledu operačního systému se budou jevit jako chybné či neexistující.

Další dobrý důvod, proč uzavírat již nepotřebné soubory je, že počet současně otevřených souborů je omezen, takže uzavřením vlastně dáváme šanci jinému souboru, aby byl otevřen.

Poznámky:

- Pomocí fclose() je možné zavřít i standardní vstupy a výstupy – stdin, stdout a stderr. Toto uzavření platí jen pro program, ve kterém byly uzavřeny. Činnost případných dalších současně nebo později spuštěných programů není nijak ovlivněna.
- Uzavření těchto proudů může mít různé výhody, např. u stdout a stderr můžeme jejich uzavřením potlačit všechny následující výpisy. Uzavřením stdin pak můžeme potlačit všechny následující vstupy. To je výhodné, jestliže např. v programu často čekáme na stisk klávesy pomocí: (void) getchar();

¹⁰ Například je plný disk, změna fp během programu, vytažená disketa, ...

¹¹ Viz bufferování – na str. 294.

¹² Viz str. 342.

a my chceme tato čekání ignorovat bez toho, že bychom chtěli volání funkce `getchar()` jakkoliv¹³ ze zdrojového kódu vymazávat.

- Uzavření `stderr` se používá často při ladění (viz str. 378).
- Standardní vstupy a výstupy `stdin`, `stdout` a `stderr` lze pomocí funkce `freopen()` programově přeměrovat do/z souboru (viz str. 306).

Příklad:

Následující program ukazuje, že standardní vstupy a výstupy lze zavřít.

```
#include <stdio.h>
```

```
int main(void)
{
    FILE *fw;

    printf("printf() - pred uzavrenim\n");
    fprintf(stdout, "fprintf() - pred uzavrenim\n");

    printf("Po stisku <Enter> bude vystup uzavren: ");
    while (getchar() != '\n')
        ;

    fclose(stdout);                /* prvni zavreni */
    if (fclose(stdout) == EOF) {    /* pokus o druhe zavreni */
        fprintf(stderr, "chyba pri zavirani stdout\n");
    }

    if (fclose(stdin) == EOF) {
        fprintf(stderr, "chyba pri zavirani stdin\n");
        return 1;
    }

    printf("printf() - PO uzavreni\n");
    fprintf(stdout, "fprintf() - PO uzavreni\n");

    fprintf(stderr, "Na dalsi stisk <Enter> uz program neceka");
    (void) getchar();

    return 0;
}
```

¹³ Například editorem nebo podmíněným příkladem – viz též str. 379.

Vypíše:

```
printf() - pred uzavrenim
fprintf() - pred uzavrenim
Po stisku <Enter> bude vystup uzavren:
chyba pri zavirani stdout
Na dalsi stisk <Enter> uz program neceka
```

Poznámka:

- Všimněte si způsobu reakce na chybu – je použit výpis na neuzavřený `stderr`, protože `stdout` byl již zavřen. Z toho je také vidět, že `stdout` a `stderr` jsou rozdílné výstupní proudy.

17.5 Ošetření chyb

Téměř každá vstupně-výstupní funkce má možnost vrátit nějakou hodnotu, představující zprávu o vzniklém chybovém stavu. Je vhodné tyto stavy testovat a ošetřovat, protože periferní zařízení vždy představují potenciální zdroj problémů.

Při snaze o testování brzy přijdeme na to, že každá funkce vrací jako chybový stav něco jiného – jinou hodnotu nebo i jiný typ. Například `fopen()` vrací ukazatel `NULL`, `fclose()` vrací `int` s hodnotou `EOF`, `setvbuf()` vrací nenulovou hodnotu typu `int`, atd.

I přesto je možné testovat chyby vstupů a výstupů do souboru alespoň částečně jednotně. V datové struktuře `FILE` je totiž položka, která se využívá pro signalizaci, že nastala chyba při práci se souborem.

Pro testování chyby, tj. pro čtení hodnoty této položky, slouží standardní makro `ferror()`, které vrací nenulovou hodnotu, nastala-li při práci se souborem chyba.

Další funkce, která se v této souvislosti používá, je `clearerr()`, která tento příznak nastaví na nulu, protože `ferror()` to neprovádí automaticky.

Příklad:

Následující funkci `stav_souboru()` lze použít pro testování, zda je vše dosud v pořádku – při nulové hodnotě `ano`, při hodnotě `1` nastala nějaká chyba.

```
#define ERR_STAV 1
int stav_souboru(FILE *f)
{
    int stav = 0;
    if (ferror(f) != 0)
        stav = ERR_STAV;
    clearerr(f);
    return stav;
}
```

Kromě makra `ferror()` lze využít i toho, že existuje globální proměnná `errno` (viz str. 322 a str. 326), do které některé funkce zapisují kód chyby jako vedlejší efekt své činnosti, a která objasňuje příčinu chyby.

Text této chyby lze pak vypsát na `stderr` pomocí funkce `perror()` – viz str. 327.

Poznámka:

- Proměnnou `errno` využívají i jiné funkce než funkce pro vstupy a výstupy. Jsou to např. funkce z `<math.h>` – viz str. 330.

Příklad:

Následující program ukazuje na vhodné použití proměnné `errno`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;

    errno = 0;
    fp = fopen("NESMYSL.DAT", "r");
    if (fp == NULL) {
        fprintf(stderr, "Soubor neotevren\n");
        if (errno != 0)
            perror("Byla to chyba typu");
    }
    return 0;
}
```

Vypíše:

Soubor neotevren

Byla to chyba typu: No such file or directory

Poznámka:

Všimněte si dvou věcí:

- 1) Příkaz: `errno = 0;`
je velmi vhodný, protože funkce musí sice při chybě `errno` nastavit, ale nemusí ho nulovat, pokud je vše v pořádku. V tomto případě by se mohlo stát, že bychom při bezchybném stavu vypisovali chybu, která nastala někdy dříve.
- 2) Test: `if (errno != 0)`
je vnořen do testu na nulovost `fp`: `if (fp == NULL)`
Je to z toho důvodu, že funkce smí vrátit `errno` nastavené, i když chyba nenastala přímo v ní. To se může stát, když sama volá další (pomocnou) funkci, ve které nastala chyba.

Z výše uvedeného vyplývá, že jen na samotnou hodnotu `errno` se tedy nelze 100% spolehnout¹⁴ a čím více možných testů provedeme, tím lépe. Při testování chyby bychom tedy měli dodržovat následující hierarchii:

```
if (fp == NULL) je test na výskyt problému
if (errno != 0) je test na příčinu tohoto problému
```

17.6 Čtení a zápis dat

S daty v souboru lze pracovat v zásadě dvěma různými způsoby – formátovaně a neformátovaně.¹⁵

Poznámka:

- Tyto dva způsoby jsou ovlivněny jen málo režimem otevření souboru, tzn. textovým nebo binárním režimem. Z toho vyplývá, že do souboru otevřeného v textovém režimu lze zapisovat formátovaně i neformátovaně stejně jako lze formátovaně i neformátovaně zapisovat do souboru otevřeného v binárním režimu. Je zřejmé, že i když jsou tyto akce možné, nejsou vždy logicky správné.

¹⁴ Nutno ovšem poznamenat, že solidní překladač (knihovna) má tyto nepříjemné výjimky ošetřeny.

¹⁵ Způsob otevírání – `fopen()` – a uzavírání – `fclose()` – souborů je pořád stejný.

17.6.1 Formátované vstupy a výstupy

Při tomto způsobu výpisové nebo čtecí funkce ovlivňují data, což znamená, že v souboru jsou data jinak reprezentovaná, než v paměti počítače.

Tyto akce se typicky provádějí funkcemi typu `fprintf()` a `fscanf()`, kde pro `fprintf()` do souboru zapisujeme obsah proměnné (výrazu), který je automaticky převeden na řetězec. To v praxi znamená, že zapisujeme-li např. proměnnou typu `int`, pak v paměti tato proměnná zabírá dva bajty¹⁶, ale stejné číslo v souboru může zabírat jeden až šest bajtů (znaků), v závislosti na jeho hodnotě (od 0 do -32768).

Formátované vstupy a výstupy jsou většinou využívány při práci s textovými soubory otevřenými v textovém režimu. Například pomocí `fprintf()` můžeme zapsat i do binárního souboru – ovšem je to prakticky nesmysl – a opět dojde k automatické konverzi typu číslo na řetězec.

Všechny funkce používané pro formátované V/V byly podrobně popsány na str. 40, 209, ...

17.6.2 Neformátované vstupy a výstupy

Data v souboru jsou reprezentovaná stejně jako v paměti počítače, tedy v souboru je totéž, co v programu. Malou výjimkou¹⁷ z tohoto pravidla tvoří neformátovaný zápis nebo čtení¹⁸ do souborů otevřených v textovém režimu.

Neformátované vstupy a výstupy se rozlišují podle najednou zpracovávaného množství dat:

<i>najednou se čte nebo zapisuje</i>	<i>výhodné použití pro</i>
jeden znak (bajt)	textové i binární soubory
jedna řádka	textové soubory
jeden blok dat	binární soubory

Typické případy použití:

- jeden znak – zpracováváme textový nebo binární soubor po bajtech a chceme reagovat bezprostředně po přečtení každého bajtu,
- jedna řádka – výhradně pro zpracování textových souborů,
- jeden blok – zpracovávání polí, popř. blokové zpracování souboru, např. při vzorkování audiodat, zpracování obrazu, ...

¹⁶ `sizeof(int) == 2`

¹⁷ Viz problémy se znakem `'\r'` na str. 276.

¹⁸ Například pomocí `putc()` nebo `getc()`.

Poznámky:

- Z výše uvedených faktů je doufejme zřejmé, že všechny tři způsoby lze použít jak pro binární soubory¹⁹, tak i pro textové soubory, ale v některých případech to není výhodné, případně je to zdrojem chyb.
- Chceme-li nejdříve načíst do paměti blok znaků a pak je najednou zpracovávat, je z časového hlediska téměř jedno, zda je čteme do paměti po jednom znaku nebo blokově. Díky bufferování (viz str. 294) bude totiž čas čtení po znacích delší jen málo.

Každá z metod má své pro a proti, což bude diskutováno v následujícím textu.

Zpracování jednoho znaku v jeden okamžik

Používají se funkce:

`int fgetc(FILE *fr);` – čtení jednoho znaku ze souboru
`int fputc(int c, FILE *fw);` – zápis jednoho znaku do souboru

a makra:

`int getc(FILE *fr);` – čtení jednoho znaku ze souboru
`int putc(int c, FILE *fw);` – zápis jednoho znaku do souboru

Poznámka:

- Makra `getc()` a `putc()` provádí totéž, co funkce `fgetc()` a `fputc()`, ovšem makra by teoreticky měla být rychlejší. Různé zdroje udávají, že rozdíly mohou být v závislosti na operačním systému, překladači a velikosti vnitřního bufferu až dvojnásobné. Praktické pokusy (byť triviální) toto tvrzení příliš nepodporují. Například načtení souboru 112 MB dlouhého trvalo (Athlon 1,4 GHz, 512 MB RAM překladač `gcc`, prostředí `cygwin`) pomocí `getc()` 1946 msec a pomocí `fgetc()` 1950 msec – průměr z deseti pokusů.

Při čtení souboru po znacích je třeba pečlivě odlišovat soubory s textovým a binárním obsahem. Chceme-li se vyhnout problémům, otevíráme soubor vždy v takovém režimu, jaký je obsah souboru. Nedokážeme-li to předem určit, použijeme vždy binární režim, ve kterém nemůže dojít ke zkreslení informace.

Problémové bajty při čtení binárních souborů v textovém režimu

Pokud provádíme experimenty a otevřeme soubor s binárním obsahem v textovém režimu, můžeme se dostat do potíží se třemi hodnotami bajtů:

¹⁹ Přesněji řečeno – pro soubory otevřené v binárním režimu.

1) Bajt s hodnotou 255 (0xFF)

Hodnota 255 představuje -1 pro typ **signed char** a právě -1 je většinou také hodnota symbolické konstanty EOF ze `<stdio.h>`.

Z toho vyplývá, že platný bajt²⁰ s hodnotou 255 může být za nevhodných okolností považován za EOF.

Příklad:

Následující výsek programu ukazuje způsob, jak si zaručit problém.

```
char c;
while ((c = getc(fr)) != EOF)
```

U typu **char** není v ANSI C stanoveno, zda je implicitně **unsigned** či **signed**, ovšem problémy nastanou v obou případech:

- 1) **signed char**: Načtený bajt s hodnotou 0xFF je chápán jako -1 a při porovnání s EOF překladač provádí implicitní konverzi²¹ ze **signed char** na **signed int**, tedy z 0xFF (-1) na 0xFFFF (-1).

Výsledkem je, že program skončí čtením tohoto bajtu, byť soubor pokračuje dále.

- 2) **unsigned char**: Načtený bajt s hodnotou 0xFF je chápán jako 255 a při porovnání s EOF překladač provádí implicitní konverzi z **unsigned char** na **signed int**, tedy z 0xFF (255) na 0x00FF (255). Výsledkem je, že program přečte bajt správně a čte dál.

Výše uvedená konverze ale proběhne i na skutečném konci programu, kdy je vrácena hodnota EOF (0xFFFF), která je přiřazením do proměnné *c* konvertována na 0xFF, protože do typu **char** se větší číslo nevejde. Hodnota 0xFF je při následném porovnání s EOF implicitně konvertována opět na 0x00FF, což se nerovná EOF. Výsledkem je, že se program zacyklí a čtení nikdy neskončí.

Tento problém odstraníme tím, že definujeme *c* typu **int**, což je správný návratový typ pro `getc()`, a tím se vyhneme implicitní konverzi **int** → **char** → **int**.

Jiným způsobem, jak si poradit s bajtem o hodnotě 0xFF, je použít makro `feof()`. Toto makro lze použít pro testování konce souboru a nahrazuje tak konstantu EOF. Vrací nulu, dokud je co číst, a nenulovou hodnotu na konci souboru.

²⁰ V souboru s binárním obsahem se mohou libovolně vyskytovat bajty s hodnotami binárně od 00000000 do 11111111.

²¹ Viz str. 87.

Poznámka:

- Pozor na skutečnost, že `feof()` vrací nenulovou hodnotu, bylo-li předchozí čtení na konci souboru. Čili po čtení posledního bajtu souboru vrací `feof()` stále nulu!

Příklad:

Následující výseky programu ukazují správné a chybné použití `feof()`.

```
/* správné použití */
int c; /* nebo char c; */
while (c = getc(fr), feof(fr) == 0) {
    ...
}
/* je sice přečten nesmyslný znak, ale není použit */
```

```
/* chybné použití - načte a zpracuje o jeden znak víc */
int c; /* nebo char c; */
while (feof(fr) == 0) {
    c = getc(fr);
    ...
}
/* přečtený nesmyslný znak je pravděpodobně v cyklu
   while dále použit */
```

2) Bajt s hodnotou ukončovacího znaku souboru

Druhým problémem, který neodstraní ani použití typu **int** ani makro `feof()`, je bajt s hodnotou ukončovacího znaku textových souborů, používaný některými editory.

V mnoha systémech totiž může být používán na konci souboru speciální ukončovací znak – ve Windows je to `Ctrl-Z` s hodnotou 26 (0x1A), v UNIXu `Ctrl-D` s hodnotou 4 (0x4).

Tento znak může²² být uveden jako poslední (nevýznamový) bajt v textovém souboru. Bajt s toutéž hodnotou se ale může objevit jako platný bajt i kdekoli uvnitř souboru s binárním obsahem.

Pokud je soubor s binárním obsahem otevřen v textovém režimu ("r") místo v binárním ("rb"), pak při načtení tohoto znaku systém místo něj automaticky vygeneruje hodnotu EOF a nastaví příznak konce souboru, takže i `feof()` vrací nenulovou hodnotu – hlásí konec souboru.

²² Ale nemusí, protože operační systém dokáže poznat konec souboru podle jeho délky.

3) Bajt s hodnotou '\r'

Při čtení je bez jakékoliv náhrady vypuštěn („požrán“). Toto je specialita jen Windows, ale v jiných systémech mohou nastat problémy s jinými hodnotami v souvislosti s organizací konce řádky u textových souborů.

Poznámka:

- Jedinou spolehlivou obranou proti problémům při výskytu těchto tří bajtů je proto otevírat soubory s binárním obsahem důsledně v režimu "rb" a číst znaky do proměnné typu `int`.

Příklad:

S ohledem na předchozí skutečnosti je zcela spolehlivé kopírování souboru neznámé délky s neznámým obsahem, tedy potenciálně binárního, jen takto:²³

```
#include <stdio.h>
```

```
int main(void)
{
    FILE *fr, *fw;
    int c;

    fr = fopen("A.DAT", "rb");
    fw = fopen("B.DAT", "wb");

    while ((c = getc(fr)) != EOF)
        putc(c, fw);

    fclose(fr);
    fclose(fw);
    return 0;
}
```

Zpracování celé řádky najednou

Je určeno téměř výhradně pro textové soubory (podrobný popis funkcí viz str. 290). Pro čtení slouží funkce:

```
char *fgets(char *str, int max, FILE *fr);
```

Pro zápis řádky do souboru se používá funkce:

```
int fputs(char *str, FILE *fw);
```

²³ Testy na správnost otevření a uzavření nejsou uvedeny jen z důvodu šetření místem!

Příklad:

Následující program zkopíruje obsah souboru A.TXT do B.TXT. Ačkoliv je délka čtené řádky omezena na 5, bude mít soubor B.TXT stejný obsah jako A.TXT nehledě na délku řádek. Kopírování bude ovšem probíhat jen po čtyřech²⁴ bajtech.

```
#include <stdio.h>
```

```
#define DELKA_RADKY 5
```

```
int main(void)
{
    char radka[DELKA_RADKY];
    FILE *fr, *fw;

    fr = fopen("A.TXT", "r");
    fw = fopen("B.TXT", "w");

    while (fgets(radka, DELKA_RADKY, fr) != NULL)
        fputs(radka, fw);

    fclose(fr);
    fclose(fw);
    return 0;
}
```

Poznámka:

- Pokud bychom chtěli uvedený program používat pro kopírování souborů s binárním obsahem a otevřeli bychom tedy oba soubory v binárním režimu, s vysokou pravděpodobností bychom se dostali do problémů. Problémovým bajtem by byl tentokrát bajt s hodnotou nula (0x00), který slouží jako koncový znak řetězce. Funkce `fputs()` by pak do souboru zapisovala bajty jen do tohoto znaku a všechny další načtené bajty, ležící za 0x00 by zůstaly nezapsány.

²⁴ Pátým bajtem je ukončovací znak '\0'.

Zpracování celého bloku dat najednou

Tento způsob je určen většinou pro binární soubory (samozřejmě otevřené v binárním režimu).

Blok si lze představit jako pole o určitém počtu prvků a určité velikosti každého prvku, přičemž pole prvků typu `char` je nejjednodušší případ.

Za blokem prvků jednoho typu může následovat jiný blok rozdílné délky složený z prvků jiného typu, např. po bloku pěti `int` následuje blok osmi `double`. Tento způsob se však nepoužívá příliš často. Nejčastější je použití jen jednoho typu prvků, takže je soubor složen např. jen z prvků typu `float`.

Nejvýhodnější pro práci s celým blokem dat jsou soubory, v nichž jsou uložena rozměrná data – nejčastěji velká pole struktur²⁵. Pro tuto činnost se používají kromě `fread()` a `fwrite()` ještě funkce `fseek()` a `ftell()`.

Pro čtení a zápis se využívají funkce:

```
int fread(void *kam, size_t rozmer, size_t pocet, FILE *fr);
int fwrite(void *odkud, size_t rozmer, size_t pocet, FILE *fw);
```

kde:

`kam` – adresa paměti, kam se uloží přečtený blok dat,
`odkud` – adresa paměti, odkud se přečte zapisovaný blok dat,
`rozmer` – velikost jedné položky z bloku dat v bajtech,
`pocet` – počet zpracovávaných položek (ne bajtů!)
`fr` a `fw` – ukazatelé na aktuální otevřený soubor

Poznámky:

- Obě funkce vrátí počet skutečně zapsaných nebo přečtených položek. Ten se liší od parametru `pocet` jen tehdy, bylo-li při čtení dosaženo konce souboru nebo došlo-li k chybě. Obě tyto události lze od sebe rozlišit pomocí makra `ferror()`.
- Zpracovávaný soubor musí být otevřen v binárním režimu. Pokud je otevřen v textovém režimu ve Windows, lze očekávat problémy – podrobně viz str. 302.

Příklad:

Následující program kopíruje soubor po blocích pěti bajtů.

```
#include <stdio.h>
#include <stdlib.h>
```

²⁵ Viz příklad na str. 302.

```
#define DELKA_BLOKU 5
```

```
int main(void)
```

```
{
```

```
    char blok[DELKA_BLOKU];
```

```
    FILE *fr, *fw;
```

```
    int precteno;
```

```
    fr = fopen("A.TXT", "rb");
```

```
    fw = fopen("B.TXT", "wb");
```

```
    while ((precteno = fread(blok, sizeof(char), DELKA_BLOKU, fr))
           == DELKA_BLOKU) {
```

```
        if (fwrite(blok, sizeof(char), DELKA_BLOKU, fw)
```

```
            != DELKA_BLOKU) {
```

```
            fprintf(stderr, "Chyba pri zapisu do souboru\n");
```

```
            exit(1);
```

```
        }
```

```
    }
```

```
    /* cteni skoncilo prectenim vseho nebo chybou */
```

```
    if (ferror(fr) != 0) {
```

```
        fprintf(stderr, "Chyba ve cteni souboru\n");
```

```
    }
```

```
    else {
```

```
        /* skoncil prectenim mene dat, nez je velikost bloku */
```

```
        /* prectena data je nutno zapsat, i kdyz jich je mene */
```

```
        if (fwrite(blok, sizeof(char), precteno, fw) != precteno)
```

```
            fprintf(stderr, "Chyba pri zapisu do souboru\n");
```

```
    }
```

```
    fclose(fr);
```

```
    fclose(fw);
```

```
    return 0;
```

```
}
```

Poznámka:

- Uvedený program ale není typickým příkladem použití blokových operací. Ten bude uveden na str. 302 spolu s ukázkou možných potíží.

17.7 Bufferování

17.7.1 Základní informace

Z hlediska operačního systému je každý soubor posloupnost bajtů uložených na nějakém médiu, nejčastěji na disku, v několika blocích. Bloky mají stejnou velikost a nemusí nutně ležet za sebou. Jak se s nimi pracuje je záležitost jen a jen operačního systému a nás to nemusí zajímat. Soubor je tedy vytvářen a spravován podle pravidel daných operačním systémem.

Z uživatelského hlediska je soubor posloupnost po sobě jdoucích bajtů (nebo řádek nebo bloků) od začátku do konce souboru. Starostí operačního systému je, aby nám tyto bajty (řádky, bloky) dodal ve správném pořadí.

V porovnání s přístupem do paměti jsou vstupně-výstupní operace velmi pomalé a existuje pochopitelná snaha omezit jejich četnost na co nejmenší míru. Jeden z nejsnazších způsobů je tzv. *bufferování*, které je pro programátora o to příjemnější, že se o něj nemusí starat, protože už je systémem zajištěno.

Buffer je část operační paměti alokovaná systémem, kde jsou data dočasně uložena před zasláním na místo konečného určení. To představuje v případě čtení datovou oblast paměti využívanou naším programem a v případě zápisu nejčastěji disk.

Operační systém pak přistupuje k perifernímu zařízení pouze tehdy, když potřebuje načíst nebo zapsat celý obsah bufferu najednou.

Zmíněný buffer má nejčastěji velikost 512 nebo 1024 bajtů, a tato velikost závisí na velikosti nejmenšího úseku dat zapsatelného na disk.

Chceme-li načíst jeden bajt, načte se celý buffer. Motivace je v tom, že čtení jednoho bajtu nebo 512 bajtů trvá téměř stejnou dobu, čili zpomalení způsobené čtením redundantních 511 bajtů a jejich přenosem do bufferu v operační paměti je minimální.

Chceme-li dále číst další bajty, což je nejčastější požadavek, pak se oněch „redundantních“ 511 bajtů v operační paměti rázem změní na 511 „vhodně přednačtených“ bajtů. Výsledkem tedy je, že čteme data několikanásobně rychleji z operační paměti a tak ušetříme 511 přístupů na disk.

Stejný princip platí i pro výstup, kdy se data zapisují ne přímo na disk, ale do bufferu (paměti). Když je buffer plný, zapíše se automaticky najednou celý jeho obsah na disk do souboru jako jeden blok dat. Výhodou je opět větší rychlost zapisování do paměti než na disk.

Poznámka:

- Celý princip má samozřejmě také nevýhody. Uvedme dvě nejpodstatnější. První z nich je, že z nejrůznějších důvodů může dojít k nesouladu dat v bufferu s daty na disku. Pak při čtení získáváme jiná data, než očekáváme. Druhá nevýhoda je, že v určitých případech může selhat automatické zapisování bufferu na disk. Pak program data prokazatelně zapisuje a v souboru prokazatelně nejsou. Tomuto druhému problému se dá s úspěchem čelit použitím funkce `fflush()` (viz str. 299), která vydá explicitní příkaz k zapsání bufferu, bez ohledu na jeho naplnění.

Operační systém přednastavuje pro různé druhy vstupně-výstupních operací různé způsoby bufferování – viz následující pododdlíl. Bez vážného důvodu se nevyplácí tato nastavení měnit, protože často dostaneme mnohem pomalejší program. Přednastavené buffery mají totiž z hlediska operačního systému „rozumnou“ velikost, a existují i případy, kdy změnu ani nejde provést – viz str. 296.

17.7.2 Možnosti bufferování poskytnuté ANSI C

1) Řádkové bufferování

Do bufferu se načítá nebo zapisuje, dokud není plný nebo dokud se v něm nevyskytne znak `'\n'`. Typicky je tak nastaveno `stdin`²⁶, kde probíhá čtení z klávesnice do bufferu do té doby, než je stisknuta klávesa *Enter*.

2) Blokové bufferování

Do bufferu se načítá vždy jeho celá velikost a nové načtení proběhne, až když jsou všechna data z bufferu programem zpracována. Do bufferu se zapisuje, dokud není plný, a je zapsán na disk až po svém úplném zaplnění. Typicky je to práce se soubory, bez ohledu na jejich režim otevření.

3) Žádné bufferování

Znak se zapisuje nebo čte okamžitě po volání příslušné funkce. Typicky jsou takto nastaveny `stdout` a `stderr` ve Windows.

17.7.3 Funkce pro práci s bufferem

Funkce `setbuf()`

Funkční prototyp: `void setbuf(FILE *f, char *buf);`

Tuto funkci lze použít pouze na otevřený soubor, ale před tím, než z něho bylo poprvé čteno nebo zapsáno. To znamená, že po jakékoli vstupně-výstupní operaci se souborem nelze `setbuf()` pro tento soubor použít.

²⁶ A v UNIXu občas také `stdout` a `stderr`.

Možná použití `setbuf()`:

1) Vypnutí bufferování

Má-li parametr `buf` hodnotu `NULL`, pak budou V/V operace se soubory (nikoliv se standardními vstupy a výstupy) prováděny okamžitě.

V manuálech se udává, že je výhodné použít `setbuf()` pro vypnutí bufferování `stdin`, příkazem: `setbuf(stdin, NULL)`;

To by bylo opravdu výhodné, protože pak by např. `getchar()` po stisknutí klávesy znaku nečekala na následný stisk `Enter`. Bohužel praktické pokusy tuto možnost nepotvrzují.²⁷

2) Přesměrování bufferu do našeho programu

Alokováním nového bufferu přesměrujeme tento buffer do oblasti paměti přímo přístupné našemu programu. Toho lze využít při nejrůznějších speciálních požadavcích, kdy například jedním voláním funkce `fgetc()` načteme celý buffer a přímým přístupem do tohoto bufferu ho můžeme „předzpracovat“.

Funkce `setvbuf()`

Funkční prototyp:

```
int setvbuf(FILE *f, char *buf, int režim, int velikost);
```

Tato funkce je podstatně rozšířená funkce `setbuf()`, která díky novým parametrům získala zajímavé možnosti – změnu režimu bufferování a hlavně změnu velikosti bufferu. Funkce vrací nulu při úspěchu a nenulovou hodnotu při neúspěchu.

Parametry mají podobný význam, jako u `setbuf()`, navíc jsou použity:

velikost – velikost bufferu v bajtech
režim – volí způsob (režim) bufferování
 – používají se tři konstanty (definované v `<stdio.h>`):
 `_IOFBF` – blokové bufferování (F = full)
 `_IOLBF` – řádkové (L = line)
 `_IONBF` – žádné (N = none)

Poznámky:

- Buffer musíme opět alokovat sami, nejlépe jako statický (**static**), a funkci `setvbuf()` musíme použít před první vstupně-výstupní operací.
- Podle manuálů jednoduchý (bohužel, ale nefunkční) způsob vypnutí bufferování `stdin` je např.:

```
if (setvbuf(stdin, NULL, _IONBF, 0) != 0)
    printf("Chyba\n");
else printf("stdin nebufferovano\n");
```

²⁷ Řešení existuje – viz str. 297, ale není založeno na vypnutí bufferování.

Slavneme-li, co je možné pomocí zmíněných funkcí provést:

- Přesměrovat buffer do oblasti paměti z programu přímo přístupné. To by umožňovalo při čtení ovlivňovat budoucnost – data načtená do bufferu by bylo možné upravit. Při zápisu je to ovlivnění minulosti, kdy jednou zapsaná data lze dodatečně prohlédnout a upravit. Obě dvě možnosti by se musely použít po velmi zralé úvaze.
- Změnit režim bufferování. Nemá přílišný význam, protože většinou nelze provést pro `stdin`, `stdout` nebo `stderr`. Prakticky jsem ověřil, že vypnutí bufferování souboru (`_IONBF`) způsobí zpomalení čtení asi 1000 krát. Z toho plyne poučení, že náhodné experimenty s režimem mohou způsobit nečekané potíže.
- Změnit velikost bufferu, což může urychlit V/V operace se souborem – to vidím jako jediný praktický přínos. Při pokusech (Athlon 1,4 GHz, 512 MB RAM překladač `gcc`, prostředí `cygwin`) dával nejlepší výsledky buffer velikosti 16 KB, kdy se čtení oproti přednastavené velikosti bufferu zrychlilo asi o 50 %.

17.7.4 Řešení problému nebufferovaného `stdin`

Stav, který je popsán na str. 296, kdy funkce, které čtou z klávesnice, nečekají na stisk klávesy `Enter`, je v mnoha případech velmi žádoucí. Bohužel ho nelze dosáhnout vypnutím bufferování.

Tuto užitečnou vlastnost potřebujeme v programu častokrát a není-li možné situaci vyřešit voláním nestandardní, ale již připravené funkce, např. `getch()` v některých překladačích, je nutné hledat jiná řešení.

Jedno z možných jednodušších řešení²⁸ v UNIXu je uvedeno dále. Podstatně lepší a komplexnější řešení viz soubory v adresáři `getch`.

```
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>
```

```
int kbhit(void)
{
    char c;
```

²⁸ Princip navrhl pan Hruša, zde uvedená verze je dílo autora této knihy, na kterého směřujte kritiku.

```

static struct termios new_term;
static struct termios old_term;
static int handle;
static int init = 0;

fflush(stdout);

if (init == 0) {
    init = 1;
    if ((handle = open("/dev/tty", O_RDWR | O_SYNC)) == 0) {
        fprintf(stderr, "Nelze číst z terminalu!\n");
        exit(1);
    }
    else {
        tcgetattr(handle, &old_term);
        new_term = old_term;
        new_term.c_lflag &= ~ICANON;
        new_term.c_lflag &= ~ECHO;
        new_term.c_cc[VMIN] = 0;
        new_term.c_cc[VTIME] = 1;
    }
}

tcsetattr(handle, TCSANOW, &new_term);
if (read(handle, &c, 1) <= 0) {
    c = 0;
}
else {
    ungetc(c, stdin);
}
tcsetattr(handle, TCSANOW, &old_term);

return c;
}

int main(void)
{
    int c;

    while(1) {
        putchar('>');

```

```

if (kbhit() != 0) {
    c = getch();
    printf("Stisknuta klavesa - '%c'", c);

    if (c == 'Z')
        break;
    }
    printf("<\n");
}
return 0;
}

```

Poznámka:

- Tento program je funkční, ale nevýhodou funkce `kbhit()` je její značná pomalost a závislost pouze na operačním systému UNIX. Důležité však je, že je to možnost, kterou lze reálně použít.

17.8 Funkce pro zápis bufferu na disk

V souvislosti s bufferováním mají význam dvě funkce – `fflush()` a `fclose()`, které jsou v této souvislosti zmiňovány téměř výhradně ve spojení se zápisem.

Pokud jsou použity na vstupní *stream*, čili při čtení, není výsledek přesně definován, ale nejčastěji se data do bufferu již načtená „zahodí“.

17.8.1 Funkce `fflush()`

Funkční prototyp: `int fflush(FILE *f);`

Volání funkce způsobí, že obsah výstupního bufferu bude okamžitě zapsán na výstupní zařízení, soubor ale zůstává i nadále otevřený. To je výhodné v případě, že si chceme být jisti, že se data, která zapisujeme na disk, na disku skutečně objeví. Možné místo použití této funkce by tedy bylo před nějakou částí programu, od které se dají očekávat potenciální problémy, které by mohly vést ke zhroucení programu.

V případě neúspěchu (např. plný disk, atd.) vrací `fflush()` konstantu `EOF`, jinak vrací nulovou hodnotu.

Druhou funkcí, která provede zápis bufferů na disk, je funkce `fclose()` (viz str. 281). Ta se liší od funkce `fflush()` tím, že soubor navíc uzavře.

Poznámka:

- Zdůrazněme ještě jednou, že bez použití `fflush()` nebo `fclose()` se může stát, že po havárii programu nebo po volání standardní funkce `abort()` (viz str. 341) nebudou data uložená ve výstupních bufferech zapsána do příslušných souborů.

17.9 Přímý přístup do souboru

V textových souborech se příliš nevyužívá možnost pohybu v souboru – jednotlivé znaky se z nich prostě sekvenčně čtou nebo zapisují. V binárních souborech, kde musíme znát přesně velikosti jednotlivých položek, je mnohdy velmi účelná možnost nastavit si ukazovátka do souboru na libovolné místo. Od tohoto místa se bude dále číst nebo se bude od něho dále zapisovat²⁹, bez nutnosti zdržování se zpracováním předchozích dat.

Poznámka:

- Možnost přímého přístupu do binárního souboru ovšem neznamená, že binární soubor nelze číst sekvenčně. Sekvenční čtení je běžně užívaná akce, kdy chceme např. načíst všechna data do paměti.

Pro práci s přímým přístupem do souboru využíváme dvě následující funkce.

17.9.1 Funkce `fseek()` – posun v souboru

Funkční prototyp: `int fseek(FILE *f, long posun, int odkud);`

Funkce přesune ukazatel na aktuální pozici v souboru na libovolné místo v souboru. Jednotlivé formální parametry mají následující význam:

<code>f</code>	Ukazatel na otevřený soubor.
<code>posun</code>	Počet bajtů od pozice v souboru dané parametrem <code>odkud</code> . Může mít kladnou i zápornou hodnotu. Pozor na typ long – při používání konstanty je nutné explicitní přetypování – viz následující příklad.
<code>odkud</code>	Místo, odkud se budeme v souboru posouvat. Může nabýt jedné ze tří předdefinovaných hodnot: <code>SEEK_SET</code> – od začátku souboru <code>SEEK_CUR</code> – od aktuální pozice <code>SEEK_END</code> – od konce souboru

²⁹ Možnost zápisu na libovolné místo v souboru záleží také na režimu otevření souboru – viz str. 278.

Funkce `fseek()` vrací nulu v případě úspěchu a nemulovou hodnotu ve zbylých případech³⁰.

17.9.2 Funkce `ftell()` – zjištění pozice v souboru

Funkční prototyp: `long ftell(FILE *f);`

Funkce se používá pro zjištění aktuální hodnoty ukazatele do souboru, tedy, kde se v souboru právě nalézáme. Návratovou hodnotou je posunutí měřené v bajtech od začátku souboru.

Poznámky:

- Obě funkce je možné použít i na textové soubory, je však nutné si uvědomit, že textové soubory jsou implementačně závislé, takže použití těchto funkcí nemusí být zcela korektní.
- V podstatě jedno z mála rozumných použití těchto funkcí v textových souborech je návrat na původní místo, např. po hledání řetězce:

```
akt_pozice = ftell(fr);
fseek(fr, 0L, SEEK_SET); /* posun na začátek souboru */
if (hledej(fr, "ahoj") == NULL) /* hledání selhalo */
    fseek(fr, akt_pozice, SEEK_SET);
```

Příklad:

Následující program ukáže:

- 1) práci s binárním souborem otevřeným v režimu pro čtení i zápis "`wb+`",
- 2) použití `fwrite()` a `fread()` i pro nehomogenní záznam (jeden `int` a jeden `double`),
- 3) použití `fflush()` pro zapsání obsahu bufferu do souboru,
- 4) dvojí použití `ftell()`,
- 5) tři³¹ způsoby přesunu na začátek souboru – všimněte si způsobů přetypování konstanty 0 na typ `long`.

```
#include <stdio.h>
```

```
int main(void)
{
    FILE *f; /* pro čtení i pro zápis */
    int i = 10;
    double d = 3.14159;
```

³⁰ Například pokus o přesun mimo rozsah souboru, soubor nebyl otevřen, ...

³¹ Stačil by jen kterýkoliv z nich – tři jsou jen z důvodu ukázky různých možností. Nejvíce přehledný mi připadá prostřední způsob.

```

f = fopen("POKUS.DAT", "wb+");
fwrite(&i, sizeof(i), 1, f); /* zapis dat do souboru */
fwrite(&d, sizeof(d), 1, f);
fflush(f); /* vyprazdneni bufferu */

printf("Pozice v souboru je %ld\n", ftell(f));

fseek(f, -ftell(f), SEEK_CUR); /* posun na zac. souboru */
fseek(f, 0L, SEEK_SET); /* posun na zac. souboru */
fseek(f, (long) 0, SEEK_SET); /* posun na zac. souboru */

i = 0; d = 0.0; /* nulovani promennych */
fread(&i, sizeof(i), 1, f); /* cteni a zobrazeni dat */
fread(&d, sizeof(d), 1, f);
printf("Nactena data: i = %d, d = %f\n", i, d);
fclose(f);
return 0;
}

```

Poznámka:

- Spleteme-li se a otevřeme-li tento soubor v textovém režimu ("w+" místo "wb+"), pak hodnota 10 (což je mimo jiné hodnota konstanty '\n') v proměnné i způsobí, že se do souboru POKUS.DAT zapíše o jeden bajt s hodnotou 13 (tj. '\r') více. Totéž funguje pro čtení, kdy je tento bajt „požírán“.

U funkcí pracujících s textem je to známá věc, ale u těchto funkcí je to velmi nepříjemné překvapení.

17.9.3 Typické použití přímého přístupu

Následující program by měl ukázat, jak se často pracuje s binárním souborem, který obsahuje pole záznamů – struktur. Současně by měl upozornit na jednu velmi závažnou chybu při použití funkcí fread() a fwrite().

Uvedený program zapíše do binárního souboru DATA.BIN POCET struktur typu ZAZNAM, které náhodně inicializuje.

Po zápisu je proveden přesun na začátek souboru a jsou postupně čteny a jednoduše upravovány jednotlivé položky. Je-li položka upravena, je zpětně zapsána do souboru.

Na závěr se celý soubor kontrolně vypíše.

Všimněte si, že všechny akce jsou důsledně testovány na možnost výskytu chyby.

Poznámka:

- Prakticky je tento příklad ukázkou, jak řešit situaci, kdy se velmi rozsáhlé pole struktur nevejde do operační paměti a proto existuje jen na disku, kde se s ním (postupně) pracuje. Výhodou je, že nejsme omezeni velikostí (do řádově jednotek GB) pole struktur, nevýhodou je značné zpomalení.

```

#include <stdio.h>
#include <stdlib.h>

```

```

#define POCET 10

```

```

typedef struct {
    int poradi;
    int hodnota;
} ZAZNAM;

```

```

int main(void)
{

```

```

    int i;
    FILE *f;
    ZAZNAM z;

```

```

    if ((f = fopen("DATA.BIN", "wb+")) == NULL) {
        printf("Nelze otevrit soubor\n");
        exit(1);
    }

```

```

    /* nahodna inicializace */
    for (i = 1; i <= POCET; i++) {
        z.poradi = i;
        z.hodnota = rand() % 100;
        if (fwrite((void *)&z, sizeof(ZAZNAM), 1, f) != 1)
            printf("Chyba zapisu %d. polozky\n", i);
    }

```

```

    if (fflush(f) == EOF)
        printf("Chyba pri zapisu bufferu\n");
    if (fseek(f, 0L, SEEK_SET) != 0)
        printf("Chyba pri presunu na zacatek souboru\n");

```



```

errno = 0;
/* ctení položek souboru a jejich oprava */
while (fread((void *)&z, sizeof(ZAZNAM), 1, f) == 1) {
    printf("%2d. %2d ", z.poradi, z.hodnota);

    if (z.hodnota < 50) {          /* oprava položky */
        z.hodnota += 50;
        printf(" -> %2d", z.hodnota);

        /* navrat na zacatek položky */
        fseek(f, -(long) sizeof(ZAZNAM), SEEK_CUR);

        /* zapis opravene položky */
        if (fwrite((void *)&z, sizeof(ZAZNAM), 1, f) != 1) {
            printf("Chyba zapisu %d. položky\n", i);
        }
    }
    putchar('\n');
}

if (errno != 0)      /* skoncilo ctení chybou? */
    perror("");

if (fflush(f) == EOF)
    printf("Chyba při zapisu bufferu\n");
if (fseek(f, 0L, SEEK_SET) != 0)
    printf("Chyba při přesunu na zacatek souboru\n");

printf("\nNove hodnoty\n");
errno = 0;
/* vypis opraveného souboru */
while(fread((void *)&z, sizeof(ZAZNAM), 1, f) == 1)
    printf("%d:%2d ", z.poradi, z.hodnota);

if (errno != 0)      /* skoncilo ctení chybou? */
    perror("");
if (fclose(f) == EOF)
    printf("Soubor nelze uzavrit\n");

return 0;
}

```

Tento jednoduchý a silně ošetřený program vypíše např. jen:³²

1. 46 -> 96

Nove hodnoty

Všimněte si, že chybové kontroly nevypsaly žádná hlášení, ačkoliv program evidentně nefunguje a chyba je někde ve V/V operaci.

Celý problém je v tom, že: „Žádná vstupní operace nesmí být nikdy přímo následována výstupní operací (nebo naopak) bez předchozího volání `fseek()`.“³³ Tuto podstatnou větu se bohužel málokde dočtete.

Řešením je použití `fseek()`, která přesune ukazovátka v souboru na totéž místo: `fseek(f, 0L, SEEK_CUR)`;

takže fungující verze programu bude vypadat takto:³⁴

```

/* ctení položek souboru a jejich oprava */
while (fread((void *)&z, sizeof(ZAZNAM), 1, f) == 1) {
    printf("%2d. %2d ", z.poradi, z.hodnota);

    if (z.hodnota < 50) {
        z.hodnota += 50;
        printf(" -> %2d", z.hodnota);

        /* navrat na zacatek položky */
        fseek(f, -(long) sizeof(ZAZNAM), SEEK_CUR);

        /* zapis opravene položky */
        if (fwrite((void *)&z, sizeof(ZAZNAM), 1, f) != 1) {
            printf("Chyba zapisu %d. položky\n", i);
        }
    }

    /* zde vynucena akce pohybu ukazovátka */
    fseek(f, 0L, SEEK_CUR);
}
putchar('\n');
}

```

³² Další možnost je, že se zacyklí – ve Windows Visual C, případně může pracovat správně – v UNIXu gcc.

³³ Volně podle [Koe88].

³⁴ Z důvodů šetření místem je uvedena pouze opravená část. Všechny ostatní části jsou zcela nezměněny.

Takto poopravený program již poskytne očekávané výsledky, např.:

```
1. 46 -> 96
2. 67
3. 34 -> 84
4. 0 -> 50
5. 69
6. 24 -> 74
7. 78
8. 58
9. 62
10. 64
```

Nove hodnoty

```
1:96 2:67 3:84 4:50 5:69 6:74 7:78 8:58 9:62 10:64
```

17.10 Zbývající užitečné funkce ze <stdio.h>

Kromě dříve popsaných funkcí, zajišťujících širokou škálu akcí se soubory, je možné pro práci se soubory použít i několik dosud nezmíněných, ale užitečných funkcí.

17.10.1 Funkce freopen() – přesměrování proudu

Funkční prototyp:

```
FILE *freopen(const char *jmeno, char *rezim, FILE *stream);
```

Funkce otevře soubor s názvem *jmeno* v příslušném režimu³⁵ a přesměruje do něho již existující otevřený *stream*. Původní *stream* je zavřen.

Pokud je původní *stream* standardní *stdin*, *stdout* nebo *stderr*, stává se jméno tohoto standardního vstupu nebo výstupu synonymem pro ukazatel do nově otevřeného souboru.

Funkce při neúspěchu vrací NULL, při úspěchu ukazatel na nově otevřený soubor, který je totožný se třetím formálním parametrem.

Poznámky:

- Rozumné použití je pro přesměrování *stdin*, *stdout* nebo *stderr* do souboru.

³⁵ Formální parametr *rezim* má stejný význam jako u *fopen()* – viz str. 278.

- Pro přesměrování jednoho souboru do druhého je z hlediska přehlednosti vhodnější první soubor uzavřít pomocí *fclose()* a druhý otevřít pomocí *fopen()* při použití stále stejného ukazatele na typ *FILE*.
- Ve Windows je *freopen()* mimořádně vhodná pro přesměrování *stderr*, protože chybový výstup nelze přesměrovat standardním příkazem „>“ z příkazové řádky Windows.

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
FILE *f;
```

```
/* presmerovani stderr do souboru */
```

```
if ((f = freopen("CHYBY.TXT", "w", stderr)) == NULL)
    printf("Chyba v presmerovani stderr\n");
```

```
printf("f = %p stderr = %p\n", f, stderr);
```

```
/* stderr a f jsou synonyma */
```

```
fprintf(stderr, "1) Tato zprava se vypise do souboru.\n");
fprintf(f, "2) Tato zprava se vypise do souboru.\n");
```

```
if (fclose(stderr) == EOF)
    printf("Chyba v uzavreni stderr\n");
```

```
if (fclose(f) == EOF)
    printf("Chyba v uzavreni f\n");
```

```
return 0;
```

```
}
```

Program vypíše na obrazovku:

```
f = 3ffc00804a0 stderr = 3ffc00804a0
Chyba v uzavreni f
```

Obsah souboru CHYBY.TXT:

- 1) Tato zprava se vypise do souboru.
- 2) Tato zprava se vypise do souboru.

17.10.2 Funkce rename() – přejmenování souboru

Funkční prototyp: `int rename(const char *old, const char *new);`

Funkce změní staré jméno souboru na nové. V případě úspěchu vrací nulu, v opačném případě pak nenulovou hodnotu.

Poznámka:

- Přejmenováváný (starý) soubor by měl být uzavřen, jinak je výsledek implementačně závislý.

17.10.3 Funkce remove() – zrušení souboru

Funkční prototyp: `int remove(const char *jmeno_souboru);`

Funkce `remove()` vymaže soubor a vrátí nulu v případě úspěchu a nenulovou hodnotu v opačném případě. Rušený soubor by měl být uzavřený.

17.10.4 Funkce tmpfile() – pomocný soubor

Funkční prototyp: `FILE *tmpfile(void);`

Při práci se soubory často potřebujeme, aby program mohl zapisovat mezivýsledky do pomocného souboru, který se pak na konci programu zruší. Elegantní vytvoření takového dočasného pomocného souboru umožňuje funkce `tmpfile()`, která otevře soubor v režimu `"wb+"36`

Tento soubor je automaticky zrušen okamžitě po svém uzavření funkcí `fclose()`.

Funkce `tmpfile()` vrací NULL v případě, že není možné soubor otevřít, jinak vrací ukazatel na otevřený soubor.

17.10.5 Funkce tmpnam() – jméno pomocného souboru

Funkční prototyp: `char *tmpnam(char *str);`

Nevyhovuje-li nám pomocný soubor vytvořený pomocí funkce `tmpfile()`, protože např. požadujeme otevření v textovém režimu nebo chceme pomocný soubor vymazat sami, pak lze použít funkci `tmpnam()`, která vygeneruje unikátní jméno pro dočasný soubor.

Tento soubor pak lze otevřít v libovolném režimu a po uzavření se automaticky nemaže – je nutné použít explicitně volání funkce `remove()`.

Příklad:

Následující program ukáže trojí možný způsob použití `tmpnam()`.

³⁶ Tedy nový soubor v binárním režimu s možností číst i zapisovat.

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    char jmeno[256], *str;
```

```
    tmpnam(jmeno);
```

```
    printf("1.docasne jmeno: %s\n", jmeno);
```

```
    str = tmpnam(NULL);
```

```
    printf("2.docasne jmeno: %s\n", str);
```

```
    str = tmpnam(jmeno);
```

```
    printf("3.docasne jmeno: %s - %s\n", str, jmeno);
```

```
    return 0;
```

```
}
```

Program vypíše např.:³⁷

```
1.docasne jmeno: /tmp/aaaaajmda
```

```
2.docasne jmeno: /tmp/baaaajmda
```

```
3.docasne jmeno: /tmp/caaaajmda - /tmp/caaaajmda
```

Poznámka:

- Nejzajímavější je druhý způsob, kdy při použití NULL si `tmpnam()` sama alokuje paměť, do které uloží vygenerované unikátní jméno. Na tuto paměť vrátí ukazatel.

17.11 Ukázka použití funkcí z UNIXové knihovny

Jak již bylo konstatováno na začátku, mnoho systémů umožňuje používat i funkce z knihovny C-funkcí pro UNIX.

Poznámka:

- Pokud budete tyto funkce využívat, je třeba nejdříve provést základní pokusy na konkrétní verzi UNIXu či Linuxu, kde jsou dále zmiňované soubory uloženy (občas v adresáři `include/sys`). Dále je třeba vzít v úvahu, že UNIX umožňuje mnohem dokonalejší ochranu souborů nastavením přístupových práv pro čtení, zápis a spuštění, což může činit potíže. Mé pokusy (nejsem odborník na UNIX) vedly ke zjištění, že se situace na různých systémech výrazně odlišuje.

³⁷ Všimněte si, že kromě vytvoření dočasného jména, směřuje UNIX tyto soubory do adresáře `/tmp`.

Pro práci se soubory potřebujeme nejčastěji `.h` soubory `<io.h>` a `<fcntl.h>`.

Jak bude zřejmé z příkladu, můžeme v podstatě použít tytéž funkce jako v ANSI C, s velmi podobnými parametry, kterým v názvu chybí první písmeno „f“³⁸ – např. `fopen()` versus `open()`.

Toto je významné zjištění, protože v případě nutnosti použití nebufferovaných V/V operací můžeme využívat téměř všechny dosavadní znalosti.

Poznámka:

- Obecně by mělo platit, že nebufferované V/V jsou rychlejší pro větší objem dat zpracovávaných najednou. Moje jednoduché praktické pokusy to potvrzují – při pokusu o načtení souboru dlouhého asi 112 MB bylo nebufferované čtení (`read()`) asi dvakrát rychlejší než bufferované (`fread()`).

Příklad:

Bude uveden modifikovaný program ze str. 301 pro přímý přístup do binárního souboru, ze kterého je vidět, že se práce s bufferovanými V/V pomocí `streamů` a nebufferovaných V/V pomocí `handle` příliš neliší.

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>

int main(void)
{
    int handle;
    int i = 10;
    double d = 3.14159;

    handle = open("POKUS2.DAT", O_CREAT | O_RDWR);
    write(handle, &i, sizeof(i)); /* zapis dat do souboru */
    write(handle, &d, sizeof(d));
    close(handle);

    i = 0; d = 0.0; /* nulovani promennych */
    handle = open("POKUS2.DAT", O_RDONLY);
    read(handle, &i, sizeof(i)); /* ctení a zobrazení dat */
    read(handle, &d, sizeof(d));
    close(handle);
    printf("Načtena data: i = %d, d = %f\n", i, d);
    return 0;
}
```

³⁸ Pozor na skutečnost, že uvedené pravidlo neplatí absolutně!

17.12 Zjištění informací o položkách v adresáři

Jako velmi častá úloha při práci se soubory je nutnost pracovat se soubory (a adresáři), které sice v okamžiku spuštění programu leží na disku, ale v době vytváření programu nebyla jejich jména k dispozici. Typická úloha zní „zpracujte všechny soubory v daném adresáři“.

Poznámky:

- Knihovní funkce popisované v této části nejsou součástí standardní ANSI C knihovny. Je proto možné, že na některých překladačích nebudete moci dále uvedené příklady spustit, případně bude nutné provést několik pokusů zejména co se týče přesného umístění dále zmiňovaných hlavičkových souborů.
- Je třeba si uvědomit, že se pohybujeme mimo oblast standardních knihoven ANSI C. Z tohoto důvodu je nutné vypnout (případně nepoužívat) nastavení kompilátoru, které dodržení ANSI C vyžaduje.
- V této části budou ukázány postupy, jak zjistit o souborech jejich jména, velikosti a časy poslední změny. Bohužel tyto postupy jsou rozdílné pro překladače typu gcc (nejčastěji v prostředí UNIXu) a překladače určené jen pod Windows (např. Visual C). Z tohoto důvodu bude výklad proveden pro každé prostředí zvlášť.

17.12.1 Prostředí UNIXu

Poznámky:

- Zjistit všechny možné informace není zcela triviální záležitost. Proto si tuto úlohu rozdělíme na více částí.
- Pod pojmem „položka adresáře“ je dále míněn buď soubor nebo adresář.

Zjištění pouze jmen položek

Získat jména souborů a podadresářů znamená použít konstanty, datové struktury a funkce popsané v `<dirent.h>`.

Pozor:

- Protože jsme mimo oblast ANSI C, není jméno souboru `<dirent.h>` a jeho umístění jednoznačné. Na některých překladačích může být nutné inkudovat

```
#include <sys/dirent.h>
```

nebo dokonce:

```
#include <sys/dir.h>
```

Z knihovny popsané tímto souborem musíme využít datové struktury `DIR` a `struct dirent` (*directory entry* – položka adresáře).

Poznámka:

- Jejich obecné použití může být složité, protože některé operační systémy mají pomocí nastavení přístupových práv možnost zakázat do adresáře přístup nebo i k souborům přístup.

Příklad:

Nejjednodušší případ nastává, pohybujeme-li se v aktuálním adresáři, ve kterém máme všechna přístupová práva. Adresář je nejprve nutno funkcí `opendir()` otevřít. Pointer na datovou strukturu pouze slouží jako parametr funkce `readdir()`, která pro každou položku adresáře vytvoří strukturu typu `dirent`. Její položka `d_name` je ukazatel na řetězec obsahující jméno položky. Na úrovni struktury `dirent` nelze rozlišit typ (adresář/soubor) položky.

```
#include <stdio.h>
#include <dirent.h>
#define AKTUALNI_ADRESAR  "."
```

```
int main(void)
{
    DIR          *p_dir;
    struct dirent *p_dirent;

    p_dir = opendir(AKTUALNI_ADRESAR);
    while ((p_dirent = readdir(p_dir)) != NULL) {
        printf("%s\n", p_dirent->d_name);
    }
    closedir(p_dir); /* nepovinne */
    return 0;
}
```

Poznámka:

- Závěrečné zavření adresáře pomocí `closedir()` není v případě, že pracujeme pouze s jedním adresářem, povinné.

Aby byl výčet používaných funkcí z `<dirent.h>` úplný, je třeba zmínit ještě funkci `void rewinddir(DIR *)`. Po použití této funkce je možné již otevřený adresář opět prohledat po jednotlivých položkách. Příklad bude uveden dále.

Příklad:

Chceme-li pracovat i s jiným adresářem, než s aktuálním, je nutné počítat s tím, že buď neexistuje (spletli jsme se v zadání cesty) nebo k němu nemáme přístupová práva. Proto je nutné vždy testovat návratovou hodnotu `opendir()` na `NULL`.

```
p_dir = opendir(argv[1]);
if (p_dir == NULL) {
    printf("Adresar '%s' neexistuje nebo "
           "nemame pristupova prava na otevreni\n",
           argv[1]);
    return 1;
}
```

Zjištění typu a stavu položek

Průslušné konstanty, datové struktury a funkce jsou v `<sys/stat.h>`. Zde je podstatná datová struktura `struct stat`, kterou pro dané jméno položky nastaví funkce `int stat(char *jmeno, struct stat *stav)`. V případě neúspěchu vrací funkce `stat()` záporné číslo.

Ve struktuře `stat` jsou nejdůležitější položky:

`st_size` – udává velikost souboru v bajtech
`st_mtime` – typu `time_t` (viz str. 353), udávající čas poslední modifikace souboru
`st_mode` – pomocí ní lze rozlišit soubor od adresáře – symbolická konstanta `S_IFDIR`.

V proměnné `st_mode` jsou mimo informace o adresáři uložena také přístupová práva. Symbolické konstanty pro tři nejznámější přístupová práva jsou:

`S_IREAD` – položku je možné číst
`S_IWRITE` – položku je možné vytvořit nebo modifikovat
`S_IXEXEC` – pokud je položka program nebo skript, máme právo jej spustit

Pozor:

- Položka `st_mtime` by se neměla využívat pro adresáře.

Příklad:

Ukázka rozlišení adresářů a souborů, kdy se nejdříve vypíše všechny adresáře a po `rewindir()` i všechny soubory. Pozor na skutečnost, že pokud nezpracováváme položky z aktuálního adresáře, je třeba zadávat jméno položky společně se jménem zkoumaného adresáře. Z tohoto důvodu se používá dostatečně velký řetězec `plne_jmeno`, v němž se vytvoří požadovaný řetězec. Test `(p_dirent->d_name[0] != '.')` odstraňuje z výpisu adresáře „.“ a „..“. Na ty je třeba dát zvláštní pozor při případném rekurzivním výpisu všech podadresářů.

```
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    DIR          *p_dir;
    struct dirent *p_dirent;
    struct stat   s_buf;
    char          plne_jmeno[1000];

    p_dir = opendir(argv[1]);
    if (p_dir == NULL) {
        printf("Adresar '%s' neexistuje\n", argv[1]);
        return 1;
    }

    printf("Podadresare:\n");
    while ((p_dirent = readdir(p_dir)) != NULL) {
        sprintf(plne_jmeno, "%s/%s", argv[1], p_dirent->d_name);
        stat(plne_jmeno, &s_buf);
        if ((s_buf.st_mode & S_IFDIR) != 0) {
            if (p_dirent->d_name[0] != '.')
                printf("%s\n", p_dirent->d_name);
        }
    }

    rewinddir(p_dir);
```

```
printf("\nSoubory:\n");
while ((p_dirent = readdir(p_dir)) != NULL) {
    sprintf(plne_jmeno, "%s/%s", argv[1], p_dirent->d_name);
    stat(plne_jmeno, &s_buf);
    if ((s_buf.st_mode & S_IFDIR) == 0) {
        printf("%s\n", p_dirent->d_name);
    }
}

return 0;
}
```

Příklad:

Ukázka výpisu běžně využívaných informací o souborech – jména, velikosti a času poslední modifikace. Pro práci s časem využíváme `<time.h>` (viz podrobně str. 356).

```
#include <stdio.h>
#include <time.h>
#include <dirent.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    DIR          *p_dir;
    struct dirent *p_dirent;
    struct stat   s_buf;
    char          plne_jmeno[1000];
    struct tm     tm;

    p_dir = opendir(argv[1]);
    if (p_dir == NULL) {
        printf("Adresar '%s' neexistuje\n", argv[1]);
        return 1;
    }

    while ((p_dirent = readdir(p_dir)) != NULL) {
        sprintf(plne_jmeno, "%s/%s", argv[1], p_dirent->d_name);
        stat(plne_jmeno, &s_buf);
        if ((s_buf.st_mode & S_IFDIR) != 0)
            continue;
        /* adresar - prechod na dalsi polozku */
```

```

printf("%s, ", p_dirent->d_name);
printf("%lu, ", s_buf.st_size);
tm = *localtime(&s_buf.st_mtime);
printf("%02d.%02d.%4d\n", tm.tm_mday, tm.tm_mon + 1,
                        tm.tm_year + 1900);
}

return 0;
}

```

Poznámka:

- Můžete si vyzkoušet program pro rekurzivní výpis všech souborů a podadresářů z daného adresáře.

17.12.2 Prostředí Windows

Zde je situace o poznání jednodušší, protože potřebujeme pouze soubor `<io.h>`. V něm je popsána struktura `struct _finddata_t` se čtyřmi nejdůležitějšími položkami:

<code>size</code>	– typu <code>unsigned long</code> , udává velikost souboru v bajtech,
<code>time_write</code>	– typu <code>time_t</code> , udávající čas poslední modifikace souboru,
<code>name</code>	– typu pole <code>char</code> – jméno souboru,
<code>attrib</code>	– typu <code>unsigned int</code> – pomocí ní lze rozlišit soubor od adresáře
	– symbolická konstanta <code>_A_SUBDIR</code> .

Strukturu `_finddata_t` plní dvě funkce popsané též v `<io.h>`.

První je:

`intptr_t _findfirst(char *maska, struct _finddata_t *p_s)`, jejíž první parametr je jméno hledaného souboru, nebo mnohem častěji maska s využitím zástupných znaků „*“ a/nebo „?“. Součástí prvního parametru může být i relativní nebo absolutní cesta, a co je zvlášť zajímavé, jako oddělovače adresářů může být použit jak znak „\“, tak i „/“. Druhým parametrem je pointer na strukturu `_finddata_t`. Funkce `_findfirst()` vrací `-1`, pokud nastala nějaká chyba (nejčastěji nesprávné jméno zkoumaného adresáře) nebo pokud žádný soubor dané masce nevyhovuje. Návrátová hodnota³⁹ je důležitá a používá se v dalším zpracování.

Tato funkce „nastartuje“ prohledávání adresáře, ale pro vlastní činnost, tj. zjišťování informací o jednotlivých položkách, se používá další funkce `_findnext()`.

³⁹ Ačkoliv je typu `intptr_t`, běžně se používá typ `int`.

Druhá je funkce:

`intptr_t _findnext(intptr_t vysledek, struct _finddata_t *p_s)`, která má jako svůj první parametr hodnotu výsledku, kterou jako svoji návratovou hodnotu vrací funkce `_findfirst()`. Druhý parametr má stejný význam, jako u `_findfirst()`.

Funkce `_findnext()` pracuje v součinnosti s nastavením provedeným pomocí `_findfirst()`, což znamená, že podle zadané masky postupně vyplňuje strukturu `_finddata_t` údaji o položkách zkoumaného adresáře. Pokud je nalezena další položka, vrací `_findnext()` hodnotu `0`.

Poslední funkcí, kterou je vhodné při prohledávání adresáře použít, je „uzavírací“ funkce `int _findclose(intptr_t vysledek)`. Její parametr má stejný význam jako u funkce `_findnext()`.

Poznámka:

- Tím, že je ve struktuře `_finddata_t` uvedeno i jméno souboru, se práce oproti UNIXu značně zjednodušuje. Proto bude celá problematika ukázána pouze na jednom příkladě. Ten bude mít shodnou funkci s příkladem na str. 315.

```

#include <stdio.h>
#include <string.h>
#include <io.h>
#include <time.h>

```

```

int main(int argc, char **argv)
{
    struct _finddata_t data;
    struct tm      tm;
    char maska[] = "*. *";
    char adresar[1000];
    intptr_t vysledek;

    if (argc == 1)
        strcpy(adresar, maska);
    else
        sprintf(adresar, "%s/%s", argv[1], maska);

    vysledek = _findfirst(adresar, &data);
    if (vysledek == -1) {
        printf("Zadny soubor nevyhovuje masce '%s'\n", maska);
        return 1;
    }
}

```

```

while (_findnext(vysledek, &data) == 0) {
    if ((data.attrib & _A_SUBDIR) != 0)
        continue;                /* adresar */

    printf("%s, ", data.name);
    printf("%lu, ", data.size);
    tm = *localtime(&data.time_write);
    printf("%02d.%02d.%4d\n", tm.tm_mday, tm.tm_mon + 1,
        tm.tm_year + 1900);
}

_findclose(vysledek);

return 0;
}

```

18 Návaznost překladače ANSI C na okolní prostředí

Jakýkoliv programovací jazyk, má-li být úspěšný, „nevisí ve vzduchu-prázdně“, ale musí nějakým způsobem navazovat na svoje okolí.

ANSI C definuje poměrně širokou kolekci prostředků, které umožňují součinnost překladače s prostředím hostitelského operačního systému.

Splňuje-li překladač tyto požadavky (tím jsou zejména míněny standardní funkce dostupné v knihovnách) a využívá-li náš program těchto prostředků, je pak zaručena velmi vysoká přenositelnost zdrojových souborů mezi nejrůznějšími počítači a operačními systémy.

Poznámka:

- Nikdy nebude zřejmě dosaženo 100% přenositelnosti v tak širokém spektru procesorů a operačních systémů podporujících ANSI C. Ovšem s těmito singulárními případy nepřenositelných programů se zřejmě tak brzy nesetkáme.

18.1 Standardní hlavičkové soubory

V podstatě lze říci, že celá návaznost překladače ANSI C na okolní prostředí je popsána v hlavičkových souborech. Ty jsou nejčastěji uloženy v adresáři `INCLUDE` a jak již víme (viz str. 98), jsou to čistě textové soubory.

V hlavičkových souborech jsou uvedeny funkční prototypy, symbolické konstanty, výčtové typy, makra, atd. většinou vždy pohromadě. Existují však „speciální“ hlavičkové soubory, které obsahují např. jen symbolické konstanty. Tyto soubory budou podrobněji popsány na str. 322.

Každá implementace překladače ANSI C musí obsahovat minimálně následující standardní hlavičkové soubory¹:

<assert.h> – makra používaná při ladění
 <ctype.h> – práce se znaky
 <math.h> – matematické funkce v přesnosti double
 <locale.h> – práce v národním prostředí
 <setjmp.h> – umožnění běžně nedovolených skoků
 <signal.h> – zpracování signálů (využití v multitaskovém OS)
 <stdarg.h> – práce s proměnným počtem parametrů
 <stdio.h> – funkce pro vstup a výstup
 <stdlib.h> – obecně užitečné funkce
 <string.h> – zpracování řetězců
 <time.h> – práce s datem a časem
 <errno.h> – reakce na systémové chyby
 <limits.h> – charakteristiky celých čísel
 <float.h> – charakteristiky reálných čísel
 <stddef.h> – standardní definice

Všechny tyto soubory mají implementačně závislý obsah, ale je přesně stanoveno, že musí obsahovat funkční prototypy konkrétních standardních funkcí a maker, definice obecně platných symbolických konstant a datových typů a deklarace externích proměnných.

Kromě nich mohou obsahovat cokoliv užitečného navíc. Například soubor <stdio.h> musí obsahovat funkční prototyp standardní funkce `fflush()`², ale někdy obsahuje navíc např. i funkční prototyp funkce `flushall()`, která zapíše obsahy všech bufferů.

Standardní hlavičkové soubory jsou připravovány velmi pečlivě. Často však obsahují složitě provázané příkazy preprocesoru pro podmíněný překlad, takže jsou na první pohled poněkud nečitelné.

Díky podmíněnému překladu je ale např. zaručeno, že <.h> soubory mohou být „inkludovány“ do téhož souboru neomezeněkrát bez výskytu chyby. Tato užitečná služba je zajištěna známým trikem (viz např. str. 379), kdy se pro podmíněný překlad definuje symbolická konstanta podobná jménu souboru, např.:

```
#ifndef __STDIO_H
#define __STDIO_H
    vlastní obsah souboru <stdio.h>
#endif
```

¹ Poslední čtyři soubory obsahují zejména symbolické konstanty.

² Zapíše obsah bufferu do souboru – viz str. 299.

Množství prostředků popsaných v těchto souborech – např. standardních funkcí jako `printf()` nebo nových typů jako `FILE` – již umíme využívat. Popis dalších užitečných věcí viz str. 322 a str. 329.

18.2 Rezervovaná jména

Použitím standardních hlavičkových souborů narůstá počet jmen, která buď nesmíme nebo bychom neměli použít pro vlastní identifikátory.

Poznámky:

- Pokud používáme pro své identifikátory češtinu (přesněji „cestinu“), nemáme se prakticky čeho bát. Častým zvykem ale je použití angličtiny, alespoň pro některé identifikátory, a pak mohou nastat dále popsané problémy.
- Ve velké většině případů – kromě požadovaného předefinování standardní funkce – je použití stejně pojmenovaného identifikátoru chybou a je velmi vhodné tento náš konflikt identifikátor pojmenovat trochu jinak. V opačném případě se pravděpodobně dostaneme do problémů.

Rezervovaná jména mohou patřit do některé z následujících skupin:

- 1) **Klíčová slova** – `if`, `int`, `register`, ...
 Tato jména se nesmí použít pro naše identifikátory – vznikne okamžitě chyba překladu.
- 2) **Příkazy preprocesoru** – `#define`, `#include`, ...
 Uvedeme-li znak `#` na začátku našeho identifikátoru, pak tento identifikátor nelze použít pro naše účely – stává se z něj samozřejmě normální příkaz preprocesoru.
 Použijeme-li tento identifikátor bez znaku `#` na začátku, je výsledek implementačně závislý.
- 3) **Jména standardních funkcí a maker** – `printf()`, `islower()`, ...
 Neměli bychom je používat, pokud nechceme napsat vlastní verzi této funkce nebo makra. V tomto případě:
 - Jsou-li různé funkční prototypy, pak nesmíme natáhnout příslušný <.h> soubor, protože překladač hlásí konflikt funkčních prototypů standardní a naší funkce.
 - Jsou-li funkční prototypy shodné, pak naše funkce překryje funkci standardní.³

³ To je vcelku častý případ, kdy se snažíme o přizpůsobení standardní funkce našim potřebám, a chceme, aby volání této funkce zůstalo nezměněno.

- Použijeme-li jméno standardní funkce v jiném smyslu⁴, pak překladač hlásí chybné použití funkce.
- 4) **Jména nových typů** – FILE, ...
Nový typ téhož jména nelze definovat. Pokud je to nutné, nesmíme „inkludovat“ příslušný <.h> soubor.
Použijeme-li proměnnou stejného jména, znepřístupníme daný typ.
- 5) **Jména symbolických konstant** – NULL, EOF, ...
Pokud nepoužijeme příkazu #undef, není možné symbolickou konstantu předdefinovat.
- 6) **Jména externích proměnných** – errno, sys_errlist, ...
Platí totéž, co pro jména funkcí – je-li naše proměnná jiného typu než externí, překladač hlásí chybu, je-li stejného typu, znepřístupní se externí proměnná.

Poznámka:

- Rezervovaných jmen v ANSI C je 260, pokud ale budeme počítat všechny identifikátory (zejména z knihoven), pak dostaneme čísla mnohem větší.

18.3 Standardní hlavičkové soubory bez funkčních prototypů

ANSI C definuje čtyři standardní hlavičkové soubory, které pouze popisují symbolické konstanty a datové typy nutné pro spolupráci operačního systému s kompilátorem C nebo využívané funkcemi z ostatních hlavičkových souborů.

I tyto soubory se výrazně zaslouhují o zajištění přenositelnosti, i když pracují vlastně jen s příkazy #define a typedef.

18.3.1 Soubor <errno.h>

Deklaruje externí proměnnou (podrobně viz str. 284 a str. 326), používanou pro předání informace o chybě.

Dále jsou v souboru uvedeny definice symbolických konstant, jejichž hodnot může errno nabývat.

Příklady některých⁵ symbolických konstant:

```
#define ESUCCESS    0    /* Successful */
#define ENOENT       2    /* No such file or directory */
```

⁴ Například pro označení proměnné: int printf;

⁵ Konstant je asi 70 – přehled všech viz v souboru <errno.h>.

```
#define EINTR        4    /* Interrupted system call */
#define EIO          5    /* I/O error */
#define ENXIO        6    /* No such device or address */
#define EEXIST       17   /* File exists */
#define ENOTDIR      20   /* Not a directory */
#define EISDIR       21   /* Is a directory */
#define EINVAL       22   /* Invalid argument */
#define EMFILE       24   /* Too many open files */
#define EFBIG        27   /* File too large */
#define ESPIPE       29   /* Illegal seek */
#define EROFS        30   /* Read-only file system */

/* math software */
#define EDOM          33   /* Argument too large */
#define ERANGE       34   /* Result too large */
```

18.3.2 Soubor <limits.h>

Obsahuje symbolické konstanty popisující charakteristiky celých čísel.

Kromě dnes poněkud kuriózní konstanty CHAR_BIT představující počet bitů v bajtu zde nalezneme minimální a maximální číslo pro každý celočíselný typ a to jak signed, tak i unsigned, např.:

```
#define INT_MAX      0x7FFF    - max. hodnota pro signed int
#define INT_MIN      ((int)0x8000) - min. hodnota pro signed int
#define UINT_MAX     0xFFFFU   - max. hodnota pro unsigned int
```

Tyto konstanty se dají vhodně využívat pro nejrůznější testování mezí, zjišťování rozsahů jednotlivých typů na konkrétním počítači, atd.

18.3.3 Soubor <float.h>

Tak, jako jsou popsány celočíselné typy v <limits.h>, jsou ve <float.h> uvedeny symbolické konstanty popisující charakteristiky reálných čísel.

Soubor obsahuje množství informací o použitých reálných číslech. Z nich jsou pro běžné případy prakticky využitelné dále uvedené čtyři trojice.

Trojice proto, že jedna vlastnost je vždy uvedena pro všechny tři reálné typy a je odlišena začátkem symbolické konstanty takto:

```
float      FLT_
double     DBL_
long double LDBL_
```

Zmíněné čtyři důležité vlastnosti reálných čísel jsou:

1. minimální hodnota – nejmenší možné číslo
FLT_MIN 1e-37
2. maximální hodnota – největší možné číslo
FLT_MAX 1e+37
3. epsilon – nejmenší číslo, pro něž platí: $1.0 + \epsilon \neq 1.0$
FLT_EPSILON 1e-5
4. přesnost – počet platných desítkových číslic mantisy
FLT_DIG 6

V souboru jsou dále uvedeny konstanty určující způsob uložení reálného čísla v paměti a jeho zobrazení, což jsou údaje, které se běžně nevyužívají. Zde alespoň pro zajímavost uvedme, že se jedná např. o:

- počet bitů mantisy⁶ – FLT_MANT_DIG
- min. a max. dvojkový exponent – FLT_MIN_EXP a FLT_MAX_EXP
- min. a max. desítkový exponent – FLT_MIN_10_EXP a FLT_MAX_10_EXP

a podobné konstanty.

18.3.4 Soubor <stddef.h>

Tento soubor obsahuje definice dvou nových typů, symbolické konstanty NULL a někdy další⁷ deklaraci proměnné `errno`.

Nové typy a symbolická konstanta jsou:

<code>size_t</code>	Definice beznaménkového celého čísla. Tento typ je často používán ve formálních parametrech funkcí pro čísla ve významu „počet“ (viz např. str. 343). Na různých platformách má různou velikost – 2, 4 nebo 8 bajtů.
<code>ptrdiff_t</code>	Definice celočíselného typu, který poskytne rozdíl dvou pointerů.
<code>NULL</code>	Definice hodnoty nulového pointeru.

Poznámka:

- Duplicitní definici uvedených položek můžete ale nalézt i v jiných standardních hlavičkových souborech. Například <stdio.h> většinou také definuje typ NULL, i když „čistější“ řešení by bylo, kdyby vnitřně „inkludoval“ <stddef.h>.

⁶ Včetně skrytého bitu.

⁷ První obsahuje <errno.h> – viz str. 322.

18.4 Předdefinovaná makra preprocesoru

Tato makra se někdy označují jako „předdefinované globální identifikátory“.

Každá ANSI implementace obsahuje pět maker:⁸

__STDC__ __FILE__ __LINE__ __TIME__ __DATE__

18.4.1 Makro __STDC__

Má-li toto makro hodnotu 1, pak překladač vyhovuje normě ANSI C. Při jiné hodnotě – nejčastěji 0 – nebo spíše při neexistenci tohoto makra (chybu ohlásí překladač) je nutno být při programování obezřetnější.

V případě, že __STDC__ nenabývá hodnoty 1, není třeba zoufat, ale je nutné vyzkoušet v praxi možnosti překladače, protože např. překladač označovaný jako UNIX V⁹ podporuje téměř totéž, co ANSI C.

Poznámka:

- U některých překladačů je norma ANSI C otázkou jen určité volby přepínačů. Například často používaný překladač gcc¹⁰ implicitně pracuje jako UNIX V, ale s použitím přepínače `-ansi` splňuje normu ANSI C.

Další čtyři makra se vždy vztahují ke zdrojovému souboru, ve kterém jsou uvedena.

18.4.2 Makro __FILE__

Makro se při překladu rozvine do řetězce, obsahujícího jméno překládaného souboru.

18.4.3 Makro __LINE__

Makro poskytne hodnotu typu `int`, udávající číslo řádku, na kterém je makro __LINE__ ve zdrojovém souboru uvedeno.¹¹

18.4.4 Makro __DATE__

Makro přečte ze systému datum, kdy byl daný soubor překládán, a zkonvertuje ho na řetězec. To je výhodné např. dáváme-li někam do programu informaci o době jeho vzniku.

⁸ V identifikátoru jsou vždy dvě podtržítka „_“ před a za.

⁹ V UNIXovém prostředí nejčastěji cc nebo gcc.

¹⁰ Z projektu GNU – viz též str. 398.

¹¹ Výhodné použití __FILE__ a __LINE__ viz např. makro `assert()` – str. 389.

18.4.5 Makro `__TIME__`

Toto makro pracuje jako `__DATE__`, ale poskytuje čas, kdy začal být zdrojový soubor překládán. Čas je opět ve formě řetězce.

Poznámka:

- Doporučuje se nezkoušet uvedená čtyři makra předefinovat příkazem `#define` ani rušit jejich definici příkazem `#undef`.

18.5 Externí proměnné

ANSI C definuje několik externích proměnných, deklarovaných¹² v některých standardních `<.h>` souborech. Tyto proměnné mohou být využívány k nejrůznějším účelům, nejčastěji je to však pro potřeby standardních funkcí.

Poznámka:

- Dále uváděná jména proměnných jsou implementačně závislá, zde uvedena jsou z gcc. V případě, že je nebudete moci pod těmito jmény nalézt, hledejte podobná jména. Například u některých kompilátorů mají některé z těchto proměnných na začátku znak „_“, tedy např. `_sys_errlist` místo `sys_errlist`.

Dále bude uveden přehled jen některých zajímavých, v praxi využitelných proměnných.

18.5.1 Proměnná `errno` z `<errno.h>`

Deklarace: `extern int errno;`

Do této proměnné je při každém chybovém stavu uložen kód chyby. Toto číslo je indexem do pole chybových hlášení `sys_errlist` a příslušné chybové hlášení se dá vypsát pomocí funkce `perror()`.

Má-li `errno` hodnotu 0 (nula) znamená to „bez chyb“, hodnoty 1 až `sys_nerr` znamenají chybu. Pro nejrůznější testy si není nutné pamatovat tato čísla, ale můžeme využít symbolických konstant popsanych v `<errno.h>` – viz str. 322.

Poznámky:

- Proměnná `errno` se dá využít pro test správnosti – viz str. 284.
- Deklaraci `errno` lze občas též nalézt ve `<stddef.h>` či ve `<stdlib.h>`.

¹² Definovány (tzn. vyhražena paměť) jsou v příslušných knihovnách.

18.5.2 Proměnná `sys_errlist` ze `<stdlib.h>`

Deklarace: `char *sys_errlist[ ];`

Je to jméno pole ukazatelů na řetězce chybových zpráv, přičemž indexem do tohoto pole je `errno`. Počet chybových zpráv, tj. velikost `sys_errlist`, je v proměnné `sys_nerr`.

Příslušné chybové hlášení lze vypsát několika způsoby:

- Pomocí funkce `perror()`, která dovoluje přidat i vlastní komentář.
- Text chybového hlášení lze získat pomocí `strerror()` a pak jej libovolně zpracovávat jako řetězec.
- Do `sys_errlist` lze přistoupit i přímo, např. použít funkci `strcpy()`.

Poznámka:

- Zajímavé využití `sys_errlist` může být např. pro překlad chybových zpráv do češtiny – viz příklad.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(void)
{
    errno = ENOENT; /* symbolická konstanta z <errno.h> */
    perror("Anglicky");

    sys_errlist[ENOENT] = "Tento soubor nebo adresar neexistuje";

    errno = ENOENT;
    perror("Česky");
    printf("CHYBA - %s", strerror(ENOENT));
    return 0;
}
```

18.5.3 Proměnná `sys_nerr` ze `<stdlib.h>`

Deklarace: `extern int sys_nerr;`

Proměnná udává celkový počet chybových hlášení v `sys_errlist` neboli velikost pole `sys_errlist`.

18.5.4 Proměnná `environ` ze `<stdlib.h>`

Deklarace: `extern char **environ;`

Je to ukazatel na „*environment*“, čili adresu paměti, kde jsou uloženy systémové proměnné, např. `PATH`, `PROMPT`, ...

Systémové proměnné jsou uloženy jako řetězce ve tvaru: `jmeno=hodnota'\0'`

Poznámka:

- Z bezpečnostních důvodů se na některých systémech deklarace `environ` vůbec nevyskytuje. Proto je vhodné zjišťovat hodnoty jednotlivých systémových proměnných pouze pomocí standardní funkce `char *getenv(char *jmeno)`.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("%s\n", getenv("PATH"));
    return 0;
}
```

Poznámka:

- Jedná se o **kopie** systémových proměnných, takže případná sebeopatrnější změna¹³ se nepromítne do trvalého nastavení systémových proměnných v operačním systému – je provedena jen v rámci aktuálně spuštěného programu. Pokus měnit systémové proměnné z programu tímto způsobem je dosti častý a velmi špatně zjištělný omyl.

18.5.5 Proměnná `_iob` ze `<stdio.h>`

Deklarace: `extern FILE _iob[ ];`

Je to jiné pole struktur typu `FILE` popisující datové prvky pro práci se soubory. Pokud otevřeme soubor, pak se nějaký volný prvek v tomto poli naplní příslušnými hodnotami a je na něj vrácen ukazatel.

Poznámky:

- První tři prvky tohoto pole jsou vždy přiřazeny pro `stdin`, `stdout` a `stderr`.
- Jméno pole `_iob` je platformově závislé a nemělo by se tedy používat.

¹³ `getenv()` vrací pointer na řetězec s hodnotou a ten principiálně lze změnit.

19 Popis funkcí a maker ze standardních knihoven

Množství funkcí a maker již bylo popsáno v předchozích kapitolách. Zde budou vysvětleny jen zbývající funkce a makra. Roztrženy budou podle toho, v jakém standardním hlavičkovém souboru leží jejich funkční prototyp, respektive definice maker.

Poznámka:

- Podrobný přehled všech funkcí není účelné uvádět, protože existuje v popisech knihoven a elektronických helpech. Pro přehled, co tam máme hledat, bude na konci popisu každého hlavičkového souboru uveden výčet všech standardních funkcí a maker s nezbytným, velmi stručným komentářem. U každé funkce nebo makra bude také uvedeno číslo strany, na které je v této publikaci popsáno nebo použito.

19.1 `<assert.h>` – makro používané při ladění

Podrobný popis viz str. 385.

- Soubor `<assert.h>` popisuje jedno následující makro:
`assert()` přidání ladicích informací do programu 389

19.2 `<ctype.h>` – práce se znaky

Podrobný popis viz str. 96.

- Soubor `<ctype.h>` popisuje následující funkce a makra:
- | | | |
|------------------------|------------------------------------|----|
| <code>isalnum()</code> | číslice nebo malá či velká písmena | 96 |
| <code>isalpha()</code> | malá či velká písmena | 96 |
| <code>iscntrl()</code> | řídící znaky | 96 |
| <code>isdigit()</code> | číslice | 96 |
| <code>isgraph()</code> | znaky s grafickou podobou | 96 |
| <code>islower()</code> | malá písmena | 96 |
| <code>isprint()</code> | tisknutelné znaky | 96 |
| <code>ispunct()</code> | interpunkční znaky | 96 |
| <code>isspace()</code> | bílé znaky (mezery, ...) | 96 |

isupper()	velká písmena	96
isxdigit()	hexadecimální číslice	96
tolower()	konverze velkých písmen na malá	96
toupper()	konverze malých písmen na velká	96

19.3 <math.h> – matematické funkce v přesnosti double

Jména matematických funkcí jsou velmi významová¹, takže s běžnými funkcemi pravděpodobně nebudou problémy.

Důležité informace využitelné pro celou skupinu těchto funkcí jsou dvě:

- 1) Parametry funkcí a návratové hodnoty jsou zásadně typu **double**.
- 2) Pokud se v těchto funkcích vyskytne chyba, pak je do **errno** dosazena jedna z těchto symbolických konstant:

EDOM	Vstupní argument (parametr) má nelegální hodnotu mimo definiční obor funkce, např. záporné číslo v <code>sqrt()</code> .
ERANGE	Výstupní parametr je příliš velký (přetečení) nebo malý (podtečení), než aby se vešel do rozsahu typu double . Při přetečení je vrácena symbolická konstanta HUGE_VAL – max. číslo v double . Při podtečení je vrácena hodnota 0.0.

Příklad:

Na následujícím výseku programu je ukázáno použití některých méně zřejmých funkcí. V příkladu je hned vedle výpisových funkcí ukázka vypsaných hodnot oddělená `//`.

```
int i;
double g, f;
printf("%f\n", ceil(3.14));           // 4.000000
printf("%f\n", floor(3.14));          // 3.000000

f = frexp(3.14, &i);
printf("%f x 2^%d\n", f, i);         // 0.785000 x 2^2

f = modf(-3.14, &g);
printf("%f + %f\n", g, f);           // -3.000000 + -0.140000

printf("%f\n", fmod(3.14, 2.0));      // 1.140000
```

¹ Funkce `sin()` počítá *sinus*, atd. :-)

■ Soubor `<math.h>` popisuje následující funkce:

<code>sin(x)</code>	sinus x	
<code>cos(x)</code>	kosinus x	
<code>tan(x)</code>	tangents x	
<code>asin(x)</code>	arkussinus x	
<code>acos(x)</code>	arkuskosinus x	
<code>atan(x)</code>	arkustangents x	
<code>atan2(x, y)</code>	arkustangents y/x	
<code>sinh(x)</code>	hyperbolický sinus x	
<code>cosh(x)</code>	hyperbolický kosinus x	
<code>tanh(x)</code>	hyperbolický tangents x	
<code>exp(x)</code>	e^x	
<code>log(x)</code>	přirozený logaritmus x	
<code>log10(x)</code>	desítkový logaritmus x	
<code>pow(x, y)</code>	x^y	
<code>sqrt(x)</code>	$\sqrt{x}$	
<code>ceil(x)</code>	nejmenší celé číslo, ne menší než x	331
<code>floor(x)</code>	největší celé číslo, ne větší než x	331
<code>fabs(x)</code>	absolutní hodnota x	
<code>ldexp(x, n)</code>	$x \cdot 2^n$	
<code>frexp(x, exp)</code>	rozdělí x na normaliz. mantisu a mocninu dvou	331
<code>modf(x, n)</code>	rozdělí x na celou (n) a desetinnou část	331
<code>fmod(x, y)</code>	reálný zbytek po dělení x/y	331

19.4 <locale.h> – přizpůsobení C národnímu prostředí

Pracujeme-li v národním prostředí, pak máme pravděpodobně s mnoha věcmi problémy, např.:

- Rozšířená znaková sada. Například „Ů“ je velké písmeno, ale `isupper()` ho určitě nerozezná.
- Různý oddělovač celé a desetinné části, např. anglo-americká desetinná tečka versus naše desetinná čárka.
- Různý způsob zápisu času a data, např. 10.6.1995 nebo 6-10-1995.

Tyto problémy by měla vyřešit funkce `setlocale()`. Ta za běhu programu nastaví příznak, že se národní prostředí změnilo a jakým způsobem. Ostatní funkce, které mají s národním prostředím co do činění, budou podle nového prostředí reagovat.

Poznámka:

- Je to sice krásná idea, ale ANSI C definuje pouze jedno prostředí (kupodivu americké ;-)) a ostatní nechává na implementacích. Při bližším zkoumání pak často zjistíte, že překladač splňuje požadavky právě jen tohoto jednoho prostředí.

V každém případě se ale vyplatí přečíst si manuál od konkrétního překladače, protože se dá očekávat, že se situace v tomto směru bude zlepšovat – tzn. postupně budou přidávány podpory jednotlivých národních prostředí.

- Soubor `<locale.h>` popisuje následující funkce:

`setlocale()` nastavuje specifikace pro národní prostředí
`localeconv()` vrací informaci o nastaveném národním prostředí

19.5 `<setjmp.h>` – umožnění běžně nedovolených skoků

Většina programovacích jazyků umožňuje sice použít příkaz typu **goto**, ale s mnoha omezeními.² Omezení se většinou váží na tzv. „nelokální skoky“ či „dlouhé skoky“, které představují např. skok z jedné funkce do druhé, což je ale omezení vcelku pochopitelné.³

Zmíněná omezení mají za úkol zabránit programátorovi v provádění nebezpečných akcí (z hlediska kompilátoru), jejichž konečný dosah může být ve svém důsledku destruktivní.

V praxi se však občas setkáváme s požadavkem provést odskok – diplomatically řečeno – „ne podle pravidel strukturovaného programování“. Pro splnění tohoto požadavku slouží v C funkce a datové typy z `<setjmp.h>`, které umožňují i tuto – assembleru velmi podobnou a nebezpečnou akci – provést relativně⁴ bezpečně a přenositelně.

Princip je v tom, že funkce `setjmp()` při svém prvním (inicializačním) volání umožní uložit aktuální obsah všech potřebných registrů procesoru (*stack pointer, program counter, ...*) do proměnné datového typu `jmp_buf`. Jako informaci o proběhlé inicializaci vrací nulovou hodnotu.

² Výjimkou je jazyk assembleru, kde lze skákat odkudkoliv kamkoliv.

³ U některých programovacích jazyků jsou restriktce ještě větší, takže není například umožněn skok do cyklu atd.

⁴ Slovo „relativně“ znamená, že například není zaručeno obnovení všech automatických proměnných.

Po volání funkce `longjmp()` jsou pak obsahy všech registrů obnoveny z této proměnné a je tak proveden návrat zpět do místa volání `setjmp()`. Funkce `setjmp()` však pozná, že není volána poprvé a vrátí nenulový návratový kód, podle kterého lze provést větvení programu.

Z výše uvedených faktů vyplývá, že `longjmp()` musí být volána až po `setjmp()`, což není nic nepochopitelného, protože aby se mohla vrátit, musí nejdříve vědět kam.

Poznámky:

- Možnost takového skoku by měla být použita pouze v nezbytných případech a nemělo by se počítat s tím, že všechny automatické proměnné budou mít tentýž obsah.
- Typicky se tento mechanismus používá jako jednotná reakce na chybu s následným návratem do místa, kde program ještě fungoval správně.
- Další časté použití – zvláště v prostředí UNIXu – je pro návrat z obsluhy signálu – viz str. 334.

Příklad:

Zde je ukázán princip jednotné reakce na chybu s následným restartem programu.

```
#include <stdio.h>
#include <setjmp.h>

void chyba(void);          /* funkční prototyp */
jmp_buf hlavni_smyčka;     /* globalní proměnná */
int main(void)
{
    int i, j;

    printf("Zacatek programu\n");

    if (setjmp(hlavni_smyčka) != 0)
        printf("Restart po chybě ...\n");

    while (1) {
        printf("Zadej dve cisla (0 0 = konec): ");
        scanf("%d %d", &i, &j);
        if (i == 0)
            break;
        if (j != 0)
            printf("%d / %d = %d\n", i, j, i / j);
    }
}
```

```

    else
        chyba();
}
return 0;
}

void chyba(void)
{
    printf("Chyba - nulovy delitel\n\n");
    longjmp(hlavni_smycka, 1);
}

```

Poznámky:

- Funkce `longjmp()` se nemusí vracet na jedno místo – vrací se podle obsahu svého prvního parametru⁵. Tím je umožněno lehce provést návraty na libovolné místo, které záleží jen na místě výskytu funkcí `setjmp()`. Funkce `chyba()` by pak vypadala např. takto:

```

void chyba(jmp_buf kam_se_vratit, char *chybova_zprava)
{
    printf("Chyba: %s\n", chybova_zprava);
    longjmp(kam_se_vratit, 1);
}

```

- Druhý parametr `longjmp()` je vlastně návratová hodnota příštího volání `setjmp()`, což lze využít k dalšímu předání informace – např. kolikrát byl již tento skok proveden⁶.
- Soubor `<setjmp.h>` popisuje následující funkce:
 - `setjmp()` připraví podmínky a návratový bod pro skok 332
 - `longjmp()` provede nelokální skok 332

19.6 <signal.h> – zpracování signálů

Praktické použití funkcí popsaných v tomto hlavičkovém souboru je ve víceúlohovém operačním systému, jakým je např. UNIX. Zde spolu jednotlivé procesy (úlohy, *tasks*) mohou komunikovat pomocí signálů.

Knihovní funkce ze `<signal.h>` umí signály vysílat a zpracovávat, např. zavést vlastní obsluhu signálu.

⁵ Zde globální proměnná `hlavni_smycka`.

⁶ Viz funkci `obsluha_signalu()` na str. 335.

Poznámky:

- V hlavičkovém souboru `<signal.h>` je popsána pouze omezená množina funkcí pracujících se signály z UNIXovské knihovny⁷ a manuály uvádějí, že není zaručeno, že každá implementace bude schopna tyto signály využívat. Je tedy vhodné seznámit se s detaily u konkrétní aplikace.
- Signály lze ale využít i v prostředí Windows, například pro zabezpečení reakce na stisk `Ctrl-C` nebo na *run/time* chybu. Příklady viz v příložených souborech.

Příklad:

Uvedený program by měl sloužit spíše jako malá ukázka možností a postupů jednak obsluhy signálů a pak také spolupráce signálů a dlouhých skoků. Program, který byl odladěn v UNIXu, v nekonečné smyčce neustále vypisuje znak `a`. Pokud by běh programu nebylo možné ovlivnit zvenčí, program by neustále vykazoval stejnou činnost. Zde je však možné ovlivnění zvenčí zasláním signálu `SIGQUIT`. Po jeho obdržení se vyvolá obsluha signálu, která změnu vypisovaného znaku. Pokud je zvolen způsob návratu ze signálu bez použití `longjmp()`, je návrat z obsluhy proveden zpět do nekonečné smyčky, kde se ovšem vypisuje změněný znak (`a`). Činnost programu lze ukončit např. stiskem `Ctrl-C`.

Je-li zvolen způsob návratu pomocí `longjmp()` (viz str. 332), řízení se vrací na místo volání funkce `setjmp()` a program je ukončen.

Poznámky k příkladu:

- Funkce `obsluha_signalu()` musí mít jeden parametr typu `int` (zde pojmenován `nevyuzity_parametr`), neboť to vyžaduje funkce `signal()`, která zavádí novou obsluhu signálu.
- Pokud budete tento program zkoušet, spusťte jej na pozadí nebo z druhého okénka, aby bylo možné zasílat z klávesnice příslušný signál.

```

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

```

```

#define VYNECH 10000000 /* znak se vypisuje jednou za VYNECH */

jmp_buf navrat;
void (* stara_adresa)(int);

```

⁷ Prakticky jen funkce `signal()` a `raise()` a symbolické konstanty pro některé signály.

```

char vypisovany_znak = 'a';
int skoncit_program = 0;

void obsluha_signalu(int nevyuzity_parametr)
{
    vypisovany_znak++;

    if (skoncit_program == 1)
        longjmp(navrat, 1);
}

int main(void)
{
    unsigned long int i = 0;

    printf("Zadej zpusob chovani po navratu ze signalu\n"
           "0=pokracovat, 1=skoncit program: ");
    scanf("%d", &skoncit_program);

    /* zavedeni nove adresy obsluhy SIGQUIT */
    stara_adresa = signal(SIGQUIT, obsluha_signalu);

    printf("\nZadej prikaz: kill -%d %d\n", SIGQUIT, getpid());

    if (setjmp(navrat) == 0) {
        while (1) {
            if (++i % VYNECH == 0) {
                putchar(vypisovany_znak);
                fflush(stdout);
            }
        }
    }
    else {
        printf("\nProgram byl ukoncen signalem...\n");
    }

    /* obnoveni puvodni adresy obsluhy SIGQUIT */
    signal(SIGQUIT, stara_adresa);
    return 0;
}

```

Vypiše:

```

$ signaly
Zadej zpusob chovani po navratu ze signalu
0=pokracovat, 1=skoncit program: 0
Zadej prikaz: kill -3 3728
aaaaaaabbbbbbbccccccccccddddddeeeeeeeeeeeeeeeeeee

```

```

$ signaly
Zadej zpusob chovani po navratu ze signalu
0=pokracovat, 1=skoncit program: 1
Zadej prikaz: kill -3 3528
aaaaaaaaaaaaaaaa
Program byl ukoncen signalem...

```

- Soubor <signal.h> popisuje následující funkce:
 signal() zavede novou obsluhu signálu 335
 raise() vyšle signál programu

19.7 <stdarg.h> – práce s proměnným počtem parametrů

Soubor obsahuje makra výhodná pro tvorbu funkcí s proměnným počtem parametrů. Podrobně viz samostatnou kapitolu na str. 365.

- Soubor <stdarg.h> popisuje následující makra:
 va_start() inicializuje seznam proměnných parametrů 365
 va_arg() poskytne hodnotu další položky seznamu 365
 va_end() ukončí práci se seznamem 365

19.8 <stdio.h> – funkce pro vstup a výstup

Zde jsou popsány funkce a makra umožňující provádět standardní vstupy a výstupy. Naprostou většinu těchto funkcí již důvěrně známe (viz např. str. 39, 69, 202) nebo byly probrány při vysvětlování souborů (viz str. 272).

- Soubor <stdio.h> popisuje následující funkce a makra:
 fopen() otevírá soubor 278
 freopen() přesměruje otevřený soubor do jiného souboru 306
 fflush() zapíše obsah bufferu do souboru 299

<code>fclose()</code>	uzavře soubor	281
<code>remove()</code>	vymaže soubor	308
<code>rename()</code>	přejmenuje soubor	308
<code>tmpfile()</code>	vytvoří dočasný soubor	308
<code>tmpnam()</code>	vytvoří dočasné jméno souboru	308
<code>setbuf()</code>	změní buffer	295
<code>setvbuf()</code>	změní buffer a způsob práce s ním	296
<code>fprintf()</code>	zapisuje formátovaně do souboru	72
<code>printf()</code>	zapisuje formátovaně do <code>stdout</code>	39
<code>sprintf()</code>	zapisuje formátovaně do řetězce	202
<code>vfprintf()</code>	zapisuje do souboru proměnný počet parametrů	366
<code>vprintf()</code>	zapisuje do <code>stdout</code> proměnný počet parametrů	366
<code>vsprintf()</code>	zapisuje do řetězce proměnný počet parametrů	366
<code>fscanf()</code>	čte formátovaně ze souboru	72
<code>scanf()</code>	čte formátovaně ze <code>stdin</code>	39
<code>sscanf()</code>	čte formátovaně z řetězce	202
<code>vfscanf()</code>	čte ze souboru proměnný počet parametrů	366
<code>vscanf()</code>	čte ze <code>stdin</code> proměnný počet parametrů	366
<code>vsscanf()</code>	čte z řetězce proměnný počet parametrů	366
<code>fgetc()</code>	čte znak ze souboru – vždy funkce	287
<code>getc()</code>	čte znak ze souboru – může být makro	287
<code>fgets()</code>	čte řetězec ze souboru – vždy funkce	290
<code>gets()</code>	čte řetězec ze <code>stdin</code>	204
<code>fputc()</code>	zapisuje znak do souboru – vždy funkce	287
<code>putc()</code>	zapisuje znak do souboru – může být makro	287
<code>fputs()</code>	zapisuje řetězec do souboru	290
<code>puts()</code>	zapisuje řetězec do <code>stdout</code>	204
<code>getchar()</code>	čte znak ze <code>stdin</code>	39
<code>putchar()</code>	zapisuje znak do <code>stdout</code>	39
<code>ungetc()</code>	vrací znak na <code>stdin</code>	84
<code>fread()</code>	čte neformátovaně ze souboru	292
<code>fwrite()</code>	zapisuje neformátovaně do souboru	292
<code>fseek()</code>	přesune se v souboru	300
<code>ftell()</code>	vrátí pozici aktuálního místa v souboru	301
<code>rewind()</code>	přesune se na začátek souboru	
<code>fgetpos()</code>	uloží pozici aktuálního místa v souboru	
<code>fsetpos()</code>	vrátí se na uloženou pozici v souboru	
<code>clearerr()</code>	nuluje indikátory chyby a konce souboru	283
<code>feof()</code>	testuje příznak konce souboru	288
<code>ferror()</code>	testuje příznak chyby práce se souborem	283
<code>perror()</code>	tiskne chybu podle obsahu <code>errno</code>	327

19.9 <stdlib.h> – obecně užitečné funkce

Zde popsané funkce se mohou rozdělit do několika následujících skupin.

19.9.1 Konverze řetězců na čísla

Existují dvě skupiny funkcí:

Funkce pro jednoduchý převod desítkových čísel

Jsou to funkce `atoi()`, `atol()`, `atof()` – „ASCII to int, long, double“⁸.

Funkce pro náročný převod čísel

Všechny tři funkce umožňují velmi variabilní zápis čísla – úvodní bílé znaky, znaménko, exponent, ... a poskytují indikaci místa výskytu případného chybného znaku.

Funkce `strtol()`, `strtoul()` dokáží převést řetězec obsahující číslo v libovolné číselné soustavě⁹ na číslo `long` respektive `unsigned long`.

Funkce `strtod()` konvertuje řetězec na `double`.

Poznámky:

- ANSI C nedefinuje opačné funkce, tj. pro převod čísel na řetězce, např. `itoa()`. V těchto případech je nutno použít funkci `sprintf()` (viz str. 198).
- V příkladu jsou za // uvedeny výsledky.

```
char *s, *chyba;
long int l;

s = "1234567";
l = atol(s);
printf("s=%s l=%ld\n", s, l);           // s=1234567 l=1234567

s = "1234567";
l = strtol(s, &chyba, 10);
printf("s=%s l=%ld\n", s, l);           // s=1234567 l=1234567

s = "FFFF";
l = strtol(s, &chyba, 16);
printf("s = %s l = %ld\n", s, l);       // s = FFFF l = 65535
```

⁸ Pozor – ne `float`.

⁹ Přesněji binární až třicetšestkové(!).


```
s = "0xFGFFF";
l = strtol(s, &chyba, 16);
printf("s = %s l = %ld\n"           // s = 0xFGFFF l = 15
      "chybny znak: %s\n", s, l, chyba); // chybny znak: GFFF

s = "1111111111111111";
l = strtol(s, &chyba, 2);
printf("s=%s\n"                     // s=1111111111111111
      "l=%ld\n", s, l);              // l=65535
```

19.9.2 Generátor pseudonáhodných čísel

Používají se dvě funkce:

- 1) `int rand(void);`
 Vrací pseudonáhodné číslo od 0 do `RAND_MAX`, ale při každém spuštění programu je posloupnost stejná.
 Potřebujeme-li náhodná čísla v určitém rozsahu, použijeme:
`rand() % ROZSAH` např. `rand() % 20`
 který poskytne čísla od 0 do `ROZSAH - 1`, např. od 0 do 19.
 Potřebujeme-li náhodná čísla v rozsahu `<0, 1)`, použijeme s výhodou konstantu `RAND_MAX`, např.:
`#define real_rand() ((double) rand() / (RAND_MAX + 1.0))`
- 2) `void srand(unsigned int start);`
 Inicializuje generátor `rand()` počáteční hodnotou `start`.
 Pozor na skutečnost, že `rand()` generuje při každém spuštění programu opět stejnou posloupnost, pokud je `start` konstanta.
 Potřebujeme-li skutečně náhodná čísla, je nutné `srand()` inicializovat náhodnou hodnotou – nejlépe nějakým časovým údajem – viz příklad.¹⁰

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int i;
    srand((unsigned int) time(NULL));
    printf("\nDeset nahodnych cisel od 0 do 99\n");
    for (i = 0; i < 10; i++)
        printf("%2d ", rand() % 100);
    ...
}
```

¹⁰ Použití funkce `time()` viz podrobně na str. 356.

19.9.3 Funkce pracující s dynamickou pamětí

Jsou to funkce typu `malloc()`, ..., jejichž podrobný popis viz str. 164.

19.9.4 Funkce pro spolupráci s operačním systémem

Funkce `abort()`

Funkční prototyp: `void abort(void);`

Funkce `abort()` nestandardně¹¹ ukončí program, tzn. že nezapíše buřery, nemaže dočasné soubory, nespouští funkce registrované pomocí `atexit()`, atd. V UNIXu se snaží zavřít soubory a generuje soubor `core` – viz str. 404.

Z těchto důvodů je vhodná jen pro **okamžité a rychlé** ukončení programu, nejčastěji v případě nějakého závažného chybového stavu, kdy lze očekávat, že program již stejně způsobil „chaos“.

Funkce `atexit()`

Funkční prototyp: `int atexit(void (* stop_funkce)(void) );`

Je to velmi užitečná funkce, která připraví k provádění (zaregistruje) funkci `stop_funkce()`. Tato zaregistrovaná funkce se spustí **po** ukončení funkce `main()`, což umožňuje opustit program vždy stejným způsobem, bez ohledu na místo nebo způsob¹² ukončení programu.

Výhodné použití je např. pro uvolnění dynamické paměti, zápis ladicích informací do souboru, atd.

Skutečně neocenitelné služby může poskytnout v případě, že na programu spolupracuje více lidí. Pak si každý zaregistruje svoji vlastní ukončovací funkci, která po skončení programu automaticky „uklidí“ vše, za co je dotýčný programátor zodpovědný, bez starostí o úklid po jiných.

Poznámky:

- Je možné mít až 32 těchto funkcí, které jsou volány v obráceném pořadí, než v jakém byly registrovány.
- Parametrem `atexit()` je ukazatel na funkci bez parametrů.

```
#include <stdio.h>
#include <stdlib.h>
```

¹¹ Srovnej s funkcí `exit()` – str. 342.

¹² S výjimkou ukončení programu pomocí funkce `abort()`.

```

void exit_fn1(void) { printf("1. exit_funkce\n"); }

void exit_fn2(void) { printf("2. exit_funkce\n"); }

int main(void)
{
    printf("Zacatek programu a registrace exit_fn funkci\n");
    atexit(exit_fn1);
    atexit(exit_fn2);
    printf("Konec programu\n");
    return 0;
}

```

Vypíše:

```

Zacatek programu a registrace exit_fn funkci
Konec programu
2. exit_funkce
1. exit_funkce

```

Funkce exit()

Funkční prototyp: `void exit(int stav);`

Standardně¹³ ukončí provádění programu před dosažením jeho konce (tj. `return` v `main()`), přičemž může být volána z libovolně vnořené funkce.

Funkce `exit()` funguje jako příkaz `return` v `main()`, tzn. zapíše se buffery, uzavřou se soubory, volají se všechny funkce registrované pomocí `atexit()`, mažou se dočasné soubory, atd.

Operačnímu systému¹⁴ je předán jediný parametr `exit()` a konvence říká, že hodnota 0 má většinou význam „program skončil bez chyby“. Nenulová hodnota pak značí chybu v ukončeném programu a opět je dáno konvencí, že čím větší číslo, tím horší chyba.

Příklad typického použití:

```

if ((fr = fopen("NESMYSL.DAT", "r")) == NULL) {
    printf("Soubor neexistuje\n");
    exit(1);
}

```

¹³ Srovnej se funkcí `abort()` – str. 341.

¹⁴ V MS-DOSu se uloží do systémové proměnné `ERRORLEVEL`. Ve skriptech UNIXu je vrácená nulová hodnota považována za *true* a nenulová hodnota za *false*.

Funkce system()

Funkční prototyp: `int system(const char *prikaz_os);`

Tato funkce umožní vyvolat z našeho programu příkaz operačního systému nebo spustit pod sebou jiný program.

Poznámka:

- Před voláním `system()` by měly být uzavřeny všechny soubory.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```

int main(void)
{
    printf("Vyvolani prikazu DIR z programu\n");
    system("dir *.c");
    return 0;
}

```

19.9.5 Funkce pro hledání a řazení

Řazení prvků pole a hledání nějakého prvku v seřazeném poli jsou akce, které používá mnoho běžných programů. S využitím standardních funkcí odpadnou starosti s naprogramováním těchto akcí. Navíc tyto funkce bývají obvykle napsány velmi dobře, takže jsou rychlé.

Řazení algoritmem *quick-sort*

Používá se funkce:

```

void qsort(void *razene_pole, size_t n_prv, size_t vel_prv,
           int (*porov_fce)(const void *, const void *));

```

Funkce seřadí *vzestupně* pole `razene_pole` o `n_prv` prvcích, kde každý prvek má velikost `vel_prv`.

Pro porovnání dvou prvků navzájem se použije funkce `porov_fce()`. Tuto funkci musíme zajistit sami¹⁵ a musí splňovat následující podmínky¹⁶:

- má dva parametry,
- vrací:
 - záporné číslo* – je-li první parametr menší než druhý,
 - nulu* – jsou-li parametry shodné,
 - kladné číslo* – je-li první parametr větší než druhý.

¹⁵ V nejhorším případě ji musíme sami napsat.

¹⁶ Tyto podmínky splňuje několik standardních funkcí, např. `strcmp()`.

Binární vyhledávání v seřazeném poli

Používá se funkce:

```
void *bsearch(const void *klic, const void *serazene_pole,
             size_t n_prv, size_t vel_prv,
             int (*porov_fce)(const void *, const void *));
```

Funkce `bsearch()` má podobné parametry jako `qsort()` včetně požadavků na porovnávací funkci. První parametr `klic` je položka, podle které se bude vyhledávat – vyhledávací klíč.

Návratovou hodnotou `bsearch()` je ukazatel na prvek `pole`, jenž (jehož položka) vyhovuje klíči, nebo `NULL`, při neúspěšném hledání.

Příklad:

Použití obou funkcí ukazuje následující program, který simuluje práci s jednoduchou databází osob. Ta je na počátku náhodně nastavena.

Poznámky k programu:

- Za povšimnutí stojí dva způsoby získání ukazatel na strukturu z generického ukazatele (`void *`):
 - 1) Buď použijeme lokální proměnnou a inicializujeme ji přetypovaným parametrem:


```
CLOVEK *p_1 = (CLOVEK *) p_a /* sort_vek() */
```

 což je přehlednější zápis, ale pomalejší program (přiřazení navíc).
 - 2) Nebo je možné přetypovat parametr přímo při použití¹⁷:


```
((CLOVEK *)p_a)->jmeno /* sort_jmeno() */
```
- Všimněte si též pomocných výpisů u `search_vek()` a `search_jmeno()`, pomocí nichž lze při ladění sledovat činnost funkce `bsearch()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 20

/* funkční prototypy */
int sort_vek(const void *p_a, const void *p_b);
int search_vek(const void *p_a, const void *p_b);
int sort_jmeno(const void *p_a, const void *p_b);
int search_jmeno(const void *p_a, const void *p_b);
void nahodna_inicializace(void);
```

¹⁷ Všechny závorky jsou nutné!

```
typedef struct {
    int vek;
    char jmeno[6];
} CLOVEK;
```

```
CLOVEK pole[MAX];
```

```
int main(void)
```

```
{
    int i;
    char jmn[6];
    int stari;
    CLOVEK *p_clovek;

    nahodna_inicializace();
    printf("Serazeno podle veku\n");

    qsort((void *)pole, MAX, sizeof(CLOVEK), sort_vek);

    for (i = 0; i < MAX; i++) {
        printf("%2d: %3d - %s\n", i, pole[i].vek, pole[i].jmeno);
    }

    while(1) {
        printf("Zadej vek (0 - 99): ");
        scanf("%d", &stari);
        if (stari < 0)
            break;

        p_clovek = (CLOVEK *) bsearch((void *) &stari,
                                     (void *)pole, MAX, sizeof(CLOVEK), search_vek);

        if (p_clovek != NULL) {
            printf("vek = %3d, jmeno: %s (poradi: %d)\n",
                  p_clovek->vek, p_clovek->jmeno, p_clovek - pole);
        }
    }

    printf("Serazeno podle jmena\n");

    qsort((void *)pole, MAX, sizeof(CLOVEK), sort_jmeno);
```

```

for (i = 0; i < MAX; i++) {
    printf("%2d: %3d - %s\n", i, pole[i].vek, pole[i].jmeno);
}

while(1) {
    printf("Zadej jmeno: ");
    scanf("%5s", jmn);
    if (jmn[0] == '*')
        break;

    p_clovek = (CLOVEK *) bsearch( (void *) jmn, (void *) pole,
                                   MAX, sizeof(CLOVEK), search_jmeno);

    if (p_clovek != NULL) {
        printf("jmeno: %s - vek = %3d, (poradi: %d)\n",
               p_clovek->jmeno, p_clovek->vek, p_clovek - pole);
    }
}

int sort_vek(const void *p_a, const void *p_b)
{
    CLOVEK *p_1 = (CLOVEK *) p_a,
            *p_2 = (CLOVEK *) p_b;

    if (p_1->vek < p_2->vek)
        return -1;
    else if (p_1->vek == p_2->vek)
        return 0;
    else
        return +1;
}

int search_vek(const void *p_a, const void *p_b)
{
    /* pomocny vypis pro ukazku prubehu hledani */
    printf("a = %2d  b = %2d\n",
           *(int *)p_a, ((CLOVEK *)p_b)->vek);

    if (*(int *)p_a < ((CLOVEK *)p_b)->vek)
        return -1;

```

```

else if (*(int *)p_a == ((CLOVEK *)p_b)->vek)
    return 0;
else
    return +1;
}

int sort_jmeno(const void *p_a, const void *p_b)
{
    return ( strcmp(((CLOVEK *)p_a)->jmeno,
                    ((CLOVEK *)p_b)->jmeno) );
}

int search_jmeno(const void *p_a, const void *p_b)
{
    /* pomocny vypis pro ukazku prubehu hledani */
    printf("a = %s  b = %s\n",
           (char *)p_a, ((CLOVEK *)p_b)->jmeno);

    return ( strcmp((char *)p_a, ((CLOVEK *)p_b)->jmeno) );
}

void nahodna_inicializace(void)
{
    int i, j;

    for (i = 0; i < MAX; i++) {
        pole[i].vek = rand() % 100;
        for (j = 0; j < 5; j++) {
            pole[i].jmeno[j] = rand() % 26 + 'a';
        }
        pole[i].jmeno[5] = '\0';
    }
}

```

Poznámky:

- Obě funkce jsou velmi rychlé. Seřazení 5 000 000 těchto struktur trvalo podle věku asi 2,6 sec. Čas vyhledávání nebyl vůbec měřitelný. (Athlon 1,4 GHz, 512 MB RAM překladač gcc, prostředí cygwin.)
- Záleží-li nám na rychlosti, pak je pro velká pole a rozměrné struktury výhodné mít pole pointerů na tyto struktury (viz str. 251) a to řadit. Pak se totiž neprohazují v poli celé struktury, ale jen pointery na ně.

■ Soubor `<stdlib.h>` popisuje následující funkce a makra:

<code>atof()</code>	konverze řetězce na <code>double</code>	339
<code>atoi()</code>	konverze řetězce na <code>int</code>	339
<code>atol()</code>	konverze řetězce na <code>long int</code>	339
<code>strtod()</code>	konverze řetězce na <code>double</code>	339
<code>strtoul()</code>	konverze řetězce na <code>long int</code>	339
<code>strtoul()</code>	konverze řetězce na <code>unsigned long int</code>	339
<code>rand()</code>	generátor náhodného čísla	340
<code>srand()</code>	inicializace generátoru náhodného čísla	340
<code>calloc()</code>	alokace dynamické paměti pro pole objektů	168
<code>malloc()</code>	alokace bloku dynamické paměti	165
<code>realloc()</code>	změna velikosti bloku dynamické paměti	182
<code>free()</code>	uvolnění dynamické paměti	167
<code>abort()</code>	nestandardní ukončení programu	341
<code>exit()</code>	standardní ukončení programu	342
<code>atexit()</code>	registrace ukončující funkce	341
<code>system()</code>	volání systémového příkazu	343
<code>getenv()</code>	poskytnutí informace o systémové proměnné	328
<code>bsearch()</code>	binární vyhledávání	344
<code>qsort()</code>	řazení algoritmem <i>quick-sort</i>	343
<code>abs()</code>	absolutní hodnota typu <code>int</code>	
<code>labs()</code>	absolutní hodnota typu <code>long int</code>	
<code>div(x, y)</code>	<code>x/y</code> a <code>x%y</code> současně pro typy <code>int</code>	
<code>ldiv(x, y)</code>	<code>x/y</code> a <code>x%y</code> současně pro typy <code>long int</code>	

19.10 <string.h> – zpracování řetězců

Zde jsou popsány funkce, které zpracovávají řetězec znaků ukončený `'\0'` – `strcpy()`, `strstr()`, `strncpy()`, ... Popis těchto funkcí viz str. 200.

Samostatnou skupinou těchto funkcí jsou tři funkce, které hledají shodné nebo rozdílné znaky v řetězci `zdroj` v porovnání s množinou znaků v řetězci `set`. Na pořadí znaků v řetězci `set` nezáleží.

19.10.1 Funkce `strstr()`

Prototyp: `size_t strstr(const char *zdroj, const char *set);`
Vrací počet prvních znaků ze `zdroj`, které jsou obsaženy v `set`.¹⁸

¹⁸ Je to prakticky hledání prvního rozdílného znaku.

19.10.2 Funkce `strcspn()`

Prototyp: `size_t strcspn(const char *zdroj, const char *set);`
Vrací počet prvních znaků ze `zdroj`, které nejsou obsaženy v `set`.¹⁹

19.10.3 Funkce `strpbrk()`

Prototyp: `char *strpbrk(const char *zdroj, const char *set);`
Vrací ukazatel na první znak ze `zdroj`, který je obsažen v `set`.²⁰

19.10.4 Funkce `strtok()`

Prototyp: `char *strtok(char *zdroj, const char *set);`
Postupně dělí řetězec `zdroj` na části, které jsou odděleny jedním ze znaků v `set`. Při prvním volání se `strtok()` inicializuje pomocí `zdroj`, druhé a další volání již mají místo `zdroj` parametr `NULL`.

```
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
    char s1[] = "nazdar ahoj,dobry-den";
    char s2[] = "qwx radzan";
    char s3[] = "qwxr";
    char s4[] = "od";
    int pozice;
    char *p_c;

    pozice = strstr(s1, s2);
    printf("Prvnich %d znaku s1 je obsazeno v s2\n", pozice);

    pozice = strcspn(s1, s3);
    printf("Prvnich %d znaku s1 není obsazeno v s3\n", pozice);

    p_c = strpbrk(s1, s4);
    if (p_c != NULL)
        printf("První shodný znak v s1 a v s4: %c\n", *p_c);
    else
        printf("Zadný shodný znak\n");
}
```

¹⁹ Je to prakticky hledání prvního shodného znaku.

²⁰ Je to prakticky hledání prvního shodného znaku.


```

p_c = strtok(s1, " , -");      /* mezera, čárka, pomlčka */
if (p_c != NULL)
    printf("%s\n", p_c);

while ((p_c = strtok(NULL, " , -")) != NULL)
    printf("%s\n", p_c);
return 0;
}

```

Vypíše:

```

Prvních 8 znaku s1 je obsazeno v s2
Prvních 5 znaku s1 není obsazeno v s3
První shodný znak v s1 a v s4: d
nazdar
ahoj
dobry
den

```

Další skupinou funkcí jsou funkce pro práci s částí paměti, která nemusí být ukončena znakem '\0'. Ty jsou užitečné pro rychlou práci s blokem paměti.

19.10.5 Funkce memchr()

Prototyp: `void *memchr(const void *pam, int ba, size_t n_ba);`

Funkce nalezne první výskyt bajtu `ba` v poli bajtů, dlouhém `n_ba`, a vrátí na něj ukazatel. V případě, že bajt není nalezen, vrátí `NULL`.

19.10.6 Funkce memcmp()

Prototyp:

```
int memcmp(const void *pam1, const void *pam2, size_t n_ba);
```

Porovná `n_ba` bajtů od adresy `pam1` s bajty od adresy `pam2` a vrátí:

záporné číslo – je-li první nalezený rozdílný bajt prvního pole číselně menší než odpovídající bajt pole `pam2`
nulu – jsou-li obsahy pamětí shodné
kladné číslo – je-li bajt v `pam1` větší než bajt v `pam2`

19.10.7 Funkce memcpy()

Prototyp:

```
void *memcpy(void *pam1, const void *pam2, size_t n_ba);
```

Zkopíruje `n_ba` bajtů z `pam2` do `pam1` a vrátí ukazatel na `pam1`.

Tato funkce je výhodná např. při kopírování polí:

```
memcpy((void *)pole1, (void *)pole2, n_prvku*sizeof(pole2[0]));
```

19.10.8 Funkce memmove()

Prototyp:

```
void *memmove(void *pam1, const void *pam2, size_t n_ba);
```

Zkopíruje `n_ba` z `pam2` do `pam1` a vrátí ukazatel na `pam1`. Narozdíl od funkce `memcpy()` se mohou pole `pam1` a `pam2` překrývat.

19.10.9 Funkce memset()

Prototyp: `void *memset(void *pam, int ba, size_t n_ba);`

Vyplní prvních `n_ba` bajtů pole `pam` bajtem `ba`, který konvertuje na `unsigned char`, a vrátí ukazatel na `pam`.

Příklad:

Následující program ukáže použití funkcí `mem...()`.

```

#include <stdio.h>
#include <string.h>
#define DELKA 1000

int main(void)
{
    int a[DELKA], b[DELKA];
    int i, *p_i;

    memset((void *)a, 0, (size_t) (DELKA * sizeof(int)));
    printf("memset: a[%d] = %d\n", DELKA / 2, a[DELKA / 2]);
    a[DELKA / 2] = 1;

    p_i = (int *) memchr((void *) a, 1,
                        (size_t) (DELKA * sizeof(int)));
    printf("memchr. i = %u\n", p_i - a);

    memcpy((void *)b, (void *)a, (size_t) (DELKA * sizeof(int)));
    printf("memcpy: b[%d] = %d\n", DELKA / 2, b[DELKA / 2]);
}

```

```

for (i = 1; i < 9; i++)
    a[i] = i - 1;
for (i = 0; i < 9; i++)
    printf("a[%d]=%d ", i, a[i]);

printf("\nmemmove:\n");
memmove((void *) &a[0], (void *) &a[1],
        (size_t) (9 * sizeof(int)));
for (i = 0; i < 9; i++)
    printf("a[%d]=%d ", i, a[i]);
return 0;
}

```

Vypíše:

```

memset: a[500] = 0
memchr: i = 500
memcpy: b[500] = 1
a[0]=0 a[1]=0 a[2]=1 a[3]=2 a[4]=3 a[5]=4 a[6]=5 a[7]=6 a[8]=7
memmove:
a[0]=0 a[1]=1 a[2]=2 a[3]=3 a[4]=4 a[5]=5 a[6]=6 a[7]=7 a[8]=0

```

■ Soubor <string.h> popisuje následující funkce a makra:

strcpy()	kopírování řetězce	200
strncpy()	kopírování řetězce s omezením délky	200
strcat()	spojení řetězců	200
strncat()	spojení řetězců	200
strcmp()	porovnání řetězců	200
strncmp()	porovnání řetězců	200
strchr()	hledání znaku v řetězci	200
strrchr()	hledání znaku v řetězci od jeho konce	200
strstr()	hledání podřetězce v řetězci	200
strlen()	aktuální délka řetězce	200
strspn()	hledání prvního rozdílného znaku	348
strcspn()	hledání prvního shodného znaku	349
strpbrk()	hledání prvního shodného znaku	349
strerror()	poskytnutí chybového hlášení	327
strtok()	dělení řetězce na části	349
memcpy()	kopírování bloku bajtů	351
memmove()	jako memcpy(), bloky se mohou překrývat	351
memcmp()	porovnávání bloků bajtů	350
memchr()	hledání znaku v bloku bajtů	350
memset()	vyplnění bloku bajtů konstantou	351

19.11 <time.h> – práce s datem a časem

V tomto souboru jsou popsány datové typy a funkce umožňující práci s časem a datem.

Poznámka:

- Při prvním seznámení nebo prohlížení konkrétních příkladů vás může zarazit, že některé funkce se dají použít několika principiálně různými způsoby se stejným výsledkem²¹.

Pro pochopení funkcí je nezbytné znát následující zde definované datové typy a konstanty:

CLK_TCK	Symbolická konstanta, původně vyjadřující počet procesorových tiků za sekundu. V současné době se již nepoužívá a je nastavena na hodnotu 1000, což nemusí odpovídat skutečnosti. Tato konstanta má někdy též název CLOCKS_PER_SEC.		
clock_t	Čtyřbajtové celé znaménkové číslo (nejčastěji long), sloužící pro funkci clock().		
time_t	Čtyřbajtové celé znaménkové číslo (nejčastěji long), sloužící pro všechny ostatní funkce z <time.h>.		
tm	Struktura, která slouží pro uložení všech možných složek času. Obsahuje tyto položky:		
	int tm_sec	sekundy po minutě	[0, 59]
	int tm_min	minuty po hodině	[0, 59]
	int tm_hour	hodiny od půlnoci	[0, 23]
	int tm_mday	den v měsíci	[1, 31]
	int tm_mon	měsíc od ledna	[0, 11]
	int tm_year	roky od 1900	[0, 2038]
	int tm_wday	dny od neděle	[0, 6]
	int tm_yday	počet dnů od 1. ledna	[0, 366]
	int tm_isdst	příznak letního času	

Poznámky:

- Pozor na záludné označení měsíců, ke kterým je nutno přičíst jedničku, a dnů v týdnu, které je též nutno též o jedničku zvětšit²².
- Položka tm_isdst má kladnou hodnotu, je-li letní čas, nulovou hodnotu, je-li normální čas, a zápornou hodnotu, když se to nedá určit.

²¹ Viz příklad funkce time() na str. 356.

²² V USA začíná týden nedělí.

Dále následuje popis některých výhodných funkcí z `<time.h>`.

19.11.1 Funkce `clock()`

Funkční prototyp: `clock_t clock(void);`

Funkce vrací počet tiků procesoru. Používala se zejména pro přesné měření doby trvání programu. V současné době, kdy se běžně používají víceúlohové a víceuživatelské operační systémy, poskytuje značně nepřesné výsledky a proto se pro tento účel nepoužívá.

19.11.2 Měření doby běhu programu

Poznámky:

- Zde se dostáváme mimo oblast ANSI C, takže platí stejná varování jako již byla uvedena na str. 311.
- Budeme potřebovat soubory `<sys/times.h>` a `<sys/unistd.h>`.

Chceme-li v současné době dostat relevantní (důvěryhodné) informace o reálném čase, po který program běžel, musíme použít funkci

`clock_t times(struct tms *p_tms)`

Jejím parametrem je pointer na strukturu `tms`, kterou funkce `times()` naplní.

Funkce vrací počet tiků procesoru, který je pro převedení na sekundy nutné podělit návratovou hodnotou funkce `sysconf(_SC_CLK_TCK)`²³.

Tato hodnota ale opět představuje celkový spotřebovaný čas, který je často ovlivněn souběžně spuštěnými dalšími programy, nikoliv jen čas spotřebovaný naším programem.

Z tohoto důvodu se často využívá položka struktury `tms.tms_utime`, do které funkce `times()` uloží počet tiků spotřebovaných programem.

Příklad:

Z dále uvedeného výpisu je vidět, že celkový a programový čas je v podstatě stejný, pokud neběží souběžně jiný program. Na druhém výpisu je vidět, že pokud spustíme současně jiný program, pak celkový čas narůstá, ale programový zůstává stále stejný.

Všimněte si též nezbytného přetypování na `double` při přepočítávání počtů tiků na sekundy. Všechny hodnoty jsou totiž celočíselné a bez přetypování alespoň jednoho operandu na `double` by dělení proběhlo celočíselně a výsledek by byl též celé číslo. Tím bychom jednak ztratili přesnost desetinných řádů a jednak by se celé číslo chybně vytisklo ve formátu `%.3f`.

²³ Konstanta `_SC_CLK_TCK` je ze souboru `<sys/unistd.h>`.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <sys/unistd.h>

#define MAX          1000
#define VNITRNI_SMYCKA 1000000

int main(int argc, char *argv[])
{
    unsigned long int i, j, vnejsi_smycka;
    int tik_u_za_sekundu;
    struct tms casova_struktura;
    clock_t celkovy_z, celkovy_k;
    clock_t program_z, program_k;

    vnejsi_smycka = (argc == 1) ? MAX : atol(argv[1]);
    tik_u_za_sekundu = sysconf(_SC_CLK_TCK);

    celkovy_z = times(&casova_struktura);
    program_z = casova_struktura.tms_utime;

    for (i = 0; i < vnejsi_smycka; i++) {
        for (j = 0; j < VNITRNI_SMYCKA; j++) {
            i++;
            i--;
        }
    }

    celkovy_k = times(&casova_struktura);
    program_k = casova_struktura.tms_utime;

    printf("celkovy = %.3f [sec]\n",
           (celkovy_k - celkovy_z) / (double) tik_u_za_sekundu);
    printf("program = %.3f [sec]\n",
           (program_k - program_z) / (double) tik_u_za_sekundu);

    return 0;
}
```

Vypíše:

Pokud program běží samostatně.

```
celkovy = 9.253 [sec]
```

```
program = 9.233 [sec]
```

Pokud program běží souběžně s jiným programem.

```
celkovy = 11.277 [sec]
```

```
program = 9.343 [sec]
```

19.11.3 Funkce time()

Funkční prototyp: `time_t time(time_t *cas);`

Funkce vrací počet sekund od 1. ledna 1970 do okamžiku svého vyvolání. Protože pracuje jen s přesností na sekundy, není pro účel měření času nejvýhodnější. Její hlavní použití je jako primární funkce pro práci se složkami času²⁴.

Pozor na její víceznačné použití – následující volání pokaždé naplní proměnnou `pocet_sec`, v prvním případě i proměnnou `n_sec`.

```
time_t pocet_sec, n_sec;
```

```
pocet_sec = time(&n_sec);
pocet_sec = time(&pocet_sec);
pocet_sec = time(NULL);
time(&pocet_sec);
```

19.11.4 Funkce difftime()

Funkční prototyp: `double difftime(time_t cas2, time_t cas1);`

Vrátí rozdíl časů – vypočte: `(cas2 - cas1)`

Spolu s funkcí `time()` se používá pro snadné, ale přibližné měření času.

19.11.5 Funkce localtime()

Funkční prototyp: `struct tm *localtime(const time_t *p_cas);`

Funkce rozloží počet sekund od 1. ledna 1970, získaný pomocí `time()` do jednotlivých složek struktury `tm` – naplní strukturu `tm`.

Funkce `localtime()` si sama strukturu `tm` alokuje. V prvním případě je pomocí dereferenčního operátoru `*` provedeno přiřazení celé struktury `tm` a ve druhém případě je přiřazen pouze ukazatel na `tm`.

```
struct tm    tm_promenna, *p_tm;
time_t       cas;

time(&cas);
tm_promenna = *localtime(&cas); /* přiřazení struktur */
if (tm_promenna.tm_min == 2)
    ...

p_tm = localtime(&cas);          /* přiřazení ukazatelů */
if (p_tm->tm_min == 2)
    ...
```

19.11.6 Funkce asctime()

Funkční prototyp: `char *asctime(const struct tm *p_tm);`

Z některých položek struktury `tm` připraví řetězec ve formátu:

```
Mon Jun 19 16:00:14 1995
```

19.11.7 Funkce ctime()

Funkční prototyp: `char *ctime(const time_t *p_cas);`

Z počtu sekund od 1. ledna 1970 připraví řetězec ve formátu:

```
Mon Jun 19 16:00:14 1995
```

Poznámka:

- Funkce `ctime` má stejný efekt jako: `asctime(localtime(p_cas));`

```
time_t t;

time(&t);
printf("%s\n", asctime(localtime(&t)));
printf("%s\n", ctime(&t));
```

Program vypíše:

```
Mon Jun 19 16:00:14 1995
```

```
Mon Jun 19 16:00:14 1995
```

²⁴ Viz struktura `tm` na str. 353 a funkce `localtime()` na str. 356.

19.11.8 Funkce strftime()

Budeme-li často pracovat s časovými údaji a zejména je vypisovat, nejsou funkce asctime() a ctime() příliš vhodné, protože zobrazují jména anglicky a údaje často ve formátu, který našim zvyklostem ne úplně vyhovuje. V tomto případě použijeme funkci strftime()²⁵, u které lze s velkou variabilitou volit formáty jednotlivých položek.

Funkční prototyp: `size_t strftime (char *str, size_t max, char *format, struct tm *p_tm);`

Kde:

`str` – řetězec, do kterého budou zapsány časové údaje,
`max` – maximální délka tohoto řetězce,
`p_tm` – ukazatel na strukturu naplněnou pomocí localtime(),
`format` – formátovací znaky (podobné jako u printf() s významem:
 %H – hodiny (24 hodinový cyklus),
 %M – minuty,
 %S – sekundy,
 %d – den v měsíci,
 %m – měsíc číslem,
 %Y – rok včetně století (čtyři místa – 2003),
 %y – rok na dvě místa (03),
 %A – jméno dne plným slovem (Monday),
 %a – jméno dne zkratkou (Mon),
 %B – jméno měsíce plným slovem (November),
 %b – jméno měsíce zkratkou (Nov),
 %X – údaj o celém čase (11:39:19),
 %x – údaj o celém datumu (11/20/03),
 %c – datum a čas (11/20/03 11:39:19),
 %I – hodiny (12 hodinový cyklus),
 %p – údaj o polovině dne (AM nebo PM).

Funkce strftime() vrací počet správně zapsaných znaků.

Příklad:

Zde je vidět použití funkce a také srovnání, jak složitě bylo nutné dříve vypisovat údaj např. o roku, kdy jsme nesměli zapomenout na přičtení konstanty 1900. To při použití strftime() odpadá.

²⁵ Pracuje podobně jako funkce sprintf().

```
time_t cas;
struct tm *p_st_cas;
char str[1000];
size_t pocet;

cas = time(NULL);
p_st_cas = localtime(&cas);

pocet = strftime(str, 1000, "rok = %Y", p_st_cas);
printf("%s (%d znaku)\n", str, pocet);

/* stary zpusob */
printf("%d = rok\n", p_st_cas->tm_year + 1900);
```

Příklad:

Téměř vyčerpávající ukázka možností použitelných formátů.

```
"Cas = %H:%M:%S, datum = %d.%m.%Y\n"
"Den = %A (%a), mesic = %B (%b)\n"
"Cas = %X, datum = %x\n"
"Datum i cas = %c\n"
"Hodiny (12h) = %I %p\n"
```

Poskytne:

```
Cas = 11:52:10, datum = 20.11.2003
Den = Thursday (Thu), mesic = November (Nov)
Cas = 11:52:10, datum = 11/20/03
Datum i cas = 11/20/03 11:52:10
Hodiny (12h) = 11 AM
```

19.11.9 Funkce mktime()

Funkční prototyp: `time_t mktime(struct tm *p_tm);`

Tato funkce má dvojí použití:

- 1) Jako inverzní funkce k funkci localtime(), kdy ze složek struktury tm vypočte a vrátí počet sekund od 1. ledna 1970.
- 2) Pokud některá položka struktury tm přesahuje meze, funkce mktime() upraví tuto a všechny navazující položky. Například, je-li nastaveno:
`p_tm->tm_sec = 86;`
 pak mktime() zvýší tm_min o 1 a tm_sec upraví na 26.

To znamená, že pro strukturu `tm` jako parametr `mktime()` neplatí meze jednotlivých položek uvedené u popisu těchto položek struktury `tm`²⁶, např. `tm_mon` zde nemusí být od 0 do 11.

Tohoto faktu se dá využít k různým „trikům“, kdy chceme např. zjistit, jaký den v týdnu připadá na určité datum.

```
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv)
{
    struct tm den;
    char *dny[] = {"Ne", "Po", "Ut", "St", "Ct", "Pa", "So"};

    if (argc == 1) {
        printf("Pouziti: den-v-tydnu den mesic rok (20 11 2003)");
        return 1;
    }

    den.tm_sec = 0;
    den.tm_min = 0;
    den.tm_hour = 12;
    den.tm_mday = atoi(argv[1]);
    den.tm_mon = atoi(argv[2]) - 1;
    den.tm_year = atoi(argv[3]) - 1900;
    mktime(&den);
    printf("%d.%d.%d byl den: %s\n",
           den.tm_mday, den.tm_mon + 1, 1900 + den.tm_year,
           dny[den.tm_wday]);

    return 0;
}
```

Vypíše:

20.11.2003 byl den: Ct

²⁶ Viz str. 353.

Další možné využití `mktime()` je pro přesné nastavení času (jak přírůstkově, tak i absolutně), kdy se má program znovu rozběhnout.

Příklad:

Následující program bude pokračovat ve své činnosti dnes ve 20 hodin plus tolik minut, kolik je nyní, plus ještě 15 minut.

```
struct tm *p_pak;
time_t ted, spusteni;

time(&ted);
p_pak = localtime(&ted);
p_pak->tm_min += 15; /* prirustkove */
p_pak->tm_hour = 20; /* absolutne */
spusteni = mktime(p_pak);
while (difftime(time(NULL), spusteni) < 0)
    ; /* prazdny cyklus cekani */
```

■ Soubor `<time.h>` popisuje následující funkce a makra:

<code>clock()</code>	počet tiků procesoru	354
<code>time()</code>	počet sekund od 1. ledna 1970	356
<code>difftime()</code>	rozdíl dvou časů	356
<code>mktime()</code>	práce s položkami času	359
<code>asctime()</code>	konverze složek času do řetězce	357
<code>ctime()</code>	konverze počtu sekund do řetězce	357
<code>gmtime()</code>	práce s „univerzálním světovým časem“	
<code>localtime()</code>	výpočet složek času z počtu sekund	356
<code>strftime()</code>	formátovaná konverze složek času do řetězce	358

20 Funkce s proměnným počtem parametrů

Funkce s proměnným počtem parametrů (dále jen FPPP) v jazyce C je něco, co již svým názvem vzbuzuje respekt, popřípadě bázeň, či odpor těch, kteří se s nimi náhodou setkají. Ať je váš vztah k této oblasti jazyka C jakýkoli, vězte, že se jedná o velmi užitečné pomocníky, na nichž není – kromě názvu – nic tak mystického.

V následujících podkapitolách budou postupně vysvětleny jejich principy, důvody pro používání a uvedeny odladěné typické způsoby použití.

20.1 Úvodní informace

Typické představitele FPPP používáme, zřejmě nevědomky, již od samých začátků práce s jazykem C, neboť standardní funkce `printf()` a `scanf()` jsou právě FPPP.

Na těchto dvou funkcích je dobře vidět jedna z výhod použití FPPP – jsou výhodné např. v případě, kdy potřebujeme jednu a tutéž funkci použít pro různé typy a různé počty skutečných parametrů. To je právě u vstupů a výstupů velmi běžný jev, například příkaz pro tisk pomocí `printf()` může být, jak určitě víte, značně variabilní.

Funkci podobnou funkci `printf()`, ovšem bez možnosti použití FPPP, by bylo prakticky možné napsat třeba tak, že by bylo předem určeno, že pro každý datový typ (`int`, `char`, `float`, ...) bude jedna funkce¹, která bude umět vytisknout jeden datový typ (jedno číslo). Takovéto funkce si dokážeme snadno představit a po určitém úsilí, věnovaném konverzi čísla v příslušném počítačovém zobrazení na jeho znakovou podobu², bychom je i naprogramovali.

¹ Pojmenovaná třeba `printfi()`, `printfc()`, `printfd()`, ...

² Tedy řetězec znaků – číslic, které se vytisknou.

Pak by běžný příkaz tisku pomocí `printf()`:

```
printf(Koupili jsme %d kusu za %f Kc\n, kusy, celkem);
```

vypadal třeba takto:

```
puts("Koupili jsme ");
printfi(kusy);
puts(" kusy za ");
printfd(celkem);
puts(" Kc\n");
```

Je vidět, že tento způsob je mnohem méně přehledný, než běžné použití `printf()`.

Dále si uvědomme, že uvedený způsob mimo jiné neposkytuje často využívanou možnost upřesnění formátu tisku, např.: `"%5.2f"`. To by bylo nutné zařídit dalším parametrem a komplikace by dále narůstaly.

Z tohoto malého příkladu je, doufejme, zřejmé, že FPPP jsou užitečná součást jazyka C a není nutné z nich mít obavy. Můžeme proto začít uvažovat, zda by se nám nějaká naše vlastní FPPP v našich programech nehodila.

Potenciální kandidáti na FPPP jsou:

- 1) vlastní – popř. námi modifikované standardní – výpisové nebo čtecí funkce³,
- 2) přidání dalšího volitelného parametru již používané funkci,
- 3) nejružnější speciální požadavky.

20.2 Základní princip

Při prvním setkání s FPPP si většinou pokládáme otázku, jak je zajištěna jejich správná funkce. Na FPPP však není nic tajemného, když si uvědomíme, že mohou správně pracovat pouze tehdy, je-li možné rozpoznat, kolik parametrů a jakých datových typů FPPP skutečně má.

Je důležité mít obě tyto informace, protože jen počet parametrů nebo jen informace o tom, že jsou všechny jednoho typu (např. `char`) většinou nestačí. Informace o počtu parametrů a jejich typech se předá pomocí minimálně jednoho pevného parametru funkce⁴, ze kterého musí být možno všechny

³ Často jsou používány pro chybové výpisy.

⁴ Obecně se vyhýbáme předávání informace pomocí globální proměnné. Toto pravidlo lze ovšem v odůvodněných případech trochu porušit – viz str. 369.

potřebné informace odvodit. To znamená, že FPPP musí mít alespoň jeden skutečný parametr, ovšem může jich mít i více.

Nejnámější ukázkou předání této informace je právě formátový řetězec funkce `printf()`, kde pro počet proměnných parametrů stačí pouze spočítat počet znaků „%“ v prvním parametru (což je vždycky řetězec) a typ parametru se odvodí z písmene, které za znakem „%“ následuje.

Je samozřejmě možné napsat FPPP, které se v pevném parametru nepředávají typy, ale jen aktuální počet následujících parametrů (viz např. str. 372). Pak ale tyto parametry musí splňovat předem dohodnuté podmínky, např. všechny budou typu `int`.

Stejně tak je možné nepředávat ani počet parametrů, ale ukončit seznam skutečných parametrů nějakou dohodnutou hodnotou (typy parametrů musí být předem známy), např. hodnotou `NULL` (viz např. str. 370).

Jsou-li splněny tyto požadavky, pak FPPP dovede seznam skutečných parametrů zpracovat díky tomu, že jazyk C definuje pořadí uložení skutečných parametrů do zásobníku. Skutečné parametry se ukládají do zásobníku v opačném pořadí než jsou ve funkci uvedeny – tj. od posledního k prvnímu. Další informace, kterou překladač musí znát, jsou velikosti jednotlivých datových typů (např. že typ `int` zabírá dva bajty) – ty jsou známy již při překladu. Tedy již v době překladu je známo, co a kde leží v zásobníku a kolik to tam zabírá místa.

Pak už jen stačí nastavit vhodný ukazatel na poslední pevný parametr v zásobníku a ze zásobníku číst vždy hodnotu příslušného datového typu, tj. další parametr v pořadí. A po přečtení hodnoty parametru se v zásobníku posunout o velikost tohoto typu.

Jestliže si tyto dvě skutečnosti uvědomíme, zřejmě bychom FPPP po určitém úsilí zvládli naprogramovat. Jediný problém, se kterým bychom se při prvních pokusech pravděpodobně setkali, by byla znalost organizace zásobníku⁵ a nestandardní přístup do něho. Zásobník je většinou implementačně závislý, takže program s naší FPPP by byl pravděpodobně nepřenosný.

Aby se práce s FPPP maximálně ulehčila⁶, sjednotila, zprůhlednila a navíc, aby byly FPPP přenositelné, umožňuje ANSI C používat následující nástroje:

1) *výpustka (ellipsis)*, což jsou tři tečky těsně za sebou – „...“

Výpustka je uvedena jak ve funkčním prototypu, tak i v hlavičce funkce. Překladač pak negeneruje varovné hlášení o různém počtu skutečných a formálních parametrů. (Příklad viz str. 373.)

⁵ *Přímý* versus *vrátný* zásobník.

⁶ Manipulace se zásobníkem je většinou zdrojem nepříjemných chyb.

2) Makra ze souboru `<stdarg.h>`, pomocí nichž se ve FPPP dají snadno a přenositelně získat hodnoty skutečných parametrů. Tato makra samozřejmě manipulují se zásobníkem, ale tato činnost je programátorovi skryta.

20.3 Standardní hlavičkový soubor `<stdarg.h>`

Využijeme-li maker z tohoto souboru, nemusíme si dělat starosti, jak jsou parametry do zásobníku ukládány, jak je zásobník organizován, jaká je velikost datových typů atd., čili program bude plně přenositelný na libovolný počítač.

Obsah souboru `<stdarg.h>` je samozřejmě silně implementačně závislý (v každém případě však musí obsahovat definice čtyř dále uvedených identifikátorů), např. může mít zhruba tento obsah:

```
typedef void *va_list;

#define va_start(ap, parmN)  ((void)((ap) = (va_list) \
                                ((char *)(&parmN)+((sizeof(parmN)+1) & 0xFFFE))))

#define va_arg(ap, type)     (*(type *)(((*(char **)&ap))+= \
                                ((sizeof(type)+1) & 0xFFFE))-(((sizeof(type)+1) & 0xFFFE)))

#define va_end(ap)           ((void)0)
```

Není to tedy nic extrémně složitého a jde jen o komplikovanější způsob adresování zásobníku. K ukázce maker pouze poznamenejme, že se jedná o *vrátný zásobník* (roste k nižším adresám) a nejmenší jednotkou v něm je *word* (slovo) uložený na sudé adrese (proto `+1) & 0xFFFE`).

Celkově tedy soubor `<stdarg.h>` poskytuje uživateli jeden nový datový typ `va_list` a tři makra s parametry `va_start()`, `va_arg()` a `va_end()`, které využívají proměnnou datového typu `va_list`.

Makra mají následující význam:

- `va_start()` – inicializuje adresu prvního proměnného parametru v zásobníku,
- `va_arg()` – poskytne hodnotu tohoto parametru a nastaví adresu dalšího parametru – zpřístupní jednorázově parametr,
- `va_end()` – standardní ukončení práce se zásobníkem.

Poznámka:

- Z výše uvedených výpisů je zřejmé, že v u uvedeném příkladě nemá `va_end()` žádnou užitečnou funkci, ale pro jiné překladače může být nezbytné – je ho tedy dobré použít např. z důvodu dobré přenositelnosti.

Ačkoliv uvedená makra již zcela stačí k naprogramování libovolné FPPP, ANSI C přesto nabízí ještě další pomocné nástroje v podobě výpisových funkcí.

20.4 Funkce `vprintf()`, `fprintf()`, `vsprintf()`

Protože FPPP jsou nejčastěji velmi podobné funkci `printf()` a její filozofii⁷, nabízí ANSI C ještě tři funkce pro výstup:

- `vprintf()` – výstup proměnného počtu parametrů na `stdout`,
- `fprintf()` – výstup proměnného počtu parametrů do souboru,
- `vsprintf()` – výstup proměnného počtu parametrů do řetězce.

Všechny tyto funkce pracují v součinnosti s makry ze `<stdarg.h>` a všechny musí bezpodmínečně dodržovat formátové konvence známé z funkce `printf()`, tedy „%“ označuje další parametr a písmeno za „%“ jeho typ.

Pokud chceme např. jen trochu poopravit funkce `printf()`, `fprintf()`, `sprintf()` – nejčastěji pro chybové výstupy, je práce s proměnným počtem parametrů velmi jednoduchá – viz str. 369.

Poznámka:

- Tak jako pro výstup pomocí `printf()`, `fprintf()` a `sprintf()` existují „duální“ funkce pro vstup `scanf()`, `fscanf()` a `sscanf()`, existují i funkce `vscanf()`, `vfscanf()`, `vsscanf()`. Tyto funkce se ale nepoužívají příliš často.⁸

⁷ Počet a typy skutečných parametrů se určí z prvního parametru, který je typu řetězec.

⁸ Osobně jsem se s jejich použitím setkal pouze jednou a informace o `vprintf()`, atp. zde uvedené stačily pro úspěšné zvládnutí problému.

20.5 Praktické příklady použití FPPP

20.5.1 Typické použití `vprintf()`, `fprintf()`, `vsprintf()`

Na příkladu jsou zajímavé tyto skutečnosti:

- 1) Pevných formálních parametrů FPPP může být i více, ale musí být nejméně jeden.
- 2) Do makra `va_start()` se dosazuje vždy jméno posledního pevného formálního parametru.
- 3) Proměnné parametry mohou být zpracovány i opakovaně, ale je nutné jejich seznam vždy „inicializovat“ pomocí `va_start()`.⁹

/ Chybový výpis pomocí `vprintf()`, ... */*

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
```

/ tiskne chybové hlášení na `stdout`, `stderr`*

** a zapisuje ho do řetězu `str` */*

```
void chyba(char *str, char *format, ...)
```

```
{
```

```
    va_list argumenty;    /* pomocná proměnná */
    char      *p_pom;
```

/ využití `vprintf()` - výstup na obrazovku - `stdout` */*

```
printf("\nChyba - obrazovka: ");
```

```
va_start(argumenty, format);
```

```
vprintf(format, argumenty);
```

```
va_end(argumenty);
```

/ opakované použití proměnných parametrů */*

/ využití `fprintf()` - výstup na `stderr` */*

```
fprintf(stderr, "\nChyba - stderr: ");
```

```
va_start(argumenty, format);
```

```
fprintf(stderr, format, argumenty);
```

```
va_end(argumenty);
```

⁹ To je výrazný rozdíl od běžných formálních parametrů funkce, ke kterým můžeme ve funkci přistupovat libovolněkrát bez jakékoliv „inicializace“.

```

/* opakovane pouziti promennych parametru */
/* vyuziti vsprintf() - vystup do retezce */
p_pom = strcpy(str, "\nChyba - string: ");
p_pom += strlen(str);
va_start(argumenty, format);
vsprintf(p_pom, format, argumenty);
va_end(argumenty);
}

int main(void)
{
    char radka[1000];
    int i = 5;
    float f = 3.14;

    /* zadne promenne parametry */
    chyba(radka, "Zadna chyba\n");
    puts(radka);

    /* dva promenne parametry */
    chyba(radka, "int = %d, float = %f\n", i, f);
    puts(radka);

    /* ctyri promenne parametry */
    chyba(radka, "int = %d, float = %f, %d, %f\n", i, f, i, f);
    puts(radka);

    return 0;
}

```

20.5.2 Přesměrování výstupu za běhu programu

Tento příklad pochází ze skutečné aplikace, kdy byl požadavek, aby existovala jedna výpisová funkce, která by volitelně (z menu) za běhu programu uměla psát na obrazovku konzole nebo do souboru, případně, aby šel výstup z důvodu rychlosti výpočtu zcela potlačit.

Protože se směr výstupu stanoví až při běhu programu – např. pomocí volby v menu – nelze ho zadávat jako další parametr funkce, ale musí to být globální proměnná, protože programátor volá funkci `vypis()` stále stejně, tj. bez ohledu na konečný (uživatelé zvolený) směr výstupu.

```

/* Ruzne typy vystupu */

#include <stdio.h>
#include <string.h>
#include <stdarg.h>

typedef enum {
    OBRAZOVKA, SOUBOR, NIC
} VYSTUP;

VYSTUP kam;      /* ovlivneni smeru vypisu */
FILE *fw;         /* vystupni soubor */

void vypis(char *format, ...)
{
    va_list argumenty;
    char radka[1000];

    if (kam == NIC)
        return;

    va_start(argumenty, format);
    vsprintf(radka, format, argumenty);
    va_end(argumenty);

    if (kam == OBRAZOVKA) {
        fputs(radka, stdout);
    }
    else {
        fputs(radka, fw);
    }
}

```



```
int main(void)
{
    if ((fw = fopen("vystup.txt", "w")) == NULL)
        return 1;

    kam = NIC;
    vypis("NIC - ahoj\n");

    kam = OBRAZOVKA;
    vypis("OBRAZOVKA - ahoj\n");

    kam = SOUBOR;
    vypis("SOUBOR - ahoj\n");

    if (fclose(fw) == EOF)
        printf("Chyba při zavírání souboru\n");

    return 0;
}
```

20.5.3 Seznam proměnných parametrů je ukončený domluvenou hodnotou

Pokud pro určení počtu skutečných parametrů FPPP neslouží nějaký typ formátovacího řetězce (což se používá zdaleka nejčastěji) nebo vyhrazený pevný parametr (viz str. 372), je nutné označit konec seznamu skutečných parametrů.

Ve FPPP pak ale není možné použít „triku“ s funkcemi typu `vprintf()` apod., které za nás vše provedou, ale k hodnotám jednotlivých skutečných parametrů je nutné přistoupit pomocí makra `va_arg()`.

Co je ale bezpodmínečně nutné znát, jsou typy skutečných parametrů. Nejčastěji jsou všechny jednoho předem domluveného typu¹⁰ a informace o tomto typu se do FPPP nepředává.

Makro `va_arg()` má dva parametry. První je typu `va_list` a ukazuje na seznam parametrů. Druhý udává typ právě zpracovávaného parametru.

Po získání hodnoty každého parametru v seznamu se testuje, zda je to ukončovací hodnota (nejčastěji `NULL`) a pokud ano, pak FPPP ukončí zpracování parametrů pomocí makra `va_end()`.

¹⁰ Ovšem není to nutné – viz str. 373.

Nedá se obecně říci, kdy je výhodné tento typ FPPP použít, bude to zřejmě při speciálních požadavcích aplikace.

Následující program ukazuje možnost spojení více řetězců do jednoho cílového řetězce, čili zobecněnou standardní funkci `strcat()`.

Na příkladu je zajímavá skutečnost, že proměnné parametry musí být zpracovány postupně (sekvenčně). To je výrazný rozdíl od běžných formálních parametrů funkce, ke kterým můžeme ve funkci přistupovat v libovolném pořadí.

Poznámka k výpustce:

- Funkce užívající výpustku musí mít alespoň jeden pevný parametr a teprve za ním může následovat výpustka, která musí být v seznamu parametrů jako poslední.

Například následující funkci `spoj_retezce()` nelze tedy řešit použitím globální proměnné `radka` a ušetřit tak jeden parametr.

```
char radka[80];
void spoj_retezce(...) /* chyba */

/* Seznam parametru ukončen definovanou hodnotou */

#include <stdio.h>
#include <string.h>
#include <stdarg.h>

/* seznam retezcu musi byt ukoncen parametrem NULL */
void spoj_retezce(char *cilovy, ...)
{
    va_list argumenty;
    char *str;

    cilovy[0] = '\0';

    va_start(argumenty, cilovy);
    while ((str = va_arg(argumenty, char *)) != NULL)
        strcat(cilovy, str);
    va_end(argumenty);
}
```

```
int main(void)
{
    char radka[80];

    spoj_retezce(radka, "ahoj", NULL);
    puts(radka);
    spoj_retezce(radka, "ahoj ", "programatori ", NULL);
    puts(radka);
    spoj_retezce(radka, NULL);
    puts(radka);
    spoj_retezce(radka, "ahoj ", "programatori ", "v C", NULL);
    puts(radka);
    spoj_retezce(radka, "Funkce ", "s ", "promennym ",
                  "poctem parametru", NULL);
    puts(radka);
    return 0;
}
```

20.5.4 Počet proměnných parametrů je předán v pevném parametru

Pro tento případ platí totéž, co v předchozím případě – používá se málo, nelze použít funkce `vprintf()` apod., a je nutné znát předem typy skutečných parametrů.

Následující program je ukázkou vyumělkovaného programu pro sčítání – formální parametr `kolik` udává počet všech skutečných parametrů.

```
/* Pocet parametru predan v pevne parametru */
#include <stdio.h>
#include <stdarg.h>

int secti(intolik, ...) /* scita pouze parametry typu int */
{
    va_list argumenty;
    int pom = 0;

    va_start(argumenty, kolik);
    while (--kolik != 0)
        pom += va_arg(argumenty, int);
    va_end(argumenty);
    return pom;
}
```

```
int main(void)
{
    printf("Soucet je %d\n", secti(1));
    printf("Soucet je %d\n", secti(3, 10, 20));
    printf("Soucet je %d\n", secti(5, 1, 2, 3, 4));
    return 0;
}
```

20.5.5 Odstrašující případ zcela netypického použití

Opět se jedná o ukázkou vyumělkovaného programu pro sčítání, na kterém je však možné dobře ukázat některé potenciální problémy.

Na příkladu jsou zajímavé tyto skutečnosti:

- 1) Je třeba dávat pozor na implicitní konverzi skutečných parametrů a tomu přizpůsobit parametry formální. V případě, že funkce `sumace()` měla zpracovat jako reálný parametr typ `float` (místo `double`), tedy:


```
case 'r' : /* realne cislo */
    *p_real += va_arg(argumenty, float);
    break;
```

 se při volání funkce `sumace()` nepodařilo předat skutečné parametry typu `float` a nepomohlo ani jejich explicitní přetypování na `float`, např. `(float)1.1`. Vždy došlo k implicitní konverzi `float` na `double`.
- 2) Protože překladač nemůže zkontrolovat dokonce ani počty skutečných parametrů, je i zde třeba maximální pozornosti. Je „menším zlem“ uvést více parametrů, protože přebytečné se prostě nevyužijí, než méně parametrů. Za chybějící parametry se totiž dosadí náhodný obsah zásobníku, takže výpočet je chybný, ale k *run-time*¹¹ chybě s největší pravděpodobností nedojde a nejsme tak žádným způsobem varováni.

```
/* Parametry ruzneho typu a neznameho poctu */
```

```
#include <stdio.h>
#include <stdarg.h>
```

```
typedef enum {
    USPECH, CHYBA
} VYSLEDEK;
```

¹¹ Chyba za běhu programu.

```

/* ukazka funkčního prototypu s vypustkou */
VYSLEDEK sumace(int *p_cele, double *p_real, char *format, ...);
int main(void)
{
    int    s_cele;
    double s_real;

    /* vse v poradku */
    if (sumace(&s_cele, &s_real, "crrc", 1, 1.1, 2.2, 2) == CHYBA)
        fprintf(stderr, "Neznamy parametr\n");
    else
        printf("Suma_c = %d \t suma_r = %f\n", s_cele, s_real);

    /* vice parametru, nez je ocekavano */
    if (sumace(&s_cele, &s_real, "crr", 1, 1.1, 2.2, 2) == CHYBA)
        fprintf(stderr, "Neznamy parametr\n");
    else
        printf("Suma_c = %d \t suma_r = %f\n", s_cele, s_real);

    /* mene parametru, nez je ocekavano */
    if (sumace(&s_cele, &s_real, "crrc", 1, 1.1, 2.2) == CHYBA)
        fprintf(stderr, "Neznamy parametr\n");
    else
        printf("Suma_c = %d \t suma_r = %f\n", s_cele, s_real);
    /* chybný typ parametru */
    if (sumace(&s_cele, &s_real, "cdrc", 1, 1.1, 2.2, 2) == CHYBA)
        fprintf(stderr, "Neznamy parametr\n");
    else
        printf("Suma_c = %d \t suma_r = %f\n", s_cele, s_real);
    return 0;
}

```

```

/* scita pouze parametry typu int    (= c - cele)
 *                                nebo double (= r - realne)
 * vraci CHYBA v pripade chybného znaku v retezci format
 * v pripade uspechu vraci USPECH
 */
VYSLEDEK sumace(int *p_cele, double *p_real, char *format, ...)
{
    va_list argumenty;
    char *p_str;

    /* nulovani sumacnich parametru */
    *p_cele = 0;
    *p_real = 0.0;

    va_start(argumenty, format);
    for (p_str = format; *p_str != '\0'; p_str++) {
        switch (*p_str) {
            case 'c' :                /* cele cislo */
                *p_cele += va_arg(argumenty, int);
                break;

            case 'r' :                /* realne cislo */
                *p_real += va_arg(argumenty, double);
                break;

            default :
                va_end(argumenty);
                return (CHYBA);
        }
    }
    va_end(argumenty);
    return (USPECH);
}

```

21 Ladění v jazyce C

Tato kapitola by měla upozornit na některé možnosti ladění programů napsaných v ANSI C. Hned v úvodu dodejme, že si nečiní nárok na úplný výčet všech prostředků.

Neustále opakovanou chybou velké části programátorů je, že nejdříve napíší program a teprve potom se starají o jeho ladění. Tato chyba vyplývá z pocitu, že ladění softwaru „nic nestojí“, pouze trochu času a pár úderů do klávesnice v editoru. Programuje se často metodou pokusu a omylu a typický výrok je: „Zkusím to, a když to nepůjde, tak to „prokrokuju“.“¹

Kdyby ale někdo chtěl použít podobný způsob při návrhu hardware – „Zkusím tu desku bez měřicích bodů, a když to nepůjde, tak navrhnu podobnou, ale s měřicími body.“ – pak by nám tento přístup připadal poněkud neprofesionální.

V obou případech jde však o stejný problém – odhalování potenciálně velmi pravděpodobných chyb, přičemž je skutečně pravda, že zjevné náklady na ladění programu jsou opravdu minimální. Když si ale uvědomíme, jak je velmi časově náročné „prokrokovat“ velký program, pak zjistíme, že pocít „nic to nestojí“, je velmi ošidný. Zvláště u velkých projektů, kdy se jedná o desetitisíce řádek zdrojového textu, je nemožné trávit čas neustálým trasováním všech modulů řádku po řádce. Vyplatí se mít nějaký mechanismus, který dokáže místo chyby alespoň přibližně určit.

Již na začátku programování (při analýze problému) se vyplatí rozhodnout, jakým způsobem a jakými pomůckami budou konkrétní části programu laděny a jakým způsobem bude laděn celek dohromady. Pak je nutné toto rozhodnutí dodržet a už při vytváření programu do něho průběžně vkládat příslušné pomocné části.

Proti tomuto postupu se staví zejména ti, co umí dobře využívat nejrůznější ladicí prostředky – „debuggery“.²

Debuggery jsou v současné době skutečně na tak vysoké úrovni, že se s nimi dá odladit téměř libovolný program, aniž by musel obsahovat další

¹ Neboli: „Teprve potom se začnu starat o nějaké ladění.“

² „Co bych si při programování dělal starosti s laděním, když mám skvělý DEBUGGER.“

části ulehčující ladění. Ovšem, jak už bylo zmíněno dříve, cena, představovaná zbytečnou ztrátou času, je někdy velmi vysoká.

V některých případech jsou však debuggery nedostačující, např. při ladění dynamických procesů, kdy se chyba neprojeví okamžitě a při každém běhu programu, ale jen někdy za souběhu nepříznivých okolností. V těchto případech již musíme použít nějakou ladicí techniku zabudovanou v našem programu.

Obecně je možno říci, že nic není samo o sobě všespasitelné, a je tedy dobré znát, jaké užitečné možnosti v tomto směru v sobě skrývá jazyk C. Známe-li tyto možnosti, je možné je vhodně kombinovat se současným využitím výkonných debuggerů s jedním jediným cílem – maximálně usnadnit a urychlit ladění.

Dalším z dobrých, ale často opomíjených důvodů, proč do programu zahrnout ladicí části, je skutečnost, že jedna věc je počáteční odladění programu a druhá věc je údržba nebo i další rozvoj tohoto programu. Údržba, pod kterou se myslí drobné vylepšování nedostatků či dodatečně nalezených chyb, je nezbytnou součástí péče o vytvořený softwarový produkt a žádná seriózní firma si nemůže dovolit ji opomenout.

Dále bude uveden základní přehled technik, jak do programu začlenit dodatečný ladicí prostředek tak, aby byl užitečný, ale aby nepřekážel, nezdržoval nebo neobtěžoval konečného uživatele zbytečnými výpisy.

Jedná se zejména o:

- 1) ladicí výpisy na `stderr`,
- 2) využití podmíněného překladu C-preprocesoru,
- 3) *self-test* modulů nebo funkcí,
- 4) využití `<assert.h>`,
- 5) prostředky pro ladění dynamických procesů.

21.1 Ladicí výpisy na stderr

Zde se nejedná o ladicí techniku v pravém slova smyslu, ale o zdánlivou maličkost. Ta však může téměř bez práce výrazně zpřehlednit výstup. Využívá se toho, že příkazy:

```
fprintf(stdout, "ahoj\n"); /* totéž je printf("ahoj\n"); */
fprintf(stderr, "ahoj\n");
```

vypíší při základním nastavení výstupů vždy `ahoj` na obrazovku.³

První – primitivní – možnost pro ovlivnění výstupu je skutečnost, že všechny standardní vstupy a výstupy (`stdin`, `stdout` a `stderr`) je také možné uzavřít pomocí standardní funkce `fclose()` stejně, jako kterýkoliv běžný soubor. Užitečná část programu pak vypadá například takto:

```
fprintf(stderr, "Chcete ladicí výpisy ? [A/N]: ");
c = toupper(getchar());
if (c == 'N')
    fclose(stderr);
```

Po volbě „n“ nebo „N“ budou výpisy na `stderr` potlačeny.

Druhou – lepší – možnost je přesměrovat výstupy do logovacího souboru funkcí `freopen()` (podrobně viz str. 306).

```
fprintf(stderr, "Chcete ladicí výpisy do souboru? [A/N]: ");
c = toupper(getchar());
if (c == 'N') {
    fw = freopen("ladieni.log", "w", stderr);
    if (fw == NULL)
        exit(1);
}
```

Shrneme-li obě použité taktiky, pak lze říci, že přesměrování a/nebo uzavření `stderr` či `stdout` nám dovoluje bez zásahu do programu výrazně změnit výpisy na obrazovku. Problémem ale je, že ladicí úseky v programu trvale zůstávají⁴, a tím ho prodlužují a zdržují jeho běh. Proto se těchto triků používá často v kombinaci s možností preprocesoru.

³ Poznamenejme, že `stdin`, `stdout` a `stderr` jsou proudy standardních vstupů a výstupů vždy otvírané po spuštění C programu – viz též str. 82.

⁴ Jsou odstraněny pouze vlastní výpisy, ne jejich kód.

21.2 Využití podmíněného překladu pomocí příkazů C-preprocesoru

Zde využíváme více způsoby základního faktu, že preprocesor umožňuje podmíněný překlad (viz též str. 101). To znamená, že se lze lehce (jednou změnou) zbavit najednou všech určitým způsobem označených částí zdrojového programu. Tyto části programu byly využívány pro ladění a po odladění se stávají nadbytečné, protože zdržují program a obtěžují konečného uživatele zbytečnými výpisy nebo zbytečnými soubory. Přebytkové části kódu zůstávají stále ve zdrojovém souboru, ale nejsou ve výsledném programu.

Nadbytečných částí, pokud je používáme, se většinou chybně zbavujeme tak, že se po odladění zruší editorem. To má dvě nevýhody:

- Často vymažeme něco navíc, takže pak program nechodí a my musíme vymazané řádky postupně a potupně dopisovat zpět.
- Při údržbě programu nebo jeho dalším vývoji zjistíme, že bychom potřebovali stejné ladicí části, které už tam jednou byly.

Rozeznáváme dva následující základní druhy podmíněného překladu.

21.2.1 Podmíněný překlad využívající existence symbolické konstanty

V závislosti na její existenci⁵ jsou (při existenci) nebo nejsou ladicí části do programu zahrnuty. Tato konstanta, na jejíž hodnotě nezáleží, je definována na začátku programu (souboru) a má nějaké jednoznačné, předem domluvené jméno. Kromě symbolické konstanty `NDEBUG` použité v `<assert.h>` (viz str. 385) můžeme jména symbolických konstant ovlivňujících podmíněný překlad volit libovolně. V dalších částech bude používána symbolická konstanta `DBG`.

Základní trik je velmi jednoduchý:

```
#ifdef DBG      /* zacatek podmíneného prekladu */
    ladicí části /* podmínené prekladana cast */
#endif         /* konec podmíneného prekladu */
```

Uzavřeme-li všechny části kódu, které chceme podmíněně překládat, do těchto závorek, pak po definici: `#define DBG` budou tyto (nejčastěji ladicí) části do programu zahrnuty.

⁵ Definování příkazem preprocesoru `#define`.

Vynecháme-li definici symbolické konstanty DBG, pak budou vynechány i ladicí části.

Tento postup vyžaduje jen jednu jedinou změnu na začátku zdrojového programu, a ovšem jeho následné překompilování.

/ Dvoji možnost potlačení ladicích vypisů */*

```
#include <stdio.h>
#include <ctype.h>

#define DBG

int main(void)
{
#ifdef DBG
    int c;

    fprintf(stderr, "Chcete ladicí vypisy ? [A/N]: ");
    c = toupper(getchar());
    if (c == 'N')
        fclose(stderr);
#endif

    /* vykonny kod programu */
    printf("printf - ahoj\n");
    fprintf(stdout, "fprintf - stdout ahoj\n");
#ifdef DBG
    fprintf(stderr, "fprintf - stderr ahoj\n");
#endif

    return 0;
}
```

21.2.2 Podmíněný překlad využívající hodnotu symbolické konstanty

Tento způsob zachovává výhody předchozí techniky a navíc je rozšiřuje o možnost, jak ladicí části jednoduše modifikovat.

/ Ovlivnění ladicích vypisů změnou zdrojového souboru */*
#include <stdio.h>
#include <ctype.h>

```
#define DBG
#define JAK 5 /* 1 = hexadecimalne, 2 = oktalove */

int main(void)
{
    int delenec = 5,
        delitel = 0;

#ifdef DBG
    int c;

    fprintf(stderr, "Chcete ladicí vypisy ? [A/N]: ");
    c = toupper(getchar());
    if (c == 'N')
        fclose(stderr);
#endif

#ifdef DBG && JAK == 1
    fprintf(stderr, "stderr: delitel= %x -hexa\n", delitel);
#elif defined(DBG) && JAK == 2
    fprintf(stderr, "stderr: delitel= %o -octal\n", delitel);
#else
    fprintf(stderr, "stderr: 'Bez ladicích vypisů'\n");
#endif

    printf("stdout: celociselny podil = %d\n", delenec / delitel);
    return 0;
}
```

Samotný příkaz: `#if defined(DBG)`

má naprosto stejný význam jako příkaz: `#ifdef DBG` z předchozího příkladu. Má ale tu výhodu, že ho lze kombinovat s dalšími logickými výrazy, např.: `#if defined(DBG) && JAK == 1`

Příkaz `#elif` má tutéž funkci jako céčkovský `else`, za kterým následuje `if`. Musí tedy za ním následovat nějaký logický výraz.

Část textu za příkazem `#else` bude překládána vždy, když není splněna žádná z předchozích podmínek (totéž co `else` v C).

Poznámka:

- Oba dva předchozí způsoby mají nevýhodu v tom, že je nutná změna⁶ ve zdrojovém souboru – definování konstant DBG popř. i JAK.

Naštěstí kompilátory C umožňují využívat výhod podmíněného překladu i bez zásahu do zdrojového souboru. Symbolickou konstantu lze totiž definovat i vně souboru a to použitím přepínače kompilátoru. Zdrojový program se pak změní takto:

```
/* ladeni.c - ovlivneni ladicich vypisu z vnejsku */
#include <stdio.h>
#include <ctype.h>

/* #define JAK 1    1 = hexadecimalne, 2 = oktaloze */

int main(void)
{
    int delenec = 5,
        delitel = 0;

    #if defined(DBG)
        int c;
        fprintf(stderr, "Chcete ladicí vypisy ? [A/N]: ");
        c = toupper(getchar());
        if (c == 'N')
            fclose(stderr);
    #endif

    #if defined(DBG) && JAK == 1
        fprintf(stderr, "stderr: delitel= %x -hexa\n", delitel);
    #elif defined(DBG) && JAK == 2
        fprintf(stderr, "stderr: delitel= %o -octal\n", delitel);
    #else
        fprintf(stderr, "stderr: 'Bez ladicich vypisu'\n");
    #endif

    printf("celociselny podil = %d\n", delenec / delitel);
    return 0;
}
```

⁶ Jen velmi malá a jen na začátku souboru.

Poznámka:

Způsoby nastavení symbolických konstant:

1) **V okenních vývojových prostředích (IDE)**

IDE jsou samozřejmě různá, takže lze doporučit jen to, že je potřeba hledat v menu někde kolem **Options/Compiler/Code generation/Defines**. Existuje ale i druhá možnost, protože i u okenních prostředí lze často spustit samostatně překladač z příkazové řádky. Pak je syntaxe stejná jako v UNIXu.

2) **V UNIXu pro překladač gcc nebo cc**

Provedeme pomocí přepínače **-D** například⁷:

```
gcc -DDBG -DJAK=1 ladeni.c -o ladeni
```

21.3 Self-testy modulů nebo funkcí

Tohoto způsobu využíváme při odděleném překládání, kdy je program sestaven z více modulů (viz též str. 127). Každý modul má pak v sobě zahrnut samotestovací (*self-test*) mechanismus, pomocí něhož může být kdykoliv otestován zcela nezávisle na ostatních modulech.

Vypnutí samotestovacího mechanismu se provede postupem popsaným na str. 379, tj. využitím příkazů pro podmíněný překlad.

Samotestovací mechanismus znamená, že v každém modulu je zahrnuta funkce `main()` s parametry, které umožňují tento modul⁸ otestovat pomocí vstupu z klávesnice.

Celý uvedený způsob má tu výhodu, že ať ladíme modul pomocí sebevýkonnějšího debuggeru, musíme v něm vždy napsat `main()`, která naši testovanou funkci volá. Po odladění je pak škoda tuto pracně vymyšlenou a napsanou funkci vymazat a při opravě tohoto modulu ji vymýšlet a psát znovu.

Příklad:

Použití tohoto způsobu ukazuje modul `hex-dec.c`, ve kterém je uložena funkce `shex2dec()`. Ta umí převést řetězec obsahující hexadecimální číslo na dekadické číslo a lokalizovat případný chybný vstup.

⁷ Přepínač **-o** označuje, že výsledný program se bude jmenovat `ladení` – bez něj by to byl `a.out`.

⁸ V nejjednodušším případě jeho jedinou funkci.

```

/* hex-dec.c
 * Konverze hexadecimalního řetězce na dekadické číslo
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#ifdef SELF
/* funkční prototyp */
unsigned int shex2dec(char *shex);

int main(int argc, char *argv[])
{
    if (argc == 1) {
        fprintf(stderr, "Použití: hex-dec hexa_cislo\n");
        exit(1);
    }
    printf("%s hexa = %u dek\n", argv[1], shex2dec(argv[1]));

    return 0;
}
#endif

unsigned int shex2dec(char *shex)
{
    unsigned char cislice;
    unsigned int vysledek = 0;

    for (; *shex != '\0'; shex++) {
        if (isxdigit(*shex) == 0) {
            fprintf(stderr, " '%c' není hexa cislice\n", *shex);
            exit(1);
        }
        cislice = toupper(*shex);
        cislice -= (cislice <= '9') ? '0' : 'A' - 10;
        vysledek = (vysledek << 4) + cislice;
    }
    return vysledek;
}

```

Celá funkce `main()` je podmíněně překládána v závislosti na symbolické konstantě `SELF`, která je definována vně souboru (způsoby viz str. 379).

Při definované `SELF` je možné lehce otestovat správnost funkce `shex2dec()` zadáváním čísel z klávesnice. Všimněte si výpisu nápovědy, jak se *self-test* použije. Nápovědu je vhodné vždy uvést, protože na způsob použití časem zcela jistě zapomeneme.

Při ladění v UNIXu budeme postupovat takto:

```
gcc -DSELF hex-dec.c -o hex-dec
```

Vznikne program `hex-dec`, který můžeme spustit a vyzkoušet, takže výstup bude např.:

```

hex-dec
Použití: hex-dec hexa_cislo
hex-dec ff
ff hexa = 255 dek
hex-dec blbost
'1' není hexa cislice

```

Po odladění (nebo jen po ověření správnosti) soubor znovu překládáme, ale ne do spustitelného programu, ale do souboru `.obj` (ve Windows) nebo do souboru `.o` (v UNIXu).

Příkazem⁹: `gcc -c hex-dec.c`

Takto připravený modul je možné kdykoliv sestavit („slinkovat“) s dalšími moduly projektu.

21.4 Využití makra ze souboru <assert.h>

Pro tuto techniku bude dále používáno pojmenování „*aserce*“, i když to možná není terminologicky zcela vhodné.

Před tím, než ukážeme skutečné použití *aserce*, bude vhodné si připomenout typy chyb, se kterými se v programu setkáváme.

V programu se chyby objeví:

1) Během překladu

Ty „máme docela rádi“, protože nás na místo jejich výskytu zcela konkrétně upozorní překladač (i když jeho popis příčiny problému není vždy hned pochopitelný).

⁹ Přepínač `-c` způsobí vytvoření `hex-dec.o`.

2) Během výpočtu vždy a okamžitě

Ty „máme rádi méně“, protože jejich výskyt musíme nalézt sami laděním. Nicméně ladění těchto chyb vyžaduje jen trpělivost.

3) Během výpočtu po dlouhé době nebo jen někdy

Ty „vůbec nemáme rádi“. Jsou to nejhorší chyby, kdy program většinu času pracuje správně, ale občas, vlivem nepříznivé shody okolností, chybně.

Nalezení posledního druhu chyb je časově velmi náročné, ale můžeme se jím částečně bránit například podmíněným překladem testovacích částí¹⁰.

Tento preventivní způsob lze ale zjednodušit použitím podobné techniky, která je předem připravena pro podobné případy – *asercce*.

Asercce je užitečná technika, která nás na možnou chybu upozorní před jejím výskytem, čili před okamžikem, než např. program zhavaruje. Je nutné předem poznamenat, že to není samospasitelná metoda, ale je velmi užitečná např. při práci s poli, řetězci a ukazateli. Při práci s nimi totiž velké množství chyb vzniká čtením nebo zápisem dat ze špatné části paměti. Nejčastěji jsou to u polí chyby „+1“ (*off-by-one-errors*), kdy nás překladač zásadně nevaruje, použijeme-li větší nebo menší meze.

Například příkaz: `buf[-5] = 'a';`

je jasně chybný, ovšem překladači nijak nevadí. Po spuštění se bude tento program chovat různě na různých systémech, někdy proběhne, jindy bude program ukončen chybou.

Právě *asercce* je velmi jednoduchý způsob, jak se těmto chybám bránit. Využívá se toho, že se při běhu programu vyhodnocuje logický výraz, o kterém se předpokládá, že je *TRUE* (nenulový) – pak se *asercce* chová jakoby nic a program běží dále. Pokud je ale *FALSE* (nulový), *asercce* „zabere“, tj. provede chybový výpis a ukončí standardně program před výskytem chyby za běhu programu, tedy před havárií programu.

21.4.1 Praktické použití

Makro, které činnost *asercce* umožňuje, je skryto v souboru `<assert.h>`, před kterým musí být uveden příkaz: `#include <stdio.h>`

Použití *asercce* je pak možné dvěma způsoby:

```
assert(logický_výraz);
assert(logický_výraz) příkaz;
```

¹⁰ Viz např. testovací výpis dělitele v `LADENI3.C` – str. 382.

V obou případech platí, že je-li *logický_výraz* nulový, pak je vypsána chybová zpráva a program je korektně ukončen.

Oba způsoby pracují naprosto stejně a záleží jen na programátorovi, který mu připadá v daném kontextu nejvhodnější.

Praktické použití *asercce* je ukázáno na výseku programu:

```
#include <stdio.h>
#include <assert.h>

#define MAX 20
int main(void)
{
    int cislo, index, pole[MAX];
    char c;
    int *p_pole;

    assert(index >= 0 && index < MAX);
    pole[index] = cislo;

    assert(p_pole != NULL) *p_pole = 5;

    assert(c >= '0' && c <= '9')
        cislo = c - '0';
```

U všech uvedených použití `assert()` je zřejmé, že testované proměnné (`index`, `p_pole`, `c`) mohou za běhu programu nabýt nevhodných hodnot, na což nedokáže upozornit kompilátor, ale *asercce* ano.

Je vidět, že zápis je velmi jednoduchý, bez komplikací s podmíněným překladem¹¹, takže se doporučuje *asercce* hojně využívat.

¹¹ Ten je ovšem schován v `<assert.h>`.

Při programování nejsou pevná pravidla, kdy *aserci* použít a kdy ne. Existují pouze dvě obecná pravidla:

- Použijte ji vždy, kdy při vytváření programu dokážete s určitostí říci, že určitá hodnota proměnné je vždy chybná, ale je velmi malá pravděpodobnost toho, že by se tato hodnota mohla objevit. Z důvodů rychlosti či přehlednosti programu tedy není úplně vhodný pevný test této hodnoty, např. pomocí `if`.
- Použijte ji **vždy** v situacích, o kterých zákazník prohlásí, že „nikdy nemohou nastat“.

Pozor:

- V obou případech ale mějme na paměti, že *aserce* by neměla být náhradou za standardní testy správnosti – viz str. 390.

```
/* aserce.c - ukazka pouziti assert pro kontrolu mezi pole
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAX 10

int main(int argc, char *argv[])
{
    int pole[MAX];
    int index, hodnota;

    if (argc != 3) {
        fprintf(stderr, "Pouziti: %s index hodnota\n", argv[0]);
        exit(1);
    }

    index = atoi(argv[1]);
    hodnota = atoi(argv[2]);

    assert(index >= 0 && index < MAX);

    pole[index] = hodnota;
    printf("pole[%d] = %d\n", index, pole[index]);
    return 0;
}
```

Po spuštění programu s chybnými parametry, např.: `aserce 20 15` se objeví:

Assertion failed:

`index >= 0 && index < MAX, file aserce.c, line 23`

nebo podobný výpis, vždy obsahující jméno souboru, obsah a číslo řádky, na níž došlo k chybě. Z tohoto výpisu je možné okamžitě vyhledat místo chyby a provést příslušná opatření.

Vynecháme-li v programu řádku: `assert(index >= 0 && index < MAX);` pak program bude na některých systémech „fungovat správně“ (tj. vypíše: `pole[20] = 15`) a na některých systémech zhavaruje.

21.4.2 Zrušení *aserce*

Protože je *aserce* založená na podmíněném překladu, je možné se jí po odladění programu lehce zbavit a tím zrychlit a zkrátit program.

To lze provést **definováním** symbolické konstanty se speciálním vyhrazeným jménem `NDEBUG`, buď uvnitř zdrojového souboru příkazem:

```
#define NDEBUG
```

nebo vně tohoto souboru pomocí přepínače kompilátoru.

Po definování symbolické konstanty `NDEBUG` nebudou žádné `assert()` do programu zahrnuty. Je samozřejmě možné do programu *aserci* opět podle potřeby vrátit zrušením definice symbolické konstanty `NDEBUG`.

21.4.3 Princip *aserce*

Definice makra `assert()` je překvapivě jednoduchá:¹²

```
#ifndef NDEBUG
# define assert(p) if (!(p)) { \
    fprintf(stderr, "Assertion failed: %s, file %s, line %d", \
        #p, __FILE__, __LINE__); \
    exit(1); \
} \
else
#else
# define assert(p)
#endif
```

¹² Zde je z důvodu jednoduchosti uveden pouze základní princip, který je však v naprosté většině případů funkční. V konkrétních aplikacích je makro `assert()` trochu komplikovanější.

Zde je vidět využití další možnosti podmíněného překladu, který může záviset i na nedefinici symbolické konstanty – příkaz: `#ifndef NDEBUG`
 Je-li `NDEBUG` definována, bude překládána část:

```
#else
# define assert(p)
```

tedy všechna makra `assert()` se nebudou rozvíjet – budou vynechána.

Není-li `NDEBUG` definována, pak se makro `assert()` rozvine do podmínky `if` a následného tisku.

V příkazu tisku jsou zajímavé příkazy:

`#p` `#` je příkaz preprocesoru, který způsobí, že výraz, který následuje, bude rozvinut uzavřený do uvozovek (např. `"p"`), tedy stane se z něho řetězec.

`__FILE__` předdefinované standardní makro, které se rozvine jako jméno zdrojového souboru, ve kterém se vyskytuje.

`__LINE__` předdefinované standardní makro, které se rozvine jako číslo řádky ve zdrojovém souboru, na které se vyskytuje.

Je zřejmé, že makro `assert()` můžeme libovolně přizpůsobovat dalším potřebám (rozšiřovat jeho možnosti), ale v naprosté většině vystačíme s tím, co už je naprogramováno.

21.4.4 Chybné použití *aserc*

Víme-li, jak *aserc* funguje, lze ukázat její chybné použití například na následujícím běžném programu (`asr.c`), který má dvakrát po sobě vypsat obsah souboru `POKUS.TXT` na obrazovku. Princip chyby je popsán za výpisem chování programu.

```
#include <stdio.h>
#include <assert.h>
```

```
int main(void)
{
    FILE *fr;
    int c;
```

```
if ((fr = fopen("POKUS.TXT", "r")) == NULL)
    printf("Soubor POKUS.TXT se nepodarilo otevrit\n");
else {
    while ((c = getc(fr)) != EOF)
        putchar(c);

    if (fclose(fr) == EOF)
        printf("1) Soubor POKUS.TXT se nepodarilo uzavrit.\n");
}

assert((fr = fopen("POKUS.TXT", "r")) != NULL);
while ((c = getc(fr)) != EOF)
    putchar(c);

if (fclose(fr) == EOF)
    printf("2) Soubor POKUS.TXT se nepodarilo uzavrit.\n");
return 0;
}
```

Následující výpis ukazuje chování programu, kde `kirke.zcu.cz` je použitý počítač a 1., 2., ... jsou doplněná čísla příkazů pro popis ukázky.

```
1. kirke.zcu.cz> cat POKUS.TXT
*** obsah souboru POKUS.TXT ***
2. kirke.zcu.cz> cc asr.c
3. kirke.zcu.cz> a.out
*** obsah souboru POKUS.TXT ***
*** obsah souboru POKUS.TXT ***
4. kirke.zcu.cz> mv POKUS.TXT p.txt
5. kirke.zcu.cz> a.out
Soubor POKUS.TXT se nepodarilo otevrit
Assertion failed:
(fr = fopen("POKUS.TXT", "r")) != NULL, file asr.c, line 20
IOT trap (core dumped)
6. kirke.zcu.cz> mv p.txt POKUS.TXT
7. kirke.zcu.cz> cat POKUS.TXT
*** obsah souboru POKUS.TXT ***
8. kirke.zcu.cz> cc -DNDEBUG asr.c
9. kirke.zcu.cz> a.out
*** obsah souboru POKUS.TXT ***
2) Soubor POKUS.TXT se nepodarilo uzavrit.
```

```
10. kirke.zcu.cz> mv POKUS.TXT p.txt
11. kirke.zcu.cz> a.out
Soubor POKUS.TXT se nepodarilo otevrit
Segmentation fault (core dumped)
```

Význam jednotlivých řádek:

1. Výpis souboru POKUS.TXT na obrazovku standardním příkazem UNIXu `cat`.
2. Překlad souboru `asr.c` překladačem `cc` s umožněním `aserc`.
3. Spuštění výsledného programu – správně se dvakrát vypsalo obsah souboru POKUS.TXT.
4. Přejmenování souboru POKUS.TXT na `p.txt` – „skrytí“ souboru.
5. Druhé spuštění programu – správně se vypisuje chybový výpis o neotevřeném souboru a chybový výpis `aserc` se stejným významem. Do této chvíle se obě části programu chovaly správně.
6. Přejmenování souboru `p.txt` na POKUS.TXT – „obnovení“ souboru.
7. Výpis POKUS.TXT.
8. Nový překlad `asr.c`, tentokrát se zakázáním `aserc`.
9. Spuštění výsledného programu. První část programu funguje správně – vypisuje existující POKUS.TXT. Druhá část nefunguje, protože makro `assert()` se nerozvinulo a tak nebyla vůbec volána funkce `fopen()`. Čtení ze souboru nemohlo proběhnout, protože nebyl správně nastaven ukazatel `fp`. Stejně tak nebylo možné uzavřít neotevřený soubor.
10. Přejmenování souboru POKUS.TXT na `p.txt` – „skrytí“ souboru.
11. Spuštění výsledného programu. První část opět funguje správně – vypisuje chybové hlášení o neotevřeném souboru. Druhá část opět funguje chybně, protože nedojde k pokusu o otevření souboru.

21.5 Prostředky pro ladění dynamických procesů

Poznámka:

- V této části bude popsána pouze filosofie a metodika řešení. Dřívější vydání této knihy obsahovalo ještě jednoduchý příklad (je stále k dispozici jako zdrojový soubor) založený na přerušeních v MS-DOSu. Pro současné operační systémy se mi nepodařilo nalézt dostatečně jednoduchý reprezentativní příklad.

Všechny dosud použité ladicí techniky využívaly toho, že informaci o chybě bylo možné vytisknout, případně zapsat do souboru. Ovšem tisk informací o chybě ale nelze občas použít, nejčastěji proto, že tisk jako periferní operace je v porovnání s probíhajícím výpočtem velmi pomalý.

Druhý důvod, který se často váže k prvnímu důvodu, je, že informace je příliš mnoho. To se často stává, když ladíme například dynamické procesy, které využívají přerušení, nebo komunikační protokoly. U těchto problémů se většinou chyba neprojevuje okamžitě a po každém spuštění programu, ale jen někdy za souběhu nepříznivých okolností třeba po několika tisících akcí.

Pak není možné použít přímo tisky na obrazovku, ani zápis do souboru, ani debugger, protože:

- 1) Použití jakéhokoliv debuggeru nebo tisku (na obrazovku nebo do souboru) např. při ladění přerušení totiž zcela „rozhodí“ časové poměry programu. Tak se může stát, že program při ladění poběží správně, protože k nepříznivému časovému souběhu událostí nikdy nedojde.
- 2) I když nám nevádí pomalost tisku ladicích informací, pak je stejně nevhodné tento způsob použít. Představme si například, že k chybě, která nezpůsobí chybu za běhu programu (havarování programu), dojde třeba až po 100 000 správných průchodech programu. Pak dostáváme soubor o velikosti řádově desítek MB, ve kterém se velmi obtížně vyznáváme.

Použit v tomto případě „krokování“ pomocí debuggeru je jasná absurdita, protože nikdo nevydrží provést tolik kroků.¹³

Smíříme-li se s myšlenkou, že nelze použít ani krokování debuggerem, ani ladicí výpisy, pak nás většinou napadne dát do programu úsek kódu navíc – nejčastěji podmíněný příkaz s čítačem průchodů, který nastavíme podle hodnoty, při které havaroval předchozí běh programu.

```
if (citac_kroku < CISLO_CHYBNEHO_KROKU)
    i = i;    /* prázdný příkaz - pro breakpoint */
```

Podmíněný příkaz zajistí potřebný počet průchodů a zastavení pomocí *breakpointu* těsně před kritickým krokem. Pak můžeme program normálně krokovat pomocí debuggeru.

Bohužel ani tento způsob často nevede k cíli – použití debuggeru totiž rozhodí časové poměry. Pokud nám ani tohle nevádí, pak často potřebujeme pro analýzu chyby znát i poměrně dlouhou historii předchozích akcí, ze které

¹³ Pokud vás teď napadlo, že by bylo možné debugger použít, protože novější verze umožňují vázat *breakpoint* na počet průchodů, pak věnujte pozornost dalšímu odstavci.

po skončení programu dokážeme analyzovat, jak, kde a kdy ke konfliktu začalo docházet. To tedy znamená zaznamenávat určité ladicí výpisy. A zde se dostáváme do konfliktu – ladicí výpisy potřebujeme, ale z časových a/nebo prostorových důvodů je nemůžeme použít.

Celý problém lze řešit, když si uvědomíme, že potřebujeme pouze několik posledních ladicích výpisů, a že je potřebujeme až **po skončení programu**¹⁴.

Postupujeme tedy následujícím způsobem:

- 1) Nejprve odladíme nejhorší chyby, způsobující havárii programu, pomocí debuggeru nebo jiné výše uvedené techniky.
- 2) Zápis ladicích informací budeme provádět ne do souboru nebo na obrazovku, ale co nejjednodušeji¹⁵ do paměti. Tím odpadnou časové konflikty.
- 3) Abychom uschovali jen opravdu potřebný počet posledních akcí, použijeme metodu zápisu ladicích informací v paměti do kruhového bufferu. V něm pak bude uschován jen určitý počet posledních ladicích informací. Při naplnění bufferu se totiž informace začínají zapisovat znovu od jeho začátku, čímž se přemazávají nejstarší informace.
Tak může program běžet libovolně dlouho, aniž bychom se dostali do problémů s omezeným místem v paměti.
- 4) Před skončením programu zavoláme funkci, která запиše obsah kruhového bufferu do souboru. Funkce ovšem provádí „inteligentní zápis“, což znamená, že do souboru doplňuje mnohé další užitečné informace, popřípadě již provádí částečnou analýzu chyby.
- 5) Po skutečném skončení programu si libovolným editorem tento soubor prohlédneme a v klidu lokalizujeme chybu.

Praktické poznámky k uvedenému způsobu:

- 1) Nejprve odhadneme, kolik asi ladicích informací (délku historie) budeme pro lokalizaci chyby potřebovat. Toto číslo¹⁶ zavedeme samozřejmě pomocí symbolické konstanty, neboť první odhad nebývá často správný.
- 2) Na začátku programu alokujeme dynamický buffer o příslušné velikosti a v každé části programu, jejíž aktivitu chceme sledovat, provedeme zápis jednoho bajtu (nebo více podle potřeby) do tohoto bufferu.

¹⁴ Je to obdoba tzv. *Post Mortem Dump* nebo v UNIXu *core dump*.

¹⁵ Neformátovaně – zapisujeme jen binární číslo (obsah proměnné), bez její konverze na řetězec číslic.

¹⁶ Bývá často řádu tisíců.

- 3) Nejvhodnější je zapisovat pomocí makra předdefinované symbolické konstanty označující typ akce, aby bylo zdržení co nejmenší a zápis co nejpřehlednější.
- 4) Je zřejmé, že tento způsob se nedá využít pro programy, které ještě obsahují kritické chyby vedoucí ke zhroucení programu. Pak nám není obsah bufferu (operační paměti) k ničemu, protože funkce, která ho zapisuje do souboru, neproběhne.

Poznámky:

- I tento způsob ladění se dá samozřejmě kombinovat se dříve uvedenými postupy, zejména lze velmi lehce dosáhnout toho, aby po odladění programu nebyly podmíněně překládány části pracující s bufferem.
- Úplně na závěr je dobré si uvědomit, že použitý způsob se nemusí využívat jen pro ladění, ale např. i pro monitorování programu za účelem jeho další optimalizace, nechceme-li pro tento účel využít již hotových *profilerů*.

22 Minimální znalosti při programování pod UNIXem

V této části, která nemá v žádném případě suplovat manuál UNIXu¹, byste se měli dozvědět ty skutečně nejzákladnější dovednosti o přípravě programů v jazyce C pod operačním systémem UNIX.

Více či méně se bude jednat jen o přehled programů s několika jejich parametry, který ale může posloužit i tak, že při jakémkoliv složitějším použití těchto prostředků budete vědět, co přibližně hledat v manuálech nebo pomocí elektronického manuálu `man`.

Poznámka:

- Protože pojem „UNIX“ je velmi široký² a navíc záleží na tom, co administrátor systému nainstaluje, je možné, že některý popisovaný prostředek nebudete moci nalézt. Není to sice příliš pravděpodobné, protože všechny prostředky jsou považovány za základní, ale pokud se to stane, zkuste pátrat po podobně pojmenovaném prostředku.

22.1 Editor

Pokud budete nuceni přecházet mezi různými UNIXy a občas využívat jen řádkový terminál – tedy ne „jen“ nějakou formu X-Windows – je velmi vhodné naučit se alespoň základy editoru `vi`³. Práce s ním je, alespoň pro začátečníka, **velmi nepříjemná**, ale `vi` má tu výhodu, že je skutečně všude a chová se stejně. Popis tohoto editoru naleznete prakticky v každém manuálu k UNIXu.

¹ Vhodná literatura viz např. [Ste94].

² Autor má možnost pohybovat se v prostředí nejméně pěti „skoro stejných“ UNIXů.

³ V současné době se často používá jeho rozšířená verze `vim` – „vi improved“.

Tento editor má také možnost přeložit a spustit program, aniž je nutné editor opustit, což je v případě, že pracujeme u řádkového terminálu, velmi výhodná vlastnost.⁴

Pracujete-li stabilně na jednom systému s X-Windows, je vhodné zjistit, zda systémem běžně používaný editor má nějaké možnosti podpory vytváření zdrojových textů v C. Tuto možnost má množství editorů a pak máme pohodlnější a příjemnější práci například díky barevným zvýrazněním skupin textu, automatickým formátováním příkazů, kontrolou závorek, atd.

Poznámky:

- Pozor na přenášení zdrojových souborů (obecně textových souborů) vytvořených ve Windows do prostředí UNIXu. Konce řádek `<CR><LF>` z Windows mohou dělat překladačům v UNIXu problémy. Těmto problémům se dá úspěšně čelit tím, že pro přenos pomocí FTP (či podobného programu) zvolíte ASCII režim přenosu, který konce řádek opraví.
- Ve jménech souborů dávejte pozor na skutečnost, že UNIX důsledně rozlišuje v názvu souborů velká a malá písmena. Pokud se ve Windows soubor jmenuje `pokus.c`, pravděpodobně se na něj lze odkázat i jako na `Pokus.c` nebo na `POKUS.C`. V UNIXu jsou to vždy tři různé soubory.

22.2 Překladač a sestavovací program

Základní překladač v UNIXu je `cc`. Měl by být dostupný všude. Druhým velmi populárním, kvalitním a volně šiřitelným překladačem je `gcc`⁵. Naštěstí mají oba dva překladače základní přepínače⁶ shodné, takže následující popis je nebude rozlišovat. Oba dva slouží také i jako sestavovací program.

Mějme zdrojový soubor `pokus.c`, pak můžeme použít následující příkazy.

```
gcc pokus.c
```

- přeloží `pokus.c`, sestaví jej a vytvoří spustitelný program pojmenovaný `a.out`

```
gcc pokus.c -o pokus
```

- totéž, ale spustitelný program bude `pokus`

```
gcc -Wall pokus.c -o pokus
```

- totéž, ale `gcc` bude generovat všechna (`all`) varovná hlášení (`-W`)

⁴ Tuto možnost mají ovšem i další editory, např. `emacs`.

⁵ Pochází z projektu GNU, což znamená (rekurzivní ;-)) „GNU's Not Unix“.

⁶ Celkový počet přepínačů u `gcc` je asi 200(!).

```
gcc -DABC pokus.c
    - totéž jako v prvním případě, ale při překladu je z vnějšku definována
    symbolická konstanta ABC (využití je např. při ladění – viz str. 379)
gcc -DABC=1 pokus.c
    - totéž jako v předchozím případě, ale symbolická konstanta ABC bude
    mít hodnotu 1
gcc -c pokus.c
    - pouze přeloží (nesestavuje) soubor pokus.c a vytvoří objektový sou-
    bor pokus.o, který je nutné dále sestavit
```

Máme-li dva zdrojové soubory, kde `hlavni.c` využívá funkce z `pomocny.c`, postupujeme takto:

```
gcc -c pomocny.c
    - pouze přeloží (nesestavuje) soubor pomocny.c a vytvoří objektový
    soubor pomocny.o, který je nutné dále sestavit
gcc pomocny.o hlavni.c
    - přeloží soubor hlavni.c a sestaví ho se souborem pomocny.o
    - výsledný program bude v souboru a.out
gcc pomocny.o hlavni.c -o spustitelny
    - jako předchozí, ale výsledný program bude uložen v souboru se jmé-
    nem spustitelny
```

Poznámky:

- Příkaz: `gcc -ansi pokus.c`
říká překladači `gcc`⁷, že má překládat podle normy ANSI C.
- Pokud si to volbou přepínačů nevyžádáme (nejčastěji `-Wall`), oba překladače hlásí pouze výskyt chyb, avšak žádná varovná hlášení.

Příklad:

Následující program `nedef.c` obsahuje nedefinovanou proměnnou `i`.

```
int main(void)
{
    i = 5;
}
```

⁷ Pro `c++` podobný přepínač neexistuje.

Při překladu pomocí `gcc` se vypíše:

```
$ gcc nedef.c
nedef.c: In function 'main':
nedef.c:3: 'i' undeclared (first use in this function)
nedef.c:3: (Each undeclared identifier is reported only once
nedef.c:3: for each function it appears in.)
```

Při překladu se zapnutými varovnými hlášeními se vypíše:

```
$ gcc -Wall nedef.c
nedef.c: In function 'main':
nedef.c:3: 'i' undeclared (first use in this function)
nedef.c:3: (Each undeclared identifier is reported only once
nedef.c:3: for each function it appears in.)
nedef.c:4: warning: control reaches end of non-void function
```

22.3 Kontrola zdrojových textů v C

Pro velmi detailní analýzu zdrojového programu existuje v UNIXu samostatný program nazvaný `lint`.

Poznámky:

- Na systémech Linux `lint` jako takový nenajdete, ale můžete použít jeho (lepší) náhradu `splint` (dříve zvaný `lclint`) – viz www.splint.org.
- Nechcete-li `splint` instalovat, velmi podobnou službu dostanete použitím `gcc -Wall -pedantic`

Programy typu `lint` dokáží velmi dobře zkontrolovat syntaxi a částečně i sémantiku zdrojového souboru v C. Upozorní na místa, která se jim zdají chybná, nepřenositelná nebo zbytečná. Odhalí výskyt nedosažitelných příkazů⁸, neinicializovaných proměnných, nesprávné skutečné parametry funkcí, přiřazení nevhodných typů a mnoho dalších prohřešků.

Stručně řečeno, `lint` poskytne stejnou nebo lepší službu, jako zapnuté volby hlášení varovných zpráv.

Příklad:

Tento program opisuje znaky z klávesnice až do stisknutí znaku „Z“ a počítá je. Po skončení programu vypíše počet přečtených znaků. Program není úmyslně zcela v pořádku, například chybí vložení `<stdio.h>`.

⁸ Například příkaz bezprostředně za příkazem `return`.


```
main()
{
    int i, j;
    char c;

    while ((c = getchar()) != 'Z') {
        putchar(c);
        i++;
    }
    printf("Zpracovano %d znaku\n", i);
}
```

Při překladu tohoto programu v gcc dostaneme následující varovná hlášení:⁹ (ovšem program se přeloží a sestaví):

```
$ gcc -Wall pro-lint.c
warning: return-type defaults to 'int'
In function 'main':
warning: implicit declaration of function 'getchar'
warning: implicit declaration of function 'putchar'
warning: implicit declaration of function 'printf'
warning: unused variable 'j'
warning: control reaches end of non-void function
```

Zkontrolujeme-li tentýž soubor programem lint, dostáváme:

```
prototype for function getchar is not in scope
prototype for function putchar is not in scope
i may be used before set
prototype for function printf is not in scope
j unused in function main
main() returns random value to invocation environment
function putchar return value is always ignored
function printf return value is always ignored
```

Porovnáním obou výpisů, zjistíme, že lint nalezl o dva „prohřešky“ více než gcc.

Poznámka:

- Samozřejmě, že lint reaguje také na syntaktické chyby v programu (např. použití nedefinované proměnné), ale ty odhalí i kompilátory.

⁹ Výpis je zkrácen o úvodní hlášení čísel řádek a jméno zdrojového souboru.

22.4 Odděleně překládané moduly

Programy se většinou skládají z několika odděleně překládaných modulů (viz str. 398). Při ladění pak bývá netriviální záležitostí udržet pořádek a přehled v tom, které moduly mají či nemají být znovu překládány. Situace může být ještě komplikovanější, pokud některé soubory závisí na jiných souborech – typicky jsou to .c soubory, které závisí na svých .h souborech.

Řešením samozřejmě je vždy vše přeložit a sestavit, např. pomocí nějakého skriptu. Ovšem toto řešení není nijak elegantní a navíc pro velké projekty může trvat dlouho.

Pro tyto případy poskytuje UNIX utilitu make, která dokáže řídit posloupnost akcí v závislosti na mnoha okolnostech.

Poznámka:

- Utilita make se nevyužívá jen pro řízení překladu a sestavení programů v C, ale má mnohem širší použití.

Z prvního – a natně zjednodušeného – pohledu je princip fungování make postaven na dvou faktech:

- 1) Operační systém u každého souboru udržuje přesný čas jeho vzniku nebo modifikace. Tak lze jednoduše poznat „mladší“ a „starší“ soubory.
- 2) Existuje textový soubor, který je nejčastěji pojmenován makefile nebo Makefile¹⁰, kde je popsáno, který soubor závisí na kterém. Dále je zde vždy uvedena případná akce, kterou je nutno provést, když je závislejší soubor „starší“ než soubor, na kterém závisí.

Při splnění těchto dvou faktů je pak činnost make jednoduchá – kontroluje soubor závislostí a vykonává požadované akce.

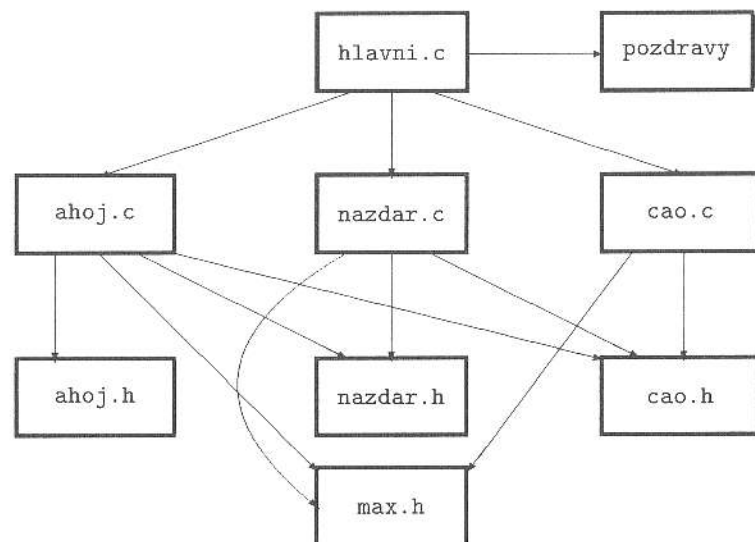
Příklad:

Ukázka jednoduše konstruovaného souboru závislostí Makefile bude provedena na vyumělkovaném případě, kdy je výsledný program složen z několika vzájemně provázaných modulů. Schema závislostí je uvedeno na následujícím obrázku¹¹, z něhož je vidět, že výsledný program pozdravy potřebuje ke své práci soubor hlavni.c a funkce ze souborů ahoj.c, nazdar.c a cao.c. Jejich funkční prototypy jsou v souborech ahoj.h, nazdar.h a cao.h a všechny využívají soubor konstant max.h.¹²

¹⁰ Implicitní parametry programu make.

¹¹ Trochu nepřesně, protože chybí vrstva souborů ahoj.o, nazdar.o a cao.o.

¹² Složitě závislosti tohoto vyumělkovaného příkladu se těžko popisují. Při případných nejasnostech viz příslušné zdrojové soubory.



Soubor Makefile, který řídí celý překlad a sestavování, má následující obsah:

```
# pokusny soubor pro make
```

```
# požadavek ladicích informací
OPT=-O3
```

```
pozdravy: hlavni.c ahoj.o nazdar.o cao.o
    gcc ${OPT} ahoj.o nazdar.o cao.o hlavni.c -o $@
```

```
ahoj.o: ahoj.c ahoj.h nazdar.h cao.h max.h
    gcc ${OPT} -c ahoj.c
```

```
nazdar.o: nazdar.c nazdar.h cao.h max.h
    gcc ${OPT} -c nazdar.c
```

```
cao.o: cao.c cao.h max.h
    gcc ${OPT} -c cao.c
```

V souboru se používají následující prvky:

- Vše, co je za znakem # je komentář, kterých se doporučuje hojně používat.
- `OPT=-O3` znamená, že zavádíme makro¹³ s názvem `OPT`, ve kterém je uveden požadavek na optimalizaci třetího (nejvyššího) stupně. Parametr `-O3` je jedním z parametrů překladače `cc` i `gcc`. Nechceme-li optimalizovat, pak tuto řádku zakomentujeme. Je samozřejmě možné i změnit stupeň optimalizace.

Makra se v Makefile používají velmi často a mají značné možnosti.

- Řádka: `pozdravy: hlavni.c ahoj.o nazdar.o cao.o`
je řádka popisu závislostí a znamená, že výsledný spustitelný program uložený v souboru `pozdravy` závisí na zdrojovém souboru `hlavni.c` a na třech objektových souborech `ahoj.o`, `nazdar.o` a `cao.o`. Závislost znamená, že soubor `pozdravy` musí být „nejmladší“ ze všech souborů, na kterých závisí. Pokud tomu tak není, vykonává se následující řádka (nebo řádky, pokud by jich bylo uvedeno více).

- Význam položek na řádce s výkonnými instrukcemi:¹⁴

```
<Tab> gcc ${OPT} ahoj.o nazdar.o cao.o hlavni.c -o $@
```

- `gcc` je označení požadovaného překladače
- `${OPT}` je volání makra `OPT`
- `ahoj.o nazdar.o cao.o` je seznam objektových souborů, které bude linker přisestavovat
- `hlavni.c` je soubor, který bude překládán
- `-o` je parametr překladače uvozující jméno výsledného programu
- `$@` je volání makra `@`, které se rozvine do jména závislého souboru – zde do `pozdravy`

- Řádka: `ahoj.o: ahoj.c ahoj.h nazdar.h cao.h max.h`
udává, že objektový soubor `ahoj.o` závisí na souborech `ahoj.c`, `ahoj.h`, `nazdar.h`, `cao.h` a `max.h`

- Řádka: `<Tab> gcc ${OPT} -c ahoj.c`
určuje programu `make`, že bude překladačem `gcc` s využitím optimalizace specifikované v makru `OPT` překládán do objektového tvaru (`-c`) soubor `ahoj.c`.

Poté, co připravíme příslušný soubor `makefile`, přestáváme mít s překladem a sestavováním jakékoliv starosti, protože stačí pouze zadat příkaz `make`, který provede vše potřebné.

¹³ Můžeme si jej představit např. jako symbolickou konstantu, ovšem použitelnou pro `make` nikoliv pro jazyk C.

¹⁴ **Pozor:** Tato řádka akcí **MUSÍ** začínat tabulátorem! Velmi častou a silně nepříjemnou chybou je použití mezer.

Příklad:

Pro popisovaný příklad se při prvním spuštění make vypisovalo na obrazovku:

```
gcc -O3 -c ahoj.c
gcc -O3 -c nazdar.c
gcc -O3 -c cao.c
gcc -O3 ahoj.o nazdar.o cao.o hlavni.c -o pozdravy
```

Při změně v souboru hlavni.c vypsalo make následující zprávu:

```
gcc -O3 ahoj.o nazdar.o cao.o hlavni.c -o pozdravy
```

Při změně souboru max.h, na kterém jsou závislé přímo či nepřímo všechny soubory, dostaneme:

```
gcc -O3 -c ahoj.c
gcc -O3 -c nazdar.c
gcc -O3 -c cao.c
gcc -O3 ahoj.o nazdar.o cao.o hlavni.c -o pozdravy
```

Poznámky:

- Jak již bylo řečeno, program make má mnoho možností a předchozí příklad by se dal napsat mnohem úsporněji, ovšem ne tak přehledně.
- Setkáte-li se se soubory makefile při instalaci programů získaných z Internetu ve formě zdrojových souborů, stačí si většinou pozorně prohlédnout definice maker na jeho začátku. Zde je možné změnit např. adresáře, typ překladače, atd. a do dalšího obsahu tohoto souboru nezasahovat.

22.5 Ladicí prostředky

V UNIXovém prostředí je k dispozici celá řada prostředků, od nejjednodušších na řádkovém terminálu až po plně grafické, které jsou zcela srovnatelné s možnostmi debuggerů známých z Windows. Je tedy nutné od administrátora systému zjistit, které prostředky jsou dostupné, zvolit si jeden a ten se naučit používat.

Existuje však jedna dobrá rada, která je dostatečně univerzální. UNIX je operační systém, který nedovoluje uživateli cokoli, a nedovolené akce, které jsou většinou následkem chyb v programu, vedou k nestandardnímu ukončení programu. Průběh tohoto předčasného ukončení je zaznamenán do souboru core, který lze pak např. použít pro nalezení místa výskytu chyby a to velmi jednoduchým způsobem.

Následující program se pokusí přiřadit hodnotu 0 do paměti přes neinicializovaný ukazatel p_i.

```
#include <stdio.h>
int main(void)
{
    int i, *p_i;

    *p_i = 0;
    for (i = 1; i <= 10; i++) {
        *p_i += i;
        printf("i = %d *p_i = %d\n", i, *p_i);
    }
    return 0;
}
```

Po překladu a spuštění je běh programu předčasně ukončen.

```
$ gcc -g neinicializovany.c -o neinicializovany
$ neinicializovany
Segmentation fault (core dumped)
$ ls -l core
-rw----- 1 herout staff 65536 Nov 20 18:04 core
```

Ze vzniklého souboru core, získáme číslo řádky a na něm výpis příkazu, který způsobil havárii, pomocí programu gdb:

```
$ gdb neinicializovany core
...
#0 0x080483fa in main () at neinicializovany.c:6
6      *p_i = 0;
(gdb)quit
```

Poznámky:

- Všimněte si, že překlad byl proveden s parametrem -g, který způsobí zahrnutí dodatečných informací potřebných pro ladění.
- Je možné, že se soubor core nevytvoří. Administrátor systému může totiž generování core souborů vypnout, protože jsou často poměrně velké a zabíraly by vám místo na disku. Pokud jejich vytváření potřebujete povolit, použijte příkaz shellu. Pro bash je to: `ulimit -c velikost` např.: `ulimit -c 10000` povolí coredumpy do 10 tisíc bajtů velké. Pro csh je to: `limit coredumpsize velikost` např.: `limit coredumpsize 10000`

23 Změny v C podle nového standardu ISO

Poznámky:

- Informace v této kapitole jsou čerpány z [Vi02], kde je uveden velmi podrobný rozbor. Zde jsou uváděny pouze věci, které jsem považoval za obecně prospěšné, což je samozřejmě subjektivní výběr. V případě zájmu o tuto problematiku doporučuji přečtení nejdříve celého článku [Vi02] a poté i nahlédnutí do normy.
- Pozor na skutečnost, že nejrozšířenější komerční překladače jazyka C, dodávané jako součást vývojových nástrojů pro C++, množství těchto novinek k dnešnímu dni (prosinec 2003) neimplementují nebo implementují pouze částečně. To tedy znamená, že některé příklady si nebudete moci ověřit na svém počítači.
- Mé doporučení tedy je: „Pokud uvažujete o tom, že budete program přenášet mezi platformami, nepoužívejte žádnou z těchto novinek. Je značná pravděpodobnost, že se nepodaří program přeložit. Budete-li ale používat jen jeden překladač, kterému tyto novinky nedělají problémy, není žádný důvod, proč jejich výhod nevyužít.“
- Dále uvedené příklady jsem zkoušel pomocí překladačů Visual C++ .NET z roku 2003 a gcc v.2.95.4. V příkladech bude dále uvedeno, zda kompilátor zmiňovanou jazykovou konstrukci podporoval (**vc** – ano, **gcc** – ano), či nikoliv (**vc** – ne, **gcc** – ne). Protože počet podporovaných konstrukcí je překvapivě nízký, berte tuto kapitolu spíše jako příslib do budoucna ;-) Je totiž opodstatněná naděje, že se situace bude postupně zlepšovat.

23.1 Stručný přehled norem a nejdůležitějších změn

Jako základ lze považovat americkou národní normu ANSI C (která byla popisována ve všech předchozích kapitolách), na které se pracovalo od roku 1983 a byla přijata v roce 1988.

Formálně vzato, tato norma platila jen do roku 1990, kdy byla nahrazena mezinárodní normou ISO/IEC 9899:1990. Protože však tato norma má stejný obsah, jako ANSI norma, označoval se jazyk vyhovující této normě stále jako ANSI C.

V polovině devadesátých let byly přijato několik verzí technických dodatků a oprav, které ale komunitu C programátorů příliš neovlivnily.

V roce 1999 pak byla přijata druhá verze mezinárodního standardu jazyka C, označená ISO/IEC 9899:1999. Tato druhá verze v plném rozsahu ruší a nahrazuje předchozí verzi včetně všech oprav a dodatků k ní.

V roce 2001 byly vydány opravy k nové normě pod označením ISO/IEC 9899:1999/Cor. 1:2001(E). Tato norma by měla být v současné době (2003) platná. Ovšem, jak již bylo uvedeno na začátku této kapitoly, platnost normy je jedna věc a podpora výrobců kompilátorů je věc druhá.

Poznámka:

- Dosud není v širším podvědomí zkratka (jako byla zkratka „ANSI C“), pod kterou by nová norma byla obecně známa. Nejčastější označení v literatuře je „C99“, ale to rozhodně není obecně známé. Proto budu dále hovořit o „nové normě“.

Nejdůležitější změny:

- Z jazyka bylo odstraněno pravidlo, známé pod zkratkou „implicitní int“.
- Přibýly nové celočíselné typy.
- Rozšířily se možnosti inicializace polí.
- Lokální pole mohou mít dynamické meze.
- V deklaraci funkce lze použít modifikátor **inline**.
- Součástí jazyka C se staly nástroje a knihovny pro práci s vícebajtovými znaky a znaky v kódování Unicode.
- Důsledkem a součástí změn normy jazyka C je i zavedení několika nových standardních hlavičkových souborů.

23.2 Milé drobnosti

23.2.1 Komentáře do konce řádky

Velmi příjemná věc, inspirovaná C++, jsou jednořádkové komentáře. Začínají znaky `//` a končí na konci řádku, např.:

```
// komentář do konce řádky
```

Implementace: gcc – ano, vc – ano

23.2.2 Definice kdekoli v kódu

V ANSI C bylo třeba v bloku nejprve definovat proměnné a teprve za definicemi mohly následovat příkazy. Nyní lze definice a kód libovolně míchat. Součástí je i definice proměnné typu **auto** nebo **register** v inicializační části cyklu **for**.

```
int pole[10];

for (int i = 0; i < 10; i++) {
    pole[i] = i + 1;
    int j = pole[i];
    printf("%d, ", j);
}
```

Implementace: gcc - ne, vc - ne

23.2.3 Co bylo zrušeno

Nová norma také zlepšila jazyk z ohledu bezpečnosti – zde uvedeme dvě významná vylepšení.

Prvním je, že typ, který funkce vrátí, je třeba explicitně deklarovat. Dřívější definice funkce, např.:

```
secti(int a, int b)
```

která se spoléhala na to, že není-li návratový typ uveden, je implicitně typu **int**, už podle nové normy není možná.

Druhé vylepšení souvisí s prvním. Stanoví, že prvnímu použití funkce musí předcházet buď její definice nebo funkční prototyp. Prakticky to nejčastěji znamená, že je nutné uvádět „include“ standardních hlavičkových souborů (typicky `#include <stdio.h>`), a to i v nejtriviálnějších příkladech.

Poznámka:

- Pokud jste se drželi doporučované štábní kultury, pak vám tato omezení připadají jako zcela samozřejmá a nijak je ve svých programech nepocítíte. Na druhou stranu je pravděpodobné, že opomenutí těchto zásad budou ještě nějakou dobu kompilátory označovat jen jako varování, nikoliv jako chybu.

Implementace: gcc – varování, vc – varování

23.3 Celočíselné datové typy

Celočíselných typů se dotkla řada změn, a některé z nich jsou poměrně závažné. Připomeňme si, že pod souhrnným označením „celočíslné typy“ se skrývají nejen typy, v jejichž jménech se může objevit klíčové slovo **int**, ale i znaky (těm bude věnována samostatná podkapitola – viz str. 420), výčtové typy a další.

23.3.1 Logické hodnoty

V hlavičkovém souboru `<stdbool.h>` jsou popsány věci potřebné pro snazší práci s logickými hodnotami.

Poznámka:

- Víceméně je nedoporučuji používat, protože systém logických hodnot jako hodnot typu **int** je natolik zažitý, že použití specializovaných datových typů atd. způsobí spíše zhoršení čitelnosti programu.

Hlavičkový soubor `<stdbool.h>` přináší:

- _Bool** – datový typ pro vyjádření logických hodnot. Je to celočíselný typ bez znaménka, který může obsahovat hodnoty 0 (nepravda, *false*) a 1 (pravda, *true*).
- true** – makro, poskytující hodnotu „pravda“.
- false** – makro, poskytující hodnotu „nepravda“.
- bool** – makro, které se rozvine v klíčové slovo **_Bool**.

Pro typ **_Bool** jsou definovány implicitní konverze, které odpovídají tradičnímu používání číselných hodnot a ukazatelů v podmínkách. To znamená, že hodnoty různé od nuly se konvertují na „pravda“ a nulové hodnoty všech typů se konvertují na „nepravda“.

```
#include <stdbool.h>
```

```
int main(void)
{
    _Bool b = true;
    bool c;

    if (b == true)
        c = false;

    printf("b = %d, c = %d\n", b, c);
    b = 1 + 2;
```



```
c = 0.0 / 3.14;
printf("b = %d, c = %d\n", b, c);
...
```

Implementace: gcc - ne, vc - ne

23.3.2 Typ long long int

Pro velmi velká čísla máme k dispozici typy **long long int** nebo **unsigned long long int**, kdy rozsah tohoto typu nesmí být menší než typ **long int** respektive **unsigned long int**. (Prakticky by měl mít šířku 64 bitů.)

Literály tohoto typu mají příponu LL nebo ll, stejně jako formátovací příkazy používané např. funkcemi `print()` a `scanf()`.

```
long long int lli = -123456789012345LL;
unsigned long long int llu = 123456789012345LL;

printf("lli = %lld\n", lli);
printf("llu = %llu\n", llu);

printf("Zadej cele cislo:");
scanf("%lld", &lli);
printf("lli = %lld\n", lli);
```

Implementace: gcc - ano, vc - typ long long zná, ale V/V funkce pracují chybně

23.3.3 Celočíselné typy s přesně danou šířkou

Základní celočíselné typy (**short int**, **int**, **long int** a **long long int** a jejich neznaménkové protějšky) musí poskytovat každá implementace. Standard nestanoví, jaký přesně rozsah mají základní celočíselné typy mít (s výjimkou typu **char**). Pouze určuje omezení zdola, např. typ **int** musí mít rozsah alespoň od -32767 do 32767, a požaduje, aby typ **int** byl „přirozený“ pro danou architekturu. To znamená, že by měl odpovídat šířce aritmetických registrů procesoru.

Vedle toho ovšem implementace mohou či musí poskytnout celočíselné typy s určitou přesně danou šířkou (počtem bitů), s určitou minimální šířkou, nejrychlejší typy s určitou minimální šířkou, typy, jež mohou obsahovat ukazatele na proměnné a typy s maximální šířkou. Zpravidla jde jen o jiná jména pro základní celočíselné typy, zavedená pomocí deklarace `typedef` v hlavičkovém souboru `<stdint.h>` (pro gcc je v `<sys/types.h>`).

Celočíselné znaménkové typy s přesně určenou šířkou mají jména tvaru **intN_t**, kde N je číslo vyjadřující počet bitů, tedy **int8_t**, **int16_t**, **int32_t**, atd.

Celočíselné typy bez znaménka s přesně určenou šířkou mají jména tvaru **uintN_t**, tedy **uint8_t**, **uint16_t**, **uint32_t**, ...

Jména tvaru **int_leastN_t**, resp. **uint_leastN_t** označují celočíselné typy se znaménkem, resp. bez něj, se šířkou alespoň N bitů. Každá implementace musí poskytnout implementace pro N rovné 8, 16, 32 a 64.

Jména tvaru **int_fastN_t**, resp. **uint_fastN_t** označují celočíselné typy se znaménkem, resp. bez něj, se šířkou alespoň N bitů, pro něž jsou v dané architektuře výpočty nejrychlejší. Každá implementace musí poskytnout implementace pro N rovné 8, 16, 32 a 64.

Typy **intptr_t**, resp. **uintptr_t** označují celočíselný typ se znaménkem, resp. bez něj, do něhož lze uložit hodnotu typu `void *` a pak ji přenést zpět, aniž se změní. Tyto typy nejsou povinné.

Typ **intmax_t** označuje celočíselný typ se znaménkem, schopný reprezentovat jakoukoli hodnotu jakéhokoli celočíselného znaménkového typu v dané implementaci (tedy celočíselný typ se znaménkem s největším rozsahem). Podobně typ **uintmax_t** označuje celočíselný typ bez znaménka s největším rozsahem. Tyto dva typy jsou povinné.

Poznámka:

- Při použití těchto typů si je ale nutné uvědomit možné potíže s formátovanými vstupy a výstupy (`print()` a `scanf()`), kdy může být obtížné určit správnou formátovou specifikaci, případně je tato specifikace nepřenositelná. Odtud mohou plynout některé velmi nepříjemné chyby. Například na jedné architektuře vyzkoušíme, že **int32_t** se správně čte a vypisuje ve formátu `"%d"`. Na jiné (starší) architektuře to ale má být správně formát `"%ld"`. Z tohoto důvodu doporučuji jejich použití pouze po zralé úvaze.

```
int32_t i32 = 0x7FFFFFFF;
printf("format %d = %d\n", i32);
printf("format %ld = %ld\n", i32);
```

Vypíše:

```
format %d = 2147483647
format %ld = 2147483647
```

Implementace: gcc - ano, vc - ne

23.3.4 Nové formátové specifikace

V souvislosti s novými datovými typy přibýly i další formátové specifikace použitelné pro funkce typu `printf()` a `scanf()`.

```
#include <stdio.h>
#include <stddef.h>

int main(int argc, char **argv)
{
    signed char sc = -1;
    unsigned char usc = 255;
    signed long long int slli = -123456789012345LL;
    unsigned long long int uslli = 123456789012345LL;
    size_t st = 789;
    ptrdiff_t ptr = argv[1] - argv[0];
    /* intmax_t imax = -1234567890;
    uintmax_t uimax = 1234567890;
    */

    printf("%hhhd = %hhhd\n", sc);
    printf("%hhu = %hhu\n", usc);
    printf("%LLd = %LLd\n", slli);
    printf("%LLu = %LLu\n", uslli);
    printf("%z = %z\n", st);
    printf("%t = %t\n", ptr);
    /* printf("%jd = %jd\n", imax);
    printf("%ju = %ju\n", uimax);
    */
}
```

Vypíše:

gcc	VC
%hhhd = -1	%hhhd = -1
%hhu = 255	%hhu = 255
%LLd = %Ld	%LLd = 2045911175
%LLu = %Lu	%LLu = 2249056121
%z = %	%z = z
%t = %	%t = t

Implementace: gcc – částečně, vc – částečně

23.4 Okrajové hodnoty reálných čísel

Se zlepšujícími se instrukčními sadami procesorů stoupají i možnosti práce s reálnými čísly, přesněji řečeno s jejich *okrajovými hodnotami*.

23.4.1 Nekonečno

Jednou z možných hodnot, kterých může reálné číslo v daném zobrazení v počítači nabýt, je kladné nebo záporné nekonečno. Nekonečno vznikne např. jako výsledek dělení nenulového čísla nulou. Platí, že $+\infty$ je větší než jakékoli jiné reálné číslo. Přičtením konečného čísla k nekonečnu dostaneme opět nekonečno atd. S nekonečnem lze tedy do jisté míry počítat podobně jako s „obyčejnými“ čísly.

23.4.2 NaN

Další ze okrajových hodnot, které by měly být reprezentovatelné v reálných číslech, je NaN. Toto označení vzniklo jako zkratka slov *Not a Number*, tj. *není číslo*. NaN vznikne jako výsledek operací, které nemají smysl – např. při násobení $+\infty \times 0$. Standard pro zobrazení reálných čísel rozlišuje navíc tichý (*quiet*) a signalizující (*signaling*) NaN. Tichý NaN představuje možný výsledek, signalizující NaN způsobí výjimku, tedy typický přerušení výpočtu a ohlášení chyby.

23.4.3 Denormální čísla

Může se stát, že výsledek aritmetické operace nebude nulový, bude ale mít tak malou absolutní hodnotu, že ho nepůjde v daném datovém typu vyjádřit v běžně používaném (normalizovaném) tvaru. Některé architektury počítačů umožňují pracovat i s takovými denormálními čísly, tzn. nezaokrouhlí je na 0 nebo na nejmenší nenulové zobrazitelné číslo. Denormální čísla jsou ovšem méně přesná, tj. mají menší počet platných cifer než normalizovaná čísla.

23.4.4 Podpora okrajových hodnot reálných čísel

Nástroje pro práci s okrajovými hodnotami by měly být definovány v rozšířeném standardním souboru `<math.h>`. Ten by měl obsahovat mj.:

INFINITY	– makro pro porovnání, zda je dané číslo nekonečno.
NAN	– makro pro porovnání, zda je dané číslo NaN (pouze v implementacích, které podporují tichý NaN).
isnan()	– makro, které vrátí kladné číslo, má-li jeho skutečný parametr hodnotu NaN
isinf()	– makro pro nekonečno

isnormal() – makro běžné (normalizované) číslo
 fpclassify() – makro, které vrací hodnoty FP_NAN, FP_INFINITE, ..., podle toho, do jaké kategorie patří jeho skutečný parametr.

```
#include <math.h>
...
double d = 5.0;
double nula = 0.0;
double nan;
double nekonečno;

nekonečno = d / nula;
nan = nula / nula;

/* if (nan == NAN)
   printf("%f = není číslo\n", nan);
*/
if (isnan(nan) != 0)
   printf("%f = není číslo\n", nan);
/*
   printf("číslo je v kategorii %d\n",
          fpclassify(nekonečno));
*/
```

Vypíše:

nan = není číslo

Implementace: gcc – malá část, vc – ne

23.5 Pole

23.5.1 Inicializace polí

Při inicializaci rozsáhlých polí jsme občas naráželi na problém, jak elegantně zadat jen některé hodnoty – např. hodnoty prvků polí s vysokým indexem.

Tento problém odstraňují pojmenované inicializátory. Pojmenovaný inicializátor (*designator*) prvku pole se skládá z indexu uzavřeného v indexových závorkách, za nímž následuje znak = a výraz představující počáteční hodnotu.

Pojmenované inicializátory lze samozřejmě kombinovat s nepojmenovanými, např.:

```
int pole[10] = { 1, 2, [8] = 128 };

for (i = 0; i < 10; i++) {
   printf("%d, ", pole[i]);
}
```

Vypíše:

1, 2, 0, 0, 0, 0, 0, 0, 128, 0,

Implementace: gcc – ano, vc – ne

Poznámka:

- Na výpisu je významná ještě jedna věc. Všimněte si, že ostatní položky tohoto **lokálního** pole mají hodnotu 0. Bez inicializace by měly náhodnou hodnotu, protože vznikaly v zásobníku. To tedy znamená, že pokud inicializujeme (jakýmkoliv způsobem) alespoň jeden prvek pole, budou ostatní neinicializované prvky nastaveny na hodnotu 0.

23.5.2 Proměnná délka pole

Pokud definujeme staticky lokální pole, je možné jej definovat s proměnnou délkou, tj. délkou, která není známa v době překladu. Lze také operátorem **sizeof** získat délku tohoto pole v bajtech – i zde musí operátor **sizeof** pracovat až za běhu programu.

```
void vypis_pole(int n)
{
   int pole[n * 2];
   int i, d;
   d = sizeof(pole) / sizeof(int);

   for (i = 0; i < d; i++)
      printf("%d, ", pole[i]);
}
```

```
int main(void)
{
   vypis_pole(2);
   ...
}
```

Implementace: gcc – ano, vc – ne

23.5.3 Minimální velikost pole jako parametr funkce

Pokud byl formálním parametrem funkce pointer, nebylo možné zařídit, aby skutečným parametrem této funkce nebyla adresa pouze jedné proměnné (místo adresy prvního prvku celého pole). V současné době to zařídit lze pomocí modifikátoru `static` v indexu. Tím slibujeme, že při každém volání této funkce bude skutečným parametrem nenulový ukazatel na první prvek pole, jež bude mít alespoň tolik prvků, kolik je jich uvedeno v definici.

```
void vypis_pole(int pole[static 3], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d, ", pole[i]);
}
int main(void)
{
    int a[3], b[2];
    vypis_pole(a, 3);
    vypis_pole(b, 2);
    ...
}
```

Implementace: gcc - ne, vc - ne

23.5.4 Složené literály

V ANSI C bylo možné, aby pointer na `char` ukazoval na řetězcový literál (jinými slovy na inicializované pole znaků), např.:

```
char *p_c = "Ahoj";
```

Pro ostatní pole, např. typu `int`, toto nebylo možné bez definice pomocného inicializovaného staticky definovaného pole.

Nová norma toto umožňuje pomocí *složených literálů* (*composite literal*). Významné použití této možnosti je zejména pro vytvoření konstantního pole, tj. pole, jehož složky nebude možné měnit.

```
int *kpole = (const int[]) {1, 2, 4, 8, 16};
```

```
for (i = 0; i < 5; i++) {
    printf("%d, ", kpole[i]);
}
kpole[0] = 3; /* chyba */
printf("\n%d", kpole[0]);
```

Implementace: gcc - ano, vc - ne

23.6 Funkce

23.6.1 Identifikátor funkce

Zejména pro účely ladění je vynikající vlastnost, kdy je v těle každé funkce automaticky definován lokální identifikátor:

```
static const char __func__[];
```

Jeho hodnotou je znakový řetězec obsahující jméno (identifikátor) dané funkce. Princip použití ukazuje následující část kódu.

```
#ifdef LADENI
#define IDENTIFIKACE printf("-> %s\n", __func__)
#else
#define IDENTIFIKACE
#endif
```

```
void moje_funkce(void)
{
    IDENTIFIKACE;
    printf("telo funkce\n");
}
```

```
int main(void)
{
    moje_funkce();
    ...
}
```

Vypíše:

Při definované symbolické konstantě `LADENI` (viz str. 379):

```
-> moje_funkce
telo funkce
```

Tímto způsobem lze lehce zařídit ladicí výpisy při vstupu do funkcí, při výstupu z nich atd.

Poznámka:

- Identifikátor `__func__` je používán i v makru `assert()` (viz str. 385).

Implementace: gcc - ano, vc - ne

23.6.2 Funkce mohou být vnořené

Pozor:

- Tato vlastnost není popsána v nové normě, ale je to vlastnost zavedená v prostředích GNU. To znamená, že přenositelnost na jiné kompilátory je (a bude) prakticky nulová.

Vřele doporučuji stále používat původně zavedený a tradicí ověřený způsob, protože výhoda snadného přístupu k lokálním proměnným nebo formálním parametrům mi nepřipadá tak velká v porovnání s nevýhodou ztráty přehlednosti programu. Podle mého názoru mají být funkce co nejjednodušší a nejpřehlednější.

V C dosud platila zásada, že všechny funkce jsou na jedné úrovni – neexistovaly vnořené funkce. Skrývání funkce před ostatními funkcemi se dělo na úrovni odděleně překládaného modulu, ve kterém byla daná funkce označena jako **static**.

V GNU C mohou být funkce vnořené (vhnížděné). To má za výhodu skrytí vnořené funkce před ostatními. Další výhodou je možnost přístupu k formálním parametrům a lokálním proměnným vnější funkce. Je možné provést vícenásobné zanoření funkcí, ale pak samozřejmě lze volat pouze funkci bezprostředně vnořenou.

```
void vnejsi(int param)
{
    int lokalni = 5;
    int vnitрни(void)
    {
        printf("vnitрни %d\n", param);
        return lokalni * param;
    }

    printf("vnejsi %d - vnitрни %d\n", param, vnitрни());
}

int main(int argc)
{
    int vnitрни_main1(void) {
        void vnitрни_main2(void)
        {
            printf("vnitрни_main2 %d\n", argc);
        }
    }
}
```

```
    printf("vnitрни_main1 %d\n", argc);
    vnitрни_main2();
    return 1;
}

printf("vnitрни_main1 z main() -> %d\n", vnitрни_main1());

/* NELZE
printf("vnitрни_main2 z main() -> ");
vnitрни_main2();
*/
printf("vnejsi z main() -> ");
vnejsi(3);

/* NELZE
printf("vnitрни_z_vnejsi z main() -> ");
vnitрни();
*/
return 0;
}
```

Vypíše:

```
vnitрни_main1 1
vnitрни_main2 1
vnitрни_main1 z main() -> 1
vnejsi z main() -> vnitрни 3
vnejsi 3 - vnitрни 15
```

Implementace: gcc – ano, vc – ne

23.6.3 Modifikátor inline

V definici funkce je možné použít klíčové slovo **inline**. Pomocí něj říkáme kompilátoru, že volání této funkce má být co nejrychlejší. To může znamenat, že se tělo funkce dosadí do místa volání (stejně, jako je tomu u makro s parametry).

Pro funkce modifikátorem **inline** platí mj. následující omezení:

- Funkce nesmí obsahovat definici statické lokální proměnné a nesmí takovéto proměnné používat.
- Pokud používáme odděleně překládané moduly, je třeba si uvědomit, že **inline** funkce nemůže být exportována do ostatních modulů. Je tedy lokální pouze pro modul v němž je definována.


```

inline int faktorial(int n)
{
    int f = 1;

    for (; n > 1; n--)
        f *= n;

    return f;
}

int main(void)
{
    printf("%d! = %d\n", 5, faktorial(5));
    return 0;
}

```

Implementace: gcc - ano, vc - ano

23.7 Vícebajtové znaky

V nové normě jazyk C také reaguje na stále postupující oblibu univerzální znakové sady Unicode (podrobnosti o Unicode viz [Her01]). Znaky v tomto kódování jsou 16bitové a nová norma proto dává k dispozici jak datové typy, tak i funkce pro práci s nimi.

Poznámky:

- Nástroje pro vícebajtové znaky nejsou žádnou novinkou. Poprvé se objevily v polovině 90 let jako dodatek ISO 9899:1990.
- Při používání vícebajtových znaků je třeba si uvědomit, jak velká část jazyka C je jimi ovlivněna. Je to veškerá práce s řetězci, všechny formátované vstupy a výstupy, práce se soubory atd. Když si uvědomíte, kolik místa výkladu tyto části zabraly v předchozích kapitolách, je jasné, že zde nemůže být uveden podrobný výklad, ale pouze několik ilustrativních příkladů.
- Dále uvedený výklad také není příliš přesný, protože většinu konstrukcí nebylo možné ověřit programově a bylo tedy nutné jen se „dohadovat“ ohledně jejich použití.
- Pokud bych se já osobně setkal s nutností zpracovávat vícebajtové znaky, pak bych s vysokou pravděpodobností volil programovací jazyk Java, který s nimi pracuje naprosto přirozeně a všechny souvislosti má domyšleny, vyřešeny a **implementovány!**

23.7.1 Kódování a definice proměnných

Pro vícebajtové (často též „široké“) znaky je k dispozici typ `wchar_t`, definovaný v hlavičkovém souboru `<wchar.h>`.

Druhý používaný datový typ je `wint_t` (*wide integer*). Jedná se o celočíselný typ schopný pojmut kromě znaků také hodnotu WEOF (obdobu EOF).

Pro vyjádření znakových konstant lze použít, kromě možností známých z dřívějších (viz str. 34), také kódy znaků v kódování Unicode ve tvaru `\uhhhh`, kde `h` označuje šestnáctkovou číslici. Tyto kódy lze získat z mezinárodního standardu ISO/IEC 10646 a je možné je použít i ve znakových řetězcích (s výjimkou těch, které jsou vyhrazené standardem pro řídicí znaky).

Pozor:

- Před úvodní uvozovkou řetězcové konstanty je nutné napsat písmeno `L`.

```

#include <wchar.h>
wchar_t wstr[] = L"\u010Cesk\u00FD\x21\n"; /* Český! */
wchar_t s[] = {'\u010C', 'e', 's', 'k', '\u00FD', '\x21', '\n'};

```

Implementace: gcc - částečně, vc - ano

23.7.2 Práce s jednotlivými znaky

Makra pro práci s jednotlivými znaky jsou popsána v `<wctype.h>` (u některých překladačů ale v `<ctype.h>`). Jedná se o upravená již dobře známá makra (viz str. 96), dělená opět do dvou skupin.

První skupinu tvoří makra pro test znaku. Jejich formálním parametrem je vždy typ `wint_t` a vracejí typ `int`, kdy 0 znamená „test nevyhověl“ a nenulová hodnota pak „test vyhověl“.

Jsou to makra:

```

iswalnum(), iswalpna(), iswdigit(), iswxdigit(), iswlower(), iswupper(),
iswpunct(), iswspace(), iswcntrl(), iswascii(), iswgraph() a
iswprint().

```

Druhou skupinu tvoří dvě makra pro převod velkých písmen na malá a naopak. Jsou to: `wchar_t towlower(wchar_t c)` a `wchar_t towupper(wchar_t c)`.

Potřebujeme-li převést jednobajtové znaky na vícebajtové, pak použijeme funkci `wint_t btowc(int c)`. Pro opačný převod pak funkci `int wctob(wint_t wc)`.

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t w1 = '1';
    wchar_t wv = L'\u010C';
    wchar_t wm;
    char c = 'a';
    wint_t wc;

    if (iswalnum(w1) != 0)
        printf("alfanumericky znak\n");
    if (iswupper(wv) != 0)
        printf("velke pismeno\n");
    wm = towlower(wv);
    printf("%04X -> %04X\n", (int) wv, (int) wm);
/*
    wc = btowc(c);
    c = wctob(w1);
*/
    ...
}
```

Implementace: gcc – částečně, vc – částečně

23.7.3 Práce s řetězci

Stejně jako existují modifikace maker pro práci s jednotlivými znaky, existují i modifikace funkcí pro práci s řetězci. Jejich funkční prototypy jsou uloženy v souboru <wchar.t>.

Poznámka:

- Není možné použít běžné funkce pro práci s jednobajtovými řetězci, protože v Unicode má každý neakcentovaný znak horní bajt roven 0, což by funkce `str...()` chápaly jako předčasné ukončení řetězce.

Následuje přehled funkcí, kde pro snažší pochopení bude za dvojtečkou uvedena i odpovídající „stará“ funkce. Parametry „nových“ funkcí odpovídají počtem a pořadím starým, liší se pouze v typu, kdy `char` je nahrazen `wchar.t`.

<code>wscpy()</code>	: <code>strcpy()</code>	<code>wscat()</code>	: <code>strcat()</code>	<code>wcslen()</code>	: <code>strlen()</code>
<code>wscmp()</code>	: <code>strcmp()</code>	<code>wchr()</code>	: <code>strchr()</code>	<code>wsstr()</code>	: <code>strstr()</code>
<code>wscncat()</code>	: <code>strncat()</code>	<code>wscncpy()</code>	: <code>strncpy()</code>	<code>wscrchr()</code>	: <code>strrchr()</code>

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t wstr1[10] = L"Ahoj";
    wchar_t wstr2[10];
    int delka;

    wcsncpy(wstr2, wstr1);
    delka = wcslen(wstr2);
    printf("%d\n", delka);
    ...
}
```

Implementace: gcc – ano, vc – ano

23.7.4 Vstupy a výstupy

Je samozřejmě pěkné umět pracovat s vícebajtovými znaky, ale pokud se nám nepodaří je správně načíst ze vstupů a správně vypsát na výstupy, pak jejich hodnota rapidně klesá.

Pro práci se soubory používáme stále stejnou datovou strukturu `FILE` a funkce `fopen()` a `fclose()`, přesně tak, jak jsme je používali dosud. Každý otevřený soubor (přesněji proud dat) má ale svoji orientaci buď na jednobajtové nebo vícebajtové znaky.

Po otevření pomocí `fopen()` není orientace stanovena (`fopen()` nemá žádný dodatečný parametr pro tuto činnost). Orientace se stanoví buď prvním použitím I/O funkce – použije-li se funkce pro jednobajtové znaky, bude orientace proudu jednobajtová a naopak. Druhou možností stanovení orientace je volání funkce `fwide()`, která jednoznačně stanoví orientaci na vícebajtové znaky.

Funkce pro práci s vícebajtovými znaky jsou popsány v souboru <wchar.t>.

Následuje přehled funkcí, kde pro snažší pochopení bude za dvojtečkou uvedena odpovídající „stará“ funkce. Parametry „nových“ funkcí odpovídají počtem a pořadím starým, liší se pouze v typu, kdy `char` je nahrazen `wchar.t`.

<code>wprintf()</code>	: <code>printf()</code>	<code>fwprintf()</code>	: <code>fprintf()</code>	<code>swprintf()</code>	: <code>sprintf()</code>
<code>wscanf()</code>	: <code>scanf()</code>	<code>fwscanf()</code>	: <code>fscanf()</code>	<code>swscanf()</code>	: <code>sscanf()</code>
<code>fgetwc()</code>	: <code>fgetc()</code>	<code>fgetws()</code>	: <code>fgets()</code>	<code>fputwc()</code>	: <code>fputc()</code>
<code>fputws()</code>	: <code>fputs()</code>	<code>getwc()</code>	: <code>getc()</code>	<code>getws()</code>	: <code>gets()</code>
<code>putws()</code>	: <code>puts()</code>				

Poznámka:

- Při pokusech o ověření funkčnosti narazíte na problém, že funkce `wfprintf()`, `fwscanf()`, atd. předpokládají ve vnějším médiu (tj. v souborovém systému) jednobajtové znaky, nikoli vícebajtové znaky. To znamená, že v proměnných v programu máme sice vícebajtové znaky, ale do souboru se zapíše typicky jednobajtové znaky v kódování CP 1250. Totéž platí při čtení, kdy se také očekávají jednobajtové znaky a pokud je vstupní soubor v kódování Unicode, je načten po bajtech.

Příklad:

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    FILE *fr, *fw;
    wchar_t wstr[100];

    if ((fr = fopen("unicode.txt", "r")) == NULL) {
        printf("neotevren\n");
        return 1;
    }
    if ((fw = fopen("velka.txt", "w")) == NULL) {
        printf("neotevren\n");
        return 1;
    }

    fwprintf(fw, L"%d, %f\n", 123, 3.14);
    fgetws(wstr, 100, fr);
    fputws(wstr, fw);

    fclose(fw);
    fclose(fr);

    return 0;
}
```

Implementace: gcc – chybně, vc – chybně

24 Desatero příkázání pro programátory v C

Poznámky:

- Tato kapitola není dílem autora knihy. Desatero napsal na počátku 90. let minulého století (asi v 1992) pan Henry Spencer (<http://www.lysator.liu.se/c/ten-commandments.html>). Překlad provedl pan Přemek Brada, s jehož laskavým souhlasem je zde uveden (<http://www.kiv.zcu.cz/~brada/desatero.html>).
- Přesto, že se vám budou následující příkázání možná zdát úsměvná, je v nich schováno mnoho programátorských zkušeností a vyplatí se nad nimi popřemýšlet.

1. Budeš hojně `lint`¹ používat i a jeho výrokům bedlivě naslouchati, neboť vpravdě jeho pozornost a úsudek často převyšují tvůj.

Toto je i dnes platná moudrá rada, ačkoli mnohé moderní překladače umí odhalit množství hříchů stejně dobře a `lint` začíná mít problémy způsobené stářím a zesláblostí, a v četných nových neznámých krajinách se nevyskytuje. K podobným účelům sloužit také jiné nástroje, například *Saber C*.

„Hojnému používání“ rozuměj tak, že jeho radami jest záhodno nechat se vést každodenně, spíše než doufat požehnání tvého kódu v záchvatu pozdní kajícínosti. Odlintování programu nikdy dříve nelintovaného podobá se často čištění stájí, jakého bys nepřál ani svým nejzavilejším nepřítelům. Zkušenost některých také praví, že pečlivé naslouchání `lint`ovým slovům je často velikou pomocí při ladění programu.

„Bedlivé naslouchání“ pak neznamená bezhlavou snahu odstranit každíčký bajt `lint`ových výstupů – už jen proto, že jest pro tebe nemožné jej umlčet v některých věcech – nýbrž porozumění důvodům jeho nespokojenosti a rozpoznávání znepokojujících znamení, o nichž se snaží tě zpravit.

¹ Viz též str. 399.

2. **Nebudeš se s NULL ukazatelem přáteliti, neboť toto přátelství vede k hrozným koncům, chaosu a zatemnění mysli.**

Toto jest zjevná známka chybného opisu písem, neboť zde mělo stát „neinicializovaným ukazatelem“ jako obrana proti zmatení konceptu prázdných (neinicializovaných) ukazatelů a makra NULL, o němž se píše dále. Jinak ovšem je význam přikázání zřejmý – neinicializovaný ukazatel ukazuje do krajín obývaných draky, demony a *core dumpy*, jejichž největším potěšením je začít skotačit ve tvých programech, vyrušíš-li je ze spánku. Neinicializovaný (prázdný) ukazatel neukazuje na nulu jakéhokoli druhu, přestože se vyskytují rouhačské kusy kódu, které s tím neuctivě počítají.

3. **Přetypuješ každý parametr předávaný funkci na očekávaný typ, pokud toho typu již není, ačkoli jsi přesvědčen o zbytečnosti svého úsilí. Odplata krutá totiž nečekaně zaskočí ty, kdo opomenou.**

Programátor by měl rozumět typové struktuře svého jazyka, jinak jej může stihnout veliké neštěstí. Jakkoli se tak někteří osídlenci Západního pobřeží mylně domnívají, *int* a *long* jsou rozdílné typy. Doba jejich stejnosti co do velikosti a reprezentace byla krátká, však agónie jež stihne zastávce jejich vzájemné zaměnitelnosti bude trvat navěky, až se 64-bitové stroje stanou běžnými.

Dále pak, na rozdíl od přesvědčení rozšířenému mezi zpátečnickými obyvateli Zkažených východních bažin, není NULL žádného typu ukazatel a musí být správně přetypován pokaždé, když je použit jako parametr funkce.

Slova proroka Ansiho, který dovolil definovat NULL jako by byl typu `void *`, jsou často vytrhávána z kontextu a špatně chápána. Byla totiž míněna jako zvláštní úleva pro případy velikých soužení v divokých končinách. Vpravdě pak každý program musí nalézt svoji cestu Houštinou typů a nespolehat, že pomocí tohoto zřídka dostupného osvobození vyřeší všechny své problémy. Tak či tak velký bůh Dmr², stvořitel C, moudře vybavil jazyk mnoha druhy ukazatelů, ne jen jedním, a tak bude vždy potřeba konvertovat prorokův NULL na žádaný typ.

Možná se někteří domnívají, že nové požehnání v podobě „prototypů“ odstraní nutnost obezřetnosti v otázce typů argumentů. Nikoli, bratří! Za prvé, problém se objeví pokaždé, stojíme-li před podivnými zákonitostmi proměnného počtu parametrů... a ten, kdo neposiloval svoji víru častým cvičením, bude do této léčky zajistě

lapen. Za druhé, zkušenost moudrých praví, že spoléhání se na prototypy otevírá dveře mnoha podivným chybám, a proto si někteří dokonce přáli aby použití prototypů bylo určeno pouze k účelům kontroly chyb, nikoli však k implicitním typovým konverzím. Konečně pak spoléhání se na prototypy přináší velké složitosti v dnešním Reálném světě, kdy mnozí stále lnou ke starým překladačům z rozmazu či z nezbytí, a nikdy nevíš, na jakých strojích bude tvůj kód nucen běhat zítra.

4. **Jestliže hlavičkové soubory, jež používáš, nemají nadeklarovány návratové typy knihovních funkcí, nadeklaruješ je sám s velikou pečlivostí, abys nedošel újmy ty ani programy tvé.**

Moudrý prorok Ansi k tomu dodává, že bys měl své Dodavatele nutit pod pohrůžkou exkomunikace, aby ke svým knihovnám poskytovali hlavičkové soubory s deklaracemi funkcí. Neboť vpravdě jen oni mohou znát přesné formule zařikáadel, která vyvolávají jejich kouzla tím nejlepším způsobem.

Prorok také poznamenává, že je nemoudré a do jam zatracení a podivných chyb vedoucí, aby ses pokoušel cizí knihovní funkce deklarovat sám, jestliže hlavičkové soubory slouží dobře a spolehlivě.

5. **Meze všech polí i řetězců budeš kontrolovati, neboť tam, kde ty použiješ slovo „test“, jiný zajistě napíše „naprostoneuveritelnedlouhoublost“.**

Skutky Velkého červa nás přesvědčily o jednom důsledku této věty, a sice že robustní software nemá nikdy používat `gets()`, kteréžto je vpravdě nástrojem Ďáblovým. Tvá rozlhaní by měla vždy informovat tvé služebníky o mezích tvých polí; a ti z nich, kdo na tyto rady shlížejí spatra či je potichu obcházejí, nechť jsou odkázáni do Krajín Rm³, kde ti nebudou moci již nikdy škodit.

6. **Vrací-li funkce chybový kód jako znamení potíží, budeš sledovati její návratovou hodnotu, ano i tehdy, když kontroly ztrojnásobí délku tvého kódu a způsobí puchýře na tvých prstech. Neboť jestliže se domníváš, že „se to nemůže stát“, budeš zajistě potrestán za svou domýšlivost.**

Jest přáním každého pravověrného, mít po ruce lepší systém pro zpracování chyb, neboť explicitní kontrola návratových kódů je unavující a pokušení kontrolu vynechat je veliké. Leč dokud onen vzdálený den vysvobození nepřijde, nezbyvá nám než jít po klikaté a

² Dennis M. Ritchie

³ „Do krajín neexistence“ – slovní hříčka, protože příkaz UNIXu `rm` maže soubory.

dlouhé cestě s trpělivostí a pozorností, neboť tvůj Dodavatel, Počítač i Software mají zálibu v překvapeních a nerozpakují se vyrobit podivně nesmyslné výsledky den před tvou Obhajobou diplomové práce nebo Velkou dodávkou zákazníkovi.

Někdy je možné, jako v případě `ferror()` ve `stdio.h`, odložit kontrolu chyb až do chvíle, kdy lze otestovat kumulativní výsledek – což často vede ke kratšímu a průhlednějšímu kódu. A ovšem také každý, i ten nejhorlivější, přívrženec by měl s rozmyslem přistupovat k ošetřování funkcí, jejichž selhání je naprosto nepodstatné... Leč pozor, přetypování na `void` je dvousečná zbraň, která bez výčitky prolije tvou vlastní krev.

7. **Dostupné knihovny budeš studovati a snažit se nevymýšleti je znovu bez příčiny, aby tvůj kód byl stručný a čitelný, a tvé dny radostné a produktivní.**

Je bezpočet nečistých pohanů, kteří pohrdají knihovnami z různých nerozumných a pochybných důvodů jako je vzývání Címových bůžků (také známých jako „efektivita“ a „rychlost“). Jest zajisté pravdou, že některé vlastnosti knihoven jazyka C byly nedobře navrženy, ovšem v naprosté většině je lepší a levnější použít výtvorů jiných lidí než se vytrvale snažit znovu vynalézt hranaté kolo. Měl by ses však usilovně snažit porozumět tomu, co tvé knihovny slibují a co nikoli, abys nespolehal na takové jejich vlastnosti, které ti v budoucnu mohou zmizet pod rukama.

8. **Budeš se snažiti, aby tvé programy měly strukturu a účel zřejmý i pro tvého bližního, a budeš proto používatí PRAVOU ŠTÁBNÍ KULTURU, i když ji nemáš v lásce. Neboť učiníš lépe, když svoji tvořivost využiješ pro řešení problémů, než pro vymýšlení báječných novot bránících čitelnosti a porozumění.**

Tato slova naneštěstí působí nejistotu mezi novici a konvertity, kteří neznají starých moudrostí. Ona Pravá Štábní Kultura, o níž se zde hovoří, pochází ze spisů Prvních proroků, Kernighana a Ritchieho. Časté a neutuchající jsou její kritiky od neznalých, dle kterých jest těžké ji používatí, zatímco ve skutečnosti je jen poněkud obtížné se ji naučit a poté se stane průzračně čistou a jasnou, byť snad poněkud citlivou na drobné chyby.

A tak zatímco ty se domníváš, že tvé způsoby štábní kultury vedou k čitelnějším programům, tvoji následovníci za ně nebudou vděční neboť budou nuceni přepisovat tvá díla a proklínat tvé jméno, a ozvěna jejich slov se může dostat až k tvému příštímu zaměstnavateli. Jest mnoho zvyků stále dodržovaných v tomto světě, které

odstraňují třenice a zlepšují produktivitu skrze jejich všeobecné přijetí, a proto zdaleka není tak důležité, jsou-li to optimální řešení. Stejně tak to je i s Pravou Štábní Kulturou.

Nutno jest učiti jednu politování hodnou poznámku na okraj. Fanatičtí přívrženci Zájmeného gestapa způsobili rozruch kolem použití mužského rodu v těchto Přikázáních, neboť věří že veliké snažení a hlasitý křik za účelem rituální očisty jazyka jakýmsi způsobem přispějí k dobru utiskovaných (na jejichž opravdové a bolavé bédnosti se v tom křiku a rozčilení zapomíná). Budou-li tato slova hlášána lidem jednoduché mysli a rychlým k rozčilení, je možné v takových případech používat neutrální výrazy jako „bytost“ apod. jakožto vhodné pseudobiblické termíny očištěné od nákazy Politické nekorektnosti.

9. **Tvé externí identifikátory budou rozlišitelné podle prvních šesti znaků, i když ti toto tvrdé omezení bude ztěžovat život a doba, kdy bylo nutné, se ti bude zdát nekonečně vzdálena. Jinak si budeš rvát vlasy a rozum tvůj tě opustí v onen soudný den, kdy zatoužíš přeložit svůj program na starém systému.**

Někteří unáhlení horlivci ovšem volají „nikoli, milénium přichází a toto učení je zastaralé a není třeba ho dále poslouchat“ – leč vpravdě jsou ve světě mnohé historické systémy a jest znám výrok obávaného boha Murphyho, že právě ve tvé příští práci můžeš takový systém potkat. Zatímco spíš, kuje proti tobě pikle; a proto se probud' a buď bdělý.

Povšimni si přitom, že není nutné aby tvé identifikátory byly pouze šest znaků *douhé*. Jediné, co podle svatého přikázání musíš zachovat, je jedinečnost v rámci prvních šesti znaků. To často není zdaleka tak těžké, jak mnozí tvrdí.

10. **Navěky se pod přísahou zřekneš toho zcestného a zavrženíhodného učení, že „Všude je VAX“⁴ a nebudeš mít nic společného s nevědomými pohany, kteří ji vyznávají. Jen pak budou dny tvých programů dlouhé, ač tvůj dnešní počítač brzy pomine.**

Tato zcestná myšlenka může mít podobný tvar „Všude je Sun“ nebo „Všude je 386“ (tento poslední je přitom obzvláště odporný vynález Satanův), leč slova přikázání platí pro všechny podobné případy stejně. Dej zvláště pozor na nenápadné ale hrozné „Všude jsou 32-bitové stroje“, což je téměř úplně pravdou dnes ale přestane jí být dříve, než se tvůj odborný životopis jen o málo prodlouží.

⁴ Systémy VAX byly velmi úspěšné a velmi rozšířené počítače bývalé firmy Digital Equipment.

Literatura

- [AA88] *Anderson, P.L. – Anderson, G.C.*: Advanced C: Tips and Techniques
Hayden Books, U.S.A., 1988
- [DM88] *Darnel, P.A. – Margolis, P.E.*: Software Engineering in C
Springer-Verlag New York Inc., U.S.A., 1988
- [Eck90] *Eckel, B.*: Using C++
Osborne McGraw-Hill, U.S.A., 1990
- [Her01] *Herout, P.*: Java – grafické uživatelské prostředí a čeština
KOPP, České Budějovice, 2001
- [HRŠ92] *Herout, P. – Rudolf, V. – Šmrha, P.*: ABC programátora
v jazyce C
KOPP, České Budějovice, 1992
- [KR78] *Kerninghan, B.W. – Ritchie, D.M.*: The C Programming Language
Prentice Hall, U.S.A., 1978
Slovenský překlad — Programovací jazyk C
Alfa, Bratislava, 1986
- [KR88] *Kerninghan, B.W. – Ritchie, D.M.*: The C Programming Language
– Second Edition
Prentice Hall, U.S.A., 1988
- [Koe88] *Koenig, A.*: C Traps and Pitfalls.
Addison-Wesley, U.S.A., 1988
- [MQ86] *Miller, L.H. – Quilici, A.E.*: Programming in C
John Wiley & sons, U.S.A., 1986
- [MS88] *Möldner, T. – Steele, P.W.*: C as a Second Language
Addison-Wesley, U.S.A., 1988
- [PB85] *Plum, T. – Brodie, J.*: Efficient C.
Plum Hall Inc., U.S.A., 1985
- [Ste94] *Stevens, R.W.*: Programování sítí operačního systému Unix
SCIENCE, Veletiny, 1994
- [Vi95] *Virius, M.*: Povídání o funkcích v Borland C++.
Bajt, 2/95
- [Vi02] *Virius, M.*: Céčko na přelomu tisíciletí.
Chip, 9/02, 10/02, 11/02

Rejstřík

Pozor:

- Tento rejstřík je společný pro první i druhý díl Učebnice jazyka C (podrobnosti viz v předmluvě). To tedy znamená, že položky rejstříku s čísly stran od 5 do 271 jsou z prvního dílu a od 272 do 429 jsou z druhého dílu.

- | | | | |
|-----------------------------|---------------|-----------------------|------------------|
| abort() | 341 | calloc() | 168 |
| absolutní adresa | 144 | casting | 88 |
| anonymní specifikace | 115 | cc | 397 |
| argc | 227 | clearerr() | 283 |
| argv | 227 | CLK_TCK | 353 |
| ASCII | 26 | clock_t | 353 |
| asctime() | 357 | clock() | 354 |
| ascerce | 385 | CLOCKS_PER_SEC | 353 |
| <assert.h> | 320, 329, 385 | compiler | 24 |
| assert() | 386 | const | 124 |
| assignment | 31 | continue | 55, 62 |
| atexit() | 341 | core | 341, 404, 426 |
| atof() | 202, 339 | CR | 75 |
| atoi() | 202, 339 | ctime() | 357 |
| atol() | 202, 339 | <ctype.h> | 96, 320, 329 |
| auto | 121 | čtení bloku dat | 292 |
| automatické zrušení souboru | 308 | čtení dat | 285 |
| bílé znaky | 26 | čtení řádky | 290 |
| binární režim | 285 | čtení znaku | 287 |
| binární soubor | 273, 276 | __DATE__ | 91, 325 |
| binární vyhledávání | 344 | debugger | 24, 376 |
| blok | 32, 126 | decrement | 36 |
| boolean | 29, 48 | default | 61 |
| break | 55, 61 | define | 91, 94, 322, 326 |
| buffer | 342 | defined | 104, 381 |
| bufferování | 69, 281, 294 | definice | 30, 107 |
| bufferování, žádné | 295 | deklarace | 30, 107 |
| bufferování, řádkové | 295 | dereferenční operátor | 145 |
| bufferování, blokové | 295 | | |

difftime() 356
 dlouhé skoky 332
 do-while 55, 57
 dočasné soubory 342
 dolní mez pole 173
 double 29, 34, 43, 262
 dynamické procesy 392

 EBCDIC 26
 editor vi 396
 editor 23
 EDOM 330
 efektivnost programu 269
 #elif 104, 381
 ellipsis 364
 #else 381
 #endif 102
 environ 328
 environment 328
 EOF 77
 EOLN 75
 epsilon 324
 ERANGE 330
 <errno.h> 320, 322
 errno 284, 322, 326, 330
 error 104
 escape character 34
 .EXE 24
 exit() 66, 342
 explicitní konverze 88
 expression 31
 extern 122

 FALSE 29, 48, 253
 fclose() 73, 79, 281, 299
 <fcntl.h> 272, 309
 feof() 77, 288
 ferror() 283, 292
 fflush() 295, 299
 fgets() 206
 FILE 71, 91, 272, 277, 325
 FILE 390
 <float.h> 320, 323
 float 29, 34, 43, 262
 fopen() 79, 278
 for 55, 58
 formální parametry 110
 fprintf() 73, 286, 366
 fputs() 208
 fread() 292
 free() 165, 167
 freopen() 306
 fscanf() 73, 286
 fseek() 292, 300
 ftell() 292, 301
 funkční prototyp 115
 funkční prototyp neúplný 115
 fwrite() 292

 gcc 325, 397
 generický ukazatel 344
 getc() 73
 getchar() 39, 140
 gets() 205
 globální definice 118
 globální deklarace 117
 GNU 397
 goto 65

 handle 272, 310
 heterogenní datový typ 232
 hexadecimální číslo 42
 hledání 343
 homogenní datový typ 232
 HUGE_VAL 330

 char 29, 192

chyba +1 174
 chyba fread(), fwrite() 302
 chybné použití 390

 if-else 53
 if 52, 381
 #ifdef 381
 #ifndef 390
 #include 25, 97
 INCLUDE 319
 indexovaná proměnná 175
 inicializace 121, 140, 224, 252, 254
 increment 36
 int() 29, 110
 interface 127
 <io.h> 272, 309
 _iob 328
 _IOFBF 296
 _IOLBF 296
 _IONBF 296
 isalnum() 96
 isalpha() 96
 isascii() 96
 iscntrl() 96
 isdigit() 96
 isgraph() 96
 islower() 96
 isprint() 96
 ispunct() 96
 isspace() 96
 isupper() 96
 isxdigit() 96

 jedničkový doplněk 265
 jméno funkce 156
 jméno pomocného souboru 308
 jmpbuf 332

 klíčová slova 27
 komentář 28, 403
 komentář, vlnízděný 102
 kompilátor 24
 konstantní pointer 177
 konstanty desítkové 33
 konstanty osmičkové 33
 konstanty reálné 34
 konstanty řetězcové 35
 konstanty šestnáctkové 33
 konstanty záporné 33
 konstanty znakové 34
 konverze explicitní 87
 konverze implicitní 87
 konverze řetězců 339
 krokování 393
 kruhový buffer 394

 ladicí informace 341
 ladicí program 24
 ladicí prostředky 404
 ladicí výpisy 378, 393
 LF 75
 .LIB 24
 <limits.h> 320, 323
 LINE 325, 390
 linker 24, 128
 lint 399, 425
 literály 35
 <locale.h> 320, 331
 localtime() 356
 lokální definice 118
 long double 29, 34, 43, 262
 long int 29
 long 29, 33, 42
 longjmp() 333
 LSB 266

 main() 32, 110

make 401
 Makefile 401
 malloc() 165, 235
 man 396
 mantisa 324
 maskování 263
 <math.h> 320, 330
 memcmp() 350
 memcpy() 351
 memchr() 350
 memmove() 351
 memset() 351
 mezera 26
 mktime() 359
 MSB 266

 nastavení času 361
 NDEBUG 379, 389
 nebufferovaný stdin 297
 nebufferované V/V 310
 nekonečný cyklus 56, 60
 nelokální skoky 332
 „nikdy nemůže nastat“ 388, 427
 nový řádek 26, 34
 nová stránka 34
 národní prostředí 331
 návrat na začátek řádky 34
 NULL 149, 166, 324
 nulový pointer 195
 nulový řetězec 195, 205

 .OBJ 24
 objektový soubor 403
 operátor čárky 51
 optimalizace 94, 180, 403
 otevření souboru 277
 ošetření chyb 283

 paragraf 166

 PATH 328
 perror 326, 327
 pevné parametry 367
 podmíněný překlad 379
 podmíněný výraz 50
 pointer na pole 219
 pole pointerů 218, 255
 pomocný soubor 308
 porovnání 48
 posun doleva 34
 posun v souboru 300
 preference 269
 preprocesor 23
 printf() 40, 198, 362, 366
 priority operátorů 269
 priority 49
 problémové bajty 287
 procedury 112
 PROMPT 328
 proud 277
 přejmenování souboru 308
 překladač 24
 přenositelnost 273, 319
 přepínač 60
 přesměrování 82
 přesměrování bufferu 296
 přesměrování proudu 306
 přesměrování výstupu 368
 přesnost 324
 přetypování 88
 přiřazení 31, 48
 příkaz 31
 přímý přístup do souboru 300, 302
 pseudonáhodná čísla 340
 ptrdiff() 324
 putc() 73
 putchar() 39

puts() 205
 písknutí 34

 qsort() 343

 raise() 337
 rand() 340
 RAND_MAX 340
 realloc() 182
 record 232
 redirekce 82
 referenční operátor 145
 register 123
 rekurze 95
 remove() 308
 rename() 308
 return 65, 111, 112
 rozhraní 127
 run-time 165
 rychlé dělení 264
 rychlé násobení 264

 řazení 343
 řádka textového souboru 274

 samotestovací mechanismus 383
 scanf() 40, 196, 362
 SEEK_CUR 300
 SEEK_END 300
 SEEK_SET 300
 sekvenční čtení 300
 self-test 383
 sestavovací program 24
 setbuf() 295
 <setjmp.h> 320, 332
 setjmp() 332
 setlocale() 331
 setvbuf() 296

 short 29
 short int 29
 signál 333
 <signal.h> 320, 334
 signed 29, 42
 signed int 266
 SIGQUIT 335
 sizeof 161
 size_t 324
 skutečné parametry 111
 soubor binární 70
 soubor textový 70
 soubor závislostí 401
 sprintf() 202, 339, 366
 srand() 340
 sscanf() 202
 stack 246
 statement 31
 static 108, 122, 129, 141
 <stdarg.h> 320, 337, 365
 STDCL 91, 325
 <stddef.h> 320, 324
 stderr 82, 277, 281, 306, 328, 378
 stdin 82, 277, 281, 306, 328
 <stdio.h> 39, 306, 309, 320, 337
 <stdlib.h> 165, 320, 339
 stdout 82, 277, 281, 306, 328
 strcat() 200
 strcmp() 200
 strcpy() 200
 strcpy() 327
 strcspn() 349
 stream() 277, 299, 310
 strerror() 327
 strftime() 358
 strchr() 200

<string.h> 200, 320, 348
 string 192
 strlen() 200
 strncpy() 201
 strpbrk() 349
 strrchr() 201
 strspn() 348
 strstr() 201
 strtod() 339
 strtok() 349
 strtol() 339
 strtoul() 339
 struct 232
 struktura 232
 switch 60, 255
 symbolické konstanty 253
 sys_errlist 327
 system() 343

 tabulátor 26, 34, 403
 ternární operátor 50
 textový režim 285
 textový soubor 274, 276, 301, 319
 <time.h> 320, 353
 time_t 353
 __TIME__ 326
 time() 356
 tm 353
 tmpfile() 308
 tmpnam() 308
 toascii() 96
 tolower() 96
 toupper() 96
 TRUE 29, 48, 253
 typecasting 88
 typedef 147, 159, 168, 233, 322

 ukončovací znak souboru 289

uložení položek 244
 #undef 93, 326
 ungetc() 84
 union ve struktuře 256
 union 255
 UNIX 333, 396
 UNIXová knihovna 309
 unsigned 29, 33, 42
 unsigned int 266
 unární mínus 35
 unární plus 35
 uzavření souboru 281, 342

 V/V bufferované 272
 V/V formátované 286
 V/V nebufferované 272
 V/V neformátované 286
 va_arg() 365
 va_end() 365
 va_list 365
 variantní záznam 259
 varovné zprávy 399
 va_start() 365
 velikost bufferu 295
 velikost pole 181
 velké projekty 401
 vfprintf() 366
 vlnízděná struktura 238
 vkládané funkce 94
 void 112, 154
 volatile 125
 volání hodnotou 116, 245
 volání odkazem 151, 245
 vprintf() 366
 vsprintf() 366
 vyhledávací klíč 344
 vymaskování 263
 vypnutí bufferování 296

výčtový typ 253
 výpustka 364
 výraz s přiřazením 35
 výraz 31

 while 55

 zaokrouhlování 203
 zápis bufferu na disk 299
 zápis dat 285
 zjištění pozice v souboru 301
 zkrácené vyhodnocení 48
 způsoby otevírání souboru 278
 zpětné lomítko 34
 zrušení souboru 308