

UČEBNICE JAZYKA

C

MZK - Technická knihovna

255765

III. upravené vydání

Pavel Herout

Kopp

255765

Učebnice jazyka C

Daně a Kateřině

Pavel Herout

255 765/a

Nakladatelství KOPP
České Budějovice, 1994

255 765

51/95

202223

MZK-TK Brno



2629991368

Synt 127 694

Microsoft a MS-DOS jsou registrované ochranné známky Microsoft Corporation.

IBM je registrovaná ochranná známka International Business Machine, Inc.

VAX/VMS je registrovaná ochranná známka Digital Equipment Corporation.

UNIX je registrovaná ochranná známka AT&T.

Všechny produkty firmy Borland jsou ochranné známky nebo registrované ochranné známky Borland International, Inc.

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

1. 1. ledna 1995

Lektoři: Ing. Martin Kvoch a Ing. Pavel Šmrha

© Ing. Pavel Herout, 1994

ISBN 80-85828-21-9



Obsah

Předmluva	12
1 Úvod	14
1.1 C — vznik, vývoj, charakteristika	14
1.2 Typografické a syntaktické konvence	16
1.3 Styl psaní programů	16
2 Základní pojmy	17
2.1 Způsob zpracování programu	17
2.2 Základní pojmy v jazyce C	18
2.2.1 Zdrojové a hlavičkové soubory	19
2.2.2 Bílé znaky	19
2.2.3 ASCII tabulka	19
2.2.4 Identifikátory	20
2.2.5 Komentáře	21
3 První začátky s C	23
3.1 Jednoduché datové typy a přiřazení	23
3.1.1 Definice proměnných	24
3.1.2 Přiřazení	25
3.2 Hlavní program	25
3.3 Konstanty	27
3.3.1 Celočíselné konstanty	27
3.3.2 Reálné konstanty	28
3.3.3 Znakové konstanty	28
3.3.4 Řetězcové konstanty (literály)	29
3.4 Aritmetické výrazy	29
3.4.1 Unární operátory	30
3.4.2 Binární operátory	30
3.4.3 Speciální unární operátory	30
3.4.4 Přiřazovací operátory	31

4 Terminálový vstup a výstup	33
4.1 Hlavičkový soubor stdio.h	33
4.2 Vstup a výstup znaku	33
4.3 Formátovaný vstup a výstup	34
4.3.1 Řídící řetězec formátu	35
5 Řídící struktury	40
5.1 Booleovské výrazy	42
5.1.1 Zkrácené vyhodnocování logických výrazů	42
5.1.2 Priority vyhodnocování logických výrazů	43
5.2 Podmíněný výraz — ternární operátor	44
5.3 Operátor čárky	45
5.4 Příkaz if a příkaz if-else	46
5.5 Iterační příkazy — cykly	49
5.5.1 Příkazy break a continue	49
5.5.2 Příkaz while	50
5.5.3 Příkaz do-while	51
5.5.4 Příkaz for	52
5.6 Příkaz switch	54
5.7 Příkaz goto	59
5.8 Příkaz return	59
6 Vstup ze souboru a výstup do souboru	63
6.1 Začátek práce se souborem	65
6.1.1 Otevření souboru pro čtení	66
6.1.2 Otevření souboru pro zápis	66
6.2 Základní operace s otevřeným souborem	66
6.3 Ukončení práce se souborem	67
6.4 Příklady základní práce se soubory	67
6.5 Testování konce řádky	69
6.6 Testování konce souboru	71
6.6.1 Pomocí symbolické konstanty EOF	71
6.6.2 Pomocí standardního makra feof()	72
6.7 Testování správnosti otevření a uzavření souboru	73
6.8 Standardní vstup a výstup	75
6.9 Vrácení přečteného znaku zpět do vstupního bufferu	78

6.10 Různé možnosti otvírání souborů	79
6.11 Rozdíl při zpracovávání textových a bin. souborů v MS-DOSu	80
6.12 Práce s binárními soubory	82
6.12.1 Čtení a zápis do binárního souboru	82
6.12.2 Pohyb v binárním souboru	83
6.12.3 Příklad použití binárního souboru	83
7 Typová konverze	86
7.1 Implicitní typová konverze	86
7.2 Explicitní typová konverze	87
8 Preprocesor jazyka C	89
8.1 Makra bez parametrů — příkaz define	90
8.2 Makra s parametry	93
8.2.1 Předdefinovaná makra	94
8.3 Vkládání souborů — příkaz include	95
8.3.1 Vkládané soubory	96
8.3.2 Standardní hlavičkové soubory	97
8.3.3 Soubor time.h — měření času	98
8.4 Oddělený překlad souborů — I.	99
8.5 Podmíněný překlad	100
8.5.1 Řízení překladu hodnotou konstantního výrazu	102
8.5.2 Řízení překladu definicí makra	103
8.5.3 Operátor defined	104
8.5.4 Direktivy #elif a #error	104
9 Funkce a práce s pamětí	107
9.1 Alokace paměti	108
9.1.1 Statická alokace	108
9.1.2 Dynamická alokace	109
9.1.3 Vymezení paměti v zásobníku	109
9.2 Funkce	109
9.2.1 Definice funkce	110
9.2.2 Procedury a datový typ void	112
9.2.3 Rekurzivní funkce	113
9.2.4 Funkce nevracející int	113
9.2.5 Problémy s umístěním definice funkcí	114
9.2.6 Konverze návratové hodnoty funkce	117

9.2.7	Parametry funkcí	117
	Konverze skutečných parametrů	118
9.3	Oblast platnosti identifikátorů	118
9.3.1	Globální a lokální proměnné	118
9.3.2	Paměťové třídy	122
	Třída auto	123
	Třída extern	123
	Třída static	123
	Třída register	125
9.3.3	Typové modifikátory	126
	Modifikátor const	126
	Modifikátor volatile	127
9.3.4	Bloky	127
9.4	Oddělený překlad souborů — II.	129
9.4.1	Rozšíření platnosti globální proměnné	129
9.4.2	Statické globální proměnné a funkce	130
9.4.3	Jak udržet pořádek ve velkém programu	132
	Doporučený obsah .C souboru	133
	Doporučený obsah .H souboru	135
9.5	Inicializace jednoduchých proměnných	142
10	Pointery	146
10.1	Základy práce s pointery	147
10.1.1	Definice dat typu pointer na typ	147
10.1.2	Práce s adresovými operátory	148
10.1.3	Přiřazení hodnoty pointerům a pomocí pointerů	148
10.1.4	Použití pointerů v přiřazovacích příkazech	149
10.1.5	Nulový pointer NULL	151
10.1.6	Konverze pointerů	152
10.1.7	Zarovnávání v paměti	152
10.2	Pointery a funkce	152
10.2.1	Volání odkazem	152
10.2.2	Pointer na typ void	156
	Pointer na typ void jako pointer na několik různých typů	156
	Pointer na typ void jako formální parametr funkce	157
10.2.3	Pointery na funkce a funkce jako parametry funkcí ...	157
10.3	Jak číst komplikované definice — I.	160
10.4	Definice s využitím operátoru typedef	161
10.5	Pointerová aritmetika	162

10.5.1	Operátor sizeof	163
10.5.2	Součet pointeru a celého čísla	163
10.5.3	Odečítání celého čísla od pointeru	164
10.5.4	Porovnávání pointerů	165
10.5.5	Odečítání pointerů	166
10.6	Dynamické přidělování a navracení paměti	166
10.6.1	Přidělení paměti	167
10.6.2	Uvolňování paměti	169
10.6.3	Příklady přidělování paměti	169
10.6.4	Funkce calloc()	170
10.7	Pointer jako skutečný parametr funkce	171
11	Jednorozměrná pole	175
11.1	Základní dovednosti	175
11.2	Pole a pointery	178
11.2.1	Dynamická pole	179
11.2.2	Podobnost statických a dynamických polí	180
11.2.3	Další zvláštnosti a dovednosti při práci s poli	181
	Práce s celým polem najednou	181
	Přístup do pole pomocí pointerů	181
	Jak zjistit velikost pole	183
11.3	Pole měnící svoji velikost	183
11.4	Pole jako parametry funkcí	185
11.5	Pole pointerů na funkce	189
11.6	Jak číst komplikované definice — II.	190
12	Řetězce	194
12.1	Základní informace a definování řetězců	194
12.2	Práce s řetězcem	197
12.2.1	Čtení řetězce z klávesnice	198
	Čtení řetězce v daném formátu	198
12.2.2	Tisk řetězce na obrazovku	200
12.2.3	Přístup k jednotlivým znakům řetězce	201
12.2.4	Standardní funkce pro práci s řetězcí	202
	Délka řetězce	202
	Kopírování řetězce	202
	Spojení řetězců	202
	Nalezení znaku v řetězci	202

Porovnání dvou řetězců	202
Nalezení podřetězce v řetězci	203
Práce s omezenou částí řetězce	203
Práce s řetězcem pozpátku	203
Převody řetězců na čísla	203
12.3 Formátované čtení a zápis z a do řetězce	204
12.4 Řádkově orientovaný vstup a výstup z terminálu	206
12.4.1 Čtení řádky z klávesnice	206
12.4.2 Výpis řádky na obrazovku	207
12.5 Řádkově orientovaný vstup a výstup ze souboru	207
12.5.1 Čtení řádky ze souboru	207
12.5.2 Zápis řádky do souboru	208
12.6 Řídící řetězec formátu pro tisk	209
12.6.1 konverze	209
12.6.2 modifikátor	210
12.6.3 šířka	210
12.6.4 přesnost	211
12.6.5 příznak	211
12.6.6 Příklady různých formátů tisku	212
13 Vícerozměrná pole	215
13.1 Základní definice a přístup k prvkům	215
13.2 Uložení vícerozměrných polí v paměti	215
13.3 Různé způsoby definice dvourozměrných polí	217
13.3.1 Statické dvourozměrné pole	218
13.3.2 Pole pointerů	218
13.3.3 Pointer na pole	219
13.3.4 Pointer na pointer	219
13.3.5 Výhody a nevýhody předchozích čtyř způsobů	220
13.3.6 Dvourozměrné pole jako parametr funkce	222
13.4 Inicializace polí všech rozměrů	223
13.5 Pole řetězců	224
13.6 Parametry funkce <code>main()</code>	226
13.7 Externí pole všech rozměrů	228
14 Struktury, uniony a výčtové typy	231
14.1 Struktury	231
14.1.1 Definice a základní dovednosti	231

14.1.2 Struktury a pointery	234
14.1.3 Struktury odkazující samy na sebe	235
14.1.4 Struktura v jiné struktuře	237
14.1.5 Alokace paměti pro jednotlivé položky struktury	239
14.1.6 Struktury a funkce	240
14.1.7 Shrnutí poznatků o práci se strukturami	243
14.1.8 Inicializace struktur	244
14.2 Výčtový typ	245
14.3 Uniony	247
14.3.1 Rozdíl mezi variantním záznamem v Pascalu a unionem	251
15 Bitové operace a bitové pole	255
15.1 Operace s jednotlivými bity	255
15.1.1 Bitový součin	256
15.1.2 Bitový součet	256
15.1.3 Bitový exkluzivní součet	257
15.1.4 Operace bitového posunu doleva	257
15.1.5 Operace bitového posunu doprava	257
15.1.6 Negace bit po bitu	258
15.1.7 Způsoby práce se skupinou bitů	258
15.2 Bitové pole	259
16 Tabulka preferencí	262
Literatura	265
Rejstřík	266

Předmluva

Doby, kdy byla u nás dostupná pouze "bible" programátora v C — **The C Programming Language** (ve slovenském překladu **Programovací jazyk C**), jsou dávno ty tam. Publikací o jazyku C existuje v současné době velké množství. Proč tedy vznikla další? Myslím, že právě tento typ knížky o C dosud chybí. Není to referenční manuál, ani příručka uživatele ke konkrétní implementaci jazyka, ani sbírka příkladů, ani učebnice pro vyspělé uživatele, je to prostě učebnice pro **začátečníky a mírně pokročilé**.

Vznik této učebnice zapříčinily hlavně dvě aktivity, díky kterým jsem se naučil něco navíc a začal uvažovat o tom, že by bylo škoda nepokusit se předat tyto znalosti dalším.

První činností bylo, že jsem vyučoval jazyk C a při přednáškách jsem měl možnost zjistit, že spousta věcí, které mě připadají jasné, nemusí být vůbec jasné druhým. Dá se říci, že téměř všechny **Poznámky**, kterými je kniha "prošpikována", vznikly na základě zvědavých dotazů studentů.

Druhá akce představovala moji účast v týmové práci — programování dvou velkých¹ softwareových projektů. Tehdy jsem zjistil, že jedna věc je umět naprogramovat v C jednoduché programy a druhá věc je umět v C programovat tak, aby programy byly co nejvíce přehledné, co nejsnáze modifikovatelné, aby nezpůsobovaly konflikty při spojení s programy ostatních spolupracovníků a mnoho dalších *aby*. Zkušenosti, které jsem zde získal — jak pozitivní, tak i negativní, jsem se také snažil zahrnout do této knížky. Najdete je nejčastěji shrnuty pod zvláštním názvem **Štábní kultura**, ale i jinde. Jsou to zkušenosti, které se nikde moc nedočtete, ale při programování v C vám mohou ušetřit spoustu času a nervů a záleží jen na vás, zda je přijmete či nikoli.

Další věc, o kterou jsem se snažil, je co největší jednoduchost a pak také poukázání na souvislosti. V textu naleznete spousty vzájemných odkazů, které by vám měly pomoci získat jakýsi nadhled na jazyk C.

Pokud již umíte Pascal, pak tuto knížku oceníte dvojnásobně, protože všechny základní konstrukce jazyka C jsou porovnány s odpovídajícími Pascalskými konstrukcemi. Jestliže jste ale s Pascalem dosud nepřišli do styku, neznamena to, že by tahle učebnice nebyla pro vás vhodná — v tomto případě stačí pouze přeskočit pasáže označené **Pascal**.

Některé příklady, které jsou v knize uvedeny, jsem sám nevytvářel, ale převzal jsem je z knih, jež jsou uvedeny v seznamu literatury. Ovšem každý

příklad jsem ověřil a každý jsem musel více či méně upravit, aby svým stylem zapadl do celkového stylu knihy.

Všechny ukázky programů, včetně těch, které jsou uvedeny jen částečně, plus všechny vyřešené příklady z cvičení byly odladěny pomocí překladače fy Borland (TC++ v.1.0) a jejich zdrojové texty si můžete dodatečně objednat na disketě na adrese nakladatelství — jedná se asi o 270 kompletních programů.

To, že si můžete okamžitě ověřit funkčnost všech uvedených příkladů, považuji za významný přínos pro studium jazyka C. Pokud máte tu možnost, rádím vám upřímně, abyste s těmito programy začali okamžitě experimentovat a opravdu řešili všechna cvičení. Jazyk C se totiž nenaučíte čtením sebelepších knih, ale pouze programováním v něm a k tomu vám přeji hodně trpělivosti a zdaru.

Na tomto místě je mou milou povinností poděkovat Martinu Kvochovi a Pavlu Šmrhovi, kteří knížku pečlivě pročetli a přispěli tak výrazně k jejímu zkvalitnění. Pokud se stane, že přes všechnu jejich i moji snahu objevíte nějakou chybu, napište mi prosím na adresu nakladatelství.

Pavel Herout

Poznámky ke třetímu vydání:

Třetí vydání této knihy se liší od prvního a druhého vydání jen minimálně.

- Opravil jsem v něm některé drobné chyby a překlepy, na které mě upozornili moji kolegové, studenti a ostatní čtenáři. Všem upřímně děkuji za jejich snahu a ochotu.
- Přidal jsem vysvětlující text na těch místech, na které jsem byl upozorněn jako na nejednoznačné nebo které by mohly být špatně pochopeny.
- Pouze v knize jsem opravil dále uvedené programy. V porovnání s téměř třemi stovkami ostatních souborů to byla změna jen velmi malá, a proto jsem se rozhodl nedělat na disketě se zdrojovými soubory žádnou změnu. Mimo jiné jsem také chtěl zamezit zmatkům, které by vznikaly distribucí dvou různých disket. Prosím proto čtenáře, který si ke knize zakoupí i disketu s příklady, aby si výše uvedené soubory opravil podle knihy sám, a omlouvám se za toto malé nedopatření.

strana	kapitola	soubor	závažnost chyby
50	5	S50A.C	malá chyba
72	6	S72.C	malá chyba
76	6	S76.C	logická chyba
99	8	S99.C	možné problémy v UNIXu
171	10	S171.C	vada na kráse

¹ "Velký" znamená desetitisíce řádek.

1 Úvod

Jestliže jste s jazykem C nikdy nepřišli do styku, bude možná vhodné přečíst si nejprve následující dva citáty.

“C je jazyk mocný, tajemný a nevyzpytatelný.”

neznámý zvědavý začátečník

“C je jako Porsche: silné, účinné a kompaktní. Programování v C, stejně jako řízení Porsche, může být zajímavé, vzrušující a zábavné — ovládl-li jste ho a umíte-li využít jeho možnosti.”

A. E. Quilici

Oba tyto citáty jsou pravdivé a odporují si pouze zdánlivě. Když začnete číst tuto knihu, budete se zřejmě přiklánět k tomu prvnímu. Pokud si ale po dočtení celé knížky začnete myslet, že je něco pravdy i na tom druhém, pak knížka splnila svůj účel a vy se můžete začít počítat do velké rodiny programátorů v C.

1.1 C — vznik, vývoj, charakteristika

V knihách o jazyku C bývá na tomto místě informace o tom, kdo jazyk C vymyslel, z čeho přitom vycházel a k čemu se C vlastně hodí. Zajímají-li Vás tyto informace, pak Vám následující řádky dají částečnou odpověď.

Jazyk C:

- je univerzální programovací jazyk nízké úrovně (*low level language*),
- má velmi úsporné vyjadřování, je strukturovaný, má velký soubor operátorů a moderní datové struktury,
- není specializovaný na jednu oblast používání,
- pro mnoho úloh je efektivnější a rychlejší než jiné jazyky.

Stačí? Jestliže ne, pak vězte, že:

- C byl navržen a implementován pod operačním systémem UNIX a téměř celý UNIX je v C napsán¹.
- C se ale na UNIX nijak neváže a neváže se ani na jiný konkrétní počítač či operační systém.
- “Jazyk nízké úrovně” znamená, že C pracuje přímo pouze se standardními datovými typy jako jsou znaky, celá a reálná čísla, ...

¹ Vypadá to možná divně, ale je to tak — viz [KR78].

- C neumožňuje přímo práci s řetězci a poli ani přímo neobsahuje nástroje pro vstupy a výstupy. Tyto všechny akce je nutné provádět pomocí volání funkcí, což přináší určité výhody např.:
 - jednoduchost jazyka,
 - jeho nezávislost na počítači.
- Z výše uvedených výhod vyplývá:
 - snadné vytvoření překladače pro konkrétní počítač a konkrétní operační systém (a přeneseně tedy i velké rozšíření jazyka C),
 - velká efektivita kódu — program v C se téměř vyrovná programu v jazyce assembleru.

Vývoj jazyka C:

- Prvním standardem jazyka byla verze jeho autorů — Brian W. Kernighan a Denis M. Ritchie — popsaná ve famózní knize *The C Programming Language*. Tato kniha vyšla v roce 1978 a kromě jiného se stala základní učebnicí jazyka C. Popisovaný standard jazyka C se běžně označuje jako K&R.
- Dnešní oficiální standard je tzv. ANSI C z roku 1988, který z K&R vychází. Jeho součástí je i přesná specifikace množiny knihovných funkcí a hlavičkových souborů (.h), které musí každá implementace ANSI C kompilátoru obsahovat. Této normě by měla vyhovovat naprostá většina dnešních překladačů a další text popisuje právě ANSI C.

Nedocenitelnou výhodou ANSI C je, že program napsaný podle tohoto standardu a pouze s využitím standardních knihovných funkcí (v ANSI C specifikovaných) je téměř 100% přenositelný na libovolný počítač pod libovolný operační systém. Pokud je nutná nějaká změna², pak je to změna opravdu minimální.

Dnes je jazyk C velmi populární programovací jazyk. Jednak díky tomu, že je to “mateřský jazyk” UNIXu a pak také díky zdařilé Turbo (Borland³) implementaci na PC.

Programovat v C je móda a tato móda občas způsobuje příliš kritický pohled na jiné jazyky. Opovržlivý pohled na všechny, kteří C nepoužívají, ale není ten nejlepší přístup, protože každý programovací jazyk má “to svoje”.

² Teoreticky by neměla být žádná, ale již Goethe pravil: “Každá teorie je šedivá, jen strom života je věčně zelený.”

³ Pro PC existují ovšem i kvalitní překladače jazyka C od jiných firem.

1.2 Typografické a syntaktické konvence

V knize jsou různými typy písma odlišeny ty části textu, které si odlišení zaslouží. Jedná se o:

- **else** klíčové (rezervované) slovo jazyka C
- **POKUS.C** jméno souboru vytvořeného uživatelem
- **stdio.h** jméno "systémového" souboru
- **proměnná mensi byla ...** jméno proměnné ve vysvětlujícím textu
- **heap** anglický výraz
- **if (a == b)** úsek programu
- **hromada** výraz, který je prvně použit a bude dále vysvětlen
- **if (podmínka)** *podmínka* je příklad obecného syntaktického objektu ve vysvětlení konstrukce jazyka
- **"x"** znak x zmiňovaný v textu
- **odrádkuje** poznámka uprostřed textu

1.3 Styl psaní programů

Zejména v prvních kapitolách se často setkáte s odstavci uvozenými nadpisem: Štábní kultura

Zde jsou uvedeny doporučení, jak psát v C čitelné a přehledné programy. Upozorňujeme, že jsou to skutečně jen doporučení a je jen na vás, zda se jich budete držet. Tedy zde slovo **muset** v nejrůznějších podobách nemusíte brát tak vážně. Vyskytne-li se ale slovo "muset" jinde v textu, popřípadě je-li navíc podtrženo — muset, pak je to míněno opravdu vážně.

Všechny v knize uvedené příklady tyto zásady dodržují. Pouze na několika málo místech byly porušeny, a to jedině vynecháním prázdné řádky nebo nezarovnáváním komentářů. K těmto "prohřeškům" došlo jen z důvodu omezeného místa na stránce knihy.

Z praktických zkušeností je známo, že právě doporučení týkající se úpravy programů vyvolávají největší nechuť. Odpor začínajících programátorů se dá shrnout do věty:

"Jsem přece svobodný člověk

a nikdo mi tedy nebude předepisovat, kde mám dělat mezeru!"

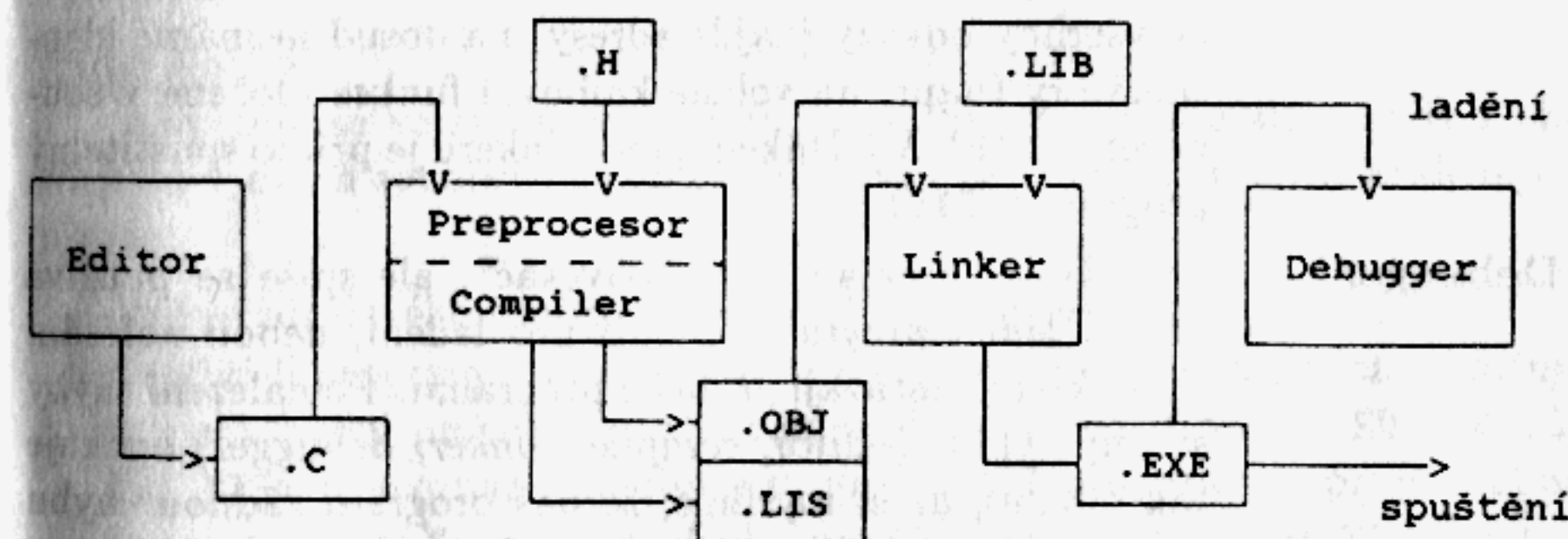
K tomu se dá poznamenat pouze jediné — pokud skutečně chcete psát v jazyce C větší programy, je nutné nějaký pořádek zavést a chcete-li nějaký zavést, pak je zbytečné vymýšlet již vymyšlené.

2 Základní pojmy

2.1 Způsob zpracování programu

Chcete-li jazyk C opravdu využívat, je nutné seznámit se i trochu více s tím, jak se program vlastně zpracovává, od napsání zdrojového textu až po spuštění přeloženého a sestaveného programu¹.

Základní zpracování programu v C probíhá několika fázemi schematicky vyjádřenými následujícím obrázkem:



Jednotlivé programy, kterými je zdrojový soubor zpracováván, mají tento význam:

Editor

Pomocí něj se vytváří a opravuje zdrojový (.C) soubor.

Preprocesor

Je to součást překladače, která předzpracovává (upravuje) zdrojový soubor tak, aby měl překladač snadnější práci. Např. vynechává komentáře, zajišťuje správné vložení hlavičkových (.H) souborů, rozvoj maker, atd. Výsledkem jeho

¹ Tiše předpokládáme, že vám pojmy "překlad a sestavení programu" nejsou úplně cizí. Pokud víte zatím pouze to, že po napsání programu se "mačká" <Ctrl> <F9>, a nejsou vám ani trochu jasné následující řádky, pak si s tím zatím nelamte hlavu. Budeme je potřebovat až v kap. 8.4.0.

práce je opět textový soubor. Ten si však můžete prohlédnout pouze tehdy, když umíte spustit preprocessor samostatně (tzn. bez kompilátoru). Jinak preprocesor předává výsledky své práce přímo "svému nadřízenému" kompilátoru.

Compiler

Nazývaný též *překladač* nebo počestně *kompilátor*, provádí překlad zdrojového souboru (zpracovaného již preprocesorem) do relativního (objektového) kódu počítače — vzniká .OBJ soubor. Relativní kód² je téměř hotový program. Slovo relativní znamená, že adresy proměnných nebo funkcí ještě nejsou vůbec známy (např. proto, že jsou uloženy v knihovně) a jsou tedy v .OBJ souboru zapsány relativně. Vedlejší produkt překladače je tzv. protokol o překladu³ (.LIS), ve kterém je uložena informace o chybách nalezených překladačem.

Linker

Neboli *sestavovací program* přidělí relativnímu kódu (respektive jeho relativním adresám) absolutní adresy a provede všechny odkazy (najde adresy) na dosud neznámé identifikátory (např. na volané knihovní funkce uložené v souborech .LIB). Výsledkem práce linkeru je přímo spustitelný program (.EXE).

Debugger

Jeho český překlad je "odvšivovač", ale spíše se používá pojem *ladící program*. Slouží pro ladění, neboli nalézání chyb, které nastávají při běhu programu. Po nalezení chyby se celý cyklus (*editor, compiler, linker, debugger*) opakuje tak dlouho, až si myslíme, že náš program žádnou chybu neobsahuje.

Poznámka:

- Tento cyklus probíhá při použití jakéhokoliv překladače. Nenechte se mýlit tím, že se s ním nesetkáváte např. u překladače fy Borland. I tam je nutné nejdříve program přeložit, pak sestavit a nakonec spustit nebo ladit. Zde se sice všechny tyto činnosti dělají téměř automaticky a najednou, ale dělají se⁴.

² Který se oficiálně nazývá jazyk relativních adres.

³ Který se většinou vytváří pouze na speciální přání uživatele.

⁴ Překladače fy Borland ale tyto činnosti umožňují také dělat postupně, což je mnohdy velmi výhodné.

2.2 Základní pojmy v jazyce C

Dosud byly popisovány základní pojmy z hlediska zpracování celého programu. Od této chvíle nás zajímá pouze to, co zpracovává překladač s preprocesorem, čili zdrojový program v jazyce C.

2.2.1 Zdrojové a hlavičkové soubory

Plný význam těchto pojmů bude vysvětlen na str. 132. Pro nás je zatím důležité, že zdrojový soubor .C, ve kterém je náš program, je pro jeho správnou funkci většinou nutno trochu doplnit *vložením souboru*.

Nejčastěji vkládáme (*include*) tzv. hlavičkové (*header-ové*) soubory .H. Používají se proto, že program většinou volá knihovní funkce (I/O, ...), jejichž správné využití umožní právě vložení příslušného .H souboru pomocí příkazu `#include`, například: `#include <stdio.h>`

Štábní kultura:

Abychom se ve svých souborech (velké programy jsou složeny až z desítek zdrojových souborů) snadno orientovali, je dobré dodržovat následující zásady:

- Jméno zdrojového souboru by mělo být jednoznačné.
- Sestává-li program z více souborů, pak první 3 písmena jména označují projekt (velký program), další pak modul, např. `EDI_KLA.C`, `EDI_OBR.C`, `EDI_TISK.C` — označují soubory z projektu `EDItoru`, přičemž první soubor je pro ovládání `KL`avesnice, druhý pro `OBR`azovku a třetí pro `TISK`árnu.
- Vlastní program začíná vždy hlavičkou podobnou této :

```
/*
 * ASCII.C          v.1.0      jméno souboru a verze
 *
 * Vypis ASCII tabulky      stručný popis programu
 *
 * P.Herout 9.1991      autor a datum vytvoření
 */
```

2.2.2 Bílé znaky

Je to překlad anglického výrazu *white spaces* a zahrnují znaky, které jsou sice velmi důležité, ale nejsou na obrazovce vidět. Jsou to tzv. oddělovací

znaky jako mezera, tabulátor, nový řádek, nová stránka, návrat na začátek řádku, atd. — viz též str. 29.

2.2.3 ASCII tabulka

Nazývaná někdy znaková sada, popisuje kódy (v Pascalu se používá pojem *ordinální čísla*), které jsou přiděleny jednotlivým znakům. ASCII tabulka má rozsah od 0 do 255, ale běžně se pracuje jen s její dolní polovinou, tedy se znaky od 0 do 127. Horní polovina ASCII tabulky je vyhrazena pro znaky národních abeced (jako např. naše písmena s háčky a čárkami) a pro některé speciální znaky (např. rámečky, matematické symboly, ...). V dalším textu budeme využívat pouze dolní polovinu ASCII tabulky, přičemž je nutné podotknout, že s horní polovinou se pracuje naprosto stejně.

Poznámka:

- Pokud jste měli někdy problémy s různou interpretací ordinálních čísel znaků (EBCDIC versus ASCII), pak v C by jste si měli tuto starost ušetřit. V jazyce C totiž nejsou většinou problémy, protože pro UNIX je doporučena ASCII a tím je to dáno i pro C.

Dolní část ASCII tabulky obsahuje:

řídící znaky	0 (00h)	31 (1Fh) <i>neviditelné</i>
mezera	32 (20h) ' '	
pomocné znaky	33 (21h) '!'	47 (2Fh) '/'
číslice	48 (30h) '0'	57 (39h) '9'
pomocné znaky	58 (3Ah) ':'	64 (40h) '@'
velká písmena	65 (41h) 'A'	90 (5Ah) 'Z'
pomocné znaky	91 (5Bh) '['	96 (60h) '`'
malá písmena	97 (61h) 'a'	122 (7Ah) 'z'
pomocné znaky	123 (7Bh) '{'	126 (7Eh) '~'

Neviditelné znaky jsou např.:

7 Bell 8 BackSpace 9 Tab 10 LineFeed 13 CarriageReturn ...

2.2.4 Identifikátory

Jazyk C je *case sensitive* jazyk, tj. rozlišuje malá a velká písmena. V praxi to znamená, že: `prom` `Prom` `PROM` jsou tři různé identifikátory!

Klíčová slova jazyka C (např. `if`, `while`, `register`, ...) musí být psána malými písmeny. Jsou-li zapsána velkými nebo kombinací malých a velkých písmen, berou se jako identifikátory.

C dovoluje u identifikátorů používat znak podtržítko "_", které se ale nepoužívá zcela libovolně, např.:

- `_prom` — nepoužívat, znamená to systémový identifikátor
- `prom_x` — používat často, zpřehledňuje text
- `prom_` — nepoužívat, protože se na konci často přehlédne

Délka identifikátoru není omezena, ale ANSI C rozeznává obecně pouze prvních 31 znaků⁵, čili případné další znaky (32-hý a další) jsou pro něho nevýznamné.

Štábní kultura:

- Jména objektů (proměnných, funkcí) psát zásadně malými písmeny s využitím podtržítek.
- Snažit se o rozlišitelnost identifikátorů podle prvních 8-mi znaků (u externích identifikátorů podle prvních 6-ti — viz též str. 130).
- Nepoužívat podobné identifikátory, např.: `systst` a `sysstst`
- V žádném případě nepoužívat dva stejné identifikátory rozlišené jen typem písma, např.: `PROM` a `prom`
- Jména objektů musí být významná, např.: `plat` a ne `p1`
- Jména se tvoří podle jednotného racionálního schématu, např.:

`view_file` `view_menu` `view_window`

- Běžně používané významové identifikátory:

i	j	k	— indexy, parametry cyklů
c	ch		— znaky
m	n		— čítače
f	r		— reálná čísla
p-			— začátek identifikátoru pointeru, např. <code>p_plat</code>
s			— řetězec ⁶

2.2.5 Komentáře

Ačkoliv jsou velmi často opomíjeny, představují komentáře důležitou část zdrojového programu, protože zpřehledňují, někdy na první pohled dost nepochopitelný, program.

⁵ Používáte-li dlouhé identifikátory, je vhodné si ověřit aktuální stav u vašeho kompilátoru — někdy je to jen 8 znaků!

⁶ Neznačená to, že každý řetězec se musí nutně jmenovat `s`, ale to, že když se v programu vyskytne identifikátor `s` nemůže to být např. identifikátor pro reálné číslo.

Komentář slouží k tomu, aby se ve vašem programu vyznal někdo cizí nebo i vy sami, když se k tomuto programu vrátíte za nějakou dobu a už nemáte v čerstvé paměti všechny "figle", které jste předtím použili.

Doporučujeme, abyste své programy průběžně komentovali již při vytváření zdrojového souboru a ne až po odladění ("Až na to někdy zbyde čas.").

Komentář je uzavřen mezi dvojicí "závorek" `/*` a `*/` a může se objevit všude tam, kde je bílý znak.

```
/* toto je komentar */
```

V komentáři se mohou objevit i znaky z horní poloviny ASCII tabulky, tedy i písmena s háčky a čárkami. Prakticky se tato možnost ale příliš nepoužívá.

```
/* toto je český komentář */
```

ANSI C nedovoluje vhnížděné komentáře (*nested comments*), např.:

```
/* toto je komentář /* toto je vhnížděný komentář */ */
```

Štábní kultura:

- Uvádějte komentář před každou logicky ucelenou částí kódu.
- U kratších funkcí stačí popis její funkce před definicí funkce.
- Komentář má obsahovat pouze užitečnou informaci!

Příklady:

Údaj o sloupci znamená číslo sloupce na obrazovce, ve kterém se objeví první z dvojice znaků `/*`. Umožňuje to vytvářet komentáře "oku lahodící".

1.sloupec

```
/*
 * Výrazný
 * víceřádkový
 * komentář
 */
```

n.sloupec

```
/* delší popis následující logické části kódu,
   který se nevejde na jednu řádku */
if ((fr = fopen(str, "r")) == NULL) {
```

1.sloupec

```
/*
 * jednořádkový popis funkce
 */
```

33.sloupec

```
x = 3 * a + b;
/* popis příkazu */
```

3 První začátky s C

Až dosud byly popisovány obecné věci a o vlastním programování v C jsme se ještě mnoho nedozvěděli. Od této chvíle se budeme snažit o nápravu.

3.1 Jednoduché datové typy a přiřazení

C poskytuje podobné datové typy jako Pascal:

Pascal	C
INTEGER	int long int též long short int též short
CHAR	char
REAL	float double long double

Poznámky:

- Verze K&R jazyka C neměla typ `long double` a typ `signed`.
- Typy `int`, `long int`, `short int` a `char` mohou být buď typu `signed` nebo `unsigned`.
- Typ `unsigned int` se často zkracuje jen na `unsigned`.
- Pro typy `int`, `long int`, `short int` je implicitní typ `signed`, pro typ `char` to záleží na implementaci.
- Rozdíl mezi `signed` (znaménkový) a `unsigned` (neznaménkový) je v rozsahu čísla. Proměnné typu `unsigned` mají rozsah od 0 do $2^n - 1$, kde n je počet bitů proměnné. To tedy znamená, že `unsigned` proměnná nemůže zobrazit záporné číslo.

Rozsah `signed` proměnných je od -2^{n-1} do $+2^{n-1} - 1$. Tedy možnost zobrazení záporných čísel je u typu `signed` zaplácena polovičním rozsahem oproti typu `unsigned`.

Například pro typ `char`, který je vždy 1 Byte (8 bitů) dlouhý, to znamená:

<code>unsigned char</code>	0 až 255
<code>signed char</code>	-128 až +127

- C neposkytuje přímo typ `Boolean`. Booleovské hodnoty jsou reprezentovány pomocí celočíselných (`int`) hodnot, kde:

nulová hodnota (0)	<code>FALSE</code>
nenulová hodnota (nejčastěji 1)	<code>TRUE</code>
- Typ `double` má přesnost asi na 20 desetinných míst.

- C zaručuje, že jednotlivé typy alokují tuto paměť¹:

```
sizeof(char)      = 1 Byte
sizeof(short int)  <= sizeof(int)      <= sizeof(long int)
sizeof(unsigned int) = sizeof(signed int)
sizeof(float)      <= sizeof(double)   <= sizeof(long double)
```

3.1.1 Definice proměnných

Pod pojmem *definice* se míní příkaz, který přidělí proměnné určitého typu jméno a paměť.

Naopak *deklarace* je příkaz, který pouze udává typ proměnné a její jméno. Deklarace nepřiděluje žádnou paměť! Smysl deklarací a jejich použití bude vysvětlen v kap. 9.2.5.

Pozor:

V některé literatuře jsou významy slov deklarace a definice právě opačné. Při nejasnostech je tedy třeba zjistit na příkladech, co má autor na mysli.

V C jsou definice v obráceném pořadí než u Pascalu:

Pascal	C
VAR i : INTEGER;	int i;
c, ch : CHAR;	char c, ch;
f, g : REAL;	float f, g;

Poznámka:

- Definice proměnných se mohou vyskytnout buď vně (globální proměnná) nebo uvnitř funkce (lokální proměnná)², např.:

```
int i;          /* globalni promenna */
main()
{
    int j;       /* lokalni promenna */
}
```

Štábní kultura:

- Nepoužívejte typ `unsigned int` jen proto, aby se zdvojnásobila velikost čísla — v tomto případě je lepší použít `long`.
- Každá proměnná by měla být definována na samostatné řádce a okomentována (výjimku tvoří pomocné proměnné), např.:

```
int c_plat;      /* celkový plat */
```

- Definice je odsazena od začátku řádky podle úrovně zahníždění bloku, kde začátkem bloku je znak "{".
- Globální proměnné se definují od začátku řádky (viz předchozí příklad s globální a lokální proměnnou).

3.1.2 Přiřazení

Přiřazovací příkaz je nejčastějším příkazem ve většině programovacích jazyků. V C je ale nutno dávat větší pozor, protože se velmi liší různé jemné odstíny tohoto slova.

Často se pracuje s pojmem *l-hodnota* (*l-value*). L-hodnota představuje adresu, tedy např. proměnná (`x`) je l-hodnotou, ale konstanta (`123`) nebo výraz (`x + 3`) l-hodnotou nejsou.

Stručně lze říci, že l-hodnota je to, co může být na levé straně přiřazení.

Pozor na následující terminologii:

česky	anglicky	symbolicky	prakticky
výraz	expression	výraz	<code>i * 2 + 3</code>
přiřazení	assignment	<i>l-hodnota</i> = výraz	<code>j = i * 2 + 3</code>
příkaz	statement	<i>l-hodnota</i> = výraz;	<code>j = i * 2 + 3;</code>

Tedy:

- výraz má vždy hodnotu (číslo),
- přiřazení je výraz a jeho hodnotou je hodnota na pravé straně,
- přiřazení se stává příkazem, je-li ukončeno středníkem.

Příklad:

Různé typy přiřazovacích příkazů:

Pascal	C
<code>j := 5;</code>	<code>j = 5;</code>
<code>d := 'z';</code>	<code>d = 'z';</code>
<code>f := f + 3.14 * i;</code>	<code>f = f + 3.14 * i;</code>

Pozor:

Následující dvě maličkosti dělají pascalským programátorům potíže:

- 1) C má přiřazení pouze pomocí `=` a ne pomocí `:=`
- 2) Porovnání v C je `==` ne pouze `=`

Protože přiřazení je výraz, je možné několikanásobné přiřazení:

```
k = j = i = 2;
```

které se vyhodnocuje zprava doleva tedy: `k = (j = (i = 2));`

¹ Operátor `sizeof` určí velikost typu v Bytech — viz též str. 163.

² Podrobně viz str. 118.



3.2 Hlavní program

Hlavní program v C se jmenuje vždy `main` a musí být v programu vždy uveden — je to první funkce, která je volána po spuštění programu.

Pascal	C
PROGRAM POKUS(INPUT, OUTPUT);	main() <i>bez středníku!</i>

Sestává-li program z více modulů³, `main` smí být pouze v jednom modulu a v tomto modulu také pouze jednou.

Funkce `main` je normální funkce v C, která se odlišuje od ostatních pouze tím, že je vyvolána na začátku programu jako první.

Pascal	C
PROGRAM POKUS(INPUT, OUTPUT);	main() /* bez středníku! */
VAR	{
i, j : INTEGER;	int i, j;
BEGIN	i = 5;
i := 5;	j = -1;
j := -1;	j = j + 2 * i;
j := j + 2 * i;	}
END.	

Poznámky:

- Pascalské **BEGIN** a **END** je v C nahrazeno znaky "{" a "}".
- Závorky "{" }" neuzavírají pouze složený příkaz, ale i blok.
- Bezprostředně za každou "{" mohou být definice.
- Blok⁴ je seznam definicí následovaný seznamem příkazů a složený příkaz je pouze seznam příkazů, např.:

```

{          /* toto je blok, protože obsahuje */
  int i;   /* definici promenne i          */

  i = 5;
  j = 6;
}

```

³ Podrobně viz str. 99.

⁴ Zatím není nutné si s ním lámat hlavu. V kap. 9.3.4 bude vysvětleno proč se používá.

```

{          /* toto je slozeny prikaz */
  i = 5;   /* protože neobsahuje */
  j = 6;   /* zadne definice promennych */
}

```

- Narozdíl od Pascalu, umožňuje C inicializaci proměnných přímo v definici, takže předchozí příklad bude také správně, když:

```
main()
{
```

```
  int i = 5,
     j = -1;
```

```
  j = j + 2 * i;
```

```
}
```

definice

příkaz

blok

Státní kultura:

- Každá inicializovaná proměnná se píše do jedné řádky.
- Mezi definicemi a příkazy je prázdná řádka (viz předchozí příklad).

3.3 Konstanty

3.3.1 Celočíselné konstanty

C umožňuje použít tři typů celočíselných konstant:

- (desítkové) — posloupnost číslic, z nichž první nesmí být 0 (nula)
- osmičkové (oktalové) — číslice 0 (nula) následovaná posloupností osmičkových číslic (0 – 7)
- šestnáctkové (hexadecimální) — číslice 0 (nula) následovaná znakem x (nebo X) a posloupností hexadecimálních číslic (0 – 9, a – f, A – F)

Příklady:

Příklady různých zápisů celočíselných konstant:

- 1) desítkové 15, 0, 1
- 2) oktalové 065, 015, 0, 01
- 3) hexadecimální 0x12, 0X3A, 0XCD, 0xcd, 0Xcd, 0x15, 0x0, 0x1

Typ konstanty je určen implicitně její velikostí nebo explicitně použitím přípony L (nebo l⁵) jako long, např. 12345678L

⁵ To se ale nedoporučuje používat, protože se příliš podobá číslici 1.

Druhá přípona, která explicitně určuje typ celočíselné konstanty je **u** (nebo **U**), jako **unsigned**, např.: **129u**, **12345LU**

Záporné konstanty jsou podle zvyklostí uvozeny znakem **-** (mínus), např.: **-56**

Poznámka:

- V K&R verzi jazyka C nebylo unární **+** (plus), např.: **+56** nebylo možno napsat jako konstantu. V ANSI C toto omezení již není.

3.3.2 Reálné konstanty

Tvoří se podle běžných zvyklostí, mohou začínat a končit desetinou tečkou a jsou implicitně typu **double**, např.:

15., **56.8**, **.84**, **3.14**, **5e6**, **7E23**.

Reálná konstanta typu **float** se definuje pomocí přípony **F** (nebo **f**), např. **3.14f**, a typu **long double** pomocí **L** (nebo **l**), např. **12e3L**.

3.3.3 Znakové konstanty

Jsou, stejně jako v Pascalu, uzavřeny mezi apostrofy, např.: **'a'**, **'*'**, **'4'**

Hodnota znakových konstant (ordinální číslo) je odvozena z odpovídající kódové tabulky — nejčastěji ASCII. Velikost znakové konstanty je kupodivu **int** a ne **char**⁶!

Potřebujeme-li znakovou konstantu z neviditelného znaku, pak použijeme zápis ve tvaru: **'\ddd'**

kde **ddd** je kód znaku složený ze tří oktalových číslic, např.: **'\012'**, **'\007'**. Počáteční nuly nejsou nutné, ale z konvenčních důvodů se většinou uvádějí.

Druhou možností, jak zapsat tyto konstanty, je použití zápisu **'\0xHH'** (nebo **'\0XHH'**), např.: **'\0x0A'**, **'\0XD'**, **'\0X1f'**

Znak **"\"** (zpětné lomítko – *backslash*) se často nazývá *escape character*, protože je používán pro změnu běžného významu. Např. **'012'** je nedovolená konstrukce (tříznaková konstanta), protože znaková konstanta může mít jen jeden znak.

Konstanty uvozené znakem **"\"** se také nazývají *escape* sekvence.

Některé často používané *escape* sekvence mají kromě numerických kódů i znakový ekvivalent (představují používané řídicí znaky):

⁶ Důvod viz str. 71.

sekvence	hodnota	význam
\n	0x0A	nová řádka (<i>newline, linefeed</i> – LF)
\r	0x0D	návrat na začátek řádky (<i>carriage return</i> – CR)
\f	0x0C	nová stránka (<i>formfeed</i> – FF)
\t	0x09	tabulátor (<i>tab</i> – HT)
\b	0x08	posun doleva (<i>backspace</i> – BS)
\a	0x07	písknutí (<i>alert</i> – BELL)
\\	0x5C	zpětné lomítko (<i>backslash</i>)
\'	0x2C	apostrof (<i>single quote</i>)
\0	0x00	nulový znak (<i>null character</i> – NUL)

Pozor:

NUL není **NULL**, čili nulový pointer — viz str. 151.

Poznámky:

- V řetězcových konstantách se používá pro zobrazení znaku uvozovky (*double quote*) tato *escape* sekvence: **"\"**
- Uvozovky jako znaková konstanta vypadají ale: **'\"'**
- Konstantu **\a** nelze někdy použít, např. v UNIXu nefunguje.

3.3.4 Řetězcové konstanty (literály)

Jsou uzavřeny, narozdíl od Pascalu, mezi uvozovky, např.:

"Tohle je ukázková řetězcová konstanta"

V řetězcových konstantách, podobně jako v komentářích, je možné použít češtinu, např.:

"Tohle je ukázková řetězcová konstanta"

ANSI C umožňuje automatické zřetězování dlouhých literálů oddělených mezerami, tabulátory nebo novými řádkami. Tato maličkost podstatně zpřehledňuje zápis např. při příkazech tisku. Následující tři literály jsou ekvivalentní:

- 1) **"Takhle vypadá velmi dlouhý řetězec"**
- 2) **"Takhle vypadá " "velmi dlouhý řetězec"**
- 3) **"Takhle vypadá " "velmi " "dlouhý řetězec"**

3.4 Aritmetické výrazy

V aritmetických výrazech je nutné si připomenout, že výraz ukončený středníkem se stává příkazem, např.:

i = 2 výraz s přiřazením
i = 2; příkaz

Pouhý středník představuje často používaný prázdný příkaz (null statement), který se používá např. v cyklu **while** nebo **for** (viz např. str. 51).

3.4.1 Unární operátory

Jsou to unární mínus **-** a unární plus **+**. Oba se používají v běžném významu.

Poznámka:

- Pomocí unárního plus se dá také ovlivnit pořadí vyhodnocovaných výrazů — podrobně viz [HRŠ92].

3.4.2 Binární operátory

Z větší části mají stejný význam jako v Pascalu:

	Pascal	C
sčítání	+	+
odečítání	-	-
násobení	*	*
reálné dělení	/	/
celočíslné dělení	DIV	/
dělení modulo	MOD	%

Poznámka:

- To, zda dělení bude celočíselné nebo reálné, závisí na typu operandů⁷:

int / int	— celočíselné
int / float	— reálné
float / int	— reálné
float / float	— reálné

Příklad:

```
int i = 5, j = 13;
j = j / 4;      — celočíselné dělení, j bude 3
j = i % 3;      — dělení modulo, j bude 2
```

3.4.3 Speciální unární operátory

Tyto operátory nemají v Pascalu obdoby. Jsou to:

inkrement **++**
dekrement **--**

Oba operátory se dají použít jako předpony (prefix), tak i jako přípony (suffix) s tímto odlišným významem⁸:

⁷ Pro double a long double platí totéž co pro float.

⁸ Pro operátor **--** je význam analogický.

- 1) **++výraz**
 - inkrementování před použitím,
 - výraz je nejprve zvětšen o jedničku a pak je tato nová hodnota vrácena jako hodnota výrazu.
- 2) **výraz++**
 - inkrementování po použití,
 - je vrácena původní hodnota výrazu a pak je výraz zvětšen o jedničku.

Pozor:

Výraz musí být l-hodnota (tedy proměnná). Časté chybné použití je:
45++ nebo **--(i + j)**

Příklad:

Různá použití operátorů **++** a **--**

```
int i = 5, j = 1, k;
```

i++;	— i bude 6
j = ++i;	— j bude 7, i bude 7
j = i++;	— j bude 7, i bude 8
k = --j + 2;	— k bude 8, j bude 6, i bude 8

3.4.4 Přiřazovací operátory

V Pascalu je pouze jeden operátor přiřazení **:=**. V C, jak již víme, je jeho ekvivalentem operátor **=**, ale C dává navíc k dispozici ještě celou škálu rozšířených přiřazovacích operátorů.

Místo přiřazení:

l-hodnota = l-hodnota operátor výraz

se velmi často používá zkrácený zápis:

l-hodnota operátor= výraz

Dají se použít následující přiřazení:

l-hodnota += výraz	l-hodnota = l-hodnota + výraz
l-hodnota -= výraz	l-hodnota = l-hodnota - výraz
l-hodnota *= výraz	l-hodnota = l-hodnota * výraz
l-hodnota /= výraz	l-hodnota = l-hodnota / výraz
l-hodnota %= výraz	l-hodnota = l-hodnota % výraz
l-hodnota >>= výraz	l-hodnota = l-hodnota >> výraz
l-hodnota <<= výraz	l-hodnota = l-hodnota << výraz
l-hodnota &= výraz	l-hodnota = l-hodnota & výraz
l-hodnota ^= výraz	l-hodnota = l-hodnota ^ výraz
l-hodnota = výraz	l-hodnota = l-hodnota výraz

Poznámka:

- Operátory `>>` a `<<` budou vysvětleny na str. 257 a operátory `&`, `^` a `|` na str. 256.

Štábní kultura:

Nepoužívejte mezeru pro oddělení operátoru a rovnítka, např.:

```
j + = 5;
```

Některým kompilátorům to dělá potíže.

Příklad:

Různá použití přiřazovacích operátorů.

```
int i = 4, j = 3;
```

```
j += i;           j bude 7
```

```
j /= --i;         j bude 2, i bude 3
```

```
j *= i - 2;       j = j * (i - 2) = 2
```

```
ne j = j * i - 2 = 4
```

4 Terminálový vstup a výstup

Jazyk C, narozdíl od Pascalu, nedefinuje žádnou I/O (vstupně/výstupní – Input/Output) operaci jako část jazyka. Nezbytné vstupy a výstupy jsou řešeny tak, že standardní knihovna obsahuje několik funkcí, které I/O zajišťují.

Důvod je jednoduchý — nejvíce strojově závislé akce jsou právě I/O a tímto se tedy důsledně oddělují strojově závislé a strojově nezávislé části jazyka. Tato skutečnost je pak významným přínosem při vytváření kompilátoru pro jiný počítač.

4.1 Hlavičkový soubor `stdio.h`

Aby bylo možno správně používat všechny funkce pro vstup a výstup, je nutné na začátku programu připojit "popis" těchto funkcí. Ten se nachází v hlavičkovém (header) souboru `stdio.h` a do našeho programu se připojí pomocí příkazu:

```
#include <stdio.h>     zde není středník !!!
```

Od tohoto okamžiku je pak možné používat dále popsané funkce.

4.2 Vstup a výstup znaku

Výstup jednoho znaku zajišťuje funkce `putchar()` a vstup jednoho znaku funkce `getchar()`.

Obě funkce pracují kupodivu¹ s proměnnými typu `int` a ne `char`.

Poznámky:

- Při volání funkce `getchar()` je nutné uvést závorky, i když nemá žádný parametr — viz dále příklady.

¹ Zatím nad tímto zjištěním jenom zakruťte hlavou, skutečný důvod bude vysvětlen na str. 71.

- Je nutné si uvědomit, že po volání `getchar()` můžeme psát znaky z klávesnice tak dlouho, dokud nestiskneme <Enter>. `getchar()` pak přečte první z těchto zadaných znaků a ostatní ignoruje — viz též str. 76.

Příklad:

Program přečte znak z klávesnice, vytiskne ho a odřádkuje.

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

    c = getchar();
    putchar(c);
    putchar('\n');
}
```

4.3 Formátovaný vstup a výstup

Pro vstupy a výstupy nepoužíváme většinou znakově orientované funkce, ale spíše funkce, které při vstupu čísla načtou celé číslo jako řetězec a převedou ho do číselné podoby automaticky (v Pascalu např. funkce `READ`) nebo naopak, při výstupu samy konvertují hodnotu číselné proměnné na odpovídající posloupnost číslic (v Pascalu např. `WRITE`). Tyto funkce existují samozřejmě i v jazyce C a velmi často se používají. Jsou to:

pro vstup: `scanf()`
pro výstup: `printf()`

Základní použití :

- 1) Příkaz: `scanf("%d", &i);`
přečte z klávesnice celé číslo a uloží ho do proměnné `i`
 - `"%d"` určuje formát čtení (zde dekadický celočíselný)
 - `&` před `i` je nezbytně nutný² a vynechání `&` představuje častou chybu
- 2) Příkaz: `printf("%d", i);`
vytiskne na obrazovku hodnotu proměnné `i`
 - `"%d"` určuje formát výpisu (zde dekadický celočíselný)
 - před `i` není³ `&`, což je rozdíl od funkce `scanf()` a často se to plete.

² Ve skutečnosti `&` představuje adresu a tedy říká překladači, kde leží proměnná, do níž se uloží přečtená hodnota, ale to se dozvíme až na str. 148 — v C jsou totiž skutečné parametry předávány pouze hodnotou.

³ Je to totiž hodnota a ne adresa.

Příklad:

Program přečte z klávesnice dvě čísla, vytiskne je v obráceném pořadí a pak vytiskne jejich součet.

```
#include <stdio.h>
main()
{
    int i, j;

    scanf("%d", &i);
    scanf("%d", &j);

    printf("%d%d", j, i);
    printf("%d je soucet", i + j);
}
```

např. pro vstup: 4 7 bude výstup⁴: 7411 je soucet

4.3.1 Řídící řetězec formátu

Protože funkce `printf()` a `scanf()` mají proměnný počet parametrů (můžeme např. tisknout najednou hodnoty jedné, dvou nebo více proměnných nebo výrazů) je nutné nějakým způsobem těmto funkcím sdělit, kolik vlastně mají v konkrétním případě zpracovávat parametrů. K tomu slouží tzv. *řídící řetězec formátu*, což je řetězec uzavřený v uvozovkách⁵ — je to vždy první parametr těchto funkcí. Tento řetězec určuje formát čtení nebo zápisu a počet hodnot (proměnných nebo výrazů), s nimiž se bude pracovat.

K ohraničení řídícího řetězce se používají uvozovky a ne apostrofy, protože v uvozovkách se zadává řetězec a v apostrofech pouze znak!

Řídící řetězec formátu obsahuje:

- | | |
|------------------------------|---|
| <i>formátové specifikace</i> | – začínají znakem <code>"%"</code> |
| | – určují formát vstupu nebo výstupu |
| <i>znakové posloupnosti</i> | – nezačínají znakem <code>"%"</code> |
| | – vypíší se tak, jak jsou zapsány |
| | – je možné použít češtinu |
| | – používají se pouze ⁶ pro <code>printf()</code> |

⁴ Není to chyba, jen jsme neoddelili oba výsledky.

⁵ Tedy nám už dobře známá řetězcová konstanta — viz str. 29.

⁶ Pro `scanf()` fungují také, ale trochu složitěji, takže se obvykle nepoužívají. Přesné vysvětlení viz str. 198.

Příklady:

- 1) `printf("Soucet je %d", i + j);`
vypíše: Soucet je 11
- 2) `printf("Pracovali na 100%%");`
vypíše: Pracovali na 100%
neboť pro výpis znaku "%" je nutné tento znak zdvojit
- 3) `printf("Soucet je %d\tSoucin je %d\n", i + j, i * j);`
vypíše: Soucet je 11 Soucin je 28 a odřádkuje
- 4) `printf("\007Chyba, pokus o deleni nulou.\n");`
pískne⁷ a vypíše: Chyba, pokus o deleni nulou. a odřádkuje
- 5) `printf("Toto je \"backslash\": '\\'\n");`
vypíše: Toto je "backslash" : '\\' a odřádkuje
Proto se při tisku používají často pouze apostrofy místo uvozovek, neboť s apostrofy nejsou žádné problémy.
- 6) `printf("Toto je 'backslash' : '\\'\n");`
vypíše: Toto je 'backslash' : '\\' a odřádkuje

Počet argumentů `printf()` a `scanf()` může být větší než dva (tj. lze tisknout nebo načítat více než jednu proměnnou), ale je vždy nutné dodržet stejný počet parametrů (proměnných nebo výrazů) jako formátových specifikací⁸. Při neshodě kompilátor nehlásí ani chybu ani varovné hlášení, ale výsledný program pracuje chybně.

Za znakem "%" může být (a bývá) uvedena celá řada formátových specifikací. Nyní budou vysvětleny pouze nejčastěji používané. Podrobný přehled viz str. 209.

Některé formátové specifikace řídicího řetězce formátu uváděné za znakem "%" použitelné jak pro `scanf()` tak i pro `printf()`:

- c – znak⁹
- d – desítkové číslo typu **signed int**
- ld – desítkové číslo typu **signed long**
- u – desítkové číslo typu **unsigned int**
- lu – desítkové číslo typu **unsigned long**
- f – float (pro `printf()` také double)
- Lf – long double (Pozor: L musí být velké!)

⁷ Důvod viz str. 29.

⁸ Jednoduše řečeno — kolik je v řídicím řetězci znaků "%", tolik musí být dalších parametrů.

⁹ Jeden znak je lepší číst `zn = getchar()` než `scanf("%c", &zn);`

- lf – double (Pozor: někdy nelze použít pro `printf()`)
- x – hexadecimální číslo malými písmeny, např. 1a2c
- X – hexadecimální číslo velkými písmeny, např. 1A2C
- o – osmičkové číslo
- s – řetězec

Příklady:

- 1) `printf("Znak '%c' ma ASCII kod %d (%XH)\n", c, c, c);`
vypíše: Znak 'A' ma ASCII kod 65 (41H)
- 2) `printf("Znak '%c' ma ASCII kod %d (%XH)\n", '*', '*', '*');`
vypíše: Znak '*' ma ASCII kod 42 (2AH)
- 3) `printf("Je presne %2d:%2d\n", hodiny, minuty);`
vypíše např.: Je presne 1:12
nebo např.: Je presne 13: 3
protože počet cifer, který se bude tisknout, se dá přímo určit
- 4) `printf("Za pivo jsme utratili: %6.2f Kcs.\n", pocet * cena.piva);`
vypíše: Za pivo jsme utratili: 13.60 Kcs.
protože 6.2 znamená, že reálné číslo bude vytištěno na 6 znaků a z nich budou dva za desetinou tečkou a jeden je desetinná tečka
- 5) `printf("Kolik stalo %s pivo?\n", "jedno");`
vypíše: Kolik stalo jedno pivo?

Štábní kultura:

Výpisy je vhodné tvořit z písmen malé abecedy. Velká písmena jsou méně přehledná a připomínají zlatou éru děrných štítků, např.:

Pocitac a POCITAC

Časté chyby:

Zde jsou uvedeny nejčastější chyby, které se týkají předchozích dvou kapitol.

- `main();` za definicí funkce se nedělá středník
- `printf("%d", i, j);` mnoho argumentů
- `printf("%d%d", i);` málo argumentů
- `scanf("%d", i);` chybí znak & tedy: `scanf("%d", &i);`
- `scanf("%d", &c);` formát pro char je %c `scanf("%c", &c);`

Co je dobré si uvědomit:

- Všechna klíčová slova musí být malými písmeny.
- Dělení (/) je operace závislá na typu operandů — pro celá čísla je to celočíselné dělení, jinak je to reálné dělení.
- Přiřazení je výraz a příkazem se stává až po ukončení středníkem.
- Počet výstupních výrazů v `printf()` nebo vstupních v `scanf()` musí přesně odpovídat počtu formátových specifikací.
- U proměnných ve `scanf()` je `&`.

Cvičení:

- 1) Napište program, který vypíše přesně následující text:
James Bond \ "Agent 007" \ # 150% zaruka # /
Spol. s rucenim neomezenym
- 2) Napište program, který přečte znak a vypíše znak s ASCII hodnotou o jednu vyšší, např.:
vstup : A
výstup : B (ASCII 66)
- 3) Napište program, který přečte celé dekadické číslo (v rozsahu 0 až 255) a vypíše jeho hexadecimální hodnotu dvouznakově, např.:
vstup : 127
výstup : 7Fh
- 4) Napište program, který připočítává 25% daň, např.:
vstup : Zadejte cenu bez dane : 100
výstup : Prodejní cena s daní (25%) : 125
- 5) Napište program, který vypočte obsah obdélníka, např.:
vstup : Zadejte delku a sirku : 5 4
výstup : Obdelnik o delce 5 a sirce 4 ma obsah 20
- 6) Jsou-li dány definice: `int a = 2, b = 2, c = 1, d = 0, e = 4;` pak napište programy, které vypíší hodnoty následujících výrazů a správnost zkontrolujte vlastním výpočtem.
 - a) `a++ / ++c * --e`
 - b) `--b * c++ - a`
 - c) `-b - --c`
 - d) `++a - --e`
 - e) `e / --a * b++ / c++`
 - f) `a %= b = d = 1 + e / 2`
- 7) Napište program, který přečte reálné číslo a vypíše jeho celou část. Zkuste využít více způsobů získání celé části.

- 8) Napište program, který přečte tři velká písmena a vypíše je jako tři malá písmena.
- 9) Napište program, který přečte tři malá písmena a vypíše je jako tři velká písmena v obráceném pořadí.
- 10) Napište program, který vypíše maximální číslo, které je možno uložit do `unsigned int` a do `signed int`.
Pomůcka: `-1` jako `signed int` je maximální `unsigned int` a maximální `signed int` je `1/2` maximálního `unsigned int`.
- 11) Napište program, který zjistí totéž pro typy `short` a `long`.
- 12) Napište program, který objasní všechny způsoby použití operátoru / (dělení celočíselné i reálné) a operátoru % (dělení modulo).
- 13) Napište program, který přečte najednou tři reálná čísla a vypíše jejich aritmetický průměr na dvě desetinná místa.
- 14) Napište program, který krátce pípne.

5 Řídící struktury

Tato kapitola popisuje podstatnou část jazyka C — řízení běhu programu. Následující příkazy je vhodné pečlivě prostudovat včetně příkladů, protože se bez nich při jakémkoliv dalším programování určitě neobejdeme.

Štábní kultura:

Dosud byly uváděny rady pro psaní přehledných a dobře vypadajících programů vždy až za určitou probranou částí jazyka C. Nyní učiníme výjimku a uvedeme tyto rady souhrnně napřed, aby byly pěkně pohromadě. Za téměř každým doporučením bude následovat příklad.

- Každé vnoření logicky podřízeného úseku programu je o 2 mezery doprava. Hodnota 2 je empiricky vyzkoušená a je "tak akorát". Vnoření o tabulátor (většinou 8 mezer) je moc při větší hloubce zanoření, protože text je pak posunut příliš vpravo. Nastavíme-li menší velikost tabulátoru (což umožňují např. Borlandské editory) dělají pak tabulátory jiných rozměrů problémy při tisku.
- Prázdné řádky se vkládají podle vlastního uvážení kdekoli, kde mohou zlepšit čitelnost programu:

```
if (x == 5)
    if (y == 6)
        z = 7;

/* prazdna radka je zde velmi vhodna */

if (k == 8)
    j = 9;
```

- Levá složená závorka "{" začínající tělo funkce je vždy sama na řádce v 1. sloupci. Totéž platí pro uzavírací závorku funkce "}"

```
main()
{
    printf("Ahoj \n");
}
```

- Otevírací závorka "{" je na téže řádce jako příkazy
if else for while do switch
musí jí předcházet mezera a za ní nesmí být žádný další příkaz.

- Uzavírací závorka "}" je u těchto příkazů na samostatné řádce ve sloupci, kde začíná klíčové slovo:

```
if (x == 5) {
    y = 6;
    z = 7;
}
else {
    y = 8;
    z = 9;
}
```

- Je-li za if else for while do jeden víceřádkový příkaz, pak ho uzavřete do { }, i když to není nutné:

```
if (x == 5) { /* { zavorka není nutná, ale vhodná */
    if (y == 6)
        z = 7;
} /* } zavorka není nutná, ale vhodná */
```

- Mezi if for while switch return a následující otevírací závorkou "(" musí být jedna mezera:

```
while (x == 5)
```

- Binární operátory, např.: * / + -
přiřazovací operátory, např.: = += -= *=
operátory porovnání, např.: == <= >
a ternární operátor: ? :

musí mít jednu mezeru před a jednu mezeru za:

```
if (i <= 5) { /* zde je { zavorka nutná !!! */
    j = k * 8 + 3;
    k += 5;
} /* zde je } zavorka nutná !!! */
```

- Unární operátory, např.: ! ++ -- * &
nesmí být odděleny od operandu mezerou:

```
i++;
j += --i;
```

- Po čárce (,) a středníku (;) musí být mezera, ale před nimi ne:

```
printf("Soucet je %d\n", i * j);
```

- Pokud se celý výraz nevejde na jednu řádku, musí pokračovat na další řádce operátorem a ne operandem a musí být zarovnán na stejný sloupec, jako je začátek výrazu:


```
if (celkovy_plat >= 1000 && celkovy_plat <= 2000
    && pridavky_na_deti <= 500)
    printf("Socialni pripad \n");
```

- Obecně by každý neřídící příkaz (přiřazení, inkrement, ...) měl být na samostatné řádce. Čestnou (a málo častou) výjimkou jsou příkazy mající velmi těsnou souvislost:

```
x = wherex(); y = wherey(); /* tesna souvislost */
```

5.1 Booleovské výrazy

Jak již bylo uvedeno na str. 23, není v jazyce C implicitně typ **Boolean**. Místo něj se používá typ **int**, kde nulová hodnota (0) znamená **FALSE** a nenulová hodnota (nejčastěji 1, ale není to podmínkou) je **TRUE**.

	Pascal	C
rovnost	=	==
nerovnost	<>	!=
logický součin	AND	&&
logický součet	OR	
negace	NOT	!

další čtyři relační operátory mají stejnou syntaxi i význam

menší	<
menší nebo rovno	<=
větší	>
větší nebo rovno	>=

Pozor:

Je důležité si uvědomit rozdíl mezi porovnáním `==` a přiřazením `=`

```
i = 5
```

je celočíselný výraz s hodnotou 5, kterou také přiřazuje do proměnné `i` — změni její původní hodnotu

```
i == 5
```

celočíselný výraz poskytující 1 (**TRUE**), jestliže má proměnná `i` hodnotu 5, nebo poskytující 0 (**FALSE**), má-li `i` hodnotu jinou, než 5 hodnota proměnné `i` se ani v jednom případě nezmění

5.1.1 Zkrácené vyhodnocování logických výrazů

Zajímavou vlastností jazyka C je, že se logický součin a součet vyhodnocují ve zkráceném vyhodnocení (*short circuit*). To znamená, že argumenty jsou vyhodnocovány zleva doprava a jakmile je možno určit konečný výsledek, vyhodnocování okamžitě končí.

Tento — od standardního Pascalu rozdílný — způsob zpracování se v C často a s výhodou využívá, např. výraz:

```
if (y != 0 && x / y < z)
```

je zcela správný a k dělení nulou v tomto případě nikdy nedojde, protože první část logického výrazu: `y != 0` ukončí vyhodnocování tohoto výrazu před dělením.

V Pascalu bude mít stejně zapsaný (vnitřní závorky jsou nezbytné) výraz:

```
if ((y <> 0) AND (x / y < z))
```

za následek možné dělení nulou a tedy přerušení výpočtu.

5.1.2 Priority vyhodnocování logických výrazů

Tabulka priorit a způsobu vyhodnocování některých operátorů¹:

operátor	směr vyhodnocení
! ++ -- - + (typ)	zprava doleva
* / %	zleva doprava
+ -	zleva doprava
< <= >= >	zleva doprava
== !=	zleva doprava
&&	zleva doprava
	zleva doprava
? :	zprava doleva
= += -= *= atd.	zprava doleva
,	zleva doprava

Pozor:

V C mají aritmetické operátory a operátor porovnání vyšší prioritu než logické operátory, takže výraz:

```
if (c >= 'A' && c <= 'Z')
```

je správný, kdežto stejně napsaný pascalský výraz:

```
if (c >= 'A' AND c <= 'Z')
```

je chybný.

V logických výrazech (a nejen v nich) platí zlaté pravidlo:

“Máš-li pochyby, závorkuj !”

¹ Tabulka není úplná — obsahuje jen ty operátory, o kterých byla dosud zmínka. Úplnou tabulku viz na str. 263.

Příklady:

Pro všechny následující příklady platí definice: `int i = 1, j = 1;`

- 1) `j = j && (i = 2);` `i` bude 2, `j` bude 1, protože `j == 1` (`TRUE`) a výraz `i = 2` je `TRUE` (ne-nulový)
- 2) `j = j && (i == 3);` `j` bude 0, protože `i == 1` (`FALSE`)
- 3) `j = j || (i / 2);` `j` bude 1, protože na `(i / 2)` nezáleží
- 4) `j = !j && (i = i + 1);` `j` bude 0, `i` bude 1 (neinkrementuje se), protože `!j` je `FALSE`

Pozor:

Nezaměňovat `&&` za `&` nebo `||` za `|`. Operátory `&` a `|` představují bitové operace, a použity nesprávně v logických výrazech dají nesprávné výsledky — viz též str. 256.

5.2 Podmíněný výraz — ternární operátor

Je to opět novinka, protože Pascal podporuje pouze podmíněný příkaz. Syntaxe podmíněného výrazu je:

`výraz_podm ? výraz_1 : výraz_2`

a má význam: `if výraz_podm then výraz_1 else výraz_2`

čili výsledná hodnota výrazu je buď `výraz_1` nebo `výraz_2`, a záleží to na hodnotě výrazu `výraz_podm`

Příklad:

```
int i, k, j = 2;

i = (j == 2) ? 1 : 3;    i bude 1
k = (i > j) ? i : j;      k bude maximum z i a j, tedy 2
```

Opět je dobré si připomenout, že z výrazu se stane příkaz, ukončí-li se středníkem, např.:

`(i == 1) ? i++ : j++;` inkrementuje buď `i` nebo `j`

Štábní kultura:

- Závorky kolem podmínky nejsou nutné, ale uvádějí se pro zvýšení čitelnosti.
- Většinou se podmíněný výraz nepoužívá, protože příkaz `if-else` je mnohem čitelnější. V některých případech (klasických) je však užitečný, např. konverze znaku na malá písmena:

1) **Pascal:**

```
IF (c >= 'A') AND (c <= 'Z'))
THEN c := CHR(ORD(c) + (ORD('a') - ORD('A')));
```

2) **C:**

```
c = (c >= 'A' && c <= 'Z') ? c + ('a' - 'A') : c;
```

5.3 Operátor čárky

Pouze čtyři operátory v C zaručují vyhodnocení levého operandu před vyhodnocením pravého operandu — logický součin (`&&`), logický součet (`||`), ternární operátor (`? :`) a operátor čárky (`,`).

Syntaxe operátoru čárky je jednoduchá — výraz: `výraz_1, výraz_2`

se zpracovává tak, že je `výraz_1` vyhodnocen a výsledek tohoto vyhodnocení je zapomenut, pak se vyhodnotí `výraz_2` a to je také závěrečný výsledek po použití výrazu s operátorem čárky. Výsledek není l-hodnota.

Příklad:

```
int i = 2, j = 4;    /* toto není operator carky */
j = (i++, i - j);    /* i bude 3 a j bude -1 */
```

Poznámka:

- V tomto případě jsou závorky nutné, protože operátor čárky má nejnižší prioritu, čili vyhodnocení by bylo: `(j = i++)`, `i - j`; což není příkaz.

Štábní kultura:

Operátor čárky používejte jen v řídicích částech příkazů `for` a `while` — viz str. 54, např.:

```
for (i = 0, j = 0; i < MAX; i++, j++)
```

Pozor:

Kromě výše uvedených čtyř operátorů, nezaručují ostatní operátory žádné pořadí vyhodnocování a je tedy chybou vytvářet výrazy tohoto typu:

```
i = 1;
j = ++i - (i = 3);
```

bude-li levý operand vyhodnocen jako první, pak bude `j` rovno `-1` a `i` bude `3`, jinak `j` bude `1` a `i` bude `4`

Příklady:

- 1) Program přečte z klávesnice dva znaky a vytiskne znak s menším ordinálním číslem.

```
#include <stdio.h>

main()
```



```

{
    int c, d;

    c = getchar();
    d = getchar();
    putchar(c < d ? c : d);
}

```

- 2) Stejný program trochu jinak.

```

#include <stdio.h>
main()
{
    int c, d;

    c = getchar();
    putchar(c < (d = getchar()) ? c : d);
}

```

5.4 Příkaz if a příkaz if-else

Stejně jako v Pascalu je i v C příkaz if jeden z nejužívanějších příkazů.

Pascal	C	
IF <i>boolean_výraz</i>	if (<i>výraz</i>)	<i>závorky jsou nezbytné</i>
THEN <i>příkaz</i>	<i>příkaz</i>	

Příklady:

- 1) Program čte znak a je-li to velké písmeno, vypíše jeho ordinální číslo.

```

#include <stdio.h>
main()
{
    int c;

    c = getchar();
    if (c >= 'A' && c <= 'Z')
        printf("%d\n", c);
}

```

- 2) Tentýž příklad zapsaný v "C-čkovkém stylu", čili s často využívanou skutečností, že přiřazení je výraz:

```

#include <stdio.h>

```

```

main()
{
    int c;

    if ((c = getchar()) >= 'A' && c <= 'Z')
        printf("%d\n", c);
}

```

Poznámka:

- Úplnému pochopení této poznámky věnujte dostatek času. Vysvětluje totiž jeden z nejvíce problémových rysů jazyka C. Vnější závorky (`c = getchar()`) jsou nutné, protože bez nich, by příkaz: `if (c = getchar()) >= 'A' && c <= 'Z')` fungoval následujícím, zcela jiným způsobem². `getchar()` přečte znak a porovná ho se znakem 'A'. Je-li výsledek testu logická 0 (neboli *FALSE*), test končí a logická 0 (ne znak 'A') je přiřazena do proměnné c. Je-li výsledek testu logická 1 (neboli *TRUE*), porovnává se stále ještě nedefinovaná hodnota proměnné c se znakem 'Z'. Výsledkem porovnání je opět logická 0 nebo 1, která se nakonec logickým součinem vynásobí s logickou hodnotou z předchozího testu. Podle výsledku tohoto součinu se do proměnné c přiřadí buď logická 0 nebo logická 1.

Příkaz if je možné rozšířit i o část *else*, která se provede, když podmínka za *if* není splněna.

Pascal	C
IF <i>boolean_výraz</i>	if (<i>výraz</i>)
THEN <i>příkaz_1</i>	<i>příkaz_1;</i>
ELSE <i>příkaz_2;</i>	<i>else</i>
	<i>příkaz_2;</i>

Poznámka:

- Pokud je do sebe zanořeno více příkazů if, pak platí pravidlo, že *else* se vztahuje vždy k nejbližšímu *if*.

Příklady:

- 1) Samostatný příkaz if.

Pascal	C
IF <i>i</i> > 3	if (<i>i</i> > 3)
THEN <i>j</i> := 5;	<i>j</i> = 5;

² Uvědomme si, že operátor = má nižší prioritu než operátor >=

2) Příkaz if-else.

```

IF i > 3          if (i > 3)
THEN j := 5       j = 5;   v C je zde středník nutný!
ELSE j := 1;      else
                  j = 1;

```

3) Složený příkaz je nutné uzavřít do bloku pomocí závorek "{" a "}" a tento blok nesmí být ukončen středníkem!

```

IF i > 3          if (i > 3) {
THEN BEGIN
    j := 5;       j = 5;
    i := 7;       i = 7;
END              }          žádný středník!
ELSE j := 1;     else
                  j = 1;

```

Příklad:

Program přečte tři reálná čísla a zobrazí největší z nich.

```

#include <stdio.h>
main()
{
    double x, y, z, max;

    printf("Zadej tri realna cisla : ");
    scanf("%lf %lf %lf", &x, &y, &z);

    /* prazdna radka zvysujeprehlednost */
    /* zavorka { neni nutna, ale vhodna */
    if (x > y) {
        if (x > z)
            max = x;
        else
            max = z;
    }
    /* zavorka } neni nutna, ale vhodna */
    /* zavorka { neni nutna, ale vhodna */
    else {
        if (y < z)
            max = z;
        else
            max = y;
    }
    /* zavorka } neni nutna, ale vhodna */
    /* prazdna radka zvysujeprehlednost */

    printf("\nNejvetsi cislo je %f \n", max);
}

```

MAX = (x > (y > z) ? y : z) ? x : (y > z ? y : z)

Pozor:

V C programech se často objevuje, že místo příkazu:

`if (výraz != 0)` se často píše jen `if (výraz)`

a místo příkazu:

`if (výraz == 0)` jen `if (!výraz)`

Štábní kultura:

- V přiřazeních a testech znaků na nulu, používat `'\0'` (nulový znak), např.:
`if (c == '\0')` a ne jenom `if (c == 0)` nebo `if (!c)`
- Totéž pravidlo platí pro pointery a `NULL`³, např.:
`if (p_i == NULL)`
- Testy a podmínky by měly být uvedeny v aserci (kladné znění tvrzení) a ne v negaci, což má význam zvláště u složených podmínek:
nevhodné `if (!(c == '\0' || c == ' ' || c == '1'))`
lepší `if (c != '\0' && c != ' ' && c != '1')`
- Obsahuje-li testovaný výraz operátor přiřazení a je testován na nulu, je vhodné ji explicitně uvést, např.:
nevhodné `if (c = getchar())` — využívá impl. komparace na nulu
lepší `if ((c = getchar()) != 0)`

5.5 Iterační příkazy — cykly

Jazyk C umožňuje použít tři příkazy pro iteraci `while`, `for`, `do-while`, které jsou velmi podobné iteračním příkazům známým z Pascalu. Než se pustíme do jejich výkladu, je vhodné vysvětlit dva příkazy, které ovlivňují provádění všech cyklů.

5.5.1 Příkazy `break` a `continue`

Oba tyto příkazy lze použít ve všech typech cyklů a oba nějakým způsobem mění "normální" průběh cyklu.

break ukončuje nejvnitřnější neuzavřenou smyčku — opouští okamžitě cyklus

continue skáče na konec nejvnitřnější neuzavřené smyčky a tím vynutí další iteraci smyčky — cyklus neopouští

³ `NULL` je symbolická konstanta pro nulový pointer definovaná v `stdio.h` — viz též str. 151.

5.5.2 Příkaz while

Tento iterační příkaz testuje podmínku cyklu před průchodem cyklem — cyklus tedy nemusí proběhnout ani jednou.

Příkaz **while** používáme, závisí-li ukončovací podmínka na nějakém příkazu v těle cyklu⁴.

Pascal	C	
WHILE <i>boolean_výraz</i> DO	while (<i>výraz</i>)	závorky jsou nezbytné
<i>příkaz</i> ;	<i>příkaz</i> ;	

Příklad:

Pascal	C
WHILE <i>x</i> < 10 DO <i>x</i> := <i>x</i> + 1;	while (<i>x</i> < 10)
	<i>x</i> ++;

Příklad:

Program čte znaky z klávesnice, tisknutelné znaky opisuje na obrazovku, neviditelných si nevšímá a zastaví se po přečtení znaku "z".

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

```
main()
{
    int c;

    while ((c = getchar()) != 'z') {
        if (c >= ' ')
            putchar(c);          /* tisk znaku */
    }
    printf("\nCtení znaku bylo ukonceno. \n");
}
```

Příklad:

Tentýž příklad s ukázkou nekonečného cyklu while a s použitím příkazů **break** a **continue**.

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

⁴ Předem tedy nedokážeme říci, proběhne-li cyklus jednou nebo stokrát — srovnej s cyklem **for** — str. 52

```
main()
{
    int c;

    while (1) { /* nekonečna smyčka */
        if ((c = getchar()) < ' ')
            continue; /* zahození neviditelného znaku */
        if (c == 'z')
            break;      /* zastavení po nactení znaku 'z' */
        putchar(c);     /* tisk znaku */
    }
    printf("\nCtení znaku bylo ukonceno. \n");
}
```

Poznámka:

- Tělo příkazu **while** může být prázdné, např. často se využívá toto vynechávání mezer na vstupu :

```
while (getchar() == ' ');
```

nebo přeskočení všech bílých znaků na vstupu:

```
while ((c = getchar()) == ' ' || c == '\t' || c == '\n')
```

Štábní kultura:

- Je-li příkaz prázdný, je středník ";" odsazen vždy na nové řádce.

5.5.3 Příkaz do-while

Je to ekvivalent Pascalského příkazu **REPEAT-UNTIL**. V tomto cyklu se podmínka testuje až po průchodu cyklem — cyklus tedy proběhne nejméně jednou.

Pascal	C
REPEAT	do {
<i>příkazy</i> ;	<i>příkazy</i> ;
UNTIL NOT <i>boolean_výraz</i>	} while (<i>výraz</i>)

Příklad:

REPEAT	do
<i>i</i> := <i>i</i> - 1;	<i>i</i> --;
UNTIL <i>i</i> <= 0;	while (<i>i</i> > 0);

Pozor:

Při vyhodnocování závěrečné podmínky cyklu **do-while** je podstatný rozdíl mezi Pascalem a C. Pascal opouští cyklus při splnění podmínce, kdežto C právě naopak — při nesplnění podmínce. Cyklus **do-while** pracuje tak dlouho, dokud má podmínka **TRUE** (nenulovou) hodnotu. Další rozdíl je v tom, že tělem cyklu **do-while** může být jen jeden příkaz a používá se tedy většinou závorek { } pro složený příkaz.

Příklad:

Program je podobný jako u cyklu **while**. Opět opisuje všechny tisknutelné znaky zadané z klávesnice, neviditelných si nevšímá a zastaví se po stisku znaku "z". Rozdíl je v tom, že vypíše i tento ukončovací znak.

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

```
main()
```

```
{
    int c;

    do {
        if ((c = getchar()) >= ' ')
            putchar(c);          /* tisk znaku */
    } while (c != 'z');
}
```

Štábní kultura:

- Pokud příkaz **do-while** obsahuje složený příkaz, pak je vhodné psát podmínku **while** za uzavírací "}" závorku složeného příkazu — viz předchozí příklad. Napíšeme-li totiž **while** na novou řádku, budeme se pak při prvním pohledu do programu divit, jaký smysl má ten prázdný cyklus **while**.

5.5.4 Příkaz for

Je to typický příkaz cyklu, který použijeme v případě, že známe předem počet průchodů cyklem⁵.

Pascal

FOR *řídící_prom* := *start*

TO *stop*

DO *příkaz*

C

for (*výraz_start*; *výraz_stop*; *výraz_iter*)
 příkaz;

⁵ Například chceme, aby cyklus běžel od jedné do dvaceti. V tomto případě není vhodný cyklus **while** — viz str. 50.

Příklad:

```
FOR i := 1 TO 10 DO      for (i = 1; i <= 10; i++)
    WRITELN(i);           printf("%d\n", i);
```

Předchozí příklad ukazoval typické použití **for**. Můžeme se však setkat s mnohem komplikovanějším zápisem tohoto příkazu.

Jsme-li pak na pochybách, co má příkaz **for** vlastně dělat, je dobré si uvědomit, že se dá vždy přepsat pomocí cyklu **while** takto:

```
výraz_start;
```

```
while (výraz_stop) {
```

```
    příkaz;
```

```
    výraz_iter;
```

```
}
```

a podle tohoto schématu také ve skutečnosti cyklus **for** pracuje.

Poznámky:

- Výrazy: *výraz_start* *výraz_stop* *výraz_iter* spolu nemusí vůbec souviset a dokonce ani nemusí být uvedeny. Ovšem použití těchto možností není to nejšťastnější využití cyklu **for**.
- Pokud není některý z výrazů: *výraz_start* *výraz_stop* *výraz_iter* uveden, pak je ale nutné vždy uvést středník ";", kterým je tento chybějící výraz oddělen od ostatních — viz dále příklady.
- Cyklus **for** probíhá tak, že se na počátku vyhodnotí *výraz_start*, otestuje se, zda je *výraz_stop* pravdivý (nenulový — **TRUE**), provede se *příkaz* a nakonec se provede *výraz_iter*. Pak nastává další obrátka cyklu.
- Tak jako pro cykly **while** a **do while** se i pro ovládání cyklu **for** dají využít příkazy **break** a **continue**.

Příklady:

Pro první čtyři příklady vždy předpokládáme definici: `int i = 0;` a všechny tisknou čísla 0 až 9.

- 1) "Klasické" (a doporučené) užití **for**

```
for (i = 0; i < 10; i++)
    printf("%d ", i);
```

- 2) Využití inicializace *i* při definici — nevhodné řešení, protože není vše pohromadě

```
for ( ; i < 10; i++)
    printf("%d ", i);
```

- 3) Řídící proměnná je měněna v těle cyklu — opět nevhodné řešení

```
for ( ; i < 10; )
    printf("%d ", i++);
```


- 4) Využití operátoru čárka — časté, ale opět ne zcela vhodné použití
- ```
for (; i < 10; printf("%d ", i), i++)
 ;
```
- 5) Použití operátoru čárka při inicializaci ( $i = 1$ ,  $sum = 0$ ; ) je vhodné, při výpočtu ( $sum += i$ ,  $i++$ ) je nevhodné
- ```
int i, sum;
for (i = 1, sum = 0; i <= 10; sum += i, i++)
    ;
```
- 6) Cyklus **for** může měnit řídicí proměnnou libovolným způsobem
- ```
int i, soucin;
for (i = 3, soucin = 1; i <= 9; i += 2)
 soucin *= i;
```

**Poznámka:**

- Často se používá *nekonečný cyklus*, který má formu: `for ( ; ; )`

**Štábní kultura:**

Pro všechny cykly platí několik obecných pravidel, které je vhodné dodržovat:

- Snažíme se mít vždy pouze jednu řídicí proměnnou.
- Řídící proměnná smí být ovlivňována pouze v řídicí části cyklu a ne v jeho těle.
- U cyklu **for** musí být všechny potřebné inicializace v inicializační části — viz poslední z předchozích příkladů.
- Pokud má cyklus prázdné tělo, musí být středník ukončující tento příkaz odsazen na nové řádce.
- Příkaz **continue** je vhodné nahradit čitelnější konstrukcí **if-else**.
- Příkaz **break** by se měl v těle cyklu vyskytnout pouze v nezbytných případech a pouze na jednom místě. Vícenásobný výskyt **break** zhoršuje srozumitelnost a čitelnost programu — viz příklady na str. 50.
- Je-li to možné, preferujte cykly **while** a **for** před cyklem **do-while**, protože jsou přehlednější.

## 5.6 Příkaz switch

C obsahuje příkaz přepínače, neboli příkaz pro mnohonásobné větvení programu. Je však nutné si uvědomit, že to není úplný ekvivalent pascalského **CASE** a že má několik následujících podstatných odlišností:

- nelze jednoduše napsat prostý výčet několika hodnot pro jeden příkaz,
- výraz, podle kterého se rozhoduje, musí být typu **int**,
- každá větev přepínače musí být ukončena příkazem **break**,
- je podporována větev **default**, která se provádí, když žádná z větví nevyhovuje.
- v každé větvi může být více příkazů, které není nutno uzavírat do závorek.

**Pascal****CASE** výraz **OF**

hodnota\_1 : příkaz\_1;

...

hodnota\_n : příkaz\_n;

**END**;

**C**

**switch** (výraz) {

case hodnota\_1 : příkaz\_1; break;

...

case hodnota\_n : příkaz\_n; break;

default : příkaz\_def; break;

}

**Poznámka:**

- Výčet několika hodnot, pro které má být proveden tentýž příkaz, má formu:

case hodnota\_1\_1 :

case hodnota\_1\_2 :

...

case hodnota\_1\_m : příkaz; break;

**Pozor:**

Není-li větev přepínače (jeden nebo více příkazů) ukončena pomocí příkazu **break**, program neopouští **switch**, ale zpracovává následující větev v pořadí! A v této činnosti pokračuje do dosažení nejbližšího příkazu **break** nebo do končení příkazu **switch**.

**Příklad:**

Tento úsek programu vypíše znaky 123 po stisku klávesy 'a' nebo 'b' nebo 'c'. Po stisku 'd' vypíše 23 a po stisku jiné klávesy vypíše 3.

```
switch (getchar()) {
 case 'a' :
 case 'b' :
 case 'c' : putchar('1');
 case 'd' : putchar('2');
 default : putchar('3');
}
```

Pro "normální" (očekávanou) funkci přepínače je nutné program poopravit takto:



```

switch (getchar()) {
 case 'a' :
 case 'b' :
 case 'c' :
 putchar('1');
 break;

 case 'd' :
 putchar('2');
 break;

 default :
 putchar('3');
 break;
}

```

Štábní kultura:

- Související větve se neoddělují novou řádkou, např.:

```

switch (getchar()) {
 case 'a' :
 case 'b' :
 putchar('1');
 break;
}

```

- Příkazy větve jsou na nové řádce a odsazeny o obvyklý počet dvou mezer.
- Větev **default** se většinou píše i když je prázdná, tedy:

```

default :
 break;

```

- Příkaz **break** za posledním příkazem poslední větve není nutný, ale z konvence se dělá.
- Větev **default** nemusí být uvedena jako poslední větev přepínače<sup>6</sup>, ale z konvence je vždy jako poslední.

Poznámky:

- Příkaz **break** ruší vždy nejvnitřnější smyčku cyklu nebo ukončuje příkaz **switch**. Je tedy nutné dávat zvýšený pozor, obsahuje-li **switch** nějaký cyklus nebo naopak<sup>7</sup>.
- Příkaz **continue** nemá se **switch** nic společného!

<sup>6</sup> Pak je její ukončující **break** nutný.

<sup>7</sup> Je-li tedy **switch** např. ve smyčce **while**, pak **break** způsobí ukončení **switch** a ne **while**.

Pozor:

Příkazy za **default** se provádí vždy až tehdy, když není nalezena žádná vyhovující větev, ať již je větev **default** uvedena v přepínači kdekoli.

```

switch (getchar()) {
 default :
 printf("Nestiskl jsi ani '1' ani '2'\n");
 break;

 case '1' :
 printf("Stiskl jsi '1'\n");
 break;

 case '2' :
 printf("Stiskl jsi '2'\n");
 break;
}

```

Příklady:

- 1) Úsek programu čte znaky z klávesnice a opisuje je na obrazovku. Bílé znaky — mezeru a tabulátor — nahradí znakem "#". Jakmile přečte znak "\*", skončí okamžitě svoji činnost.

Toto řešení je uvedeno jako odstrašující příklad, jak nemá program vypadat. Míchají se totiž do sebe věci, které nemají nic společného!

```
int c = 0;
```

```

while (c != '*') {
 switch (c = getchar()) {
 case ' ' :
 case '\t' :
 putchar('#');
 continue;

 case '*' :
 break;

 default :
 putchar(c);
 break;
 }
}
/* konec switch */
/* konec while */

```



Program pracuje takto:

- Proměnná `c` byla inicializována na 0, takže 1. průchod přes `while` určitě proběhne.
  - Načtená mezera nebo tabulátor způsobí tisk znaku `"#"` a pak příkaz `continue`<sup>8</sup> způsobí přímý odskok na konec smyčky `while`, odkud je řízení předáno na její začátek a testována podmínka (`c != '*'`)
  - Načtený znak `"#"` způsobí odskok na konec `switch`, kde nastává další obrátka cyklu `while`, která ale neproběhne, protože podmínka (`c != '*'`) bude mít hodnotu `FALSE`.
  - Každý jiný načtený znak je vypsán beze změny.
- 2) Mnohem lepší řešení tohoto příkladu — test ukončovacího znaku `"#"` se provede bez průtahů hned na začátku.

```
while ((c = getchar()) != '*') {
 switch (c) {
 case ' ':
 case '\t':
 putchar('#');
 break;
 default:
 putchar(c);
 break;
 }
}
```

- 3) Kratší, ale méně přehledná úprava téhož programu. Je z ní vidět, že když ušetříme 3 řádky, můžeme program dobře "zamlžit".

```
while ((c = getchar()) != '*') {
 switch (c) {
 case ' ':
 case '\t':
 c = '#';
 default:
 putchar(c);
 }
}
```

<sup>8</sup> Který, jak víme, nemá s příkazem `switch` nic společného!

## 5.7 Příkaz `goto`

Příkaz `goto` se v dobře napsaných programech používá málokdy, protože ve strukturovaném jazyku, jako je C, se mu lze vždy vyhnout.

Pokud je použit, musí být pro to seriózní důvod a musí to být "rozumně provedené" použití.

Jedno z mála seriózních použití je výskok z vnořených (zahnížděných) cyklů — viz dále příklad.

Narozdíl od Pascalu se návěští pro `goto` nemusí předem definovat a skočit se dá téměř<sup>9</sup> libovolně, což činí příkaz `goto` zvláště nebezpečným.

Příklad:

Jedno z "klasických" použití `goto`. Využívá se proto, že zjednodušuje a zpřehledňuje program.

```
for (i = 0; i < 10; i++) {
 for (j = 0; j < 10; j++) {
 for (k = 0; k < 10; k++) {
 if (x[k] == 0)
 goto error;
 a[i] = a[i] + b[j] / x[k];
 }
 }
 goto dalsi_vypocet;
error:
printf("Nulovy delitel \n");
```

Štábní kultura:

- Nepoužívat `goto` pro výskok z jednoduché smyčky — v tomto případě je mnohem vhodnější použít `break`.

## 5.8 Příkaz `return`

Dojde-li provádění programu na příkaz `return` ukončí se provádění funkce, která tento příkaz obsahuje.

Ve funkci `main()` ukončí příkaz `return` celý program.

Často se pomocí `return` vrací nějaká hodnota, jejíž typ záleží na typu funkce<sup>10</sup>.

<sup>9</sup> Nedá se skočit např. z jedné funkce do druhé funkce.

<sup>10</sup> Podrobně viz str. 111.



Příklad:

Byla-li by operace s poli z předchozího případu provedená voláním funkce, je pak vhodnější použít příkazu **return** a ne **goto**. Pro zjištění, zda funkce splnila svoji úlohu správně, nám poslouží návratová hodnota. Její hodnota 1 bude znamenat provedení bez chyb a hodnota 0 s chybou.

```
nasobeni()
{
 int i, j, k;

 for (i = 0; i < 10; i++) {
 for (j = 0; j < 10; j++) {
 for (k = 0; k < 10; k++) {
 if (x[k] == 0)
 return (0); /* neuspech */
 a[i] = a[i] + b[j] / x[k];
 }
 }
 }
 return (1); /* uspech */
}
```

Poznámka:

- Občas se můžeme setkat s použitím funkce **exit()**<sup>11</sup>. Ta má podobný význam jako příkaz **return**. Rozdíl je v tom, že je-li vyvolána z kterékoliv funkce, ukončí bezprostředně program, bez návratu do funkce volající.

Časté chyby:

|                                        |                                                  |
|----------------------------------------|--------------------------------------------------|
| <code>if (x == 1) then</code>          | then není klíčové slovo                          |
| <code>if x == 1</code>                 | chybí závorky                                    |
| <code>if (x == 1)</code>               |                                                  |
| <code>y = x</code>                     | chybí středník                                   |
| <code>else</code>                      |                                                  |
| <code>x++;</code>                      |                                                  |
| <code>if (x = 1)</code>                | přiřazení (=) místo porovnání (==)               |
| <code>if (c = getchar() == '*')</code> | chybějící závorky                                |
|                                        | má být: <code>if ((c = getchar()) == '*')</code> |

<sup>11</sup> Je popsána v hlavičkovém souboru **stdlib.h**.

**while** (x == 1) do                      za **while** není do  
**for** (i = 0; i < 10; i++);        zde nemá být středník  
    x += i;

Co je dobré si uvědomit:

- pro přiřazování se používá operátor =
- pro porovnání se užívá operátor ==
- logické **&&** (AND) a **||** (OR) mají zkrácené vyhodnocování
- pro ukončení smyčky cyklu se používá příkaz **break**
- za každou větví příkazu **switch** musí být **break** — není-li ovšem mezi jednotlivými větvemi souvislost
- jste-li na pochybách s prioritou, závorkujte
- vyhýbejte se důsledně podezřelým a komplikovaným kódům a zápisům

Cvičení:

- 1) Upravte pomocí explicitní typové konverze dělení `f = i / j`; tak, aby byla v proměnné `f` hodnota 2.5, kterou vypíšete. V přiřazení hodnot proměnným `i` (5) a `j` (2) ověřte také funkci operátoru čárky.
- 2) Napište program, který přečte dvě celá čísla a pomocí ternárního operátoru (`? :`) vytiskne např.:  
     Mensi cislo : 5  
     Vetsi cislo : 8
- 3) Napište program, ve kterém budou tyto příkazy:  
     i = 5;  
     printf("%d \n", i == 8);  
     printf("%d \n", i = 8);  
     printf("%d \n", i == 8);  
     a zdůvodněte dosažené výsledky.
- 4) Napište program, který přečte dva znaky v rozsahu 0 – 9 nebo A – F. Pozor — nečtěte jedno hexadecimální číslo! Tyto dva znaky pak považujte za hexadecimální číslo a jeho hodnotu vypíšete dekadicky.
- 5) Vyzkoušejte funkci programu:  
     int c;  
     if ((c = getchar()) >= 'A' && c <= 'Z')  
         printf("%d \n", c);  
     se závorkami: (c = getchar())



a bez závorek: `c = getchar()`

- 6) Přečtěte znak, a není-li to ani písmeno ani číslice, pak vytiskněte nápis: **Interpunkční znak**

v ostatních případech vytiskněte nápis: **Alfanumerický znak**

Použijte jen jednoho příkazu `if-else` a nezapomeňte na malá písmena.

- 7) Přečtěte jeden znak. Podle jeho hodnoty vypište jeden z následujících nápisů a přečtený znak vypište v apostrofech, např:

`interpunkcni znak : '>'`

`cislice : '3'`

`velke pismeno : 'K'`

`male pismeno : 'd'`

- 8) Ověřte na příkladu, že podmínky:

`if (vyraz != 0)      a      if (vyraz)`

mají totožnou funkci. Totéž ověřte i pro opačné podmínky:

`if (vyraz == 0)      a      if (!vyraz)`

- 9) Dopište následující program tak, aby bylo možné ověřit zkrácené vyhodnocování `&&` podmínky:

`if (i == 5 && ++j == 3)`

`printf("podminka platna\n");`

- 10) Dopište následující program tak, aby bylo možné ověřit zkrácené vyhodnocování `||` podmínky:

`if (i == 5 || ++j == 3)`

`printf("podminka platna\n");`

- 11) Čtete znaky z klávesnice až do `'\n'` a spočítejte počet velkých a malých písmen. Ostatních znaků si nevšímejte.

Použijte cyklu `while`.

- 12) Napište program, který přečte `n` čísel z klávesnice (`n` se zadá také z klávesnice). Pro každý vstup vypište např:

Zadej 1. číslo :

Z těchto čísel určete, kolik jich leží v intervalu `<25, 38>`.

Použijte cyklu `for`.

- 13) Na programu, který vypisuje přirozená čísla (1, 2, 3, ...) vyzkoušejte činnost příkazů `break` a `continue`.

Použijte cyklu `for`.

- 14) Napište program, který pro zadané číslo vypočte faktoriál. Použijte typ `long int`.

- 15) Napište program, který přečte dvě kladná reálná čísla `f` a `g` a pak vypíše všechna sudá celá čísla v intervalu `<f, g>`.

## 6 Vstup ze souboru a výstup do souboru

V současných programech se pro naprostou většinu vstupů používají soubory (uživatel programu zadává z klávesnice max. několik krátkých údajů, ale určitě ne stovky čísel). Stejně tak programy, kromě perfektní práce s obrazovkou, ve velké většině umožňují zápis výsledků své práce do souboru. Z těchto důvodů je nezbytné práci se soubory v jazyce C bezpečně ovládat.

### Poznámka:

- V následujících odstavcích budou vysvětleny některé "jemnosti", které se týkají souborů. Pokud se zajímáte zatím jen o to, jak ze souboru číst a jak do něj zapisovat klidně přejděte na str. 65.

Z implementačního (hardwareového) hlediska je každý soubor posloupnost Bytů<sup>1</sup> uložených na nějakém médiu (nejčastěji 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 podle pravidel daných operačním systémem. Přístup k souboru je možný jak sekvenčně tak i náhodně<sup>2</sup>.

Z uživatelského hlediska je soubor posloupnost po sobě jdoucích Bytů od začátku do konce souboru. Starostí operačního systému je, aby nám tyto Byty dodal v tom správném pořadí.

Z důvodů co největšího omezení počtu periferních operací (tj. zvýšení rychlosti), jsou I/O operace bufferované<sup>3</sup>, tzn.:

- Pro vstup:

Najednou se přečte celý blok dat z disku do paměti (bufferu). Jednotlivé položky se pak čtou z paměti a ne přímo z disku. Potřebujeme-li tedy přečíst např. 2 znaky ze souboru, načte se do paměti celý úsek souboru (např. blok dat velikosti 512 Byte), ve kterém jsou tyto 2 znaky. Pak se z této paměti přečte první znak a potom i druhý znak — již bez přístupu na vnější médium, čili podstatně rychleji.

<sup>1</sup> Pro větší odlišení bytu a bitu, bude dále psáno *Byte* s velkým "B".

<sup>2</sup> My se budeme dále zabývat pouze sekvenčním přístupem — s výjimkou kapitoly 6.12.

<sup>3</sup> Pro tyto soubory se často používá výraz *proudý dat* (*stream*).



- Pro výstup:

Data se zapisují ne přímo na disk, ale do bufferu (paměti) a když je plný, zapíše se automaticky celý obsah bufferu na disk do souboru jako jeden blok dat. Výhodou je opět větší rychlost zapisování do paměti než na disk.

#### Poznámky:

- Knihovna funkcí UNIXu i MS-DOSu umožňuje používat i nebufferované I/O (viz soubor io.h), což znamená, že každá I/O operace je okamžitě provedena. ANSI C možnost nebufferovaných I/O operací přímo nepřipouští. Je ale možné, např. pomocí standardní funkce `setvbuf()`, nastavit bufferování blokové, řádkové nebo žádné. Těmito možnostmi I/O se nebudeme dále zabývat, čili všechny další programy budou využívat výhod bufferování přednastaveného operačního systému.
- Vstupy z klávesnice a výstupy na obrazovku (tzv. "interaktivní I/O") se mohou považovat za speciální případ souborových I/O.

Krátké osvěžení paměti, jak to je v Pascalu:

|                                     |                                                                                         |
|-------------------------------------|-----------------------------------------------------------------------------------------|
| <code>VAR f : TEXT;</code>          | - definice proměnné <code>f</code> , pomocí níž se bude přistupovat k textovému souboru |
| <code>ASSIGN(f, 'DATA.TXT');</code> | - "spojení" proměnné <code>f</code> se souborem <code>DATA.TXT</code>                   |
| <code>RESET(f);</code>              | - otevření souboru pro čtení                                                            |
| <code>REWRITE(f);</code>            | - otevření souboru pro zápis                                                            |
| <code>READ(f, c);</code>            | - čtení znaku ze souboru                                                                |
| <code>WRITE(f, c);</code>           | - zápis znaku do souboru                                                                |
| <code>CLOSE(f);</code>              | - uzavření souboru po ukončení práce s ním                                              |

#### Poznámky:

- Příkaz `CLOSE(f)`; je často nutný, protože po ukončení programu není vždy zajištěno, že je soubor automaticky celý zapsán na disk a uzavřen (*flush*). V praxi to znamená, že obsah posledního bufferu už nemusí být z paměti zapsán do souboru. Projeví se to tak, že do souboru něco prokazatelně programem zapisujeme, a v souboru to ale po skončení programu není.
- Pascal rozeznává dva typy souborů:

|                                   |                                                                           |
|-----------------------------------|---------------------------------------------------------------------------|
| <code>VAR f : FILE OF INT;</code> | - soubor s udaným typem (typ <code>INT</code> je uveden jen jako příklad) |
|-----------------------------------|---------------------------------------------------------------------------|

|                            |                  |
|----------------------------|------------------|
| <code>VAR f : TEXT;</code> | - textový soubor |
|----------------------------|------------------|

- ANSI C umožňuje rozlišit binární soubor a textový soubor. Je ovšem nutné hned poznamenat, že s oběma typy souborů lze pracovat naprosto stejně a že se liší jen v několika maličkostech.
- Na str. 80 bude uvedena ukázka programu a jeho výstupu, ze které bude patrné, jaký je rozdíl mezi textovými a binárními soubory v MS-DOSu.

#### Pozor:

- Konec souboru je někdy ukončen speciálním znakem, např. v CP/M a MS-DOS znak "Ctrl Z" (= 1Ah = 26 decimálně), ale mnohé systémy (MS-DOS, UNIX, VAX/VMS) tento znak (nebo podobný znak se stejným významem) nemusí (ale mohou) využívat, protože operační systém může skutečnost, že je na konci souboru, zjistit ze známé délky souboru.
- Znak "Ctrl Z" je ale často používán jako znak konce souboru (EOF) při vstupu z terminálu.

## 6.1 Začátek práce se souborem

Nejdůležitější rada je: "Nehledat v práci se soubory žádné složitosti."

Základní datový typ pro práci se souborem v jazyce C je:

`FILE *` — což je pointer na objekt typu `FILE`<sup>4</sup>

Definice proměnné `f` pro práci se souborem:

`FILE *f;`

#### Poznámky:

- Identifikátor<sup>5</sup> `FILE` musí být velkými písmeny!
- Proměnná `f` se dá použít jak pro čtení, tak i pro zápis do souboru. Proměnné pro různou práci se soubory se tedy, stejně jako v Pascalu, od sebe nijak neliší.

Chceme-li definovat více proměnných, čili pracovat s více soubory najednou (např. pro čtení a zápis), musí se znak "\*" (hvězdička) opakovat, tj.:

`FILE *fr, *fw;`

<sup>4</sup> Co to vlastně znamená se dozvíme na str. 147 a zatím se s tím nebudeme zatěžovat.

<sup>5</sup> Není to klíčové slovo, ale nový typ.



Štábní kultura:

- Je vhodné používat identifikátor `*fr` pro soubor, který čteme, a identifikátor `*fw` pro soubor, do něhož zapisujeme.
- Tentýž identifikátor (např. `*f`) by neměl být používán v jednom programu jak pro čtení se souboru, tak i pro zápis do něho.

Chceme-li se souborem pracovat, je třeba ho otevřít, přičemž způsob otevření souboru určuje dále, jaké činnosti budeme moci se souborem provádět.

**6.1.1 Otevření souboru pro čtení**

Soubor POKUS bude možné jen číst.

**Pascal** **C**  
`ASSIGN(f, 'POKUS');` `f = fopen("POKUS", "r");` `"r"`<sup>6</sup> jako read  
`RESET(f);`

**6.1.2 Otevření souboru pro zápis**

Do souboru POKUS bude možné jen zapisovat.

**Pascal** **C**  
`ASSIGN(f, 'POKUS');` `f = fopen("POKUS", "w");` `"w"` jako write  
`REWRITE(f);`

Poznámky:

- Existují ještě další režimy otevření souboru (kromě `"r"` a `"w"`), které ale budou popsány dále — viz str. 79.
- Některé kompilátory<sup>7</sup> umožňují specifikovat režim otevření souboru jako `"wt"` nebo `"rt"` (versus `"wb"` nebo `"rb"`), kde `"t"` značí textový režim a `"b"` binární režim. Pro další příklady uvažujeme vždy textový režim, protože všechny uváděné funkce (`fprintf()`, ...) s ním implicitně pracují.
- Obecně platí, že `"w"` nebo `"r"` bez dalšího písmene, znamená otevření souboru v textovém režimu.

**6.2 Základní operace s otevřeným souborem**

V následující tabulce jsou uvedeny funkce ze standardní knihovny psané ve `stdio.h`, které umožňují pracovat se souborem<sup>8</sup>. Proměnná `f` je typu `FILE *`.

<sup>6</sup> Je nutné použít uvozovky (`"r"`) a ne apostrofy (`'r'`).

<sup>7</sup> Například od fy Borland.

<sup>8</sup> Nezapomeňte, že před použitím těchto funkcí je nutné soubor otevřít ve správném modu.

|                              |                                              |
|------------------------------|----------------------------------------------|
| čtení znaku ze souboru       | <code>c = getc(f)</code>                     |
| zápis znaku do souboru       | <code>putc(c, f)</code>                      |
| formátované čtení ze souboru | <code>fscanf(f, "formát", argumenty)</code>  |
| formátovaný zápis do souboru | <code>fprintf(f, "formát", argumenty)</code> |

Pozor:

U funkce `putc()` je první parametr zapisovaný znak a druhý soubor. To se často plete s funkcí `fprintf()`, která má jako první parametr soubor.

Poznámka:

- Pro osvěžení paměti a také jako ukázkou, že se práce se soubory příliš neliší od práce s obrazovkou a klávesnicí, je uveden i přehled korespondujících (již známých) funkcí.

|                                |                                          |
|--------------------------------|------------------------------------------|
| čtení znaku z klávesnice       | <code>c = getchar()</code>               |
| zápis znaku na obrazovku       | <code>putchar(c)</code>                  |
| formátované čtení z klávesnice | <code>scanf("formát", argumenty)</code>  |
| formátovaný zápis na obrazovku | <code>printf("formát", argumenty)</code> |

**6.3 Ukončení práce se souborem**

Po skončení práce se souborem — už z něho nebudeme dále číst nebo do něho nebudeme dále zapisovat — je nutné tuto skutečnost operačnímu systému nějak sdělit. Tato akce se jmenuje uzavření souboru a provádí se pomocí funkce `fclose(f)`, kde `f` je typu `FILE *`.

Poznámka:

- Ačkoliv se v mnoha systémech uzavírají soubory po ukončení programu automaticky, je velmi špatným zvykem se na to spoléhat, už jen proto, že počet současně otevřených souborů je omezený. Dalším dobrým důvodem, proč uzavřít soubor ihned po ukončení práce s ním, je zápis bufferu do souboru (viz též str. 64). Při případné další havárii programu se pak nemůže stát, že v souboru budou chybět některá data, která jsme do něj sice programem zapsali, ale zůstala jen v bufferu.

**6.4 Příklady základní práce se soubory**

Program vytvoří soubor POKUS.TXT a zapíše do něho čísla od 1 do 10, každé na novou řádku.

```
#include <stdio.h>
main()
```



```
{
FILE *fw;
int i;

fw = fopen("POKUS.TXT", "w");
for (i = 1; i <= 10; i++)
 fprintf(fw, "%d \n", i);
fclose(fw);
}
```

Pokud si po skončení programu prohlédneme soubor POKUS.TXT libovolným editorem nebo jej pomocí příkazu operačního systému<sup>9</sup> vypíšeme na obrazovku, uvidíme smysluplné výsledky.

Program přečte tři double čísla ze souboru DATA.TXT<sup>10</sup> a vypíše na obrazovku jejich součet.

```
#include <stdio.h>
main()
{
FILE *fr;
double x, y, z;

fr = fopen("DATA.TXT", "r");
fscanf(fr, "%lf %lf %lf", &x, &y, &z);
printf("%f\n", x + y + z);
fclose(fr);
}
```

#### Poznámka:

- Funkce fscanf() vrací počet úspěšně přečtených položek<sup>11</sup>. Lze tedy jednoduše otestovat případ, kdy soubor DATA.TXT nebude obsahovat všechna tři double čísla.

```
if (fscanf(fr, "%lf %lf %lf", &x, &y, &z) == 3)
 printf("%f\n", x + y + z);
else
 printf("Soubor DATA.TXT neobsahuje 3 realna cisla\n");
```

<sup>9</sup> V MS-DOSu příkazem: type pokus.txt

<sup>10</sup> Vytvořeného dříve např. editorem, přičemž je lhostejné, zda budou čtená čísla ležet v souboru DATA.TXT na jedné řádce oddělená mezerou nebo každé číslo na samostatné řádce.

<sup>11</sup> V případě čtení na konci souboru vrací fscanf() hodnotu EOF.

Program přečte dva znaky ze souboru ZNAKY.TXT a zapíše je do souboru KOPIE.TXT.

#### Pozor:

Znaky se čtou do proměnné typu int — důvod viz str. 71.

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

fr = fopen("ZNAKY.TXT", "r");
fw = fopen("KOPIE.TXT", "w");

c = getc(fr); /* ctení prvního znaku */
putc(c, fw); /* zapis prvního znaku */
putc(getc(fr), fw); /* ctení a zapis druhého znaku */

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

## 6.5 Testování konce řádky

Programátoři v Pascalu jsou zvyklí, že se v textových souborech přečte najednou celá řádka ukončená znaky <CR><LF>, např. pomocí READLN. Tato možnost v C přímo není<sup>12</sup>, a proto, když chceme přečíst jen jednu řádku, musíme se sami postarat o to, že skutečně přečteme jen ji.

#### Poznámky:

- Konec řádky se v literatuře často označuje jako *End-of-Line* (EOLN).
- Konstanta EOLN není nikde definována jako symbolická konstanta<sup>13</sup>.

V textových souborech se může vyskytnout menší problém s označením konce řádky. Je totiž možné použít tři základní přístupy:

- 1) Samostatný znak <CR> (carriage return — návrat vozíku)  
znakově: '\r', hexadecimálně: 0xD, dekadicky: 13  
– posun (kurzoru) na začátek téže řádky

<sup>12</sup> Zajišťuje ji ale plně standardní funkce fgets() — viz str. 207.

<sup>13</sup> Narozdíl od konstanty EOF — viz str. 71.



- 2) Samostatný znak <LF> (linefeed — posun o řádku)  
 znakově: '\n', hexadecimálně: 0xA, dekadicky: 10  
 – posun (kurzoru) na novou řádku v tomtéž sloupci
- 3) Znak <CR> následovaný bezprostředně znakem <LF>. Opačně, tedy dvojice znaků <LF><CR> se vyskytuje málokdy.

Jaký se použije způsob označení konce řádky, záleží na okolnostech např. po stisku klávesy <Enter> (někdy označovaná jako <Return>) na klávesnici dostáváme kód znaku <CR>, ale na obrazovku jsou vyslány dva znaky — <CR> a <LF>.

Naštěstí výše popsané tři způsoby nemusí programátora většinou zajímat, protože C poskytuje standardní znak pro konec řádky ('\n'), který se dá v naprosté většině případů práce s textovými soubory použít pro všechny tři výše uvedené způsoby. Používá se jak pro testování konce řádky tak i pro zápis konce řádky — viz dále. Tento "trik" je umožněn díky tomu, že soubory jsou standardně otevírány jako textové soubory a pak také díky "moudrosti" překladače, který ví, jakým způsobem konkrétní systém označuje konec řádky v souboru. V UNIXu žádné problémy nejsou (používá se jen <LF>) a pokud vás zajímá, jak to vypadá v MS-DOSu, pak viz str. 80.

Z tohoto povídání je důležité si zapamatovat pouze to, že interpretace znaku '\n' je vnitřní záležitost překladače a my se o ni nemusíme dlouhou dobu starat, ať pracujeme pod jakýmkoliv operačním systémem. V naprosté většině případů stačí tedy pracovat pouze s '\n' jak pro čtení, tak i pro zápis.

#### Poznámka:

- Ve speciálních případech (např. při psaní do okénka na obrazovce pod MS-DOSem — což ale není práce se souborem) a většinou jen při zápisu je třeba použít jak znaku '\n', tak i znaku '\r'.

Je třeba si uvědomit, že znak '\n' je znakem jako každý jiný a může být tedy použit všude tam, kde C dovoluje použít znak.

Jak již bylo řečeno v úvodu, funkce `getchar()` nebo `getc()` neprovádí test konce řádky. Tato činnost je zcela věcí programátora. Základní trik pro čtení až do konce řádky je tedy:

```
if ((c = getc(fr)) != '\n')
```

#### Příklad:

Následující program přečte jednu řádku ze souboru `DOPIS.TXT` a opíše ji na obrazovku včetně znaku nové řádky.

```
#include <stdio.h>
main()
```

```
{
 int c;
 FILE *fr;

 fr = fopen("DOPIS.TXT", "r");
 while ((c = getc(fr)) != '\n')
 putchar(c);
 putchar(c); /* vypis '\n' - odradkovani */
 fclose(fr);
}
```

## 6.6 Testování konce souboru

Testování konce souboru je v C možné dvěma rovnocennými způsoby. U textových souborů záleží jen na vás, zda budete používat pro testování konce souboru konstantu `EOF` nebo makro `feof()`. Je nutné poznamenat, že při použití makra `feof()` bude program pravděpodobně pomalejší, neboť je zde navíc volání tohoto makra.

### 6.6.1 Pomocí symbolické konstanty `EOF`

Při čtení na konci souboru se automaticky vrací konstanta `EOF`, která je většinou definována v souboru `stdio.h` a má<sup>14</sup> většinou hodnotu `-1`.

Základní trik pro čtení všech znaků až do konce souboru je tedy:

```
if ((c = getc(fr)) != EOF)
```

#### Poznámka:

- Proměnná `c` nesmí být definována jako `char`, protože konstanta `EOF` je reprezentována často `int` hodnotou `-1`. Ta by byla konvertována na `char` a tedy na něco jiného<sup>15</sup> než je `-1`.

#### Příklad:

Následující program zkopíruje soubor `ORIG.TXT` do souboru `KOPIE.TXT`.

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

<sup>14</sup> Ale nemusí! Je třeba využívat `EOF` místo `-1`.

<sup>15</sup> Nová hodnota by záležela na implementaci, např. je-li `char` implicitně `unsigned`, pak se `-1` konvertuje na `255`.



```
fr = fopen("ORIG.TXT", "r");
fw = fopen("KOPIE.TXT", "w");
```

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

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

### 6.6.2 Pomocí standardního makra feof()

Konec souboru lze také testovat standardním makrem `feof()`, které vrací hodnotu `TRUE` (nenulovou), pokud poslední čtení bylo již za koncem souboru, nebo hodnotu `FALSE` (nulu), pokud jsme při čtení na konec souboru ještě nedošli.

Tento způsob je vhodnější, čteme-li znaky (Byty) z binárního souboru, protože v něm se totiž může objevit Byte s libovolnou hodnotou — tedy i s hodnotou `0xFF`. Ten pak může být pomocí implicitní typové konverze převeden na hodnotu `EOF`. V tomto případě by čtení ze souboru pomocí `EOF` skončilo uprostřed souboru.

Program, který zkopíruje soubor `ORIG.TXT` do souboru `KOPIE.TXT` pomocí makra `feof()`, vypadá takto:

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

```
main()
```

```
{
 FILE *fr, *fw;
 int c;
```

```
fr = fopen("ORIG.TXT", "r");
fw = fopen("KOPIE.TXT", "w");
```

```
while (c = getc(fr), feof(fr) == 0) {
 putc(c, fw);
}
```

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

## 6.7 Testování správnosti otevření a uzavření souboru

Protože otevírání a zavírání souboru jsou akce, které se z nejrůznějších příčin nemusí povést<sup>16</sup>, je velmi vhodné mít možnost otestovat, zda tyto akce proběhly správně, a podle výsledku testu buď pokračovat nebo nepokračovat ve zpracování programu. Jazyk C nám tuto možnost poskytuje a je velmi vhodné ji využívat. To, co se zpočátku zdá jako zbytečnost, nám v budoucnosti ve složitějším programu význačně usnadní ladění, protože tyto chyby jsou velmi časté.

Jak funkce `fopen()`, tak i `fclose()` jsou funkce, které vrací hodnotu<sup>17</sup>. Tato hodnota se používá pro test, zda příkaz otevření či uzavření souboru proběhl správně.

Při nesprávně provedeném otevření souboru vrací `fopen()` konstantu `NULL`<sup>18</sup>. Použití je tedy<sup>19</sup>:

```
if ((fr = fopen("TEST.TXT", "r")) == NULL)
 printf("Soubor TEST.TXT se nepodarilo otevrit\n");
```

Při nesprávně provedeném uzavření souboru vrací `fclose()` hodnotu `EOF`. Použití je tedy:

```
if (fclose(fr) == EOF)
 printf("Soubor se nepodarilo uzavrit\n");
```

### Příklad:

Následující program je přepsaný program ze str. 72 tak, jak by měl vypadat s ošetřením všech souborových operací.

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

```
main()
```

```
{
```

```
 FILE *fr, *fw;
 int c;
```

```
if ((fr = fopen("ORIG.TXT", "r")) == NULL) {
 printf("Soubor ORIG.TXT se nepodarilo otevrit\n");
 return; /* ukončení programu */
}
```

<sup>16</sup> Nejčastější příčinou je, že se spletete ve jménu souboru.

<sup>17</sup> Podrobně viz str. 113

<sup>18</sup> Konstanta `NULL` je definována v `stdio.h` a má většinou hodnotu 0.

<sup>19</sup> Žádná kulatá závorka nesmí být vynechána — viz str. 47.



```

if ((fw = fopen("KOPIE.TXT", "w")) == NULL) {
 printf("Soubor KOPIE.TXT se nepodarilo otevrit\n");
 return; /* ukončení programu */
}

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

if (fclose(fr) == EOF) {
 printf("Soubor ORIG.TXT se nepodarilo uzavrit\n");
 return; /* nevhodné */
}

if (fclose(fw) == EOF) {
 printf("Soubor KOPIE.TXT se nepodarilo uzavrit\n");
 return; /* zbytečné */
}
}

```

**Poznámka:**

- Uvedený program obsahuje jednu logickou chybu. Kdyby se totiž podařilo soubor ORIG.TXT správně otevřít pro čtení a soubor KOPIE.TXT se nepodařilo otevřít pro zápis, pak by program skončil a soubor ORIG.TXT by zůstal stále otevřený a musel by ho zavřít operační systém.

Tato chyba není nikterak velká, protože operační systém by za nás zavření provedl, ale je dobrým programátorským zvykem nespolehat se na něco, co *může* nebo *měl by* udělat někdo jiný.

**Příklad:**

Program vypíše soubor DOPIS.TXT na obrazovku tak, že zkonvertuje všechna malá písmena na velká a nakonec vypíše délku nejdelší přečtené řádky.

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

```
main()
```

```
{
```

```
 FILE *fr;
```

```
 int c,
```

```
 nejdeli = 0,
```

```
 pocet = 0;
```

```

if ((fr = fopen("DOPIS.TXT", "r")) == NULL) {
 printf("Soubor DOPIS.TXT se nepodarilo otevrit\n");
 return;
}

while ((c = getc(fr)) != EOF) {
 putchar(c >= 'a' && c <= 'z' ? c + 'A' - 'a' : c);
 if (c == '\n') {
 if (nejdeli < pocet)
 nejdeli = pocet;
 pocet = 0;
 }
 else
 pocet++;
}

printf("Nejdeli radka obsahovala %d znaku.\n", nejdeli);

if (fclose(fr) == EOF) {
 printf("Soubor DOPIS.TXT se nepodarilo uzavrit\n");
 return;
}
}

```

## 6.8 Standardní vstup a výstup

Možná, že jste si již všimli, že práce se soubory se moc neliší od práce s obrazovkou a klávesnicí. Tento postřeh je správný. C totiž ve skutečnosti pracuje s klávesnicí a obrazovkou jako se souborem.

V souboru `stdio.h` jsou definovány dva konstantní pointery<sup>20</sup>, které představují dva soubory, otevřené operačním systémem při spuštění programu. Jsou to:

```

FILE *stdin; Pozor: ne *stdio!
FILE *stdout;

```

Tyto pointery se označují jako standardní vstupní/výstupní proud (*standard input/output stream*) a většinou představují vstup z klávesnice a výstup na obrazovku. Tyto I/O proudy je možno v mnoha systémech (UNIX, MS-DOS) jednoduše změnit pomocí přesměrování (*redirekce*) např. na vstup ze souboru nebo výstup do souboru bez zásahu do vlastního programu.

<sup>20</sup> Nejsou to proměnné — nedá se jim přiřadit hodnota.



Například v MS-DOSu program TISKNI.EXE spuštěný příkazem:

```
A:\>tiskni
```

tiskne něco na obrazovku. Pokud ale použijeme stejný program a spustíme jej příkazem:

```
A:\>tiskni > vystup.txt
```

nebude program TISKNI.EXE vypisovat nic na obrazovku, ale celý svůj výstup zapíše do souboru VYSTUP.TXT, který sám vytvoří, otevře a nakonec uzavře.

#### Poznámky:

- V souboru `stdio.h` je definován ještě třetí proud `stderr`, který se používá pro vypisování chybových zpráv programu.
- V aplikacích pod MS-DOSem bývají také navíc:
  - `stdaux` – seriové rozhraní
  - `stdprn` – paralelní rozhraní (většinou tiskárna)
- `stdin` a `stdout` mohou být použity v programu jako argumenty operací se soubory, např.:

|                              |                 |                         |
|------------------------------|-----------------|-------------------------|
| <code>getc(stdin)</code>     | je ekvivalentem | <code>getchar()</code>  |
| <code>putc(c, stdout)</code> | je ekvivalentem | <code>putchar(c)</code> |

#### Příklad:

Následující program ukáže, jak lze využít `stdout` a také jak se vyhnout problémům při čtení bufferovaného vstupu. Při čtení z tohoto vstupu je třeba mít na paměti, že ačkoliv očekáváme pouze jeden znak, může přijít celý zbytek řádky, tedy např. i znak `'\n'`, protože uživatel musel stisknout klávesu `<Enter>`, aby mohl být operačním systémem do bufferu zapsaný znak z tohoto bufferu programátorem přečten. Je tedy nutné přečíst z bufferu všechny znaky, které se v něm nacházejí, čímž se v budoucnu vyhneme problémům, že program čte znak (nejčastěji `'\n'`), který uživatel napsal nechtěně, ale nuceně z klávesnice.

Program nejdříve vypíše dotaz, zda má být výstup vypsán na obrazovku nebo do souboru VYSTUP.TXT. Pokud uživatel zvolí soubor VYSTUP.TXT, program jej zkouší otevřít pro čtení<sup>21</sup>, čímž testuje, zda již tento soubor existuje. V kladném případě vypíše druhý dotaz, zda má být tento soubor přepsán.

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

```
main()
```

```
{
```

```
 FILE *fw;
```

<sup>21</sup> Nutno poznamenat, že je to trik, který předpokládá, že se existující soubor podaří vždy otevřít, což za nepříznivých podmínek nemusí být vždy splněno.

```
int c;
```

```
printf("Stisknete 0 pro vypis na Obrazovku \n");
```

```
printf("nebo jiny znak pro zapis do souboru VYSTUP.TXT : ");
```

```
c = getchar();
```

```
/* vyprazdneni bufferu - preskoci zbytek radky */
```

```
while (getchar() != '\n')
```

```
;
```

```
if (c == 'o' || c == '0')
```

```
 fw = stdout;
```

```
else {
```

```
 if ((fw = fopen("VYSTUP.TXT", "r")) != NULL) {
```

```
 printf("Soubor VYSTUP.TXT existuje, prepsat? [A/N]: ");
```

```
 c = getchar();
```

```
 while (getchar() != '\n')
```

```
;
```

```
 if (fclose(fw) == EOF) {
```

```
 printf("Chyba pri uzavirani souboru\n");
```

```
 return;
```

```
 }
```

```
 if (!(c == 'a' || c == 'A'))
```

```
 return;
```

```
 /* konec programu */
```

```
 }
```

```
 if ((fw = fopen("VYSTUP.TXT", "w")) == NULL) {
```

```
 printf("Soubor VYSTUP.TXT se nepodarilo otevrit\n");
```

```
 return;
```

```
 }
```

```
 printf("Piste text a ukoncete jej znakem * \n");
```

```
 while ((c = getchar()) != '*')
```

```
 putc(c, fw);
```

```
 if (fw != stdout && fclose(fw) == EOF) {
```

```
 printf("Soubor VYSTUP.TXT se nepodarilo uzavrit\n");
```

```
 return;
```

```
 }
```

```
}
```



Poznámka:

- Smyčka: `while (getchar() != '\n')`

je nezbytná proto, že po stisku prvního znaku (zápis na obrazovku nebo do souboru) následuje ještě znak <Enter>, který by byl z bufferu přečten jako odpověď na dotaz zda přepsat soubor VYSTUP.TXT.

## 6.9 Vrácení přečteného znaku zpět do vstupního bufferu

Při programování reálných aplikací se v mnoha případech při čtení znaků dozvídáme, že máme přestat číst, až když přečteme jeden znak navíc. Tento znak ale není možné vždy "zahodit", protože je součástí — začátkem — další informace.

V těchto případech je možné použít funkci `ungetc(c, fr)`, která vrátí naposledy přečtený znak zpět do vstupního bufferu. Provede-li se vrácení znaku úspěšně, funkce `ungetc()` vrací pro kontrolu tento znak. Při neúspěšném provedení se vrací EOF. Obvykle lze do vstupního bufferu vrátit pouze jeden znak.

Příklad:

Následující část programu konvertuje znakový řetězec na odpovídající číselnou hodnotu.

```
int c, hodnota = 0;

while ((c = getchar()) >= '0' && c <= '9') {
 hodnota = hodnota * 10 + (c - '0');
}

ungetc(c, stdin);
```

Příklad:

Další program řeší případ, kdy je číslo v souboru uvozeno neznámým počtem znaků (v tomto případě znaků "\$") a my toto číslo chceme číst pomocí funkce `fscanf()`. Předpokládáme již otevřený soubor pro čtení.

```
int c, hodnota = 0;
```

```
while ((c =getc(fr)) == '$')
;
/* precte vsechny predchazejici znaky $ */
ungetc(c, fr); /* vraceni prvnici cislice cisla do buff. */
fscanf(fr, "%d", &hodnota);
```

Poznámka:

- Zpátky do vstupního bufferu lze vrátit samozřejmě i jiný znak než znak naposledy přečtený. Tohoto triku lze využít např. pro předvolení klávesy, která bude později jakoby stisknuta.

## 6.10 Různé možnosti otvírání souborů

Jak již víme z kap. 6.1, soubory se otevírají stále stejnou funkcí — `fopen()` — nehledě na to, zda se jedná o textový či binární soubor a také bez ohledu na to, zda se bude soubor číst nebo se bude do něho zapisovat. Všechny tyto možnosti jsou určeny druhým parametrem funkce `fopen()`, která má následující funkční prototyp:

```
FILE *fopen(const char *jmeno, const char *rezim);
```

Poznámka:

- Klíčové slovo `const`<sup>22</sup> zde neznamena, že skutečnými parametry funkce `fopen()` mohou být jen řetězcové konstanty. Znamená pouze to, že takto označené parametry jsou brány jen jako vstupní a tedy že ve funkci `fopen()` budou pouze čteny a ne měněny.

Různé významy parametru `rezim`:

|                  |                                                 |
|------------------|-------------------------------------------------|
| <code>r</code>   | — textový soubor pro čtení                      |
| <code>w</code>   | — textový soubor pro zápis nebo pro přepsání    |
| <code>a</code>   | — textový soubor pro připojení na konec         |
| <code>rb</code>  | — binární soubor pro čtení                      |
| <code>wb</code>  | — binární soubor pro zápis nebo pro přepsání    |
| <code>ab</code>  | — binární soubor pro připojení na konec         |
| <code>r+</code>  | — textový soubor pro čtení a zápis              |
| <code>w+</code>  | — textový soubor pro čtení, zápis nebo přepsání |
| <code>a+</code>  | — textový soubor pro čtení a zápis na konec     |
| <code>rb+</code> | — binární soubor pro čtení a zápis              |
| <code>wb+</code> | — binární soubor pro čtení, zápis nebo přepsání |
| <code>ab+</code> | — binární soubor pro čtení a zápis na konec     |

Poznámky:

- 1) Jak již víme ze str. 66, některé implementace umožňují explicitně označit, že se jedná o textový soubor, takže dovolují i režimy:

"rt" "wt" "at"

<sup>22</sup> Viz též str. 126.



- 2) Otevřeme-li již existující soubor v režimu "w" (nebo "wb"), pak se tento soubor nejdříve vymaže a pak znovu založí. Dochází tedy k přepsání souboru.
- 3) Otevřeme-li již existující soubor v režimu "a" (nebo "ab"), pak se tento soubor otevře a ukazatel pozice se v něm přesune na jeho konec. Dochází tedy k rozšíření existujícího souboru (append). Pokud soubor ještě neexistuje, pak se vytvoří.
- 4) Pokud použijeme režim rozšířený o znak "+", je možné soubor používat zároveň pro čtení i pro zápis, což má ale praktický význam pouze v binárních souborech.
- 5) UNIX rozeznává jen jeden oddělovač řádku a to '\n' (<LF> - 0Ah). Znak '\r' (<CR> - 0Dh) se pro oddělení nové řádky nepoužívá. Z tohoto důvodu není v UNIXu důvod rozlišovat binární a textové soubory a tedy doplnění režimů o "b" nebo "t" (např. "wb") nemá žádný význam.

## 6.11 Rozdíl při zpracovávání textových a binárních souborů v MS-DOSu

Jak se využívají binární soubory, bude popsáno v následující podkapitole. Zde si pouze ukážeme, jak systém odlišně interpretuje znak konce řádky v textových a binárních souborech.

Dopředu si řekněme, že obsah binárního souboru není nijak funkcemi zápisu a čtení ovlivňován. To znamená, že co do binárního souboru zapíšeme, to v něm přesně bude a co je zapsáno v binárním souboru, to se také přesně přečte.

Narozdíl od toho je textový soubor modifikován při zápisu — když do něj zapíšeme znak '\n' (0Ah), systém před něj automaticky dodá i znak '\r' (0Dh). Podobně je textový soubor modifikován při čtení, kdy jsou všechny znaky '\r' (0Dh) automaticky vyřazovány ("požírány").

Z následující tabulky je vidět, co se může stát, když si spleteme binární a textový soubor. Upozorňujeme ještě jednou, že dosud popisované funkce (jako např. fprintf(), fscanf(), getc(), putc(), ...) neoznámí, že pracují s binárním souborem, ale zpracovávají ho poněkud jinak než textový.

|                           |                               |
|---------------------------|-------------------------------|
| zapisováno programem:     | 61 0A 62 0D 63 0A 0D 64       |
| 1) režim zápisu "w"       |                               |
| obsah souboru A.TXT:      | 61 0D 0A 62 0D 63 0D 0A 0D 64 |
| vypsáno při režimu čtení: |                               |
| a) "r":                   | 61 0A 62 63 0A 64             |
| b) "rb":                  | 61 0D 0A 62 0D 63 0D 0A 0D 64 |
| 2) režim zápisu "wb"      |                               |
| obsah souboru A.TXT:      | 61 0A 62 0D 63 0A 0D 64       |
| vypsáno při režimu čtení: |                               |
| a) "r":                   | 61 0A 62 63 0A 64             |
| b) "rb":                  | 61 0A 62 0D 63 0A 0D 64       |

Pro tyto pokusy byl použit následující program:

```
#include <stdio.h>

main()
{
 FILE *fr, *fw;
 char rezim[3];
 int c;

 printf("\nZadej rezim zapisu : ");
 scanf("%2s", rezim);
 fw = fopen("A.TXT", 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("\nZadej rezim cteni : ");
 scanf("%2s", rezim);
 fr = fopen("A.TXT", rezim);
 while ((c = getc(fr)) != EOF)
 printf("%02X ", c);
 fclose(fr);
}
```



## 6.12 Práce s binárními soubory

V předchozích kapitolách jsme se učili pracovat pouze s textovými soubory a pokud jsme se zmiňovali o binárních souborech, pak jsme jimi jenom strašili. Textové soubory mají totiž tu ohromnou výhodu, že je možné si jejich obsah kdykoliv prohlédnout, vytvořit nebo opravit běžným editorem. Jejich nevýhodou ale je, že pro uchování stejného množství informace potřebují mnohem více prostoru. Například číslo 65535 zabere v textovém souboru prostor 5-ti Byte, zatímco v binárním souboru třeba jen 2 Byte<sup>23</sup>.

Druhou výhodou binárních souborů je, že se s nimi pracuje mnohem rychleji než s textovými soubory. Důvody jsou dva — jednak jsou binární soubory 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 náročné<sup>24</sup>. Tyto konverze v binárních souborech odpadají, protože se do nich zapisuje přímo obsah paměti po Bytech.

Z těchto důvodů se binární soubory v profesionálních programech využívají poměrně často. Nejvýhodnější je jejich použití pro ukládání rozměrných dat — velkých polí, struktur, atd. Pro pouhé uložení znaků nemají příliš význam, protože znak v textovém souboru zabírá jeden Byte stejně jako v binárním.

Pro práci s binárními soubory se využívají následující funkce:

### 6.12.1 Čtení a zápis do binárního souboru

Pro čtení dat se používá funkce `fread()` a pro zápis funkce `fwrite()`. Funkce mají tyto funkční prototypy:

```
int fread(char *kam, int velikost, int pocet, FILE *soubor);
int fwrite(char *odkud, int velikost, int pocet, FILE *soubor);
```

kde jednotlivé formální parametry mají tento význam:

|                 |                                                         |
|-----------------|---------------------------------------------------------|
| <b>kam</b>      | — adresa paměti, kam se bude ukládat přečtený blok dat  |
| <b>odkud</b>    | — adresa paměti, odkud se bude brát zapisovaný blok dat |
| <b>velikost</b> | — délka jedné položky z bloku dat <sup>25</sup>         |
| <b>pocet</b>    | — počet datových položek (ne Bytů !)                    |
| <b>soubor</b>   | — proměnná pro práci se souborem                        |

Obě funkce vrací počet skutečně úspěšně zapsaných/přečtených položek — pozor, ne Bytů!

<sup>23</sup> Záleží to na vnitřní implementaci čísla.

<sup>24</sup> U čtení dochází také ke konverzi, ale k obrácené — z textu na číslo.

<sup>25</sup> Pro její určení je vhodné použít operátor `sizeof`.

### 6.12.2 Pohyb v binárním souboru

V textových souborech se moc nevyužívá možnost pohybu v souboru — jednotlivé znaky se z nich prostě sekvenčně čtou. V binárních souborech, kde známe 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 zapisovat, bez nutnosti zdržování se čtením předchozích dat.

Pro nastavení nové pozice v souboru se využívá funkce `fseek()`:

```
int fseek(FILE *soubor, long posun, int odkud);
```

kde jednotlivé formální parametry mají tento význam:

|               |                                                               |
|---------------|---------------------------------------------------------------|
| <b>soubor</b> | — proměnná pro práci se souborem                              |
| <b>posun</b>  | — počet Bytů od pozice v souboru dané parametrem <b>odkud</b> |
| <b>odkud</b>  | — místo, odkud se bude v souboru posouvat                     |
|               | — může mít jednu ze tří hodnot:                               |
|               | • <code>SEEK_SET</code> — od začátku souboru                  |
|               | • <code>SEEK_CUR</code> — od aktuální pozice                  |
|               | • <code>SEEK_END</code> — od konce souboru                    |

Funkce `fseek()` vrací nulu (`FALSE`) v případě úspěšného přesunu nebo nenulovou hodnotu (`TRUE`) v případě neúspěšného přesunu<sup>26</sup>.

Pokud někdy potřebujeme zjistit, kde se v souboru právě nalézáme, je možné použít funkci:

```
long ftell(FILE *soubor);
```

která vrátí posunutí měřené v Bytech od začátku souboru.

### 6.12.3 Příklad použití binárního souboru

Následující program zapíše do binárního souboru `POKUS.DAT` hodnoty dvou proměnných (`i` a `d`). Soubor je otevřen jako binární a pro čtení i zápis, což umožňuje, aby se po zápisu obou proměnných a přesunu ukazatele v souboru dalo z tohoto souboru ihned číst. Všimněte si též, jak se udává adresa proměnné (stejně jako při použití funkce `scanf()`) a jak se nejlépe zjistí velikost proměnné.

<sup>26</sup> Např. při pokusu o přesun delší než je délka souboru, soubor není otevřen atd.



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

```
main()
```

```
{
 FILE *f; /* pro cteni i pro zapis */
 int i = 5;
 double d = 3.14159;

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

 printf("Pozice v souboru je %ld \n", ftell(f));
 fseek(f, 0, SEEK_SET); /* posun na zacatek 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);
}
```

#### Časté chyby:

```
if (c = getc(fr) != EOF) má být if ((c = getc(fr)) != EOF)
close(f); má být fclose(f);
```

#### Co je dobré si uvědomit:

- Pro čtení znaku ze souboru používejte vždy proměnnou typu `int`.
- Vždy testujte, zda funkce `fopen()` a `fclose()` proběhly správně.
- Uzavírejte soubor okamžitě, jakmile s ním přestanete pracovat.

#### Cvičení:

- 1) Vytvořte textový soubor `ZNAKY.TXT`, ve kterém budou náhodné znaky. Tento soubor čtěte po znacích v cyklu `do-while`. Je-li přečtený znak "q" (Quit), pak ukončete program. Je-li znak "0", pak vypište: Byla to nula

Je-li přečtený znak "1" pak vypište: Byla to jednička  
Testujte i EOF a použijte `switch`.

- 2) Vytvořte textový soubor `PISMENA.TXT`, ve kterém bude několik řádek složených z malých a velkých písmen a mezer. Tento soubor celý přečtěte a opište na obrazovku. Současně do souboru `VELKY.TXT` zapište obsah čteného souboru, ale malá písmena převedte na velká.
- 3) Napište program, který spočte celkový počet znaků souboru `PISMENA.TXT`.
- 4) Vytvořte program, který přečte soubor `PISMENA.TXT` po řádcích. Každou řádku přesně opište do souboru `KOLIK.TXT` a na nové řádce uveďte, kolik malých písmen na ní bylo.
- 5) Vytvořte program, který zapiše do souboru `CISLA.TXT` dvacet reálných čísel — násobků 3.14. Před každé číslo napište znak "\$" a každé číslo napište na samostatnou řádku. Např.:  
\$3.14  
\$6.28
- 6) Čtěte všechna reálná čísla ze souboru `CISLA.TXT` a vypočtěte jejich aritmetický průměr. Ve funkci `fscanf()` vyzkoušejte formáty čtení "\$%lf", "\$%lf\n", "%lf", "%lf\n". Konec souboru netestujte pomocí EOF, ale pomocí návratové hodnoty funkce `fscanf()`.
- 7) Napište program, který porovná obsah souborů `PISMENA1.TXT` a `PISMENA2.TXT` (vytvořte je např. pomocí příkazu operačního systému `copy`). Program vypíše jedno z těchto hlášení:  
Soubory jsou shodné  
Soubory se liší v x znacích  
Odlišné znaky průběžně vypisujte.
- 8) Napište program, který se pokusí číst neexistující soubor. Zajistěte, aby program vhodně reagoval na tuto situaci.
- 9) Napište program, který čte znaky ze souboru `PISMENA.TXT` a opisuje je buď na obrazovku, nebo do souboru `NOVY.TXT`. Uživatel má možnost si zvolit směr výstupu.
- 10) Napište program, který vypisuje přirozená čísla od 1. Po každých deseti číslech vypíše dotaz:  
Mam pokračovat ? [A/N] :  
a podle typu odpovědi buď pokračuje nebo skončí.
- 11) V souboru `CISLA.TXT` je na každé řádce jedno celé číslo, kterému předchází neznámý počet znaků "\$". Sečtěte tato čísla a výsledek vypište na obrazovku.



## 7 Typová konverze

Pod pojmem typová konverze se míní převod proměnné určitého typu na typ jiný, např. `int` na `float`<sup>1</sup>.

Jazyk C rozeznává dva druhy typové konverze:

- *implicitní* neboli samovolná či automatická
- *explicitní* neboli vynucená či požadovaná

### 7.1 Implicitní typová konverze

Implicitní typová konverze má tato základní pravidla :

- 1) Před vykonáním operace se samostatné operandy konvertují takto:
  - Kdykoliv se objeví typ `char` nebo `short int`, konvertuje se na typ `int`.
  - Všechny operandy `unsigned char` a `unsigned short` se konvertují na `int` pouze tehdy, když typ `int` může reprezentovat jejich hodnotu ("nepřeteče"). Jinak se konvertují na `unsigned int`.
- 2) Mají-li dva operandy jedné operace různý typ, pak je typ operandu s nižší prioritou konvertován na typ s prioritou vyšší, podle následující hierarchie (kde `int` má nejnižší prioritu):

|                            |                              |
|----------------------------|------------------------------|
| <code>int</code>           | ⇒ <code>unsigned int</code>  |
| <code>unsigned int</code>  | ⇒ <code>long</code>          |
| <code>long</code>          | ⇒ <code>unsigned long</code> |
| <code>unsigned long</code> | ⇒ <code>float</code>         |
| <code>float</code>         | ⇒ <code>double</code>        |
| <code>double</code>        | ⇒ <code>long double</code>   |

Např. je-li první operand typu `float` a druhý typu nižšího, zkonvertuje se druhý operand na typ `float`.

<sup>1</sup> Tyto "triky" pravděpodobně nebudeme ve skutečnosti dlouho potřebovat. Protože však některé konverze probíhají automaticky, je nutné alespoň vědět, že něco takového existuje a při zvláštním chování programu začít zkoumat, zda to nemůže být způsobeno právě typovou konverzí.

- 3) V přiřazovacích výrazech je typ na pravé straně konvertován na typ z levé strany, což je také typ výsledku.

Poznámka:

- Z výše uvedených pravidel tedy vyplývá, že se typ `float` nemusí nutně implicitně konvertovat na typ `double`, jak tomu bylo u K&R jazyka C.

Příklady:

- 1) Je-li definice: `char c;` pak:
  - `c = 1;` 1 je konvertována na `char`, tj. `c` obsahuje znak "Ctrl A" (jako v Pascalu `c := CHR(1);`)
  - `c++;` v `c` bude znak "Ctrl B"
  - `c = c + '1';` je jako `c := ORD(c) + ORD('1')`
- 2) Je-li definice: `int i;` pak:
  - `i = 'A';` je jako `i := ORD('A');`
  - `i = 'A' + 2;` je jako `i := ORD('A') + 2;`
  - `i = 3.8;` i bude 3 (0.8 se odřízne)
- 3) Je-li definice: `double g;` pak:
  - `g = 5;` g bude 5.0
- 4) `i = g * c;`  
Nejprve se zkonvertuje `c` na `int` (pravidlo 1), pak se `c` zkonvertuje na `double` (pravidlo 2). Výsledek výrazu `g * c` je `double`, ale podle pravidla 3) se zkonvertuje na `int` a přiřadí do `i`.

### 7.2 Explicitní typová konverze

Narozdíl od implicitní konverze, kterou nejsme schopni v podstatě ovlivnit, můžeme explicitní konverzi využívat téměř podle libosti, ovšem s tím rizikem, že nevhodné použití způsobí značné problémy.

Explicitní konverze se též nazývá přetypování (*casting* nebo *typecasting*) a má formu: `(typ) výraz` která znamená, že *výraz* (nebo proměnná) je v čase překladu konvertován na požadovaný typ.

Příklady a význam často používaných konverzí:

|                                   |                                                 |
|-----------------------------------|-------------------------------------------------|
| <code>(int) char_vyraz</code>     | – převod znaku na ordinální číslo               |
| <code>(char) int_vyraz</code>     | – převod ordinálního čísla na odpovídající znak |
| <code>(int) float_vyraz</code>    | – odříznutí desetinné části                     |
| <code>(double) int_vyraz</code>   | – převod celého čísla na reálné                 |
| <code>(double) float_vyraz</code> | – zvětšení přesnosti                            |



Explicitní přetypování je nutné v mnoha případech, nejčastěji při používání pointerů (viz např. str. 168). Ve verzi K&R jazyka C se většinou uvádí "klasický" případ, kdy bylo přetypování nutné:

```
int i = 10;
double f;
f = sqrt((double) i);
```

Bez přetypování byly výsledky chybné, protože funkce pro výpočet odmocniny (`sqrt()`) potřebuje jako parametr typ `double` a v K&R verzi jazyka C se nedělala automatická (implicitní) konverze typu `int` na typ `double`.

Tento problém sice v ANSI C odpadá, ale v podobných případech je vhodné explicitní přetypování<sup>2</sup> uvést, protože zprůhledňuje programátorovy úmysly a je ihned jasné, že je to chtěná změna. Někdy je explicitní přetypování nutné, abychom se zbavili varovných hlášení o nevhodném typu parametru.

#### Pozor:

Přetypování není l-hodnota, takže např.: `(int) f = 3;` je chybné.

<sup>2</sup> Například: `sqrt((double) i)` viz též str. 118.

## 8 Preprocesor jazyka C

Preprocesor je něco, co Pascal nezná. My jsme zatím využívali víceméně nevědomky nejčastěji používaný příkaz preprocesoru `#include`. Preprocesor má ovšem navíc řadu jiných možností, které dávají jazyku C další "sílu". Je vhodné se s těmito příkazy seznámit, protože, i když je nebudeme hned využívat úplně všechny, je dobré alespoň vědět, že něco takového existuje.

Činnost preprocesoru se dá shrnout do několika základních bodů:

- Zpracovává zdrojový text programu **před** použitím překladače.
- Nekontroluje syntaktickou správnost programu.
- Provádí pouze záměnu textů, např. identifikátorů konstant za odpovídající číselné hodnoty<sup>1</sup>.
- Vypustí ze zdrojového textu všechny komentáře.
- Provádí podmíněný překlad.

#### Poznámka:

- Řádka, která je určena pro zpracování preprocesorem musí začínat znakem `"#"` (*pound sign*). Znak `"#"` by měl být jako první znak na řádce a za ním by neměla být mezera<sup>2</sup>.

Seznam konstrukcí, které rozeznává preprocesor C:

- definování makra  
`#define jméno_makra libovolny text rozvoje`
- zrušení definice makra  
`#undef jméno_makra`
- podmíněný překlad textu v závislosti na hodnotě **konst\_výraz**  
`#if konst_výraz`  
`#elif #else #endif`
- vložení textu ze specifikovaného souboru v adresáři uživatele  
`#include "filename"`
- vložení textu ze specifikovaného souboru ze systémového adresáře  
`#include <filename>`

<sup>1</sup> Tato činnost se někdy nazývá zpracování maker (*macro processing*).

<sup>2</sup> V ANSI C to již není podmínka, ale je to lepší dodržovat.



- podmíněný překlad textu v závislosti na tom, zda je makro jméno\_makra definováno či undefinováno  

```
#ifdef jméno_makra
...
#else
...
#endif
```
- podmíněný překlad textu v závislosti na tom, zda je makro jméno\_makra undefinováno či definováno  

```
#ifndef jméno_makra
...
#else
...
#endif
```
- výpis chybových zpráv během preprocesingu  

```
#error Chybova zprava
```

**Poznámka:**

- Kromě výše uvedených konstrukcí, které se v praxi často využívají, rozpoznává preprocesor ještě:
  - direktivu `#line`
  - operátory `#` a `##`
  - operátor `defined`
  - předdefinovaná standardní makra: `__LINE__`, `__FILE__`, `__TIME__`, `__DATE__` a `__STDC__`
  - direktivu `#pragma`

Tyto konstrukce se nepoužívají příliš často a proto budou v následujících příkladech použity jen některé z nich. Zájemce o podrobnější informace odkazujeme např. na [HRŠ92].

## 8.1 Makra bez parametrů — příkaz define

Makra bez parametrů, známější pod názvem *symbolické konstanty*<sup>3</sup>, se využívají velmi často, protože zbavují program “magických čísel”, tj. nejrušnějších konstant, které se bez vysvětlení objevují v programu.

Většinou jsou konstanty definovány na začátku programu (modulu). Jejich rozumné použití také významně zvyšuje modularitu programu.

Náhrada konstanty skutečnou hodnotou se nazývá rozvojem (expanzí) makra nebo též substitucí makra.

Pro psaní symbolických konstant platí následující pravidla:

- Jména konstant jsou z konvence psána vždy VELKÝMI PÍSMENY.
- Jméno konstanty je od její hodnoty odděleno alespoň jednou mezerou.
- Za hodnotou může a měl by být komentář.

<sup>3</sup> Dále bude používán též výraz *konstanta* ve smyslu symbolická konstanta.

- Nové konstanty mohou využívat již dříve definovaných konstant.
- Pokud je hodnota konstanty delší než řádka, musí být na konci řádky znak “\”, který se ale do makra nerozvine — je to pouze pomocný znak.

**Příklady:**

```
#define MAX 1000 /* max. rozmer pole */
#define PI 3.14
#define DVE_PI (2 * PI)
#define MOD %
#define AND &&
#define JMENO_SOUBORU "DOPIS.TXT"
#define DLOUHA_KONSTANTA Toto je dlouha konstanta, ktera se\
 nevejde na jednu radku.
```

**Poznámky:**

- Za hodnotou není (v 99%) středník.
- Mezi jménem konstanty a hodnotou není znak `=`, jako je to u Pascalu.
- Konstanta se může objevit kdekoli v programu s jedinou výjimkou — neměla by být součástí řetězce (mezi uvozovkami), protože tam k rozvoji konstanty nedojde — viz dále str. 92.
- Konstanta začíná platit od místa definice a platí až do konce souboru, ve kterém byla definována.

**Příklady:**

- 1) Použití symbolické konstanty pro Ludolfovo číslo.

```
#include <stdio.h>
#define DVE_PI (2 * 3.14)
main()
{
 double r;

 printf("Zadej polomer : ");
 scanf("%lf", &r);
 printf("Obvod kruhu s polomerem %f je %f\n",
 r, r * DVE_PI);
}
```

- 2) Program přečte řádku textu a každé malé písmeno zobrazí jako velké, ale vytiskne před ním znak “#”, tedy např. “b” bude “#B”.

```
#include <stdio.h>
#define POSUN ('a' - 'A')
#define EOLN '\n'
#define PRED_MALE '#'
```



```

main()
{
 int c;

 while ((c = getchar()) != EOLN) {
 if (c >= 'a' && c <= 'z') {
 putchar(PRED_MALE);
 putchar(c - POSUN);
 }
 else
 putchar(c);
 }
}

```

**Poznámky:**

- Budeme-li někdy v budoucnu chtít tisknout místo znaku “#” znak “\*”, stačí pak jen jedna změna na začátku programu.
- Z definice symbolické konstanty POSUN je vidět, že symbolickou konstantou může být i výraz. Pak je ale velmi vhodné ho uzavřít do závorek. Kdyby v našem případě závorky chyběly, měl by příkaz:  
`putchar(c - POSUN);`  
 podstatně jiný význam.

**Pozor:**

Makro se nerozvine, je-li uzavřeno v uvozovkách, např.:

|                                          |                                      |
|------------------------------------------|--------------------------------------|
| <code>#define JMENO Katka</code>         | řešením může být např.:              |
| <code>printf("Jmenuji se JMENO");</code> | <code>#define JMENO "Katka"</code>   |
| <code>vytiskne: Jmenuji se JMENO</code>  | <code>printf("Jmenuji se %s",</code> |
| <code>a ne: Jmenuji se Katka</code>      | <code>JMENO);</code>                 |

Nová definice makra nejčastěji překrývá starou jen tehdy, pokud je to stejná definice. To většinou nebývá splněno, protože definovat znovu stejnou konstantu stejně pojmenovanou nemá příliš smysl. V tom případě je nutné napřed starou definici zrušit použitím direktivy `#undef`, např.:

```

#define POCET 10 /* stara definice POCET */
#undef POCET /* POCET pozbyl platnost */
#define POCET 20 /* nova definice POCET */

```

Občas se makro užívá jako skrytá část programu, např.:

```
#define ERROR { printf("Chyba v datech \n"); }
```

Při použití nesmí být toto makro ukončeno středníkem:

```

if (x == 0)
 ERROR /* zde není středník */
else
 y = y / x;

```

## 8.2 Makra s parametry

Při řešení programů se často vyskytne případ, kdy mnohokrát používáme nějakou funkci, která je velmi krátká, např. provádí jednoduchý výpočet nějaké hodnoty. Takovou funkci lze v samozřejmě napsat bez problémů, ale pak někdy nastává menší problém s efektivitou programu. Je-li totiž funkce velmi krátká, je někdy její “administrativa”, tj. předání parametrů, úschova návratové adresy, skok do funkce, návrat z funkce do místa volání a výběr použitých parametrů, delší než samotný užitečný kód funkce. Tato administrativa samozřejmě zdržuje výpočet programu. Naštěstí jazyk C dává možnost, jak tuto administrativu zcela potlačit. Samozřejmě nic není zadarmo, takže při použití dále uvedeného způsobu — maker s parametry — se zvětší délka programu. Na programátorovi, který tímto způsobem *optimalizuje* svůj program tedy je, aby vybral menší ze dvou zel — buď bude program kratší, ale pomalejší, nebo bude delší, ale rychlejší.

Makra s parametry se též někdy nazývají vkládané funkce (*in-line functions*), protože, na rozdíl od skutečných funkcí, se makra s parametry nevolají, ale před překladem nahradí preprocesor jméno makra konkrétním textem.

Praktické použití tedy je jen pro velmi krátké akce, kdy by administrativa funkce trvala srovnatelnou dobu s vlastním výpočtem funkce.

Syntaxe makra s parametry je následující:

```
#define jméno_makra(arg1, ..., argN) hodnota_makra
```

**Pozor:**

Mezi *jméno\_makra* a otevírací kulatou závorkou “(” nesmí být mezera! Argumenty by pak byly považovány za hodnotu makra.

**Štábní kultura:**

- Narozdíl od maker bez parametrů (symbolických konstant), jejichž jména se píší velkými písmeny) se jména maker s parametry píší malými písmeny, stejně jako jména funkcí.

Syntaxe volání makra pak je: `jméno_makra(par1, ..., parN)`

**Příklad:**

Makro pro test, zda je znak velké písmeno:



```
#define je_velke(c) ((c) >= 'A' && (c) <= 'Z')
```

je voláno ve zdrojovém souboru např. pro převod velkých písmen na malá jako:

```
ch = je_velke(ch) ? ch + ('a' - 'A') : ch;
```

a po zpracování zdrojového souboru preprocesorem se rozvine jako<sup>4</sup>:

```
ch = ((ch) >= 'A' && (ch) <= 'Z') ? ch + ('a' - 'A') : ch;
```

#### Poznámky:

- Všimněte si, že argument *c* v definici makra je uzavřen do závorek. Neuděláme-li to, je velká šance pro vznik chyb, např.:

```
definice: #define sqr(x) x * x
```

se po volání: `sqr(f + g)`

rozvine do: `f + g * f + g`

Správně má být definice: `#define sqr(x) ((x) * (x))`

která se rozvine do: `((f + g) * (f + g))`

- Je dobré vždy uvádět i vnější závorky, protože např.:

```
#define cti(c) c = getchar()
```

se po volání: `if (cti(c) == 'a')`

rozvine do známé chyby: `if (c = getchar() == 'a')`

#### Pozor:

Objeví-li se argument v hodnotě makra vícekrát, pak by makro nemělo být voláno s aktuálním parametrem, který může mít vedlejší účinek, např.:

```
#define cislice(x) ((x) >= '0' && (x) <= '9')
```

po volání: `if (cislice(c++))`

způsobí, že proměnná *c* bude inkrementována dvakrát, což zřejmě není správné.

#### 8.2.1 Předdefinovaná makra

Vrátíme-li se na str. 66, vidíme, že se např. funkce pro čtení ze souboru a čtení z klávesnice nápadně podobaly. Ve skutečnosti je tato služba zajišťována pouze jednou funkcí, která se díky předdefinovanému makru může používat dvěma způsoby.

Makra pro zjednodušení vstupu a výstupu na terminál jsou většinou definovaná v souboru `stdio.h` a mají tuto podobu:

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

<sup>4</sup> Nutno ovšem poznamenat, že nás tento rozvoj makra v naprosté většině případů nezajímá a ani jej nevidíme, protože se děje v rámci kompilace.

Další soubor, ve kterém je uvedeno množství užitečných maker, je soubor `ctype.h`. Makra v něm definovaná pracují se znaky a dělí se do dvou skupin:

- 1) Makra pro určení typu znaku.

Tato makra začínají písmeny *is*, např. `isdigit(c)`, které vrací znak v *c* (nenulovou hodnotu), je-li v *c* znak číslice, a jinak vrací nulu (`FALSE`).

Seznam maker:

| vrací | jméno                 | rozsah použití                                     |
|-------|-----------------------|----------------------------------------------------|
| znak  | <code>isalnum</code>  | čísllice a malá a velká písmena                    |
| znak  | <code>isalpha</code>  | malá a velká písmena                               |
| 1     | <code>isascii</code>  | ASCII znaky (0 až 127)                             |
| znak  | <code>isctrl</code>   | Ctrl znaky (1 až 26)                               |
| znak  | <code>isdigit</code>  | čísllice                                           |
| znak  | <code>islower</code>  | malá písmena                                       |
| 1     | <code>isprint</code>  | tisknutelné znaky (32 až 126)                      |
| znak  | <code>ispunct</code>  | interpunkční znaky (tečka, čárka, lomítko, ...)    |
| znak  | <code>isspace</code>  | bílé znaky (mezera, tabulátor, nový řádek, ...)    |
| znak  | <code>isupper</code>  | velká písmena                                      |
| znak  | <code>isxdigit</code> | hexadec. číslice ('0' - '9', 'A' - 'F', 'a' - 'f') |
| 1     | <code>isgraph</code>  | znak s grafickou podobou (33 až 126)               |

- 2) Makra pro konverzi znaku.

Tato makra začínají písmeny *to*, např.: `b = tolower(c)`; které konvertuje velké písmeno v *c* na malé; ostatní znaky ponechá nezměněny a parametr *c* ponechá také nezměněn.

Seznam maker:

|                      |                                                          |
|----------------------|----------------------------------------------------------|
| <code>tolower</code> | - konverze na malá písmena                               |
| <code>toupper</code> | - konverze na velká písmena                              |
| <code>toascii</code> | - převod na ASCII — jen nejnižších 7 bitů je významových |

#### Poznámky:

- Chceme-li používat výše uvedená makra, je nutné na začátku programu použít kromě příkazu: `#include <stdio.h>` též příkaz: `#include <ctype.h>`
- U maker s parametry nelze — n rozdíl od funkcí — použít rekurze! ANSI C preprocesor zabrání případné nekonečné rekurzi maker tím, že potlačí náhradu jména makra v jeho vlastní definici.



### 8.3 Vkládání souborů — příkaz include

S tímto příkazem preprocesoru jsme se setkali již dávno ve známém příkazu: `#include <stdio.h>`

Skutečně se vkládání souborů (*file inclusion*) používá v C programech velmi často — v podstatě vždy. Nejčastěji používaný příkaz pro preprocesor je tedy právě:

`#include "jmeno_souboru"` nebo `#include <jmeno_souboru>`  
který způsobí, že zdrojový soubor `jmeno_souboru` bude "vtažen" ("vlepen", "inkludován") do "volajícího" souboru<sup>5</sup> do místa, kde se v něm nachází příkaz `#include`

V souboru, který je vtahován, mohou být další příkazy `#include`, ale obecně se to nedoporučuje a je nutné dávat velký pozor na zacyklení.

Jak již bylo uvedeno, příkaz `#include` má dva tvary, které určují, kde se má specifikovaný soubor hledat:

- 1) Příkaz: `#include "KONSTANTY.H"`  
hledá soubor `KONSTANTY.H` ve stejném adresáři, ve kterém leží "volající" soubor. Nenajde-li ho tam, je možné, že ho bude dále hledat v dalších adresářích, což však už závisí na konkrétní implementaci. Používá se pro práci se soubory, které jsme vytvořili my sami.
- 2) Příkaz: `#include <ctype.h>`  
hledá soubor `ctype.h` v systémovém adresáři. Používá se pro práci s již hotovými speciálními soubory, kterým se říká *standardní hlavičkové soubory* — viz str. 98.

#### Poznámka:

- Někteří programátoři používají konvenci, že jména standardních hlavičkových souborů píší malými písmeny, kdežto jména hlavičkových souborů, které vytvořili, písmeny velkými.

#### 8.3.1 Vkládané soubory

Soubory, které se dají vkládat pomocí příkazu `#include`, mohou být libovolné textové soubory. Význam má samozřejmě vkládání jen zdrojových souborů — s příponou `.C`. Druhá — a mnohem častější — možnost, je vkládání tzv. *hlavičkových souborů* (*header files*) — s příponou `.H`.

<sup>5</sup> Tedy souboru, který obsahuje příkaz `#include`.

Vkládání hlavičkových souborů je velmi užitečný mechanismus, jak program sestávající se z více souborů udržet čitelný. Například všechny definice konstant využívané více soubory se uvedou pouze jednou do souboru typu `.H`, který se pomocí `#include` připojí do všech souborů, které tyto definice konstant potřebují. To má obrovskou výhodu, že případná změna konstant se pak provede pouze v jednom souboru `.H` a ostatní soubory, využívající tyto konstanty, stačí pouze přeložit<sup>6</sup>.

#### Příklad:

Definice konstant popisující poměry na obrazovce jsou uloženy v tomto souboru:

```
/*
 * OBRAZ.H
 *
 * Konstanty obrazovky
 *
 */

#ifndef OBRAZ
#define OBRAZ

#define RADKY_OBRAZOVKY 25
#define SLOUPCE_OBRAZOVKY 80

#endif
/* konec souboru OBRAZ.H */
```

Ve všech souborech, které potřebují pracovat s obrazovkou, se pak použije příkaz: `#include "OBRAZ.H"`

#### Poznámka:

- Zde jsme trochu předběhli, protože příkazy `#ifndef` a `#endif` budou vysvětleny až za chvíli. Ovšem výše popsáný způsob je velmi užitečný "trik", jak zabránit několikanásobnému natažení souboru `OBRAZ.H` do jednoho modulu. Tento omyl se stane velmi lehce, zvláště, jsou-li soubory s příkazy `#include` do sebe zanořeny. Pokud máme hlavičkové soubory vytvářené tímto způsobem, nic se nemůže stát ani při několikanásobném natažení souboru `OBRAZ.H` do jednoho souboru. Při prvním natažení se totiž definuje symbolická konstanta `OBRAZ` a ta ve spolupráci s příkazem `#ifndef` zabrání opětovnému natažení obsahu souboru — viz též str. 103.

<sup>6</sup> Podrobný příklad složitějšího programu je uveden na str. 136.



### 8.3.2 Standardní hlavičkové soubory

S představitelem těchto souborů — souborem `stdio.h` — jsme se již mnohokrát setkali. Ovšem tento soubor není sám, protože každá implementace překladače C má několik desítek standardních `.h` souborů a v každém z těchto souborů je popsána část funkcí a konstanty ze standardní knihovny. To, co tyto soubory popisují<sup>7</sup>, je definováno ANSI normou, což má tu ohromnou výhodu, že program v C, který pomocí těchto souborů využívá pouze standardní knihovnu, by měl být v podstatě 100% přenositelný mezi nejrozličnějšími počítači a nejrozličnějšími operačními systémy.

Soubor `stdio.h` obsahuje kromě definic základních vstupních a výstupních funkcí také definice rozličných konstant a typů např. `getc()`, `EOF`, `NULL`, `FILE`, atd.

V `.h` souborech nejsou uvedeny celé zdrojové texty příslušných funkcí, ale pouze jejich hlavičky (tzv. funkční prototypy — viz str. 115), tj. popisy, jaké má ta která funkce parametry a jaký typ hodnoty vrací. S využitím těchto znalostí pak může překladač zjistit, že je např. knihovní funkce pro vstup znaku z klávesnice volaná v programu jako: `getchar(c)`;<sup>8</sup> použita chybně. V souboru `stdio.h` je totiž její funkční prototyp uveden bez parametru jako: `int getchar()`.

Důležité je, že všechny standardní `.h` soubory jsou normální textové soubory<sup>9</sup>, které je možno si prohlédnout libovolným editorem.

Z mnoha dalších standardních `.h` souborů se zmíníme pouze o dalších třech často využívaných. Jsou to již známý `ctype.h` pro práci se znaky a `math.h`, který je využíván pro práci s matematickými funkcemi jako jsou `sin`, `cos` atd. Dalším je soubor `time.h`, kterému věnujeme trochu větší pozornost.

### 8.3.3 Soubor `time.h` — měření času

Hlavičkový soubor `time.h` obsahuje popisy mnoha užitečných funkcí pro práci s časem. Jako příklad uvedeme použití funkce `clock()`, která vrací počet "tiků" procesoru od začátku spuštění programu — je tedy užitečná pro zjištění, jak dlouho program — nebo jeho část — běžel. Abychom tento údaj převedli na srozumitelnější počet sekund, je třeba získané číslo dělit konstantou `CLOCKS_PER_SEC`, která je rovněž v `time.h` definována. K příkladu

<sup>7</sup> Různé implementace tedy mohou mít různé obsahy standardních `.h` souborů, ale navenek se musí chovat všechny stejně. Tak např. symbolická konstanta `EOF` musí být vždy definována, ale to, jakou má hodnotu, záleží na implementaci.

<sup>8</sup> Místo: `c = getchar()`;

<sup>9</sup> Jsou uloženy nejčastěji v adresáři `\INCLUDE`

dodejme ještě, že funkce z `time.h` často využívají místo typu `long` své definované synonymum `clock_t`, a pak také, že konstanta `CLOCKS_PER_SEC` se občas jmenuje jen `CLK_TCK` a že je velmi vhodné ji přetypovat na typ `double`.

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

```
main()
{
 clock_t start, end; /* long start, end; */
 unsigned int i;

 start = clock();

 for (i = 0; i < 1000; i++)
 printf("*");

 end = clock();
 printf("\nProgram trval: %.2f sec\n",
 (end - start) / (double) CLOCKS_PER_SEC);
}
```

## 8.4 Oddělený překlad souborů — I.

Vkládání souborů by nemělo být zaměňováno s odděleným překladem souborů. Je-li program dělen do více souborů, které se pomocí `#include` vloží do jednoho souboru, vznikne po překladu pouze jeden `.OBJ` soubor.

Oddělený překlad souborů znamená, že se každý soubor přeloží zvlášť — vznikne tedy několik `.OBJ` souborů, a ty se spojí do jednoho programu až pomocí sestavovacího programu (*linkeru*). Na první pohled to vypadá jako zbytečné zesložení, ale v praxi je to jediný způsob, jak rozumně zvládnout práci s velkými programy.

Moderní kompilátory tento způsob rovněž podporují<sup>10</sup>. Čím je totiž větší překládaný soubor, tím více narůstají kompilátoru požadavky na různé administrativní informace, např. tabulky identifikátorů, atd. To může vést až ke stavu, kdy se při překladu obsadí veškerá operační paměť a překlad se v horším případě neúspěšně ukončí a v lepším případě extrémně zpomalí.

<sup>10</sup> Například příkaz `Project` u překladače fy Borland.

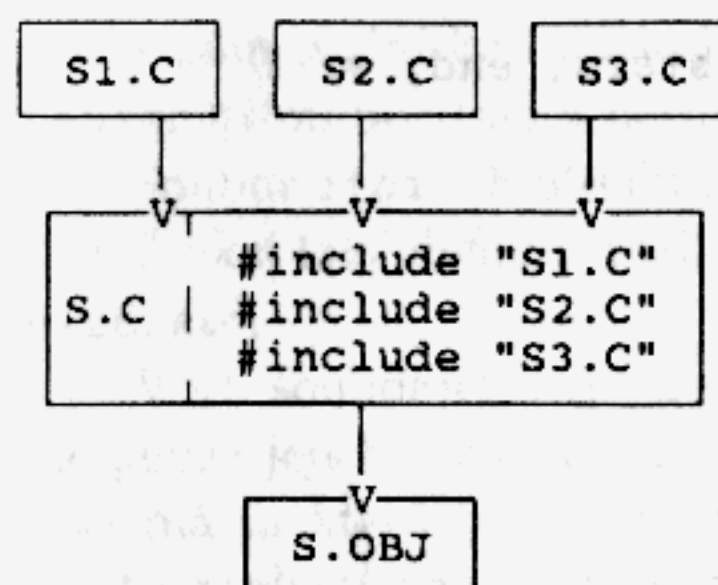


Zpomalení je způsobeno tzv. *swapováním*, čili stavem, kdy je procesor nucen odkládat momentálně nepotřebné úseky operační paměti na disk.

#### Příklad:

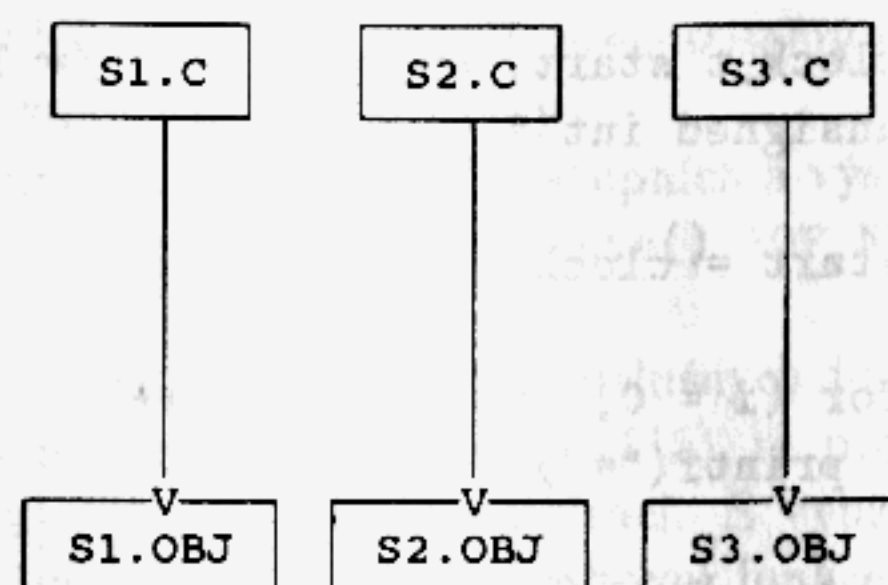
Mějme tři soubory S1.C, S2.C a S3.C, kde funkce main() je v souboru S1.C.

##### 1) Vkládání souborů



sestavení: link s

##### 2) Oddělený překlad



sestavení: link s1, s2, s3

#### Poznámky:

- Při vkládání souborů se všechny tři soubory "inkludují" do souboru S.C<sup>11</sup>, který se přeloží a vznikne jeden .OBJ soubor a ten se samostatně sestaví ("slinkuje").

Nevýhodou tohoto způsobu je, že při změně v kterémkoliv ze souborů S1.C, S2.C nebo S3.C se musí překládat zcela zbytečně i oba dva zbývající, což může být někdy velmi časově náročné.

Další nevýhodou je, že nelze skrýt mezi jednotlivými soubory jejich globální identifikátory<sup>12</sup>.

- Při odděleném překladu se každý soubor samostatně přeloží do .OBJ souboru. Výhodou pak je, že se při opakovaném překladu (nejčastěji při ladění) překládá pouze ten soubor, který byl skutečně měněn. Tento způsob překladu se doporučuje používat, protože nás explicitně nutí rozdělit problém do více menších částí. To sice zpočátku zabere určité množství času, ale při ladění velkého programu se to bohatě vyplatí.

<sup>11</sup> Ten může obsahovat pouze tři příkazy #include a nic víc.

<sup>12</sup> Viz dále paměťovou třídu static na str. 124.

## 8.5 Podmíněný překlad

V mnoha případech se složitější programy píšou tak, že obsahují ladicí části. To jsou nejčastěji pomocné výpisy, které mají usnadnit ladění, ale mohou to být např. i funkce, které hlídají meze polí atd. Tyto části se do programu dávají, i když máme k dispozici výkonný debugger<sup>13</sup>.

Je dobrým zvykem počítat již při návrhu programu s tím, že jej bude nutno ladit a už při vytváření programu tyto části do programu zařazovat. Obecné i konkrétnější doporučení jsou uvedeny např. v [AA88].

Po odladění programu však nastává typický problém — jak tyto ladicí části, které vypisují již nepotřebné (a tedy nevhodné) informace a zdržují provádění programu, z odladěného programu odstranit. Nejjednodušším řešením je samozřejmě pomocí editoru projít celý program a ladicí části jednoduše vymazat. Pokud jste to již zkoušeli, víte, že toto řešení s sebou přináší některá úskalí. Například že vymažeme v "mazací euforii" i to, co jsme vymazat neměli nebo že ladicí části byly nějakým způsobem potřebné i pro skutečný program — nejčastěji se jedná o sdílené proměnné definované v ladicí části a využívané i v programu. Po tomto výmazu tedy většinou překládáme již odladěný program s napjatým očekáváním, zda bude chodit. Bohužel často nechodí a my musíme ladit znovu a v nejhorším případě musíme postupně a postupně do programu dopisovat stejné ladicí části, které jsme předtím vymazali.

Jazyk C naštěstí pamatuje i na tento problém, takže pomocí příkazu preprocesoru můžeme určit, které části programu se mají překládat podmíněně. To znamená, že všechny ladicí části již při vytváření programu označíme jako podmíněně překládané a při ladění je překládáme a po odladění je nepřekládáme. Ladicí části jsou tak trvalou součástí zdrojového souboru, ale volitelnou součástí programu<sup>14</sup>. Preprocesor pak na náš jeden příkaz všechny tyto části vypustí sám, ale budou-li někdy v budoucnu<sup>15</sup> opět potřebné, pak je jedním příkazem opět do programu zařadíme.

Podmíněný překlad je řízen dvěma základními podmínkami<sup>16</sup>:

<sup>13</sup> Například Code View od fy Microsoft nebo když ladíme v prostředí v již vestavěném debuggerem — Turbo C nebo Borland C.

<sup>14</sup> Vymazání na věčné časy to tedy není.

<sup>15</sup> A to je velmi často!

<sup>16</sup> Ve skutečnosti až čtyřmi — viz dále.



### 8.5.1 Řízení překladu hodnotou konstantního výrazu

Podmíněný překlad řízený konstantním výrazem, což může být číslo, symbolická konstanta nebo i podmíněný výraz z těchto možností, má tuto syntaxi:

```
#if konstantní_výraz
 část_1
#else
 část_2
#endif
```

Překladač bude tuto část programu zpracovávat tak, že je-li hodnota *konstantní\_výraz* rovna 0 (*FALSE*), překládá se pouze *část\_2* a v opačném případě (nenulová hodnota — *TRUE*) se překládá pouze *část\_1*.

#### Poznámky:

- Části *#else* a *část\_2* mohou být vynechány.
- Existuje jednoduchá užitečná aplikace tohoto příkazu. Nechceme-li dočasně překládat část programu, můžeme samozřejmě tuto část uzavřít do komentářových závorek */\* \*/*. Tento jednoduchý způsob má ale jeden háček, a to, že komentáře nesmí být vhnížďené, čili v takto uzavíraném textu nesmí být žádný jiný komentář! Je-li tam komentář, pomůžeme si lehce tímto trikem:

```
#if 0
 část programu, která má být vynechána
#endif
```

- Často se podmíněný překlad používá při vývoji programů, které jsou sice závislé na konkrétním počítači, ale měly by po minimálních změnách fungovat i na počítačích jiných, tedy např.:

```
#if PCAT
 #include <conio.h> /* consol input/output */
#else
 #include <stdio.h> /* standard input/output */
#endif
```

Budeme-li používat program na PC/AT stačí předřadit příkaz:

```
#define PCAT 1
```

Na ostatních počítačích pak příkaz změníme na:

```
#define PCAT 0
```

- V praxi je možné ještě rozšířit působnost tohoto příkazu využitím direktiv *#elif* a *#error* a operátoru *defined* — viz str. 104.

### 8.5.2 Řízení překladu definicí makra

Podmíněný překlad řízený hodnotou konstantního výrazu — viz předchozí odstavec — je mocný nástroj, ale mnohem častěji se pro řízení podmíněného překladu používá jeho jednodušší verze, která je závislá pouze na tom, zda byla určitá symbolická konstanta definována či ne<sup>17</sup>. Nebudeme zde uvádět syntaxi, která je velmi podobná syntaxi předchozí a uvedeme hned modifikovaný příklad použití z předchozího odstavce.

```
#ifndef PCAT
 #include <conio.h> /* consol input/output */
#else
 #include <stdio.h> /* standard input/output */
#endif
```

Což vypadá velmi podobně a má naprosto stejnou funkci, jako v předchozím případě. Hlavní změna je v tom, že budeme-li používat program na PC/AT stačí příkaz:

```
#define PCAT /* prazdny, ale definovany */
```

Na ostatních počítačích pak příkaz změníme na:

```
#undef PCAT /* zrusena definice makra PCAT */
```

nebo jednodušeji symbolickou konstantu *PCAT* vůbec nedefinujeme.

#### Poznámky:

- Příkazy *#else* a *#include <stdio.h>* mohou být opět vynechány<sup>18</sup>
- K dispozici je ještě jeden příkaz, který je pouze negací předchozího, čili je řízen tím, že symbolická konstanta *nebyla* definována. Opět nebudeme uvádět syntaxi příkazu, ale rovnou modifikovaný předchozí příklad.

```
#ifnndef PCAT
 #include <stdio.h> /* standard input/output */
#else
 #include <conio.h> /* consol input/output */
#endif
```

Po příkazu: *#undef PCAT* se provede příkaz: *#include <stdio.h>*

Po příkazu: *#define PCAT* pak příkaz: *#include <conio.h>*

<sup>17</sup> Na její hodnotě pak vůbec nezáleží.

<sup>18</sup> Což ale v tomto konkrétním případě nemá valný smysl.



## 8.5.3 Operátor defined

Test existence definice symbolické konstanty pomocí direktiv `#ifdef` nebo `#ifndef` umožňuje zjišťovat existenci pouze jednoho symbolu (jména konstanty). Aby bylo možné vytvářet při podmíněné kompilaci logické výrazy z více symbolů, je nutné použít operátor `defined`. Způsob použití operátoru `defined` ukazují následující tři ekvivalentní příkazy:

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>pro direktivu #ifdef</i>    | <i>pro direktivu #ifndef</i>    |
| <code>#ifdef TEST</code>       | <code>#ifndef TEST</code>       |
| <code>#if defined TEST</code>  | <code>#if !defined TEST</code>  |
| <code>#if defined(TEST)</code> | <code>#if !defined(TEST)</code> |

Příklad použití operátoru `defined` viz následující odstavec.

8.5.4 Direktivy `#elif` a `#error`

Možnosti všech uvedených typů podmíněné kompilace rozšiřuje direktiva `#elif`, která má analogický význam jako příkaz `else-if` v podmíněném příkazu.

Další direktiva `#error` umožňuje výpis chybových zpráv již během fáze preprocessingu. Libovolný text, který následuje za příkazem `#error`, je vypsán na standardní chybové zařízení (nejčastěji obrazovku) a kompilace je ukončena chybou. Tato direktiva se typicky používá pro kontrolu hodnot symbolických konstant ovlivňujících podmíněnou kompilaci.

Nyní bude uveden příklad<sup>19</sup>, který by měl osvětlit jak použití direktiv `#elif` a `#error`, tak i operátoru `defined`.

```
#if defined(ZAKLADNI) && defined(DEBUG)
 #define VERZE_LADENI 1
#elif defined(STREDNI) && defined(DEBUG)
 #define VERZE_LADENI 2
#elif !defined(DEBUG)
 #error Pozor, ladici verzi není možné připravit!
#else
 #define VERZE_LADENI 3
#endif
```

<sup>19</sup> Všimněte si odsazení jednotlivých `#define`, které zvyšuje přehlednost.

## Časté chyby:

|                                        |                                                           |
|----------------------------------------|-----------------------------------------------------------|
| <code>#define A = 1</code>             | před 1 nemá být =                                         |
| <code>#define PI 3.14;</code>          | za 3.14 nemá být středník                                 |
| <code>#define inc(x) x + 1</code>      | má být <code>((x) + 1)</code>                             |
| <code>#define inc (x) ((x) + 1)</code> | mezi <code>inc</code> a <code>(x)</code> nesmí být mezera |

## Co je dobré si uvědomit:

- Všechny parametry v definici makra s parametry by měly být uzavřeny do závorek.
- Vyhýbejte se možnosti vzniku vedlejších efektů při vyhodnocení parametrů makra.
- Každou použitou konstantu definujte jako symbolickou a to hned na začátku programu — zlepšuje to přenositelnost programu a samozřejmě i jeho čitelnost.
- Použijte makra s parametry pro skrytí dlouhých, opakujících se a komplikovaných výrazů — významně to zlepšuje čitelnost programu.
- Používejte podmíněnou kompilaci pro vynechání ladících částí programu.
- Hlásí-li překladač nějakou chybu, na kterou nemůžete přijít, je vhodné<sup>20</sup> prohlédnout si soubor po zpracování preprocesorem. Je možné, že chyba bude v rozvoji některého makra.

## Cvičení:

- 1) Napište program, který sečte `N` prvních přirozených čísel a vypíše např.:  
Součet prvních 5-ti čísel je 15.  
`N` definujte jako symbolickou konstantu.
- 2) Čtete znaky z klávesnice až do `EOLN`, které si definujete. Po skončení čtení vypíšte počet zadaných číslic. Dále opište jen zadaná písmena, která převedte na velká. Využijte maker `isdigit()`, `isalpha()` a `toupper()` ze souboru `ctype.h`.
- 3) Napište makro `na_treti(x)`, které bude počítat třetí mocninu. Vyzkoušejte ho na výrazech:  
`na_treti(3)`  
`na_treti(i)`  
`na_treti(2 + 3)`  
`na_treti(i * j + 1)`

<sup>20</sup> Samozřejmě, existuje-li vůbec tato možnost.



- 4) Definujte makro `je_velke(c)`, které vrátí 0 není-li znak velké písmeno a 1, je-li to velké písmeno.
- 5) Definujte makro `lze_tisknout(c)`, které zjistí, zda je znak tisknutelný (od ASCII hodnoty 32 do 126). Pomocí něho vytiskněte ASCII tabulku.
- 6) Definujte makro `cti_int(i)`, které čte z klávesnice celé číslo. Makro musí jít použít i ve výrazu, např.:  

```
if ((j = cti_int(k)) == 0)
```

Nápověda: Využijte operátoru čárky.
- 7) Předchozí příklad rozdělte do dvou souborů `MAKRO.C` a `HLAVNI.C` a pomocí příkazu `#include` zajistěte správné chování programu.
- 8) Napište program, který bude číst 10 celých čísel. Číslo bude číst buď ze souboru `CISLA.TXT`, kde je každé číslo na nové řádce, nebo z klávesnice. Při obou způsobech čtení se bude na obrazovku vypisovat např. text:  
Zadej 1. cislo :  
Určete počet sudých čísel.

## 9 Funkce a práce s pamětí

Jazyk C je založený na funkcích. Samozřejmě, že pro kratší programy nám stačí funkce jen jedna — `main()`, kterou již dobře známe, ale takových programů je velmi málo. Většina programů využívá větší či menší počet funkcí<sup>1</sup>.

Předtím, než se ponoříme do tajů funkcí, bude vhodné vymezit některé pojmy, které budou dále používány. Na tomto místě je opět nutné říci, že ti, kteří pouze potřebují vědět, jak funkce vypadá, mohou klidně následující části přeskočit. Přijdou pouze o některé "jemnosti", ke kterým se lze vždy vrátit.

Přehled dále používaných pojmů:

- *Deklarace identifikátoru* — specifikace (dání na vědomí) jména identifikátoru a jeho typu. Žádná alokace paměti pro tento identifikátor se neprovádí.
- *Definice identifikátoru* — kompletní specifikace identifikátoru včetně typu. V tomto místě překladač generuje požadavek na alokaci paměti pro příslušný typ.
- *Pole působnosti identifikátoru* — část programu, ve které je identifikátor definován nebo deklarován (kde je viditelný).
- *Existence identifikátoru* — časový interval, během kterého program pracuje v poli působnosti identifikátoru.
- *Oddělený překlad*<sup>2</sup> — soubor obsahující zdrojový text programu je rozdělen do několika souborů, z nichž každý je překládán samostatně (několik `.OBJ` souborů), ale všechny jsou sestavovány společně sestavovacím programem.
- *Dynamická alokace paměti* — místo v paměti (ve *hromadě* — *heap* nebo v *zásobníku* — *stack*) se alokuje za běhu programu.
- *Statická alokace paměti* — místo v paměti (v *datové oblasti* — *data area*) se alokuje již během překladu, tedy před spuštěním programu.
- *Statická oblast působnosti* — je určena během překladu.

<sup>1</sup> Procedury, tak jak je známe z Pascalu, jazyk C nezná, ale toto omezení lze jednoduše obejít.

<sup>2</sup> Podrobně viz str. 99.



- *Statická globální proměnná* — proměnná, jejíž existence začíná se spuštěním programu a končí s ukončením programu. Je definována vně jakékoliv funkce. Překladač ji uloží do *datové oblasti* paměti.
- *Globální proměnná* — má stejný význam jako statická globální proměnná, pouze nemá uvedeno klíčové slovo *static*.
- *Statická lokální proměnná* — proměnná, jejíž existence začíná při vstupu do funkce, v níž je definována, a končí s ukončením programu, přičemž je dostupná pouze z funkce, kde je definována. Překladač ji uloží do *datové oblasti* paměti.
- *Lokální proměnná* — proměnná, jejíž existence začíná při vstupu do funkce, v níž je definována, a končí s ukončením této funkce<sup>3</sup>. Překladač ji uloží do *stacku*.

## 9.1 Alokace paměti

Každá proměnná musí mít během své existence přidělen paměťový prostor, který velikostí odpovídá typu proměnné. Jméno proměnné (identifikátor) je vlastně symbolická adresa tohoto paměťového prostoru.

Akce, která vyhrazuje paměťový prostor se nazývá *alokace*, přičemž rozeznáváme dva základní typy alokace — *statickou* a *dynamickou*.

### 9.1.1 Statická alokace

Statická alokace se používá mnohem častěji a z Pascalu jsme možná zvyklí jen na ni<sup>4</sup>.

Tento typ alokace se používá tehdy, když umíme překladači předem přesně říci, jaké budeme mít v programu paměťové nároky<sup>5</sup>.

Překladač sám určí požadavky na paměť pro všechny definované proměnné a zavaděč<sup>6</sup> alokuje tuto paměť v čase začátku spuštění programu.

Během provádění našeho programu se neprovádí žádná manipulace s přidělováním paměti. Existence staticky alokovaných proměnných je od začátku programu do jeho konce, čili do odevzdání řízení zpět operačnímu systému, který všechny alokace, provedené před spuštěním našeho programu, najednou zruší.

<sup>3</sup> To je podstatný rozdíl od statické lokální proměnné.

<sup>4</sup> Pascal ale samozřejmě umožňuje použít i alokaci dynamickou.

<sup>5</sup> Například budeme-li v programu číst a sečítat dvě celá čísla, je předem jasné, že potřebujeme dvě proměnné typu *int*.

<sup>6</sup> Systémový program (*loader*), který vlastně spustí náš program.

Statická alokace je sice účinná, ale v některých případech nedostačující. Například při rekurzivním volání funkcí — každé rekurzivní volání funkce totiž vyžaduje nový blok paměti pro své proměnné a překladač není schopen určit, kolikrát bude funkce rekurzivně volána. Nebo potřebujeme-li do paměti načíst obsah celého souboru, nemůžeme samozřejmě vědět v době překladu bude-li to krátký nebo dlouhý soubor<sup>7</sup>.

Statická alokace proměnných vymezuje místo v *datové oblasti*.

Globální proměnné mohou být alokovány pouze staticky.

Není-li z nějakého důvodu možné (např. viz výše uvedený případ rekurze) nebo vhodné (viz výše uvedený příklad načítání souboru do paměti) použít statickou alokaci, pak je možné využít výhod následujících dvou alokací.

### 9.1.2 Dynamická alokace

Při jejím použití vymezujeme paměť na hromadě (*heap*).

Její principem je, že za běhu programu je možné dynamicky přidělit (alokovat) oblast paměti určité délky. Tato paměť není pojmenována (nemá identifikátor — symbolické jméno) a přistupuje se do ní (čili využívá se) pomocí *pointeru*<sup>8</sup>.

### 9.1.3 Vymezení paměti v zásobníku

O tuto alokaci paměti se nemusíme vůbec starat<sup>9</sup>, zajišťuje se sama při volání funkce s využitím informací, které ke kódu této funkce dodal překladač.

Pro naprostou většinu lokálních proměnných (definovaných uvnitř některé funkce) je použit tento druh alokace.

Existence lokálních proměnných začíná při vstupu do funkce a končí při výstupu z této funkce. Pak může být paměť, použitá ve funkci pro tuto proměnnou, využita pro jiné účely, např. pro jinou lokální proměnnou definovanou v jiné, právě probíhající, funkci.

Jestliže je funkce volána znovu, proměnná má při vstupu do funkce ne-definovanou hodnotu. Z toho vyplývá, že proměnná, která je sice potřebná pouze uvnitř funkce, ale musí si ponechávat svoji hodnotu mezi voláními této funkce<sup>10</sup>, nemůže mít paměť alokovanou ve *stacku*.

<sup>7</sup> Nabízí se ovšem myšlenka alokovat staticky tak velké pole, aby stačilo pro uložení libovolného souboru. To je sice možné, ale jednak je to značné plýtvání pamětí a jednak nemusí být tolik paměti ani k dispozici.

<sup>8</sup> Viz dále od str. 166.

<sup>9</sup> Upřímně řečeno, nemáme k tomu ani moc možností.

<sup>10</sup> Statická lokální proměnná.



## 9.2 Funkce

Program v C obsahuje jednu nebo více definicí funkcí, z nichž jedna se musí vždy jmenovat `main()`.

### Poznámka:

- Abychom odlišili jméno proměnné od jména funkce, budeme stále důsledně používat konvenci, že za jménem funkce budou následovat kulaté závorky, tedy `ahoj()` označuje funkci a `ahoj` označuje proměnnou.

Zpracování programu začíná voláním funkce `main()` a končí opuštěním této funkce.

Narozdíl od Pascalu nemohou být funkce vhnížděné — jedna funkce nemůže obsahovat ve svém těle definici druhé funkce. Z toho vyplývá, že formální parametry a lokální proměnné jsou tedy přístupné pouze ve funkci, v níž byly definovány, a jsou skryté z vnějšku této funkce.

Všechny funkce v C jsou skutečné funkce, čili vrací hodnotu. Dají se však použít i jako procedury — možnosti viz str. 112.

### 9.2.1 Definice funkce

Definice funkce určuje jak hlavičku funkce, tak i její tělo, zatímco *deklarace* funkce specifikuje pouze hlavičku funkce, tj. jméno funkce, typ návratové hodnoty a případně i typ a počet jejích parametrů.

### Příklad:

Příklad hlavičky funkce, která zjistí, který ze dvou parametrů je větší:

|                                             |                            |
|---------------------------------------------|----------------------------|
| <b>Pascal</b>                               | <b>C</b>                   |
| <code>FUNCTION max(a, b : INTEGER) :</code> | <code>int max(a, b)</code> |
| <code>INTEGER;</code>                       | <code>int a, b;</code>     |

nebo v novějším stylu podle ANSI: `int max(int a, int b)`

### Poznámky:

- Hlavička funkce není ukončena středníkem.
- Všechny parametry jsou volané hodnotou; neexistuje přímo volání odkazem<sup>11</sup>.
- Mezi jménem funkce a levou závorkou se nedělá mezera — aby nebylo nutno odlišovat volání funkce od volání makra s parametry.

<sup>11</sup> Toto omezení se dá lehce obejít — viz str. 153.

- Typ funkce (její návratová hodnota) může být vynechán<sup>12</sup>. V tom případě bude implicitně typu `int`, např. hlavička funkce `max()` by měla tvar:

```
max(a, b) nebo nověji max(int a, int b)
int a, b;
```

- Novější způsob definice formálních parametrů je doporučován ANSI, protože umožňuje překladači lepší kontrolu skutečných parametrů při volání funkce. Navíc umožňuje urychlit program, protože odpadají implicitní konverze (viz str. 86). Tento způsob bude dále výhradně používán.
- Tělo funkce (program) je uzavřeno do závorek “{” a “}” naprosto stejně, jako u funkce `main()`, a může obsahovat jak příkazy, tak i definice proměnných.
- Narozdíl od Pascalu, kde se výstupní hodnota funkce předává přiřazením jménu funkce, se v C používá příkaz:

```
return (vyraz); nebo return vyraz;
```

který vypočte hodnotu výrazu `vyraz`<sup>13</sup>, přiřadí ji jako návratovou hodnotu funkce a tuto funkci ukončí.

### Příklad:

Kompletní funkce `max()`, která vrátí větší ze svých dvou parametrů.

|                                            |                                         |
|--------------------------------------------|-----------------------------------------|
| <b>Pascal</b>                              | <b>C</b>                                |
| <code>FUNCTION max(a, b : INTEGER):</code> | <code>int max(int a, int b)</code>      |
| <code>INTEGER;</code>                      |                                         |
| <code>BEGIN</code>                         | <code>{</code>                          |
| <code>IF a &gt; b THEN max := a</code>     | <code>return (a &gt; b ? a : b);</code> |
| <code>ELSE max := b</code>                 | <code>}</code>                          |
| <code>END;</code>                          |                                         |

Funkce jsou volány použitím běžné konvence, např.:

```
x = max(10 * i, j - 15);
```

### Poznámka:

- Funkce, která nemá žádné parametry, musí být definována i volána včetně obou kulatých závorek, např. funkce, která přečte dvě celá čísla z klávesnice a vrátí jejich součet:

```
int secti()
{
 int a, b;
```

<sup>12</sup> Ovšem toto “šetření” se nedoporučuje.

<sup>13</sup> Výraz `vyraz` se z konvence uzavírá do kulatých závorek.



```
scanf("%d %d", &a, &b);
return (a + b);
}
```

je volána: `j = secti();`

### 9.2.2 Procedury a datový typ void

Formálně sice v C procedury neexistují, ale jsou dvě cesty, jak to obejít:

- 1) Funkce návratovou hodnotu sice vrací, ale nikdo ji nechce. Typický příklad je čekání na stisk klávesy pomocí funkce `getchar()`, která při normálním použití vrací stisknutý znak

```
getchar(); /* cekani na stisk klavesy */
```

Novější překladače vyžadují v této situaci explicitní přetypování na typ `void` ("prázdný"), aby bylo jasné, že programátor návratovou hodnotu skutečně nepotřebuje, tedy:

```
(void) getchar(); /* cekani na stisk klavesy */
```

- 2) Funkce se definuje, jako funkce vracející typ `void`, např.:

```
void tisk_int(int i)
{
 printf("%d", i);
}
```

volání je pak: `tisk_int(a + b);`

#### Poznámky:

- Příkaz `return` pak není nutný. Pokud není uveden, nahrazuje ho uzavírací závorka funkce `}`. Příkaz `return` se pak používá pouze pro nucené ukončení funkce před dosažením jejího konce, např. po nějaké podmínce.
- Typ `void`<sup>14</sup> se používá i v případě, že funkce nemá žádné formální parametry, aby o tom byl překladač ujistěn, tedy např.:

```
int secti(void)
{
 int a, b;

 scanf("%d %d", &a, &b);
 return (a + b);
}
```

<sup>14</sup> Třetí použití typu `void` viz str. 156.

- Procedura bez parametrů tedy v C vypadá takto:

```
void ahoj(void)
{
 printf("ahoj\n");
}
```

a volá se: `ahoj();`

### 9.2.3 Rekurzivní funkce

Funkce v C mohou být rekurzivní. Rekurzivní funkce vypadají naprosto stejně, jako všechny ostatní, např. program pro výpočet faktoriálu.

```
#include <stdio.h>
int fakt(int n)
{
 return ((n <= 0) ? 1 : n * fakt(n - 1));
}
```

```
main()
{
 int i;

 printf("Zadej cele cislo : ");
 scanf("%d", &i);
 printf("Faktorial je %d\n", fakt(i));
}
```

### 9.2.4 Funkce nevracející int

Jak již určitě tušíme, funkce nemusí vracet jen typ `int`. V tom případě se opět nezmění nic jiného než návratový typ funkce. Narozdíl od funkcí s návratovým typem `int`, není možné u funkcí s jiným návratovým typem označení typu vynechat, protože pak by návratová hodnota byla implicitně `int`.

#### Příklad:

Funkce vynásobí parametr číslem 3.14

```
double pikrat(double x)
{
 return (x * 3.14);
}
```



### 9.2.5 Problémy s umístěním definice funkcí

Jak již víme, funkce nesmí být definována<sup>15</sup> uvnitř jiné funkce. Samozřejmostí je, že funkce může být uvnitř jiné funkce volána<sup>16</sup>.

Problémy nastávají, když je funkce definována až za definicí funkce, která tuto funkci volá. Volající funkce v tomto případě nemá dosud žádné informace o funkci volané (o návratové hodnotě, počtu a typu parametrů). Problém je trochu menší, když má volaná funkce typ návratové hodnoty `int`, protože funkce v C mají implicitní typ právě `int` (viz str. 111). Tato výhoda je však velmi pochybná, protože když volaná funkce má jiný typ návratové hodnoty než `int`, volající funkce o tom neví a s návratovým typem pracuje jako s typem `int`, což většinou končí velmi nepříjemnou chybou.

V tomto případě je třeba určit překladači alespoň návratový typ a jméno volané funkce<sup>17</sup> před jejím voláním<sup>18</sup>. Tento požadavek se řeší dvěma způsoby:

#### 1) Deklarací návratového typu a jména

Deklarace volané funkce se provede ve funkci volající nebo kdekoli na globální úrovni (vně všech funkcí).

Tento způsob pochází z K&R verze jazyka C. V současné době je méně vhodný, protože nepodává žádnou informaci o parametrech funkce<sup>19</sup>.

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

```
main()
{
 double pikrat(); /* totez: double pikrat(), r; */
 double r;

 printf("Zadej polomer : ");
 scanf("%lf", &r);
 printf("Obvod kruhu je %f\n", 2 * pikrat(r));
}
```

<sup>15</sup> Připomínáme, že *definice* funkce je místo, kde leží tělo (program) funkce.

<sup>16</sup> Jinak by funkce ztratily svůj smysl.

<sup>17</sup> Obdoba Pascalského **FORWARD**.

<sup>18</sup> Všimněte si, že počet a typ parametrů volané funkce není zatím důležitý — viz dále.

<sup>19</sup> Které ovšem v K&R verzi C nebyly "podstatné".

```
double pikrat(double x)
{
 return (x * 3.14);
}
```

Program by fungoval stejně dobře i při deklarování funkce `pikrat()` na globální úrovni, jedním z těchto dvou rovnocenných způsobů:

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

double pikrat(); extern double pikrat();

main() main()
{ ... } { ... }
```

#### Štábní kultura:

- Používaná konvence říká, že první způsob: `double pikrat();` by znamenal, že funkce `pikrat()` byla definována v tomtéž modulu. Druhý způsob: `extern double pikrat();` by se použil v případě, že by byla `pikrat()` definována v jiném modulu. Jemné rozdíly mezi oběma způsoby viz též na str. 131.

#### 2) Pomocí funkčního prototypu

Tento způsob přináší ANSI verze jazyka C. Doporučuje se ho používat, protože oproti výše uvedenému způsobu umožňuje překladači navíc i kontrolu počtu a typů skutečných parametrů volané funkce.

```
#include <stdio.h>
double pikrat(double x); /* uplny funkcní prototyp */

main()
{
 double r;

 printf("Zadej polomer : ");
 scanf("%lf", &r);
 printf("Obvod kruhu je %f\n", 2 * pikrat(r));
}
```

```
double pikrat(double x)
{
 return (x * 3.14);
}
```



Program by fungoval stejně dobře i při deklarování funkce `pikrat()` pomocí *funkčního prototypu s anonymními typy*<sup>20</sup>:

```
double pikrat(double);
```

Stejně jako v případě deklarace, lze též použít klíčové slovo `extern`, tedy:

```
extern double pikrat(double);
```

#### Pozor:

Ještě jednou zdůrazněme, že má-li funkce jiný typ než `int` a je volána bez předchozí definice nebo bez uvedení funkčního prototypu nebo bez deklarace, pak se její návratová hodnota konvertuje na `int`, což může způsobit mnohé nepřijemnosti.

Problém funkčních prototypů nebo deklarací funkcí se týká i procedur — funkcí s návratovým typem `void`. Nebyla-li procedura definována nebo deklarována před svým voláním, překladač ji považuje za typ `int`. Když je tato procedura později definována jako funkce vracející typ `void`, překladač ji považuje za funkci s jiným typem a při jejím volání hlásí chybu nebo varovné hlášení.

#### Štábní kultura:

- Dáváme přednost úplnému funkčnímu prototypu, např.:  

```
double vydel(double delenec, double delitel);
```

 před neúplným: `double vydel(double, double);`  
 protože uvedení jmen parametrů dále zvyšuje čitelnost programu.
- Je dobrým zvykem uvést na začátku programu seznam všech funkčních prototypů<sup>21</sup>. Tato práce "navíc" se bohatě vyplatí, protože překladač může provést mnohem lepší kontroly jak návratových typů, tak i parametrů, a upozornit tak na místa, která mohou působit v programu potíže.

#### Poznámka:

- Funkce ze standardní knihovny nemusí být v programu deklarovány, ani nemusí být uváděny jejich funkční prototypy. To je již učiněno v příslušném `.h` souboru. Tedy např. není nutné psát: `FILE *fopen();`<sup>22</sup>  
 protože v souboru `stdio.h` je uveden funkční prototyp této funkce a do našeho souboru se připojí pomocí příkazu: `#include <stdio.h>`

<sup>20</sup> Často též *neúplný funkční prototyp* nebo *anonymní specifikace*.

<sup>21</sup> Nebo ještě lépe je zapsat prototypy do speciálního `.h` souboru, aby mohly být využívány i v jiných souborech — podrobně viz str. 132.

<sup>22</sup> Jak je to uváděno v [KR78].

### 9.2.6 Konverze návratové hodnoty funkce

Pokud není typ návratového výrazu nebo hodnoty shodný s návratovým typem funkce, pak se provádí implicitně typová konverze, např.:

```
int konverze(double d)
```

```
{
 return (d);
}
```

Volání: `k = konverze(4.5);`

způsobí, že hodnota `4.5` je přetypována na `int` (oříznuta) a proměnná `k` bude mít tedy hodnotu `4`.

### 9.2.7 Parametry funkcí

Jazyk C umožňuje pouze jeden způsob předávání parametrů a to *hodnotou* (*call by value*). To znamená, že skutečné parametry nemohou být ve funkci změněny, ale pouze čteny. Jakákoliv změna parametrů uvnitř funkce je pouze dočasná, a po opuštění funkce se ztrácí.

Tento zjevný nedostatek samozřejmě jazyk C neponechává neřešen. Jazyk C umožňuje využít *pointery*<sup>23</sup>, pomocí nichž se řeší volání odkazem<sup>24</sup>.

Pokud nejsou uvedeny funkční prototypy, neprovádí C (na rozdíl od Pascalu) žádnou kontrolu ani typu skutečných parametrů, ani jejich počtu. Je tedy možné psát funkce, které mají proměnný počet parametrů<sup>25</sup>. To však vyžaduje detailní znalosti implementace — zejména způsobu předávání parametrů a manipulace s nimi.

ANSI verze C poskytuje pro tyto speciální případy zvláštní formální parametr "`...`" (tři tečky — *ellipsis*) a také makra popsaná v hlavičkovém souboru `stdarg.h` — podrobně viz [HRŠ92]. V zásadě se ale doporučuje vytvářet funkce s proměnným počtem parametrů až v případě krajního nezbytí.

Z výše uvedeného vyplývá, že je-li např. funkce definována s dvěma formálními parametry typu `int` a pak volána s jedním skutečným parametrem typu `double` a není uveden její funkční prototyp, překladač neodhalí neshodu mezi formálními a skutečnými parametry, a potom se při ladění staneme svědky nepředvídatelného chování programu.

<sup>23</sup> Viz str. 153.

<sup>24</sup> Čili trvalá změna hodnoty skutečných parametrů uvnitř funkce je možná, ale musí se to umět.

<sup>25</sup> Jako např. `printf()`.



### Konverze skutečných parametrů

Typ skutečného parametru by měl souhlasit s odpovídajícím formálním parametrem. Pokud tomu tak není a byl uveden funkční prototyp, pak nás překladač může upozornit na tuto skutečnost varovným hlášením a budou se provádět typové konverze na typy uvedené ve funkčním prototypu — tedy vše bude v pořádku.

#### Pozor:

V praxi se nejčastěji setkáme s problémem, že nepoužijeme funkční prototyp, ale jen deklaraci funkce. Pak se skutečný parametr např. typu `int` nekonvertuje na požadovaný typ formálního parametru, např. `double`. Například výše uvedená funkce `pikrat()` po volání: `r = pikrat(7);` může při absenci implicitní konverze `int` na `double` způsobit nepříjemné chování programu. Vždy je dobré použít volání<sup>26</sup>:

`r = pikrat(7.0);`  
nebo, jde-li o proměnnou, pak volání: `r = pikrat((double) i);`

#### Poznámka:

- V tomto případě kompilátory nehlásí chybu, ale pouze varovné hlášení. Je velkou chybou mávnout nad podobnými hlášeními rukou s tím, že o nic nejde.

## 9.3 Oblast platnosti identifikátorů

Jazyk C umožňuje mnoho "jemností" ve stanovení, kde, kdy a jaký identifikátor (proměnná, funkce, typ, ...) bude viditelný (dostupný) a komu. Další popis se bude soustřeďovat na proměnné, přičemž bude výrazně upozorněno na všechny případné souvislosti s funkcemi.

### 9.3.1 Globální a lokální proměnné

Základní organizace definic v programu je:

globální definice a deklarace  
definice funkcí

Globální deklarace jsou deklarace proměnných, které jsou definovány v jiných souborech (modulech). Protože jsou tyto deklarace často specifikovány

<sup>26</sup> Zde je vidět další výhoda funkčních prototypů před deklaracemi funkcí.

použitím klíčového slova `extern`, nazývají se též často *externí deklarace*<sup>27</sup>. Tyto externí deklarace budou podrobně probrány dále — viz str. 131.

Globální definice definují proměnné, jejichž rozsah platnosti je od místa definice do konce souboru<sup>28</sup> — ne programu! Jak již bylo řečeno dříve<sup>29</sup>, vyskytují se tyto definice vně definic funkcí, např.:

```
int i;

prvni()
{
 ... /* tělo funkce prvni() */
}

int j;

druha()
{
 ... /* tělo funkce druha() */
}

treti()
{
 ... /* tělo funkce tretí() */
}

/* konec souboru */
```

Proměnná `i` je platná ve všech třech funkcích, kdežto proměnná `j` pouze ve funkcích `druha()` a `treti()`.

#### Pozor:

Na rozdíl od Pascalu, kde jsou proměnné definované v hlavním programu globální pro všechny funkce a procedury, jsou v C proměnné definované ve funkci `main()` lokální pouze pro tuto funkci.

<sup>27</sup> Někdy ale též *externí definice* ve stejném slova smyslu.

<sup>28</sup> Pozor! Slovo "souboru" je použito záměrně, protože program se může sestávat z více souborů odděleně překládaných — viz str. 99.

<sup>29</sup> Viz str. 24.



*Lokální definice* definují proměnné, jejichž rozsah platnosti je od místa definice do konce funkce, ve které jsou definovány. Tyto definice se tedy vyskytují uvnitř definic funkcí, např.:

```
prvni()
{
 int i;
 ... /* tělo funkce prvni() */
}
```

Stejně jako v Pascalu mohou být i v C některé globální identifikátory překryty (zastíněny) identifikátory lokálními, např.:

```
int i1, i2;

prvni()
{
 int i1, j1;
 ...
}

int j1, j2;

druha()
{
 int i1, j1, k1;
 ...
}
```

Ve funkci `prvni()` je globální proměnná `i1` překryta lokální proměnnou stejného jména. V této funkci mohou být použity tři proměnné — `i2` (globální) a `i1` a `j1` (lokální). Stejně tak ve funkci `druha()` mohou být použity dvě globální proměnné (`i2` a `j2`) a tři lokální (`i1`, `j1` a `k1`).

#### Příklad:

Následující program vypíše délku nejdelší řádky ze souboru `DOPIS.TXT`. Je složen ze čtyř funkcí, z nichž tři využívají stejnou globální proměnnou `fr` — pointer na čtený soubor. Ta je definována před definicí všech čtyř funkcí, takže je ve všech viditelná.

```
#include <stdio.h>
#define ERROR(mes) { printf("%s\n", mes); return; }
#define EOLN '\n'

FILE *fr;

/*
 * otevírá soubor DOPIS.TXT
 * vrací nenulovou hodnotu při úspěšném otevření, jinak vrací 0
 */
int otevri_soubor(void)
{
 return ((fr = fopen("DOPIS.TXT", "r")) != NULL);
}

/*
 * zavírá soubor DOPIS.TXT
 * vrací nenulovou hodnotu při úspěšném zavření, jinak vrací 0
 */
int zavri_soubor(void)
{
 return (fclose(fr) != EOF);
}

/*
 * čte řádku ze souboru
 * vrací délku přečtené řádky nebo EOF při konci souboru
 */
int cti_radku(void)
{
 int delka = 0,
 c;

 while ((c = getc(fr)) != EOF) {
 if (c == EOLN)
 return (delka + 1);
 else
 delka++;
 }
 return (EOF);
}
```



```

main()
{
 int nejdelsi = 0,
 aktualni;

 if (!otevri_soubor())
 ERROR("Nelze otevrit soubor DOPIS.TXT")

 while ((aktualni = cti_radku()) != EOF)
 nejdelsi = (aktualni > nejdelsi) ? aktualni : nejdelsi;

 printf("Delka nejdelsi radky je %d\n", nejdelsi);

 if (!zavri_soubor())
 ERROR("Nelze zavrit soubor DOPIS.TXT")
}

```

#### Poznámky:

- Lokální proměnné nejsou automaticky inicializovány — jejich hodnota je vždy náhodná.
- Globální proměnné jsou implicitně inicializovány na nulu, tj. **int** proměnné mají hodnotu 0, **float** hodnotu 0.0, **char** hodnotu '\0', atd. Je však dobrým zvykem nespolehat se na tuto službu a u všech proměnných, které mají být inicializovány, tuto inicializaci výslovně uvést.

### 9.3.2 Paměťové třídy

Kromě rozličných typů, mohou být proměnné uvedeny i v různých paměťových třídách. Ty určují, ve které části paměti bude proměnná kompilátorem umístěna, a také, kde všude bude proměnná viditelná. Rozšiřují tak možnosti viditelnosti proměnných, které dosud byly jen globální a lokální.

Jazyk C rozeznává tyto paměťové třídy<sup>30</sup>:

- **auto**
- **extern**
- **static**
- **register**

<sup>30</sup> Přesněji *modifikátory paměťové třídy*.

#### Třída auto

O těchto proměnných se často hovoří jako o automatických. Je to implicitní paměťová třída pro lokální proměnné.

Je-li proměnná definovaná uvnitř funkce bez určení typu paměťové třídy, je její implicitní typ právě typ **auto** a proměnná je uložena ve *stacku* — viz též str. 109.

Proměnná typu **auto** existuje od vstupu do funkce a zaniká při výstupu z funkce. Při každém vstupu do funkce má náhodnou hodnotu — není tedy ani implicitně<sup>31</sup> inicializována na 0, ani si neponechává svoji původní hodnotu mezi dvěma voláními funkce, např.:

|                 |                |            |
|-----------------|----------------|------------|
| func()          |                | func()     |
| {               |                | {          |
| auto int i;     | je stejné jako | int i;     |
| auto int j = 5; | je stejné jako | int j = 5; |
| ...             |                | ...        |
| }               |                | }          |

#### Třída extern

Je to implicitní<sup>32</sup> paměťová třída pro globální proměnné. Tyto proměnné jsou uloženy v datové oblasti.

Klíčové slovo **extern** se používá při odděleném překladu souborů, kdy je třeba, aby dva nebo více souborů sdílelo tutéž proměnnou. Tato globální proměnná je v jednom souboru definována zásadně bez klíčového slova **extern** a ve všech ostatních musí být deklarována s použitím **extern**, např.:

|                          |                                  |
|--------------------------|----------------------------------|
| soubor S1.C              | soubor S2.C                      |
| int suma; /* definice */ | extern int suma; /* deklarace */ |

#### Poznámka:

- Tento způsob je nutné dodržovat i v případě, že se soubory "inkludují" do jednoho souboru. Pak by více definic téže proměnné (bez **extern**) způsobilo problémy.

#### Třída static

Pro tuto paměťovou třídu neexistuje žádná implicitní definice, čili klíčové slovo **static** musí být při definici vždy uvedeno. Proměnné této třídy jsou uloženy v datové oblasti.

Paměťovou třídu **static** využívají nejčastěji lokální proměnné (definované uvnitř funkce), které si ponechávají svoji hodnotu i mezi jednotlivými voláními této funkce<sup>33</sup>.

<sup>31</sup> Inicializace však může být v programu explicitně uvedena.

<sup>32</sup> Slovo "implicitní" má zde poněkud jiný význam než u třídy **auto** — viz dále.

<sup>33</sup> To je podstatný rozdíl mezi **static** a **auto**.



Tato proměnná existuje od prvního volání příslušné funkce až do doby ukončení programu, ale jako lokální proměnná není samozřejmě přístupná z vnějšku funkce.

```
void f(void)
{
 int x = 2;
 static int i = 0;

 printf("f byla volana %d-krat, x = %d\n", i, x);
 i++;
 x++;
}
```

Proměnná *x* je lokální automatická proměnná a proměnná *i* je lokální statická proměnná. Pokaždé, když je *f()* zavolána, je místo pro proměnnou *x* alokováno v jiné části paměti (stacku) a vždy je explicitně inicializováno na hodnotu 2.

Proměnná *i* je inicializována<sup>34</sup> na 0 při prvním vstupu do funkce *f()* a svou hodnotu si zachovává mezi jednotlivými voláními — inicializace na nulu se při dalších voláních funkce *f()* již nikdy neprovede.

Například pro volání:

```
for (j = 0; j < 3; j++)
 f();
```

bude výstup:

```
f byla volana 0-krat, x = 2
f byla volana 1-krat, x = 2
f byla volana 2-krat, x = 2
```

#### Poznámky:

- Globální proměnné i funkce mohou být označeny také jako **static**, což má ten význam, že jsou viditelné pouze v modulu, ve kterém jsou definovány — viz též str. 130.
- Potřebujeme-li více statických proměnných jednoho typu, je vhodné definovat každou proměnnou samostatným příkazem. Některé překladače totiž definici: `static int i, j;` zpracují tak, že statická proměnná bude pouze *i* a proměnná *j* bude mít implicitní paměťovou třídu — tedy **auto**. Obě proměnné budou ale typu **int**.

<sup>34</sup> Opět díky explicitní inicializaci, která sice není nutná, ale zvyšuje přehlednost programu.

Buď je vhodné si aktuální stav překladače ověřit na jednoduchém příkladu nebo je lépe použít implementačně nezávislou definici:

```
static int i;
static int j;
```

#### Třída register

Protože jazyk C je "nízkourovňový jazyk" může programátor požadovat, aby některá proměnná nebyla uložena v paměti, ale pouze v registru počítače. To má výhodu mnohem rychlejšího přístupu k proměnné a tedy i rychlejšího programu.

Označení proměnné jako **register** neváže tuto proměnnou na určitý konkrétní register počítače — to záleží zcela na překladači. Definice proměnné jako: `register int i;` znamená pouze, že tato proměnná může být uložena do registru, pokud je nějaký volný a pokud je to z nejrůznějších systémových důvodů možné.

Označíme-li tedy všechny použité proměnné jako **register**, pak skutečně v registrech budou umístěny pouze některé. Z toho vyplývá, že jako registrovou proměnnou je vhodné označit např. proměnnou jednoduchého cyklu nebo často používaný formální parametr.

U registrových proměnných se neprovádí žádné implicitní inicializace a používají se výhradně jako lokální proměnné<sup>35</sup>.

#### Poznámky:

- Pro definici více proměnných shodného typu v paměťové třídě **register** platí totéž, co pro **static**, je tedy vhodnější psát:  
`register int i;      místo: register int i, j;`  
`register int j;`
- Nelze získat adresu registrové proměnné — viz též str. 148.

#### Příklad:

Funkce pro výpis malé násobilky:

```
void nasobilka(register int k)
{
 register int i;

 for (i = 1; i <= 10; i++)
 printf("%2d x %d = %2d \n", i, k, i * k);
}
```

<sup>35</sup> Nelze mít globální proměnnou typu **register**.





### 9.3.3 Typové modifikátory

Libovolná proměnná určitého datového typu (např. `i` typu `unsigned int`, která je zařazena do určité paměťové třídy (viz předchozí podkapitulu — např. do třídy `static`) může být navíc ještě modifikována *typovým modifikátorem*.

Typové modifikátory rozeznává jazyk C dva:

- `const`
- `volatile`

#### Poznámka:

- Definice proměnné `i` by pak byla<sup>36</sup>:

```
static const unsigned int i = 5;
```

#### Modifikátor `const`

Tento typový modifikátor je zaveden až od ANSI verze jazyka C. Jeho použití specifikuje, že definovanému objektu nesmí být po jeho inicializaci již měněna hodnota. V podstatě je to jakási nedotažená náhrada za symbolické konstanty<sup>37</sup>, protože proměnou v paměťové třídě `const` nelze použít všude tam, kde symbolické konstanty, např. při definování mezí polí<sup>38</sup>. Význam tohoto modifikátoru je v tom, že takto definovaná konstanta je určitého typu (narozdíl od symbolických konstant) a je tedy možno tento typ využít k některým kontrolám prováděným kompilátorem, např. typ skutečných parametrů funkce.

#### Příklad:

```
const float pi = 3.14159; /* definice a inicializace */
const max = 100; /* typ int je implicitní */
int pole[max]; /* chybné - nelze */
pi = 3.14; /* chybné - přiřazení nelze */
```

Mnohem častěji je ale `const` využívána při definici formálních parametrů funkce<sup>39</sup>. Setkáme-li se s takovouto hlavičkou funkce, která vyhledá v řetězci určeném parametrem `str` znak uložený v parametru `co` a vrátí jeho pozici<sup>40</sup>:

```
int hledej(const char *str, char co)
```

<sup>36</sup> To je hrůza, co? Existují však ještě spleťtější definice!

<sup>37</sup> Definované příkazem `#define`

<sup>38</sup> Viz též str. 177

<sup>39</sup> Viz též str. 79.

<sup>40</sup> Viz též str. 202.

neznamená to, že funkce `hledej()` musí být volána vždy s konstantním prvním parametrem (což ovšem lze také), např:

```
i = hledej("nazdar", c);
```

ale to, že parametr označený `const` bude zpracováván pouze jako vstupní (bude jen čten). Skutečný parametr, dosazený při volání funkce na místo `str`, nebude ve funkci `hledej()` měněn, i když by to bylo pomocí pointerů (viz str. 153) proveditelné.

#### Modifikátor `volatile`

Modifikátor `volatile` je opět zaveden až od ANSI verze jazyka C. Jeho použití je ještě méně časté než modifikátoru `const`. Modifikátor `volatile` upozorňuje kompilátor, že takto definovaná proměnná může být modifikována nějakou blíže nespecifikovanou asynchronní událostí mimo váš program, např. pomocí přerušení. Kompilátor tedy nemůže činit žádné závěry o možnosti změny nebo konstantnosti této proměnné např. pro účely optimalizace.

Typickým příkladem je cyklus, jehož ukončení závisí na proměnné `pocet`, která se v těle tohoto cyklu vůbec nevyskytuje, protože je modifikována asynchronně hardwareovým přerušením. Pokud by kompilátor prováděl optimalizaci tohoto cyklu, mohl by se domnívat, že test této proměnné, na které závisí ukončení cyklu, je možno z podmínky ukončení cyklu vynechat a nahradit ho hodnotou, kterou měla tato proměnná před začátkem cyklu.

```
/* ukazky různých definic pomocí volatile */
volatile int i1 = 0; /* definice a inicializace */
volatile int i2; /* definice */
volatile i3; /* typ int je implicitní */
```

```
/* příklad cyklu ovlivňovaného napr. vnějším přerušením */
```

```
volatile pocet;
void wait(int maximum)
{
 pocet = 0;
 while (pocet < maximum)
 ;
}
```

### 9.3.4 Bloky

Blok v C je prvek programu, který v Pascalu není definován.

Blok je uzavřen mezi "{" a "}" a může obsahovat jak definice lokálních proměnných tak i příkazy. Většinou však obsahuje pouze příkazy a pak se jmenuje *složený příkaz* — ten se ovšem v Pascalu vyskytuje.



Poznámky:

- Bezprostředně za "{" mohou být definovány lokální proměnné buď v paměťové třídě **auto** nebo **static**.
- Existence — u automatických proměnných — nebo viditelnost — u statických proměnných — je pouze v bloku, ve kterém jsou definovány.

Příklad:

Program čte jedno celé číslo a je-li kladné, pak čte ještě druhé celé číslo a vytiskne jejich součet. Je-li první číslo záporné, pak čte reálné číslo a vytiskne jejich součin.

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

 scanf("%d", &i);
 if (i > 0) {
 int j; /* j definovano uvnitř bloku */
 scanf("%d", &j);
 printf("Soucet je %d\n", i + j);
 }
 else {
 double f; /* f definovano uvnitř bloku */
 scanf("%lf", &f);
 printf("Soucin je %f\n", i * f);
 }
}
```

Poznámky:

- Uvádějí se dva důvody proč definovat proměnné ve vnitřním bloku:
  - 1) Šetření paměti  
Teoreticky se paměť alokuje buď jen pro **j** nebo jen pro **f**, ale prakticky to záleží na implementaci.  
Pragmaticky vzato — šetření paměti není nutné, pokud jde o jednotlivé proměnné, ale jednalo-li by se o větší pole, pak je možné o tomto způsobu uvažovat.
  - 2) Čistota a přehlednost kódu  
Držíme-li se chvályhodné zásady, že proměnná se definuje pouze tam, kde se použije, pak je definice ve vnitřním bloku vhodná.
- Občas se objeví makro, které využívá definic v bloku, např.:  

```
#define swap(x, y) {double f; f = (x); (x) = (y); (y) = f;}
```

## 9.4 Oddělený překlad souborů — II.

V této chvíli již známe všechny jemnosti paměťových tříd a můžeme tedy využívat všech výhod odděleného překladu<sup>41</sup>. První, co je nutné udělat, je zamyslet se nad rozdělením problému na víc, na sobě co možná nejméně závislých částí. Někdy takové rozdělení vyplývá přímo z podstaty problému, někdy je třeba nad ním popřemýšlet. V každém případě je vhodné strávit úvahou něco času, protože při nevhodném rozdělení programu do více souborů si pravděpodobně více problémů přiděláme než ušetříme.

Druhým krokem je, že po rozdělení je nutné precizně stanovit *interface* (*rozhraní*) neboli styčné body, či jinak — způsoby komunikace oddělených částí navzájem. Vhodnější způsob komunikace je pomocí volání funkcí druhého modulu než pomocí sdílených globálních proměnných, viditelných ve všech modulech<sup>42</sup>.

## 9.4.1 Rozšíření platnosti globální proměnné

Zvolíme-li si způsob (buď zcela nebo jen částečně) komunikace pomocí sdílených globálních proměnných, pak se rozsah jejich platnosti rozšiřuje ze souboru, ve kterém byly definovány, i na všechny soubory, kde jsou deklarovány pomocí **extern**.

Příklad:

Soubor **A.C** obsahuje:

```
/* zacatek souboru A.C */
/* prikaz: #include <stdio.h>
 není nutný - soubor nevyuziva I/O funkce */

int x; /* definice */
extern double f; /* deklarace */

int fun(void)
{
 return (x + (int) f);
}

/* konec souboru A.C */
```

<sup>41</sup> Připomínáme, doufejme, že již celkem zbytečně, že se nejedná o soubory, které se "inkludují" do jednoho souboru.

<sup>42</sup> To je obecný trend, prakticky je vždy nutné přihlídnout ke konkrétním podmínkám.



Soubor B.C obsahuje:

```
/* zacatek souboru B.C */
#include <stdio.h>

extern int x; /* deklarace */
extern int fun(void); /* nebo jen extern int fun();
 nebo jen int fun(void);
 nebo jen int fun(); */

double f; /* definice */

main()
{
 x = 3;
 f = 3.5;
 printf("%d\n", fun());
}

/* konec souboru B.C */
```

Program po spuštění vytiskne číslo 6

#### Poznámky:

- Deklarace: `extern int fun(void);` může být zcela vynechána, protože `fun()` má návratový typ `int`. To je však velmi nevhodný zvyk, který přináší množství problémů, např. vynechá-li se pak omylem externí deklarace funkce, která není typu `int`.
- Rozsah platnosti proměnných `x` a `f` je v celém souboru A.C i B.C.

#### Pozor:

Protože externí deklarace musí být zpracovány linkerem (sestavovacím programem), může to přinést další problémy, např. s délkou identifikátorů. Dobrou zásadou tedy je, aby externí identifikátory (sdílené proměnné a funkce) měly co nejkratší jména — doporučuje se max. 6 znaků.

#### 9.4.2 Statické globální proměnné a funkce

Speciální, ale vcelku časté, je použití paměťové třídy `static` pro globální proměnné v odděleně překládaném programu. Tyto proměnné mají tu výhodu, že jsou viditelné (dostupné) pouze v souboru (modulu), ve kterém byly definovány, ale ne v ostatních souborech<sup>43</sup>.

<sup>43</sup> Nepomůže ani i kdyby byly v jiných souborech deklarovány jako `extern`.

Stejný způsob (paměťová třída `static`) se používá pro označení funkcí, např.: `static int fun(float a)`. Má to naprosto stejný význam, jako u statických globálních proměnných, čili `static` funkce je dostupná pouze ve vlastním souboru a nikoliv v ostatních souborech.

Označení `static` jak globálních proměnných, tak i funkcí je velmi výhodné zejména v případě, kdy na velkém programu<sup>44</sup> pracuje více programátorů najednou. Ti se pak nemusí obávat, že zvolili stejná jména pro identifikátory jako jejich kolegové. Na pojmenování globálních funkcí a proměnných zajišťujících interface se lehce domluví, a všechny ostatní funkce a globální proměnné označí jako `static`. Pak nemůže dojít ke kolizi jmen, protože každý "vidí" jen ty svoje.

#### Poznámka:

- Často se pro označení "lokálních funkcí" využívá makro:

```
#define LOCAL static
```

#### Příklad:

Soubor A.C obsahuje:

```
/* zacatek souboru A.C */
static int x; /* definice */
extern double f; /* deklarace */

static double fun(double x)
{
 return (x);
}

int funkce(void)
{
 return((int) fun(f) + x);
}

/* konec souboru A.C */
```

Soubor B.C obsahuje:

```
/* zacatek souboru B.C */
#include <stdio.h>

extern int x; /* deklarace */
extern int funkce(void); /* funkční prototyp */
```

<sup>44</sup> Pro velké programy — tisíce řádek — se používá označení *projekt*.



```
double f; /* definice */

static void fun(void)
{
 printf("%d\n", funkce());
}

main()
{
 x = 3;
 f = 3.5;
 fun();
}

/* konec souboru B.C */
```

Překlad těchto souborů proběhne úspěšně, ale při sestavení bude linker hlásit chybu, že není definována proměnná `x`. Ta byla totiž definována jako `static` v `A.C` a deklarace: `extern int x;` v `B.C` její paměťovou třídu nezmění.

Po vynechání `static` v definici `x` v `A.C` bude program pracovat správně a nebude zmaten tím, že jsou tam dvě funkce stejně pojmenované `fun()`, ale různých typů.

Funkce `fun()` v souboru `A.C` nemá nic společného s `fun()` z `B.C` a každá je viditelná (dostupná, volatelná) pouze ze souboru, ve kterém je definována.

### 9.4.3 Jak udržet pořádek ve velkém programu

Obecný způsob, jak nejlépe udržet pořádek v externích a statických proměnných a funkcích a jak všeobecně zorganizovat rozložení programu do modulů tak, aby se na nic nezapomnělo, má zhruba následující části:

- 1) Zdrojové texty (těla) všech funkcí (definice funkcí) a definice všech proměnných jsou v souboru `PGM_1.C` — důležitá je přípona `.C`.
- 2) V tomto souboru je striktně rozlišeno, které funkce (popř. nezbytné množství proměnných) se budou dávat k dispozici vně modulu `PGM_1`. Snažíme se navrhnout program tak, aby bylo co nejméně sdílených (externích) proměnných — k manipulaci s nimi poskytujeme spíše speciální funkce.
- 3) Všechny ostatní funkce a globální proměnné označíme jako `static` a tím je ochráníme před nechtěným či nevhodným použitím.
- 4) Funkční prototypy (hlavičky) těchto funkcí a definice globálních proměnných zkopírujeme do souboru `PGM_1.H` — důležitá je přípona `.H` a označíme je jako `extern`.

- 5) Pokud budeme z modulu `PGM_1` poskytovat i některé symbolické konstanty, uvedeme je pouze v souboru `PGM_1.H` a ne v `PGM_1.C`.
- 6) Soubor `PGM_1.c` pak začíná:

```
#include <stdio.h>
#include "PGM_1.H"
```

Tímto trikem máme zaručeno, že v souboru `PGM_1.C` budou známy všechny funkce pomocí funkčních prototypů i všechny symbolické konstanty. Deklarace globálních proměnných jako `extern` v `PGM_1.H` nezpůsobí žádnou kolizi s definicemi těchto proměnných v `PGM_1.C`.

- 7) Soubor `PGM_2.C`, který obsahuje další modul programu a využívá některé funkce nebo proměnné nebo symbolické konstanty z modulu `PGM_1`, pak začíná opět

```
#include <stdio.h>
#include "PGM_1.H" /* sdílené funkce a proměnné */
#include "PGM_2.H" /* deklarace vlastního modulu */
```

Programátor modulu `PGM_2` se vůbec nemusí starat o to zda má správně napsány a uvedeny všechny potřebné sdílené identifikátory. To je starost programátora modulu `PGM_1`. Ten, kdo programuje modul `PGM_2`, si pouze prohlédne soubor `PGM_1.H`, kde má uloženy všechny potřebné informace.

Dodrží-li výše uvedený postup všichni, kteří spolupracují na projektu, omezí se výrazně množství problémů.

### Doporučený obsah .C souboru

Pokud jsme se rozhodli dodržovat předchozí doporučení, je též dobré se seznámit s tím, jak by měly vypadat soubory `.C`, aby se v nich dalo vyznat.

#### Poznámka:

- Pro další popis se předpokládá existence souboru `STDDFN.H` s tímto obsahem:

```
/*
 * STDDFN.H ver. 1.0
 *
 * Standardní definice - modifikováno podle P. Smrhy
 *
 * P. Herout 1992
 */
```



```
#define GLOBAL /* */
#define LOCAL static
#define IMPORT extern
#define EXPORT extern
```

Soubor .C by měl obsahovat postupně tyto části:

- 1) Dokumentační část
  - jméno souboru a verze
  - stručný popis modulu
  - jméno autora a datum
- 2) Všechny potřebné `#include`
  - pouze soubory .H a nikdy soubory .C — využíváme výhod odděleného překladu
  - nejdříve systémové .H soubory příkazem: `#include < >`
  - pak vlastní .H soubory příkazem: `#include " "`
- 3) Deklarace `IMPORT`ovaných objektů
  - Pouze v tom případě, že nejsou v příslušném .H souboru spolupracujícího modulu, čemuž dáváme zásadně přednost.
  - Tyto objekty (funkce nebo proměnné) jsou v jiných modulech definovány jako `GLOBAL`.
- 4) Definice `GLOBAL`ních proměnných
  - Jsou to proměnné (definované vně funkcí), které mají být sdíleny jinými moduly. V nich jsou označeny jako `IMPORT`, pokud to již není učiněno v příslušném .H souboru.
  - Počet těchto sdílených proměnných se snažíme omezit na co nejmenší míru.
- 5) Lokální `#define`
  - Definice symbolických konstant a maker s parametry použitých pouze v tomto modulu.
  - Pokud by byly příliš rozsáhlé (desítky řádek), lze je umístit do speciálního .H souboru — jen pro zvýšení přehlednosti.
- 6) Definice lokálních typů<sup>45</sup>
  - Zásadně se používá definice nového typu pomocí `typedef`.
  - Pokud by byly příliš rozsáhlé, lze je umístit do .H souboru podobně jako předchozí definice symbolických konstant.

<sup>45</sup> Podrobně viz str. 161

- 7) Definice `LOCAL`ních proměnných
  - Jsou to globální proměnné využívané více funkcemi tohoto modulu. Aby nebyly viditelné v jiných modulech, mají paměťovou třídu `static`.
  - Počet těchto sdílených proměnných se snažíme omezit na co nejmenší míru.
- 8) Úplné funkční prototypy `LOCAL`ních funkcí
- 9) Funkce `main()`
  - Uvádí se samozřejmě pouze v případě, že v daném modulu existuje.
- 10) Definice `GLOBAL`ních funkcí
  - Jsou to funkce, které mohou být volány i z jiných modulů.
  - Jejich úplné funkční prototypy jsou uloženy v příslušném .H souboru.
- 11) Definice `LOCAL`ních funkcí
  - Jsou to funkce, které mohou být volány pouze v tomto modulu.
  - Jejich úplné funkční prototypy jsou uloženy v tomto souboru<sup>46</sup>

#### Doporučený obsah .H souboru

Stejně tak, jako platí určitá pravidla pro soubory .C, platí podobná pravidla i pro soubory .H.

Předtím, než bude uvedena doporučená struktura .H souboru, věnujte pozornost ještě dvěma doporučením:

- V .H souboru nesmí být zásadně žádné definice (vymezení paměti pro proměnné nebo kód funkcí) nebo dokonce inicializace sdílených proměnných.
- Neměly by zde být žádné příkazy `#include`

Soubor .H by měl obsahovat postupně tyto části:

- 1) Dokumentační část
  - jméno souboru a verze
  - stručný popis modulu
  - jméno autora a datum
- 2) Definice symbolických konstant
  - Symbolické konstanty, o kterých se dá předpokládat, že budou využívány i v jiných modulech.
- 3) Definice maker s parametry
  - Platí totéž, co u symbolických konstant z předchozího bodu.

<sup>46</sup> Viz bod 8).



- 4) Definice globálních typů
  - Nové datové typy využitelné i v jiných modulech definujeme zásadně pomocí `typedef`.
- 5) Deklarace globálních proměnných příslušného .C modulu
  - Jsou zde označeny jako `EXPORT` — exportují se do jiného modulu.
- 6) Úplné funkční prototypy globálních funkcí příslušného .C modulu
  - Jsou zde označeny jako `EXPORT` — exportují se do jiného modulu.

V této chvíli vám to musí připadat složité, ale tato složitost je pouze zdánlivá. Pokud budete vytvářet programy o desítkách řádků, pak se vám bohatě vyplatí zavést tuto nebo podobnou strukturu souborů. Je ale nutné dodržovat pořádek už od začátku. Předsevzetí, že to opravíte příště, vás bude stát spoustu času a nervů.

#### Příklad:

Výše popsané uspořádání bude demonstrováno následujícím příkladem. Program bude počítat obvod a obsah kružnice a bude složen z modulů `KRU_MAIN.C` a `KRU_IO.C`. K modulu `KRU_MAIN.C` přísluší hlavičkový soubor `KRU_MAIN.H` a ke `KRU_IO.C` přísluší hlavičkový soubor `KRU_IO.H`. Využívá se výše uvedený hlavičkový soubor definic `STDDFN.H`.

#### Poznámky:

- Tento program je samozřejmě vyumělkovaný a tedy i zbytečně složitý, neboť byl psán se snahou ukázat v maximální možné míře možnosti vzájemné provázanosti různých modulů. Ve skutečnosti se doporučuje, aby provázanost byla pouze jedním směrem, tzn. že hlavní modul (v našem případě `KRU_MAIN.C`) může využívat globální funkce podřízených modulů (`KRU_IO.C`), ale nikoliv naopak. Pokud by bylo obtížné tomu zabránit, měly by podřízené moduly využívat pouze minimální počet sdílených globálních proměnných hlavního modulu.
- Jednotlivé funkce nejsou z důvodu šetření místem nijak komentovány, což by bylo v reálném případě nutné. Funkce `vyp_obvodu()` by tedy měla podle dříve uvedených pravidel správně vypadat takto:

```
/*
 * funkce pro vypocet obvodu kruznice
 * polomer kruznice je globalni staticka promenna
 */
LOCAL double vyp_obvodu(void)
{
 return (PI * vyp_prumeru(polomer));
}
```

```
/*
 * KRU_MAIN.C ver. 1.12
 *
 * Program pro vypocty obvodu a obsahu kruznice
 * =====
 *
 * P. Herout, duben 1992
 *
 */

/* systemove hlavickove soubory */
#include <stdio.h>

/* vlastni hlavickove soubory */
#include "STDDFN.H" /* natazeni standardnich definic */
#include "KRU_MAIN.H" /* natazeni symbolickych konstant,
 funkcnich prototypu GLOBAL funkci
 a globalnich typu vlastniho
 modulu */
#include "KRU_IO.H" /* natazeni symbolickych konstant,
 funkcnich prototypu GLOBAL funkci
 a globalnich typu spolupracujiciho
 modulu */

/* deklarace IMPORTovanych objektu */
/* neni nutna, protoze importovane
 * objekty jsou jiz znamy díky příkazu #include "KRU_IO.H"
 * pokud by byly pouzity, vypadaly by napr.: IMPORT int i;
 */

/* definice GLOBALnich promennych */
GLOBAL double obvod;

/* lokalni definice symbolickych konstant */
#define PI 3.14

/* lokalni definice novych typu */
/* zde nejsou zadne typedef pouzity
 * pokud by byly pouzity, vypadaly by napr.:
 * typedef int MOJE_INT;
 */
```



```

/* definice LOCALních promenných */
LOCAL double polomer;
LOCAL double obsah;

/* uplne funkční prototypy LOCAL funkcí */
LOCAL void vyp_obsahu(double polomer);
LOCAL double vyp_obvodu(void);
LOCAL void vypocet(void);

/* funkce main() */
main()
{
 polomer = vstup_dat();

 if (polomer == CHYBA_DAT) {
 /* Chybne zadana vstupni data - zaporna hodnota */
 printf("Polomer kruznice chybne zadan \n");
 }
 else {
 vypocet();
 vystup_dat(obsah);
 }
}

/* definice LOCAL funkcí */
LOCAL void vyp_obsahu(double polomer)
{
 obsah = PI * polomer * polomer;
}

LOCAL double vyp_obvodu(void)
{
 return (PI * vyp_prumeru(polomer));
}

LOCAL void vypocet(void)
{
 obvod = vyp_obvodu();
 vyp_obsahu(polomer);
}

```

```

/* definice GLOBAL funkcí */
GLOBAL double vyp_prumeru(double polomer)
{
 return (2 * polomer);
}

```

#### Poznámky:

- Ošetření správnosti zadaných vstupních dat s dodatečnou možností jejich opravy by mělo být ve funkci, která vstupní data čte.
- Z hlediska čitelnosti programu není vhodné, aby funkce, které dělají podobnou akci ( `vyp_obsahu()` a `vyp_obvodu()` ) pracovaly různě, tj. měly různé typy parametrů a návratové hodnoty.

```

/*
 * KRU_MAIN.H ver. 1.1
 *
 * Hlavickovy soubor pro modul KRU_MAIN.C
 * =====
 *
 * P. Herout, duben 1992
 *
 */

/* definice symbolických konstant vyuzivanych i v jinych modulech */
/* v tomto modulu zadne nejsou */

/* definice maker s parametry */
/* v tomto modulu zadne nejsou */

/* definice globalnich typu */
/* v tomto modulu zadne nejsou */

/* deklarace globalnich promennych modulu KRU_MAIN.C */
EXPORT double obvod;

/* uplne funkční prototypy GLOBAL funkcí modulu KRU_MAIN.C */
EXPORT double vyp_prumeru(double polomer);

```



```

/*
 * KRU_IO.C ver. 1.15
 *
 * Vstupy a vystupy programu pro vypocty kruznice
 * =====
 *
 * P. Herout, duben 1992
 */
/* systemove hlavickove soubory */
#include <stdio.h>

/* vlastni hlavickove soubory */
#include "STDDFN.H" /* natazeni standardnich definic */
#include "KRU_IO.H" /* natazeni symbolickych konstant,
 funkcni prototypu GLOBAL funkci
 a globalnich typu vlastniho
 modulu */
#include "KRU_MAIN.H" /* natazeni symbolickych konstant,
 funkcni prototypu GLOBAL funkci
 a globalnich typu spolupracujiciho
 modulu */

/* deklarace IMPORTovanych objektu */
/* nejsou nutne, protoze importovane
 * objekty jsou jiz znamy díky příkazu #include "KRU_MAIN.H"
 */

/* definice GLOBALnich promennych */
/* v tomto modulu zadne nejsou */

/* lokalni definice symbolickych konstant a maker */
#define kontrola(x) ((x >= 0.0) ? x : CHYBA_DAT)

/* lokalni definice novych typu */
/* v tomto modulu zadne nejsou */

/* definice LOCALnich promennych */
LOCAL double polomer;

/* uplne funkcní prototypy LOCAL funkci */
/* v tomto modulu zadne nejsou */

```

```

/* funkce main() */
/* v tomto modulu není */

/* definice LOCAL funkci */
/* v tomto modulu žádné nejsou */

/* definice GLOBAL funkci */
GLOBAL double vstup_dat(void)
{
 printf("\nZadej polomer kruznice (kladne realne cislo) : ");
 scanf("%lf", &polomer);

 return (kontrola(polomer));
}

GLOBAL void vystup_dat(double obsah)
{
 double prumer;

 printf("Obsah kruhu o polomeru %6.2f je %.2f \n",
 polomer, obsah);

 prumer = vyp_prumeru(polomer);
 printf("Obvod kruhu o prumeru %6.2f je %.2f \n",
 prumer, obvod);
}

```

#### Poznámky:

- Řádka: `#include "KRU_IO.H"`  
je v tomto modulu nezbytná, protože se v `KRU_IO.H` vyskytuje definice symbolické konstanty `CHYBA_DAT`. Ta se nachází v souboru `KRU_IO.H` proto, protože je využívána i modulem `KRU_MAIN.C`. Pokud by byla využívána pouze v modulu `KRU_IO.C`, pak by byla definována pouze v něm — viz též symbolickou konstantu `PI` v modulu `KRU_MAIN.C`.
- Stejně tak řádka: `#include "KRU_MAIN.H"`  
je v `KRU_IO.C` nutná, protože touto modulem popisuje, jak vypadá globální proměnná `obvod` a také funkční prototyp funkce `vyp_prumer()`.
- Za povšimnutí stojí obě proměnné `polomer`, které jsou v obou modulech definovány jako lokální a nemohou si tedy nijak vzájemně vadit.



```

/*
 * KRU_IO.H ver. 1.1
 *
 * Hlavickovy soubor pro modul KRU_IO.C
 * =====
 *
 * P. Herout, duben 1992
 */

/* definice symbolických konstant vyuzivanych i v jinych
 modulech */
#define CHYBA_DAT -1.0

/* definice maker s parametry */
/* v tomto modulu zadne nejsou */

/* definice globalnich typu */
/* v tomto modulu zadne nejsou */

/* deklarace globalnich promennych modulu KRU_IO.C */
/* v tomto modulu zadne nejsou */

/* uplne funkcní prototypy GLOBALnich funkci modulu KRU_IO.C */
EXPORT double vstup_dat(void);
EXPORT void vystup_dat(double obsah);

```

## 9.5 Inicializace jednoduchých proměnných

Inicializace proměnných je definice s určením počáteční hodnoty proměnné. Tento způsob se využívá poměrně často, protože zkracuje a zpřehledňuje program. Inicializace je pro všechny typy proměnných a paměťových tříd syntakticky shodná, ale při pozornějším pohledu se dají nalézt drobné odchylky, které budou popsány dále.

Lokální proměnné jednoduchých typů mohou být explicitně inicializovány. Pokud se této možnosti nevyužije, pak mají náhodné hodnoty. Výjimkou je lokální proměnná v paměťové třídě `static`, která je inicializována na nulu<sup>47</sup>.

<sup>47</sup> Vždy je ale lépe uvést explicitní inicializaci.

Globální proměnné jsou před zahájením programu inicializovány na nulu, ale je dobrým zvykem explicitně inicializovat ty proměnné, které to vyžadují. Tento přístup zvyšuje čitelnost programu.

ANSI C se liší od K&R C v tom, že jakmile je u globální proměnné uvedena inicializace, pak nezáleží na předchozím klíčovém slovu `extern`. V následujících dvou příkladech je proměnná `f` vždy definována s naprosto stejným výsledkem.

|                             |                                    |
|-----------------------------|------------------------------------|
| <code>float f = 1.0;</code> | <code>extern float f = 1.0;</code> |
| <code>main()</code>         | <code>main()</code>                |
| <code>{ ...</code>          | <code>{ ...</code>                 |

### Poznámka:

- ANSI C v tomto případě ve skutečnosti rozlišuje pojmy *deklarace*, *předběžná definice* a *skutečná definice*. Jsou to jemnosti takové úrovně, že se není třeba jimi zabývat. Zájemce se může bližší podrobnosti dozvědět v [HRŠ92].

Inicializace automatických proměnných<sup>48</sup> může být provedena pomocí komplexního výrazu. To znamená, že může obsahovat proměnné nebo i funkční volání — tedy to, co se může objevit na levé straně přiřazovacího příkazu, např.:

```
auto int c = getchar();
```

V podstatě se však doporučuje pouze inicializace pomocí konstantního výrazu.

Globální a statické proměnné mohou být inicializovány pouze pomocí konstantního výrazu, tj. konstant s různými operátory, neboť výraz musí být možno vypočítat v čase překladu, např.:

```
static int i = 5 * 1024;
```

### Co je dobré si uvědomit:

- Funkce nemohou být vkládané (vhnížděné).
- Funkce, která vrací jiný typ než `int` musí být alespoň deklarována ve funkci, která jí volá. Lepším řešením je uvedení jejího funkčního prototypu před definicí volající funkce. Funkční prototyp se uvádí buď za začátku souboru, když program sestává z jednoho souboru nebo "inkludováním" příslušného `.H` souboru při odděleném překladu souborů.

<sup>48</sup> Lokální proměnné v paměťové třídě `auto`.



- Používejte typ `void` pro označení funkce, která nevrací hodnotu nebo nemá parametry.
- Externí identifikátory by měly být co nejkratší — max. 6 znaků.
- Externí proměnné by měly být inicializovány pouze jednou, a to v místě definice.

Cvičení:

- 1) Napište funkci `power(double x, int n)`, která vypíše tabulku mocnin `x` od 1 do `n`.
- 2) Napište funkci `troj(char c, int n)`, která zobrazí na obrazovce trojúhelník o `n` řádcích složený ze znaků `v` a `c`, např.:  

```

*


```

První znak na poslední řádce musí být v prvním sloupci.
- 3) Napište funkci `exist()`, která vrátí 1, když soubor `TEST.TXT` existuje, jinak vrátí 0. Nezapomeňte soubor uzavřít.
- 4) Napište funkci `vyskyt()` se dvěma parametry. První je ukazatel na soubor, druhý je `char`. Funkce vrátí počet výskytů tohoto znaku v souboru.
- 5) Napište funkci `void vypis(FILE *fr)`, která vypíše na obrazovku obsah souboru.
- 6) Zdokonalte funkci `vypis()` tak, aby zajišťovala stránkování po zaplnění obrazovky.
- 7) Napište funkci `double vypis_soubor(void)`, která vypíše pomocí volání funkce `vypis()` na obrazovku obsah libovolného souboru. Jméno souboru čtete z klávesnice a vhodně reagujte na případ, kdy soubor neexistuje. Funkce `vypis_soubor()` vrací 2.2, je-li jméno souboru pouze `K` nebo `k`.  
 Nebyl-li soubor nalezen, vrací hodnotu 1.1, jinak vrací 0.5

## Poznámky:

- Pozor na rozdíl mezi `'K'` a `"K"`.
- Nejednodušujte typ nebo návratovou hodnotu funkce.

- 8) Napište program, který s využitím funkce `vypis_soubor()` bude vypisovat na obrazovku libovolné soubory tak dlouho, dokud uživatel nezdá jméno souboru jako `K`.
- 9) Funkce `vypis()` a `vypis_soubor()` uložte do souboru `FCE.C` a funkci `main()` do souboru `HLAVNI.C`.

- 10) Pomocí `#include` připojte soubor `FCE.C` do souboru `HLAVNI.C`.
- 11) Vyzkoušejte funkci programu vytvořeného v předchozích cvičeních, budou-li oba soubory (`HLAVNI.C` a `FCE.C`) odděleně (bez `#include`) překládány. Zdůvodněte špatnou funkci programu.

## Poznámka:

Soubor `.EXE` sestavte příkazem: `link hlavni, fce`

Používáte-li kompilátor fy Borland, použijte `Project`.

- 12) Oba soubory odděleně překládejte a v souboru `HLAVNI.C` uveďte deklaraci funkce `vypis_soubor()`.
- 13) Oba soubory odděleně překládejte a v souboru `HLAVNI.C` uveďte úplný funkční prototyp funkce `vypis_soubor()`.
- 14) Vytvořte soubor `FCE.H`, do něhož vložte úplný funkční prototyp funkce `vypis_soubor()`. Pomocí `#include` zajistěte spojení mezi odděleně překládanými soubory `FCE.C` a `HLAVNI.C`.
- 15) Napište funkci `long cti_znak(FILE *fr, int *p_c)`, která přečte jeden znak ze souboru a vrátí ho pomocí druhého parametru. Návratovou hodnotou funkce bude počet volání této funkce (využití lokální static proměnné). Hlavní program vypíše pomocí této funkce soubor a nakonec i počet přečtených znaků.
- 16) Vytvořte dva soubory `A.C` a `B.C`, které budou sdílet proměnnou `iii`. Proměnná `iii` se v `A.C` definuje a nastaví a v `B.C` se vypíše její hodnota. Soubory odděleně překládejte a sestavte příkazem: `link a, b` nebo s využitím příkazu `Project` u Borlandských kompilátorů.
- 17) Změňte předchozí program tak, aby `iii` bylo definováno jako globální static a vyzkoušejte chování programu.
- 18) Napište program, který přečte znak z klávesnice. Bude-li to znak `"d"`, pak čtete a vytisknete celé číslo. Bude-li to znak `"f"`, pak totéž provede s číslem `float`. Proměnné, do kterých bude číslo uloženo, definujte až uvnitř bloku za `if` nebo `else`.
- 19) Dopište tento program: (MAX zkuste asi 5000)
 

```

for (i = 0; i < MAX; i++)
 for (j = 0; j < MAX; j++) {
 i = i;
 j = j;
 }

```

 tak, že `i` a `j` budou nejprve globální `int` a pak lokální register `int`. Pokuste se změřit rychlost obou programů.

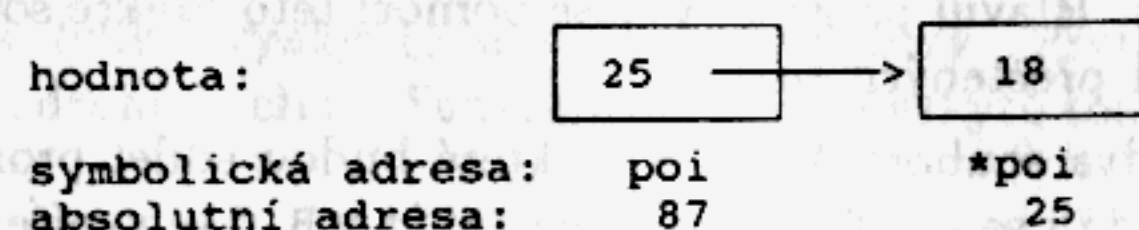


## 10 Pointery

Pointery (též ukazatelé nebo směrničky) jsou "srdce a duše C jazyka". To tedy znamená, že teprve po porozumění pointerům můžeme tvrdit, že umíme programovat v C a že jsme schopni využít všech jeho možností.

Hned z počátku je důležité si uvědomit, že pointer není něco tajuplného, před čím je nutno mít respekt. Pointer je proměnná jako každá jiná, pouze hodnota v této proměnné uložená má odlišný význam od hodnot proměnných, na které jsme byli dosud zvyklí.

Pointer představuje adresu v paměti a na této adrese se teprve ukrývá příslušná hodnota, na kterou jsme byli dosud zvyklí. Jinak řečeno, pointer je proměnná uschovávající paměťovou adresu, např.:



Význam obrázku je:

- proměnná `poi` je pointer (leží na symbolické adrese `poi`)
- hodnota `poi` je 25 (číslo uložené na symbolické adrese `poi` je 25)
- číslo 25 se nevyužije přímo k výpočtu, ale představuje absolutní adresu v paměti
- na absolutní adrese 25 v paměti je hodnota 18 (což je číslo, které se již k výpočtu použít dá)
- `poi` *ukazuje* na hodnotu 18, ale sám má hodnotu 25

Absolutní adresa samotného `poi` není v našem případě důležitá, ale pro úplnost — když je definována proměnná (tedy i pointer), je jí přiděleno potřebné místo v paměti. Toto místo je tak velké, aby se do něj mohla umístit proměnná podle definovaného typu. Protože používáme symbolické adresy (např. `i`, `poi`, ...) není tato adresa (adresa vlastní proměnné) důležitá. Dejme tomu, že `poi` bude ležet v paměti na adrese 87. Pak můžeme říci, že obsah adresy 87 ukazuje na adresu 25 a na této adrese je uložena hodnota 18.

Tato odlišná interpretace obsahu proměnné nás staví před nutnost jak nějakým vhodným způsobem překladači sdělit, že hodnota proměnné je adresa

a ne už cílová hodnota. To se v Pascalu provede pomocí operátoru `^` a v C pomocí operátoru `*`. Identifikátor `poi` doplněný vpředu operátorem `*` — `*poi` — se nyní stává jakousi symbolickou adresou pro hodnotu 18.

### Poznámka:

- Operátor `*` se označuje jako *dereferenční operátor*. V literatuře se občas vyskytne i pojem opačného významu — *referenční operátor*, který je představován operátorem `&`.

Pomocí operátoru `*` můžeme jednak získat obsah na adrese, na níž ukazuje pointer (např. `i = *poi;`), ale je možná i opačná akce — zapsání hodnoty na tuto adresu (např. `*poi = 5;`).

Toto zdánlivě neškodné přiřazení<sup>3</sup>, které při případném omylu nemohlo způsobit mnoho škody u jednoduchých proměnných, má u pointerů skryto velké nebezpečí a s ním tedy i velké omezení — musíme pracovat jen s pamětí, která je naše. Zatímco v případě, kdy přiřazujeme nějakou hodnotu jednoduché proměnné, kterou jsme nikde nedefinovali, nás na tuto chybu upozorní překladač, v případě zápisu do nepřidělené paměti pomocí pointeru nás neupozorní nikdo, leda snad chybná funkce programu.

Toto byly nejzákladnější informace pro práci s pointery. Jak se s pointery opravdu pracuje, to nám ukáží následující stránky.

## 10.1 Základy práce s pointery

### Poznámka:

- Pointer je vždy svázán s nějakým datovým typem. Správně by se místo termínu "*pointer*" mělo vždy uvádět "*pointer na typ ...*".

### 10.1.1 Definice dat typu pointer na typ

#### Poznámky:

- Pro ukázkou budeme definovat pointer na typ `int`. Definice pointerů na jiné datové typy je analogická.
- Všechny identifikátory pointerů budou dále začínat jednotně znaky `p-`.

<sup>1</sup> Toto je jeho druhý význam — první význam je násobení.

<sup>2</sup> Ampérsand — je to jeho druhý význam, druhý význam je bitový součin.

<sup>3</sup> Buďme si stále vědomi toho, že jakékoliv čtení nemůže způsobit větší škodu než získání nesprávné informace — je nedestructivní. Ovšem zápis je destruktivní operace, a proto při jakékoliv operaci zápisu dávejte větší pozor než při operaci čtení.



Je to sice jen maličkost, která není nutná, ale výrazně zvyšuje přehlednost programu, protože, když v programu vidíme např. `p_i`, víme okamžitě, že s touto proměnnou nesmíme zacházet jako s obyčejnou proměnnou, ale jako s pointerem.

Definice proměnné typu pointer na typ `int` je následující:

|                                  |                        |
|----------------------------------|------------------------|
| <b>Pascal</b>                    | <b>C</b>               |
| <code>VAR p_i : ^INTEGER;</code> | <code>int *p_i;</code> |

Je možné — a často se to dělá — uvést definici proměnné typu `int` a pointeru na typ `int` najednou, např.: `int *p_i, i;`  
Z této možnosti vyplývá častá chyba při definici více pointerů najednou:

`int *p_i, p_j;`  
kde pouze `p_i` je pointer na `int` a `p_j` je proměnná typu `int`<sup>4</sup>.

### 10.1.2 Práce s adresovými operátory

Zatím jsme se dozvěděli, jak získat hodnotu proměnné, jejíž adresu (tedy pointer na tuto proměnnou) známe. K úplné znalosti je tedy třeba se naučit i opačný způsob, čili, jak získat adresu proměnné, která již existuje. Opět to není nic těžkého, protože adresa libovolné proměnné se dá získat pomocí referenčního operátoru <sup>5</sup> `&`, tedy:

`int i, *p_i = &i;`  
což je definice `p_i` a jeho současná inicializace adresou proměnné `i`  
nebo: `p_i = &i;`  
což je přiřazovací příkaz, který v programu<sup>6</sup> provede přiřazení adresy proměnné `i` do proměnné (pointeru) `p_i`

#### Poznámka:

- Proměnná `p_i` má samozřejmě také adresu, která se ale moc nevyužívá — viz str. 146.

### 10.1.3 Přiřazení hodnoty pointerům a pomocí pointerů

Protože každá pointerová proměnná je *l-hodnota*<sup>7</sup>, je tedy možné napsat:

`*p_i = 1;`                      zcela v pořádku  
`*(p_i + 3) = 5;`              podezřelé, pokud `p_i` neukazuje na pole

<sup>4</sup> Z tohoto důvodu bylo: `FILE *fr, *fw;` — viz str. 65.

<sup>5</sup> Vzpomeňme si jen na známou funkci `scanf()`.

<sup>6</sup> Ne v definici! Pozorně si všimněte rozdílu mezi inicializací a přiřazením — viz též str. 151.

<sup>7</sup> Může stát na pravé straně přiřazovacího příkazu, protože se jedná o zápis na adresu v paměti — viz též str. 25.

`*(3 + k) = 6;`                      je to možné, ale ne moc dobré, protože zapisujeme na nějakou podezřelou adresu v paměti

Operátor `&` je ale možné použít pouze na proměnné. Ty totiž mají vždy adresu, narozdíl od konstant a výrazů, které ji nemají.

|                                  |                            |
|----------------------------------|----------------------------|
| <code>p_i = &amp;i;</code>       | <code>/* spravne */</code> |
| <code>p_i = &amp;(i + 3);</code> | <code>/* chyba */</code>   |
| <code>p_i = &amp;15;</code>      | <code>/* chyba */</code>   |

#### Poznámka:

- U mikroprocesorů se často používají pointery pro přímý přístup do paměti. Pak má smysl přiřadit pointeru absolutní adresu, tedy např:

```
typedef unsigned char BYTE;
BYTE *p_mem;
p_mem = (BYTE *) 0x80;
```

### 10.1.4 Použití pointerů v přiřazovacích příkazech

Zde se situace poněkud komplikuje, protože je třeba rozeznávat dva typy správnosti přiřazení:

- *statické* — přiřazení je správné v době překladu  
pravidlo: levá strana musí být stejného typu jako pravá strana
- *dynamické* — přiřazení je správné v době překladu i při běhu programu  
pravidla:
  - levá strana by měla být stejného typu jako pravá strana, tzn. nemíchat pointery na různé typy
  - přiřazovat jen přes inicializované nebo správně nastavené pointery

#### Příklady:

Pro všechny příklady platí definice: `int i, *p_i;`

#### 1) Staticky správné:

|                            |                                                                             |
|----------------------------|-----------------------------------------------------------------------------|
| <code>i = 3;</code>        | do <code>i</code> dá hodnotu 3                                              |
| <code>*p_i = 4;</code>     | na adresu v <code>p_i</code> (kde je uložen <code>int</code> ) dá hodnotu 4 |
| <code>i = *p_i;</code>     | do <code>i</code> dá obsah z adresy v <code>p_i</code>                      |
| <code>*p_i = i;</code>     | na adresu v <code>p_i</code> dá obsah <code>i</code>                        |
| <code>p_i = &amp;i;</code> | naplní <code>p_i</code> adresou <code>i</code>                              |

#### 2) Staticky nesprávné:

|                       |                                                                                               |
|-----------------------|-----------------------------------------------------------------------------------------------|
| <code>p_i = 3;</code> | místo adresy je do <code>p_i</code> dána bez přetypování hodnota 3, tedy (absolutní) adresa 3 |
|-----------------------|-----------------------------------------------------------------------------------------------|



`i = p_i;` do `i` se dá obsah `p_i`, tedy adresa místo int hodnoty  
`i = &p_i;` do `i` se dá adresa `p_i`

## 3) Dynamicky správné:

`p_i = &i; *p_i = 4;` je to totéž jako: `i = 4;`  
 před přiřazením hodnoty na adresu v `p_i` musí být `p_i` inicializován

## 4) Dynamicky nesprávné:

`*p_i = 4;` 4 je přiřazena na náhodnou adresu, která je v `p_i`  
 toto je nejčastější chyba!

Příklad:

Na následujících přiřazovacích příkazech budou ukázány některé možnosti práce s pointery.

Předpokládejme definici: `int i, *p_i1, *p_i2;`  
 a dále předpokládejme, že proměnná `i` leží na absolutní adrese 10, `p_i1` na adrese 20 a `p_i2` na adrese 30.

| přiřazení                   | obsah na adrese<br>10 ( <code>&amp;i</code> ) | obsah na adrese<br>20 ( <code>&amp;p_i1</code> ) | obsah na adrese<br>30 ( <code>&amp;p_i2</code> ) |
|-----------------------------|-----------------------------------------------|--------------------------------------------------|--------------------------------------------------|
| <code>i = 1;</code>         | 1                                             | ?                                                | ?                                                |
| <code>p_i1 = &amp;i;</code> | 1                                             | 10                                               | ?                                                |
| <code>*p_i1 = 2;</code>     | 2                                             | 10                                               | ?                                                |
| <code>i = *p_i1 + 1;</code> | 3                                             | 10                                               | ?                                                |
| <code>p_i2 = &amp;i;</code> | 3                                             | 10                                               | 10                                               |

Poznámky:

- Operátor `*` má vyšší prioritu než operátor `+`, takže příkaz:  
`i = *p_i1 + 1;`  
 je ve skutečnosti příkaz:  
`i = (*p_i1) + 1;`
- Operátor `++` má stejnou prioritu jako operátor `*`, ale je použit jako postfix, takže příkaz: `i = *p_i1++;`  
 má význam dvou příkazů: `i = *p_i1; p_i1++;`  
 tedy do `i` se dá obsah z adresy, na kterou ukazuje `p_i1` a pak se pointer `p_i1` inkrementuje. Bude tedy ukazovat na bezprostředně následující prvek (adresu) za `i`. Tento trik se často používá při operacích s řetězcí — viz str. 166.

Příklad:

Program čte dvě celá čísla a zobrazí větší z nich.

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

```
main()
```

```
{
 int i, j, *p_i;

 scanf("%d %d", &i, &j);
 p_i = (i > j) ? &i : &j;
 printf("Vetsi je %d \n", *p_i);
}
```

Poznámka:

- Potřebujeme-li někdy (zřídka) vytisknout adresu na kterou ukazuje pointer, čili hodnotu pointeru, pak použijeme:

```
int i, *p_i = &i;
printf("Adresa i je %p, hodnota p_i je %p \n", &i, p_i);
```

Výpis adresy, na kterou ukazuje pointer, použijeme nejčastěji při ladění, když si nejsme jisti, zda pointer ukazuje tam, kam má.

Pozor:

Znovu (viz str. 148) upozorňujeme, že je třeba dát si velký pozor na tyto dvě odlišnosti:

```
int i, *p_i = &i; int i, *p_i;
 p_i = &i;
```

- V případě vlevo se jedná o definici `p_i` jako pointeru na typ `int` (proto tam musí být `*`) a současně o jeho inicializaci na adresu dříve definované proměnné `i`.
- V případě vpravo je to opět definice `p_i` jako pointeru na typ `int`. Ovšem v přiřazovacím příkazu (už to není inicializace) nesmí být `*`, protože `p_i` přiřazujeme adresu proměnné `i` — viz též příklady statické správnosti a nesprávnosti na str. 149.

## 10.1.5 Nulový pointer NULL

Tak jako je v Pascalu konstanta `nil` je i v C podobná konstanta `NULL`. Je to symbolická konstanta definovaná v `stdio.h` jako např.:

```
#define NULL 0 nebo jako #define NULL ((void *) 0)
```

`NULL` je možné přiřadit bez přetypování všem typům pointerů (pointerům na libovolný typ `dat`) a používá se pro označení, že tento pointer neukazuje na nic.



### 10.1.6 Konverze pointerů

Obecně se jí snažíme vyhýbat, protože přináší množství problémů, např. s pointerovou aritmetikou nebo s ukládáním některých datových typů na určité (sudé) adresy v paměti (*memory alignment*).

Jsou ovšem standardní situace — typicky přidělování dynamické paměti<sup>8</sup> — při kterých je nutné konverzi pointerů používat.

Pokud se nelze v jiných než obvyklých situacích konverzi pointerů vyhnout, pak je dobré použít explicitního přetypování. Spolehnout se na implicitní přetypování je sice možné, ale není to vhodné.

#### Příklad:

```
char *p_c;
int *p_i;
p_c = p_i; /* nevhodne */
p_c = (char *) p_i; /* lepsi */
```

### 10.1.7 Zarovnávání v paměti

Pokud při konverzi pointerů dochází k neočekávané chybě, je vhodné zkusit přetypování tam a zpět a vypsat hodnoty pointerů, např.:

```
printf("p_c pred konverzi %p \n", p_c);
p_i = (int *) p_c;
p_c = (char *) p_i;
printf("p_c po konverzi %p \n", p_c);
```

Odlišné výsledky, které můžeme dostat, jsou způsobeny díky tomu, že některé kompilátory používají taktiku, že určité datové typy, např. `int` jsou uloženy v paměti od sudých adres<sup>9</sup> (*memory alignment*). Při přetypování pointeru s tím překladač počítá a zaokrouhluje případnou lichou adresu na nejbližší nižší sudou. To má pak výhodu rychlejšího přístupu k těmto datům, ale nevýhodu právě při pointerové konverzi. Další nevýhodou je horší využití paměti.

Obecně se dá říci jen to, že bez problémů je pointerová konverze pouze z vyšších (delších) datových typů na nižší (kratší)<sup>10</sup>, tedy např. pointer na

<sup>8</sup> Viz str. 168.

<sup>9</sup> Některé kompilátory, např. od fy Borland, dovedou toto zarovnávání potlačit, takže většinou nejsou problémy.

<sup>10</sup> Viz délku základních datových typů na str. 23.

`long` lze bez problémů přetypovat na pointer na `int` nebo na `char`. Zpětně to ale nemusí vždy vyjít správně.

## 10.2 Pointery a funkce

### 10.2.1 Volání odkazem

Jednou z velmi užitečných vlastností pointerů je, že umožňují volání parametrů "odkazem". Pointery v tomto případě použijeme, když chceme ve funkci trvale změnit hodnotu skutečného parametru. V praxi to znamená, že nepředáváme hodnotu proměnné, ale adresu této proměnné.

#### Poznámka:

- Volání odkazem ve funkci nebo proceduře není ve skutečnosti volání odkazem, tak jak je známe z Pascalu, kdy se formální parametr označí klíčovým slovem **VAR**. V tomto případě kompilátor Pascalu zajistí, že se při volání funkce (procedury) do *stacku* předá adresa proměnné (adresa skutečného parametru), která má být změněna. Ve funkci (proceduře) se pak s formálním parametrem pracuje "zcela normálně" — tedy bez jakýchkoliv triků s pointery. Kompilátor sám zajistí, že se při přiřazovacím příkazu nebude měnit hodnota ve *stacku*, ale hodnota skutečného parametru.

V C je toto "volání odkazem" opět volání hodnotou, kdy se ve *stacku* vytvoří lokální kopie pro uložení parametru — adresy skutečného parametru. Tato lokální proměnná sice zaniká s ukončením příslušné funkce, ale má tu vlastnost, že je v ní uložen pointer, pomocí něhož se nepřímou změnou data, která nemají s touto funkcí nic společného — byly definovány vně této funkce a nezanikají s jejím koncem. Čili výsledek je stejný jako při skutečném volání odkazem v Pascalu, ale postup zpracování jiný. To co v Pascalu prováděl automaticky kompilátor díky klíčovému slovu **VAR**, to musíme v C udělat sami — čili nepracovat s formálním parametrem jako s normální proměnnou, ale jako s pointerem.

Pro zjednodušení bude však dále používán termín *volání odkazem*<sup>11</sup>.

#### Příklad:

Toto je klasický příklad funkce pro výměnu obsahů dvou proměnných.

<sup>11</sup> Nejasnosti mohou vzniknout až při studiu jazyka C++ — objektově orientovaného C — který umožňuje skutečné volání odkazem.



```

void vymen(int *p_x, int *p_y)
{
 int pom;

 pom = *p_x;
 *p_x = *p_y;
 *p_y = pom;
}

```

Funkci `vymen()` pak voláme se skutečnými parametry `i` a `j` takto:

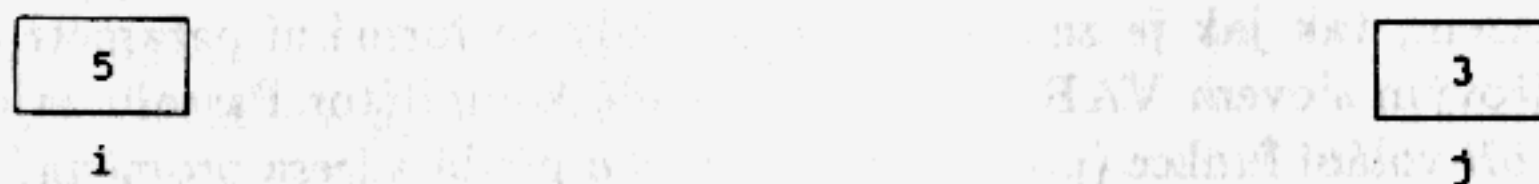
```

int i = 5, j = 3;
vymen(&i, &j);

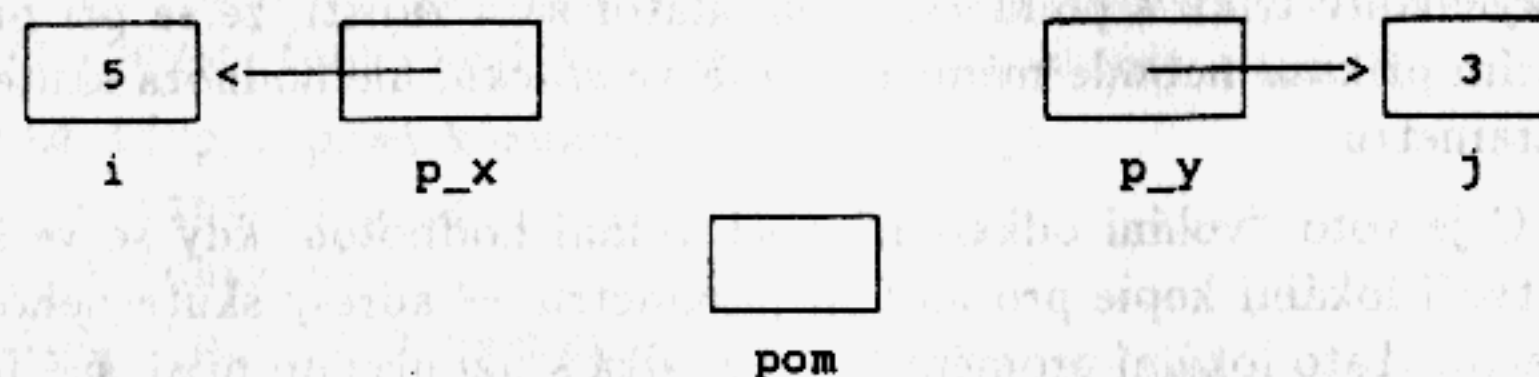
```

Na následujících obrázcích bude podrobně znázorněno, jak vypadají jednotlivé proměnné při postupném provádění funkce `vymen()`:

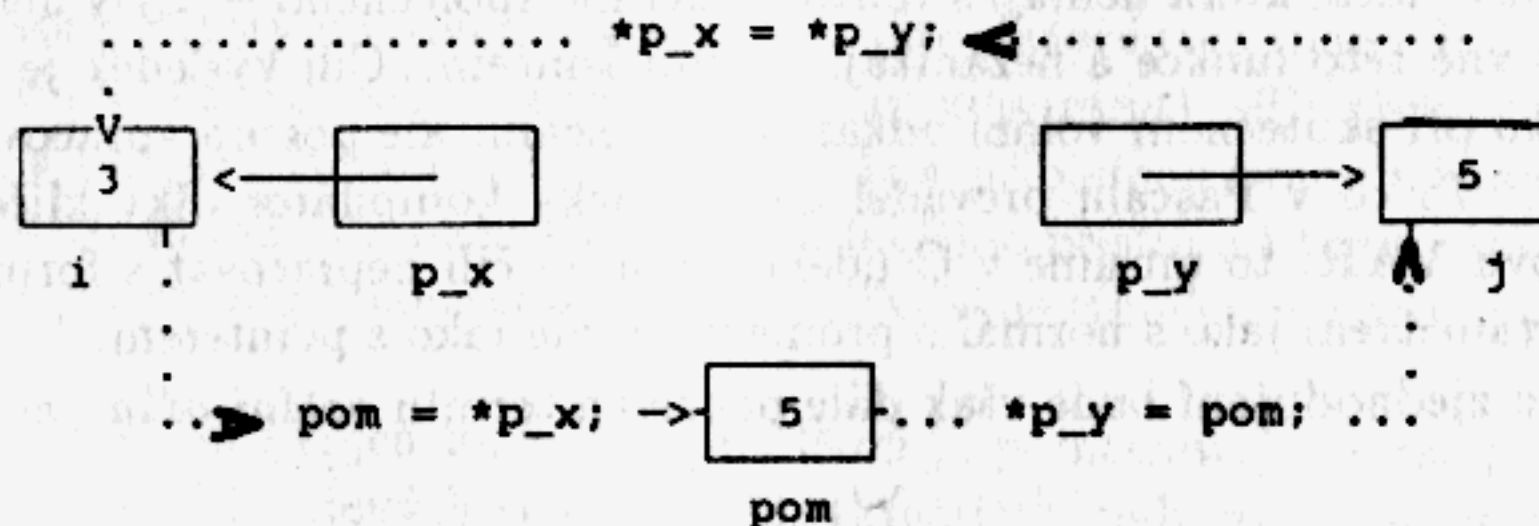
1) Před voláním funkce `vymen(&i, &j)`



2) Těsně po zavolání funkce `vymen(&i, &j)`



3) Po provedení funkce `vymen(&i, &j)` těsně před jejím koncem:



4) Po opuštění `vymen(&i, &j)`



### Pozor:

- Velmi častá chyba při volání je: `vymen(i, j);` která způsobí, že se bude zapisovat na adresy dané obsahem proměnných `i` a `j`, tedy na absolutní adresu 3 a 5, což vede v naprosté většině případů k zhroucení programu.
- Druhou častou chybou je volání: `vymen(*i, *j);` kdy bude zápis proveden na adresy adres z obsahů `i` a `j`, čili např. z absolutní adresy 3 se vezme hodnota, která se použije jako adresa, na které se něco změní. Výsledkem je opět nepochybně zhroucení programu.

### Poznámka:

- Z pragmatického hlediska je ale nutné poznamenat, že zhroucení programu je ještě nejlepší případ, protože pozitivně víme, že v programu je "cosi shnilého". Mnohem horší je stav, kdy nechtěně měníme obsah na adrese, která je někdy využívána a někdy ne. To jsou potom chvíle, které přivádějí do zoufalství, protože někdy program chodí bezchybně a jindy se "nevysvětlitelně" boří. Podle zákona o svinstvu v přírodě se tento stav projeví nejspíše při předvádění programu.

### Příklad:

Program čte pomocí funkce `cti_radku()` řádky z klávesnice a počítá, kolik na ní bylo mezer a malých písmen. Funkce `cti_radku()` vrací hodnotu 1, když na řádce byly nějaké znaky a hodnotu 0, byla-li řádka prázdná. Program skončí, přečte-li prázdnou řádku.

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

```

int cti_radku(int *p_mezery, int *p_male)
{
 int c, pocet = 0;

 *p_mezery = 0; *p_male = 0;
 while ((c = getchar()) != '\n') {
 pocet++;
 if (c == ' ')
 (*p_mezery)++;
 else if (c >= 'a' && c <= 'z')
 (*p_male)++;
 }
 return ((pocet == 0) ? 0 : 1);
}

```



```

main()
{
 int mezery, male;

 while (cti_radku(&mezery, &male) == 1) {
 printf("Na radce bylo %d mezer a %d malych pismen. \n",
 mezery, male);
 }
}

```

**Poznámka:**

- Pokud by byly proměnné `mezery` a `male` definovány jako:

```

int *mezery, *male;

```

pak by nebylo možné volat funkci `cti_radku()` jako:

```

cti_radku(mezery, male);

```

protože nebyla přidělena paměť, na kterou ukazují tyto pointery.

**10.2.2 Pointer na typ void**

Třetí možnost<sup>12</sup>, jak použít typ `void`, je definovat pointer na `void`, např.:

```
void *p_void;
```

Pointer `p_void` neukazuje na žádný konkrétní typ, čili dá se využít pro ukazování na zcela libovolný typ, ovšem pro důsledném přetypování<sup>13</sup>. Občas se pro něj používá výraz *generický pointer*.

Jsou zhruba dvě oblasti použití:

**Pointer na typ void jako pointer na několik různých typů**

```

int i;
float f;
void *p_void = &i; /* p_void ukazuje na i */

```

```

main()
{
 *(int *) p_void = 2; /* nastavení i na 2 */
 p_void = &f; /* p_void ukazuje na f */
 *(float *) p_void = 1.1; /* nastavení f na 1.1 */
}

```

<sup>12</sup> První dvě viz str. 112.

<sup>13</sup> Bez přetypování také nefunguje pointerová aritmetika.

**Pointer na typ void jako formální parametr funkce**

```

#include <stdio.h>
void vymen_pointery(void **p_x, void **p_y)
{
 void *p_pom;

 p_pom = *p_x;
 *p_x = *p_y;
 *p_y = p_pom;
}

```

```

main()
{
 char c = 1, *p_c = &c,
 d = 2, *p_d = &d;
 FILE *fin = stdout, /* zamerna chyba */
 *fout = stdin;

 fprintf(fin, "c = %d, d = %d \n", *p_c, *p_d);
 vymen_pointery((void *)&p_c, (void *)&p_d);
 vymen_pointery(&fin, &fout);
 fprintf(fout, "c = %d, d = %d \n", *p_c, *p_d);
}

```

**Poznámky:**

- Přetypování na `(void *)` při prvním volání funkce `vymen_pointery()` je jen z důvodu zamezení varovnému hlášení o nestejných typech parametrů. Program funguje i bez něho, ale takhle je to čistější.
- Zde jsme opět poněkud předběhli. Pro úplné pochopení funkce `vymen_pointery()` viz str. 172.

**10.2.3 Pointery na funkce a funkce jako parametry funkcí**

Vzpomeneme-li si na funkci `fopen()`, víme, že funkce může vracet také pointer na nějaký datový typ. Této možnosti se v C využívá poměrně často, např. funkce: `char *najdi_adresu_znaku(char c)`. Tato funkce by hledala v paměti od nějaké adresy zadaný znak a vracela by pointer na tento znak.

Často se též využívá možnosti definovat proměnnou jako pointer na funkci vracující nějaký typ, např.: `double (*p_fd)();`



Poznámky:

- Prázdné závorky () před ukončovacím středníkem jsou nezbytné, protože:  
`double (*p_fd);`  
 by znamenalo totéž, co: `double *p_fd;`  
 tedy, že `p_fd` je pointer na `double` a nikoliv pointer na funkci vracející `double`.
- Závorky kolem jména proměnné jsou nezbytné, protože:  
`double *p_fd();`  
 by znamenalo, že tato řádka je deklarace funkce pojmenované `p_fd`, která vrací pointer na `double`.

Máme-li definovanou funkci:

```
double secti_dbl(double f, double g)
```

pak lze napsat přiřazení:

```
p_fd = secti_dbl; /* Pozor! žádný & */
```

které přiřadí pointeru `p_fd` adresu funkce `secti_dbl()`.

Z toho, co již o funkcích víme, vyplývá, že jméno funkce se může v programu objevit v těchto případech:

- 1) Definice funkce:  
`double secti_dbl(double f, double g) {return (f + d);}`
- 2) Deklarace funkce:  
`double secti_dbl();`
- 3) Úplný funkční prototyp:  
`double secti_dbl(double f, double g);`
- 4) Neúplný funkční prototyp:  
`double secti_dbl(double, double);`
- 5) Volání funkce:  
`w = secti_dbl(f1, f2);`
- 6) Přiřazení adresy funkce do proměnné, která je typu pointer na odpovídající typ:  
`p_fd = secti_dbl;`

Poznámka:

- Pomocí pointeru `p_fd`, je pak možno volat funkci `secti_dbl()` jako:  
`w = (*p_fd)(f1, f2);`  
 nebo naprosto stejně jako:  
`w = p_fd(f1, f2);`  
 Přičemž první způsob byl jako jediný možný v K&R verzi C, a oba dva jsou možné v ANSI C.

Příklad:

Následující program vypíše tabulku hodnot polynomů

$x * x + 8$  a  $x * x * x - 3$

v intervalu  $<-1, +1>$  s krokem 0.2

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

```
#define DOLNI (-1)
```

```
#define HORNÍ 1
```

```
#define KROK 0.2
```

```
double pol1(double x)
```

```
{
 return (x * x + 8);
}
```

```
double pol2(double x)
```

```
{
 return (x * x * x - 3);
}
```

```
main()
```

```
{
 int i;
 double x;
 double (*p_fd)(); /* pointer na funkci vracejici double */

 for (i = 0; i < 2; i++) {
 /* prirazeni adresy funkce pointeru fd */
 p_fd = (i == 0) ? pol1 : pol2;
 /* tabulace */
 for (x = DOLNI; x <= HORNÍ; x += KROK)
 printf("%15.5lf \t %15.5lf \n", x, (*p_fd)(x));
 }
}
```

Když se nad předchozím případem zamyslíme, zjistíme, že by bylo vhodné napsat funkci `tabulace()`, takto :

```
void tabulace(double d, double h, double k, double (*p_f)())
{
 double x;
```



```
for (x = d; x <= h; x += k)
 printf("%15.5lf \t %15.5lf \n", x, (*p_f)(x));
}
```

a funkce `main()` by pak vypadala např. takto :

```
main()
{
 tabulace(DOLNI, HORNÍ, KROK, pol1);
 tabulace(-2.0, 2.0, 0.05, pol2);
}
```

### 10.3 Jak číst komplikované definice — I.

Zatím jsme neměli problémy s tím, jak rozluštit, co která definice znamená, protože tyto definice byly velmi jednoduché. S příchodem pointerů se však situace rapidně mění, protože lze definovat pointer na velmi komplikovaný typ<sup>14</sup>.

Příklady zápisů definic pomocí pointerů:

```
int x; x je typu int
float *y; y je pointer na typ float
double *z(); z je funkce vracející pointer na double15
int *(*v)(); v je pointer na funkci vracející pointer na int
```

Pro tuto nepochopitelnou složitost našťastí funguje jednoduché mnemotechnické pravidlo, jak přečíst libovolně komplikovaný zápis.

- 1) Nalezneme identifikátor a od něho čteme doprava.
- 2) Pokud narazíme na samostatnou pravou kulatou závorku `)` (ne na dvojici `()`), vracíme se na odpovídající levou kulatou závorku `(` a od ní čteme opět doprava a samozřejmě přeskakujeme vše již přečtené.
- 3) Narazíme-li na ukončující středník, pak se vracíme na nejlevější dosud zpracované místo a od něj čteme doleva.

Čtení si prakticky ukážeme na naposledy uvedeném příkladu definice proměnné `v`: `int *(*v)();`.

Tento zápis definice čteme následujícím způsobem:

- Ve spleti závorek a hvězdiček se nalezne identifikátor, tedy `"v"` a řekneme: `v je ...`

<sup>14</sup> Situace bude ještě horší, až se dozvíme, jak vypadá pole — viz str. 190.

<sup>15</sup> Toto není definice proměnné, ale deklarace funkce.

- Od něho se čte doprava, dokud nenarazíme na pravou kulatou závorku `)`. Pravá kulatá závorka nás vrací doleva až na odpovídající levou kulatou závorku `(` a od ní se čte zase doprava, tedy znak `"*`

a přidáme: `... pointer na ...`

- Přeskakujeme jméno proměnné `"v"` a pravou kulatou závorku `)`, které nám už posloužily, a čteme stále doprava, dokud nenarazíme na další samostatnou `)` nebo na ukončující středník, v našem případě tedy `"()"`<sup>16</sup>

a přidáme: `... funkci vracející17 ...`

- Ukončovací středník nás vrací doleva před již zpracovanou `"(` a protože jsme vpravo již všechno přečetli, čteme nyní opačně<sup>18</sup> — doleva — tedy `"*"`

a přidáme: `... pointer na ...`

- Čteme stále doleva, tedy `"int"`

a dodáme: `... int` a jsme hotovi.

Výsledek "průzkumu" tedy dohromady zní:

`v je pointer na funkci vracející pointer na int`

Když to zkontrolujeme s příklady nahoře, tak vidíme, že vítězíme.

Tímto jednoduchým způsobem se dá kdykoliv přečíst libovolná definice. Jen je nutné trochu si to vyzkoušet na několika příkladech.

### 10.4 Definice s využitím operátoru `typedef`

Pomocí operátoru `typedef` lze vytvořit nový datový typ. Při definování proměnných jednoduchých datových typů se `typedef` příliš nepoužívá, zatímco při použití pointerů a struktur<sup>19</sup> se využívá velmi často.

Příkaz: `typedef float *P_FLOAT;`

vytvoří nový typ jako pointer na `float` a pojmenuje tento typ identifikátorem `P_FLOAT`.

Pozor:

Není to definice proměnné, která vyhrazuje paměť, ale definice nového typu, která pouze určuje vzorec (šablonu) pro další akce.

<sup>16</sup> Toto je případ, kdy je pravá kulatá závorka součástí dvojice závorek `"()"`, která jako obvykle označuje funkci.

<sup>17</sup> Každá funkce něco vrací.

<sup>18</sup> Kdyby nás sem vrátila `"")`, pak bychom četli doprava, jako v předchozích bodech, protože by vpravo ještě něco zbývalo — alespoň středník.

<sup>19</sup> Viz str. 231.



Identifikátor `P_FLOAT` se stává synonymem typu `float` a je možné ho dále v programu použít pro definice, deklarace, přetypování, atd.

Další možná definice pomocí `typedef`, která se u pointerů často používá:

```
typedef int *P_INT;
```

`P_INT p_i, p_j;`      *správná definice dvou pointerů na int*

Ukazují-li pointery na další pointery, lze to též zapsat pomocí `typedef`, např.:

```
int *p_i; **p_p_i; typedef int *P_INT;
 typedef P_INT *P_P_INT;
 P_INT p_i;
 P_P_INT p_p_i;
```

kde `p_i` je *pointer na int* a `p_p_i` je *pointer na pointer na int*, čili jinak — *pointer na typ pointer na typ int*.

Příklad:

Definice nového typu: `typedef double (*P_FD)();`

kde `P_FD` je typ pointer na funkci vracející `double`

Pak jsou možné následující definice:

- Definování proměnné:

```
P_FD p_fd;
```

definuje `p_fd` jako pointer na funkci vracející `double`

- Definování návratového typu funkce:

```
P_FD g(void)
{
 return (sqrt);
}
```

kde `g` je funkce vracející pointer na standardní funkci `sqrt()`  
odmocnina z 10 by se pak vypočítala jako: `*(g()))(10.0)`

## 10.5 Pointerová aritmetika

S pointery lze provádět některé aritmetické operace. Nutno ovšem poznamenat, že je jich mnohem méně, než aritmetických operací s normálními proměnnými.

Platné operace s pointery jsou:

- součet pointeru a celého čísla
- rozdíl pointeru a celého čísla
- porovnávání pointerů stejných typů
- rozdíl dvou pointerů stejných typů

Ostatní aritmetické operace s pointery jdou mnohdy sice provést, ale nemají žádný smysl a stávají se zdrojem chyb.

Předtím, než se budeme aritmetickými operacemi s pointery hlouběji zabývat, je nutné vysvětlit ještě jeden operátor.

### 10.5.1 Operátor `sizeof`

Operátor `sizeof` zjistí velikost zkoumaného datového typu v Bytech. Občas se používá i při práci s jednoduchými datovými typy — viz str. 23, ale těžiště jeho práce je právě ve spolupráci s pointery a se složenými datovými typy — viz str. 234.

Často je totiž nutné zjistit velikosti objektů<sup>20</sup>, na které pointery ukazují nebo mají ukazovat.

Poznámka:

- Pro ty, co se zajímají i o efektivitu programu, dodáváme, že `sizeof` nepracuje po spuštění programu, ale je vyhodnocen v čase překladač, takže vlastní běh programu nijak nezdržuje. Je tedy velmi vhodné jej používat.

Mějme definice: `int i, *p_i;`

- Po příkazu: `i = sizeof(p_i);`

bude v `i` počet Bytů nutný pro uložení pointeru na typ `int` — tento příkaz se používá málokdy.

- Po příkazu: `i = sizeof(*p_i);`<sup>21</sup>

bude v `i` počet Bytů nutný pro uložení objektu, na který ukazuje `p_i`, tedy objektu typu `int` — tento příkaz se naopak používá velmi často.

Poznámka:

- Typ hodnoty, kterou vrací `sizeof`, není určen, ale většinou to bývá `unsigned int` nebo `unsigned long`.

Nyní se můžeme vrátit ke slibované pointerové aritmetice.

### 10.5.2 Součet pointeru a celého čísla

Výraz: `p + n`

znamená, že se odkazujeme na `n`-tý prvek za prvkem, na který právě ukazuje pointer `p`.

Hodnota adresy tohoto prvku je pak: `(char *) p + sizeof(*p) * n`  
čili k pointeru se nepřičítá příslušné celé číslo, ale násobek tohoto čísla a velikosti typu, na který pointer ukazuje.

<sup>20</sup> Například při dynamické alokaci paměti — viz str. 166.

<sup>21</sup> Pozor, ne: `i = sizeof(p_i*);`.



Mějme definice a předpokládejme, že pro tyto typy platí:

```
char *p_c = 10; sizeof(char) == 1
int *p_i = 10; sizeof(int) == 2
float *p_f = 10; sizeof(float) == 4
```

Zpočátku všechny pointery ukazují na adresu 10. Po inkrementaci ale platí:

```
p_c + 1 == 11
p_i + 1 == 12
p_f + 1 == 14
```

ale: (char \*) p\_i + 1 == 11

stejně jako: (char \*) p\_f + 1 == 11

protože přetypování na (char \*) změnilo velikost objektu.

Protože součet celého čísla a pointeru je opět pointer, je možné psát výrazy typu: p\_i = p\_i + 5;

kde p\_i bude ukazovat na 5-tý prvek za původním prvkem.

#### Příklad:

Program přečte double číslo a zobrazí odpovídající Byty z adresy v paměti, na níž je toto číslo uloženo.

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

```
typedef unsigned char *P_BYTE;
```

```
main()
```

```
{
```

```
 double f;
```

```
 P_BYTE p_byte;
```

```
 int i;
```

```
 printf("Zadej realne cislo : ");
```

```
 scanf("%lf", &f);
```

```
 p_byte = (P_BYTE) &f;
```

```
 for (i = 0; i < sizeof(double); p_byte++, i++)
```

```
 printf("%d. byte = %02Xh \n", i, *p_byte);
```

```
}
```

#### Poznámky:

- Přetypování p\_byte = (P\_BYTE) &f; je nutné, protože &f je adresa typu double a p\_byte je typu pointer na unsigned char.
- Odkazujeme-li se do paměti, používáme explicitně typ unsigned char a protože char může být jak signed, tak unsigned (což záleží na im-

plementaci), ale my většinou chceme přechíst z paměti Byte ve významu neznaménkového celého čísla, čili unsigned char.

#### 10.5.3 Odečítání celého čísla od pointeru

Tato aritmetická operace má naprosto stejnou filosofii jako přičítání, tedy výraz: p - n znamená, že se odkazujeme na n-tý prvek před prvkem, na který právě ukazuje pointer p

Hodnota adresy tohoto prvku je pak: (char \*) p - sizeof(\*p) \* n

#### 10.5.4 Porovnávání pointerů

Pro porovnávání velikostí dvou pointerů lze použít operátory:

```
< <= > >= == !=
```

Výrazy typu: p\_i < p\_j mají smysl pouze tehdy, když oba pointery jsou stejného typu a oba ukazují na tentýž úsek paměti, např. na jedno pole. Pak má tento výraz hodnotu 1 (TRUE), je-li p\_i (jako adresa) menší než p\_j (také jako adresa) nebo 0 (FALSE) v ostatních případech.

#### Poznámka:

- Omezení, že oba pointery musí ukazovat na tentýž úsek paměti, se zavádí proto, že mnoho systémů má paměť segmentovanou a porovnávání pointerů z různých segmentů nedává žádné rozumné výsledky.

#### Pozor:

Bez explicitního přetypování nelze porovnávat pointery různých typů kromě porovnání pointeru s NULL pointerem.

Například, jsou-li p\_c a p\_d pointery na char, přičemž p\_c ukazuje na začátek bloku dat délky MAX, pak je možné zjistit, zda p\_d ukazuje dovnitř tohoto bloku takto:

```
(p_d >= p_c && p_d < p_c + MAX)
```

Znaky z tohoto pole bychom tiskli:

```
for (p_d = p_c; p_d < p_c + MAX; p_d++)
 printf("%c", *p_d);
```

Pointerovou aritmetiku lze využít pro rychlé kopírování paměti. S využitím předchozího příkladu budeme kopírovat blok délky MAX z adresy p\_c na adresu p\_d. Využijeme pomocné proměnné p\_t, která je též pointer na char.

```
for (p_t = p_c; p_t < p_c + MAX; *p_d++ = *p_t++)
 ;
```



zde je základní trik: `*p_d++ = *p_t++`

Ten se dá rozepsat do tří příkazů:

- Nejprve se provede: `*p_d = *p_t;` tedy kopírování jednoho Bytu
- V druhé části se provedou dva příkazy: `p_d++; p_t++;` tedy inkrementace obou pointerů.

Po ukončení kopírování ukazuje `p_d` na první Byte za nově zkopírovaným blokem, takže potom je vhodný příkaz: `p_d -= MAX;` který nastaví `p_d` na začátek nového bloku dat.

### 10.5.5 Odečítání pointerů

Výraz: `p_d - p_c`

má smysl pouze ukazují-li pointery na stejné pole dat. V tomto případě slouží ke zjištění počtu položek pole mezi těmito pointery, přičemž položkou pole je datový typ, na který jsou oba pointery definovány<sup>22</sup>.

Výraz: `p_d - p_c`

se dá přepsat jako: `((char *) p_d - (char *) p_c) / sizeof(*p_d)`

Předpokládejme opět předchozí příklad bloku dat. Následující část programu nalezne v tomto bloku znak "?" a vytiskne jeho pozici. Pokud není v bloku dat znak "?" nalezen, vytiskne se -1.

```
for (p_d = p_c; *p_d != '?' && p_d < p_c + MAX; p_d++)
;
printf("%d \n", (p_d < p_c + MAX) ? p_d - p_c + 1 : -1);
```

**Pozor:**

Sčítat pointery nelze! Výsledkem sečtení by byla nesmyslná hodnota.

## 10.6 Dynamické přidělování a navrácení paměti

Při vysvětlování paměťových tříd (viz str. 109) jsme se zmínili o dynamickém přidělování paměti. Jak již víme, paměť se dá dynamicky, tj. za chodu programu, přidělit buď ze *stacku*, což v naprosté většině případů za nás provádí operační systém, a pak z *heapu*, kde naopak přidělování paměti řídíme většinou pouze my. Dále se tedy budeme zabývat jen tím, co můžeme ovlivnit — tedy *heapem*.

<sup>22</sup> Předpokládáme, že je již zbytečné zdůrazňovat, že oba pointery musí ukazovat na stejný typ.

Přidělování paměti za chodu programu tak, aby nedošlo ke kolizi s ostatními daty, je dosti zapeklitý problém. Naštěstí programovací jazyky poskytují pro tuto složitou a citlivou operaci standardní procedury a funkce, které ji provedou za nás, takže se možnost omylu výrazně snižuje<sup>23</sup>. Nám pak stačí tyto *run-time* procedury a funkce pouze zavolat. Jejich označení jako *run-time*, znamená, že tyto rutiny provádějí operace za běhu programu. Jsou to operace, jejichž požadavky (parametry) nemohly být určeny v čase překladu. V Pascalu je to např. funkce `NEW` a v C např. funkce `malloc()`. Ty přidělí z *heapu* blok paměti potřebné velikosti a vrátí jeho adresu. Na strojové úrovni je to volání funkcí, které manipulují s pamětí — provádějí správu paměti, což je operace poměrně složitá. Velikost paměti, kterou přidělí, si v C musíme sami přímo určit, kdežto v Pascalu toto určení velikosti za nás může udělat kompilátor. V obou jazycích ale obecně platí, že velikost přidělené dynamické paměti je závislá na velikosti objektu, na který příslušný pointer ukazuje.

V souvislosti s dynamickým přidělováním paměti se mluví o tzv. *životnosti dat*, což znamená, jak dlouho nově alokovaný objekt v paměti existuje. Vzpomeňme si např. na automatické proměnné<sup>24</sup>, které mají dobu životnosti určenou dobou provádění funkce, v níž jsou definovány. Životnost dat v *heapu* je od doby přidělení paměti až do jejího uvolnění nebo do skončení programu. V C se uvolnění paměti provede např. voláním funkce `free()` a v Pascalu např. voláním funkce `DISPOSE`.

**Poznámka:**

- Z předchozích řádek je již asi jasné, že není dobré míchat data ve *stacku* a v *heapu* — např. definovat pointer na data ve *stacku*.

**Pozor:**

Funkční prototypy dále popisovaných funkce jsou uvedeny v souborech `stdlib.h` (nebo někdy též v `alloc.h`), který je nutné do programu připojit pomocí příkazu: `#include <stdlib.h>`

### 10.6.1 Přidělení paměti

Standardní a nejčastěji používanou funkcí pro přidělení paměti je funkce `malloc()`, jejíž jediný parametr je typu `unsigned int`. Tento parametr udává počet Bytů, které chceme alokovat.

Funkce `malloc()` vrací pointer na `void`<sup>25</sup>, který představuje adresu prvního přiděleného prvku. Tento pointer je velmi vhodné přetypovat na pointer na odpovídající typ.

<sup>23</sup> Že ale není nulová, to jistě poznáte sami.

<sup>24</sup> Viz str. 123.

<sup>25</sup> Viz též str. 156



Není-li v paměti dost místa pro přidělení požadovaného úseku, vrací `malloc()` hodnotu `NULL`.

#### Poznámky:

- Je dobrým zvykem při každém přidělování paměti testovat návratovou hodnotu na `NULL` a nespolehat se na pocit, že paměti musí být dost. Předjedeme tím mnoha problémům. Ladíme totiž nejčastěji programy s malými daty, pro která paměť většinou stačí. V reálném provozu bude ale program použit pro skutečná data, kterých je většinou mnohem více.
- V tomto případě se nemusíme starat o problémy s zarovnáváním na určité adresy v paměti (*memory alignment*), protože `malloc()` na tyto problémy pamatuje.

#### Příklad:

Ukázka použití `malloc()` včetně reakce na případný neúspěch.

```
int *p_i;

if ((p_i = (int *) malloc(1000)) == NULL) {
 printf("Malo pameti \n");
 exit(1);
}
```

#### Poznámka:

- Při přidělování dynamické paměti je třeba si uvědomit, že sice žádáme o přidělení určitého počtu Bytů, ale je jenom věcí operačního systému, kolik paměti navíc se skutečně z *heapu* přidělí. Například v MS-DOSu se přiděluje paměť po tzv. *paragrafech*, což jsou násobky 16-ti Bytů. V praxi to tedy znamená, že žádáme-li příkazem: `p_c = malloc(1);` o přidělení jednoho jediného Byte, systém jich ve skutečnosti přidělí 16. Důvod tohoto "plýtvání" pamětí je, že systém musí mít pro každý přidělený blok dynamické paměti nějakou administrativu, aby např. věděl, že je právě tenhle kousek paměti obsazený. Tuto skutečnost je třeba si uvědomit zejména v programech, které se snaží přidělit větší množství kratších úseků paměti. Pak paměť dojde dříve, než kdyby program žádal o přidělení jednoho velkého úseku<sup>26</sup>.

### 10.6.2 Uvolňování paměti

Uvolňování neboli navrácení paměti je opačná akce než přidělování. Platí obecná zásada, že již nepotřebnou paměť je dobré okamžitě vrátit, a nečekat až na konec programu.

Pro uvolnění paměti se využívá funkce `free()`, jejímž parametrem je pointer na typ `void`, který ukazuje na začátek dříve přiděleného bloku.

#### Poznámka:

- Funkce `free()` vrátí již nepotřebnou paměť zpět do *heapu*, čili uvolní ji pro další libovolné použití. Důležité ale je, že `free()` nemění hodnotu svého parametru. To znamená, že pointer stále ukazuje na totéž místo v paměti. S touto pamětí se dá tedy dále pracovat, ale ve skutečnosti už programu nepatří! Takové využívání uvolněné paměti může způsobit množství problémů.

Po příkazu: `free((void *) p_c);`

je tedy vhodné uvést bezprostředně i příkaz: `p_c = NULL;` čímž zabráníme možnému přístupu do uvolněné paměti.

### 10.6.3 Příklady přidělování paměti

Mějme definice:

```
char *p_c;
int *p_i;
```

Pak příkaz: `*p_c = 'a';`

není zcela korektní<sup>27</sup>, protože `p_c` ukazuje někam do paměti, kterou nemáme přidělenou.

Před tímto příkazem je tedy třeba uvést příkaz: `p_c = malloc(1);`

Abychom dodržovali dobré návyky, je třeba navíc zjistit, zda se nám požadovanou paměť podařilo přiřadit, příkaz tedy bude:

```
if ((p_c = malloc(1)) == NULL) {
 print("Neni volna pamet\n");
 return;
}
```

Chceme-li v následujícím kroku výpočtu alokovat 20 Bytů pomocí stejného pointeru `p_c`, pak příkaz<sup>28</sup>: `p_c = malloc(20);`

<sup>27</sup> Viz též dynamická nesprávnost — str. 149.

<sup>28</sup> Pouze z důvodů šetření místem uvedeno bez testu na `NULL`.

<sup>26</sup> Je to stejný systém jako v obchodě — stejný objem zboží odebraný najednou je levnější než tentýž objem odebraný postupně.



není úplně nejvhodnější, protože jsme ztratili pointer na dříve alokovanou paměť — je v ní znak "a" — tuto paměť se nám již nikdy nepodaří uvolnit a do konce běhu programu bude znak "a" "viset" někde v paměti.

Před každou další novou alokací je vhodné použít příkaz: `free(p_c);` který již nepotřebnou paměť uvolní.

Jestliže potřebujeme alokovat paměť pro uložení `int` hodnoty pak příkaz:

```
p_i = malloc(2);
```

není opět nejvhodnější, protože je systémově závislý — typ `int` nemusí nutně využívat jen 2 Byte. Lepší je způsob: `p_i = malloc(sizeof(int));`

Ale ani tato varianta však není optimální, protože jsme `p_i` přiřadili pointer na typ `void`.

Bezchybná<sup>29</sup> varianta je: `p_i = (int *) malloc(sizeof(int));`

Používáme-li při alokování paměti operátor `typedef`<sup>30</sup>, pak např.:

```
typedef int *P_INT;
```

```
P_INT p_i;
```

```
p_i = (P_INT) malloc(sizeof(int));
```

je naprosto korektní, narozdíl od:

```
p_i = (P_INT) malloc(sizeof(P_INT));
```

protože velikost pointeru na `int` může být odlišná od velikosti datového typu `int`.

#### Poznámka:

- Je důležité si stále uvědomovat, že pointery nejsou celá čísla a nelze je s nimi navzájem míchat, přinejmenším pro udržení dobré přenositelnosti.

#### 10.6.4 Funkce `calloc()`

V mnoha případech je nutné alokovat paměť pro `n` prvků, z nichž každý má velikost `size`. Pro tento případ slouží funkce `calloc(n, size)`, která alokuje toto pole prvků stejně jako příkaz: `malloc(n * size)` a navíc jej vynuluje.

Funkce `calloc()` opět vrací pointer na začátek alokované oblasti nebo `NULL`, nepodařilo-li se požadovanou paměť přidělit.

#### Pozor:

V některých systémech je nutno paměť přidělenou pomocí `calloc()` uvolnit pomocí funkce `cfree()` a ne `free()`!

<sup>29</sup> Aby byla opravdu bezchybná, musel by být uveden test na úspěšnost přidělení paměti!

<sup>30</sup> Což se vřele doporučuje zejména u složených datových typů.

## 10.7 Pointer jako skutečný parametr funkce

Potřebujeme-li ve funkci změnit ne hodnotu proměnné jednoduchého typu, ale hodnotu pointeru, je to samozřejmě možné. Jak na to, to nám ukáže druhá část následujícího programu.

Program přečte z klávesnice 10 `double` čísel, uloží je do paměti a vy počítá jejich součin. Program je rozdělen do tří pomocných funkcí a funkce `main()`.

První funkce alokuje blok paměti a vrátí pointer na jeho začátek, nebo vrátí `NULL`, není-li paměti dostatek.

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

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

```
#define SIZE 10
```

```
double *init(void)
```

```
{
```

```
 return ((double *) malloc(SIZE * sizeof(double)));
```

```
}
```

Druhá funkce přečte čísla z klávesnice a uloží je do paměti. Její formální parametr je pointer na začátek alokované oblasti. Hodnota tohoto parametru se nemění.

```
void cteni(double *p_f)
```

```
{
```

```
 int i;
```

```
 for (i = 0; i < SIZE; i++) {
```

```
 printf("Zadejte %d. cislo : ", i + 1);
```

```
 scanf("%lf", p_f + i); /* zde není & !!! */
```

```
 }
```

```
}
```

Třetí funkce provádí součin všech přečtených čísel tak, že se posledním číslem naplní formální parametr `p_soucín` a zbylými čísly se násobí. Parametr `p_soucín` je zpracován pomocí pointeru, protože se jeho hodnota bude měnit.

#### Poznámka:

- Ve skutečnosti by bylo vhodnější, kdyby funkce `nasob()` vracela typ `double` jako výsledek násobení. Pointer na parametr `soucín` je použit pouze z pedagogických důvodů.



```
void nasob(double *p_f, int size, double *p_soucín)
{
 for (size--, *p_soucín = *(p_f + size); --size >= 0;)
 *p_soucín *= *(p_f + size);
}
```

Hlavní funkce `main()` pouze volá funkce předchozí.

```
main()
{
 double *p_dbl, souc;

 if ((p_dbl = init()) == NULL)
 /* nedostatek pameti - konec */
 return;

 cteni(p_dbl);
 nasob(p_dbl, SIZE, &souc);
 printf("Soucín čísel je: %12.3lf \n", souc);
}
```

#### Poznámka:

- Při volání funkce `nasob()` se první parametr `p_dbl` uvádí bez `&`, protože je to pointer na `double`, kdežto třetí parametr `souc` je uveden s `&`, protože je to proměnná typu `double`.

Funkci `init()` lze přepsat tak, aby adresu alokované paměti nevracela, ale aby ji uložila do svého parametru.

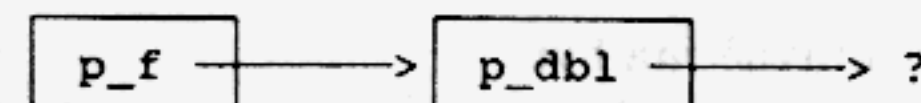
```
void init(double **p_f)
{
 *p_f = (double *) malloc(SIZE * sizeof(double));
}
```

a v hlavním programu by byla pak volána jako:

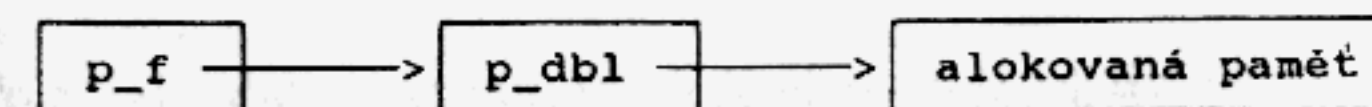
```
main()
{
 double *p_dbl;
 init(&p_dbl);
 ...
}
```

Situace v paměti bude následující:

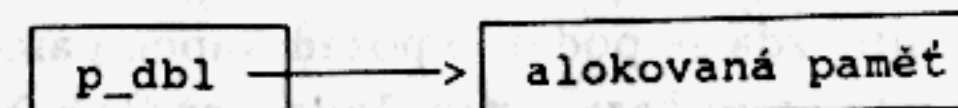
- 1) Těsně po volání funkce `init()`



- 2) Po volání `malloc()` se změní na



- 3) Po opuštění `init()` se stane



#### Časté chyby:

`int *p_i = 2;` inicializace `p_i` na adresu 2  
`*p_i = 3;` zápis hodnoty 3 do paměti na adresu 2

```
void nastav(int *i)
{
 *i = 5;
}
```

```
main()
{
 int j, *p-j;
 nastav(j); má být: nastav(&j);
 nastav(*j); má být: nastav(&j);
 nastav(*p-j); má být: nastav(p-j);
 nastav(&p-j); má být: nastav(p-j);
}
```

v obou případech s `p-j` musí být někde dříve alokována paměť, tedy:

`p-j = (int *) malloc(sizeof(int));`

nebo provedeno přiřazení: `p-j = &j;`



```
main()
{
 int *p_i;
 p_i = malloc(5);
}
```

1) chybí `#include <stdlib.h>` nebo alespoň deklarace: `void *malloc();`  
 2) správně má být:  
`p_i = (int *) malloc(5 * sizeof(int));`  
 3) chybí test na `NULL`

#### Co je dobré si uvědomit:

- Pointery nejsou celá čísla.
- Vždy používejte přetypování na správný typ pointeru.
- Velikost objektů pro alokaci paměti určujte zásadně pomocí `sizeof`.
- Vždy testujte, zda se podařilo požadovanou paměť přidělit.
- Nezapomeňte na příkaz: `#include <stdlib.h>` nebo: `#include <alloc.h>` protože `malloc()` nevrací typ `int`, ale pointer na `void`.
- Uvolňujte pouze přidělenou paměť.
- Funkce `free()` uvolní paměť, ale nezmění hodnotu pointeru. Použití této hodnoty má za následek nepředvídatelné chování programu.
- Pro předávání parametrů odkazem je nutné formální parametr definovat jako pointer a skutečný parametr uvádět s operátorem `&`.
- Má-li se změnit hodnota skutečného parametru, který je pointer, je nutno použít formální parametr jako pointer na pointer.
- Pokud je skutečným parametrem `NULL` pointer, proveďte explicitní přetypování na typ formálního parametru.

#### Cvičení:

- 1) Napište program, který metodou půlení intervalu zjistí, kolik je k dispozici paměti v *heapu* s přesností na 10 Byte. Používejte funkci `free()`.
- 2) Zjistěte velikosti všech základních typů dat (`int`, `float`, ...) v Bytech.
- 3) Napište funkci `set()` s jedním vstupním a druhým výstupním parametrem. Funkce jako svoji návratovou hodnotu vrátí 1, bylo-li ve vstupním parametru písmeno, a 0 v ostatních případech. Do výstupního parametru uloží funkce opačný typ písmena (velká převádí na malá a naopak), byl-li vstupní znak písmeno, nebo jej nezmění — pro ostatní znaky.

## 11 Jednorozměrná pole

Pole, jako datová struktura složená ze stejných prvků, je ve všech programovacích jazycích často využíváno. Jazyk C zavádí pro práci s polem oproti Pascalu některá omezení — týkající se zejména stanovení mezí pole, ale díky těsné souvislosti polí a pointerů umožňuje dělat s poli mnohem větší "kouzla"<sup>1</sup> než Pascal.

Hned v začátcích je ale nutné poznamenat, že základní práce s poli je v C velmi podobná Pascalu, např. přístup k jednotlivým prvkům pole je naprosto stejný. Pokud tedy nechceme při práci s poli využívat pointery, nikdo<sup>2</sup> nás k tomu nenutí.

### 11.1 Základní dovednosti

Při jakékoliv práci s poli v C je nutné mít neustále na paměti, že pole v C **nemají volitelnou dolní mez** tak, jak tomu je v Pascalu. V C je dolní mez pole vždy 0 (nula), tedy pole vždycky začíná prvkem s indexem 0! Tento způsob práce s poli je zaveden kvůli zvýšení efektivity při přístupu do pole a také kvůli spolupráci polí a pointerů.

|                                     |               |
|-------------------------------------|---------------|
| <b>Pascal</b>                       | <b>C</b>      |
| VAR x: ARRAY [0 .. pocet-1] OF TYP; | TYP x[pocet]; |

Tento příkaz staticky alokuje blok paměti pro `pocet` objektů typu `TYP`, přičemž rozsah indexů je od 0 do `pocet-1`.

Hodnota `pocet` musí být známa v čase překladu, čili musí to být konstantní výraz — nejčastěji se používá symbolická konstanta — viz dále příklady.

#### Poznámka:

- Je-li pole definované tímto způsobem, označuje se jako statické. Pokud jsme pozorně četli kapitolu 9.1 na str. 108 můžeme se nyní zeptat, zda

<sup>1</sup> Tyto možnosti však plně oceníme až při práci s vícerozměrnými poli — viz str. 217.

<sup>2</sup> Snad jen menší efektivita programu v některých případech.



i pole, definované uvnitř funkce — pro které se tedy alokuje dynamická paměť ve *stacku* — je statickým polem. Odpověď zní ano, je. Jakmile je pole definováno pomocí “[ ]”, čili jeho velikost je známá již při překladu, jedná se o pole statické. Pole, pro které se alokuje místo až během výpočtu v *heapu*, je pak pole dynamické. Potěšující fakt pro nás je, že jakmile je pro pole alokováno místo, pak nezáleží na tom, zda je to pole statické nebo dynamické, protože práce s oběma těmito typy polí je naprosto stejná. To je velký rozdíl od Pascalu, kde lze také alokovat dynamické pole, ale práce s ním se zásadně liší od práce s polem statickým.

**Pozor:**

Častou chybou, kterou určitě alespoň jednou uděláte, je, že si sice uvědomujete, že pole v C mají nulovou dolní mez, ale zapomenete na to, že index posledního prvku pole je o jedničku menší, než je hodnota uvedená v definici pole!

```
int x[10]; definice pole x o deseti prvcích typu int
x[0] = 5; správný přístup k prvku pole — v pořadí prvnímu x
x[9] = 7; správný přístup k poslednímu (desátému) prvku pole x
x[10] = 1; chybný přístup k jedenáctému prvku pole x
 x[10] není poslední prvek pole, protože pro pole bylo
 vyhrazeno pouze 10 prvků
 tento příkaz přiřadí hodnotu 1 do paměti bezprostředně
 za polem x
```

**Poznámky:**

- Výše uvedený příklad chyby je o to zrádnější, že: jazyk C zásadně nekontroluje meze polí<sup>3</sup> tedy ani dolní, ani horní mez. Tyto kontroly, Pascalem běžně prováděné, jsou totiž časově velmi náročné a C, jako jazyk nižší úrovně, je z důvodů efektivity neprovádí. Pokud tedy předpokládáme, že by mohla v programu nastat situace, kdy by meze polí mohly “přetéci” nebo “podtéci”, pak to musíme sami ošetřit.
- Přetečení nebo podtečení mezí je většinou zdrojem špatně odhalitelných chyb. Nejčastější jsou tzv. “chyby +1”. To jsou chyby, kde index o jedničku překračuje rozsah pole<sup>4</sup>. Z toho plyne poučení, že když program pracuje špatně a je podezření na chybnou práci s poli, je třeba se nejprve zaměřit na “chybu +1”.
- Dobrým programátorským zvykem je vyhýbat se kódům závislým na implementaci. Často uváděný případ takového kódu je: `a[i] = i++;`

<sup>3</sup> Kompilátor dokonce neposkytuje ani varovná hlášení!

<sup>4</sup> Různé výzkumy říkají, že tvoří asi 80% chyb při práci s poli.

**Příklad:**

“Klasická” definice statického pole v C:

```
#define MAX 10
int x[MAX], y[MAX * 2], z[MAX + 1];
```

Byla definována tři pole:

- 1) `x` o deseti prvcích s indexy od 0 do 9
- 2) `y` o 20-ti prvcích s horním indexem 19
- 3) `z` o 11-ti prvcích s indexy od 0 do 10

Způsob definice pole `z` je často používán, potřebujeme-li, aby horní index byla hodnota uvedená v definici pole a nemuseli jsme při práci s polem stále myslet na to, že je třeba odečítat jedničku.

Pokud potřebujeme zavést nový typ pole — což děláme většinou jen tehdy, když nový typ není základním datovým typem<sup>5</sup> — definujeme ho následujícím způsobem:

Pascal: `TYPE NOVY_TYP = ARRAY [0 .. pocet-1] OF ZNAMY_TYP;`

C: `typedef ZNAMY_TYP NOVY_TYP[pocet];`

**Příklad:**

Definice nových typů:

Pascal

C

TYPE

```
VEC5 = ARRAY [0 .. 4] OF INTEGER; typedef int VEC5[5];
VEC3 = ARRAY [0 .. 2] OF CHAR; typedef char VEC3[3];
FVEC = ARRAY [0 .. 9] OF REAL; typedef float FVEC[9];
```

Definice nových proměnných:

```
VAR v5 : VEC5; VEC5 v5;
 v3 : VEC3; VEC3 v3;
 f : FVEC; FVEC f;
```

Syntaxe indexované proměnné (prvku pole) je stejná v C i v Pascalu, např.: `v5[3]`

V C jsou prvky pole *l-hodnotami*, tzn. že mají adresu, kterou lze získat, a prvky pole se dají samozřejmě měnit.

**Příklad:**

Program zjistí počet jednotlivých písmen v souboru `TEXT.TXT`. Malá a velká písmena jsou považována za totožná. Pouze z důvodu max. jednoduchosti je vynechán test správnosti otevření a uzavření souboru.

<sup>5</sup> V C není třeba mít stejné typy např. pro formální a skutečné parametry funkcí jako v Pascalu.



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

#define PO CET ('Z' - 'A' + 1)

main()
{
 FILE *fr;
 int c, i;
 int pole[PO CET];

 for (i = 0; i < PO CET; i++)
 pole[i] = 0; /* nulovani pole */

 fr = fopen("TEXT.TXT", "r");

 while ((c = getc(fr)) != EOF) {
 if (isalpha(c))
 /* precteny znak je pismeno */
 pole[toupper(c) - 'A']++;
 }

 printf("V souboru byl tento pocet jednotlivych pismen\n");
 for (i = 0; i < PO CET; i++)
 printf("%c - %d \n", i + 'A', pole[i]);

 fclose(fr);
}
```

## 11.2 Pole a pointery

Naše dosavadní akce s poli zřejmě nebyly příliš překvapující, protože výše popsaný způsob práce s polem je podobný ve všech programovacích jazycích. Jak již bylo řečeno, tento způsob pro práci s poli docela postačuje. Chceme-li však opravdu využít všech možností, které nám jazyk C dává, je nutné prozkoumat a naučit se vztah mezi pointery a poli.

Teprve pak totiž můžeme pracovat dobře i s dynamickými poli, o nichž jsme se dosud jen zmínili.

Začíná-li v C každé pole od 0, pak se dá lehce vypočítat adresa libovolného prvku podle vztahu:

$\&x[i] = \text{bázová adresa } x + i * \text{sizeof}(typ);$

Proměnná pole — v našem případě  $x$  — je považována za adresu do paměti. Protože pointery jsou také adresy do paměti, je možné již vytuší, jak C zachází vnitřně s poli. Dopředu si prozradíme, že práce s poli pomocí indexů a pomocí pointerové aritmetiky<sup>6</sup> je shodná a že velmi podobná je tedy i práce s poli a práce s pointery.

Jestliže se ale vrátíme k problémům indexování pole, pak výraz:  $x[i]$  je totožný s výrazem:  $*(x + i)$  protože  $x + i$  je adresa daná součtem bázové adresy pole představované hodnotou  $x$  a indexu představovaného hodnotou  $i$ . Operátor  $*$  pak umožňuje získat obsah na této adrese.

### 11.2.1 Dynamická pole

Mějme definice:

TYP a\_var[pocet];      definice statického pole  
TYP \*p\_var;              definice pointeru

Jak a\_var, tak i p\_var jsou pointery na typ TYP. Liší se ale v následujícím:

- 1) a\_var je konstanta (konstantní pointer) a její hodnota se nedá měnit. Není to l-hodnota<sup>7</sup>. Hodnota a\_var je adresa začátku bloku paměti alokovaného pro statické pole.
- 2) p\_var je proměnná s neurčenou počáteční hodnotou. Alokuje paměť pouze pro sebe (pro pointer), ale nealokuje žádnou paměť pro pole. Je to l-hodnota.

Mějme pointer na blok paměti alokované pomocí funkce malloc().

```
int *p_i;
p_i = (int *) malloc(4 * sizeof(int));
```

pak se p\_i dá považovat za dynamické pole, které jak známo vzniká v čase běhu programu, takže následující dvojice výrazů jsou zcela ekvivalentní:

```
p_i[0] == *p_i
p_i[1] == *(p_i + 1)
p_i[2] == *(p_i + 2)
p_i[3] == *(p_i + 3)
```

<sup>6</sup> Viz též str. 162.

<sup>7</sup> Má sice adresu, ale její hodnota se nedá měnit.



### 11.2.2 Podobnost statických a dynamických polí

Z toho, co již o polích a pointerech víme, lze tušit, že práce se statickými a dynamickými poli bude velmi podobná. Ve skutečnosti je tomu tak, že rozdílný je pouze způsob definice a alokace paměti. Další práce je pak naprosto stejná.

#### Poznámka:

- Je sice skutečností, že mnoho programátorů dodržuje konvenci, že do statického pole se přistupuje pomocí indexů a do dynamického pole pomocí pointerové aritmetiky, ale to nelze v žádném případě zevšeobecňovat ani důrazně doporučovat. Zcela přípustný je i obrácený způsob nebo použití pouze jednoho způsobu (indexů nebo pointerové aritmetiky) u obou typů polí.

Bude-li statické pole  $x$  definováno jako: `int x[4];` pak<sup>8</sup>:

`(x + 1) == *((char *) x + sizeof(int) * 1)`

použijeme-li adresní operátor `&` pak:

`&x[0] == &*(x + 0) == x`

`&x[1] == &*(x + 1) == x + 1`

tedy obecně:

`&x[i] == &*(x + i) == x + i`

Mějme definici: `int x[1], *p_x;` pak:

- 1)  $x$  je konstantní pointer a nemůže být změněn. Avšak  $*x$  není konstanta — je to obsah staticky alokované paměti.

Tedy výraz: `*x = 2;`

je správný, a znamená totéž co výraz: `x[0] = 2;`

- 2) Pointer  $p_x$  není inicializován, takže výraz: `*p_x = 2;` je správný pouze syntakticky<sup>9</sup>, ale sémanticky je chybný<sup>10</sup>, protože měníme obsah na neznámé adrese, kterou jsme předtím nealokovali.

Takovýto příkaz lze napsat až po inicializaci  $p_x$  např.:

`p_x = (int *) malloc(sizeof(int));`

- 3) Ovšem je možné i přiřazení: `p_x = x;` kdy  $p_x$  a  $x$  ukazují na stejnou adresu v paměti.

Opačné přiřazení: `x = p_x;`

možné není, protože  $x$  je konstantní pointer.

<sup>8</sup> Viz též pointerovou aritmetiku na str. 162

<sup>9</sup> Terminologií ze str. 149 bychom řekli staticky správný.

<sup>10</sup> Tedy dynamicky chybný.

- 4) Často se vyskytne definice s inicializací: `int x[1], *p_x = x;` Zde se nepřisazuje adresa pole  $x$  na nedefinovanou adresu v paměti, ale do proměnné  $p_x$ , protože se jedná o inicialiaci<sup>11</sup>.

### 11.2.3 Další zvláštnosti a dovednosti při práci s poli

Kromě základních zvláštností při práci s poli v C (začínají od 0, nekontrolují se meze, ...), které byly popsány výše, budou v této části uvedeny některé další, které se "tak moc nepotřebují".

#### Práce s celým polem najednou

Zde je nutno bohužel poznamenat, že pracovat s celým polem najednou, narozdíl od standardního Pascalu, jazyk C neumožňuje. Operace s celými poli najednou se nedají provádět<sup>12</sup>, i když jsou pole stejného typu, takže:

```
typedef int VEC9[9];
```

```
VEC9 x, y;
```

```
x = y; takto kopírovat nelze !
```

```
x == y; takto porovnávat nelze !
```

V těchto případech je nutné použít cyklu:

| kopírování                              | porovnávání                             |
|-----------------------------------------|-----------------------------------------|
| <code>for (i = 0; i &lt; 9; i++)</code> | <code>for (i = 0; i &lt; 9; i++)</code> |
| <code>x[i] = y[i];</code>               | <code>if (x[i] != y[i])</code>          |
|                                         | <code>break;</code>                     |

#### Přístup do pole pomocí pointerů

Adresa pole a pointer se dají porovnávat<sup>13</sup>, tedy: `double f[4], *p_f;` pak  $p_f$  ukazuje na prvek pole  $f$ , platí-li: `f <= p_f < f + 4`

Stejně lze kdykoliv nahradit průchod prvky pole pomocí indexů a pointerů, např.:

```
for (p_f = f; p_f < f + 4; p_f++)
 printf("%d. prvek pole f ma hodnotu : %f \n", p_f - f, *p_f);
```

Stejně je i kopírování nebo porovnávání polí z předchozích příkladů:

```
for (p_x = x, p_y = y; p_x < x + 9;)
 *p_x++ = *p_y++; /* if (*p_x++ != *p_y++) */
```

<sup>11</sup> Viz též str. 151.

<sup>12</sup> Když se to udělá chytře, tak jdou "obchvatem" — viz str. 233.

<sup>13</sup> Viz porovnávání pointerů — str. 165.



**Poznámka:**

- Přístup k prvkům pole pomocí pointerů bývá obvykle mnohem efektivnější než přístup pomocí indexace. V tomto případě se totiž pro získání dalšího prvku pole pouze připočítává konstanta — tj. velikost prvku pole — k aktuální adrese současného prvku, kdežto při indexaci je nutné nejprve touto konstantou vynásobit index a výsledek pak přičíst k báze adrese. Samozřejmě ale záleží na implementaci a na tom, jak překladač optimalizuje. U dobrých překladačů a v jednoduchých příkladech by tyto časy měly být stejné, protože kompilátor by měl přístup pomocí indexů převést na přístup pomocí pointerů.

**Příklad:**

Program ze str. 177 je přepsán pomocí pointerů. Je na něm vidět, že ne všude je použití pointerů výhodné. Při závěrečném tisku je sice přístup pomocí pointerů možná rychlejší, ale získání indexu (tedy písmene), je komplikovanější a mnohem méně čitelné.

Z toho plyne jedno poučení:

*“Optimalizujte jen tehdy, když to přinese podstatné úspory.”*

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

```
#include <ctype.h>
```

```
#define POCET ('Z' - 'A' + 1)
```

```
main()
```

```
{
```

```
FILE *fr;
```

```
int c, *p_i;
```

```
int pole[POCET];
```

```
for (p_i = pole; p_i < pole + POCET; p_i++)
```

```
 p_i = 0; / nulovani pole */
```

```
fr = fopen("TEXT.TXT", "r");
```

```
while ((c = getc(fr)) != EOF) {
```

```
 if (isalpha(c))
```

```
 /* precteny znak je pismeno */
```

```
 (*(pole + toupper(c) - 'A'))++;
```

```
}
```

```
printf("V souboru byl tento pocet jednotlivych pismen\n");
```

```
for (p_i = pole; p_i < pole + POCET; p_i++)
 printf("%c - %d \n", p_i - pole + 'A', *p_i);
```

```
fclose(fr);
```

```
}
```

**Jak zjistit velikost pole**

Zde je jedna z mála odlišností<sup>14</sup> při práci se statickým a dynamickým polem a to v rozdílné interpretaci operátoru `sizeof`.

Mějme dvě pole definované jako:

```
int x[10], *p_x;
```

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

Po alokaci dynamické paměti pomocí `malloc()` budou jak `x`, tak i `p_x` považovány za pointery na pole deseti prvků typu `int`, přičemž `x` představuje statické pole a `p_x` pole dynamické.

Na oba pointery lze použít operátor `sizeof`, který ale poskytne odlišné, leč správné a zdůvodnitelné výsledky.

```
sizeof(x) == 10 * sizeof(int) tedy např. 20
```

```
sizeof(p_x) == sizeof(int *) tedy např. 4 — velikost adresy
```

**Poznámka:**

- Občas bývá vhodné používat makra pro zjištění počtu prvků pole. Následující makro však pracuje správně pouze se statickými poli.

```
#define pocet(pole) (sizeof((pole)) / sizeof(pole[0]))
```

**11.3 Pole měnící svoji velikost**

Občas se v programu dostaneme do situace, kdy je potřeba větší pole, než jsme si původně mysleli. V jazyku C s využitím dynamických polí to není žádný problém. Ukážeme si způsob, jak může pole během výpočtu dynamicky měnit svoji velikost — zvětšovat se nebo i zmenšovat (“dýchat”). Výhodou tohoto způsobu je, že se pole jmenuje stále stejně a že se využívá jen tolik paměti, kolik jí skutečně potřebuje.

**Poznámka:**

- Možná si řeknete, proč pole zmenšovat? Je to opět z důvodů šetření pamětí, protože jsou programy, které využívají velmi mnoho dynamické paměti<sup>15</sup> a pak je každý ušetřený KB dobrý.

<sup>14</sup> Upřímně řečeno, ne že bychom se s ní setkávali příliš často.

<sup>15</sup> Například textové editory.



Příklad:

Mějme program, kdy využíváme dynamické pole<sup>16</sup> o 10-ti prvcích. Během výpočtu se stane, že nám počet 10-ti prvků přestane stačit a tak alokujeme místo pro dalších 10 prvků, takže pole bude mít nyní velikost 20 prvků. Tato užitečná vlastnost se udělá pomocí jednoduchého triku. Vždy, když nám nedostačuje velikost pole, alokujeme nové pole o 10 prvků větší, původní pole do něj překopírujeme a pak toto původní pole uvolníme pomocí funkce `free()`. Začínáme tedy s polem `x`<sup>17</sup>.

```
int *x, /* pole */
 pocet = 10, /* velikost pole */
 *p_pom1, *p_pom2, *p_nove; /* pomocne pointery */
x = (int *) malloc(pocet * sizeof(int));
```

Nyní normálně pracujeme s polem `x`, např.: `x[5] = 3;`  
V této chvíli přestává stačit velikost pole `x` a chceme ho zvětšit o dalších deset prvků.

```
p_nove = (int *) malloc((pocet + 10) * sizeof(int));
p_pom1 = x; p_pom2 = p_nove;
while (p_pom1 < x + pocet) /* kopirovani stareho pole */
 *p_pom2++ = *p_pom1++; /* na novou adresu */
pocet += 10;
free((void *) x); /* uvolneni stareho pole */
x = p_nove;
```

Poznámky:

- Překopírování starého pole je časově náročné, takže než se do této strategie pustíme, je nutné se rozhodnout, zda potřebujeme spíše rychlost nebo šetření paměti, popřípadě udělat vhodný kompromis mezi nimi, což znamená toto pole používat, ale neměnit jeho velikost příliš často.
- Standardní knihovna také pamatuje na možnost práce s polem o proměnné délce. Dává nám k dispozici funkci `realloc()`. Ta je popsána v hlavičkovém souboru `stdlib.h` takto:

```
void *realloc(void *pole, unsigned int size);
```

kde `pole` je pointer na již dříve alokovanou oblast paměti a `size` je počet Bytů nově požadovaného pole.

<sup>16</sup> Vytvořené pomocí `malloc()`.

<sup>17</sup> Zde, pro lepší názornost operací s indexy, porušíme zásadu, že identifikátor pointeru začíná pomocí `p_`. Pro jednoduchost zde také netestujeme úspěšnost přidělení paměti.

`realloc()` upravuje velikost alokované paměti na hodnotu `size` — při zmenšování, nebo alokuje jinou větší oblast paměti a původní paměť do ní překopíruje — při zvětšování — a pak uvolní.

`realloc()` vrací pointer na nově alokovanou paměť nebo nulový pointer `NULL`, pokud nelze pole realokovat.

## 11.4 Pole jako parametry funkcí

Pole může být samozřejmě parametrem funkce. Skutečný parametr je pak do funkce předáván odkazem<sup>18</sup>, tzn. předá se adresa začátku pole pomocí pointeru, tj. pomocí jména pole. To má ovšem ten význam, že položky pole mohou být ve funkci měněny a tuto změnu si ponechají i po opuštění funkce.

Příklad:

Následující funkce nalezne největší prvek z pole o `ROZSAH` prvcích.

```
double maxim(double pole[])
{
 double *p_max = pole,
 *p_pom;

 for (p_pom = pole + 1; p_pom < pole + ROZSAH; p_pom++) {
 if (*p_pom > *p_max)
 p_max = p_pom; /* zmena pointeru na max. prvek */
 }
 return (*p_max);
}
```

Pole jako formální parametr je specifikováno jako identifikátor následovaný prázdnými závorkami `[]` neboli: `double pole[]` přičemž je nutno poznamenat, že tentýž význam by měla i specifikace pomocí pointeru: `double *pole`

Poznámka:

- Prvnímu způsobu se však dává často přednost, protože je pak jasnější, že se jedná o pole typu `double` a ne o pointer na `double`<sup>19</sup>.

Volání funkce by bylo např.: `max = maxim(pole_a);`

<sup>18</sup> Tedy ne hodnotou, jak jsme byli dosud zvyklí u běžných proměnných.

<sup>19</sup> Což má ovšem, jak již víme, naprosto stejný význam a práce s oběma je zcela stejná.



kdy skutečný parametr `pole_a` říká pouze: "od symbolické adresy `pole_a` začíná pole s prvky typu `double`".

Z výše uvedeného vyplývá fakt, že pokud je ve funkci pracující s polem nutno znát jeho velikost, která není konstantní, pak se tato velikost musí předat jako další formální parametr. Z pouhého skutečného parametru jména pole není překladač totiž schopen velikost pole zjistit. Hlavička funkce by v tomto případě vypadala např. takto:

```
double maxim(double pole[], int pocet)
```

Tělo funkce by bylo zcela stejné, jen nový formální parametr `pocet` by v cyklu `for` nahradil symbolickou konstantu `ROZSAH`.

Jak totiž již bylo řečeno dříve:

|                               |                                                    |
|-------------------------------|----------------------------------------------------|
| <code>sizeof(pole)</code>     | vrací velikost pointeru na typ <code>double</code> |
| <code>sizeof(*pole)</code>    | vrací velikost proměnné typu <code>double</code>   |
| <code>sizeof(double)</code>   | vrací velikost typu <code>double</code>            |
| <code>sizeof(double *)</code> | vrací velikost pointeru na typ <code>double</code> |

Prakticky to znamená, že pole jako skutečný parametr ztrácí ve funkci statut pole a jeho velikost — ať již se jedná o statické nebo dynamické pole — se nedá ve funkci nijak zjistit.

#### Poznámka:

- Častý, ale chybný, pokus o předání pole a současně jeho délky jako jediného formálního parametru je:

```
double maxim(double pole[10])
```

Hodnota 10 zde nemá žádný význam a překladač ji ignoruje. Formální parametr `pole` bude i nadále považován za: `double pole[]`

Další důležitou odlišností formálního parametru pole od skutečného pole je to, že formální parametr není konstanta<sup>20</sup>, ale je to *l-hodnota*. To znamená, že ho lze ve funkci změnit<sup>21</sup>, protože je vytvořena jeho lokální kopie ve stacku, přičemž nezáleží na tom, zda bude skutečný parametr statické nebo dynamické pole.

#### Příklad:

Následující program ukazuje využití dříve definované funkce `maxim()`

```
#include <stdio.h>
#define POCET 10
```

<sup>20</sup> Není to konstantní pointer.

<sup>21</sup> Změna ve funkci se ale neprojeví jako změna skutečného parametru. Tuto možnost určitě nebudeme používal příliš často.

```
/* funkční prototyp funkce maxim() je nutný - maxim() vrácí
 typ double */
double maxim(double pole[], int pocet);

main()
{
 double f[POCET];
 int i;

 for (i = 0; i < POCET; i++) {
 printf("Zadej %d. číslo : ", i + 1);
 scanf("%lf", &f[i]);
 }
 printf("Max. z %d čísel je %f \n", POCET, maxim(f, POCET));
}
```

#### Poznámka:

- Z faktu, že se funkci předává pomocí skutečného parametru pouze adresa začátku pole, vyplývá, že jde velmi snadno funkci použít i pro práci s úseky tohoto pole. Pokud bychom si někdy přáli najít maximum pouze z výseku pole, pak lze jen jednoduše určit meze, např. pro maximum ze třetího až sedmého prvku pole je volání: `max = maxim(f + 2, 5);` nebo: `max = maxim(&f[2], 5);`

#### Příklad:

Přepíšeme-li funkci `maxim()` jako proceduru `maxim()`, pak je nutno maximální hodnotu vrátit pomocí jednoho z parametrů, který musí být samozřejmě volán odkazem.

```
void maxim(double pole[], int pocet, double *p_max)
{
 double *p_pom;

 *p_max = pole[0];
 for (p_pom = pole + 1; p_pom < pole + pocet; p_pom++) {
 if (*p_pom > *p_max)
 *p_max = *p_pom; /* změna hodnoty na adrese p_max */
 }
}
```

Zde je pointer `p_max` formální parametr volaný odkazem, takže se do funkce `maxim()` předává adresa proměnné, na kterou `maxim()` uloží nalezenou



hodnotu největšího prvku pole.

Častou chybou je ve funkci `maxim()` přiřazení:

```
p_max = p_pom; /* místo *p_max = *p_pom; */
```

Pak by se totiž ztratila adresa proměnné, do které má být uložena hodnota maximálního prvku pole. Nic by se sice nezniciilo, protože se pouze ve stacku přepíše lokální kopie adresy skutečného parametru. Protože tato lokální kopie zaniká po opuštění funkce, nic dalšího se nestane, a tedy procedura `maxim()` nebude mít žádný účinek.

Hlavní funkce by pak pro proceduru `maxim()` vypadala následovně:

```
main()
{
 double f[POCET], max;
 int i;

 for (i = 0; i < POCET; i++) {
 printf("Zadej %d. cislo : ", i + 1);
 scanf("%lf", &f[i]);
 }

 maxim(f, POCET, &max);
 printf("Maximum z %d cislic je %f \n", POCET, max);
}
```

#### Příklad:

Program ukazuje další častou chybu s předáváním adresy neexistujícího pole. Funkce `init()` načte do pole 5 `double` čísel a vrátí adresu tohoto pole.

```
void init(double **p_f)
{
 double a[5];
 int i;

 for (i = 0; i < 5; i++) {
 printf("Zadej %d. cislo : ", i + 1);
 scanf("%lf", &a[i]);
 }

 *p_f = a;
}
```

```
main()
```

```
{
 double *p_dbl;

 init(&p_dbl);
 ...
}
```

Volání `init()` způsobí, že pointer `p_dbl` bude skutečně ukazovat na pole pět `double` čísel načtených z klávesnice. Problém je však v tom, že toto pole bylo vytvořeno ve stacku po vstupu do funkce `init()`<sup>22</sup> a tento stack je po návratu z funkce `init()` vrácen zpět systému. Pointer `p_dbl` tedy ukazuje na blok paměti, který už nám nepatří a který může být kdykoliv legálně přepsán.

Pokud bychom použili tento nepříliš šťastný způsob řešení<sup>23</sup>, pak by bylo nutné ve funkci `init()` alokovat pole `a[]` pomocí funkce `malloc()` — tedy trvale.

## 11.5 Pole pointerů na funkce

Tak, jako může být pole složeno z jednoduchých proměnných, mohou být prvky pole i pointery. Pokud jsou to pointery opět na jednoduché proměnné, pak se většinou jedná o vícerozměrná pole, která budou probírána v dalších kapitolách. Zvláštním a občas využívaným polem pointerů je pole pointerů na funkce. Všechny funkce<sup>24</sup> musí být samozřejmě stejného typu a pole pointerů na ně se definuje takto:

```
typedef void (* P_FCE)(); /* definice pointeru na funkci
 vracející typ void */
P_FCE funkce[10]; /* definice pole 10-ti pointerů */
```

Toto pole je pak nutné naplnit adresami existujících funkcí, což se dělá na prostě stejně jako při přiřazování adresy funkce do pointeru na funkci — viz str. 158.

Možná praktická aplikace pole pointerů na funkce, je program řízený pomocí menu. Adresy jednotlivých funkcí provádějících příslušné příkazy menu jsou uloženy v poli a odtud mohou být přímo volány pomocí indexu.

<sup>22</sup> Jako pole automatických proměnných.

<sup>23</sup> Lepší je mít pole `a[5]` definované ve funkci `main()`.

<sup>24</sup> Jejichž adresy jsou prvky tohoto pole.



Využijeme-li předchozí definice nového typu `P_FCE`, pak je možné definovat<sup>25</sup> pole pointerů na funkce včetně jeho inicializace.

`P_FCE funkce[] = {file, edit, search, compile, run};`  
kde identifikátory `file`, `edit`, `search`, ... jsou názvy jednotlivých funkcí.

Volání funkce<sup>26</sup> je pak: `(* funkce[1])()`;

nebo — jen v ANSI C: `funkce[1]()`;

V našem případě by bylo pravděpodobně ještě definováno pole přístupových znaků, ve kterém by se hledal příslušný jednopísmenný příkaz, a podle jeho indexu by se volala pomocí téhož indexu i příslušná funkce.

Takhle by vypadala definice pole přístupových znaků včetně inicializace:

`char prikaz[] = {'F', 'E', 'S', 'C', 'R'};`

#### Poznámky:

- Toto je jeden ze způsobů inicializace řetězce — další viz na str. 195. Tento inicializační příkaz je v tomto případě přehlednější<sup>27</sup> než obvyklá inicializace: `char prikaz[] = {"FESCR"};`
- Asi už vás správně napadlo, že by se tento problém dal také — a pravděpodobně jednodušeji a přehledněji — řešit pomocí přepínače `switch` bez jakéhokoliv použití pointerů na funkce. To je pravda, ale možná, že se někdy setkáte s problémem, kdy se vám bude pole pointerů na funkce hodit.

## 11.6 Jak číst komplikované definice — II.

Na str. 160 jsme se dozvěděli, jakým způsobem můžeme přečíst libovolně komplikovanou definici. Tehdy jsme ještě neznali pole, které do definic vnáší další komplikace. Ovšem není třeba se jich obávat. Postup čtení, který již známe, platí naprosto stejně i pro pole, jen je třeba pole do tohoto čtení začlenit.

Příklady zápisů definic pomocí pointerů a polí.

`double (* f[])();` `f` je pole pointerů na funkce vracející typ `double`

`double (* f())[];` `f` je funkce vracející pointer na pole prvků typu `double`

`double *(f[])();` `f` je pole funkcí vracející pointer na typ `double`  
*Pozor!* Tato možnost v C neexistuje.

<sup>25</sup> Musí to být pole globální nebo statické lokální — viz též str. 223.

<sup>26</sup> Což je v našem případě volání funkce `edit()`.

<sup>27</sup> Protože to ale není řetězcová konstanta, pole `prikaz` bude mít velikost 5 prvků a jeho poslední prvek bude znak `'R'` — ne znak `'\0'`. Ale to již hodně předbíháme.

`double *f()[];` `f` je funkce vracející pole pointerů na typ `double`  
*Pozor!* Tento zápis je opět nemožný.

Pro ukázkou, jak se tyto zápisy čtou, budeme zkoumat uvedený příklad definice funkce vracející pointer na pole prvků typu `double`:

`double (* f())[];`

Tak jako na str. 160 postupujeme následujícím způsobem:

- Ve spleti kulatých závorek, hranatých závorek a hvězdiček se nalezne identifikátor, tedy `"f"`  
a řekneme: `f` je ...
- Od něho se čte doprava, dokud nenarazíme na prázdné kulaté závorky `"()`"<sup>28</sup>  
a přidáme: ... *funkce vracející* ...
- Čteme dále doprava a pravá kulatá závorka nás vrací doleva až na odpovídající levou kulatou závorku `"("` a od ní se čte zase doprava, tedy `"*"`  
a přidáme: ... *pointer na* ...
- Přeskakujeme jméno proměnné, prázdné kulaté závorky a pravou kulatou závorku `)`, které nám už posloužily, a čteme stále doprava, dokud nenarazíme pravou kulatou závorku `)` nebo na středník `;`, které nás vždy vrací doleva. Přečteme tedy prázdné hranaté závorky `[]`<sup>29</sup>  
a přidáme: ... *pole prvků typu* ...
- Čteme stále doprava až nás ukončovací středník vrátí doleva před již zpracovanou `"("`. Nyní čteme opačně<sup>30</sup> — doleva — tedy `"double"`  
a dodáme: ... *double* ... a jsme hotovi.

Výsledek "průzkumu" tedy dohromady zní:

`f` je funkce vracející pointer na pole prvků typu `double`

Když setřeme pot z čela a zkontrolujeme výsledek s příklady nahoře, tak vidíme, že opět vítězíme.

#### Časté chyby:

`int b[3];`

`b[3] = 5;` rozsah `b` je od 0 do 2

`int b[3];`

`b = 5;` `b` je konstantní pointer, který nelze změnit

<sup>28</sup> Dvojice závorek `"()`" jako obvykle označuje funkci.

<sup>29</sup> Dvojice závorek `"[]"` jako obvykle označuje pole.

<sup>30</sup> Kdyby nás sem vrátila `"("`, pak bychom četli doprava, jako v předchozích bodech.



```
int b[3];
x = &b;
```

**b** není l-hodnota (**b** představuje přímo adresu)

```
f(float b[])
{
 ...
 sizeof(b) sizeof nevrátí rozměr pole b, ale
 velikost pointeru na float
}
```

#### Co je dobré si uvědomit:

- Identifikátor pole je konstantní pointer.
- Identifikátor pole jako formální parametr je proměnný pointer.
- Je-li **x** jednorozměrné pole a je-li předáváno jako parametr, pak je deklarace v hlavičce funkce: `int x[]31` nebo `int *x`

#### Cvičení:

- 1) Zjistěte počet jednotlivých písmen v souboru. Zjištěné počty vypište jak číslem, tak i pomocí řádkového histogramu, např.  
 A: 5 \*\*\*\*\*  
 B: 11 \*\*\*\*\*
- 2) Napište funkci `void ahoj(void);`, která vytiskne na obrazovku číslo a slovo `ahoj`, přičemž číslo bude pořadové číslo volání funkce `ahoj()`. Dále definujte proměnnou `p_ahoj` jako pointer na funkci `ahoj()`. V cyklu volejte funkci `ahoj()` pomocí pointeru `p_ahoj`.
- 3) V souboru `POZDRAVY.C` napište několik funkcí typu `void ahoj(void)`, které vytisknou vždy jeden příslušný pozdrav (`ahoj`, `nazdar`, ...). V souboru `HLAVNI.C` definujte pole pointerů na tyto funkce a inicializujte ho adresami těchto funkcí. Dále definujte pole znaků, které inicializujete počátečními (nebo přístupovými) znaky jednotlivých pozdravů. Vytvořte program, který bude číst znaky z klávesnice, rozpozná je a vytiskne příslušný pozdrav pomocí pointeru na funkci. Nepoužívejte přepínač.
- 4) Napište program, který zjistí počet jednotlivých písmen na každé řádce čteného souboru. Výsledky vytiskněte v přehledné tabulce, kde ve vodorovném směru budou písmena (malá a velká nerozlišujte) a ve svislém směru čísla řádek. Zpracujte pouze prvních 20 řádek souboru.

<sup>31</sup> V tomto případě může být uvedena velikost pole: `(int x[5])` ale není na ni brán zřetel.

- 5) Napište funkci `serad(int x[], int y[], int pocet);`, která seřadí pole **x** podle velikosti vzestupně do pole **y**.
- 6) Napište funkci `int sude(int x[], int y[], int pocet);`, která zkopíruje z pole **x** do pole **y** pouze sudé prvky a vrátí počet prvků pole **y**.



## 12 Řetězce

Řetězec (*string*) je speciální typ jednorozměrného pole. Je vždy složen z prvků typu `char` a v podstatě se chová jako každé jiné jednorozměrné pole, ale s jedním malým dodatkem. Protože je práce s řetězcem v programech velmi častá, je řetězcům v C věnována tato speciální kapitola. V ní se nedozvíte žádné převratné novinky oproti již probraným jednorozměrným polím, ale spíše větší množství drobných, leč užitečných informací, které se mohou hodit.

### Poznámka:

- Řetězcové konstanty, které byly vysvětleny na str. 29, zde nebudou znovu vysvětlovány, ale budou samozřejmě používány.

### 12.1 Základní informace a definování řetězců

Podstatná informace, na kterou při práci s řetězcem nesmíme nikdy zapomenout, je, že řetězec je vždy ukončen znakem `'\0'`<sup>1</sup>. Podle tohoto znaku se tedy pozná délka řetězce. Tento fakt má následující důsledky:

- Řetězec může mít libovolnou délku, omezenou pouze velikostí paměti. Z této celkové přidělené paměti je ale "aktivní" (právě využitá) jen její část od začátku až do prvního znaku `'\0'`. Veškeré další informace uložené až za `'\0'` jsou při standardním zpracování řetězců nedostupné<sup>2</sup>, protože práce s řetězcem končí vždy dosažením prvního znaku `'\0'`.
- Při definování řetězce, tedy alokování místa pro něj, musíme alokovat o jeden Byte více, právě pro tuto `'\0'`.
- Pokud zapomeneme na konec řetězce dát znak `'\0'` nebo tento znak omylem přepíšeme, považuje se za řetězec celá následující oblast paměti tak dlouho, dokud se někde dále v paměti tento znak neobjeví<sup>3</sup>.

<sup>1</sup> Znak s ASCII hodnotou nula. Doporučuje se nepoužívat prostou 0, i když je to pro počítač stejné. Použití znakové nuly `'\0'` zvyšuje čitelnost programu.

<sup>2</sup> Lze je ale využít, pokud se na řetězec díváme jako na jednorozměrné pole prvků typu `char`.

<sup>3</sup> To samozřejmě většinou vede k chybné funkci programu, pokud do této paměti zapisujeme.

Předpokládejme, že řetězec `str` má délku 10 Byte a je v něm uložen text `ahoj`. Pak je možné se podívat, jak je tento řetězec uložen v paměti:

|          |   |   |   |   |    |   |   |   |   |
|----------|---|---|---|---|----|---|---|---|---|
| str      | a | h | o | j | \0 |   |   |   |   |
| posunutí | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 |

volná paměť

Kromě této zvláštnosti je řetězec normální jednorozměrné pole, složené ovšem vždy z prvků typu `char`. Z toho, co již víme o jednorozměrných polích, je možné usoudit, že lze vytvořit řetězec staticky nebo dynamicky, což je naprosto správná úvaha.

Dále bude uveden příklad vytvoření řetězce o 10-ti prvcích, přičemž je nutné si uvědomit, že:

- Nejdelší text, do tohoto řetězce uložený, má délku 9 znaků.
- Pokud budeme přistupovat k jednotlivým znakům pomocí indexace, pak nejvyšší index je 8. Protože prvek s indexem 9 je právě ukončovací `'\0'`, a prvek s indexem 10 už neexistuje<sup>4</sup>.

|                               |                                           |
|-------------------------------|-------------------------------------------|
| <i>staticky</i>               | <i>dynamicky</i>                          |
| <code>char s_stat[10];</code> | <code>char *s_dyn;</code>                 |
|                               | <code>s_dyn = (char *) malloc(10);</code> |

Oba řetězce (`s_stat` a `s_dyn`) si jsou dále naprosto rovnocenné.

### Poznámka:

- Důrazně upozorňujeme, že pro dynamický řetězec nestačí jen definovat pointer, ale je nutný i další krok — alokace paměti. Velmi častou chybou je, že se na tuto alokaci zapomene, a pak se program samozřejmě hroutí, protože pracujeme s pamětí, která není naše.

Pokud definujeme statický řetězec, často se využívá možnost jej současně inicializovat. Inicializace je velmi jednoduchá, neboť se použije již známá řetězcová konstanta:

`char s1[10] = "ahoj";`      *definice s inicializací*

Možnosti inicializace jdou ještě dále, takže když potřebujeme definovat řetězec dlouhý "právě akorát" pro určitý text, nemusíme pracně počítat jednotlivé znaky. To za nás udělá překladač, který zároveň alokuje potřebné místo v paměti.

`char s2[] = "nazdar";`      *definice s inicializací bez udané délky řetězce*

<sup>4</sup> Viz též "chyba +1" — str. 176.



Poznámky:

- Pozorného čtenáře již určitě napadlo, co se v tomto případě děje s ukončovací nulou. Potěšující zpráva je, že ji překladač na konec řetězcové konstanty<sup>5</sup> doplní sám. Pro případ řetězce `s2` to tedy znamená, že bude mít délku 7 Byte.
- Narozdíl od Pascalu není možné přiřadit statickému řetězci konstantu, např.:

```
char str[10];
str = "ahoj"; /* nelze */
```

Adresa pole `str` totiž není l-hodnota a nemůže se objevit na levé straně přiřazovacího příkazu.

Možná, že jste již někde v programu viděli podobnou definici:

```
char *str = "Ahoj";
```

a teď vám vrtá hlavou, jak je možné inicializovat řetězec, když pro něj není nikde alokováno místo. Nemějte obavy, vše je v naprostém pořádku, protože `str` zde nepředstavuje dynamický řetězec, ale pointer na typ `char`<sup>6</sup>. A tento pointer je inicializován adresou řetězcové konstanty, která má obsah `Ahoj`. Pro tuto konstantu překladač vyhradil místo v datové paměti a samozřejmě ji nezapomněl ukončit znakem `'\0'`. V tomto případě se žádná dynamická paměť pro řetězec `str` nealokuje.

Ještě více pozorného čtenáře teď asi napadlo, jak se tedy inicializuje dynamicky vytvořený řetězec. Prozradíme dopředu, že to nejde, protože inicializace se provádí v době překladu a alokace dynamické paměti v době běhu programu. Paměť, která byla řetězci dynamicky přidělena pomocí funkce `malloc()`, musíme sami v programu naplnit. V následujících dvou způsobech je ten vlevo špatně a ten vpravo dobře, ačkoliv program by fungoval pro oba dva.

|                                           |                                           |
|-------------------------------------------|-------------------------------------------|
| <code>/* chybná verze */</code>           | <code>/* správná verze */</code>          |
| <code>char *s_dyn;</code>                 | <code>char *s_dyn;</code>                 |
| <code>s_dyn = (char *) malloc(10);</code> | <code>s_dyn = (char *) malloc(10);</code> |
| <code>s_dyn = "Ahoj";</code>              | <code>strcpy(s_dyn, "Ahoj");</code>       |

Co se vlastně stalo? V obou případech jsme správně definovali pointer na `char`, alokovali jsme 10 Byte v dynamické paměti a tuto adresu jsme přiřadili do `s_dyn`. V příkladu nalevo jsme v dalším příkaze přiřadili do `s_dyn` adresu

<sup>5</sup> Dodejme, že řetězcové konstanty jsou konstantní pointery do paměti.

<sup>6</sup> Chcete-li pak *pointer na řetězec*.

řetězcové konstanty `Ahoj`. Od této chvíle `s_dyn` skutečně ukazuje na tento řetězec, ale nenávratně jsme ztratili adresu původně alokované dynamické paměti! Pokud budeme chtít tento řetězec měnit, bude to možné, ale pozor — budeme měnit řetězcovou konstantu, pro kterou bylo přiděleno pouze 5 Byte paměti. Je velká pravděpodobnost, že budeme využívat celých 10 Byte v blahé víře, že jsou naše, což samozřejmě způsobí problémy. Je možné, že se tyto problémy neprojeví zhroucením programu, ale pouze jeho špatnou funkcí. Řetězcová konstanta je totiž uložena v datové oblasti společně s dalšími daty programu. Podaří se nám tedy zřejmě přepsat data a ne kód programu.

V druhém případě jsme správně využili standardní funkci pro práci s řetězci — viz dále, která provedla zkopírování řetězcové konstanty `Ahoj` do alokované paměti znak po znaku.

Znamená to, že nyní je text `Ahoj` v paměti dvakrát. Jednou ve statické paměti jako řetězcová konstanta, použitá jako parametr funkce `strcpy()`, a podruhé v dynamické paměti jako obsah řetězce `s_dyn`. Oba tyto texty jsou ale na sobě zcela nezávislé a text v řetězci `s_dyn` můžeme libovolně měnit a hlavně můžeme využívat všech alokovaných 10 Byte.

Poznámky:

- Občas se pro definici řetězců používá nový typ:  
`typedef char *STRING;`
- Je nutné rozlišovat *nulový pointer* a *nulový řetězec*. Nulový pointer má hodnotu `NULL` (což je většinou 0), zatímco nulový řetězec je řetězec, jehož první znak je `'\0'`, např.:

```
char *p_null_point, *p_null_str;
p_null_point = NULL;
p_null_str = (char *) malloc(10);
p_null_str[0] = '\0';
```

- Pozor na rozdíl mezi: `"x"` a `'x'`  
`"x"` je řetězcová konstanta délky 2 Byte (znak `'x'` + znak `'\0'`)  
`'x'` je znaková konstanta délky 1 Byte
- Je asi zbytečné zdůrazňovat, že všude, kde se objevují řetězcové konstanty (texty v uvozovkách), lze použít i řetězec, např.:  
`strcpy(s1, "POKUS.TXT");`  
`fr = fopen(s1, "r");`



## 12.2 Práce s řetězcem

Protože již umíme pracovat s poli a víme, že práce s řetězcem je v podstatě stejná, omezíme se v dalším textu pouze na komentované ukázky jednotlivých akcí.

### 12.2.1 Čtení řetězce z klávesnice

Nejčastěji se provádí pomocí funkce `scanf()`<sup>7</sup>. Příkaz vypadá takto:

```
scanf("%s", s1);
```

#### Poznámky:

- Formát pro čtení řetězce je `"%s"`, stejně tak jako pro tisk řetězce.
- Příkaz přečte řetězec zadaný z klávesnice tím způsobem, že přeskočí všechny bílé znaky, přečte řetězec, který je ukončen bílým znakem a uloží ho na adresu `s1`.  
Ukončovacím bílým znakem je nejčastěji klávesa <Enter>, ale je dobré si uvědomit, že bílý znak je i mezera. To tedy prakticky znamená, že kdybychom zadali na vstupní řádce text " ahoj evo ", uložilo by se do řetězce `s1` pouze slovo "ahoj" a slovo "evo" by zůstalo v bufferu klávesnice. Toto je velmi častá chyba<sup>8</sup>. Potřebujeme-li přečíst celou řádku, nehledě na mezery, je nutné použít např. funkci `gets()` — viz str. 206.
- Možná, že vás při podrobném prohlížení příkazu `scanf()` napadlo, že tam chybí operátor `&`. To je správný postřeh, ale špatná úvaha. Operátor `&` tam v tomto případě nebude, protože `s1` představuje sám o sobě adresu<sup>9</sup>.

#### Čtení řetězce v daném formátu

Tak, jako je možné zadat formát výpisu, je možné zadat i formát čtení, i když se to nepoužívá příliš často.

Následující program ukazuje, jak číst řetězce v určitém formátu. Program řeší situaci, kdy existuje soubor `PENIZE.TXT`. Do tohoto souboru byly průběžně zapisovány denní příjmy a výdaje v amerických dolarech. O tomto souboru víme jen to, že obsahuje celá čísla a že každé toto číslo uvozuje znak "\$". Dále víme, že příjem je uvozen znakem "+" a výdaj znakem "-",

přičemž není vždy splněno, že znak znaménka bezprostředně předchází číslu. Řádka v souboru `PENIZE.TXT` může vypadat např. takto:

```
+ $10 -$5- $8 +$20
```

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

```
main()
```

```
{
```

```
 long int suma = 0;
```

```
 FILE *fr;
```

```
 int kolik;
```

```
 char akce[2];
```

```
 fr = fopen("PENIZE.TXT", "r");
```

```
 while (fscanf(fr, "%1s", akce) != EOF) {
```

```
 fscanf(fr, "%d", &kolik);
```

```
 suma += (akce[0] == '+') ? kolik : (-1 * kolik);
```

```
 }
```

```
 printf("Celkem: %ld \n", suma);
```

```
 fclose(fr);
```

```
}
```

#### Poznámky:

- Na konci souboru vrací `scanf()` hodnotu EOF.
- Je důležité, aby operátor `oper` byl čten ve formátu `"%1s"`<sup>10</sup>. Byl-li by totiž vstup např.: " -\$5" pak čtení např. pomocí: `fscanf(fr, "%c", &c)` by přečetlo do znakové proměnné `c` první mezeru před znakem "-".
- Jiný formát, např.: `fscanf(fr, "%s", akce)` přečte do řetězce `oper` znaky "-", "\$" a "5" a ukončí je '\0', čímž jednak přeteče pole `akce` vyhrazené jen pro jeden znak a jednak se nenačte číslo 5 do proměnné `kolik`.
- Řetězec `akce` má délku 2, protože je vždy nutno počítat s místem pro ukončující nulový znak.
- Mezera ve formátu " %d" je velmi důležitá, protože říká, že se mají přeskočit všechny předcházející bílé znaky.
- Formát u `scanf()` umožňuje mnohá další kouzla. Pokud je budete chtít objevovat, doporučuje se dělat co nejjednodušší pokusy.

<sup>10</sup> Čili řetězec délky jeden znak.

<sup>7</sup> Další možnosti viz str. 206.

<sup>8</sup> Ovšem neznamená to, že by v řetězci nemohla být mezera. Znamená to pouze, že `scanf()` neumí mezeru načíst.

<sup>9</sup> Viz též str. 178.



### 12.2.2 Tisk řetězce na obrazovku

Nejčastěji se provádí pomocí funkce `printf()`, ale jsou možné další způsoby, které jsou uvedeny na str. 207.

Příkaz vypadá takto: `printf("%s", s1);`  
a tiskne znaky od adresy `s1` až do okamžiku dokud nenarazí na ukončovací znak `'\0'`.

#### Poznámka:

- Čtenáře občas napadne, že když jsme ve `scanf()` vynechali operátor `&`, měli bychom v `printf()` přidat dereferenční operátor `*`, tedy:

```
printf("%s", *s1);
```

Tato úvaha opět není správná. Formát `"%s"` totiž funkci `printf()` říká, že má očekávat adresu a nikoliv hodnotu.

Výsledkem těchto komplikací budiž tvrzení, že se řetězec čte jinak, než jednoduchá proměnná, ale tiskne se úplně stejně jako jakákoliv jiná proměnná.

#### Příklad:

Program přečte řetězec ne delší než 10 znaků a vypíše ho na obrazovku.

```
#include <stdio.h>
main()
{
 char str[11];
 printf("Zadej řetězec : ");
 scanf("%s", str);
 printf("Řetězec je : %s \n", str);
}
```

Definovaná délka řetězce je 11, ale řetězec nesmí přesáhnout délku 10-ti znaků, protože pak bude přepisovat nepřidělenou paměť, ležící za `str`. V těchto případech je vhodnější číst řetězec v omezeném formátu, např.:

```
scanf("%10s", str);
```

#### Příklad:

Pro dynamický řetězec bude program vypadat takto:

```
#include <stdio.h>
#include <stdlib.h>
main()
```

```
{
 char *str;

 if ((str = (char *) malloc(11)) == NULL) {
 printf("Malo pameti \n");
 return;
 }
 printf("Zadej řetězec : ");
 scanf("%s", str);
 printf("Řetězec je : %s \n", str);
}
```

### 12.2.3 Přístup k jednotlivým znakům řetězce

Často se nám stane, že potřebujeme pracovat s jednotlivými znaky řetězce<sup>11</sup>. Jak už jsme několikrát zdůrazňovali, řetězec je normální jednorozměrné pole, takže přístup k jeho jednotlivým prvkům (znakům) by nám neměl dělat žádné problémy.

#### Příklad:

Následující část programu vyplní `s1` hvězdičkami.

```
for (i = 0; i < 10 - 1; i++)
 s1[i] = '*';
s1[10 - 1] = '\0'; /* ukončení řetězce */
```

#### Poznámka:

- U předchozího příkladu si všimněte posledního řádku. Na ukončovací znak se nesmí nikdy zapomenout!

Časté chyby tedy jsou:

```
for (i = 0; i < 10; i++) for (i = 0; i < 10; i++)
 s1[i] = '*'; s1[i] = '*';
 s1[10] = '\0';
```

Kdy jsme v prvním případě zapomněli na ukončovací znak zcela a ve druhém případě jsme ho zapsali do oblasti paměti, která už není naše.

Je-li řetězec definován jako: `char s[MAX];`

pak využitelná délka je: `MAX - 1`

<sup>11</sup> Pokud nevyužíváme funkcí ze standardní knihovny (viz dále), pak je to vlastně nezbytnost.



### 12.2.4 Standardní funkce pro práci s řetězci

Zde je opět nutné zdůraznit, že C nedefinuje práci s řetězci jako součást jazyka. Protože jsou ale práce s řetězci velmi časté, poskytuje jazyk C díky své standardní knihovně množství funkcí, pomocí nichž se pracuje s řetězci snadno a rychle<sup>12</sup>.

Chceme-li tyto funkce využívat, je nutné připojit do našeho programu standardní hlavičkový soubor `string.h` příkazem:

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

Po tomto příkazu můžeme využívat množství funkcí, z nichž dále uvedeme jen ty nejdůležitější vždy formou úplného funkčního prototypu, stručného popisu a příkladu použití.

#### Délka řetězce

```
int strlen(char *s);
```

Vrací délku řetězce `s` bez ukončovacího znaku `'\0'`.

Např.: `strlen("ahoj")` vrátí hodnotu 4

#### Kopírování řetězce

```
char *strcpy(char *s1, char *s2);
```

Zkopíruje obsah řetězce `s2` do `s1`. Vrací pointer na první znak řetězce `s1`.

Např.: `strcpy(str, "ahoj");` v `str` bude "ahoj"

#### Spojení řetězců

```
char *strcat(char *s1, char *s2);
```

Připojí `s2` k řetězci `s1`. Vrací pointer na první znak řetězce `s1`.

Např.: `strcat(str, " + nazdar");` v `str` bude "ahoj + nazdar"

#### Nalezení znaku v řetězci

```
char *strchr(char *s, char c);
```

Pokud se znak v proměnné `c` vyskytuje v řetězci `s`, pak je vrácen pointer na jeho první výskyt, jinak je vrácena hodnota `NULL`.

Např. z předchozího řetězce: `strchr(str, 'x')` vrátí `NULL`

#### Porovnání dvou řetězců

```
int strcmp(char *s1, char *s2);
```

Vrátí 0, jsou-li řetězce `s1` a `s2` stejné. Vrací záporné číslo, je-li `s1` lexikograficky menší než `s2` a kladné číslo v opačném případě.

<sup>12</sup> V tomto případě to není reklamní trik, protože funkce jsou optimalizovány, takže jsou skutečně rychlejší, než když si napíšete vlastní.

### Nalezení podřetězce v řetězci

```
char *strstr(char *s1, char *s2);
```

Nalezne první výskyt řetězce `s2` v řetězci `s1` a vrátí pointer na tento výskyt, nebo vrátí `NULL` v případě neúspěchu.

### Práce s omezenou částí řetězce

Ve standardní knihovně jsou také implementovány funkce, které nemusí pracovat s celým řetězcem (tedy až do ukončujícího znaku `'\0'`), ale pouze se zadaným počtem jeho prvních znaků. Tyto funkce vypadají podobně, jako funkce výše uvedené, jen s tím rozdílem, že mají v názvu písmeno "n" — jako *number*. Druhým rozdílem je samozřejmě další formální parametr navíc, pomocí něhož se předává maximální počet zpracovávaných znaků řetězce.

Funkce zpracovávají řetězec buď do určené délky — je-li řetězec delší než požadovaná délka zpracování, nebo do ukončujícího znaku `'\0'` — je-li řetězec ve skutečnosti kratší než požadovaná délka zpracování.

#### Příklad:

Jako příklad těchto funkcí uvedeme funkci, která zkopíruje nejvýše `max` znaků z `s2` do `s1` (znak `'\0'` přidá jen pro `max > strlen(s2)`)

```
char *strncpy(char *s1, char *s2, int max);
```

Po příkazu: `strncpy(str, "alkoholické", 7);`

bude v `str` řetězec "alkohol" neukončený znakem `'\0'`

### Práce s řetězcem pozpátku

Další množina funkcí pracuje s řetězci "odzadu". Tyto funkce se opět velmi podobají příslušným základním funkcím, s tím rozdílem, že mají ve svém názvu písmeno "r" — jako *reverse*. Rozdíl v jejich práci je, že řetězec nezpracovávají od počáteční adresy řetězce, ale od adresy jeho koncového znaku `'\0'` směrem k počátku řetězce.

#### Příklad:

Jako příklad těchto funkcí uvedeme funkci, která hledá znak v proměnné `c` v řetězci `str` od jeho konce. Pokud se znak v proměnné `c` vyskytuje v řetězci `str`, pak je vrácen pointer na jeho poslední výskyt, jinak je vráceno `NULL`.

```
char *strrchr(char *str, char c);
```

### Převody řetězců na čísla

V programu často potřebujeme zkonvertovat číselnou hodnotu proměnné na odpovídající řetězec číslic či naopak. Pokud se to děje v rámci vstupů nebo výstupů, nemusíme si s tímto problémem lámat hlavu, protože např. funkce `scanf()` a `printf()` to vlastně dělají automaticky.



Nejedná-li se o vstupy a výstupy, můžeme použít několik funkcí, které převádí řetězec na číslo<sup>13</sup>. Poznamenejme, že funkční prototypy těchto funkcí nejsou v souboru `string.h`, ale v souboru `stdlib.h`.

Pro konverzi řetězce na číslo se používá funkce `atoi()` — název vznikl jako zkratka z *A*SCII *t*o *i*nteger. Její úplný funkční prototyp je:

```
int atoi(char *string);
```

Tato funkce konvertuje řetězec obsahující číslo na celočíselnou hodnotu typu `int`, kterou vrátí jako svůj funkční parametr.

Další funkce s podobnou činností jsou:

```
long atol(char *string); konverguje string na long
double atof(char *string); konverguje string na double
```

## 12.3 Formátované čtení a zápis z a do řetězce

V některých případech bychom v programu potřebovali využít výhod formátovaného výpisu, tak jak ho známe z funkce `printf()`, ale nechceme tento výstupní řetězec tisknout. V tomto případě můžeme využít služeb funkce `sprintf()`, která pracuje naprosto stejně, jako funkce `printf()`, ale výsledek své práce zapíše do řetězce. Tento řetězec lze pak dále programem libovolně zpracovávat.

To může být v mnoha případech užitečné, např. když více funkcí připravuje do společného bufferu zprávy, které mají být vytištěny, a jedna funkce je skutečně tiskne podle nějakého algoritmu. Pak všechny podřízené funkce zapisují do řetězců pomocí `sprintf()` a tyto řetězce předávají funkci, která je uloží do bufferu a další funkce je ve vhodné chvíli zobrazí.

### Poznámka:

- Pokud nám způsob práce funkce `sprintf()` není jasný, můžeme si ji představit jako funkci `fprintf()`<sup>14</sup>, která zapisuje do souboru. Funkce `sprintf()` dělá totéž, ale nezapisuje do souboru nýbrž do řetězce.

Stejně tak, jako k funkci `printf()` patří funkce `scanf()` a k funkci `fprintf()` funkce `fscanf()`, je i k funkci `sprintf()` duální funkce `sscanf()`. Funkce `sscanf()` čte do specifikovaných proměnných ze zadaného řetězce.

Zamyslíme-li se nad funkcemi `sprintf()` a `sscanf()`, přijdeme na to, že vlastně významně rozšiřují možnosti konverze čísel na řetězce a naopak — viz též str. 203. Tam byly popsány jen funkce pro převod řetězce na číslo

<sup>13</sup> Opačný převod — čísla na řetězec viz na str. 204.

<sup>14</sup> Podrobně viz str. 66.

a opačný převod — čísla na řetězec vůbec nebyl k dispozici. Pomocí funkcí `sprintf()` a `sscanf()` můžeme tyto převody dělat s veškerým komfortem a převádět např. i do/z hexadecimální soustavy.

### Příklad:

Následující program přečte čtyři hexadecimální číslice do řetězce `s1` a z tohoto řetězce<sup>15</sup> je pak zkonvertuje do proměnné `i`. Proměnná `i` je nejdříve převedena na řetězec osmičkových čísel `s2` a tento řetězec je pak vytištěn.

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

```
main()
```

```
{
```

```
 int i;
```

```
 char s1[5], s2[10];
```

```
 printf("Zadej 4 hexa cislice : ");
```

```
 scanf("%s", s1);
```

```
 sscanf(s1, "%x", &i);
```

```
 sprintf(s2, "%o", i);
```

```
 printf("%s \n", s2);
```

```
}
```

### Příklad:

Funkce `urednik()` je typická funkce, která připravuje výstupní zprávu z několika položek a tuto zprávu vrací jako svůj parametr. Starostí funkce `sef()` je pouze definovat dostatečně velký řetězec a správně volat funkci `urednik()`<sup>16</sup>.

```
void urednik(char *zpr, double polomer)
```

```
{
```

```
 double obsah, obvod;
```

```
 obsah = 3.14 * polomer * polomer;
```

```
 obvod = 2 * 3.14 * polomer;
```

```
 sprintf(zpr, "Kruh o polomeru %f ma obsah %f a obvod %f \n",
 polomer, obsah, obvod);
```

```
}
```

<sup>15</sup> Uvědomme si, že funkce typu `atoi()` nedávaly možnost převodu hexadecimálních čísel.

<sup>16</sup> Analogie se životem by byl snaživý úředník, kterému zadá šéf množství podkladů, on je pracně dá dohromady do hlášení, které potom šéf na nějaké schůzi doslova přečte, popřípadě ji ještě vydává za své dílo.



```
void sef(void)
{
 char zprava[120];

 urednik(zprava, 5.0);
 strcat(zprava, "\t\tVypracoval sef\n");
 printf("Dovoluji si prednest zpravu o kruhu: \n %s", zprava);
}
```

## 12.4 Řádkově orientovaný vstup a výstup z terminálu

Na stranách 198 a 200 byly popsány funkce `scanf()` a `printf()`, které zajistí vstup a výstup "jednoduchého" řetězce. Následující funkce pracují s řetězcem, ve kterém je uložena celá řádka.

### 12.4.1 Čtení řádky z klávesnice

Funkce `scanf()`, která načítá řetězec zadaný z klávesnice do proměnné typu řetězec, má někdy tu nevýhodu, že si zadaný řetězec předzpracuje. To znamená, že vynechá všechny *bílé znaky* před řetězcem a čtení řetězce skončí na prvním bílém znaku, bez ohledu na to, zda ještě řetězec pokračuje, či nikoliv.

Zadali-li bychom například z klávesnice řetězec: " ahoj lidi " pak by příkaz: `scanf("%s", str);` načetl do řetězce `str` pouze "ahoj".

Z tohoto příkladu je vidět, že když potřebujeme načíst z klávesnice celou řádku až do ukončujícího znaku '\n', musíme použít jinou funkci než funkci `scanf()`. Standardní knihovna funkcí v C poskytuje funkci `gets()`, která požadavek na přečtení celé řádky najednou splňuje. Má následující funkční prototyp: `char *gets(char *str);`

Funkce `gets()`<sup>17</sup> čte celou řádku až do znaku '\n' a uloží ji do řetězce `str`, který ukončí znakem '\0'. Znak '\n' se neukládá! Zároveň `gets()` vrátí pointer na řetězec `str`.

Pokud byla řádka prázdná, vrátí `NULL` a do `s[0]` dá ukončovací znak '\0', čili `str` bude nulový řetězec.

<sup>17</sup> Je podobná funkci `READLN` z Pascalu.

### 12.4.2 Výpis řádky na obrazovku

Pokud chceme na obrazovku vypsát řetězec `str`<sup>18</sup> jako řádku ukončenou znakem '\n' můžeme samozřejmě použít příkaz:

```
printf("%s \n", str);
```

který funguje přesně tak, jak je požadováno. Malý problém je ale v menší efektivitě programu, protože `printf()` musí nejdříve pomocí formátové specifikace "%s \n" poměrně složitě zjistit, cože to má tisknout.

Funkce `puts()`<sup>19</sup> je pro tento případ vhodnější, protože nad ničím "nedumá" a hned zadaný řetězec tiskne a navíc ještě sama odřádkuje — vypíše znak '\n'. Její funkční prototyp je: `int puts(char *str);` Pokud funkce `puts()` z nějakého důvodu nemohla pracovat<sup>20</sup>, pak vrátí EOF, jinak vrátí nezáporné číslo.

## 12.5 Řádkově orientovaný vstup a výstup ze souboru

### 12.5.1 Čtení řádky ze souboru

Pro vstup řádky ze souboru slouží funkce:

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

která čte řetězec ze souboru `fr` až do konce řádky, nejvýše však `max` znaků, a včetně znaku '\n'<sup>21</sup> ho uloží do řetězce `str`.

Funkce `fgets()` vrací pointer na `str` nebo, při dosažení konce souboru, vrátí `NULL`.

#### Poznámky:

- Přečte správně i poslední řádku, která není ukončena znakem '\n'.
- Protože `fgets()` umí určit maximální délku čtené řádky, může být užitečná i pro čtení z terminálu, kde tato možnost není. V tomto případě použijeme příkaz: `fgets(s, max, stdin);`
- Jestliže `fgets()` načte ze souboru znak používaný pro *End-Of-File*<sup>22</sup>, pak ho do řetězce většinou neuloží — je vhodné provést experiment s konkrétním překladačem.

<sup>18</sup> Připravený dříve např. funkcí `sprintf()` — viz str. 204.

<sup>19</sup> Je podobná funkci `WRITELN` z Pascalu.

<sup>20</sup> Například při uzavření `stdout`.

<sup>21</sup> Rozdíl s `gets()`!

<sup>22</sup> Například znak "Ctrl Z" v MS-DOSu — nezaměňovat se symbolickou konstantou EOF!



**Příklad:**

Program přečte soubor DOPIS.TXT po řádcích a vytiskne ho na obrazovku. Délka čtené řádky je omezena na 80 znaků.

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

```
main()
```

```
{
```

```
 char radka[81];
```

```
 FILE *fr;
```

```
 if ((fr = fopen("DOPIS.TXT", "r")) == NULL) {
```

```
 printf("Soubor DOPIS.TXT nejde otevrit \n");
```

```
 return;
```

```
 }
```

```
 while(fgets(radka, 80, fr) != NULL)
```

```
 printf("%s", radka);
```

```
 if (fclose(fr) == EOF)
```

```
 printf("Soubor DOPIS.TXT nejde uzavrit \n");
```

```
}
```

**Poznámka:**

- Všimněte si funkce `printf()`. Byla použita namísto funkce `puts()`, protože funkce `fgets()` načte řetězec i se znakem `'\n'`. Kdybychom tento řetězec tiskli pomocí `puts()`, byla by za každou vypsanou řádkou jedna řádka prázdná.

**12.5.2 Zápis řádky do souboru**

Potřebujeme-li zapsat řetězec do souboru a odřádkovat, použijeme funkci `fputs()`, jejíž funkční prototyp je: `int fputs(char *s, FILE *fw);`

Funkce `fputs()` po zapsání řetězce neodřádkuje<sup>23</sup> a ani nezapisuje do souboru ukončovací znak řetězce `'\0'`. Její návratovou hodnotou je nezáporné číslo. V případě neúspěchu<sup>24</sup> vrací `EOF`.

<sup>23</sup> Narozdíl od funkce `puts()`.

<sup>24</sup> Například při pokusu o zápis do neotevřeného souboru nebo při plném disku.

**12.6 Řídící řetězec formátu pro tisk**

V této chvíli už máme dostatek vědomostí na to, aby bylo možné podrobně popsat řídící řetězec formátu pro funkci `printf()` a jí podobné funkce. Tato podkapitola bude spíše podobná manuálu než učebnici, ovšem považujeme za vhodné ji uvést, protože tisk je velmi často používaná akce.

Za každým znakem `"%"` v řídícím řetězci formátu mohou být tyto položky<sup>25</sup>:

`% [příznaky][šířka][.přesnost][modifikátor]konverze`

kde položky: *příznaky* *šířka* *přesnost* *modifikátor*  
jsou nepovinné — to naznačují hranaté závorky [ ]

Dále budou podrobně popsány všechny položky.

**12.6.1 konverze**

Jedná se o jeden znak, který musí být vždy uveden. Mezi ním a znakem `"%"` mohou být další položky.

**celá čísla**

|   |                                                                          |
|---|--------------------------------------------------------------------------|
| d | desítkové číslo typu <b>signed int</b>                                   |
| i | desítkové číslo typu <b>signed int</b>                                   |
| u | desítkové číslo typu <b>unsigned int</b>                                 |
| o | osmičkové (oktalové) číslo typu <b>unsigned int</b>                      |
| x | hexadecimální číslo typu <b>unsigned int</b> malými písmeny, např. 1a2c  |
| X | hexadecimální číslo typu <b>unsigned int</b> velkými písmeny, např. 1A2C |

**znaménková čísla v pohyblivé řádové čárce**

|   |                                                                                                              |
|---|--------------------------------------------------------------------------------------------------------------|
| f | desítkové číslo typu <b>float</b> i <b>double</b> ve tvaru:<br>[-]dddd.dddd                                  |
| e | jako u konverze f, ale v semilogaritmickém tvaru:<br>[-]d.dddd e[+/-]ddd                                     |
| E | jako u konverze e, ale s velkým E ve tvaru:<br>[-]d.dddd E[+/-]ddd                                           |
| g | jako u konverze f nebo e, ale podle tištěné hodnoty a přesnosti se zvolí normální nebo semilogaritmický tvar |
| G | jako u konverze g, ale v případném semilogaritmickém tvaru se tiskne E                                       |

<sup>25</sup> Dosud jsme se zmiňovali pouze o položce *konverze* a letmo o položkách *šířka*, *přesnost* a *modifikátor* — viz str. 35.



## znaky a řetězce

- c jeden znak
- s řetězec ukončený znakem '\0' (znak '\0' se netiskne)
- % tiskne vlastní znak "%"

## pointery

- p adresa argumentu — ukazatel na void  
je implementačně závislý, tiskne nejčastěji hexadecimální číslo
- n nic netiskne, ale ukládá na adresu, na kterou ukazuje příslušný argument typu pointer na int, počet doposud vytištěných znaků

## 12.6.2 modifikátor

Jedná se o jeden volitelný znak, který, je-li použit, musí bezprostředně předcházet znaku *konverze*. Mezi ním a znakem "%" mohou být další položky.

*Modifikátor* mění implicitní velikost *konverze*.

- h modifikuje *konverze d i* na typ *signed short int*  
a *konverze u o x X* na typ *unsigned short int*
- l modifikuje *konverze d i* na typ *signed long int*  
a *konverze u o x X* na typ *unsigned long int*
- L modifikuje *konverze f e E g G* na typ *long double*

## 12.6.3 šířka

Je to dekadické číslo, které nastavuje minimální počet vypisovaných znaků. Bude-li vypisované číslo větší než *šířka*, pak se hodnota čísla vytiskne správně a na velikost *šířka* se nebere ohled.

Mezi *šířka* a znakem "%" může být položka *příznaky* a mezi *šířka* a položkou *modifikátor* může být položka *přesnost*, která je od položky *šířka* oddělena znakem tečka ".".

Minimální počet tištěných znaků se dá také určit nepřímo pomocí znaku "\*". V tomto případě určuje počet znaků argument typu *int*, který musí v seznamu argumentů bezprostředně předcházet argumentu, který se má tisknout.

Různé významy položky *šířka*:

- n tiskne se alespoň *n* znaků (např. 5 pro pět znaků)  
má-li výstupní hodnota méně než *n* znaků, doplňují se mezery zprava — viz též *příznaky*
- 0n tiskne se alespoň *n* znaků (např. 05 pro pět znaků)  
má-li výstupní hodnota méně než *n* znaků, doplňují se nuly zleva
- \* počet znaků je udán hodnotou předchozího argumentu

## 12.6.4 přesnost

Je to dekadické číslo, které:

- Pro konverze *d i u o x X* nastavuje minimální počet cifer čísla na výstupu — tedy jako *šířka*
- Pro konverze *f e E* nastavuje počet cifer za desetinnou tečkou
- Pro konverze *g G* nastavuje maximální počet významových cifer
- Pro konverzi *s* nastavuje maximální počet tištěných znaků — umí tedy "oříznout" vypisovaný řetězec

## Poznámka:

- Při určování počtu desetinných míst nezapomeňte, že desetinná tečka je také jeden znak.

Opět se dá určit počet znaků nepřímo pomocí znaku "\*". V tomto případě určuje počet znaků argument typu *int*, který musí v seznamu argumentů bezprostředně předcházet argumentu, který se má tisknout. Použijeme-li dvě hvězdičky pro *šířka* a *přesnost*, pak musí být v seznamu argumentů dva argumenty.

- .n tiskne se *n* znaků nebo *n* desetinných míst  
má-li výstupní hodnota více než *n* znaků, může být výstup oříznut (u řetězce) nebo zaokrouhlen (u reálného čísla)
- .0 pro konverze *d i u o x X* nastavuje implicitní hodnotu  
pro konverze *f e E* se netisknou žádné desetinné cifry ani desetinná tečka
- . totéž co .0
- \* počet znaků je udán hodnotou předchozího argumentu

## 12.6.5 příznak

Dále uvedené hodnoty *příznak* se jako jediné z dosud popsaných položek mohou kombinovat mezi sebou. Znaky *příznaku* musí bezprostředně následovat za znakem "%".

- výsledek se zarovnává doleva a zprava se doplňují mezery  
není-li uveden, výsledek se zarovnává doprava a zleva se doplňují mezery nebo nuly
- + číslo bude vždy vytištěno se znaménkem "+" nebo "-"
- nic je-li číslo nezáporné, nebude uvedeno znaménko "+" a ani se pro něj nevynechává mezera; je to opak *příznaku* +  
záporná čísla jsou vždy uvedena se znaménkem "-"



|           |                                                                            |  |
|-----------|----------------------------------------------------------------------------|--|
| #         | bude mít na následující konverze tento vliv:                               |  |
| d i u c s | žádný vliv                                                                 |  |
| o         | před osmičkové číslo se přidá znak "0"                                     |  |
| x X       | před šestnáctkové číslo se přidá "0x" ("0X")                               |  |
| f e E     | výsledek vždy obsahuje desetinnou tečku, i když za ní nejsou žádné číslice |  |
| g G       | totéž co e E s dodatkem, že se koncové nuly neodstraňují                   |  |

### 12.6.6 Příklady různých formátů tisku

Vše, co bylo dosud popsáno vypadá velmi složitě. Naštěstí většinu těchto informací v nejbližší době asi nevyužijete. Vraťte se k nim až v případě, že budete chtít nějaký komplikovaný tisk. Pak je vhodné udělat sérii pokusů, na kterých se naučíte jednotlivé "triky" tisku používat.

Aby jste měli alespoň nějaké vodítko, můžete se zkusit zorientovat v následujícím výpisu programu. Je na něm vidět, jaký vliv má položka *příznak*. Bylo vždy tištěno celé číslo o hodnotě 555 a reálné číslo o hodnotě 5.5, přičemž položky *šířka*, *přesnost* a *konverze* jsou uvedeny v záhlaví tabulky. Položka *modifikátor* nebyla použita.

První dvě čísla na první řádce byla tedy vytištěna formáty:

%-+#06d a %-+#06o

| priznak | 6d      | 6o     | 8x       | 10.2e       | 10.2f       |
|---------|---------|--------|----------|-------------|-------------|
| =====   | =====   | =====  | =====    | =====       | =====       |
| %-+#0   | +555    | 01053  | 0x22b    | +5.50e+00   | +5.50       |
| %-+#    | +555    | 01053  | 0x22b    | +5.50e+00   | +5.50       |
| %-+0    | +555    | 1053   | 22b      | +5.50e+00   | +5.50       |
| %-+     | +555    | 1053   | 22b      | +5.50e+00   | +5.50       |
| %-#0    | 555     | 01053  | 0x22b    | 5.50e+00    | 5.50        |
| %-#     | 555     | 01053  | 0x22b    | 5.50e+00    | 5.50        |
| %-0     | 555     | 1053   | 22b      | 5.50e+00    | 5.50        |
| %-      | 555     | 1053   | 22b      | 5.50e+00    | 5.50        |
| %+ #0   | +000555 | 001053 | 0x00022b | +005.50e+00 | +0000005.50 |
| %+ #    | +555    | 01053  | 0x22b    | +5.50e+00   | +5.50       |
| %+0     | +000555 | 001053 | 0000022b | +005.50e+00 | +0000005.50 |
| %+      | +555    | 1053   | 22b      | +5.50e+00   | +5.50       |
| %#0     | 000555  | 001053 | 0x00022b | 005.50e+00  | 0000005.50  |
| %#      | 555     | 01053  | 0x22b    | 5.50e+00    | 5.50        |
| %0      | 000555  | 001053 | 0000022b | 005.50e+00  | 0000005.50  |
| %       | 555     | 1053   | 22b      | 5.50e+00    | 5.50        |

### Časté chyby:

```
char c;
scanf("%1s", &c); c není řetězec

char s[10];
scanf("%s", &s); s je přímo adresa pole — operátor & je navíc
```

### Co je dobré si uvědomit:

- Řetězec je ukončen znakem '\0' a je potřeba mít pro tento znak místo, čili alokovat paměť o jeden char větší než je využitelná délka řetězce.
- Pokud využíváme funkce pro práci s řetězci, je nutné připojit hlavičkový soubor `string.h`.
- Používáme-li funkce pro práci s řetězci a jejich skutečné parametry jsou pointery<sup>26</sup>, je třeba před voláním funkce alokovat paměť pomocí funkce `malloc()`. Je nutné pracovat jen s pamětí, kterou máme k dispozici!

### Cvičení:

- 1) Napište program, který bude číst z klávesnice řetězec, opíše je uzavřené do rámečku ze znaku '\*', např. takto:
 

```

* toto je pekny priklad *

```
- 2) Definujte `s` jako řetězec znaků délky 20. Přiřaďte mu obsah:
 

Toto je priklad

 pomocí funkce `strcpy()`. Dále definujte pointer na `char`. Alokuje dynamicky paměť, do které přepokopírujte řetězec `s`. Vytiskněte nově vzniklý řetězec.
- 3) Přečtěte řetězec délky max. 20 znaků a vytiskněte tyto znaky na jednu řádku seřazené podle abecedy vzestupně.
- 4) Existují slova, která se čtou stejně zleva doprava i zprava doleva (např. radar). Vytvořte program, který bude tyto slova generovat z jejich první načtené poloviny.

<sup>26</sup> Čili řetězce jsou dynamické.



- 5) Napište program, který přečte řetězec a v závislosti na jeho posledním znaku provede:
  - l (L) převod řetězce na malá písmena (*lower*)
  - u (U) převod řetězce na velká písmena (*upper*)
  - x (X) prohození malých a velkých písmen (*exchange*)
 Proveďte změny v řetězci, ne pouze na výstupu.
- 6) Definujte řetězcovou konstantu a zjistěte, od které adresy je uložena v paměti.
- 7) Napište program, který přečte řádku znaků z klávesnice a pak přečte jeden znak. Vstupní řádku opište a všechny znaky v tého řádku, které se shodují se zadaným znakem podtrhněte pomocí znaku "\*". Z cvičných důvodů vyžijte funkci `strchr()`.
- 8) Napište program, který bude vypisovat na obrazovku obsah souboru. Jeho jméno a příponu zadávejte odděleně z klávesnice a nezadávejte oddělovací znak "." (tečka). Využijte funkce `strcat()` a zajistěte také, aby přípona souboru nemohla být zadána delší než třípísmenná.
- 9) Čtete soubor a vypište pouze ty řádky, ve kterých se bude vyskytovat slovo zadané z klávesnice.
- 10) Pomocí funkce `fgets()` vypište počet znaků v jednotlivých řádcích souboru. Zajistěte správné chování i pro prázdné řádky v souboru.
- 11) Přečtete z klávesnice řádku sestávající ze slov oddělených jednou mezerou. Pomocí funkce `strchr()` vytiskněte na každou řádku vždy jen jedno slovo z této řádky bez úvodní mezery.
- 12) Napište funkci `strins(char *s1, char *s2, int i);`, která vloží do řetězce `s1` od pozice `i` řetězec `s2`. Využijte funkci `strcat()`.
- 13) Napište funkci `strdel(char *s, int i, int n);`, která vymaže z řetězce `s` `n` znaků od pozice `i` včetně.
- 14) Napište příkaz tisku čísla  $\pi$ , které nadefinujte jako symbolickou konstantu s přesností alespoň na 7 míst. Z klávesnice budete zadávat, na kolik desetinných míst má být toto číslo vytištěno. Použijte řetězec, jako řídicí formát `printf()`.

## 13 Vícerozměrná pole

Jazyk C umožňuje, aby pole mělo více dimenzí než jen jednu. Nejčastěji se používají dvoudimenzionální pole (tabulky), ale troj a vícerozměrná pole jsou také dovolena. Pro vícerozměrná pole platí, že všechny indexy začínají od 0 (nuly).

### 13.1 Základní definice a přístup k prvkům

Definice vícerozměrného pole je jednodušší než v Pascalu:

|                                            |              |
|--------------------------------------------|--------------|
| Pascal                                     | C            |
| VAR x : ARRAY [0 .. 1, 0 .. 2] OF INTEGER; | int x[2][3]; |

Pro definici více polí se občas používá definici pomocí nového typu:

|         |                                               |
|---------|-----------------------------------------------|
| Pascal: | TYPE DVA = ARRAY [0 .. 1, 0 .. 2] OF INTEGER; |
|         | VAR x : DVA;                                  |
| C:      | typedef int DVA[2][3];                        |
|         | DVA x;                                        |

Nový typ pro dvojrozměrné pole lze také vytvořit pomocí již existujícího jednorozměrného typu:

|        |                                         |
|--------|-----------------------------------------|
| Pascal | TYPE JEDEN = ARRAY [0 .. 1] OF INTEGER; |
|        | DVA = ARRAY [0 .. 2] OF JEDEN;          |
| C      | typedef int JEDEN[2];                   |
|        | typedef JEDEN DVA[3];                   |

Přístup k prvkům vícerozměrného pole pomocí indexů je naprosto stejný, jako přístup do jednorozměrného pole, ale pozor — narozdíl od Pascalu musí být v C každý index v samostatných závorkách, tedy např.:

```
int tabulka[5][10];
tabulka[1][6] = 4;
tabulka[4][9] = 0;
tabulka[0, 0] = 0; /* chyba */
```

Tři a více rozměrná pole jsou podobná dvourozměrným, tedy např.:

```
int troj[5][6][7];
troj[0][5][0] = 10;
```



## 13.2 Uložení vícerozměrných polí v paměti

Při prvních pokusech s C nás příliš nezajímá jak je co uloženo v paměti a proč. Když však budeme chtít pracovat s vícerozměrnými poli efektivně, je nutné vědět, že dvourozměrné pole je uloženo v paměti po řádcích, tedy definice:

```
int x[2][3];
```

alokuje v paměti 2 \* 3 prvků, např. 2 \* 3 \* 2 Byte<sup>1</sup>.

Předpokládejme, že počáteční adresa pole je 100, pak je obsazení paměti:

|            |            |                         |
|------------|------------|-------------------------|
| 100        | 106        | 112                     |
| řádka č. 0 | řádka č. 1 | první volný Byte paměti |

nebo detailně:

|         |         |         |         |         |         |       |
|---------|---------|---------|---------|---------|---------|-------|
| 100     | 102     | 104     | 106     | 108     | 110     | 112   |
| x[0][0] | x[0][1] | x[0][2] | x[1][0] | x[1][1] | x[1][2] | volno |

Až sem to bylo pravděpodobně jasné. Následující řádky jsou poněkud méně srozumitelné, ale vyplatí se snaha o jejich pochopení.

Jazyk C je konzistentní<sup>2</sup> v přístupu k polím — dvourozměrné pole v C je při prvním pohledu vlastně jednorozměrné pole, které má prvky pointerů. To, že jsou to pointery na další jednorozměrné pole, je věcí až druhého, podrobnějšího pohledu<sup>3</sup>. Obsahem prvního prvku x[0] jednorozměrného pole, je pointer na první řádku dvourozměrného pole x. Jinými slovy řečeno — typ jednorozměrného pole x je tříprvkové pole prvků typu int a velikost tohoto typu je<sup>4</sup> 6.

Je to složité? Asi ano, ale snad vám pomůže následující modifikovaný obrázek:

|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
| 100     | 102     | 104     | 106     | 108     | 110     |
| x[0][0] | x[0][1] | x[0][2] | x[1][0] | x[1][1] | x[1][2] |
| x[0]    |         |         | x[1]    |         |         |
| x       |         |         |         |         |         |

Z obrázku je jasné, že:

- x a x[0] představují tutéž adresu (ale jiné typy pointerů)
- x + 1 a x[0] + 1 jsou odlišné adresy (106 a 102)

<sup>1</sup> Kde 2 == sizeof(int).

<sup>2</sup> Pracuje s poli pořád stejně.

<sup>3</sup> Pro třírozměrné pole bychom tyto pohledy potřebovali už tři.

<sup>4</sup> Tedy 3 \* sizeof(int), kde sizeof(int) předpokládáme rovno 2.

Podobně lze vypočítat, že:

|                         |                                             |
|-------------------------|---------------------------------------------|
| x                       | je pointer na dvourozměrné pole             |
| x[i]                    | je pointer na i-tou řádku                   |
| *(x + 1) == x[1] == 106 | je adresa 1-ní řádky <sup>5</sup>           |
| x[i][j]                 | je hodnota prvku [i][j] dvourozměrného pole |

Protože: x[i] == \*(x + i)

je bazová adresa i-té řádky, pak:

- adresa indexované proměnné je:

&x[i][j] == x[i] + j == \*(x + i) + j

- hodnota indexované proměnné je:

x[i][j] == \*(x[i] + j) == (\*(x + i) + j)

Příklad:

Následující úsek programu ukazuje definici, inicializaci a použití dvourozměrného pole.

```
int x[2][3], i, j;
```

```
for (i = 0; i < 2; i++) /* pro každou radku */
 for (j = 0; j < 3; j++) /* pro každý sloupec */
 x[i][j] = i + j;
```

```
for (i = 0; i < 2; i++) {
 printf("mapa pameti pro %d. radku: \n", i);
 for (j = 0; j < 3; j++)
 printf("\t sloupec %d \t adresa %p \t hodnota %d \n",
 j, (x[i] + j), x[i][j]);
 putchar('\n');
```

## 13.3 Různé způsoby definice dvourozměrných polí

Potřebujeme-li pracovat s dvourozměrným polem, které je "obdélníkové" a lze ho definovat staticky<sup>6</sup>, pak to už nyní umíme a jak víme, nejsou v tom

<sup>5</sup> Neboli pointer na 1-ní řádku

<sup>6</sup> Vejde se do paměti, známe jeho meze v době překladu, ...



žádné zrady. Problémy ale nastávají s vícerozměrným dynamickým polem. Naštěstí jsou to problémy pouze při definici pointeru na toto pole a při alokaci paměti pro pole. Jakmile se nám podaří tyto dva kroky zvládnout, můžeme zase pracovat s dynamickým polem naprosto stejně jako se statickým.

Pro vytvoření dvourozměrného pole jsou k dispozici 4 hlavní způsoby, z nichž ten první již známe — je to statické pole. Další tři umožňují vytvářet dvourozměrné dynamické pole, z nichž každé je trochu jiné. V následujícím výkladu budeme předpokládat znalost vztahů z předchozí podkapitoly.

#### Poznámky:

- Ve všech čtyřech případech budeme vytvářet dvourozměrné pole s prvky typu `int` a předpokládáme `sizeof(int) == 2`.
- Při přidělování dynamické paměti funkcí `malloc()` nebudeme z důvodů jednoduchosti zápisu testovat úspěšnost přidělení. V praxi by však byl tento test "životně" nutný, zvláště když by se jednalo o rozsáhlá pole.

#### 13.3.1 Statické dvourozměrné pole

Tento způsob již důvěrně známe, takže jen pro úplnost — definice je:

```
int xa[2][3];
```

Pole je alokováno při překladu jako souvislý blok 6-ti prvků a je uloženo po řádcích v datové oblasti paměti.

`xa` je konstantní pointer, jehož hodnota nemůže být změněna — není to *l-hodnota*.

#### 13.3.2 Pole pointerů

Protože už víme, že jazyk C se dívá na dvourozměrná pole dvouúrovňovým pohledem, je možné tohoto faktu využít a definovat pole pouze pro "první pohled". Samozřejmě je pak nutné, abychom sami zajistili i ten "druhý pohled", který za nás při definici statického pole dělá překladač.

Definice jednorozměrného pole pointerů je:

```
int *xb[2];
```

`xb` není dvourozměrné pole, ale jednorozměrné pole dvou pointerů na typ `int` a tyto pointery pak dále využijeme jako ukazatele na jednotlivé řádky pole. Tyto řádky ale zatím neexistují a proto musíme pro každou řádku<sup>7</sup> jednotlivě alokovat paměť pomocí funkce `malloc()`, tedy:

```
xb[0] = (int *) malloc(3 * sizeof(int));
```

```
xb[1] = (int *) malloc(3 * sizeof(int));
```

<sup>7</sup> Kde řádka je zde vlastně synonymem pro každý prvek (pointer) tohoto jednorozměrného pole.

Po této alokaci, je pak možné teď už dvourozměrné dynamické pole `xb` normálně používat např.: `xb[0][2] = 5;`

#### Poznámky:

- Vždy je ale nutné si uvědomit, že druhá řádka pole `xb` nemusí v paměti ležet bezprostředně za první řádkou — obě řádky spolu v paměti nijak nesouvisí. Jinak řečeno — kdybychom u statického pole použili chybný příkaz: `xa[0][3] = 8;` pak by se hodnota 8 zapsala do prvku `xa[1][0]`, protože přetekla mez první řádky pole a my jsme se dostali na první prvek druhé řádky. Pokud tentýž příkaz použijeme pro dynamické pole `xb`, tedy: `xb[0][3] = 8;` pak s největší pravděpodobností nezměníme prvek `xb[1][0]`, protože ten leží v paměti úplně jinde, neboť patří ke druhé řádce pole `xb`.
- Tento druh pole se velmi často používá.

#### 13.3.3 Pointer na pole

Následující způsob definice dynamického pole není příliš často používaný, ale v některých speciálních případech má svoje opodstatnění.

Vyžívá se skutečnosti, že jazyk C umožňuje také víceúrovňový pohled na dvourozměrné pole "z nižší úrovně", tedy od sloupců. Je to tedy zhruba opačný způsob, než byl předchozí.

Definice je:

```
int (*xc)[3];
```

`xc` není pole, ale jen jeden pointer na pole tří `int`.

Alokujeme-li pomocí `malloc()` "dostatek paměti", můžeme s ní pomocí `xc` pracovat jako s dvourozměrným polem, protože překladač zná velikost řádky — 3 prvky typu `int`.

Výraz "dostatek paměti" znamená celé dvourozměrné pole, tedy:

```
xc = (int *) malloc(2 * 3 * sizeof(int));
```

`xc` nyní ukazuje na blok 6-ti `int` sdružených díky definici `xc` po trojicích, které leží v paměti za sebou. Je to tedy prakticky obdoba statického pole, jen s tím rozdílem, že je celé uloženo v dynamické paměti.

Přístup k prvkům je opět shodný, např.:

```
xc[0][1] = 6;
```

```
xc[1][2] = 9;
```

#### Poznámka:

- Použijeme-li chybný příkaz: `xc[0][3] = 8;`



pak se hodnota 8 zapíše do prvku `xc[1][0]`, stejným způsobem, jako u statického pole.

### 13.3.4 Pointer na pointer

Tento způsob je vůbec nejkomplicovanější a vychází vlastně z "nultého pohledu" na pole, kdy se díváme na jednorozměrné pole jako na pointer.

Definice je:

```
int **xd;
```

a pak platí:

`xd` je pointer na pointer na typ `int`

`*xd` je pointer na typ `int`

`**xd` je prvek typu `int`

Pro vytvoření dvourozměrného pole je nutné učinit dva kroky:

1) Alokovat dva pointery na řádky:

```
xd = (int **) malloc(2 * sizeof(int *));
```

čili nyní je možno využívat dva prvky pole `xd[0]` a `xd[1]`<sup>8</sup>

2) Alokovat paměť pro tři prvky typu `int` na řádce (což je stejné, jako u pole `xb`):

```
xd[0] = (int *) malloc(3 * sizeof(int));
```

```
xd[1] = (int *) malloc(3 * sizeof(int));
```

Po těchto dvou akcích platí pro pole `xd` stejná pravidla, jako pro pole `xb`.

### 13.3.5 Výhody a nevýhody předchozích čtyř způsobů

Výše popsané čtyři způsoby alokace dvourozměrného pole lze hodnotit z různých hledisek:

1) Typ pole:

- Definice `xa` představuje statické pole.
- Definice `xb`, `xc` a `xd` představují po alokaci dynamická pole.

2) Paměťové nároky<sup>9</sup>:

- Definice `xa` je paměťově nejvýhodnější, protože vyžaduje pouze paměť pro prvky pole a žádné pomocné pointery. Také přístup k jednotlivým položkám pole bude pravděpodobně nejefektivnější.
- Definice `xb` vyžaduje navíc paměť pro 2 pointery na typ `int`.
- Definice `xc` vyžaduje navíc paměť pro 1 pointer na typ `int`. Přístup do pole bude pravděpodobně téměř stejně rychlý jako u pole `xa`.

<sup>8</sup> A tím jsme se vlastně dostali do výchozí pozice pro pole `xb`.

<sup>9</sup> Je nutné poznamenat, že každá alokace dynamické paměti potřebuje navíc určité místo pro "administrativu" — viz též str. 168.

- Definice `xd` vyžaduje navíc paměť pro 3 pointery. Dá se předpokládat, že přístup do pole `xd` bude pravděpodobně nejpomalejší ze všech předchozích variant.

3) Charakter pole:

- Definice `xa` vytvoří pouze "pravoúhlé" pole — každá řádka má stejný počet sloupců. Počet řádek i počet sloupců je pevně dán při překladu a nelze ho v průběhu výpočtu měnit.
- Definice `xb` umožňuje vytvořit pole s pevným počtem řádek (dáno při překladu), ale každý řádek může mít jinou velikost — jiný počet sloupců. Pole `xb` tedy může být "zubaté".
- Definice `xc` umožňuje vytvořit pole s pevným počtem sloupců (dáno při překladu), ale libovolným počtem řádek. Pole `xc` je opět "pravoúhlé".
- Definice `xd` umožňuje vytvořit kompletní dynamické pole s libovolným, za běhu programu volitelným, počtem řádek a každá řádka může mít libovolný počet sloupců.

#### Příklad:

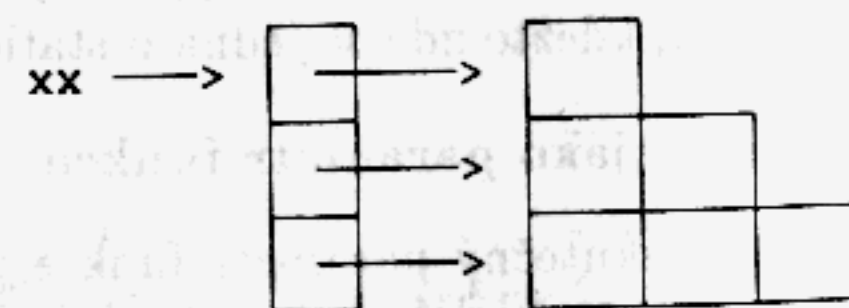
Potřebujeme-li vytvářet dvourozměrné pole s různou délkou řádek, např. pro polovinu matice pod diagonálou, pak použijeme definice podle způsobu pole `xb` nebo `xd` — v našem případě `xb`.

```
int *xx[3];
```

```
for (i = 0; i < 3; i++)
```

```
xx[i] = (int *) malloc((i + 1) * sizeof(int));
```

Tyto příkazy vytvoří v paměti:



Z dvourozměrného pole `xx` jsou pak dostupné prvky:

```
xx[0][0] xx[1][0] xx[1][1] xx[2][0] xx[2][1] xx[2][2]
```

#### Poznámka:

- Prvek `xx[0][1]` je syntakticky správně, ale odkazuje do neznámé oblasti paměti — ne tedy na `xx[1][0]` !

#### Příklad:

Následující funkce umožňuje vytvořit dynamické dvojrozměrné obdélníkové pole prvků `int` libovolných rozměrů. Pro jednoduchost se netestuje



úspěšnost přidělení paměti.

```
int **create(int radky, int sloupce)
{
 int **p_p_x, i;

 p_p_x = (int **) malloc(radky * sizeof(int *));
 for (i = 0; i < radky; i++)
 p_p_x[i] = (int *) malloc(sloupce * sizeof(int));

 return (p_p_x);
}
```

Funkce `create()` by s použila např.:

```
int **a, **b;
a = create(3, 11);
b = create(8, 4);
```

#### Pozor:

Máme-li definice: `int **xx, yy[5][6];`  
pak lze sice napsat: `xx = (int **) yy;`  
ale proměnnou `xx` nelze přesto používat pro přístup k prvkům pole `yy`.  
Proměnná `xx` není správně inicializována pro použití jako pointer pro přístup k dvourozměrnému poli — nenese si v sobě informaci o velikosti řádek.

Pokud vás zarazilo, že to v předchozím případě u funkce `create()` šlo, pak si uvědomte, že tam se jednalo o pole typu `xd`, které je dynamické a je složené ze dvou částí, kdežto zde se jedná o statické pole.

#### 13.3.6 Dvourozměrné pole jako parametr funkce

Dvourozměrné pole jako skutečný parametr funkce je naprosto stejné, jako když je skutečným parametrem pole jednorozměrné — uvádí se pouze název pole.

Malá odlišnost nastává u formálních parametrů, kdy se velikost první dimenze vynechává stejně jako u jednorozměrného pole (prázdné `[]`), ale druhá dimenze musí být uvedena jako konstanta. Z toho, co již o polích víme, můžeme usoudit, že se předává pointer na řádku pole určitého typu a určité délky.

Z toho vyplývá:

- 1) Počet řádek je opět nutno předat jako další parametr funkce, ale občas se navíc předává i počet sloupců.

- 2) Skutečný parametr může být jen statické pole typu `xa` — viz str. 218 nebo dynamické pole typu `xc` — viz str. 219, protože obě jsou "pravoúhlá" pole.

Máme-li pole definované jako: `double x[3][4];`  
pak formální parametr bude:

`double x[][4]` nebo `double (*x)[4]`

#### Pozor:

Častou chybou je formální parametr: `double *x[4]`  
což jak již víme znamená: "`x` je pole čtyř pointerů na typ `double`" — a to je něco jiného než požadované: "`x` je pointer na pole čtyř prvků typu `double`".

#### Příklad:

Následující funkce vrátí největší prvek z dvourozměrného pole, jehož každá řádka má 4 prvky typu `double`

```
double maxim(double pole[][4], int radky)
{
 double pom = pole[0][0];
 int i, j;

 for (i = 0; i < radky; i++) {
 for (j = 0; j < 4; j++) {
 if (pole[i][j] > pom)
 pom = pole[i][j];
 }
 }
 return (pom);
}
```

## 13.4 Inicializace polí všech rozměrů

Inicializace polí se provádí nejčastěji u řetězců — viz str. 195, ale je samozřejmě možné podobně inicializovat i pole jiného typu.

Základní podmínka pro inicializaci je, že pole musí být globální nebo lokální typu `static`, protože automatická pole nelze inicializovat.

Inicializační hodnoty jsou uzavřeny do složených závorek, např:

```
double f[3] = { 1.5, 3.0, 7.6 };
```

Není-li uveden počet prvků pole, kompilátor si ho určí sám podle počtu inicializačních hodnot.



```
double f[] = { 1.5, 3.0, 7.6 };
```

Je-li uveden počet prvků pole a inicializačních hodnot je méně, pak zbývající prvky budou mít nulovou hodnotu:

```
double f[3] = { 1.5, 3.0 };
f[2] bude mít hodnotu 0.0
```

Není možné uvést více inicializací, než je prvků pole:

```
double f[3] = { 1.5, 3.0, 7.6, 9.2 }; /* chybné */
```

Dvourozměrné pole se inicializuje takto:

```
double f[][2] = {
 { 1.5, 3.0 },
 { 7.6, 9.2 }
};
```

přičemž počet sloupců musí být uveden a počet řádek může být uveden.

## 13.5 Pole řetězců

Pole řetězců je asi nejčastěji využívané dvourozměrné pole s různou délkou jednotlivých řádek. Téměř každý program pracující s textem<sup>10</sup> využívá pole řetězců. Stojí tedy za námahu proniknout trochu více do způsobů práce s poli řetězců, protože je dosti pravděpodobné, že se s nimi ve své praxi setkáme.

Definice proměnné `p_text` jako pole čtyř pointerů na řetězec<sup>11</sup>:

```
char *p_text[4];
```

Tomuto poli pointerů můžeme přiřadit hodnoty — adresy řetězců<sup>12</sup> — např.:

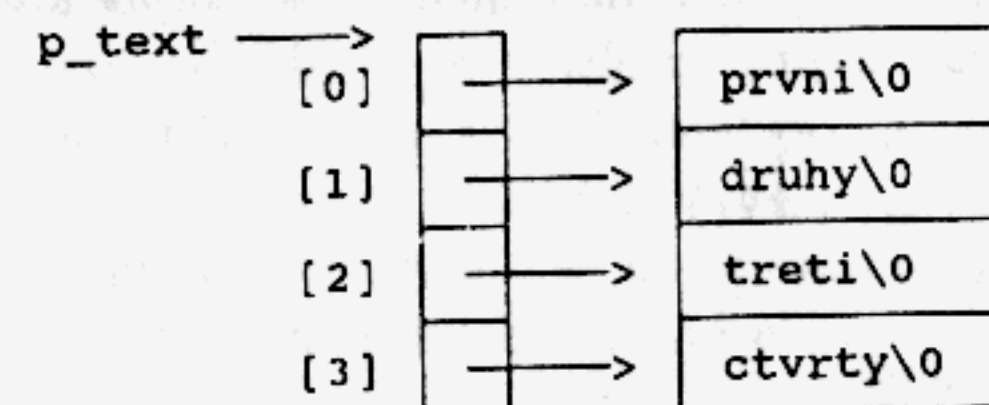
```
p_text[0] = "prvni";
p_text[1] = "druhy";
p_text[2] = (char *) malloc(6);
strcpy(p_text[2], "treti");
p_text[3] = "ctvrty";
```

<sup>10</sup> Typickým představitelem je textový editor.

<sup>11</sup> Což je nám již známé pole typu `xb` — viz str. 218

<sup>12</sup> Pokud jsme překvapeni, že pro jednotlivé řetězce není alokována paměť, je vhodné se vrátit na str. 218.

Po takto provedených přiřazeních bude situace v paměti vypadat následovně:



Jednotlivé znaky z prvního řetězce se budou číst z adres:

```
p_text[0][0] p_text[0][1] p_text[0][2] atd.
```

Tisk prvního řetězce (text "prvni") po znacích pomocí pointeru:

```
char *p_pom = p_text[0];
while (*p_pom != '\0')
 putchar(*p_pom++);
```

Tisk druhého řetězce (text "druhy") pomocí `printf()`, tedy najednou celý řetězec:

```
printf("%s \n", p_text[1]);
```

Tisk třetího řetězce (text "treti") pomocí `puts()`, tedy opět najednou celý řetězec:

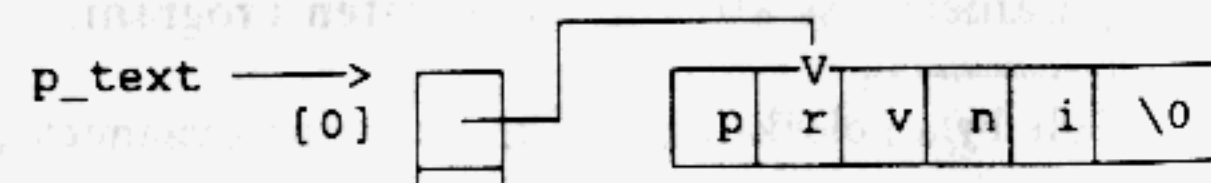
```
puts(p_text[2]);
```

Je tedy vidět, že se s polem řetězců nepracuje žádným zvláštním způsobem. Velký pozor je však nutno dávat v případě, že zkusíme využít "ne-standardní" postupy pomocí dalších pointerů, např.:

```
char **p_pom = p_text;
puts(++p_pom);
```

Tento příkaz vytiskne text "rvni" místo očekávaného textu "druhy", protože `*p_pom` je pointer na nultý prvek pole `p_text` a příkaz: `++p_pom` zvětšil hodnotu na této adrese o 1<sup>13</sup>

Situace tedy bude následující:



`p_text[0]` nyní ukazuje na podřetězec "rvni" a tato změna je trvalá!

<sup>13</sup> Přesněji o `1 * sizeof(char)`





Použijeme-li však stejně definovaný pointer, ale jiný příkaz, dostaneme očekávané výsledky, protože zde se neinkrementuje obsah na adrese `*p_pom`, ale samotný pointer `p_pom`, např.:

```
char **p_pom = p_text;
for (i = 0; i < 4; i++)
 puts(*p_pom++);
```

V tomto případě se vytiskne:

```
prvni
druhy
treti
ctvrty
```

#### Poznámka:

- Pole řetězců používané v předchozím případě bychom v praxi definovali a inicializovali nejspíše takto:

```
char *p_pole[] = {"prvni", "druhy", "treti", "ctvrty"};
```

kde by všechny položky tohoto pole řetězců byly statické<sup>14</sup>.

## 13.6 Parametry funkce main()

Dosud jsme používali funkci `main()` jako funkci bez parametrů a s implicitní návratovou hodnotou typu `int`.

Pomocí návratové hodnoty lze předat volajícímu — což je v případě funkce `main()` operační systém, který program spustil — výsledek práce programu. Například v MS-DOSu se takto předaná hodnota zapisuje do systémové proměnné `ERRORLEVEL`, odkud může být přečtena např. v dávkovém souboru. Je ovšem nutné poznamenat, že způsob využití návratové hodnoty ANSI C nijak nedefinuje a záleží vždy na konkrétním operačním systému.

Narozdíl od návratové hodnoty, platí pro parametry funkce `main()` zcela přesná pravidla. `main()` může mít formální parametry<sup>15</sup> a to jeden nebo dva nebo žádný, na což jsme byli doposud zvyklí.

Má-li `main()` parametry, jsou z historických důvodů pojmenovány vždy jako `argc` a `argv`<sup>16</sup>. Jejich účel je předat programu argumenty ze vstupní příkazové řádky, tedy parametry, se kterými byl spuštěn program.

<sup>14</sup> V předchozím příkladě byla položka `p_pole[2]` alokována dynamicky, ostatní položky jsou stejné.

<sup>15</sup> Skutečné parametry dodává spouštějící operační systém.

<sup>16</sup> Pojmenovat je jinak by znamenalo výsměch všem opravdovým C-čkovským programátorům!

#### Poznámka:

- Dobře udělané programy totiž pracují tak, že umožňují zadat údaje potřebné pro své ovládání<sup>17</sup> již z příkazové řádky. To má pak výhodu, že program může být spuštěn i z dávkového souboru a po spuštění se již neptá na další informace. Pokud uživatel programu nezadá při spuštění programu parametry, vypíše se buď návod nebo je umožněno zadání těchto parametrů standardně z klávesnice.

Je-li program `TEST.EXE` spuštěn příkazem:

```
test param1 param2
```

a funkce `main()` má hlavičku<sup>18</sup>:

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

pak má parametr `argc` hodnotu 3, protože udává počet řetězců na vstupní řádce — tedy "test" "param1" "param2"

a pole pointerů na řetězce `argv` ukazuje takto:

```
argv[0] na řetězec: test
argv[1] na řetězec: param1
argv[2] na řetězec: param2
```

Argument příkazové řádky, který je uzavřen do uvozovek, se počítá za jeden řetězec. Například po příkazu:

```
test "ahoj, jak" se "porad mas"
```

bude situace následovná:

```
argc == 4
argv[0] == test
argv[1] == ahoj, jak
argv[2] == se
argv[3] == porad mas
```

#### Poznámka:

- Protože jednotlivé parametry vstupní řádky jsou normální řetězce, lze pro práci s nimi využít všech funkcí, které s řetězci pracují.

<sup>17</sup> Například jména souborů, se kterými bude program pracovat.

<sup>18</sup> Též se používá: `main(int argc, char **argv)`



Příklad:

Program vypíše počet argumentů příkazové řádky včetně názvu programu a tyto parametry opíše na nové řádky.

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

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

```
{
```

```
 int i;
```

```
 printf("Vstupní řádka má %d parametru \n", argc);
```

```
 for (i = 0; i < argc; i++)
```

```
 printf("%s \n", argv[i]);
```

```
}
```

## 13.7 Externí pole všech rozměrů

Pole může být specifikováno jako **extern**, což znamená deklaraci pole, když je toto pole definováno např. v jiném souboru.

Opět platí, že první dimenze pole je uváděna dobrovolně, ale vřele se doporučuje ji při externích deklaracích uvádět, protože to zvyšuje čitelnost programu.

Poznámka:

- Filosofickou otázkou ovšem je, zda má být vůbec pole externí, tzn. sdílené více moduly. Tomuto stavu se snažíme vyhýbat, protože při sdílení pole je mnohem větší pravděpodobnost nevhodného zásahu do něj<sup>19</sup>, než při sdílení jednoduché proměnné.

Příklad:

Soubor A.C — definice:

```
int x[10];
```

```
float y[2][3];
```

Soubor B.C — deklarace:

```
extern int x[10];
```

```
extern float y[2][3];
```

nebo hůře extern int x[];

nebo hůře extern float y[][3];

Co je dobré si uvědomit:

- Pole jsou v paměti ukládána po řádcích.
- Je-li definice: `int x[2][3];` pak:
  - typ `x` je pointer na řádku tří `int`
  - typ `*x` je pointer na `int`
  - ve výrazu `x + i` je `i` násobeno počtem sloupců (3)
  - `*x == x[0]`
  - `**x == *(x[0]) == x[0][0]`
  - `x[i] == *(x + i) == *x + i * počet sloupců`
- Je-li `x` dvourozměrné pole, pak jako formální parametr funkce musí mít uvedený druhý rozměr pomocí konstanty: `int x[][5]`<sup>20</sup> nebo `int *x[5]`

Časté chyby:

`f(double b[][2])` není uveden druhý rozměr pole `b`

Cvičení:

- Napište program, který naplní matici 10 x 10 `int` čísly 0–99 a pomocí pointerů vytiskne její obsah přehledně na obrazovku. Při nastavování hodnot prvků přistupujte do matice pomocí indexů.
- Definujte statické pole 5 x 5 `int` a vypište adresu začátku pole, 0-té a 1-ní řádky, prvku `[0][0]`, `[1][0]` a posledního prvku. Pomocí `sizeof` zjistěte velikost pole a porovnejte toto číslo s bazovou adresou pole a adresou jeho posledního prvku. Adresy vypisujte formátem `%p` nebo `%x`.
- Napište program, který pomocí pole pointerů na `char` přečte soubor a uloží ho do dynamické paměti po řádcích. Ukládejte jen skutečné délky řádek. Čtený soubor nebude mít více než 1000 řádek. Do druhého souboru zapište přečtený soubor pozpátku, tedy nejdříve poslední řádku a naposledy první řádku. Pro čtení celé řádky použijte funkci `fgets()`.
- Upravte předchozí program tak, že bude na obrazovku vypisovat pouze řádky, ve kterých se objeví slovo, které bude zadáno z klávesnice. Pro hledání řetězce využijte funkci `strstr()`.

<sup>20</sup> V tomto případě může být uveden i 1. rozměr pole (`int x[4][5]`), ale není na něj brán zřetel.

<sup>19</sup> Stačí jen fakt, že nejsou kontrolovány meze pole.



- 5) Upravte předchozí program tak, že vytvoří soubor **ODKAZY.TXT**, kde ke každému slovu ze souboru **SLOVA.TXT** přiřadí čísla řádek, ve kterých se příslušné slovo nacházelo.
- 6) Definujte v **main()** lokální automatické pole  $10 * 10$  prvků typu **int** a vytvořte funkce **napln()** a **tiskni()**. První funkce toto pole naplní čísly 0–99 a druhá ho vytiskne na obrazovku. Pole předávejte do obou funkcí jako skutečný parametr.
- 7) Vytvořte program, který vytvoří dynamické pole jako dolní trojúhelníkovou matici. Tuto matici naplníte čísly  $x_{ij}$ , kde  $x_{ij} = i * 10 + j$  a vytisknete ji.
- 8) Napište program, který přečte parametry vstupní řádky a opíše je velkými písmeny.
- 9) Napište program, který vypíše na obrazovku soubor, jehož jméno je první parametr příkazové řádky. Pokud bude uveden druhý parametr, pak ho považujte za jméno souboru, do kterého se bude čtený soubor kopírovat místo výpisu na obrazovku. Nebude-li uveden žádný parametr, vypíše přehledný návod k použití.
- 10) Upravte předchozí program tak, že návod k použití se vypíše, bude-li první parametr **-h** nebo **-H** (*help*). Nebude-li zadán žádný parametr, program bude po spuštění žádat zadání jména čteného souboru z klávesnice.
- 11) Napište program, který bude v souboru, jehož jméno bude uvedeno jako 1. parametr příkazové řádky, hledat řetězec, který bude zadán jako druhý parametr. Řádky obsahující tento řetězec se vypíšou na obrazovku. Program bude reagovat i na následující znaky, které se mohou objevit v libovolném pořadí jako 3. parametr příkazové řádky:
  - c** (*case sensitive*) rozlišuje velká a malá písmena, jinak je považuje za totožná
  - n** (*numbering*) každá vypsaná řádka bude očíslována, jinak se nebude číslovat

## 14 Struktury, uniony a výčtové typy

### 14.1 Struktury

Zatímco pole je homogenní datový typ — všechny jeho prvky jsou stejného typu — je *struktura* (*structure*) datový typ heterogenní. Heterogenní znamená, že datový typ je složen (v podstatě libovolně) z datových prvků různých typů, což je ale jeho vnitřní záležitost, protože navenek vystupuje jako jednolitý objekt.

Jak už asi tušíte, není struktura v C nic nového — v Pascalu je obdobný datový typ, který se jmenuje *záznam* (*record*).

| Pascal         | C        |
|----------------|----------|
| VAR a : RECORD | struct { |
| položky;       | položky; |
| END;           | } a;     |

#### 14.1.1 Definice a základní dovednosti

Předchozí srovnání byla ukázkou podobnosti struktury s Pascalským záznamem. Ve skutečnosti lze v C definovat strukturu a definovat proměnné typu struktura pěti následujícími způsoby. Budeme požadovat strukturu složenou ze dvou položek — *vyska* a *vaha* a chceme mít tři proměnné typu struktura — *pavel*, *honza* a *karel*.

Pět způsobů definice struktury:

- 1) Základní způsob. Vytvořená struktura není pojmenována a nedá se tedy nikde dále v programu využít. Dají se ovšem využívat definované proměnné *pavel*, *honza* a *karel*

```
struct {
 int vyska;
 float vaha;
} pavel, honza, karel;
```



- 2) Modifikace základního způsobu, kdy je struktura pojmenována `miry`<sup>1</sup> a dá se tedy využít i později v programu (viz též následující způsob).

```
struct miry {
 int vyska;
 float vaha;
} pavel, honza, karel;
```

- 3) Je to předchozí způsob, kdy je jen oddělena definice struktury od definic proměnných. Je vidět, že definice proměnných se mohou dělat i vícekrát, což nebylo v prvním případě možné.

```
struct miry { /* definice struktury */
 int vyska;
 float vaha;
};
struct miry pavel; /* definice promenných */
struct miry honza, karel;
```

Pozor na častou chybu — nelze definovat proměnné tímto způsobem:

```
miry pavel, honza, karel; /* chyba */
```

- 4) Definice nového typu struktury pomocí příkazu `typedef`. Struktura není pojmenována, což nám příliš nevádí, neboť nový typ struktura pojmenován je (`MIRY`) a dá se tedy dále libovolně používat, např. pro definice proměnných, přetypování, ...

```
typedef struct { /* definice typu struktury */
 int vyska;
 float vaha;
} MIRY;
MIRY pavel, honza, karel; /* definice promenných */
```

Všimněte si, že v definici proměnných, narozdíl od předchozího způsobu, není použito klíčové slovo `struct`.

- 5) Modifikace předchozího způsobu, kdy je pojmenována jak struktura, tak i nový typ struktura. Pro použitý příklad nemá tento způsob opodstatnění, ale je nutné ho použít, když struktura odkazuje sama na sebe — viz str. 235.

```
typedef struct miry { /* definice typu struktury */
 int vyska;
 float vaha;
} MIRY;
```

<sup>1</sup> Toto pojmenování se někdy označuje jako `tag`.

```
MIRY pavel, honza, karel; /* definice promenných */
```

#### Štábní kultura:

- Doporučuje se pojmenovat strukturu i nový typ shodně, pouze je rozlišit velikostí písmen.

#### Poznámka:

- Praxe ukazuje, že je nejvýhodnější používat dva naposledy uvedené typy, protože klíčové slovo `struct` použijeme pouze jednou, a pak už pracujeme pouze s novým typem — zdrojový text je kratší a přehlednější. Tyto způsoby definice struktury budou v dalším textu výhradně používány.

Přístup k prvkům struktury je naprosto stejný jako v Pascalu, tedy pomocí *tečkové notace*.

```
pavel.vyska = 186;
karel.vaha = 89.5;
honza.vyska = pavel.vyska;
```

#### Poznámka:

- Příkaz **WITH** využívaný v Pascalu jazyk C nepodporuje.

Struktury se často používají v kombinaci s poli. Většinou se jedná o pole struktur, ale je samozřejmě možný i opačný způsob — pole ve struktuře. Pole struktur se definuje naprosto stejně, jako jakékoliv jiné pole, např:

```
MIRY lide[100];2
```

pak je přístup k prvku struktury: `lide[50].vyska = 176;`

K&R verze jazyka C neumožňovala pracovat najednou s celou strukturou. Např. příkaz: `honza = pavel;` nebyl možný. Toto omezení již v ANSI C není, takže v programu je možný např. příkaz, který zkopíruje obsah jedné struktury do druhé pomocí jednoho přiřazovacího příkazu.

```
typedef struct { /* definice struktury jejimz */
 int a[10]; /* jediným prvkem je pole int */
} STR_POLE;
```

```
main()
{
 MIRY pavel, honza;
 STR_POLE aaa, bbb;
```

<sup>2</sup> Pro úplnost — podle typu 3) by to bylo: `struct miry lide[100];`



```
pavel.vyska = 186;
aaa.a[1] = 5;
honza = pavel;
bbb = aaa;
}
```

**Poznámka:**

- Všimněte si triku se STR\_POLE, pomocí něhož lze pracovat s celým polem najednou.

**14.1.2 Struktury a pointery**

Pointery na struktury mají dvě velké oblasti použití:

- při práci se strukturami v dynamické paměti,
- při práci se strukturou ve funkci.

Pointer na strukturu se definuje již známým způsobem:

```
typedef struct {
 char jmeno[30];
 int rocnik;
} STUDENT;
```

```
STUDENT s, *p_s;
```

Kde *s* je struktura typu STUDENT a *p\_s* je pointer na strukturu typu STUDENT, která nemá samozřejmě zatím přidělenou paměť. Paměť pro strukturu lze dynamicky přidělit opět pomocí funkce malloc(), např.:

```
p_s = (STUDENT *) malloc(sizeof(STUDENT));
```

Pointer lze samozřejmě použít pro odkaz na již existující strukturu, tedy např.: *p\_s = &s;*

Občas se definuje i nový typ pointer na strukturu, např.:

```
typedef struct {
 char jmeno[30];
 int rocnik;
} STUDENT, *P_STUDENT;
```

```
STUDENT s;
```

```
P_STUDENT p_s;
```

```
p_s = (P_STUDENT) malloc(sizeof(STUDENT));
```

Máme-li definice: STUDENT *s*, \**p\_s* = &*s*;

pak lze do struktury *s* přistupovat:

- pomocí jména struktury *s* *s.rocnik* = 3;
- pomocí pointeru *p\_s* komplikovaně (\**p\_s*).rocnik = 4;
- pomocí pointeru *p\_s* jednodušeji *p\_s->rocnik* = 4;

**Pozor:**

Příkaz: \**p\_s.rocnik* = 4; je chybně, protože operátor . (tečka) má vyšší prioritu než operátor dereference \*

V tomto případě by byl příkaz vyhodnocen jako:

```
*(p_s.rocnik) = 4;
```

což znamená: "na adresu, kam ukazuje obsah proměnné *p\_s.rocnik*, zapiš hodnotu 4"

Srovnáme-li stručně přístupy k prvkům statické a dynamické struktury, pak dostaneme:

**Pascal**

```
s.rocnik := 4;
```

```
p_s^.rocnik := 4;
```

**C**

```
s.rocnik = 4;
```

```
p_s->rocnik = 4;
```

**14.1.3 Struktury odkazující samy na sebe**

Jazyk C nedovoluje v definici struktury přímý odkaz sám na sebe — rekurzi, např.:

```
struct data {
 struct data d; /* chyba */
 ...
}
```

Tento pochopitelný nedostatek lze ale snadno obejít pomocí pointeru na strukturu.

Pomocí pointerů se ve struktuře může definovat odkaz sám na sebe<sup>3</sup>.

```
typedef struct polozka {
 int hodnota;
 struct polozka *p_dalsi;
} POLOZKA;
```

**Pozor:**

Častou chybou bývá tato definice:

<sup>3</sup> Zde je právě nutné využít posledního uvedeného způsobu definice struktury — viz str. 232.



```
typedef struct {
 int hodnota;
 POLOZKA *p_dalsi; /* chybne */
} POLOZKA;
```

protože, v okamžiku, kdy definujeme položku struktury `p_dalsi`, není ještě znám typ struktury `POLOZKA`.

#### Poznámka:

- Prvku `p_dalsi` struktury `POLOZKA` se občas říká dynamický prvek struktury. Je to protiklad ke statickému prvku struktury z následující podkapitoly.

#### Příklad:

Následující program představuje spojový seznam. Nejprve se seznam vytvoří a inicializuje hodnotami 1 až *n*. Pak se seznam projde a vynechá se každá položka, jejíž hodnota je dělitelná 3. Nakonec se seznam vytiskne. Využívá se skutečnosti, že poslední prvek seznamu ukazuje na `NULL`.

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

```
typedef struct prvek {
 int hodnota;
 struct prvek *dalsi;
} PRVEK;
```

```
main()
{
 int i, pocet; /* pocet prvku seznamu */
 /* pointer na aktualni, prvni a predchozi prvek */
 PRVEK *p_akt, *p_prv, *p_pred;
```

```
printf("Zadej pocet prvku seznamu : ");
scanf("%d", &pocet);
```

```
/* vytvoreni seznamu */
/* vytvoreni prvniho prvku */
if ((p_prv = (PRVEK *) malloc(sizeof(PRVEK))) == NULL) {
 printf("Malo pameti\n");
 return;
}
```

```
p_akt = p_prv;
p_prv->hodnota = 1;
```

```
/* vytvoreni vseh dalsich prvku */
for (i = 2; i <= pocet; i++) {
 if ((p_akt->dalsi = (PRVEK *) malloc(sizeof(PRVEK)))
 == NULL) {
 printf("Malo pameti\n");
 break;
 }
 p_akt = p_akt->dalsi;
 p_akt->hodnota = i;
}
p_akt->dalsi = NULL; /* ukonceni seznamu */
```

```
/* vynechani kazdeho prvku, ktery je delitelný 3 */
for (p_pred = p_akt = p_prv; p_akt != NULL;
 p_pred = p_akt, p_akt = p_akt->dalsi) {
 if (p_akt->hodnota % 3 == 0) {
 p_pred->dalsi = p_akt->dalsi;
 free((void *) p_akt);
 p_akt = p_pred;
 }
}
```

```
/* vypis zbytku seznamu */
for (p_akt = p_prv; p_akt != NULL; p_akt = p_akt->dalsi)
 printf("%d \n", p_akt->hodnota);
}
```

#### 14.1.4 Struktura v jiné struktuře

V praktických příkladech je častý případ, kdy potřebujeme mít jako prvek struktury jinou strukturu. To je v C umožněno bez problémů.

#### Poznámky:

- Prvek struktury je zde míněn statický prvek ve významu, že celá struktura, která musí být definována dříve, je uložena uvnitř jiné struktury<sup>4</sup>. V předchozí podkapitole jsme si ukázali jak by vypadal dynamický prvek — ve struktuře byl uložen pouze pointer na jinou strukturu, která ovšem nebyla částí struktury původní.

<sup>4</sup> Dalo by se říci: "Stará struktura v nové struktuře."



- Struktura, která je prvkem jiné struktury, se někdy označuje jako *vnížená struktura*.

Předpokládejme, že údaje o zaměstnancích budou uloženy ve poli následujících struktur:

```
typedef struct {
 char ulice[30];
 int cislo;
} ADRESA;
```

```
typedef struct {
 char jmeno[20];
 ADRESA adresa;
 float plat;
} OSOBA;
```

```
OSOBA lide[1000];
```

a chceme vytisknout adresu člověka s nejvyšším platem. Předpokládáme, že položky ulice a jmeno jsou řetězce, tedy ukončené znakem `'\0'`.

```
int i, kdo = 0;
float max = 0,
 pom;
```

```
for (i = 0; i < 1000; i++) {
 if ((pom = lide[i].plat) > max) {
 max = pom;
 kdo = i;
 }
}
```

```
printf("Zamestnanec s nejvetsim platem bydlí v : %s %d\n",
 lide[kdo].adresa.ulice, lide[kdo].adresa.cislo);
```

Jiný způsob řešení tohoto problému je pomocí pointerů, díky nimž se budou efektivněji (rychleji) počítat adresy:

```
float max = lide[0].plat;
OSOBA *p_pom, *p_kdo;
for (p_pom = p_kdo = lide; p_pom < lide + 1000; p_pom++) {
 if ((p_pom->plat) > max) {
 p_kdo = p_pom;
```

```
 max = p_pom->plat;
 }
}
printf("Zamestnanec s nejvetsim platem bydlí v : %s %d\n",
 p_kdo->adresa.ulice, p_kdo->adresa.cislo);
```

*Vysvětlující poznámky:*

- `max` je inicializováno platem nultého člověka
- `p_pom` je inicializováno jako bazová adresa pole struktur
- výraz `p_pom++` znamená další prvek pole `lide`, čili další strukturu
- `p_pom++` tedy neznamená, že se k `p_pom` přičte jednička, ale že se přičte `1 * sizeof(OSOBA)` — viz též str. 163

Poznámka:

- Jako cvičení můžete zkusit porovnat efektivnosti obou způsobů.

#### 14.1.5 Alokace paměti pro jednotlivé položky struktury

Chceme-li zcela využít možnosti jazyka C, pak je nutné, abychom se — stejně jako u polí (viz str. 216) — seznámili s tím, jak jsou prvky struktury uloženy v paměti<sup>5</sup>.

Položky struktury obsazují paměť v pořadí definic shora dolů a na řádce zleva doprava, např.:

```
typedef struct {
 char c;
 int i, j, k;
 char d;
} PRIKLAD;
PRIKLAD pokus;
```

Ve struktuře `pokus` leží položky v tomto pořadí:

```
c i j k d
```

Pořadí uložení položek platí vždy, ale situaci v paměti komplikují skutečnosti, že:

- 1) Položky struktury jsou většinou zarovnávány na sudé adresy (*memory alignment*), čili za položkou `c` je většinou 1 Byte prázdná výplň.
- 2) Struktura většinou končí na stejně zarovnané (sudé) adrese jako začínala, čili za položkou `d` může být opět 1 Byte prázdná výplň.

<sup>5</sup> Upřímně řečeno, u polí byla tato znalost nezbytná, u struktur se využije jen ve speciálních případech.



Předpokládáme-li `sizeof(int) == 2`, pak uvedená struktura **pokus** může alokovat 8 nebo 9 nebo i 10 Byte paměti a ne jen 8 Byte, jak by nám vyšlo prostým sečtení velikosti všech položek.

Z toho plynou dvě poučení:

- 1) Velikost struktury se zásadně zjišťuje pomocí operátoru `sizeof` pro celou strukturu najednou.
- 2) Programy, kdy se v prvcích struktury přistupuje ne pomocí operátoru (tečka) nebo operátoru `->`, ale přímo pomocí nějak získaného offsetu<sup>6</sup>, jsou zcela nepřenositelné. Při jejich vytváření je nutno přesně zjistit, jak jsou prvky v paměti uloženy.

#### 14.1.6 Struktury a funkce

K&R verze C umožňuje, že funkce může vracet pointer na strukturu a také struktura může být parametrem funkce, ale pouze pomocí volání odkazem, tedy pomocí pointeru.

ANSI C tyto možnosti ponechává a přidává k nim i možnost, kdy návratová hodnota funkce může být struktura a struktura může být předána funkci jako skutečný parametr hodnotou<sup>7</sup>.

##### Příklad:

Funkce pro sčítání komplexních čísel.

```
typedef struct {
 double re, im;
} KOMP;

KOMP secti(KOMP a, KOMP b)
{
 KOMP c;

 c.re = a.re + b.re;
 c.im = a.im + b.im;
 return (c);
}
```

<sup>6</sup> Tzn. posunutí od počáteční adresy struktury dané jejím jménem — v našem případě **pokus**

<sup>7</sup> Je to složitě řečeno, význam je ten, že funkce pracuje se strukturou stejně jako např. se základním datovým typem `int`.

```
main()
{
 KOMP x, y, z;

 x.re = 1.1; x.im = 3.14;
 y = x;
 z = secti(x, y);
}
```

Tento způsob je výhodný pouze v případě, že struktury mají malou velikost. Uvědomme si totiž, že předáváme-li strukturu hodnotou, musí se ve *stacku* vytvořit její lokální kopie. Má-li takto předávaná struktura více položek, pak je toto kopírování samozřejmě časově náročné a zabírá mnoho paměti ve *stacku*.

Je také nutné si uvědomit, že se jedná o předávání hodnotou, takže jakákoliv změna prvku struktury — jako skutečného parametru ve funkci — se mimo funkci neprojeví.

Podobné pravidlo o vytváření kopie struktury platí i při vracení hodnoty struktury jako návratového typu funkce. Zde se opět musí kopírovat — v našem případě lokální struktura `c` — někde do paměti.

Z těchto důvodů se velmi často používá jen způsob, který již umožňovala K&R verze jazyka C, totiž předávání parametrů odkazem pomocí pointerů. V tomto případě se totiž kopíruje do *stacku* pouze jedna adresa, což je v případě větších struktur značná úspora času a místa. Další výhodou je, že se skutečné parametry dají ve funkci změnit.

##### Příklad:

Předchozí příklad přepsaný pomocí pointerů.

```
typedef struct {
 double re, im;
} KOMP;

void secti(KOMP *a, KOMP *b, KOMP *c)
{
 c->re = a->re + b->re;
 c->im = a->im + b->im;
}
```



```
main()
{
 KOMP x, y, z;

 x.re = 1.1; x.im = 3.14;
 y = x;
 secti(&x, &y, &z);
}
```

#### Poznámka:

- Všimněte si, že předáváme-li funkci strukturu odkazem (skutečný parametr funkce je tedy adresa struktury, čili vlastně pointer na strukturu), je ve funkci nutné přistupovat k prvkům struktury pomocí operátoru ->

#### Příklad:

Následující program je trochu složitější a ukazuje už reálné použití struktur ve funkcích.

Funkce vytvor1() a vytvor2() alokují místo pro strukturu typu STUDENT stejným způsobem — pomocí malloc(), ale s touto adresou pracují každá jinak:

- vytvor1() vrací pointer na nově alokovanou strukturu
- vytvor2() nastaví adresu nové struktury do svého parametru

Funkce nastav() přiřadí jméno a ročník.

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

```
typedef struct {
 char jmeno[30];
 int rocnik;
} STUDENT;
```

```
STUDENT *vytvor1(void)
{
 STUDENT *p_pom;

 p_pom = (STUDENT *) malloc(sizeof(STUDENT));
 if (p_pom == NULL)
 printf("Malo pameti \n");
 return (p_pom);
}
```

```
void vytvor2(STUDENT **p_s)
{
 *p_s = (STUDENT *) malloc(sizeof(STUDENT));
 if (*p_s == NULL)
 printf("Malo pameti \n");
}
```

```
void nastav(STUDENT *p_s, char *jmn, int rok)
{
 p_s->rocnik = rok;
 strcpy(p_s->jmeno, jmn);
}
```

```
main()
{
 STUDENT s, *p_s1, *p_s2;

 p_s1 = vytvor1(); /* pouziti vytvor1() */
 vytvor2(&p_s2); /* pouziti vytvor2() */

 /* prace s polozkami struktur */
 s.rocnik = 3;
 p_s1->rocnik = s.rocnik + 1;
 p_s2->rocnik = 5;
 nastav(&s, "pavel", 1);
 nastav(p_s1, "karel", 2);
 nastav(p_s2, "honza", 3);
}
```

#### 14.1.7 Shrnutí poznatků o práci se strukturami

- 1) Statická struktura může být skutečným parametrem funkce předávaným hodnotou a může to být i návratový typ funkce, což je ale vhodné pouze při malé velikosti struktury.
  - funkční prototyp: KOMP secti(KOMP a, KOMP b);
  - přístup k prvkům struktury ve funkci: c.re = a.re + b.re;
  - volání funkce: z = secti(x, y); /\* bez & \*/
- 2) Statická struktura se často předává jako skutečný parametr odkazem, což je vhodné při větší velikosti struktury.
  - funkční prototyp: void secti(KOMP \*a, KOMP \*b, KOMP \*c)
  - přístup k prvkům struktury ve funkci: c->re = a->re + b->re;



- volání funkce: `secti(&x, &y, &z); /* s & */`
- 3) Kombinace předchozích způsobů jsou možné.
- 4) U dynamicky vytvořené struktury jsou také možné oba způsoby (hodnotou a odkazem), ale v naprosté většině případů se používá jen způsob druhý — odkazem, protože pointer k tomu přímo vybízí.
- funkční prototyp: `void secti(KOMP *a, KOMP *b, KOMP *c)`
  - přístup k prvkům struktury ve funkci: `c->re = a->re + b->re;`
  - volání funkce: `secti(p_x, p_y, p_z); /* bez & */`

#### 14.1.8 Inicializace struktur

Struktury se inicializují podobně jako pole uvedením seznamu inicializačních hodnot uzavřeným ve složených závorkách "{" a "}".

```
typedef struct {
 int i, j;
 float f;
} PRIKLAD;
PRIKLAD a = { 1, 2, 6.4 };
```

Podobně lze inicializovat i pole struktur:

```
PRIKLAD b[] = {
 { 4, 5, 1.2 },
 { 2, 8, 9.6 },
 { 1, 1, 1.0 }
};
```

#### Poznámka:

- Počet prvků — jednotlivých struktur — tohoto pole lze zjistit pomocí příkazu:

```
pocet = sizeof(b) / sizeof(PRIKLAD);
```

Stejně jako u polí lze i u struktur inicializovat pouze globální nebo statické lokální struktury.

```
/* ukazka chybyne inicializace */
```

```
main()
{
 PRIKLAD a = { 1, 2, 6.4 }; /* automaticka */
}
```

```
/* ukazka spravne inicializace */
```

```
main()
```

```
{
 static PRIKLAD a = { 1, 2, 6.4 }; /* staticka lokalni */
}

/* ukazka spravne inicializace */
PRIKLAD a = { 1, 2, 6.4 }; /* staticka globalni */
main()
{
}
```

## 14.2 Výčtový typ

Výčtový typ (enumerate type), který je podobný výčtovému typu v Pascalu, se v C programech využívá velmi často. Dovoluje totiž velmi přehlednit program a zvýšit jeho modularitu.

Výčtový typ v C má sice podobnou konstrukci jako dříve probraná struktura, ale zcela jinou filosofii práce. Pomocí něho lze snadno definovat seznam symbolických konstant<sup>8</sup>, které mohou být — a nejčastěji jsou — vzájemně závislé.

Výčtový typ lze opět definovat různými způsoby<sup>9</sup>, z nichž preferujeme vždy definici pomocí `typedef`, tedy např.:

```
typedef enum {
 MODRA, CERVENA, ZELENA, ZLUTA /* zadny strednik ! */
} BARVY;
```

```
BARVY c, d;
c = MODRA;
d = CERVENA;
```

#### Poznámky:

- Položky výčtového typu nejsou *l-hodnoty*.
- Protože jsou položky v podstatě totéž, co symbolické konstanty, píšeme je z konvence velkými písmeny.
- Pokud explicitně nepřiradíme číselné hodnoty jednotlivým prvkům vyjmenovaného typu, pak mají tyto prvky implicitní hodnoty 0, 1, 2 atd.

<sup>8</sup> Takto definované symbolické konstanty nemají sice nic společného s preproceessingem, ale používají se stejným způsobem.

<sup>9</sup> Viz str. 90.



- Výčtový typ se použije vždy, mají-li konstanty nějakou vzájemnou souvislost, např. booleovské hodnoty *TRUE* a *FALSE* nebudou definovány jako:

```
#define FALSE 0
```

```
#define TRUE 1
```

ale spíše jako:

```
typedef enum {
 FALSE, TRUE
} BOOLEAN;
```

Takto definované konstanty pak lze využívat v nejrůznějších případech např.: `if (isdigit(c) == FALSE)`

čili není vůbec nutné definovat proměnnou tohoto výčtového typu. Stačí pouze definice výčtového typu *BOOLEAN*.

- Výčtový typ byl přidán do ANSI C a původní K&R verze C ho neobsahovala.

Jednotlivé definované proměnné typu *enum* jsou vnitřně reprezentovány jako paměťově nejméně náročný znaménkově chápaný celočíselný typ, který ještě může obsahovat hodnoty daného typu *enum*. Definovali-li bychom proměnnou dobře jako: `BOOLEAN dobre;`

pak bude proměnná *dobre* vnitřně reprezentována typem *signed char*.

Občas se také u vyjmenovaného typu používají explicitní inicializace. Je nutné poznamenat, že je možné explicitně inicializovat jen některé prvky a pro zbývající prvky inicializované implicitně pak platí, že jejich hodnota je vždy o 1 větší než hodnota předchozího prvku, např.:

```
typedef enum {
 MODRA = 0,
 CERVENA = 4,
 ZELENA = 2,
 ZLUTA /* neinicializovano */
} BARVY;
```

Zde má položka *ZLUTA* hodnotu 3.

Toto je nejhorší možný příklad inicializace, protože:

- 1) Když už položky inicializujeme, pak se snažíme srovnat položky podle velikosti.
- 2) Použijeme-li<sup>10</sup> explicitní inicializaci pak inicializujeme vždy všechny prvky.

<sup>10</sup> Musí pro to být rozumný důvod — viz např. str. 249.

### Pozor:

Častou chybou je představa, že je možné vytisknout jméno položky výčtového typu jako řetězec, tedy např.:

```
c = MODRA;
printf("Barva byla %s \n", c); /* chybne */
```

Je možné vytisknout pouze hodnotu položky výčtového typu a je vhodné ji předtím přetypovat na *int*, tedy např.:

```
printf("Barva mela cislo %d \n", (int) c);
```

Potřebujeme-li skutečně vytisknout jméno položky, pak je vhodným řešením např. použití přepínače *switch*, což je univerzální, ale poněkud rozvleklé řešení. Je výhodné v případě, že jsou položky inicializovány různými hodnotami např.:

```
typedef enum {
 MODRA = 5,
 CERVENA = 8,
 ZELENA = 11,
 ZLUTA = 15
} BARVY;
```

```
switch (barva) {
 case MODRA :
 printf("Byla to barva Modra");
 break;
```

Druhou, častěji využívanou možností, jak vytisknout jméno položky, je využití pole pointerů na *char*<sup>11</sup>. Tento způsob je elegantní, ale prakticky použitelný jen pro neinicializovaný výčtový typ, kdy položky mají hodnoty od 0, tedy např.:

```
typedef enum {
 MODRA, CERVENA, ZELENA, ZLUTA
} BARVY;
```

```
BARVY barva = MODRA;
char *nazvy[] = { "Modra", "Cervena", "Zelena", "Zluta" };
printf("Byla to barva %s \n", nazvy[barva]);
```

<sup>11</sup> Podrobně viz str. 224.



## 14.3 Uniony

Kdybychom hledali v Pascalu obdobu pro typ **union** v C, pak se mu nejvíce podobá Pascalský variantní záznam.

Datový typ **union** znamená, že se vyhradí paměť pro největší položku ze všech položek v **unionu** definovaných. Všechny položky **unionu** se překrývají<sup>12</sup>, což znamená, že v **unionu** může být v jednom okamžiku pouze jedna položka.

**Uniony** se v praxi používají málokdy a použijí-li se, měl by pro to být skutečný důvod. Jedním z důvodů může být potřeba šetřit paměť a **union** se tedy používá hlavně ve velkých polích.

Syntaxe **unionu** je velmi podobná již známé syntaxi struktury. Uvádíme porovnání pascalského variantního záznamu a Céčkovského **unionu**:

| Pascal             | C                             |
|--------------------|-------------------------------|
| TYPE typ = RECORD  | union typ {                   |
| spolecne polozky;  | /* spolecne polozky nejsou */ |
| CASE varTyp OF     |                               |
| val1 : varianta_1; | varianta_1;                   |
| ...                | ...                           |
| valn : varianta_n; | varianta_n;                   |
| END;               | }                             |

Stejně jako u struktur, i typ **union** a proměnná typu **union** mohou být definovány různými způsoby — viz str. 231.

Opět se preferuje definice pomocí **typedef**, která bude dále výhradně používána, např.:

```
typedef union {
 char c;
 int i;
 float f;
} ZN_INT_FLT;

ZN_INT_FLT a, *p_a = &a;
```

K jednotlivým položkám **unionu** se přistupuje naprosto stejně jako k položkám struktury, např.:

<sup>12</sup> Ve struktuře by ležely v paměti za sebou.

```
a.c = '#';
p_a->i = 1; /* premaze znak '#' */
a.f = 2.3; /* premaze cislo 1 */
```

### Pozor:

Je nutné si uvědomit, že **union** neposkytuje informaci o typu prvku, který do něj byl naposledy uložen!

Tento problém se často řeší tak, že se **union** vloží do struktury, jejíž první položka je výčtový typ a druhá položka **union**, např.:

```
typedef enum {
 ZNAK, CELE, REALNE
} TYP;

typedef union {
 char c;
 int i;
 float f;
} ZN_INT_FLT;

typedef struct {
 TYP typ;
 ZN_INT_FLT polozka;
} LEPSI_UNION;
```

Jak se podobná konstrukce používá, je ukázáno v následujícím příkladě.

### Příklad:

Program čte 10 krát z klávesnice. Pokaždé je nejdříve přečten jeden znak. Je-li to "I" nebo "i" bude následovat celé číslo, je-li to "C" nebo "c" bude následovat znak. Načtené hodnoty se ukládají do pole pomocí **unionu**. Na závěr se celé pole vytiskne.

Je použit **union** ve struktuře, aby bylo možné zaznamenat, jaký typ prvku je v **unionu** uložen.

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

#define POCET 10
/* vyprazdneni bufferu klavesnice
 - ukoncuji strednik se doda pri volani */
#define cisti() while (getchar() != '\n')
```



```

typedef enum {
 CISLO = 'I',
 ZNAK = 'C'
} TYP;

typedef union {
 char c;
 int i;
} CHARINT;

typedef struct {
 CHARINT hodnota;
 TYP typ;
} PRVEK;

PRVEK pole[POCET];

main()
{
 int i, c;

 for (i = 0; i < POCET; i++) {
 printf("%d. polozka : \ntyp : ", i + 1);
 c = toupper(getchar());
 cisti();
 switch (c) {
 case CISLO :
 pole[i].typ = CISLO;
 printf("cislo : ");
 scanf("%d", &pole[i].hodnota.i);
 break;
 case ZNAK :
 pole[i].typ = ZNAK;
 printf("znak : ");
 scanf("%c", &pole[i].hodnota.c);
 break;
 default :
 printf("Neznamy typ \n");
 break;
 }
 }
}

```

```

 }
 cisti();
}

/* tisk nactenych hodnot */
for (i = 0; i < POCET; i++) {
 if (pole[i].typ == CISLO)
 printf("%d. polozka = %d \n", i + 1, pole[i].hodnota.i);
 else
 printf("%d. polozka = %c \n", i + 1, pole[i].hodnota.c);
}
}

```

**Poznámka:**

- V tomto příkladě bylo vhodné použít výčtový typ s inicializovanými položkami, protože to zjednodušilo vazbu mezi vstupem z klávesnice a přepínačem switch.

**14.3.1 Rozdíl mezi variantním záznamem v Pascalu a unionem**

Jak bylo vidět z porovnání variantního záznamu a unionu (viz str. 248), C nepodporuje v unionu společné položky. Proto je možné Pascalský variantní záznam přepsat do C pouze pomocí kombinace struktur a unionů, tedy trochu komplikovaněji.

**Příklad:**

Variantní záznam v Pascalu:

```

TYPE ABC = RECORD
 a, b : INTEGER;
CASE BOOLEAN OF
 TRUE : (i, j : INTEGER);
 FALSE : (f, g : REAL);
END;

```

Odpovídající datový objekt v C:

```

typedef enum {
 FALSE, TRUE
} BOOLEAN;

```



```
typedef union {
 struct {
 int i, j;
 } v1;
 struct {
 float f, g;
 } v2;
} VARIANTY;
```

```
typedef struct {
 int a, b; /* společné položky */
 BOOLEAN cele; /* rozhodovací prvek */
 VARIANTY var;
} ABC;
```

Definice proměnné a přístup k jejím prvkům:

```
ABC x;
x.a = 1;
x.cele = TRUE;
x.var.v1.i = 3;
x.cele = FALSE;
x.var.v2.f = 5.6;
```

#### Co je dobré si uvědomit:

- Struktury jsou uloženy v bloku souvislé paměti a mohou v nich být "výplně", tzn. že položky nejsou nutně uloženy bezprostředně za sebou. Velikost struktury je vhodné získávat jedině pomocí operátoru `sizeof`.
- Se strukturou jako parametrem funkce se pracuje ve funkci někdy pomocí operátoru `->` i když to byla struktura definovaná staticky.
- Uniony jsou struktury, jejichž každá položka má offset 0 a tedy jednotlivé položky se při nevhodném použití navzájem přepisují.
- Výčtový typ lze s výhodou použít pro definování souvisejících symbolických konstant.
- Výčtový typ výrazně podporuje čitelnost a bezpečnost programu a doporučuje se ho používat.

#### Časté chyby:

```
struct test {
 int a, b;
};
```

tento typ definice nepoužívat!

test x;      má být:    struct test x;

```
typedef struct link {
 int hodnota;
 LINK *dalsi;
} LINK;
```

má být:    struct link \*dalsi;

```
typedef struct {
 int a, b;
} AAA;
AAA x, *p_x = &x;
*p_x.a = 1;
```

má být:    p\_x->a = 1;

```
typedef enum {
 blue, red;
} BARVY;
```

zde není středník  
jména položek je vhodné psát velkými písmeny

#### Cvičení:

- 1) Definujte strukturu se třemi prvky (`float`, `char` a `int` v tomto pořadí) a `union` s týmiž prvky. Zjistěte adresy struktury, `unionu` a všech jejích položek.
- 2) Definujte strukturu, která bude mít položky `jmeno` a `stari`. Napište funkci `napln()`, která tuto strukturu naplní vašimi daty. Pointer na strukturu bude předáván jako první parametr funkce. Tutéž strukturu alokujte dynamicky a naplňte ji daty vaší přítelkyně nebo přítele opět pomocí volání funkce `napln()`. Obsahy obou struktur vytiskněte.
- 3) Upravte předchozí program tak, že se jméno a věk budou číst ze souboru (oba údaje budou na jedné řádce a soubor nebude mít více než 1000 řádek). Načtené hodnoty se budou ukládat do statického pole struktur. Vypočtete průměrný věk a vytiskněte jména všech lidí, kteří jsou v tomto věku.
- 4) Předchozí program upravte tak, že místo do pole struktur se budou načtené údaje ukládat do jednosměrně zřetěženého dynamického seznamu těchto struktur. Ten se bude vytvářet dynamicky podle aktuální délky čteného souboru.



- 5) Vytvořte program, který pomocí funkce `fgets()` přečte soubor a uloží ho do paměti po řádcích pomocí jednosměrně seřazeného seznamu struktur. Položkami struktury budou mimo jiné řetězec max. délky 80 a aktuální délka řádky. Vytiskněte nejdelší řádku souboru.
- 6) Upravte předchozí program tak, aby položkami struktury byl pointer na řetězec a délka aktuální řádky. Vypočtete průměrnou délku řádky a vytiskněte všechny řádky, které mají tuto délku.
- 7) Napište program, který přečte soubor délky max. 1000 řádek a do pole struktur uloží délku řádky. Do prvků vhnížděné struktury `pocet` uloží jednak počet písmen na této řádce, a pak i počet ostatních znaků. Zjistěte počet písmen v souboru.
- 8) Napište program, který bude zjišťovat, zda se soubor, jehož jméno bude zadáno z klávesnice, nalézá na disku nebo ne. K tomuto účelu vytvořte funkci `BOOLEAN je_tu(char *jmeno)`. Typ `BOOLEAN` vytvořte pomocí výčtového typu s položkami `ANO` a `NE`.
- 9) Definujte výčtový typ `AUTOMOBILY` a vytiskněte jak názvy, tak i hodnoty jednotlivých položek. Názvy položek tiskněte:
  - po znacích pomocí dvou cyklů
  - pomocí `printf()`
  - pomocí `puts()`
- 10) Předchozí výčtový typ zkuste inicializovat, jak částečně, tak i úplně a opět tiskněte názvy jeho položek.

## 15 Bitové operace a bitové pole

Až dosud jsme se zmiňovali o vyšší úrovni jazyka C, což lze dokumentovat např. tím, že téměř každá konstrukce v C měla svůj protějšek v Pascalu. Bitové operace se týkají nižší úrovně jazyka — vykazují totiž rysy jazyka assembleru — a standardní Pascal nic takového nepodporuje.

Obecně platné doporučení je vyhýbat se těmto operacím. Většinou totiž vyžadují hlubší znalosti systému, např. formát uložení čísel, a i malá chyba může mít velký negativní dopad.

Jsou ovšem situace, kdy se jejich použití nevyhneme. Buď již z důvodu nemožnosti napsat nějaký úsek programu pomocí "vyšších" příkazů jazyka<sup>1</sup>, nebo, což je mnohem častější, že při vhodném použití dovolí v určitých případech výrazně urychlit program.

### 15.1 Operace s jednotlivými bity

Pro účely manipulací s bity C poskytuje 6 operátorů:

|                       |                                                                               |
|-----------------------|-------------------------------------------------------------------------------|
| <code>&amp;</code>    | bitový součin — AND                                                           |
| <code> </code>        | bitový součet — OR                                                            |
| <code>^</code>        | bitový exklusivní součet — XOR (nonekvivalence) <sup>2</sup>                  |
| <code>&lt;&lt;</code> | posun doleva                                                                  |
| <code>&gt;&gt;</code> | posun doprava                                                                 |
| <code>~</code>        | jedničkový doplněk — negace bit po bitu <sup>3</sup> — <u>unární</u> operátor |

#### Poznámky:

- Argumenty bitových operací nesmějí být proměnné typů `float`, `double` a `long double`.
- K&R verze C umožňuje provádět bitové operace pouze s proměnnými typu `unsigned int`.
- ANSI verze C toto omezení nemá a bitové operace je možné použít i pro typy `signed`. Z hlediska čitelnosti programu je ale lepší používat typ `unsigned`.

<sup>1</sup> Protože pro ně je nejmenší jednotkou informace Byte.

<sup>2</sup> Znak "stříška" nebo "šipka nahoru".

<sup>3</sup> Znak "tilda" neboli "vlnka".



## 15.1.1 Bitový součin

$i$ -tý bit výsledku bitového součinu:  $x \& y$   
bude 1, pokud  $i$ -tý bit  $x$  a  $i$ -tý bit  $y$  budou 1, jinak bude 0

Čili jednotlivé bity výsledku budou záležet na jednotlivých bitech operandů.

Příklad:

Následující makro může být použito pro test, zda je číslo liché<sup>4</sup>.

```
#define je_liche(x) (1 & (unsigned)(x))
```

Bitový součin se často používá pro vymaskování (nastavení na nulu) určitých bitů, např. chceme-li proměnnou typu `int` převést na ASCII znak, tedy využít jen nejnižších 7-mi bitů:

```
c = c & 0x7F; /* 0x7F je 0000 0000 0111 1111 */
nebo zkráceně:
c &= 0x7F;
```

Poznámka:

- Uvědomme si, že je rozdíl mezi bitovým součinem a logickým součinem<sup>5</sup>:

```
unsigned int i = 1, j = 2, k, l;
k = i && j; /* k == 1 */
l = i & j; /* l == 0 */
```

protože:  $1 = 0000\ 0001$   
 $2 = 0000\ 0010$

## 15.1.2 Bitový součet

$i$ -tý bit výsledku bitového součtu:  $x \mid y$   
bude 1, pokud  $i$ -tý bit  $x$  nebo  $i$ -tý bit  $y$  bude 1, budou-li oba nulové, bude výsledek 0

Bitový součet se často používá pro nastavení určitých bitů na 1, přičemž ostatní bity nechá nedotčeny.

Příklad:

Následující makro vrátí sudé číslo o jedničku větší (tj. liché číslo) a liché číslo vrátí nezměněno.

```
#define na_liche(x) (1 | (unsigned)(x))
```

<sup>4</sup> Lichá čísla mají nultý bit nastaven na 1.

<sup>5</sup> Podobně to platí i pro bitový a logický součet.

## 15.1.3 Bitový exkluzivní součet

$i$ -tý bit výsledku bitového XOR:  $x \sim y$   
bude 1, pokud  $i$ -tý bit  $x$  se nerovná  $i$ -tému bitu  $y$   
budou-li oba nulové, nebo oba jedničkové bude výsledek 0

Tato operace se dá využít k porovnání dvou celých čísel:

```
if (x ~ y)
/* čísla jsou rozdílná */
```

## 15.1.4 Operace bitového posunu doleva

Příkaz:  $x \ll n$

posune bity v  $x$  doleva o  $n$  pozic

Při tomto posunu se zleva bity ztrácí — jsou vytlačovány — a zprava jsou doplňovány 0

Bitový posun doleva se občas používá pro rychlé násobení dvěma, respektive mocninou dvou, např. příkaz:

```
x = x << 1; nebo x <<= 1;
```

vynásobí  $x$  dvěma

nebo příkaz:  $x \ll= 3$ ; násobí  $x$  8-mi

## 15.1.5 Operace bitového posunu doprava

Příkaz:  $x \gg n$

posune bity v  $x$  doprava o  $n$  pozic

Při tomto posunu se zprava bity ztrácí<sup>6</sup> — jsou vytlačovány — a zleva jsou doplňovány 0

Bitový posun doprava má opačný význam než posun doleva, tedy celočíselné dělení dvěma, respektive mocninou dvou, např. příkaz:

```
x = x >> 1; nebo x >>= 1;
```

dělí  $x$  dvěma

nebo příkaz:  $x \gg= 4$ ; dělí  $x$  16-ti

Poznámka:

- Tyto operace "násobení" a "dělení" dvěma pomocí posuvů jsou rychlejší než skutečné násobení a dělení. Při výpočtech s celými čísly, kde jde o rychlost, je dobré zamyslet se nad tím, zda by se nedaly využít.

<sup>6</sup> To platí pro proměnnou typu `unsigned`. Pro typ `signed` je tato operace implementačně závislá.



Příklad:

Než násobit 80, je lepší "násobit" 64 a 16 a sečíst výsledek:

```
i = j * 80; /* pomalejší */
i = (j << 6) + (j << 4); /* rychlejší */
```

Pozor:

Je třeba si uvědomit, že priority operátorů `>>`, a `<<` jsou velmi nízké, takže je nutno téměř vždy závorkovat.

Příklad:

Operátor `>>` se také často používá pro získání hodnoty konkrétního bitu. Používá se jednoduchý trik, kdy se s bity posouvá tak dlouho, až je požadovaný bit na nejnižší pozici.

Funkce `bit()` vrátí hodnotu `i`-tého bitu svého parametru.

```
#define ERROR -1
#define CLEAR 1
#define BITU_V_CHAR 8

int bit(unsigned x, unsigned i)
{
 if (i >= sizeof(x) * BITU_V_CHAR)
 return (ERROR);
 else
 return ((x >> i) & CLEAR);
}
```

Příkaz: `x >> i`

posune `i`-tý bit na poslední pozici doprava a příkaz: `& CLEAR`

nastaví na nulu všechny vyšší bity.

## 15.1.6 Negace bit po bitu

Pro tuto akci se často používá také název *jedničkový doplněk*.

Příkaz: `~x`

převrátí nulové bity na jedničkové a naopak.

Tento operátor se používá např. v situacích, kdy se chceme vyhnout počítačově závislé délce celého čísla. Například příkaz:

```
unsigned int x;
```

```
x &= 0xFFFF0;
```

nastaví na nulu nejnižší čtyři bity `x`. Bude ale pracovat správně jen na počítačích, kde platí `sizeof(int) == 2`

Řešením je příkaz: `x &= ~0xF;`

který bude pracovat správně na všech typech počítačů.

## 15.1.7 Způsoby práce se skupinou bitů

Často se bitové operace používají pro práci se skupinou bitů, kterou např. představuje *stavové slovo* definované jako:

```
unsigned int status;
```

Poznámka:

- Pro označení krajních bitů slova se často používají anglické zkratky:  
MSB – nejvýznamnější (nejlevější) bit (*most significant bit*)  
LSB – nejméně významný (nejpravější) bit (*least significant bit*)

Nejprve se definují konstanty, které určí pozice příznakových bitů ve stavovém slově. Např. použijeme bity 3, 4, a 5 pro příznaky číst, psát, vymazat (READ, WRITE, DELETE).

```
bit číslo: 7 6 5 4 3 2 1 0
 [] [] [] [] [] [] [] []
 MSB LSB
```

```
#define READ 0x8
#define WRITE 0x10
#define DELETE 0x20
```

Po této přípravě je možné provést:

- nastavení všech příznaků na 1 je:  
`status |= READ | WRITE | DELETE;`
- nastavení příznaků READ a WRITE na 1 je:  
`status |= READ | WRITE;`
- nastavení všech příznaků na 0 je:  
`status &= ~(READ | WRITE | DELETE);`
- nastavení příznaku READ na 0 je:  
`status &= ~READ;`
- test zda jsou oba příznaky WRITE a DELETE nulové:  
`if (!(status & (WRITE | DELETE)))`

## 15.2 Bitové pole

Bitové pole si lze představit jako strukturu, jejíž velikost je ale pevně omezena velikostí typu `int`. Nejmenší délka jedné položky v bitovém poli je právě 1 bit. Bitové pole se také definuje velmi podobně jako se definuje



struktura, odlišnost je pouze v tom, že je každá položka bitového pole určena jednak svým jménem a jednak délkou v bitech.

ANSI C umožňuje definici prvků jak **signed int**, tak i **unsigned int** a je proto vhodné vždy uvést, zda bude položka znaménková či bezznaménková.

Bitové pole má dvě základní oblasti použití:

- 1) Uložení několika celých čísel v jednom slově, což bývá používáno zejména pro šetření paměti, ovšem nepříliš často.
- 2) Pro přístup k jednotlivým bitům slova pomocí identifikátorů, což je použití mnohem častější, protože se pak operace s jednotlivými bity provádějí velmi elegantně a přehledně.

#### Příklad:

Ukázka využití bitového pole pro uložení datumu zhuštěně do jednoho slova — předpokládáme `sizeof(int) == 2`. Datum bude uloženo tak, že položka **den** bude zabírat nejnižších 5 bitů, položka **mesic** následující 4 bity a položka **rok** zbývajících 7 bitů.

Protože však 7 bitů je na letopočet málo — nejvyšší číslo sem uložené by bylo:  $2^7 - 1 = 127$

použijeme triku, kdy k datumu zde uloženému vždy přičteme konstantu 1980<sup>7</sup>.

```
typedef struct {
 unsigned den : 5; /* bity 0 - 4 */
 unsigned mesic : 4; /* bity 5 - 8 */
 unsigned rok : 7; /* bity 9 - 15 */
} DATUM;
```

```
DATUM dnes, zitra;
dnes.den = 25;
dnes.mesic = 6;
dnes.rok = 1992 - 1980;
zitra.den = dnes.den + 1;
```

#### Příklad:

V této části programu se pokusíme pomocí bitového pole řešit případ z předchozí podkapitoly, kdy jsme pracovali se stavovým slovem a příznakovými bity **READ**, **WRITE** a **DELETE**. Všechny údaje zůstávají v platnosti.

```
typedef struct {
 unsigned zacatek : 3; /* bity 0 - 2 */
 unsigned read : 1; /* bit 3 */
```

<sup>7</sup> Tento způsob uložení datumu používá MS-DOS pro práci se soubory.

```
unsigned write : 1; /* bit 4 */
unsigned delete : 1; /* bit 5 */
} FLAG;
```

Položka **zacatek** je použita pro přeskočení bitů 0, 1 a 2.

Po definici proměnné: **FLAG status**;

je možné snadno pracovat s jednotlivými bity:

- Nastavení všech příznaků na 1 je:

```
status.read = status.write = status.delete = 1;
```

- nastavení všech příznaků na 0 je:

```
status.read = status.write = status.delete = 0;
```

- test zda jsou oba příznaky **WRITE** a **DELETE** nulové:

```
if (! (status.write | status.delete))
```

#### Poznámky:

- Adresy (&) položek bitových polí není možné získat a stejně tak není možné používat pointerů na jednotlivé položky.
- Častou chybou je idea, že položky bitového pole jsou ukládány v pořadí od vyšších bitů k nižším — od MSB k LSB. Ve skutečnosti je to právě naopak — začíná se od LSB.

#### Cvičení:

- 1) Napište funkci `int delka_int(void)`, která vrátí délku proměnné typu `int` v bitech. Zajistěte, aby funkce pracovala správně na libovolném typu počítače.
- 2) Napište funkci `unsigned rotuj_doprava(unsigned x, int n)`, která způsobí rotaci (ne posun!!!) čísla `x` o `n` bitů doprava.
- 3) Napište funkci `unsigned invert(unsigned x, int p, int n)`, která invertuje (změní 1 za 0 a naopak) `n` bitů proměnné `x` počínaje od pozice `p` včetně. Ostatní bity zůstanou nezměněny.
- 4) Vytvořte program, který bude využívat bitového pole pro úschovu datumu. Položky pole budou:

```
bit 15 - 9 : rok, ke kterému se přičte konstanta 1980
 tedy 12 znamená ve skutečnosti rok 1992
```

```
bit 8 - 5 : měsíc
```

```
bit 4 - 0 : den
```

Program vyzkoušejte tak, že načtete z klávesnice datum, uložíte ho do bitového pole, to vytisknete jako typ **unsigned** a pak datum vytisknete jako opravdové datum. Pro tisk proměnné jako **unsigned** se pokuste využít **unionu**.



## 16 Tabulka preferencí

Tabulka preferencí — neboli priorit operátorů — je velmi důležitou pomůckou zkušenějších a zvláště experimentujících programátorů v C. K tomu pouze jednu poznámku — komplikovaný výraz lze v naprosté většině případů rozepsat do několika méně komplikovaných, byť za cenu částečné ztráty efektivnosti programu. Je to někdy rychlejší cesta k odladění programu než velké přemítání nad prioritami operátorů.

Při práci s tabulkou preferencí platí dvě zásady:

- 1) Čím nižší číslo skupiny, tím vyšší priorita, neboli tím dříve bude konkrétní operátor pracovat.

Máme-li například výraz: `x++ - y / 4`  
 kterému předcházela definice: `signed int x = 3, y = 9;`  
 pak se vypočte nejdříve výraz: `x++` priorita 2), výsledek 4  
 pak výraz: `y / 4` priorita 3), výsledek 2  
 a nakonec výraz: `4 - 2` priorita 4), výsledek 2

- 2) Pokud se ve výrazu vyskytnou operátory se stejnou prioritou (což jsou položky ve stejné skupině, protože na pořadí ve skupině nezáleží), pak o

Vlastní tabulka vypadá takto:

| pr. | Operátory                         | Asociativita  |
|-----|-----------------------------------|---------------|
| 1   | () [] -> .                        | zleva doprava |
| 2   | ! ~ ++ -- + - (typ) * & sizeof    | zprava doleva |
| 3   | * / %                             | zleva doprava |
| 4   | + -                               | zleva doprava |
| 5   | << >>                             | zleva doprava |
| 6   | < <= > >=                         | zleva doprava |
| 7   | == !=                             | zleva doprava |
| 8   | &                                 | zleva doprava |
| 9   | ~                                 | zleva doprava |
| 10  |                                   | zleva doprava |
| 11  | &&                                | zleva doprava |
| 12  |                                   | zleva doprava |
| 13  | ?:                                | zprava doleva |
| 14  | = += -= *= /= %= >>= <<= &=  = ^= | zprava doleva |
| 15  | ,                                 | zleva doprava |

Význam jednotlivých operátorů je:

- 1) primární operátory — vyhodnocování zleva doprava
  - () uzávorkování
  - [] index do pole
  - > přístup k prvku struktury pomocí pointeru na strukturu
  - . přístup k prvku struktury pomocí jména struktury



- 4) aditivní operátory — vyhodnocování zleva doprava
  - + aritmetický součet
  - aritmetický rozdíl
- 5) operátory posunů — vyhodnocování zleva doprava
  - << posun bitů doleva
  - >> posun bitů doprava
- 6) relační operátory — vyhodnocování zleva doprava
  - < menší než
  - <= menší nebo rovno než
  - > větší než
  - >= větší nebo rovno než
- 7) operátory rovnosti — vyhodnocování zleva doprava
  - == rovnost
  - != nerovnost
- 8) & operátor bitového součinu (AND) — vyhodnocování zleva doprava
- 9) ^ operátor exkluzivního bitového součtu (XOR) — vyhodnocování zleva doprava
- 10) | operátor bitového součtu (OR) — vyhodnocování zleva doprava
- 11) && operátor logického součinu (AND) — vyhodnocování zleva doprava
- 12) || operátor logického součtu (OR) — vyhodnocování zleva doprava
- 13) ?: ternární podmínkový operátor — vyhodnocování zprava doleva
- 14) přiřazovací operátory — vyhodnocování zprava doleva
  - = přiřazení
  - += součet a přiřazení
  - = rozdíl a přiřazení
  - \*= součin a přiřazení
  - /= dělení a přiřazení
  - %= dělení modulo a přiřazení
  - >>= bitový posun doprava a přiřazení
  - <<= bitový posun doleva a přiřazení
  - &= bitový součin a přiřazení
  - |= bitový součet a přiřazení
  - ^= nonekvivalence a přiřazení
- 15) , operátor čárky — vyhodnocování zleva doprava

## Literatura

- [AA88] Anderson Paul L. - Anderson Gail C.: Advanced C: Tips and Techniques  
Hayden Books, U.S.A., 1988
- [BS89] Brodský Jan - Skočovský Luděk: Operační systém Unix a jazyk C  
SNTL, Praha, 1989
- [DEC89] Digital Equipment Corporation: Guide to VAX C  
Digital Equipment Corporation, U.S.A., 1989
- [DM88] Darnel Peter A. - Margolis Philip E.: Software Engineering in C  
Springer-Verlag New York Inc., U.S.A., 1988
- [Eck90] Eckel Bruce: Using C++  
Osborne McGraw-Hill, U.S.A., 1990
- [HRŠ92] Herout Pavel - Rudolf Vladimír - Šmrha Pavel: ABC programátora v jazyce C  
KOPP, České Budějovice, 1992
- [KR78] Kernighan Brian W. - Ritchie Dennis M.: The C Programming Language  
Prentice Hall, U.S.A., 1978  
Slovenský překlad — Programovací jazyk C  
Alfa, Bratislava, 1986
- [KR88] Kernighan Brian W. - Ritchie Dennis M.: The C Programming Language — Second Edition  
Prentice Hall, U.S.A., 1988
- [MQ86] Miller Lawrence H. - Quilici Alexander E.: Programming in C  
John Wiley & sons, U.S.A., 1986
- [MS88] Müldner Tomasz - Steele Peter W.: C as a Second Language  
Addison-Wesley, U.S.A., 1988
- [Šmr90] Šmrha Pavel: Úvod do Turbo C jazyka  
Skriptum postgraduálního kurzu "Informatika", VŠSE Plzeň, 1990



# Rejstřík

absolutní adresa 146

alloc.h 167

anonymní specifikace 116

argc 226

argv 226

ASCII 20

assignment 25

atof() 204

atoi() 204

atol() 204

auto 123

bílé znaky 19

blok 26, 127

boolean 23, 42

break 49, 55

bufferování 63

calloc() 170

casting 87

cfree() 170

char 23, 194

chyba +1 176

clock() 98

clock-t 99

compiler 18

const 79, 126

continue 49, 56

CR 69

ctype.h 95

DATE 90

debugger 18

default 55

defined 104

define 91, 93

definice 24, 107

deklarace 24, 107

dekrement 30

dereferenční operátor 147

do-while 49, 51

dolní mez pole 175

double 23, 28, 36, 255

EBCDIC 20

editor 17

efektivnost programu 262

elif 104

else(p) 102

endif 102

enum 245

EOF 65, 71

EOLN 69

error 104

escape character 28

.EXE 18

exit() 60

explicitní konverze 87

expression 25

extern 123

FALSE 23, 42, 246

fclose() 67, 73

feof() 71

fgets() 207

FILE 65, 90

float 23, 28, 36, 255

fopen() 73

for 49, 52

fprintf() 67

fputs() 208

fread() 82

free() 167, 169

fscanf() 67

fseek() 83

ftell() 83

funkční prototyp neúplný 116

funkční prototyp 115

fwrite() 82

getc() 67

getchar() 33, 143

gets() 206

globální definice 119

globální deklarace 118

goto 59

heterogenní datový typ 231

hexadecimální číslo 37

homogenní datový typ 231

if(p) 102

if-else 47

if 46

include 19, 96

indexovaná proměnná 177

inicializace 122, 142, 223, 244, 246

inkrement 30

interface 129

int 23, 110

isalnum() 95

isalpha() 95

isascii() 95

iscntrl() 95

isdigit() 95

isgraph() 95

islower() 95

isprint() 95

ispunct() 95

isspace() 95

isupper() 95

isxdigit() 95

jedničkový doplněk 258

jméno funkce 158

klíčová slova 20

komentář vhnížděný 102

komentáře 21

kompilátor 18

konstantní pointer 179

konstanty šestnáctkové 27

konstanty desítkové 27

konstanty osmičkové 27

konstanty reálné 28

konstanty záporné 28

konstanty znakové 28

konstanty řetězcové 29

konverze explicitní 86

konverze implicitní 86

ladící program 18

LF 69

.LIB 18

line 90

LINE 90

linker 18, 130

literály 29



- lokální definice 120
- long double 23, 28, 36, 255
- long int 23
- long 23, 27, 36
- LSB 259
- main() 26, 110
- malloc() 167, 234
- maskování 256
- math.h 98
- mezera 19
- MSB 259
- návrat na začátek řádky 29
- nekonečný cyklus 50, 54
- nová stránka 29
- nová řádka 29
- nový řádek 19
- NULL 151, 168
- nulový pointer 197
- nulový řetězec 197, 206
- .OBJ 18
- operátor čárky 45
- optimalizace 93, 182
- paragraf 168
- písknutí 29
- podmíněný výraz 44
- pointer na pole 219
- pole pointerů 218, 247
- porovnání 42
- posun doleva 29
- pořádek 132
- pragma 90
- preference 262
- preprocesor 17
- printf() 34, 200
- priority operátorů 262
- priority 43
- procedury 112
- putc() 67
- putchar() 33
- puts() 207
- překladač 18
- přepínač 54
- přesměrování 75
- přetypování 87
- příkaz 25
- přiřazení 25, 42
- realloc() 184
- record 231
- redirekce 75
- referenční operátor 147
- register 125
- rekurze 95
- return 59, 111, 112
- rozhraní 129
- run-time 167
- rychlé dělení 257
- rychlé násobení 257
- scanf() 34, 198
- sestavovací program 18
- short int 23
- short 23
- signed int 260
- signed 23, 36, 255
- sizeof 163
- soubor binární 64, 79, 80
- soubor textový 64, 79
- sprintf() 204
- sscanf() 204
- stack 241
- statement 25

- static 108, 123, 130, 143
- stdarg.h 117
- stdaux 76
- STDC 90
- stderr 76
- stdin 75
- stdio.h 33
- stdlib.h 167
- stdout 75
- stdprn 76
- strcat() 202
- strchr() 202
- strcmp() 202
- strcpy() 202
- string.h 202
- string 194
- strlen() 202
- strncpy() 203
- strrchr() 203
- strstr() 203
- struct 231
- struktura 231
- swapování 100
- switch 54, 247
- symbolické konstanty 245
- tabulátor 19, 29
- ternární operátor 44
- time.h 98
- TIME 90
- toascii() 95
- tolower() 95
- toupper() 95
- TRUE 23, 42, 246
- typecasting 87
- typedef 149, 161, 170, 232
- tři tečky (ellipsis) 117
- uložení položek 239
- unární mínus 30
- unární plus 30
- undef 92
- ungetc() 78
- union ve struktuře 249
- union 248
- unsigned int 255, 260
- unsigned 23, 27, 36, 255
- variantní záznam 251
- velikost pole 183
- vhnížděná struktura 238
- vkládané funkce 93
- void 112, 156
- volání hodnotou 117, 240
- volání odkazem 153, 240
- volatile 127
- výčtový typ 245
- vymaskování 256
- výraz s přiřazením 29
- výraz 25
- while 49
- záznam 231
- zkrácené vyhodnocení 42



Vážení, obracíme se na Vás s nabídkou odborné technické literatury z produkce nakladatelství KOPP. Jedná se o tyto tituly (ceny včetně DPH):

## POČÍTAČE

|                                               |                                         |
|-----------------------------------------------|-----------------------------------------|
| Poprvé u počítače aneb Začínáme pracovat s PC | 128 stran, 39,- Kč                      |
| MS-DOS prakticky 6.2                          | 260 stran, 89,- Kč                      |
| MS-DOS prakticky 6.0                          | 232 stran, 89,- Kč                      |
| Začínáme s Windows 3.1                        | 136 stran, 59,- Kč                      |
| Začínáme s Windows CS 3.1                     | 136 stran, 69,- Kč                      |
| Norton Commander 4.0                          | 120 stran, 59,- Kč                      |
| AmiPro 3.0 a počítačová typografie            | 248 stran, 99,- Kč                      |
| Slovník počítačových zkratk                   | 124 stran, 69,- Kč                      |
| Programování v Turbo Pascalu 7.0              | 240 stran, 99,- Kč                      |
| Programování v Turbo Pascalu 6.0              | 195 stran, 79,- Kč                      |
| Turbo Vision 2.0                              | 196 stran, 89,- Kč                      |
| Sbírka úloh z jazyka Pascal                   | 128 stran, 59,- Kč                      |
|                                               | Volitelně disketa s příklady za 59,- Kč |
| Borland Pascal a objekty ve Windows           | 248 stran, 119,- Kč                     |
| Učebnice jazyka C 3. upravené vydání          | 272 stran, 139,- Kč                     |
|                                               | Volitelně disketa s příklady za 69,- Kč |
| Borland C++ - návod k použití                 | 188 stran, 99,- Kč                      |
| Objektově orientované programování v C++      | 208 stran, 139,- Kč                     |
|                                               | Volitelně disketa s příklady za 69,- Kč |
| Praktický úvod do TCP/IP                      | 112 stran, 89,- Kč                      |
| Internetworking pomocí TCP/IP                 | 148 stran, 149,- Kč                     |
| Programovací techniky                         | 192 stran, 89,- Kč                      |
| Abeceda OrCADu                                | 192 stran, 119,- Kč                     |
| Používáme AutoCAD Release 10 a Release 11     | 228 stran, 99,- Kč                      |
| Používáme AutoCAD Release 12 2. vydání        | 336 stran, 149,- Kč                     |
| Strategické hry na PC                         | 64 stran, 39,- Kč                       |

## ELEKTRONIKA

|                                              |                    |
|----------------------------------------------|--------------------|
| Amatérské elektronické konstrukce a zapojení | 112 stran, 59,- Kč |
| Elektronické hračky a přístroje              | 120 stran, 59,- Kč |
| Poznáváme elektroniku I                      | 184 stran, 69,- Kč |

## TECHNIKA

|                              |                    |
|------------------------------|--------------------|
| Tvorba technické dokumentace | 312 stran, 99,- Kč |
|------------------------------|--------------------|

## AUTO - MOTO

|                                                              |                                            |
|--------------------------------------------------------------|--------------------------------------------|
| Údržba a opravy vozů                                         | každý svazek přibližně 250 stran, 430,- Kč |
| FORD SIERRA 75 - 120 PS a diesel 6/82 - 2/93                 |                                            |
| FORD ESCORT/ORION 50-132 PS 8/80 - 8/90 a diesel 2/84 - 8/90 |                                            |
| FIAT UNO 45-105 PS od 9/82 a diesel 45/58 PS od 8/83         |                                            |
| VW GOLF II/JETTA diesel 54-80 PS 9/83 - 6/92                 |                                            |
| VW GOLF II/JETTA benzin 55-160 PS 9/83 - 6/92                |                                            |
| PEUGEOT 205 42 - 128 PS a diesel 60/64 PS od 9/83            |                                            |

Dále připravujeme knihy Údržba a opravy vozů  
OPEL KADETT E benzin 55-150 PS 9/84 až 8/91  
OPEL KADETT E diesel 54/57 PS 9/84 až 8/91  
RENAULT 19 / CHAMADE, FORD FIESTA, OPEL VECTRA

Připravované tituly si můžete objednat již dnes! Vaši objednávku zaevidujeme a knihu obdržíte okamžitě po jejím vydání.

Uvedený stav odpovídá stavu z října 1994.

Ceny jsou uvedeny včetně DPH.

Dealerské podmínky poskytneme ihned na vyžádání.

Žádejte zdarma podrobný katalog!

Knihy si můžete koupit za hotové

či objednat na dobírku i na fakturu na adrese:

KOPP nakladatelství, Máchova 16, 370 01 České Budějovice,  
tel/fax: 038-60243

Dodavatel pro Slovensko: ELFA s.r.o., Letná 9, 042 00 Košice  
tel/fax: 095-53839



27 června 1995

9. října 1995

18. listopad 1995

18. listopad 1995

1. října 1997

1. červce 1998

15. února 1999

9. dubna 1999

8. května 2000

Ing. Pavel Herout

Učebnice jazyka C

Třetí upravené vydání

Sazbu programem TeX připravil autor

Vydalo nakladatelství KOPP, Máchova 16, 370 01 České Budějovice  
jako svou 42. publikaci

Tisk PB tisk, Příbram

1994

10. dubna 1995

18. listopad 1995

18. listopad 1995

24. září 1996

24. února 1998

13. října 1998

3. června 1999

17. listopad 1999