Kurz jazyka C

Jan “Bazil” Otte

Verze: 1.0ca, 2001

Obsah

Poděkování

Za to, že jsem mohl napsat tuto příručku, vděčím mnoha lidem. Nebudu je zde všechny jmenovat, ale chtěl bych vyzdvihnout dva, kteří mají právě na tomto díle největší zásluhu.

Prvním z nich je můj táta, který mě přivedl k výpočetní technice vůbec a byl – a stále je – mi v mnoha věcech vzorem.

Druhým je profesor Jan Kučera, který mě přivedl k jazyku C a naučil mě jeho základy.

Děkuji.

Bazil qi*.

Úvod

Tento materiál vznikl z potřeby mít nějaký text, podle kterého bych mohl učit kurzy jazyka C. Po menší úvaze jsem ho ale trochu přepracoval, aby se z něj mohl učit kdokoliv, například i sám – tedy jsem doplnil příklady a podobně.

Snažil jsem se jej upravit do podoby, ze které bych se kdysi já sám rád C učil – do něčeho, co by bylo jasné, srozumitelné a názorné. Nakolik se mi to povedlo, to ale musíte určit vy.

Držíte tedy v ruce materiál k jazyce C. Netroufám si tvrdit, že je to kompletní popis jazyka, to ani náhodou. Je to věc, která by vás měla naučit všechny důležité prvky jazyka. Po prostudování těchto "skript" byste měli být opravdoví programátoři v C, schopní sami se zdokonalovat dál a dál[1].

Jak "skripta" procházet? Pokud jazyk neovládáte, je nejlepší začít od začátku a nepřeskakovat, zejména důležité je programovat všechna cvičení. Pokud si myslíte, že už něco víte, tak se vždy podívejte na prvních několik vět z podkapitoly a kdyžtak ji jen proleťte – ale cvičení programujte! Pokud už jazyk umíte, můžete si projít kapitolu 6, pravděpodobně tam i tak najdete něco zajímavého.

Ještě něco k obsahu skript. Budu předpokládat, že neovládáte žádný programovací jazyk – když jsem se C učil, velice mě štvalo, když se všude, v každém příkladě, odvolávali na Pascal[2]. Dost už planého tlachání a hodně štěstí při učení jazyka C.

Licence: Tento materiál by měl být dostupný ke stáhnutí ve formě `.ps` na mých stránkách (*http://www.fi.muni.cz/~xotte*). Můžete si jej pro osobní (nekomerční) účely rozmnožovat a předávat kamarádům atp. Pokud by podle nich chtěl někdo náhodou vyučovat, budu jen rád, ale může tak učinit pouze pokud se jedná o neplacený kurz/předmět/cokoliv. V jiných případech si musí napsat o dovolení (*mailto: xotte@informatics.muni.cz*). Text je možno upravovat, neztrácí však tím nic ze své nekomerčnosti. Pokud nějakou pasáž přidáte, je vaší povinností poslat mi ji a já ji mohu do textu zahrnout. Totéž se týká úprav pasáží nebo oprav chyb. Není možno vypustit nebo upravit tento odstavec (*Licence*).

[1] Za předpokladu že budete alespoň část z přečteného umět.

[2] I když jsem ho dost dobře uměl, připadalo mi tak, že C je forma Pascalu, což *není*.

Kapitola 1

Seznámení s jazykem C

1.1 Historie, klasifikace a specifikace jazyka C

Historie

Za původní podobu jazyka budeme považovat takzvanou K&R verzi, kterou vytvořili páni B. W. Kernighan a D. M. Ritchie, někdy kolem roku 1978, i když první verze jazyka existovaly již kolem roku 1973 a snad první zmínku najdeme o roce 1971[1]. Jazyk C vznikl z jazyka B (jak jinak), který zase vznikl z fortranu.

Jazyk C se vyvíjel paralelně s operačním systémem UNIX. Ve skutečnosti kromě několika málo původních verzí byl UNIX v jazyce C přímo napsán (asi od r.1973). Samozřejmě není třeba používat minulý čas, protože snad všechny OS založené na Unixu (včetně Linuxu) jsou stále v tomto jazyce napsány. O UNIXech ale někdy jindy.

Klasifikace

Jazyk C můžeme klasifikovat jako vyšší programovací jazyk. Co to je v. p. jazyk? Zkusím nějakou vlastní definici.

Takže nejprve nižší p. jazyk (*low-level*). V takovém jazyce píšete program za pomoci příkazů využívajících přímo HW daného stroje – tzn. datové typy odpovídají velikosti registrů, příkazy představují abstrakci nad instrukcemi procesoru nebo jim přímo odpovídají (*assembler*).

Zato programovací jazyk vyšší úrovně nabízí příkazy představující abstrakci nad určitými logickými kroky a abstraktní datové typy, většinou nezávislé na HW. Z toho plyne přenositelnost napsaného programu – logický krok můžete vykonat na kterémkoliv počítači, danou posloupnost pseudo-instrukcí pouze tam, kde existují všechny její pseudo-instrukce. Samozřejmě, logický krok nakonec na instrukce převést musíte, ale o to se již nebudete starat vy, nýbrž překladač jazyka – viz 1.2.

Pokud bychom šli do detailů, většina moderních *assemblerů*[2] už je také trochu *high-level*, takže je všechno vlastně otázka míry – zda vám jazyk poskytuje prvky ke strukturalizaci dat, rozumné volání funkcí, modularitu... Dost. C má všechny rysy vyššího programovacího jazyka.

[1]Na těchto verzích pracovali také Kernighan & Ritchie.

[2]Např. NASM

Specifikace

Kromě výše zmíněné specifikace K&R C, existují ještě další specifikace. Jedna z nejdůležitějších je "ANSI C" z roku 1989[3] a drtivá většina všech funkcí a vlastností ve všech učebnicích popisovaná, splňuje tento standard – v téměř všech kursech a předmětech se učí "ANSI C". I já se budu ponejvíce držet tohoto standardu, uvedu však některé věci, které do něj nepatří – v takových případech na tuto skutečnost vždy upozorním.

Další důležitou specifikací je "ISO C99"[4], do které patří snad všechno co do "ANSI C", plus některé další funkce. Až na ojedinělé výjimky bude vše, co prezentuji, splňovat tuto normu.

Tyto ojedinělé výjimky budou rozšířením GNU C oproti normě "ISO C99". Jen krátce, GNU je *free-software*[5] a dle mého názoru to nejlepší co můžete používat. Překladač samotný se jmenuje *gcc*, ale to zde nebudu rozvádět, protože to sem nepatří. GNU překladač seženete bez problémů pro všechny UNIX/LINUX OS a pro některé další. Dost.

1.2 Zpracování zdrojového kódu

Ač je tato kapitolka zařazena na začátek, její pročtení doporučuji až poté, co budete znát strukturu programu, pročtenou celou druhou kapitolu, a mít za sebou několik prvních prográmků v C.

Pokusím se zde popsat proces překladu, lépe řečeno postup, kterým se ze zdrojového kódu stává spustitelný program. Budu přitom popisovat fáze tohoto procesu, ne přímo způsob fungování.

Do první fáze překladu vstupují samotné zdrojové soubory (`.c`) a hlavičkové soubory (`.h`). Je vykonávána *preprocesorem*. *Preprocesor* odstraňuje komentáře, rozvíjí makra, řeší podmíněnou kompilaci a skládá texty dohromady. Jinými slovy provádí různé textové náhrady ve zdrojovém kódu. Některé překladače nedovolují přerušit překlad po této fázi.

Druhá fáze je realizována vlastním *kompilátorem*, který provede překlad do *assembleru*. Opět, některé překladače neumožňují přerušení překladu v této fázi.

Třetí fáze je překlad z *assembleru* do jazyka relativních adres, což je již strojový kód, ovšem s tabulkami relativních adres proměnných a funkcí – absolutní adresy některých proměnných a funkcí tak, jak budou ve spustitelném programu, ještě nejsou známy – např. některé funkce mohou být v knihovnách. Neznám překladač, který by neuměl překlad v tuto chvíli přerušit. Výsledný soubor je tvaru `.o` nebo `.obj`, v závislosti na překladači. V takovémto souboru ještě mohou být uloženy nějaké ladicí a profilovací informace, to ale pomineme.

Čtvrtá fáze spočívá v linkování[6] `.o` souborů a statických knihoven[7] do výsledné binárky. O dynamických knihovnách[8] mluvit nebudeme, protože jsou linkovány až "za běhu" programu.

To by bylo k překladu všechno. Velice často použijete ještě *debugger*[9], to je ale samostatný program a se samotným překladem nemá nic společného. Navíc o něm budu mluvit na jiném místě – viz kap. 6.

---

[3] Přesně ANSI C3.159-1989

[4] Přesně ISO/IEC 9899:1999

[5] Můžete jej používat pro překlad svých zdrojových souborů a výsledné programy pak prodávat, aniž byste museli něco platit.

[6] Někdy se využívá samostatný program, tzv. *linker* .

[7] `lib*.a na UNIX/Linux`

[8] *Elf*, *některé* DLL soubory.

[9] Nástroj pro vyhledávání chyb – v překladu do češtiny "odvšivovač".

Kapitola 2

Základní kameny pro programování v jazyce C a programování vůbec

V této kapitole si lehce nastíníme, jak vlastně vypadá program v C, z čeho se skládá a poté přejdeme k základním pojmům. Na konci bychom měli být schopni naprogramovat jednoduchý a funkční prográmek v C.

2.1 Struktura programu a syntaxe zápisu

2.1.1 Členění programu

Jakýkoliv program je posloupností logických kroků. Nutno říci, že pořadí, ve kterém se budou tyto kroky vykonávat, nám není v drtivé většině případů při programování jasné[1]. Jinými slovy, chod programu závisí na vnějších okolnostech. Proto je velice výhodné program rozdělit do menších kousků, u kterých jsme si jisti, že se budou vykonávat vždy nepřerušovaně.

Tímto rozdělením získáme možnost kousky opakovaně volat. Příkladem je třeba kus programu zajišťující vstup čísla z klávesnice a jeho kontrolu. T.j. tento kus programu přečte vstup z klávesnice a pokud to bude opravdu číslo, někam ho uloží a je hotov. Pokud ale vstup nebude číslo (překlep atp.), vypíše uživateli chybovou hlášku “Nezadal(a) jste číslo, opakujte prosím vstup:” a bude číst znovu.

Takovouto smyčku lze velice lehce realizovat cyklem, čili příkazy pro vstup a kontrolu napíšeme jen jednou, přičemž tento kus programu bude probíhat např. tak dlouho, dokud uživatel číslo nezadá.

Další kus může provádět nějaký výpočet atp.

2.1.1.1 Funkce

V jazyce C se kusy kódu nazývají funkce. Funkce je nejdůležitějším elementem programu. Funkce se do sebe mohou zanořovat – v jedné funkci zavoláte pomocí jména druhé funkce tuto druhou funkci a po jejím skončení se bude v provádění programu pokračovat ve funkci první – přesně tam, kde volání druhé funkce skončilo.

[1] Např. nevíme, kolik dat chce uživatel zapsat, kolik dat přečteme, kolikrát budeme vykonávat jaký cyklus.

Samotný program je vlastně funkce. Aby bylo překladači jasné, která funkce má představovat samotný program[2], má hlavní funkce předepsané jméno a v každém programu se musí vyskytovat právě jedna takováto funkce. Její jméno je `main()`[3] .

2.1.1.2 Soubory

Existuje ještě jedno rozdělení programu na menší kousky. Nevyplývá ani tak z logiky programu, jako spíše ze způsobu tvorby programů. Je to rozdělení na soubory.

Pokud píše program více lidí, rozdělí si práci a každý programuje něco jiného. Rozdělí si jej buďto přímo na funkce (v případě malého programu), nebo na nějaké větší díly[4].

Ve druhém případě bývá pravidlem, že se program skládá z více zdrojových souborů[5]. Toto není nijak na škodu, neboť všechny zdrojové soubory vstupují do překladu najednou a překladač si s tím poradí.

2.1.1.3 Hlavičkové soubory[6]

Program se skládá nejen ze zdrojových souborů, ale také ze souborů hlavičkových (`.h`). Ze začátku budete tyto soubory pouze používat, nikoliv tvořit.

Hlavičkový soubor obsahuje jména funkcí (příkazů), hodnoty různých konstant, definice datových typů a makra – vše podrobně rozebereme později. Pokud chcete používat jiné než základní příkazy, a to určitě chcete, protože základních je zatraceně málo, musíte "použít" nějaký hlavičkový soubor.

Zdrojový nebo objektový kód funkcí (příkazů), uvedených v hlavičkových souborech, je uveden v dalších souborech, které musíte mít samozřejmě instalovány, pokud chcete příkazy používat. Nemusíte se ale bát, pokud máte hlavičkový soubor, máte pravděpodobně i ostatní potřebné soubory – a pokud ne, překladač vás upozorní.

Použití hlavičkového souboru:

```
#include <soubor>
```

nebo

```
#include “soubor”
```

Uvedené možnosti se liší ve způsobu vyhledávání souborů.

V prvním případě překladač při překladu prohledá adresáře, které si pamatuje jako systémové a své vnitřní adresáře.

Naopak v případě druhém prohledává nejprve aktuální adresář a až poté ostatní.

Pokud potřebujete zadat soubor nacházející se někde jinde, jednoduše zadejte absolutní cestu – v tomto případě překladač soubor buďto najde a je spokojen nebo nenajde a hledání vzdá.

Slovo "include" znamená česky "zahrň" a je možno jej použít i pro zahrnutí zdrojového souboru (`.c`). Uvádí se na začátku programu a pro každý zahrnovaný soubor zvlášť.

Např:

[2]Která má býti puštěna jako první, přičemž její ukončení má za následek ukončení celého programu.
[3]V překladu do češtiny "hlavní".
[4]Jeden naprogramuje vstupy a výstupy na obrazovku a `WWW`, druhý přístup k databázím atp.
[5]Přípona `.c`
[6]Angl. *header*.

```
#include <stdio.h>
#include “moje.h”
```

Při složitějších programech budete hlavičkové soubory sami vytvářet, ale o tom až v 4.5.

2.1.2 Zápis identifikátoru

Co je identifikátor?

Pojmenování funkce, proměnné, typu, makra[7].

Pokud budeme chtít nadefinovat proměnnou, abychom v ní mohli uchovávat hodnotu, samozřejmě si nebudeme v programu pamatovat a všude psát adresu v paměti[8].Místo toho ji jednoduše pojmenujeme a budeme se na ni odvolávat pomocí jejího jména.

V pojmenování takové proměnné, vlastně jakéhokoliv identifikátoru, nemáme zcela volné ruce. Musíme dodržovat následující tři pravidla (norma “ANSI C”):

1. Identifikátor začíná znakem _ nebo písmenem anglické abecedy.
2. Jako druhý až poslední znak jména identifikátoru můžeme použít znak _, písmena anglické abecedy nebo čísla.
3. V rámci celého projektu musí být identifikátory rozlišitelné v prvních 31 znacích.

Co se týče znaků národní abecedy, ty můžeme používat toliko v komentářích a v řetězcích.

Dále platí ještě jedno pěkné pravidlo – na místo bílého znaku můžeme použít libovolný počet mezer, tabelátorů nebo i znaků konce řádku.

A teď přichází na řadu veledůležitá věc: *C JE CASE SENZITIVE !!!*

Česky: Záleží na tom, který znak je velký a který malý. Jména AHOJ, ahoj, Ahoj jsou tři různé identifikátory!

2.1.3 Blok příkazů

Vezmeme to poněkud netradičně. Řekneme si, k čemu blok příkazů potřebujeme a z toho nám vyplyne, co to vlastně je.

Zatím sice neznáme příkaz podmínky, ale jsme natolik dobří, abychom o něm uvažovali aniž jej oficiálně známe.

Takže: Uživatel odpovídá na otázku typu ano/ne. Samozřejmě, program musí na každou možnost zareagovat jinak – dále budou dvě možnosti pokračování – činnosti rozdělení se odborně říká větvení.

Použijeme příkaz if:

```
if (odpoví ano) then udělejtohle; else udělejtamto;
```

Česky: if=jestliže, then=tak, else=jinak

Problém je v tom, že my chceme v každém z obou případů udělat více než jen vykonat jeden příkaz. Ani volání funkce (což je také příkaz), by nám nepomohlo, neboť funkce může

[7]Pojmy jsou zatím intuitivní, bližší vysvětlení v 2.2 a 4.4.

[8]Už proto, že to nejde. Při psaní programu nemůžeme vědět, kterou adresu dostane proměnná přidělenu.

obsahovat také pouze jeden příkaz[9]. Kolem a kolem, funkce `main()` je také funkce, takže může obsahovat také jen jeden příkaz. Co s tím?

Abychom tedy mohli vůbec něco naprogramovat, potřebujeme nějaký způsob, nějaký nástroj, kterým nacpeme více příkazů tam, kde byl dříve jen jeden. Ano, přesně to poskytuje blok příkazů. Překladač jej bere úplně stejně jako jeden příkaz. Už potřebujeme vědět jen to, jak jej zapsat. Je to dvojice složených závorek `{}`, otevírací závorka značí začátek bloku příkazů a zavírací jeho konec.

Z výše uvedené limitace funkce vyplývá, že tělo funkce začíná otevírací složenou závorkou a končí jejím protějškem.

Například:

```
main()
{
   prvnípříkaz;
   funkce();
   dalšípříkaz;
}
```

Je třeba si uvědomit, že ke každé otevírací závorce musí nutně v kódu programu existovat odpovídající zavírací závorka.

Bloky příkazů se samozřejmě mohou zanořovat. Blok příkazů obsahuje příkazy a těm nic nebrání býti dalšími bloky příkazů, které opět obsahují příkazy a ty zase...

Třeba:

```
<...>
int read_time();
{
   napis("Hay, what's o'clock Tom?");
   save = precti_cas();
   if (save == 0) {
      napis("Make up your mind, man.");
      exit();
   } else {
      napis("Thanks a'lot.");
      go_on();
   }
   return save;
<...>
}
```

2.1.4 Komentáře

Z praxe:

Když napíšeme nějaký program a on náhodou funguje, okamžitě jej zpeněžíme. Jenže uživatel programu si bude po roce vymýšlet úpravy a změny[10].

[9] Což může být sice volání další funkce, ale to jsme si pořád nepomohli!

[10] Což nás samozřejmě z hlediska peněz jen těší, změny obratem naúčtujeme.

Podíváme se tedy na kód našeho programu a zjišťujeme, že se v tom nějak nevyznáme. Je to spousta slov, čísel a nesmyslných znaků bez hlavy a paty, člověk by ani nevěřil že to napsal on sám[11]. Takhle nějak chaoticky by to mohlo dopadnout...

Jenže my víme, že můžeme používat komentáře, což jsou texty volně vložené do zdrojového kódu. Jak vyplývá z názvu, popisují, co která část programu a která funkce dělá. Tyto texty se v jazyce C uzavírají mezi "závorky" `/*` a `*/` a překladačem jsou naprosto ignorovány[12].

Příklad:

```
main()
{
   function_call(); /* tohle je komentář */
   /* a tohle je taky komentář */
   another_function_call();
   /* a tohle
    * je komentář
    * vyrobený
    * hezkým způsobem
    */
}
```

V případě komentářů se není třeba rozpakovat, jen je pište a co nejhojněji. Dávejte si ovšem pozor, aby byly jasně opticky příslušné ke správné části kódu a nezhoršovaly přehlednost programu.

2.2 Proměnná, funkce a výraz

Definice, které zde uvedu, není možno chápat jako formální, absolutní a podobně. Chci vám jimi hlavně jasně sdělit, co co znamená a k čemu se používá – co nejpřesněji popsat daný pojem.

2.2.1 Proměnná

Pojem proměnná je používán velice volně. Souvisí s ním totiž několik dalších a velice blízkých pojmů a slovo proměnná se používá pro všechny. Takže si to trochu upřesněme.

Proměnná je místo v paměti.

Adresa proměnné je ukazatel na toto místo. V C adresy (snad) nikdy nepíšeme do zdrojového kódu.

Jméno proměnné je její zástupce, používáme ho, abychom určili, které místo v paměti myslíme.

Obsah proměnné jsou data na daném místě uložená. Nazýváno také hodnota proměnné. Jistě je vám jasné, že se mohou měnit.

[11] A co teprve když máte upravovat program po někom jiném!

[12] Přesněji řečeno jsou vyhazovány ještě když je překlad někde vpředu daleko za zatáčkou.

Typ proměnné označuje způsob, jakým mají být data interpretována.

Pokud se nad popisem zamyslíte, zjistíte, že pojem proměnná je, tak jak byl definován, úplně k ničemu. Samo místo není nic. Není to ani adresa, ani obsah. A právě proto se pojmem proměnná označuje i jméno proměnné.

Pokud někde narazíte na zápis "do proměnné p si uložíme hodnotu 10", znamená to, že obsah adresy paměti, odpovídající proměnné se jménem p, která má velikost odpovídající typu proměnné se jménem p, nastavíme na hodnotu 10.

Dále snad již není třeba rozepisovat, jistě jste pochopili že se budu držet prvního způsobu zápisu.

Mám dojem, že účel proměnných je evidentní. Jakmile si budete nuceni zapamatovat jakoukoliv hodnotu, ať již to budou přečtená data či snad výsledek nějaké operace, použijete pro to proměnnou.

Poznámka. Ještě poznámka ke slovu "snad", použitého u definice pojmu adresa proměnné. Není tam proto, že bych si nebyl jistý, nýbrž proto, že doufám, že tak činit nebudete.

Možná vás nenapadá příklad, proč byste vůbec chtěli něco takového dělat. Máte pravdu, že adresy vlastních proměnných nemůžete znát[13]. Mohli byste ji však chtít napsat třeba pro přístup na určitou adresu paměti, kde je neustále uložena nějaká hodnota.

Typickým příkladem je přístup k nějakému HW zařízení, například k notoricky známému BIOSu, nebo třeba na grafickou kartu. Zápis takové adresy je ale v každém případě špatnou věcí, protože program se pak stává ještě více závislý než je[14]. Místo toho použijte předdefinovanou konstantu[15]. Pokud taková konstanta nebude existovat, budiž, zapište přímo adresu, ale ve formě makra vyvedeného na začátek programu (viz 4.4).

Jméno proměnné je identifikátor a platí pro něj stejná pravidla zápisu jako pro všechny ostatní identifikátory (viz 2.1.2).

Typ proměnné

Podstatnou věcí je typ proměnné. Co si máme představit pod rozdílnou interpretací proměnné?

Představme si, že z určitého místa v paměti budeme číst jeden *byte*. Přečteme číslo 32. Pokud to budeme interpretovat jako celé číslo bez znaménka, dostaneme číslo 32. Pokud to ale budeme interpretovat jako znak, podle ASCII tabulky dostáváme znak mezera.

To je ale pouze jedna strana problému. Druhá strana spočívá ve velikosti proměnné. Jak má překladač odhadnout, kolik místa potřebujeme – jeden *byte* pojme pouze 256 kombinací, co když ale chceme uložit číslo větší? Tím problémy nekončí, protože nám zbývají reálná čísla, řetězce, složené typy a další a další.

Proto existuje typ proměnné, kterým specifikujeme, jak mají být data interpretována a zároveň řekneme, kolik místa má být pro tato data vyhrazeno.

Navíc si tím v mnohém zjednodušujeme práci. Máme například typ `float`, díky kterému můžeme bez jakýchkoliv našich vlastních procedur uchovávat zvlášť desetinnou a celou část

[13] Můžete si je zjistit a uložit do jiné proměnné a přes tu pak například adresu první proměnné vytisknout, ale mluvím o zápisu adresy v paměti stroje přímo do zdrojového kódu.

[14] Už ne pouze na výskytu daného typu zařízení v systému, ale přímo na místě, kam je paměť toho zařízení namapována.

[15] Myslím třeba z ovladače pro toto zařízení – pokud přenesete program jinam, zařízení bude namapováno jinde, ale ovladač bude mít stejnou konstantu se stejným jménem a program bude fungovat.

čísla[16]. Existuje spousta dalších typů, navíc si můžeme definovat vlastní, ale o tom později – v 5.2.1.

2.2.2 Funkce

Už víme, proč funkce existují, nyní se na ně podíváme zblízka.

Funkci můžeme definovat mnoha způsoby. Nejlepší by asi bylo vypůjčit si definici z nějakého funkcionálního jazyka, ale tohle je jazyk C a já jsem tvrdohlavý a svéhlavý. Uděláme to takhle: každá z následujících větiček částečně definuje funkci a všechny dohromady[17] dávají dost přesně obraz, který potřebujeme:

- Funkce je kus kódu (= zdrojového textu) programu.
- Funkce sama se dá rozdělit na dvě části – hlavičku a tělo.
- Tělem funkce je blok příkazů.
- Funkce je za běhu programu uložena na nějaké adrese v paměti a může být volána.
- Syntaxe volání (ze zdrojového kódu) se řídí podle hlavičky funkce – funkci mohou být předávány data, která mohou ovlivňovat její chování. Tato data jsou nazývána parametry funkce.
- Funkce má vždy návratovou hodnotu určitého typu[18], která je k dispozici volajícímu.
- Voláním funkce se funkci "předá řízení" a vykonávání kódu volajícího je pozastaveno až do doby ukončení volané funkce.
- Uvnitř těla funkce je možno provádět cokoliv, co v hlavním programu - třeba volat jinou funkci, nebo klidně sama sebe (*rekurze*).

Nyní je třeba přiblížit hlavičku a návratovou hodnotu.

Syntaxe hlavičky vypadá takto:

```
[paměťová_třída] typ jméno ([parametry])
```

kde:

paměťová_třída je nepovinná, udává viditelnost funkce, viz později.

typ specifikuje typ návratové hodnoty. Může být cokoliv kromě pole a jiné funkce[19].

jméno udává jméno funkce. Je to identifikátor – platí pro něj pravidla pro zápis identifikátorů – viz 2.1.2.

parametry výčet parametrů ve formátu typ jméno, typ jméno, ...

[16] Tím chci říci, že můžeme bez problémů uchovávat reálné číslo.
[17] Tedy průnik jimi vymezených množin.
[18] Pokud nevrací typ `void`, či `void *`
[19] Může to být ovšem ukazatel, tedy bez problémů ukazatel na pole či funkci.

Je třeba si všimnout, že i funkce bez parametrů musí mít v hlavičce závorky – a také se s nimi musí volat!

Příklad hlavičky funkce `main()`:

```
int main(int argc, char **argv);
```

Pojďme se podívat na návratovou hodnotu. K čemu se tato hodnota používá? Většinou buď k předání výsledku funkce, nebo k předání informace o úspěšnosti funkce – například 0 pokud funkce neuspěla, pokud uspěla tak jakoukoliv jinou hodnotu.

Ještě nás zajímá jakým způsobem ukončit funkci a předat návratovou hodnotu. Dosáhneme toho pomocí příkazu `return`, který může mít parametr. Parametrem je právě návratová hodnota, typem samozřejmě odpovídá typu uvedenému v hlavičce funkce.

Funkce v C mají ještě jeden zajímavý rys – nemohou měnit hodnotu svých parametrů. Proto, pokud chceme, aby funkce měla k hodnotám přístup, zavoláme ji nikoliv se jmény proměnných, ale s adresami na proměnné[20]. Tedy parametry jsou nyní adresy, a ty měnit nebudeme. Změníme až obsah paměti na daných adresách.

Na závěr této rádoby definice se podívejme na příklady dvou funkcí, z nichž druhá je napsána s chybou[21], ale o tom později.

Zmíněné příklady:

```
float cube (float a)
/* funkce pocita obsah krychle
 * parametrem funkce je velikost strany krychle
 */
{
   return a*a*a;
}


 int input_ok()
/* funkce ceka na vstup z klavesnice
 * jakmile se vstup objevi, skonci s jednickou
 * pokud selze funkce getchar(), input_ok()
 * vrati nulu
 */
{
   if (getchar()) return 1;
   else return 0;
}
```

2.2.3 Co je to výraz

Velice jednoduchá odpověď. Výraz se skládá z operátorů a operandů. Za operátory si zatím představte aritmetické operace, podrobně je rozebereme později. Co může být operandem:

[20] Přesněji řečeno s jinými proměnnými, ve kterých jsou uloženy adresy na proměnné, které chceme měnit.
[21] Ne že by nefungovala, ale dělá něco jiného než bychom čekali, resp. než co by měla.

1. Proměnná, makro, volání funkce (identifikátor)
2. Číslo, znak, řetězec (konstanta)

Stačí zapřemýšlet a dojdeme k tomu, že definice je mírně řečeno ubohá, neboť připouští např. výraz `+*++a//c“papá”tre()-*`, což za korektní výraz asi považovat nebudeme. Prostě řečeno, nemáme definici v matematickém slova smyslu, ale nástin výrazu a popis jeho částí, ze kterých tvor o průměrné a vyšší inteligenci pochopí, co je výraz.

Pro jistotu pár příkladů:

```
(4 + mocnina(18)) / 2
MAX * (MIN - 1) * konvertuj_na_cislo(“123”)
```

2.3 Operátory a přiřazení

Tato kapitola obsahuje pouze výčet operátorů, jejich popis a na závěr ujasnění problému přiřazení.

2.3.1 Unární

Unární operátory se vztahují pouze k jednomu argumentu.

Klasické unární operátory

Jsou operátory `+` a `-`.

Unární operátor `+` snad nemá praktického využití, zato operátorem `-` určujeme, že číslo je záporné. Tyto operátory píšeme před operand.

Speciální unární operátory

Tím myslíme operátory `++` a `--`.

Vysvětlíme si na příkladu:

`++a` znamená inkrementaci proměnné (zvětšení o 1) *a* ještě před vyhodnocováním výrazu

`a++` znamená inkrementaci proměnné (zvětšení o 1) *a*, ovšem až po vyhodnocení výrazu

Operátor se dá použít pouze na tzv. *l-hodnotu*, tedy na objekt, který má přidělenu nějakou adresu v paměti a je ji možno měnit.

Rozdíl si opět vysvětlíme na příkladu:

položme `a=5`, pak

```
b=++a*5
```

dá `a=6 b=30` zatímco

```
b=a++*5
```

dá `a=6 b=25`

Pro operátor `--` platí totéž, jen s tím, že místo inkrementace je prováděna dekrementace.

2.3.2 Binární

Binární operátory se samozřejmě vztahují ke dvěma operandům. Binárních operátorů je v C nejvíce.

Aritmetické

Aritmetickými operátory se provádějí běžné matematické operace.

Operátory `+`,`-` a `*` fungují přesně jak bychom očekávali, ovšem operátor dělení `/` má malou zvláštnůstku. Nedá se totiž říci, zda představuje reálné nebo celočíselné dělení. Ve skutečnosti zastupuje obojí a operace, která se opravdu provede, závisí na typu operandů.

`15 / 2` ... celočíselné (*div*)

`15 / 2.0` ... reálné (druhé číslo není celé)

Dalším operátorem je `%` představující zbytek po celočíselném dělení.

`15 % 5 = 0`

Relační

Relačními operátory testujeme vztah mezi dvěma operandy. Výsledkem testu je pravda či nepravda.

Operátor `==` je testem na rovnost, operátory `<`, `>`, `<=` a `>=` se chovají podle očekávání. Operátorem nerovnosti je `!=`.

Můžeme testovat dokonce i vztah dvou ukazatelů, ale protože ještě nevíme, co to je, odložíme tento problém na později.

2.3.3 Logické

Máme k dispozici také několik operátorů matematické logiky a to logický součin (konjunkci), logický součet (disjunkci) a negaci. Logické operátory můžeme aplikovat na hodnoty pravda nebo nepravda.

Ač by se samozřejmě daly rozdělit mezi unární a binární, uvádím je zvlášť.

Zápis operátorů:

`&&` konjunkce (AND)

Například

```
if ((podmínka1) && (podmínka2)) ...
```

Je pravda tehdy a jen tehdy, když jsou obě podmínky vyhodnoceny jako pravdivé.

`||` disjunkce (OR)

```
if ((podmínka1) || (podmínka2)) ...
```

Je pravda pokud je alespoň jedna z podmínek pravdivá. Nepravdivá je pouze v případě, že jsou nepravdivé obě podmínky.

`!` negace (NOT)

```
if (! (podmínka1)) ...
```

Výsledkem je opak podmínky. Tedy, pokud je podmínka pravdivá, výsledkem negace je nepravda a pokud je podmínka nepravdivá, výsledkem negace je pravda.

2.3.4 Speciální

Dalšími zajímavými operátory jsou bitové operátory a operátor čárky. Po absolvování této kapitolky za sebou budeme mít téměř všechny operátory v C.

Jedinými výjimkami budou operátory týkající se ukazatelů, polí a struktur, stejně jako operátor otazníku. Nepřijdete o ně, jen si je nechám na později.

Bitové operátory

Jakékoliv celé číslo je reprezentováno několika *byte*. Každý *byte* má osm bitů a jak jistě víme, bit je buďto 0 nebo 1. Číslo si tedy můžeme představit jako řadu nul a jedniček. Nad takovýmto číslem si ukážeme bitové operace.

Posun bitů doleva – operátor << Syntaxe: `x << n`

Posune "x" o "n" bitů doleva. Bity, které přetekly, se ztratí. Nové bity, které přišly zprava, jsou nuly.

Význam: násobení n-tou mocninou 2

Příklad:

```
5 << 3
```

dá `5*2exp3` tedy `5*8` tedy `40`

Poznámka: Bitový posun je mnohem rychlejší než obyčejné násobení.

Posun bitů doprava – operátor >> Syntaxe: `x >> n`

Posune "x" o "n" bitů doprava. Vypadlé bity se ztrácejí, zleva přibývají nuly.

Význam: Dělení mocninou dvou.

Bitový součin – operátor & Syntaxe: `x & y`

Význam: Nastavení určitých bitů na 0.

Výsledkem bitového součinu dvou čísel je číslo, které má v bitové masce jedničky tam, kde je mělo jak x, tak y.

Příklad:

```
x=10101010 (bin); y=11110000 (bin);
x & y= 10100000 (bin);
```

Bitový součet – operátor | Syntaxe: `x | y`

Výsledné číslo má jedničky na místě, kde je má alespoň jedno z čísel.

Význam: Nastavení určitých bitů na 1.

Příklad:

```
x=10101010 (bin); y=00001111 (bin);
x | y=10101111 (bin);
```

Exkluzivní bitový součet – operátor ^ Syntaxe: `x ^ y`

Výsledné číslo má jedničky tam, kde je mělo právě jedno z čísel.

Bitová negace – operátor ~ Syntaxe: `~x`

Výsledné číslo je negací bit po bitu, tj. má jedničky tam, kde mělo původní nuly a naopak.

Operátor čárky

Operátor čárky je velice zvláštním a také velice zřídka používaným operátorem[22]. Navíc je třeba podotknout, že ne všude, kde na čárku v nějakém kódu narazíte, ji můžete za tento operátor pokládat. Například v definicích proměnných čárka ztrácí svůj význam.

Příklad:

```
int i=5, a=10;
```

(čárka zde není operátorem)

Patří mezi binární operátory a zapisuje se:

```
výraz , výraz2
```

Uvedený výtvor považujeme opět za výraz. Výsledná hodnota celého výrazu je ale rovna výsledné hodnotě druhého výrazu a výsledek výrazu prvního je zapomenut. Což ovšem neznamená, že by nebyl vyhodnocen.

Typický příklad využití operátoru čárky je příkaz `for`:

```
for (b=50; b<a; a--, b++)
```

(viz 2.6.3)

Poznámka: Ač je operátor čárky šikovná věc, obecně znesnadňuje čitelnost programu (většině jedinců).

2.3.5 Priorita

Popis priority

Priorita operátorů určuje, jak se bude daný výraz vyhodnocovat. Např. dělení a násobení musí mít vyšší prioritu než sčítání a odečítání, neboť `2+6*8` je `50`, nikoliv `64`. Operátorů máme mnohem více a protože se mohou v jednom výrazu vyskytovat snad všechny, musí mít prioritu určenu i mezi sebou navzájem.

Samozřejmě nám nebude vždy priorita vyhovovat. Například pokud chceme opravdu spočítat v prvé řadě `2+6` a teprve potom násobit `8`, nebudeme psát dva řádky programu, nýbrž prioritu upravíme pomocí závorek: `(2+6)*8`.

Tento postup funguje na všechny operátory, závorky mohou být samozřejmě zanořené. Obecně platí: Pokud nevíte, jak se bude daný výraz vyhodnocovat, závorkujte! Na druhou stranu, není moc pěkné závorkovat všechno a proto nám neuškodí alespoň základní povědomí co má jakou prioritu.

Operátory se stejnou prioritou se mohou provádět buďto zleva doprava nebo zprava doleva.

Např:

`5*9/8*4` se provádí jako `((5*9)/8)*4` (zleva doprava)

[22] Dosti ale záleží na programátorovi – jsou tací, kteří ho využívají docela často.

`~!~!1` se provede jako `~(!(~(!1)))` (zprava doleva)

Obecně se dá říci, že pokud se operátor píše před operand, provádí se zprava doleva, v ostatních případech – všechny binární operátory – se provádí zleva doprava.

Přiřazení se ovšem provádí zprava doleva – a také operátor otazník, to ale viz dále.

Vlastní priorita

Nejvyšší prioritu mají

`()` ... volání funkce

`[] . ->` ... pole, struktura a ukaz. na strukturu (viz 3.3.2, 3.3.3 a 3.6.3)

Nižší priorita

`! ~` ... negace a bitová negace

`+ -` ... unární plus a mínus

`++ --` ... inkrementace a dekrementace

`& * ...` získání adresy a dereference ukazatele (viz 3.5.1)

`(typ)` ... přetypování (viz 3.4)

`sizeof(objekt)` ... zjištění velikosti objektu (viz 3.5.3)

Nižší priorita

`* / %` ... násobení, dělení a zbytek

Nižší priorita

`+ -` ... sčítání a odečítání

Nižší priorita

`<< >>` ... bitový posun vlevo a vpravo

Nižší priorita

`< <= > >=` ... operátory menší, menší nebo rovno atd.

`== !=` ... test na rovnost a nerovnost

Následují operátory s neustále se snižující prioritou.

`&` ... bitový součin

`^` ... bitový exkluzivní logický součet

`|` ... bitový logický součet

`&&` ... logický součin

`||` ... logický součet

`?:` ... operátor otazníku (viz 3.1.4)

`Přiřazení` (viz 2.3.6)

`,` ... operátor čárky

2.3.6 Přiřazení

Přiřazení samozřejmě znamená zápis nějaké hodnoty někam do paměti. To, co zapisujeme, nemusí být jen konstanta, můžeme zapsat výsledek výrazu – pokud má tento výsledek stejný typ jako proměnná do které zapisujeme, nebo se automaticky konvertuje (viz 3.4).

Nejprve si vysvětlíme rozdíl mezi přiřazovacím výrazem a příkazem.

Přiřazovací *výraz* se skládá z *l-hodnoty* (měnitelné adresy v paměti), rovnítka a *r-hodnoty* (výrazu)

Přiřazovací *příkaz* je přiřazovací výraz se středníkem na konci. Teprve on uloží výsledek výrazu do proměnné.

Jednodušeji řečeno, teprve přidáním středníku se z výrazu stává příkaz.

Rozšířené přiřazení

Přiřazení je snad nejčastěji prováděná činnost v imperativním programování a proto byla v C zavedena konvence umožňující zkrácení zápisu přiřazení, a to:

```
proměnná operátor= výraz;
```

což znamená totéž jako

```
proměnná = proměnná operátor výraz
```

Takto zkrátit se dá zápis při použití operátorů `+ - * / & << >> & ^ |`.

Například:

```
a += 4*b;
```

je totéž co

```
a = a + 4 * b;
```

2.4 Základní datové typy

Jak už jsme si uvedli výše, data uložená v paměti mají nějaký typ. Lépe řečeno, náš program je čte a ukládá určitým způsobem a to nazýváme typ. V této kapitole si uvedeme nejdůležitější předpřipravené datové typy.

2.4.1 Celá čísla

Jazyk C zná několik typů celých čísel:

```
int, unsigned int, short int, unsigned short int,
long int, unsigned long int.
```

Typy začínající slovem *unsigned* jsou bez znaménka a má li typ `int` rozsah `-32768...+32767`, tak `unsigned int` bude mít rozsah `0...65535`.

Dále platí, že typ bez znaménka zabere stejně paměti jako jeho znaménkový ekvivalent.

Ještě platí následující nerovnost ve velikosti alokované paměti:

```
short int <= int <= long int
```

Určitě vás trochu zmate, že neexistuje žádná norma udávající přesně velikost těchto typů. Většina překladačů na PC ale bude mít `int` o velikosti 2 *byte*, tedy s výše uvedeným rozsahem, typ `long int` by mohl mít čtyři byte a typ `short`? Pravděpodobně bude mít stejnou velikost jako typ `int`.

Cvičení: Zjistěte velikost typů, které používá váš překladač (použijte operátor `sizeof()` – viz 3.5.3)

Existuje jistá benevolence zápisu výše uvedených typů – pokud má typ více než jedno slovo, můžeme slůvko "int" vypustit. Tedy, místo `unsigned int` stačí napsat `unsigned`.

Cvičení již jistě máte za sebou, takže teď jednodušší cesta jak zjistit limity typů: hlavičkový soubor `limits.h`. (jako další cvičení jej najděte)

2.4.2 Reálná čísla

Typů reálných čísel zná jazyk C tři, a to:

```
float, double, long double
```

Maxima, minima a přesnost v desetinných místech najdeme v hlavičkovém souboru `float.h`.

Když se tam podíváme, nebo prostě jen zkusíme něco v těchto číslech spočítat, zjistíme, že jsou zapisována v podivném formátu. Podivný je pouze jen na první pohled. Trošku si jej přiblížíme.

Číslo typu `float` (nebo `double` nebo `long double`) se skládá ze dvou částí a to z *mantisy* a *exponentu*. *Mantisa* je normální desetinné číslo, převážně ve tvaru `a.b` kde `a` se pohybuje mezi `0` až `9` a `b` je desetinnou částí.

Exponent je od *mantisy* oddělen písmenem "e", ať již malým nebo velkým. Může být kladný nebo záporný a představuje exponent při základu deset, neboli o kolik míst máme posunout desetinnou čárku, abychom dostali požadované číslo tak, jak jsme zvyklí. Znaménko pak určuje směr, kterým budeme čárku posouvat. Jednoduše řečeno, *exponent* `3` znamená násobení tisíci (posunutí desetinné čárky o tři doprava) a *exponent* `-5` dělení `100000`.

2.4.3 Logické hodnoty

V matematické logice se pracuje pouze se dvěma logickými hodnotami a to je pravda a nepravda. Samozřejmě i jazyk C zná v konečném důsledku pouze tyto dvě logické hodnoty, jsou ale representovány typem `int`, kde 0 znamená nepravdu (*false*) a cokoliv jiného pravdu (*true*).

2.4.4 Znaky

Znaky jsou representovány typem `char`. Při používání typu `char` si nemůžete být jistí, zda se používá typ `unsigned char` nebo `signed char` (rozdíl v rozsahu). Ne že by to překladač udělal jednou tak a podruhé onak, stejný překladač na stejné platformě to dělá pořád stejně, ale není žádná norma, která by jednu z těchto variant předepisovala.

Pokud si chcete být jisti, že váš program bude používat např. kladný rozsah, stačí psát `signed char`.

Proč to zde tak rozvádím? Spíše pro úplnost, neboť typ `char` , ať již jakýkoliv, stejně moc používat nebudete. Proč?

Pojďme na to od lesa. Makro EOF[23] má hodnotu -1 a při konvertování na `char` není jisté, jaké hodnoty by nabylo[24] a proto se při čtení znaků používá typ `int` místo typu `char`. Také standardní vstupně/výstupní funkce znaku (viz 2.7.2) vracejí hodnotu `int`.

Při výstupu budeme používat formátovaný výstup a v něj je překonvertování `int` na znak velice jednoduché – viz 2.7.3.

2.4.5 Řetězce

Řetězce jsou ve své podstatě pole znaků a tak vlastně v této kapitole nemají co dělat. Protože to ale je pro téměř všechny programy přímo esenciální typ, dovíme se o něm něco již tady.

Řetězce v C jsou ukončeny znakem `\0` (viz 2.5.2), proto v paměti zabírají o jeden element více, než je jejich délka.

Je třeba říci, že při čtení řetězce se čte až dokud se nenarazí na znak `\0`, pokud tedy nějakým omylem znak `\0` v paměti přepíšeme a potom použijeme funkci výstupu řetězce na disk...

Jednoduše řečeno, je třeba dávat si pozor, co děláme[25]. Na druhou stranu tento přístup přináší obrovskou výhodu – velikost řetězce nemá limit, leda velikost operační paměti.

2.5 Zápis hodnot, konstanta

Nyní se naučíme správně v C zapisovat čísla a tvořit konstatnty. Znát tyto pravidla je nutné, neboť jinak by vznikalo mnoho a mnoho zbytečných chyb.

2.5.1 Čísla

Celá čísla

Zapisujeme tak jak jsme zvyklí, jen nesmíme vkládat žádné mezery (ne `1 000`, ale `1000`).

Oktalové číslo (osmičková soustava má číslice 0 až 7) označíme nulou na začátku[26] a číslo šestnáctkové označíme prefixem `0x` nebo `0X`[27].

Příklady:

[23] Konec souboru, *End-Of-File.*

[24] Při `unsigned char` by se -1 překonvertovala na 255.

[25] Vzato kolem a kolem, nezanedbatelný počet bezpečnostních děr v mnoha programech pramení právě z neopatrné manipulace s řetězci.

[26] Z čehož vyplývá, že desítkové číslo nesmí začínat nulou.

[27] Velikost písma může být různá i u a...f.

`0xfa54, 066, 0x41A0, -800`

Reálná čísla

Zapisuje ve tvaru *mantisaEexponent*, kde *exponent* je oddělen znakem "e".

Příklady:

`45.2698, 12.5e4, 1.77E-15, 6., .86`

2.5.2 Znaky

Pokud chceme zapsat znak, uzavřeme jej do apostrofů,

např:

`'a', '#', ' ', '5'`

Pokud chceme zapsat nějaký z řídících znaků (např. znak nového řádku), můžeme do apostrofů zapsat jeho oktalovou hodnotu v ASCII tabulce následujícím způsobem:

`'\012'`[28]

Tedy ve tvaru `\xxx` kde xxx jsou oktalové číslice.

Existuje další možnost zápisu a to jako `\xaa` nebo `\Xaa` kde x resp. X je opravdu znak x, aa je číslo v šestnáctkové soustavě, opět hodnota z ASCII tabulky.

Tedy `'\012'` je totéž jako `'\x0A'` (nebo `'\x0a'` ...).

U některých znaků máme k dispozici zápis pomocí speciálních značek, které se opět zapisují do apostrofů.

Tab. ASCII - viz `man ascii`

Značka	Kód v ASCII (dec)	Popis
\0	0	nulový znak (NUL)
\a	7	pípnutí (BEL)
\b	8	backspace (BS)
\t	9	horizontální tab. (HT)
\n	10	nová řádka (LF)
\v	11	vertikální tab. (VT)
\f	12	konec stránky (FF)
\r	13	návrat na začátek řádky (CR)
\\	92	backslash

2.5.3 Řetězce

Zápis řetězců není nijak složitý – řetězec se prostě uzavře do uvozovek.

Uvnitř řetězce můžeme zapisovat speciální znaky podle předcházející tabulky. Záleží na překladači, ale na platformě MSDOS nebo MSW by měl znak `\n` uvnitř řetězce být interpretován jako `\r\n`, neboť pokud píšeme `\n`, počítáme s odřádkováním.

Pokud potřebujeme zapsat řetězec delší, než nám dovoluje řádka, jednoduše jej rozdělíme na dva řetězce, oddělené od sebe pouze bílými znaky. Překladač tyto dva řetězce spojí.

[28] Pozor, v MSDOS a MSW je konec řádku CR,LF nikoliv pouze LF – použijeme značení \n !

2.5.4 Zápis konstanty

Pojem konstanta si ujasníme později – viz 4.3, zatím pouze zmíníme několik modifikátorů, které při zápisu čísel použijeme k určení typu konstanty. Například při zápisu celých čísel můžeme za číslem uvést písmena l,L (*long*), u,U (*unsigned*), nebo jejich kombinace.

Pokud žádné takové písmeno nepoužijeme, implicitně je konstanta typu `int` nebo `long int` (pokud se do `int` nevejde).

Pro reálné konstanty je implicitní typ `double`, písmeny f,F určíme typ `float` a l nebo L typ `long double`.

2.6 Příkazy podmínky a cyklu

V této kapitolce se seznámíme se základními prostředky jazyka C, umožňujícími větvení a cyklení programu.

Jaký přínos nám plyne z možnosti větvení a cyklení?

Větvení nám umožňuje rozlišovat mezi několika případy a podle toho se zachovat. Proto se odborně říká, že se program větví, což má vyjádřit, že existuje více rozdílných kroků závislých na situaci. Typickým případem použití větvení je při interakci s uživatelem – na základě uživatelovy volby provedeme tu kterou akci.

Naopak cyklení nám umožnuje provádět kus programu dokola a dokola, dokud není splněna nějaká podmínka (nebo naopak dokud nějaká podmínka platí). Jednoduchý příklad – opět při interakci s uživatelem:

Potřebujeme od uživatele dostat odpověď typu ano/ne, takže se ho musíme ptát neustále dokola, dokud jednu z možných odpovědí nenapíše. Je třeba ignorovat všechny ostatní odpovědi a tak cyklíme, dokud nedostaneme jednu z použitelných variant.

2.6.1 `if`

Je základním příkazem větvení. Českým ekvivalentem "if" je slovo "jestliže".

Příkaz `if` funguje následujícím způsobem:

```
if (podmínka) příkaz;
```

Podmínka se vyhodnotí a je-li jejím výsledkem pravda, provede se příkaz. Je-li výsledkem podmínky nepravda, ve vykonávání programu se pokračuje za koncem těla příkazu `if`.

Pokud příkaz nahradíme blokem příkazů, za zavírací závorkou již nepíšeme středník.

To, co jsme si uvedli, je ve skutečnosti neúplná forma příkazu `if`. Samozřejmě ji můžeme používat, v mnoha případach ale vhodně využijeme plnou formu příkazu `if`:

```
if (podmínka) příkaz1; else příkaz2;
```

Slovo "else" znamená česky "jinak". Schéma plného `if` je intuitivní. Pokud je podmínka splněna, provede se příkaz1, jinak (podmínka splněna není) se provede příkaz2.

Po skončení příkazu se opět pokračuje za koncem těla příkazu `if`.

Zkusme trochu popřemýšlet. `if (podmínka) příkaz;` hmm... a co kdyby byl příkazem zase `if`?

To je samozřejmě povoleno a je to velice jednoduchý způsob jak udělat složitější větvení programu. Jen je třeba pamatovat, jak se při složitějším větvení chová příkaz `else`. Například zápis

```
if (po1) if (po2) přík1; else přík2;
```

By mohl být poněkud matoucí. Stačí si ale zapamatovat pravidlo, že `else` se vztahuje k nejbližšímu předchozímu `if`, tedy tomu nejvíce zanořenému. Pokud potřebujeme, aby se `else` vztahovalo k vyššímu `if`, musíme použít složené závorky.

Cvičení: Vhodně uzávorkujte výše uvedený příklad tak, aby se `else` vztahovalo k prvnímu `if`.

Cvičení2: Vytvořte program, který se uživatele zeptá na několik otázek. Použijte alespoň dvojnásobné zanoření příkazů `if`.

2.6.2 switch

Příkaz `switch` je dalším příkazem větvení a to větvení vícenásobného. Pokud se podíváme do slovníku, mezi mnoha významy najdeme například vyhýbka, přepínač, ale taky přepnout či přehodit vyhýbku.

Tento příkaz se používá zvláště v případech, kdy se chceme rozhodnout mezi více variantami. Jeho syntaxe je následující:

```
switch (výraz) {
    case (případ): příkaz; ... příkaz; [break;]
    ...
    case (případ): příkaz; ... příkaz; [break;]
    [default: příkaz; ... příkaz; [break;]]
}
```

Výrazy uvedené v hranatých závorkách jsou nepovinné – nemusíme je vůbec uvést.

`(výraz)` je výraz, který se vyhodnotí a bude se porovnávat vůči jednotlivým případům uvedeným za klíčovým slovem “case”.

Příkazy uvedené za dvoutečkou nemusí být uzavřeny v bloku příkazů.

Větev *default* se provede právě když se nevyskytuje ani jeden vyhovující případ v case-větvích.

Klíčové slovo `break` znamená, že po ukončení provádění této větve se má ukončit i příkaz `switch`.

Provádění větve normálně pokračuje tak dlouho, dokud se nenarazí na `break` nebo na konec `switch`. Pokud tedy za nějakou větví slovo `break` neuvedeme, provedou se i příkazy uvedené v jiných větvích!.

Jak vypadá `(případ)`? Je to jediná konstanta, na jejím místě se nemůže vyskytnout výčet. Pokud chceme udělat větev pro více případů, použijeme následující fintu:

```
switch (a) {
   case 1:
   case 2:
   case 3: funny_funkce(“pripad 1 nebo 2 nebo 3”); break;
```

```
    case 4:
    case 5: funkce2(); funkce3();
            funny_funkce("pripad 4 nebo 5");
            break;
    case 10: funny_funkce("pripad 10"); break;
    default : funny_funkce("Jakykoliv jiny pripad. (:-) ");
}
```

Tedy, za dvoutečkou ve větvi `case` nemusí být žádný příkaz. A protože nenásleduje slovo `break`, pokračuje se ve vykonávání příkazu `switch`.

`(případ)` může být samozřejmě i reálnou, znakovou, nebo řetězcovou konstantou.

Znakové konstanty jsou automaticky konvertovány na `int`.

2.6.3 `for`

Příkaz `for` je prvním z příkazů cyklu. Je z příkazů cyklu nejtěžší a pokud chcete nejprve něco lehčího, podívejte se napřed na "while" a "do ... while" – viz 2.6.4 a 2.6.5. Syntax `for` je následující:

```
for (inicializace; podmínka; iterace) příkaz;
```

Příkazem může samozřejmě být i blok příkazů.

Část *iterace* říká, jaký příkaz se má provádět při ukončení každé iterace cyklu. Typicky je to zvedání proměnné, přes kterou určujeme počet iterací cyklu a kterou testujeme v podmínce. Řekněme, že chceme udělat jednoduchý cyklus přes proměnnou a, potom jako iteraci pravděpodobně použijeme `a++`.

Část *podmínka* je podmínkou, která určuje, kdy nastane další iterace cyklu, resp. kdy se skončí provádění cyklu. Když je podmínka splněna, nastává další průchod cyklem. Pokud splněna není, provádění cyklu se ukončí. Řekněme, že chceme cyklus ukončit až bude proměnná a větší než 10, a proto napíšeme `a<11` jako podmínku.

Část *inicializace* určuje co se provede při startu cyklu – typicky zde umístíme inicializaci proměnné přes kterou iterujeme – například `a=0`.

Cyklus `for` je velice dobře vymyšlen, co se sofistikovanosti týče. Lze s ním udělat spoustu věcí, ale složitější kejkle si rozebereme až později, prozatím dáme dohromady jednoduchý příklad, který jsem zmiňoval postupně již při popisu příkazu `for`, tedy:

```
for (a=0; a<11; a++) printf("%d\n",a);
```

který nám na obrazovku vytiskne čísla 0 až 10, každé na zvláštním řádku.

Obecně se říká, že příkaz `for` se používá převážně když chceme provést předem daný počet iterací cyklu. Já s tímto tvrzením (co se týče jazyka C) nesouhlasím, neboť kromě nejtriviálnějších cyklů, které jsou ideálně zvládnutelné pomocí `while`, se snad vždy používá cyklus `for`, protože s ním lze krásně logicky namodelovat na první pohled dosti složité věci.

2.6.4 `while ...`

Příkaz `while` je druhým příkazem cyklu. Je podstatně jednodušším než příkaz `for`, což lze vypozorovat už z jeho syntaxe:

```
while (podmínka) příkaz;
```

Přeloženo do češtiny znamená slůvko “while” “dokud”. O `while` se dá uvažovat následujícím způsobem: Dokud platí podmínka, pováděj příkaz. Z tohoto pohledu vyplývá několik věcí:

- Podmínka je splněna – vykoná se jedna iterace cyklu.
- Podmínka je nesplněna – tělo cyklu se neprovede a program pokračuje za koncem těla cyklu.

Podmínka se vyhodnocuje na začátku cyklu a tedy se může stát, že cyklus neproběhne ani jednou.

Jednoduchý příklad:

```
while (i!=150) i=getchar();
```

Který načítá znaky z klávesnice tak dlouho, dokud nepřečte znak s ASCII kódem 150.

Příkaz `while` je dosti jednoduchý na pochopení. Proto pokud máte problémy s chápáním cyklu `for`, můžeme si jej jednuduše přepsat na `while`, a to následujícím způsobem:

For:

```
for (inicializace; podmínka; iterace) příkaz;
```

Přepis:

```
inicializace;
while (podmínka) { příkaz; iterace; }
```

2.6.5 `do ... while`

Je třetím příkazem cyklu. Jeho syntax vypadá takto:

```
do příkaz;
while (podmínka);
```

Platí podobná pravidla jako u cyklu `while`, t.j. cyklus končí, když je podmínka nesplněna. Můžeme si ale všimnout, že podmínka je testována až za tělem příkazu. Pokud si opět význam přeložíme do češtiny, dostaneme:

“dělej příkaz dokud platí podmínka”

Je třeba si uvědomit, že příkaz se provádí před testováním podmínky, tedy tělo cyklu `do ... while` se vždy provede alespoň jednou.

2.7 Vstupy a výstupy I – terminál

Nyní si povíme něco o základních vstupech a výstupech programu, a to pro (převážně) komunikaci s uživatelem. Rozebereme si několik základních funkcí a podrobně se ponoříme do formátování vstupu a výstupu.

2.7.1 Hlavičkový soubor

Následující funkce pro vstup a výstup jsou deklarovány v hlavičkovém souboru `stdio.h`. Název je zkratkou z *standart input/output* – a česky: standardní vstup/výstup.

Musíme jej zahrnout do svého programu předtím, než budeme jakoukoliv z těchto funkcí používat. Pokud chceme provádět vůbec nějaký vstup nebo výstup, stejně budeme muset nějakou funkci uvedenou v některém hlavičkovém souboru použít, neboť mezi základními příkazy nemáme žádnou funkci pro vstup/výstup.

2.7.2 `putchar()` a `getchar()` – vstup/výstup znaku

Funkce `getchar()` a `putchar()` slouží ke vstupu a výstupu jediného znaku ze standardního vstupu a na standardní výstup.

2.7.2.1 Standardní vstup a výstup

Zřejmě je na čase povědět si něco o standardním vstupu a výstupu. Dobrá.

Nejprve si nastíníme pojem datový proud.

Údery z klávesnice, údaje uložené v souboru nebo výstup na obrazovku jsou všechno informace, pro tento případ nazývaná data. Taková data mohou proudit ze svého zdroje k nějakému cíli. Například z klávesnice do naší funkce čtoucí uživatelův vstup. V počítači je takových proudů spousta. My potřebuje najít ten, co vede z klávesnice a potom také ten, co vede na obrazovku, protože chceme uživateli poslat odpověď na jeho vstup. Takovýto vstup a výstup je nejběžnější a kdybychom museli při každé takové akci psát pojmenování proudu, který chceme použít, zbytečně bychom se upsali.

Proto existují funkce, pracující s takzvaným standardním vstupem a výstupem. To, co jim dáme za příkaz vypsat, pošlou na standardní výstup, který je napojen na jeden z proudů a když zase chceme číst, čtou implicitně standardní vstup, který je rovněž již někam napojen. Standardní vstup je normálně napojen na proud dat z klávesnice a standardní výstup na obrazovku.

To by pro začátek stačilo, více o proudech a manipulaci s nimi později.

2.7.2.2 `getchar()`

Funkce `getchar()` slouží ke vstupu jednoho znaku ze standardního vstupu. Nemá žádné parametry. Vrací hodnotu `int`, což je onen přečtený znak.

Formálně:

```
int getchar();
```

Funkce `getchar()` sice vrací pouze jeden znak, ovšem až poté, co uživatel stiskne klapku `enter`. Bohužel i klapka `enter` má svůj kód a je tudíž také čtitelným znakem. Co to znamená? Pokud uživatel na odpověď typu `_a_no/_n_e` odpoví vyťukáním celého slova ano, následovaného klapkou enter, ve vstupním *bufferu*[29] budou čtyři znaky.

Použitím `getchar()` dostaneme první z nich – tedy to, co chceme. Problémem ale je, že ostatní znaky z bufferu nezmizí – zbude nám v něm `no<ENTER>`. Při dalším použití `getchar()` dostaneme samozřejmě znak "n". Z toho vyplývá nutnost *buffer* vyprazdňovat ihned po přečtení znaku.

Jako **cvičení** naprogramujte funkci, která přečte celý vstup a vrátí z něj první znak. V *bufferu* nesmí zbýt nic, ani znak nového řádku. Pokud se vám to nepovede, potřebný cyklus je napsán někde níže.

2.7.2.3 `putchar()`

Funkce `putchar()` slouží k výstupu jednoho znaku na standardní výstup. Má jeden parametr typu `int`, kterým je znak posílaný na výstup.

Formálně:

```
int putchar(int vystupni_znak);
```

Nepovinné **cvičení**: Pomocí funkce `putchar()` vypište na obrazovku text "Hello world!".

2.7.3 `printf()` a `scanf()` – formátovaný vstup/výstup

Následující funkce se budou týkat poněkud složitějšího vstupu a výstupu. S jistou formou abstrakce je možno říci, že po absolvování této kapitolky budeme umět jediným řádkem kódu vypsat skloubení mnoha datových struktur a textů přes polovinu obrazovky – ale co, nebuďme troškaři, klidně přes celou.

Nejdříve si obě funkce trochu přiblížíme a pak přejdeme na to, co je činí tak silnými – formátovací řetězce.

2.7.3.1 `printf()`

Funkce `printf()` zabezpečuje formátovaný výstup na standardní výstup. Česky je "print" "tiskni" a znak "f" je zástupce za slovo *formatted*, tedy formátovaně.

Nejdříve si uvedeme formální deklaraci:

```
int printf(const char *format,...);
```

A nyní podrobněji co je co.

Návratovou hodnotou je počet znaků, které se vytiskly.

Parametry funkce `printf()` můžeme rozdělit na dvě skupiny. První skupinu tvoří tzv. "formátovací řetězec" (uveden jako `const char *format`) a do druhé skupiny patří všechny ostatní parametry, v deklaraci označené třemi tečkami.

Jak jste jistě již uhodli, funkce `printf()` může být volána s proměnlivým počtem parametrů. Druhá skupina parametrů může být dokonce prázdná, čili formátovací řetězec bude jediným parametrem, v takovém případě ale samozřejmě nesmíme psát čárku.

[29] Vyrovnávací paměť, kde se znaky uloží. Tato paměť zachovává pořadí znaků. Čtením se znaky z bufferu mažou.

Formátovací řetězec vlastně popisuje, jak bude výstup vypadat a odvolává se na všechny proměnné, které budou ve výstupu použity. Tyto proměnné musíme uvést v druhé skupině parametrů a počet těchto parametrů musí být stejný jako počet proměnných, které formátovací řetězec referuje. Podstatné je také pořadí – proměnná, která je ve formátovacím řetězci referována jako první, je první parametr za formátovacím řetězcem (první parametr druhé skupiny) a tak dále.

Formátovací řetězec je rozebrán níže.

2.7.3.2 `scanf()`

Funkce `scanf()` provádí formátovaný vstup[30].

Formálně:

```
int scanf(const char *format,...);
```

Funkce vrací počet objektů které se podařilo načíst (podle formátovacího řetězce), nebo EOF (-1) pokud při čtení narazila na znak EOF nebo došlo k chybě.

Podobně jako u `printf()`, parametry se rozdělují na formátovací řetězec a parametry objektů, které mají být čteny. Předchozí věta ovšem není přesná, protože místo proměnných se jako parametry funkce `scanf()` píší adresy těchto proměnných[31], neboť funkce v C nesmí měnit hodnoty svých parametrů.

Formátovací řetězec je trošku odlišný od formátovacího řetězce pro funkci `printf()`, což je rozebráno níže.

2.7.3.3 Formátovací řetězec pro `printf()`

Zkusíme si pravidla pro formátovací řetězec zesumarizovat v několika bodech:

1. Jak už vyplývá z názvu řetězec, je celý uzavřen do uvozovek.

2. Běžné znaky, které do řetězce zapíšeme – nezačínají na znak procento – se vytisknou tak, jak jsme je napsali.

3. Speciální znaky (znak nového řádku, tabulátor) se vytisknou tak, jak mají.

4. Znak procento je řídícím znakem a znamená, že následující znak (znaky) specifikují typ a úpravu výpisu proměnné, která je jedním z parametrů funkce `printf()`.

5. Pokud chceme vytisknout znak procento, napíšeme jednoduše dva tyto znaky za sebou.

Zadání odkazu na proměnnou vypadá obecně takto (nelekněte se):

```
%[úprava][šířka][přesnost][velikost]typ
```

Rozebereme si postupně všechny části:

[30] *Scan* znamená pohlížet, bádat, hledat.

[31] Přesněji řečeno ukazatele na proměnné.

Úprava

- `-` ... výstup bude zarovnán vlevo. Má význam pouze tehdy, pokud nějak specifikujeme šířku výstupu.
- `+` ... číslo bude na výstupu vždy uvozeno znaménkem[32].
- `#` ... vliv na výstup se různí podle typu výstupu. Např pro oktalová čísla přidá prefix 0, pro šestnáctková prefix 0x a u reálných čísel si vynutí zápis tečky, i kdyby neměly desetinnou část.

Šířka

Specifikuje, na kolik míst minimálně se bude příslušný výstup tisknout. Pokud tedy bude mít výstup méně znaků než udává šířka, tak se na šířku znaků doplní příslušným počtem mezer. Pokud šířku uvodíme nulou, jako doplňující znaky se místo mezery použijí nuly.

Přesnost

Závisí na typu výstupu, začíná vždy tečkou[33]. U reálných čísel určuje počet desetinných míst a u řetězců maximální počet tištěných znaků.

Velikost

Rozebírat nebudeme, pravděpodobně ji nebudete potřebovat (ani popis u ostatního nebyl úplný), viz literatura, manuálové stránky nebo help[34].

Typ

Je pro nás nejdůležitějším argumentem. Určuje typ příslušné proměnné, uvedené za formátovacím řetězcem jako jeden z dalších parametrů.

d,i výstup celého čísla (`int`)

u výstupní číslo je `unsigned`

o taktéž `unsigned`, výstup jako oktalové číslo

x `unsigned`, výstup hexadecimálně

X `unsigned`, výstup hexadecimálně s velkými znaky

f `float` nebo `double`, vystoupí celé (ne jako *mantisa* a *exponent* ale v přirozené podobě). Před výstupem se konvertuje na `float`.

e,E `float` nebo `double`. Vystoupí v podobě *mantisa*E*exponent*. Před výstupem se konvertuje na `double`.

[32] I v případě kladného čísla.
[33] Aby bylo poznat, kde končí šířka a začíná přesnost.
[34] Jen manuálová stránka pro rodinu funkcí `printf()` má téměř 600 řádek.

g,G `float` nebo `double`. Vystoupí jako kratší z podob konverzí f nebo e.

c očekává se `int`, na výstup se napíše znak s odpovídajícím ASCII kódem

s řetězec ukončený binární nulou (znak NUL)

p očekává ukazatel na `void`, výstupem by měla být adresa v hexadecimálním tvaru

n odpovídající parametr ukazuje na adresu, kam se uloží počet dosud vytištěných znaků[35]

Ufff, to bylo něco, že? Takže nějaké příklady:

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

Vytiskne tři prázdné řádky

```
printf("%d",i);
```

Vytiskne obsah proměnné i, i je celočíselné

```
printf("Prvni: %e\tDruha: %d\n", o_f, i);
```

Vytiskne text "Prvni: ", dále obsah proměnné o_f typu `float` v podobě *mantisaEexponent*, dále tabulátor, řetězec "Druha: ", obsah proměnné i, která je decimální, a odřádkuje.

```
printf("%s%e%s%d\n", "Prvni: ", o_f, "\tDruha:", i);
```

Totéž jako předchozí příklad.

2.7.3.4 Formátovací řetězec pro `scanf()`

Je třeba si uvědomit, že cokoliv kromě odkazů na adresy proměnných uvedených ve volání funkce za řetězcem do řetězce zapíšeme, bude porovnáváno se vstupem a tudíž se v drtivé většině případů nijak neexperimentuje s formátovacím řetězcem pro `scanf()`, nýbrž se zde napíší pouze ony odkazy.

Proč? Protože velice často dochází k chybám. Do formátovacího řetězce se vyplatí cpát jiné znaky jen tehdy, když jsme si absolutně jisti, že ve vstupu skutečně budou, a chceme je při čtení do proměnné vyřezat – například při čtení ze souboru. To je ale lepší si vyzkoušet než přečíst.

Opět se pokusím o výčet vlastností řetězce:

- Řetězec je uzavřen do uvozovek
- Mezera v řetězci spolkne jakákoliv bílá místa
- Běžné znaky se porovnávají se skutečným vstupem
- Odkazem na adresy proměnných je opět znak procento

[35] V tomto případě se tedy nic netiskne.

Zadání odkazu na ukazatel (adresu proměnné) vypadá v scanf takto:

```
%[*][šířka][velikost]typ
```

* ... následující objekt o velikosti typ, se nikam nepřiřadí – tento odkaz se ve skutečnosti nikam neodkazuje – proměnná pro tento odkaz není ani uvedena.

Šířka ... udává maximální počet znaků, které budou načteny

Velikost ... pro upřesnění velikosti.

Typ

d decimální číslo, odpovídající ukazatel je ukazatel na `int`

i čte decimální, oktalové (uvozeno `0`) nebo hexadecimální (uvozeno `0x` nebo `0X`) číslo, ukazatel na `int`.

u decimální číslo, ukazatel na `unsigned`

o oktalové číslo, ukazatel na `unsigned`

x hexadecimální číslo, ukazatel na `unsigned`

e,f,g reálné číslo, ukazatel na `float`

c vstup jednoho znaku

s vstup řetězce, za konec se přidá znak NUL

[...] Vstupem je nejdelší řetězec skládající se ze znaků uvedených v závorkách.

[^...] Vstupem je nejdelší řetězec neobsahující ani jeden ze znaků v závorkách.

Jako oddělovač řetězců jsou považovány jakékoliv bílé znaky. Při čtení řetězce jsou tedy všechny úvodní bílé znaky přeskočeny a čte se až první nebílý znak. To byla kupodivu ta dobrá část zprávy. Ta špatná je, že čtení řetězce je ukončeno prvním bílým znakem – pokud chceme přečíst větu nebo řádek, raději bychom měli použít funkci `gets()`.

Nějaké příklady:

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

Načte celé číslo do proměnné *i*.

```
scanf("%d:%*d:%d", &a, &b);
```

Načte trojici celých čísel, oddělených dvojtečkami, první a třetí číslo si zapamatuje a prostřední zahodí.

```
scanf("%s", str);
```

Načte řetězec do *str*, není třeba operátor & ke zjištění adresy, protože řetězec je vlastně adresa prvního znaku (viz 3.7.1).

Cvičení: Máme soubor ve kterém jsou řádky typu:

```
Jméno:Příjmení<nějaké mezery>plat
```

Napište program, který čte data pomocí `scanf()` a na standardní výstup vytiskne pomocí `printf()` odpovídající řádky ve tvaru *Příjmení* = *plat*. Protože zatím asi neumíte používat soubory, čtěte ze standardního vstupu. Nezapomeňte na dvě věci:

1. Při `scanf()` uvádíte adresu proměnné.

2. Zadat z klávesnice vstup.

Kapitola 3

Stavíme první patro

Tato kapitola má velice skromný název, ale troufám si říci, že pokud ji úspěšně zvládnete, nebude vám už chybět mnoho, abyste se mohli považovat za programátory v C.

Musím ale ještě zdůraznit předpoklad, že ovládáte kapitolu předchozí a máte ji důkladně procvičenu. Pokud ne, k dalšímu čtení se vraťte až když si vše procvičíte a teď toho ihned nechejte.

Dobrá, čtou už tedy pouze ti, kteří si vše předchozí důkladně procvičili a vše kromě nějakých dat, která si vždy dohledají[1], si pamatují.

Výborně, co se tady naučíme?

Podíváme se na nějaké další možnosti jak ovlivňovat průchody cyklů a běh programu vůbec, pořádně se pustíme do souborů, ukážeme si některé složitější datové typy, řekneme, jak funguje přetypování, naučíme se alokovat paměť pro dynamické objekty a nakonec si uvedeme další operace s řetězci a vůbec se na ty zvláštní potvůrky podíváme z menší vzdálenosti než doposud.

Pojďme na to.

3.1 Další řídící příkazy

Nyní si prozradíme některé další příkazy ovlivňující běh cyklů, skákající v těle programu z místa na místo, ukončující funkce a celý program.

Na závěr mini-kapitolky si pro odlehčení uvedeme jeden z dalších zbývajících operátorů – jediný ternární operátor v C, neboli operátor otazníku. Je to další céčkovská specialitka, už teď se na ni můžete těšit.

3.1.1 `break` a `continue`

Občas se nám v těle cyklu může stát, že potřebujeme cyklus ukončit a pokračovat prováděním příkazů následujících po těle cyklu.

K takovému účelu slouží příkaz `break`, samozřejmě pouze tehdy, pokud není schován do některé větve příkazu `switch`. *Break* znamená česky přestávka, zlom, přeruš.

Je docela dobře možné představovat si, že `break` znamená zrušení nějaké nadřazené instance, která právě probíhá. Touto instancí může být buďto příkaz `switch` nebo kterýkoliv

[1] Třeba jak se určuje počet desetinných míst při vstupu `float`.

z cyklů. Podstatné je, že `break` má sílu potřebnou k prolomení pouze jedné takové instance, a to té jemu nejbližší. To znamená, že pokud máme cyklus, uvnitř cyklu `switch` a `break` v nějaké větvi tohoto `switch` , `break` přeruší `switch` (nejbližší, nejméně nadřazenější), kdežto na vyšší instanci – cyklus – již nedosáhne.

Podobně, pokud bychom měli cyklus v nějaké větvi příkazu `switch`, tak příkaz `break` umístěný v tomto cyklu by přerušil pouze cyklus, ne větev. A pokud bychom měli cyklus ve větvi `switch` a tu v cyklu a ten... dost. Myslím, že to už chápeme.

Někdy se zase stane, že potřebujeme přeskočit zbylé operace prováděné v této smyčce cyklu a je třeba pokračovat v další iteraci. Tedy, potřebujeme ze současné polohy v cyklu skočit těsně před konec smyčky cyklu. K tomu slouží příkaz `continue`. Samozřejmě se vztahuje k nejbližšímu nadřazenému cyklu.

Že by nějaký příklad? Slyším vás křičet dostatečně hlasitě na to, abych nějaký uvedl. Budiž.

```
#include <stdio.h>
#define MAXDELKA 100
main()
{
   int delka;
   char *ret;

 /* alokovani pameti pro dynamicky retezec,
  * neleknete se
  * rozebereme si pozdeji, v tomhle finta
  * (viz cviceni) neni
  */
  ret = (char *)
      malloc((MAXDELKA+1)*sizeof(char));

/* Sikovna cesta jak zapsat nekonecny cyklus */
   for (;;) {

      printf("Zadej slovo (konec=konec): ");
      scanf("%s", ret);

/* pokud prijde ‘‘konec’’, skoncime cyklus */
      if ( strcmp(ret, "konec") == 0 ) {
         break;
      }

/* pokud je delka retezce ‘‘out of range’’,
 * jdeme od zacatku
 */
      if ( (delka = strlen(ret)) > 10 ) {
         printf("Retezec je moc dlouhy!\n");
         continue;
      }
```

```
/* jinak tiskneme delku retezce */
      printf("Delka retezce je: %d\n", delka);
   }
}
```

Jako **cvičení** máte zjistit, proč se s příkladem děje to, co se děje, když zadáte větičku "jsem vetsi nez deset!" nebo "co to sakra delas?" nebo "tohle strasnedlouheslovo mi to ujasni".

3.1.2 `goto`

Tak. A jsme u toho. Ti z vás, kteří už někdy nějaký strukturovaný jazyk používali, asi tuší, kde je problém. Ostatní se to hned dozví.

Problém je v samotném příkazu, o kterém se budeme bavit. Je totiž pramenem sporů. Ač mnoho lidí tvrdí, že ve strukturovaném jazyce nemá tento příkaz co dělat, stále ho zde nacházíme. Proč?

Abychom nechodili kolem horké kaše, uvedeme si velice krátký výřez dialogu mezi zavilým odpůrcem a nadšeným zastáncem goto, učitelem-programátorem Antigoto a programátorem-hackerem Gotoheaven (Antigoto kontroluje program Gotoheavena (značně upraveno)).

Antigoto (zděšeně): "Proboha, co to tady je?"

Gotoheaven (lakonicky): "Goto."

Antigoto (znalecky): "To se přece dá nahradit cykly a podmínkou."

Gotoheaven (znechuceně): "To vím taky, ale bude to delší a neefektivní."

Antigoto (skepticky): "O efektivitě by se dalo pochybovat."

Gotoheaven (zaníceně): "Ale o eleganci určitě ne."

Antigoto (pohoršeně): "Porušuje to zásady strukturovaného programování a navíc se v tom chaosu pak nikdo nevyzná."

Gotoheaven (nevzrušeně): "Já se v tom vyznám ... a zásady? Who cares?"

Co dodat? Radši `goto` nepoužívejte, protože člověk musí hledat místo, na které se skáče a pokud máte příkazů `goto` více, začíná to být opravdový chaos. Navíc, pokud se jedná o větší projekt, který dáváte dohromady s někým jiným nebo po vás má někdo kód číst a upravovat, `goto` určitě není to pravé ořechové.

Na druhou stranu, pokud vidíte, jak krásně by vám mohlo ulehčit práci, tak jej použijte. Ale připravte se na to, že váš kód pak někdo může označit za nečistý a že ztrácíte hodně na přehlednosti.

Dobrá, co vlastně `goto` je?

Příkaz `goto` zajišťuje skok na předem připravený bod programu, který se nachází ve stejné funkci[2]. Takovému bodu se říká návěští a značí se jménem a dvojtečkou, např:

```
SeMhLe:
```

[2]Pokud chcete opravdový guláš (skok do jiné funkce), projděte si `longjmp()` a `setjmp()`

Na návěští skočíme příkazem `goto návěští`, v našem případě tedy

```
goto SeMhLe;
```

Proboha, když už `goto` jednou za deset let použijete, hlavně si nevolte takové pitomé názvy jako já.

Cvičení? Ne, radši ne, potom byste všude roztrubovali že vás nutím používat `goto`.

3.1.3 `return` a `exit()`

Příkaz `return` jsme si již prošli, jen je třeba uvědomit si, že jeho požití ve funkci `main()` má za následek ukončení programu a také to, že i funkce `main()` vrací nějakou hodnotu. Nejčastěji je to hodnota typu `int` a pokud vše skončilo dobře, vracíme nulu – což je standardní hodnota, kterou většina funkcí vrací, využívá se například v *shellu*, obecněji ve volajícím procesu (odborně "rodič" – angl. *parent*).

Existuje ještě jedna funkce na ukončení programu a to sice funkce `exit()`. Voláním této funkce dosáhneme ukončení programu, ať ji voláme z kteréhokoliv místa – třebas z funkce vnořené.

Formálně:

```
void exit(int status);
```

Jako parametr zadáváme hodnotu, která se má vrátit rodiči. Použití této funkce není příliš časté, v drtivé většině případů se program končí příkazem `return` ve funkci `main()`. Asi nejčastějším použitím je případ, kdy program narazí na závažnou chybu a nemá smysl dále pokračovat. Vypíše tedy hlášku a korektně se ukončí.

3.1.4 Operátor otazníku

A nyní přichází dlouho slibovaný operátor otazníku. Co to je? Tohle: `?`

Ne pardon, jeho syntax je vlastně opačný `? :`

Legrácek bylo dost, formální zápis vypadá takto:

```
(podmínka) ? výraz1 : výraz2 ;
```

Nu a co tento prapodivný operátor dělá?

Pokud se podmínka vyhodnotí jako pravdivá, operátor vyhodnotí první výraz a vrátí jeho výsledek. Jestliže je podmínka nepravdivá, operátor vrátí výsledek vyhodnocení druhého výrazu.

Nějaký příklad (všude je uváděn příklad s maximální hodnotou a já už ho nemůžu vystát, takže dáme něco jiného):

```
int save, answer;
pritntf(‘‘\n\tyes\tor\tDIE\n\n’’);

/* precteme cely vstupni buffer,
 * prvni znak do answer
```

```
 */
for (save=answer=getchar(); save!='\n';) {
    save=getchar();
}
```
[3]

```
/* podle answer vytiskneme odpoved
 * (jako parametr pro printf
 * se dosadi jeden z vyrazu za otaznikem
 * po vyhodnoceni podminky)
 */
printf ("\n%s\n",
   (answer=='y') ? "Coward but alive" :
   "You're like bear - without fear!.");
```

Jejda, slyším vás křičet jak je to nespravedlivé, co za zrůdný příklad jsem vám podstrčil. Nu dobrá, ukažme si i něco jednoduššího, třeba výše zmiňované maximum ze dvou čísel (*grrrr*!).

```
/* do x ulozi vetsi z a,b */
x = (a>b) ? a : b;
```

Cvičení: Vymyslete krátký prográmek s vhodně použitým operátorem otazníku.

3.2 Vstup a výstup II – soubory

Ano, konečně se dostáváme k souborům. V této kapitolce si probereme soubory velice důkladně. Dá se říci, že se o nich už o moc více nedozvíme, leda nám bude něco jasnější při povídání o ukazatelích a proudech – i když o těch bude řeč již tady.

3.2.1 Úvod k souborům

Půjdeme na to velice jednoduše. Co je to soubor? Posloupnost *byte* na disku, či na nějakém jiném médiu. Nebudeme se zabývat *low-level* přístupem k médiím, poněvadž díky operačnímu systému nemusíme vědět, na kolika blocích jaké velikosti je soubor uložen, na kolik částí je fragmentován a tak dále. Z hlediska C jej vidíme jako posloupnost znaků, lépe řečeno, *byte*.

Ke každému souboru můžeme v C přistupovat dvěma způsoby.

První způsob je jako k binárnímu souboru, kdy C při čtení a psaní manipuluje se znaky přesně tak, jak mu náš program káže.

Můžeme se ale k souboru chovat jako k souboru textovému, což přináší určité změny v operacích s daty. Při tomto přístupu C do souboru určité znaky přidává a některé nám neukazuje, naprosto bez našeho vědomí.

U textového souboru se totiž předpokládá, že je dělen na řádky. Jak jsme si již říkali, konec řádky je v různých OS řešen různě a my se opravdu nepotřebujeme starat, jestli máme číst/psát CR+LF nebo jenom LF, ať se o to postará překladač za nás. Navíc máme několik hezkých funkcí, které nám umožňují číst a zapisovat řádky pěkně najednou.

[3]Cyklus se dá zjednodušit tak, aby neměl vůbec žádné tělo – t.j. aby tělem cyklu byl pouze prázdný příkaz.

3.2.2 Datový typ FILE

Co je všem souborům společné, je datový typ, který používáme při definici proměnné používané pro přístup k souborům. Je to typ FILE. Proměnnou, kterou budeme používat pro ukazování na nějaký soubor definujeme takto:

```
FILE *f1, *f2, *f3;
```

Hvězdička před jménem proměnné značí, že proměnná je ukazatelem. Ukazatele si budeme probírat později (v 3.5.1), zatím potřebujeme vědět pouze tolik, že ona hvězdička je v definici proměnné ukazující na soubor nezbytná.

Takový soubor, to je vlastně také datový proud, že? Když se nad tím zamyslíme, standardní vstup a výstup jsou také datové proudy... Hmm, mohly by mít něco společného, že ano?

Mno, je načase to prozradit, existují konstanty `stdin` a `stdout`, které jsou ukazateli na naprosto stejnou strukturu FILE, kterou budeme používat k přístupu k souborům, přičemž ukazují na standardní vstup a výstup.

Už vám jistě svítá. Není tomu dávno, co jsem tvrdil, že funkce `putchar()`, `getchar()`, `printf()` a `scanf()`, jsou funkcemi pro přístup k datovému proudu, specializované na práci se `stdin` a `stdout`. Jistě tedy existují nějaké obecnější a velmi podobné funkce na práci s jakýmkoliv datovým proudem. Přesně tak to je, ale s jejich prozrazením ještě chvilku počkám, nejprve se naučíme naše soubory otevírat[4].

Ještě než se do všeho pustíme bych se chtěl omluvit za možné zmatení nad mícháním pojmů FILE, FILE *, datový proud (angl. *stream*) a soubor. Je třeba vyjasnit co je co.

Soubor je kus dat uložených na nějakém médiu.

Datový proud je obecně něco, co poskytuje data, nebo je možno do toho data zapisovat, tento pojem již dříve přiblížen byl.

`FILE *` je ukazatel na něco typu FILE.

`FILE` je typ v jazyce C. A teď přijde to matoucí. Ačkoliv se jmenuje FILE (česky "soubor"), je to jakýkoliv obecný datový proud. Jeho jméno je zachováno z důvodů historických a zřejmě zpětné kompatibility. Datový proud ze/do souboru se chová jako datový proud a proto je na něj odkazováno typem FILE.

Budu se snažit, abych nepsal soubor a nemyslel tím FILE, což je datový proud, ale znáte to... nikdo není bez chybičky.

3.2.3 Otevírání a zavírání

Pro otevření souboru použijeme funkci `fopen()`

Formálně:

```
FILE *fopen (const char *path, const char *mode);
```

[4] `stdin` a `stdout` se otevřou automaticky, proto jsme se o tom nezmiňovali.

Čili funkce `fopen()`[5] vrací ukazatel na typ `FILE`[6] a sama má dva parametry – řetězce[7] *path* a *mode*. V případě neúspěchu vrací ukazatel `NULL`.

Řetězec *path* obsahuje jméno a případnou cestu k souboru ve standardním tvaru.

Řetězec *mode* označuje v jakém módu bude soubor otevřen, což ovlivňuje jak prostředky které budeme mít k dispozici, tak druh přístupu k souboru. Zde jsou možné módy:

r ... otevře existující soubor pro čtení. Ukazatel je nastaven na začátek souboru.

r+ ... otevře existující soubor pro čtení i psaní, ukazatel na začátek.

w ... pokud soubor neexistuje, je vytvořen nový, pokud existuje je oříznut na nulovou délku – data se ztratí, ukazatel samozřejmě na začátek.

w+ ... čtení i psaní, jinak jako w.

a ... soubor se otevře pro psaní na konec (připisování). Jestliže soubor dosud neexistoval, je vytvořen. Ukazatel na konec souboru.

a+ ... jako a, jen máme možnost čtení.

Dále můžeme k řetězci popisujícímu mód přidat písmeno *b*, které určuje binární přístup k souboru – textový přístup je implicitní.

Poznámka pro módy umožnující zároveň čtení i zápis: Při přechodu od psaní ke čtení a naopak je třeba zavolat jednu z funkcí nastavujících pozici ukazatele v souboru (viz 3.2.8) a to z důvodu synchronizace – není totiž zaručeno pořadí provádění operací čtení/psaní.

K zavírání souboru použijeme funkci `fclose()`

Formálně:

```
int fclose(FILE *stream);
```

Jediným parametrem funkce je ukazatel na soubor, který chceme zavřít. Po úspěšném uzavření vrací funkce nulu.

3.2.4 Vstup a výstup

Již umíme soubory otevírat a zavírat, ještě je třeba do nich něco zapsat, popřípadě z nich něco přečíst. Vrhněme se na ty tak dlouho očekávané funkce.

3.2.4.1 Vstup a výstup znaku

Pro čtení jednoho znaku použijeme funkci `fgetc()`

Formálně:

```
int fgetc(FILE *stream);
```

[5] fopen=*file open*, česky *file*=soubor, *open*=otevření.

[6] Kvůli hvězdičce před jménem funkce.

[7] Pokud jsem vám nasadil brouka do hlavy a uvažujete, proč jsou tam ty hvězdičky, chvilku si počkejte, v kapitolce 3.7.1 se to dozvíte.

Parametrem funkce je ukazatel na soubor, ze kterého chceme číst.

Návratová hodnota je `int`, tedy hodnota znaku. Ukazatel se v binárním souboru posune o jeden znak dále[8]. V případě chyby vrátí `EOF`.

Je možno použít funkci `getc()` s naprosto stejnou syntaxí, většinou však bývá implementována jako makro a tudíž se může hodnota *stream* vyhodnocovat vícekrát než jednou, což náš program určitě neurychlí.

Pro zvídavé: Co bychom dostali, kdybychom napsali

```
a=fgetc(stdin); resp. a=getc(stdin);?
```

Odpověď: Totéž co při `a=getchar();`

Pro zápis jednoho znaku použijeme funkci `fputc()` – nebo opět kratší `putc()`.

Formálně:

```
int fputc(int c, FILE *stream);
```

Parametrem je po řadě kód zapisovaného znaku a ukazatel na soubor.

Návratová hodnota je `int`, konkrétně kód zapsaného znaku, v případě chyby `EOF`.

```
fputc(a, stdout);
```

je totéž co

```
putchar(a);
```

3.2.4.2 Formátovaný vstup a výstup

Pro tento druh operací máme funkce `fprintf()` a `fscanf()`.

Formálně:

```
int fprintf(FILE *stream, const char *format, ... );
int fscanf(FILE *stream, const char *format, ... );
```

Vrací počet znaků na výstupu/vstupu, nebo -1 při chybě.

Formátovací řetězec se řídí naprosto stejnými pravidly jako u funkcí `printf()` a `scanf()`.

Cvičení (z minula, klidně vezměte hotový příklad a předělejte jej na soubory): Máme soubor ve kterém jsou řádky typu:

```
Jméno:Příjmení<nějaké mezery>plat
```

Napište program který čte data pomocí `scanf()` a na standardní výstup vytiskne pomocí `printf()` odpovídající řádky ve tvaru “Příjmení = plat”.

[8] V případě textového souboru není zaručeno.

3.2.4.3 Vstup a výstup řádky

Když chceme zapsat pouze jednoduchý řetězec, můžeme použít funkci `fputs()`
Formálně:

```
int fputs(const char *s, FILE *stream);
```

Parametry jsou řetězec a ukazatel na soubor. Funkce do souboru nezapíše nulový znak (`\0`), ale ani znak konce řádky. Při úspěchu vrací nenulové číslo, jinak `EOF`.

Existuje samozřejmě úprava funkce `fputs()` pro výpis na standardní výstup a to – nečekaně – funkce `puts()`. Její syntaxe i návratová hodnota je vám určitě jasná, ovšem je zde jeden malý rozdíl oproti `fputs()` – funkce `puts()` na standardní výstup vypíše znak nové řádky.

Pro vstup řetězce ze souboru – limitním znakem je konec řádky – používá se u textových souborů – existuje funkce `fgets()`.

Formálně:

```
char *fgets(char *s, int size, FILE *stream);
```

Což vypadá dosti hrozivě, ale je ve skutečnosti velice jednoduché. Prvním parametrem funkce je buffer, kam se budou data číst. Druhým je velikost tohoto bufferu a třetím ukazatel na soubor, ze kterého se bude číst.

Funkce vrací ukazatel na buffer, v případě neúspěchu ukazatel `NULL`.

Funkce čte data tak dlouho, dokud nenarazí na znak nového řádku – v tom případě jej ale také uloží do řetězce – nebo dokud nenarazí na konec souboru. Řetězec je uzavřen nulovým znakem.

Funkce přečte nejvíce *size-1* znaků (jeden znak pro `\0`).

Před používáním funkce je třeba umět pracovat s řetězci – viz 3.7.1.

Existuje samozřejmě varianta funkce pro čtení standardního vstupu – funkce `gets()`
Formálně:

```
char *gets(char *s);
```

Má dva podstatné rozdíly: Znak nového řádku není uložen do řetězce, nijak nekontroluje velikost *bufferu* (není parametr *size*).

Cvičení: Napište program, který cyklí v následující smyčce:

1. vypíše uživateli jméno vstupně/výstupního souboru (na začátku žádné), nabídne změnit jméno tohoto souboru a vybrat si mezi vstupem a výstupem
2. pokud se změní jméno souboru, cyklus jde opět od začátku
3. při výstupu ze souboru se obsah souboru vytiskne na obrazovku, nová iterace cyklu
4. při vstupu do souboru bude program číst další uživatelův vstup a tisknout jej do souboru, tak dlouho, dokud nenarazí na `EOF` (`<Ctrl-D>`) nebo na předem domluvený řetězec (zařiďte se jak chcete)

- dejte pozor, aby se program nepokoušel otvírat soubor, pokud uživatel ještě nezadal jméno!
- k alokování řetězce o délce *delka* použijte následující:

```
char *retezec; retezec = (char *) malloc(delka*sizeof(char));
```

3.2.5 Konec řádky a souboru

Jen malinká poznámečka ohledně konce řádky. Konec řádky se anglicky řekne *end-of-line* a někdy se označuje jako `EOLN`. Tato konstanta ale není nikde definována, proto jestli chceme testovat konec řádky, nemůžeme ji použít.

Jediný možný způsob jak porovnávat nějaký znak se znakem konce řádky je porovnat jej se znakem `\n`, přičemž se spoléháme na překladač, že tento symbol v závislosti na platformě správně interpretuje[9].

3.2.5.1 Konec souboru

K testování konce souboru můžeme použít dva způsoby. V případě textových souborů jsou ekvivalentní, v případě binárních souborů ne.

Prvním způsobem je porovnání načteného znaku s konstantou `EOF`. Pokud se znaky rovnají, jsme na konci souboru, pokud ne, přečtený znak je kousíčkem dat souboru. V textových souborech je vše v pořádku, ovšem použití v binárních souborech není zrovna nejlepší. Mohlo by se stát, že díky konverzi při čtení by soubor byl menší. Lépe řečeno, první znak -1 (hodnota `EOF`) by znamenal konec souboru, i když skutečný konec by to nebyl[10].

Druhým způsobem je volání funkce `feof()`, formálně

```
int feof(FILE *stream);
```

Parametrem je samozřejmě ukazatel na soubor. Funkce vrátí nenulovou hodnotu, pokud ukazatel ukazuje na konec souboru. Pokud nejsme na konci souboru, funkce vrací nulu. Tato metoda je bezpečná i pro binární soubory.

3.2.6 Binární soubory

Již několikrát byly opěvovány, tak teď je tady konečně máme.

Kromě toho, že nám k zápisu a čtení fungují již dříve uváděné funkce[11], máme k dispozici další dvě a to `fread()` a `fwrite()`. Podívejme se na ně pořádně.

```
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream);
```

Funkce `fread()` čte ze souboru *stream* *nmemb* bloků velikosti *size* a ukládá je na adresu, kam ukazuje *ptr*. Vrací počet načtených bloků, nebo, v případě chyby či konce souboru, nulu. Rozlišení mezi chybou a koncem souboru se provede pomocí volání `feof()` resp. `ferror()` – viz 3.2.7.

Paměť, kde se bude ukládat, je nejdříve třeba správně naalokovat.

Pokud takto čteme textový soubor, jsou prováděny obvyklé konverze znaků (konec řádky).

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

[9] Tedy, v MSDOS a MSW jako CR+LF, v OS Un*x a Linux jako LF.

[10] Možná se vám něco podobného již stalo při otevírání nějakého binárního souboru textovým editorem.

[11] Proč by nefungovaly, vždyť jsou definovány pro typ `FILE*`, ne pro textové soubory.

Funkce `fwrite()` zapisuje do souboru *stream nmemb* bloků velikosti *size*, které čte z adresy *ptr*. Vrací počet úspěšně zapsaných bloků, v případě chyby nulu.

Cvičení: Napište program, který kopíruje soubory z jednoho místa na druhé pomocí `fread()` a `fwrite()`, přičemž soubory je možno zadávat. Nezapomeňte testovat návratovou hodnotu `fread()` a `fwrite()`.

- k alokování paměti "pocet" bloků, kde je jeden blok velký "bytes" *byte* použijte:

  ```
  void *pamet; pamet = malloc(pocet*bytes);
  ```

3.2.7 Ošetření chyb a vrácení znaku

Datový proud není jen posloupnost znaků. Má také nějaké vlastnosti, které můžeme ověřovat, resp. jejichž hodnotu můžeme ověřovat. Patří mezi ně příznaky určující co všechno se může s tímto proudem dělat a spousta dalších, nás ale zajímá příznak chyby.

Zcela jistě jste si všimli, že mnoho funkcí vrací při chybě konstantu `EOF`. Tento rys je sice zajímavý, někdy hezky použitelný a někdy jediný rozumný, bohužel ale nepříjemný. Proč? Protože když dostanete takový `EOF`, nevíte zda došlo k chybě při např. čtení, nebo zda je pouze ukazatel na konci souboru – v čemž je samozřejmě podstatný rozdíl.

Vzato z reálného hlediska, i *end-of-file* je vlastně příznak. Je to příznak konce souboru a testuje se právě již dříve zmiňovanou funkcí `feof()`.

Stejně jako při dosažení konce souboru je nastaven příznak *end-of-file*, při chybě je nastaven příznak chyby. Testování příznaku chyby se provádí pomocí funkce `ferror()`, formálně

```
int ferror(FILE *stream);
```

a vrací nenulovou hodnotu pokud příznak nastaven je (došlo k chybě), no a nulu pokud k chybě nedošlo.

Příznaky datového proudu ovšem samy od sebe nemizí a jakmile je něco nastaví, zůstanou nastaveny, ať provedeme testů kolik chceme.

U příznaku *end-of-file* nám to vůbec nevadí, právě naopak. Tento příznak se automaticky vynuluje při přesunu ukazatele v souboru někam jinam než na konec, což je přesně to, co bychom chtěli.

Vraťme se ale k příznaku chyby. Příznak chyby musíme vyčistit sami, pokud chceme být schopni zjistit další výskyt chyby. Učiníme tak pomocí volání funkce `clearerr()`, formálně

```
void clearerr(FILE *stream);
```

která nevrací nic.

Pro určité případy, kdy jsme z bufferovaného proudu přečetli více než jsme chtěli, je možno zpět do bufferu jeden znak vrátit – při dalším čtení bude vrácen jako první.

Není nutné vrátit přímo znak, který byl přečten, je možno vrátit znak jiný, jen je nutno si pamatovat, že pokud bude proudem soubor, tento jiný znak se sice zapíše do paměťového bufferu, ale do souboru ne. Navíc, takto vrácené znaky se mohou ztratit i z bufferu – a to v případě, kdy je mezi vrácením a opětovným čtení volána některá z funkcí přenastavujících pozici ukazatele v souboru – v takovém případě se totiž obsah bufferu znovu načítá.

Ke vrácení znaku se použije funkce `ungetc()`, formálně

```
int ungetc(int c, FILE *stream);
```

Parametrem je znak, který se má do bufferu vrátit a samozřejmě ukazatel na proud. Funkce vrací kód znaku, v případě neúspěchu `EOF`[12].

Bude nám ale takové vracení vůbec k něčemu dobré?

Pro ilustraci použiji mírně upravený krásný příklad z referenční příručky knihovny GNU C:

```
int skip_whitespaces(FILE *f)
/* preskakuje bila mista */
{
   int c;
   do
      c=fgetc(f);
   while (isspace(c));
   if (ungetc(c, f)==EOF) return EOF;
   return 0;
}
```

Příklad přečte všechny bílé znaky na vstupu. Zamyslete se, co udělá, pokud narazí na konec souboru a jak je to zde řešeno.

3.2.8 Další operace

Stále nám zbývá spousta dalších operací se soubory. Neumíme s ukazatelem v souboru hýbat jinak než čtením (dál) nebo zavřením a otevřením (na začátek). Ani neumíme zjistit, kde v souboru se právě ukazatel nalézá. Hned to napravíme.

```
long int ftell(FILE *stream);
```

Funkce vrací aktuální pozici v souboru, -1 při neúspěchu. V případě binárního souboru vrátí počet *byte* od začátku souboru, v případě textového souboru záleží na implementaci, každopádně půjde vrácená hodnota použít pro opětovné nastavení ukazatele na stejné místo – prostě si ji schovejte a později použijte.

```
int fseek(FILE *stream, long int offset, int whence);
```

Nastavuje pozici v souboru *stream*. Parametr *whence* určuje, odkud se počítá vzdálenost *offset*. Za *whence* můžeme uvést konstanty SEEK_SET, SEEK_CUR, nebo SEEK_END, které znamenají začátek souboru, aktuální pozici v souboru a konec souboru.

Při úspěšné operaci vrací nulu, při chybě nenulovou hodnotu.

Funkce zajišťuje vylití bufferů[13], lépe řečeno, může být použita při přechodu od psaní ke čtení a naopak.

Funkce čistí příznak *end-of-file*.

[12] Vyzkoušejte si: co když jako c dáme `EOF`?

[13] Nebudu vám lhát, nezajišťuje, ale efekt je v podstatě stejný.

Při práci s POSIX kompatibilními systémy (UN*X, Linux, ...) se doporučuje používat varianty `fseeko()`[14].

Následující dvě funkce by měly být při nastavování a zjišťování pozice v souboru spolehlivé na kterémkoliv systému.

```
int fgetpos(FILE *stream, fpos_t *position);
```

Na adresu *position* je uložena pozice. Funkce vrací nulu, v případě chyby nenulovou hodnotu.

```
int fsetpos(FILE *stream, fpos_t *position);
```

Nastavuje pozici v souboru *stream*. Vrací nulu, při chybě nenulovou hodnotu. Čistí příznak *end-of-file*, zajišťuje synchronizaci při čtení/psaní.

Ještě si uvedeme dvě další funkce:

```
int fflush(FILE *stream);
```

Tato funkce opravdu zajišťuje vylití bufferů na disk. Vrací nulu, při chybě nenulovou hodnotu.

Pokud jí předáme nulový ukazatel (`NULL`), vylije *všechny* výstupní buffery.

Cvičení: Napište program, který bude schopen vypsat několik posledních znaků zadaného souboru (uživatel zadá jméno souboru a počet *byte*), na požádání uživatele tento výstup uloží do nějakého jiného souboru. Výše uvedenou činnost opakuje program tolikrát, kolikrát bude uživatel chtít.

Cvičení 2: Vyzkoušejte si fungování `fgetpos()` a `fsetpos()`.

3.3 Další datové typy

Pojďme se pustit do nějakých složitějších datových typů.

3.3.1 Výčtový typ

Kdybychom chtěli výčtový typ definovat jednoduše, tak řekneme, že jím určujeme množinu celých čísel, z nichž každému musíme přiřadit jméno. Fajn, zní to pěkně, ale co to je?

Nejlépe pochopíme na příkladu:

```
enum gogo {prvni=1, druhy=2, patnacty=15, xty,
           dvacaty=20, posledni};
```

Takto jsme přímo definovali objekt *gogo*. Samozřejmě můžeme udělat i datový typ *gogo*, ale o tom budeme mluvit až značně později – v 5.2.1.

Výčtový typ je tedy uvozen klíčovým slovem `enum`.

Všechny prvky množiny uzavíráme do složených závorek.

Každý prvek má přiděleno celé číslo, pokud jej neinicializujeme, dostane číslo o 1 větší než jeho předchůdce. První prvek má implicitně nulu. Zachování vzestupného nebo sestupného pořadí je pouze otázkou zvyku a pořádkumilnosti. Taktéž pokud inicializujeme, inicializujme

[14] Pro extrémně velké soubory `fseeko64()`.

všechny prvky, protože z příkladu je až na druhý pohled jasné, že "xty" bude mít hodnotu 16 a "posledni" hodnotu 21.

Jediné rozumné uplatnění tohoto typu vidím v možnosti přehledného pole symbolických konstant, které budou později v programu používány, např.

```
enum buffers {
   SIMPLE_ANSWER=5,
   PASS_READ=30,
   LOGIN_INFO=600,
   PERS_INFO=10000
};
enum connection_limits {
   CONN_PER_USER=30,
   FTP_CONN=100,
   HTTP_CONN=1000,
   MAX_USERS=500,
   TOP_CONN_LIMIT=6000
};
```

místo abychom definovali každou konstantu na zvláštním řádku, bez ladu a skladu.

3.3.2 Pole

Pole je souvislá řada prvků stejného typu. Souvislá nejenom pro programátora, ale i v paměti. To znamená, že pokud pole začíná na adrese 10, má dvacet prvků a každý prvek zabere velikost 3, tak druhý prvek začíná na adrese 13, třetí prvek na adrese 16... a konec pole je kde? Velikost je 20*3=60, počátek 10, takže poslední prvek zabere adresy 67,68,69 (už adresa 10 je obsazena, tedy 9+60=69).

Takováto datová struktura má své výhody i nevýhody. Výhody spočívají v jednoduchém výpočtu adresy prvku, a tudíž rychlému přístupu k prvkům, který má za následek velice rychlou práci s nimi, ať již s polem děláme co chceme.

Na druhou stranu je takováto homogenní struktura monstrózní v tom slova smyslu, že musí ležet v jednolitém kuse paměti a pokud je moc velká, místo v paměti se hledá špatně[15].

Co je však horší – neboť je problémem jakéhokoliv OS, jsou změny velikosti pole, protože při nich často dochází ke kopírování celého pole na jiné místo v paměti [16].

Fajn. Jak se pole tvoří a používá?

Syntaxe:

```
typprvku jménopole[početprvků]
      [={výčethodnotprvkůoddělenýchčárkami}];
```

Typ prvku může být snad cokoliv kromě funkce – nemůžeme mít pole funkcí.

[15]Některé OS mají totiž stále problémy se segmentováním paměti, přičemž většina rozumných už poskytuje nějakých 1-4 GB lineárně adresovatelné paměti.

[16]Což je na druhou stranu rychlé, protože se jedná o souvislý blok, ale i tak je to nepříjemné.

Pokud pole neinicializujeme – nebo inicializujeme jen částečně[17], musíme uvést počet prvků pole, aby překladač věděl, kolik paměti se má alokovat. Inicializované prvky se řadí automaticky od začátku pole – pokud např. vynecháme inicializaci osmi prvků, budeme mít na konci pole osm volných míst.

Neinicializované prvky dostanou přiřazenu hodnotu 0.

Pokud uvádíme výčet hodnot všech prvků, nemusíme psát počet prvků, neboť překladač si je spočítá sám.

Jednoduchý příklad:

```
int gar_field[40];
```

definujeme pole 40 prvků typu `int`.

```
char *conan[]={"Je", "nejlepší!"};
```

definujeme pole řetězců o dvou prvcích (to je tak trochu lež, ale o tom v 3.7.1).

```
float nice_numbers[15]={3.14,1024,333};
```

Definujeme pole 15 čísel typu `float`, z nichž první tři mají přiřazeny uvedené hodnoty.

3.3.2.1 Používání pole

Přístup k jednotlivým prvkům pole je přes index, čili

```
gar_field[10];
```

Je jedenáctý prvek pole *gar_field*.

Ne, nespletl jsem se, opravdu jedenáctý, poněvadž první prvek pole má vždy index 0. V tom se dělají obrovské chyby, ty si ale nechme na později.

S takto označeným prvkem se operuje přesně jako s normální proměnnou typu `int`, ona to ve skutečnosti taková proměnná přímo je. Můžeme z ní číst, přiřazovat jí hodnotu, prostě normální proměnná.

3.3.3 Struktury

Struktura se skládá z několika kousků – lépe řečeno z několika položek ne nutně stejného datového typu.

Představme si ji jako sumář vlastností nějakého objektu reálného světa. Mějme například za úkol vytvořit nějakou kartotéku zaměstnanců (velice jednoduchou). Nejprve se podíváme na jednoho zaměstnance a rozhodneme se u něj vést jméno, příjmení, rodné číslo, adresu bydliště a plat.

Uděláme tedy odpovídající strukturu vystihující jednoho zaměstnance a potom deklarujeme pole těchto struktur – čili pole zaměstnanců – a máme jednoduchou kartotéku hotovou.

[17] Neuvádíme všechny hodnoty ve složených závorkách.

```
struct zamestnanec {
   char jmeno[40];
   char prijmeni[60];
   char adresa[150];
   float plat;
};
struct zamestnanec kartoteka[500];
```

Čímž jsme si definovali kartotéku, která pojme 500 zaměstnanců[18].

Už máme takový obecný přehled co to ta struktura vlastně je, pojďme se podrobněji podívat na její zápis a možnosti.

Obecněji by se její syntaxe dala vyjádřit nějak takto:

```
struct [jmenovka] {
   typ_1 polozka_1;
   ...
   typ_n polozka_n;
} [promenne];
```

Konkrétněji, co je k čemu a kdy se to použije:

Přímý výpis jmen proměnných (poslední položka) použijeme tehdy, když chceme rovnou definovat nějaké proměnné.

S tím má souvislost jmenovka struktury. Pokud ji uvedeme, budeme schopni tuto definici struktury uplatnit ještě dále v programu (viz předchozí příklad), pokud ji nepoužijeme a budeme chtít definovat další instance takové struktury, budeme muset její definici vypsat znovu – a to je k ničemu, protože pokud pak budeme chtít udělat nějakou změnu, budeme muset pamatovat na to, abychom ji dělali na dvou místech. V podstatě nemá smysl definovat strukturu bez uvedením alespoň jednoho – jmenovky anebo jména proměnné.

Definice jedné položky je naprosto stejná jako definice jakékoliv proměnné.

Pro upřesnění, kdybychom chtěli přepsat předchozí příklad tak, abychom nemuseli použít u struktury jmenovku, udělali bychom to takto:

```
struct {
   char jmeno[40];
   char prijmeni[60];
   char adresa[150];
   float plat;
} kartoteka[500];
```

Pokud bychom chtěli být úplní, existuje ještě další způsob definování struktury – můžeme si definovat vlastní datový typ, nebo používat struktury přes ukazatele, ale o tom později (5.2.1).

Nyní si ještě povíme jak strukturu přímo inicializovat a jak se s ní pracuje. Předpokládejme, že máme definovanou strukturu se jmenovkou zamestnanec, dále si chceme inicializovat konkrétního zaměstnance přímo v kódu programu:

[18]Způsob definování je velice nevhodný, ale o tom až budeme probírat makra – v 4.4.

```
struct zamestnanec jenda= {“Jenda”,
      “Janků”, “Stodolní 650”, 15800};
```

Pokud chceme inicializovat pole zaměstnanců:

```
struct zamestnanec kartoteka[500]={
  {“Jenda”, “Janků”, “Stodolní 650”, 15800}
  {“Petr”, “Novák”, “Stodolní 651”, 15900} };
```

Jak jste si jistě všimli, platí stejná pravidla jako při inicializaci pole.

Jak se se strukturou pracuje?

Se strukturou se dá operovat jako s celým objektem, lépe řečeno, pokud máme dvě struktury stejného typu, můžeme jedné přiřadit obsah druhé prostým přiřazovacím znaménkem.

Pro přístup k položce struktury použijeme operátor tečka.

Tedy položku “jmeno” naší struktury “zamestnanec” – konkrétně pole “kartoteka” záznamu druhého, vyplníme takto:

```
kartoteka[1].jmeno=“Johnny”;
```

Pokud pracujeme s ukazateli na struktury (což je velice používáno), máme k dispozici operátor `->`, kterým se na prvek struktury odkazujeme přímo z ukazatele, aniž bychom jej potřebovali dereferencovat, tedy po

```
struct zamestnanec *ukaz_jenda=&jenda;
```

ukazují

```
jenda.plat;
ukaz_jenda->plat;
(*ukaz_jenda).plat;
```

na stejné místo v paměti, všechny přitom se správným typem (`float`). Pokud tomu nerozumíte, nic si z toho nedělejte, vysvětlíme si to podrobněji u operací s ukazateli (viz 3.5.1).

Cvičení: Nadefinujte nějak rozumně strukturu pro osobu (obecně). Poté udělejte další dvě struktury, a to učitel a žák, v každé z nich použijte jako položku strukturu pro osobu[19]. Dále alokujte pole učitelů a pole žáků. Konečně dejte možnost uživateli programu upravovat seznam učitelů a seznam žáků. Mazání realizujte tak, že se položky záznamu nastaví na nějakou implicitní hodnotu.

Implementujte “setřesení” seznamu – tzn. pokud bude např. prvek s indexem 4 volný, všechny s indexem vyšším se v poli posunou o 1 dolů.

Poznámka: U učitele realizujte minimálně plat a posudek (řetězec o pevné délce), u žáka zase studijní průměr a třídu (taktéž řetězec o pevné délce). V příkladu tedy nebudete potřebovat žádnou dynamickou alokaci. V každém případě použijte v programu zanořené struktury[20].

Pomoc: k položce struktury která je schovaná v další struktuře, se přistupuje normálně, jen přes dvě tečky, např. `polezak[45].osob_udaj.jmeno`.

[19] Některé vlastnosti mají učitel a žák stejné – např. oba mají jméno – a (nejen) to bude uloženo v osobě.

[20] Ve strukturách žák a učitel musí být položka, která je typu struktury osoby.

3.3.4 Uniony

Dalším ze složitějších typů je union, někdy také nazýván unie. Union má sice definovaných n položek, ve skutečnosti však obsahuje pouze právě jednu z nich.

Uvedené položky představují varianty, můžeme tedy union definovat jako nějakou variantní datovou strukturu, která nabývá jednoho z možných typů. Co se velikosti paměti týče, alokuje se paměť potřebná pro největší možnou položku, bát se tedy nemusíme.

Definice vypadá takto:

```
union [jmenovka] {
   typ_1 polozka_1;
   ...
   typ_n polozka_n;
} [promenne];
```

Pravidla jmenovky jsou stejná jako u struktury[21]

Nějaký konkrétní příklad:

```
union varianta {
   int i;
   float f;
   char s[20];
};
union varianta data1;
```

Při přiřazování hodnot pochopitelně musíme říci, jaký typ do unionu ukládáme, což učiníme specifikací položky, například:

```
data1.i=10;
```

Od této chvíle obsahuje data1 číslo 10, uchované jako `int`.

```
data1.s="Hell-o world";
```

Nyní obsahuje uvedený řetězec, hodnota 10 byla přemazána.

Ještě to zde nebylo uvedeno – položky se v paměti překrývají, sdílí jednu paměť.

Pokud chceme navíc nějakou rozhodovací proměnnou, která by nám řekla, jaký typ se v unii momentálně vyskytuje, vložíme unii do struktury:

```
struct var_data {
   enum {CELE, REALNE, RETEZ} rozhodni;
   union varianta var;
};
struct var_data data;
```

Poté můžeme napsat kousek kódu takto:

[21] Pokud ji uvedeme, můžeme v průběhu programu tvořit další unie takto: `union jmenovka proměnná;`

```
switch (data.rozhodni) {
   CELE: data.var.i=1; break;
   REALNE: data.var.f=1.0; break;
   RETEZ: data.var.r=”jedna”;
}
```

Prvkem unie nemůže být další unie stejného typu [22] .

Cvičení: Udělejte program s funkcí stejnou jako u minulého cvičení (klidně kód okopírujte), s tím, že budete mít jedno pole, jehož prvkem bude struktura "clovek", která bude mít rozhodovací položku (jestli je v ní uložen učitel nebo žák) a union, který bude buďto učitel, nebo žák. Samozřejmě budete muset trošku upravit funkce pro manipulaci se seznamem (polem) – už jen proto, že bude jen jedno pole.

3.4 Přetypování

Již několikrát byl v tomto textu použit pojem přetypování[23], zde si jej konečně vysvětlíme. Již jsme si objasnili, že každá proměnná se ukládá do nějaké oblasti paměti a je určitého typu – přesněji, data jsou interpretována určitým způsobem. Například pokud interpretujeme 32 jako `int`, nic se nezmění, pokud to má ale být `char`, značí znak mezera.

Přetypování slouží pro změnu interpretace dat, tedy pro změnu typu předávaných dat. Nemění se přímo typ proměnné, která jej již má někde uveden, ale je možno přetypovat její data, která se někam předávají. Například se ukládají do jiné proměnné, která je právě toho typu, na který přetypováváme.

Ve skutečnosti je jedním z nejčastějších případ, kdy potřebujeme obsah proměnné uložit do proměnné jiného typu a proto musíme použít přetypování. Toto jsou příklady tzv. explicitního přetypování, jinými slovy, přetypování, které jsme si sami vynutili.

Co to vlastně znamená změna typu? Je třeba mít na paměti, že se mění způsob interpretace dat a původní hodnota je konvertována do nového typu, čímž se může kompletně změnit!

3.4.1 Implicitní

Tento typ přetypování je uplatňován automaticky, je to vlastně přetypování, které při splnění určitých podmínek (čtení ze souboru do typu `int`) proběhne vždy, bez našeho přičinění. Ono vůbec záměna mezi `char` a `int` se moc specifikovat nemusí, protože samy znaky jsou reprezentovány typem `int`.

Co a jak se provádí:

1. Cokoliv typu `char` a `short int` je přetypováno na typ `int` ještě dříve než se s tím vůbec začne pracovat!

2. `unsigned char` a `unsigned short` se přetypují na `int`, pokud se hodnota do `int` nevejde, jsou přetypovány na `unsigned int`.

[22] Zanořená pouze pojmenováním typu – proč?
[23] Angl. *cast*.

3. V případě vyhodnocování operace, kdy mají oba operandy různý typ, se nižší z nich přetypuje na vyšší (podle následující nerovnice)
   `int=>unsigned int=>long=>unsigned long=>`
   `float=>double=>long double`

Pokud se jedná o přiřazení, překladač se bude vždy snažit přetypovat výsledek pravé strany tak, aby odpovídal typu proměnné, do které se má výsledek uložit. A pokud nebude takové přetypování schopen provést, dostanete *warning* nebo *error*, v závislosti na překladači a nastavení.

3.4.2 Explicitní

Vyskytne se spousta případů, kdy budeme potřebovat přetypování popsat, neboť se samo neprovede, nebo se provede špatně. Nejčastější případ bude asi při předávání parametru funkci, nebo naopak ukládání výsledku funkce.

Přetypování provedeme pomocí obyčejných závorek, mezi něž vložíme typ, takto:

```
(typ) výraz
```

Hodnota "výraz" je konvertována na odpovídající hodnotu v typu "typ".

Příklad:

```
float f = 3.14;
int i = (int) f;
```

Do i je uložena hodnota 3. Přetypování z reálného na celé číslo znamená oříznutí desetinné části. Někdy se explicitní přetypování uvádí i tehdy, když se má provést přetypování implicitní – kvůli přehlednosti. Tím, že takové přetypování uvedeme, zdůrazníme, že jsme to měli v úmyslu, že nejde o věc, se kterou jsme nepočítali.

3.4.3 Chyby

Mezi lehčí chyby, které jsou odhaleny už překladačem, patří například přetypovávání nikoliv výsledku výrazu, ale proměnné – nejčastěji jako:

```
(int) c = d;
```

Což je špatně, neboť se nepřetypovává předávaná hodnota, ale proměnná c, která je někde výše (ve své definici) vázána jiným typem[24].

Daleko horší je chyba, která není odhalena překladačem, ale dělá neplechu až za běhu programu, protože je pak značně obtížnější ji hledat.

Při přetypovávání je třeba mít se na pozoru a dobře uvažovat, co že vlastně děláme. Třeba přetypování čísla na řetězec asi není moc dobrý nápad, protože řetězec bývá ukončen znakem `\0`.

[24]Kdo je zmaten: při `c=(int) d;` se nepřetypuje proměnná d, nýbrž se přečtená hodnota konvertuje na typ `int` – čili typ ani jedné z obou proměnných není změněn.

Pokud takovou hodnotu chceme uložit do nějaké proměnné typu řetězce, může to dopadnout všelijak.

Musíme jednat opatrně i při přetypování různých složitějších, například vlastních typů.

Ještě jedna poznámka: Pokud vám překladač vyhodí varování, že nesedí typ proměnných při přiřazování, vždy se na příslušný řádek podívejte a dobře uvažte, zda nepoužijete přetypování. Těžko radit obecně, ale ve většině případů má překladač pravdu a je lepší přetypovat, i když na druhou stranu existují výjimky, kdy bude takové varování nesmyslné – záleží na schopnostech překladače.

K přetypovávání se budeme ještě průběžně vracet.

3.5 Alokace paměti

Je načase, abychom si pověděli něco o alokaci paměti.

Říkali jsme si, že adresa proměnné je vlastně ukazatel na tuto proměnnou. Tak teď si pojem ukazatel trochu předefinujeme, aby lépe odpovídal (Cčkové) realitě.

Pojďme na to.

Ukazatel na proměnnou A je jakási proměnná, jejímž obsahem je adresa proměnné A.

A je to venku. Vše důležité bylo právě řečeno. Snad ještě to, že nemusíme mít ukazatele jen na proměnné, ukazatel může ukazovat kamkoliv do paměti. To si ale ještě v této kapitole rozebereme.

Teď k přidělování paměti.

Jazyk C zná tři typy přidělování paměti. Statické, automatické a dynamické.

Statické

Do této kategorie patří například všechny globální proměnné a vůbec cokoliv, pro co jsme paměť definovali na začátku programu. Samozřejmě, taková paměť se alokuje (přiděluje) při startu programu a uvolňuje až při jeho ukončení.

Automatické

Takzvaně automaticky přidělována je paměť proměnným a vůbec datovým objektům, které samovolně vznikají a zanikají za běhu programu, typicky proměnným definovaným v nějaké funkci (*lokální* proměnné), nebo parametrům funkce. Automatická alokace přiděluje paměť na zásobníku.

Takto alokovaná paměť je uvolněna po skončení příslušné funkce.

Dynamické

O tomto druhu se budeme bavit po celý zbytek kapitoly.

K čemu dynamická alokace slouží a co to vlastně je?

Dynamická alokace znamená alokaci paměti “za běhu” programu. Navíc, od alokace automatické, ji můžeme velice dobře ovlivňovat – např. naalokovat si kolik paměti chceme a kdy chceme.

A k čemu to použijeme?

Jednoduchým příkladem je práce s větším množstvím dat, například s textovými soubory. Potřebujeme načíst soubor do paměti, nejlépe tak, abychom byli schopni adresovat každý řádek zvlášť, neboť potom s obsahem textu chceme pracovat. Čili chceme mít každý řádek jako zvláštní řetězec. Jsme schopni dopředu říci, kolik řetězců budeme potřebovat, potažmo kolik paměti budeme potřebovat? Za předpokladu, že píšeme program, který bude pracovat nad více než jedním neměnným a předem známým textem, zcela jistě to říci neumíme.

Horu dalších příkladů si jistě vymyslíte sami. V podstatě je jakákoliv práce s předem neznámými daty prováděna za pomoci dynamické alokace.

Dynamicky alokovaná paměť se nám totiž po skončení funkce neztrácí, poněvadž není alokovaná na zásobníku, ale někde jinde v operační paměti. Dealokuje se sama po skončení programu, resp. funkce `main()`.

3.5.1 Ukazatele

Zpátky k ukazatelům. Ukazatel je tedy proměnná, která obsahuje adresu něčeho v paměti.

Pokud bychom měli jen ukazatel a žádnou další informaci, nebyli bychom schopni určit, na co ukazujeme. Na `int`? Na začátek řetězce? Na nějakou strukturu? Sami vidíte, že to není tak jednoduché. Pokud ukazujeme na objekt, tento objekt má svůj typ. Ukazatel má také typ, resp. je ukazatelem na typ.

Huh? Nějaký příklad:

```
int *uk_int;
```

Je definice ukazatele na (nějakou) proměnnou typu `int`.

```
float *uk_flo;
```

Máme ukazatel na `float` atd.

Jakmile si blíže vysvětlíme význam znaku `*`, řekneme si, jak vybrat proměnnou, na kterou máme ukazovat atp.

`*` je operátor, nazývaný dereference ukazatele. To zní hezky, ale co to je? Pokud máme ukazatel A, který ukazuje na objekt B, je vlastně adresa objektu B uložena v obsahu proměnné A. Dereference je operace, která překladači říká, že nemá pracovat s obsahem proměnné A, ale s objektem nacházejícím se na adrese uložené v proměnné A.

Pokud je tedy A ukazatel, a použijeme jej bez hvězdičky, pracujeme s obsahem proměnné, čili s adresou. Pokud ale uvedeme hvězdičku, pracujeme s obsahem objektu, na který ukazatel (obsah proměnné A) ukazuje.

Jestli to chcete méně jasněji ale více názorněji, představte si, že A je šipka (ukazatel), ukazující na objekt B. Pokud chceme pracovat s objektem B, použijeme `*`, díky níž provedeme dereferenci – pocestujeme po šipce a budeme pracovat s tím, co na jejím konci najdeme. Když ale chceme manipulovat šipkou, napíšeme pouze A.

Nyní lehce pochopíme definici ukazatele. Podívejme se řádně na definici `uk_int`, a vidíme že říká: “Dereferencováním `uk_int` dostaneme typ `int`” neboli “Obsahem proměnné `uk_int` je adresa objektu typu `int`”, nebo jednodušeji “Tam, kde ukazuje `uk_int`, je uložen `int`”.

Zatím ještě dynamickou alokaci nezvládáme, podívejme se tedy, co dovedeme s pointery a statickými proměnnými[25].

[25] Obecně jakýmikoliv objekty, už to ale opakuji naposledy.

Mějme definici

```
int a=5;
```

Čímž se staticky[26] naalokuje paměť potřebná pro typ `int` a uloží se do ní obsah (číslo) 5.

Adresu nějakého objektu zjistíme pomocí operátoru &.

Čili že `&a` nám dá adresu objektu *a*.

Dále zápisem

```
int *uk_int;
```

Vyrobíme ukazatel na objekt typu `int` a přiřazením

```
uk_int=&a;
```

mu přikážeme aby ukazoval na objekt *a*.

Ještě naposledy zopakuji, že `uk_int` je proměnná obsahující adresu a `*uk_int` je už samotný objekt[27].

Z výše uvedeného vyplývá, že

```
printf("%d",a);
a printf("%d",*uk_int);
```

Nám vytiskne totéž, neboť v obou případech voláme funkci `printf()` se stejnými parametry. Pokud bychom ovšem napsali

```
printf("%d",uk_int);,
```

pokusili bychom se vytisknout pointer (adresu) jako decimální číslo[28].

Definujeme si b:

```
int b;
```

Nyní

```
uk_int=&b;
```

ukazuje `uk_int` na b;

`uk_int=&a;` nyní opět a.

`uk_int=&b;` nyní opět na b.

`uk_int=`... (nebuďme jak děti)

Pokud budeme chtít přiřadit do *a* hodnotu 5, můžeme to udělat následovně:

```
a=5;
```

[26]Nebo automaticky, pokud jsme jinde než v `main()`.
[27]Přesněji jeho obsah, nyní tedy `int`, hodnota 5.
[28]Vyzkoušejte si co dostanete.

nebo také

```
uk_int=&a; *uk_int=5;
```

Překvapeni? Doufám že ne, protože se celou dobu snažím vysvětlit, že pokud máme ukazatel A, ukazující na objekt B, zápis *A je ekvivalentní zápisu B[29].

Cvičení: Napište funkci, která jako své parametry vezme ukazatele na dvě proměnné typu `int` a na výstup vydá jejich součet (např. pro *3 a *5 ukáže "3+5=8")

3.5.2 NULL

Pro práci s pointery[30] existuje makro `NULL`, které symbolizuje ukazatel nikam – no dobře, ne nikam, ale na nic existujícího, viz GNU C dokumentace:

> ...je garantováno, že pointerová konstanta *null* neukazuje na žádný reálný objekt...

K čemu to?

Předně si musíme uvědomit, že definicí ukazatele

```
int *ukaz;
```

Je definována statická proměnná *ukaz*, jejímž obsahem je adresa na objekt typu `int`.

A kdo vám zaručí, že obsahem paměti, která bude této proměnné přidělena, nebude nějaká hodnota (třeba `0xff` v případě 2-byte ukazatele)?

Někde v programu potom – samozřejmě chybně – čteme z tohoto pointeru a to předtím, než jsme mu nějakou adresu přiřadili. A bác ho – on někam ukazuje! A na daném místě v paměti nějaká data opravdu jsou! Takže opravdu cosi přečteme. A potom koukáme, kde že to ten program pořád ta stejná data bere, vždyť tam zadávám pořád něco jiného!

To je ještě v pohodě, větší sranda je, když na adresu zapisujeme. Pokud program nespadne, stane se pravděpodobně něco horšího[31].

Výše popsaná chyba je velice častá a proto můžeme neinicializované ukazatele inicializovat hodnotou `NULL`, a to například takto:

```
int *ukaz=NULL;
```

Čtení a psaní z a na tuto adresu opět způsobí chybu, ale snadněji zjistitelnou – viz cvičení.

Tato konstanta se dále používá například při práci s dynamickými seznamy, vrací ji v případě neúspěchu spousta funkcí[32] atp.

Vnímavější z vás možná přišli na následující problém:

"Jak můžeme přiřazovat jakémukoliv ukazateli (ukazateli na jakýkoliv typ) ukazatel `NULL`, vždyť ten musí také ukazovat na nějaký typ, a neprovádět přitom žádné přetypování?"

Odpověď: `NULL` je ukazatel na `void`, přesněji

---

[29] Pro korektnost, (*A) - hvězdička má nízkou prioritu, závorkami ji zvýšíme.

[30] Omlouvám se za guláš, ale slova pointer a ukazatel volně zaměňuji. Nechci vás tím zmást, ale naopak naučit vás zvyknout si na oba pojmy, které (naposledy!!!) znamenají totéž.

[31] Nezapracovala ochrana paměti, my přepsali nějakou systémovou rutinku a hups – kde jen mám ten *reset*?

[32] Například funkce pro alokaci paměti – malloc() – viz 3.5.5.

```
void * NULL Macro
```

a samotné makro

```
#define NULL ((void *)0)
```

v souboru `stddef.h`

Co je ale podstatnější, ukazatel na `void` se automaticky přetypuje na potřebný typ, naopak, pokud například porovnáváme ukazatele, přetypujeme je na `void *`. Adresa se tím samozřejmě vůbec nezmění.

Cvičení: Zjistěte co způsobí čtení a psaní z/na adresu `NULL`.

3.5.3 `sizeof()`

`sizeof()` je jedním z mála operátorů, který nám ještě zbývá probrat. Velice úzce souvisí s alokací paměti, neboť zjišťuje velikost svého operandu a výsledek pak použijeme při alokování paměti.

Například

```
sizeof(int);
```

vrátí velikost typu `int`.

Návratová hodnota je typu `size_t`, který je celočíselný typ bez znaménka, buďto `int` nebo `long` v závislosti na implementaci. Spíše `long`.

Je doporučeno uchovávat velikost objektů v proměnných typu `size_t`, nikoliv `int`[33].

Cvičení: Zjistěte velikosti všech základních typů (včetně ukazatele) na vašem SW&HW.

3.5.4 Operace s ukazateli

Pojďme se vrhnout na věc, která se všude nazývá pointerová aritmetika.

S pointery je možno kromě přiřazování a alokování provádět ještě další kousky.

Definujme si například pole a dva pointery:

```
int pole[200];
int *uk1=&pole[50];
```

Ukazatel na 51. prvek pole

```
int *uk2=&pole[110];
```

Ukazatel na 111. prvek pole

Výrazem uk1+3 dostaneme ukazatel na 54. prvek. Z uvedeného vyplývá, že přičtením celého čísla dostaneme ukazatel na příslušný následující prvek. Nepřičítáme ovšem *byte*, ale velikost celého objektu (v tomto případě `int`), čili v případě pole například struktur bychom se posunuli taktéž o tři prvky dále.

Odečítání funguje analogicky, můžeme tedy lehce dosáhnout na nějaký z předchozích prvků.

[33] Zkuste uhodnout proč.

Je třeba dávat pozor, abyste náhodou "nevyletěli za pole". Pokud překladači řeknete, že má přičíst 80 prvků, on to klidně udělá. To, že máte pole o velikosti 20 prvků, je váš problém.

Pokud ukazují dva ukazatele do stejné datové struktury (např. do pole), můžeme jejich odečtením získat počet prvků mezi nimi, např. uk1-uk2 (vyzkoušejte si).

Pro porovnávání pointerů slouží klasické `==`, `!=`, `<`, `<=` ,`>` ,`>=`.

Pointer je větší než pointer ukazující na prvek ve stejné datové struktuře, ovšem s menším indexem, například platí `uk2>uk1`.

Pro porovnávání adres pointerů je nejdříve přetypujeme na `void *`. Pokud tedy chceme porovnávat dva pointery, tak napíšeme

```
if ( (void *) uk_1 == (void *) uk_2 ) příkaz;
```

Cvičení: Naprogramujte jednoduchý prográmek umožňující uživateli měnit položky pole (čísla) a pohybovat se v poli – o jedno políčko dále nebo zpět. Pohyb a měnění v poli realizujete přes jeden ukazatel, neustále do pole ukazující – posunujte jej pomocí ± 1.

3.5.5 `malloc()` a `free()`

Dostáváme se k nejpodstatnější části kapitolky, to jest samotná alokace a dealokace paměti.

Pokud chceme alokovat paměť, použijeme v drtivé většině případů funkci `malloc()`, formálně

```
void * malloc (size_t size);
```

Parametrem je velikost alokované paměti v *byte*.

Jak je vidět, funkce vrací ukazatel na `void`, což je v případě úspěchu adresa ukazující na naalokované místo, v případě neúspěchu to je `NULL`.

Pravděpodobně budeme používat ukazatele na nějaký typ, ne na `void`, tudíž si musíme hodnotu přetypovat, např

```
#define BUFFER 500
char *uk_retez;
uk_retez = (char *) malloc(sizeof(char)*BUFFER);
```

Čímž jsme dynamicky alokovali řetězec o délce BUFFER znaků. *CHYBA!* Kdo si jí všimnul? Kdepak vážení, nemáme řetězec o délce BUFFER znaků, vůbec ne, to by se nám mohlo pěkně vymstít. Spadnutí programu by opět nemuselo být tím nejhorším.

Nevíte? Tak se pěkně zapoťte než vám to prozradím.

Problém je v tom, že řetězec v C je ukončen znakem `\0`, tedy máme kapacitu BUFFER-1 znaků a když na to zapomeneme, budiž nám země lehká.

Na alokaci řetězce o délce BUFFER znaků použijeme následující fintu:

```
uk_retez = (char *) malloc(sizeof(char)*(BUFFER+1));
```

Pokud jsme navíc paranoidní a chceme se vyvarovat potenciálních budoucích překlepů – t.j. chceme ošetřit i případy, že někdy příště zapomeneme řetězec ukončit nulou, dáme ji na konec hned.

Tím ovšem moc nezískáme – program nám při takové chybě nespadne, neboť na konci řetězce budeme mít “pojistnou” nulu, ale při čtení řetězce přečteme až do této nuly a budeme v něm mít pravděpodobně nesmyslné znaky (pokud jsme je dříve nesmazali).

Kromě funkce `malloc()` máme například i `calloc()`, která pracuje podobně, jen s tím, že zadaný prostor bude vyplněn nulami. Formálně:

```
void * calloc(size_t count, size_t eltsize)
```

Kde *count* je počet prvků, které chceme alokovat a *eltsize* velikost jednoho elementu. Správně tušíte, že v tomto případu si žádnou “pojistku” ukládat nemusíme, protože máme vlastně každý znak tvaru `\0`.

Pokud chceme nějakou datovou strukturu přealokovat (změnit její velikost), použijeme

```
void * realloc(void *ptr, size_t newsize);
```

kde *ptr* je ukazatel na objekt, který chceme přealokovat a *newsize* je nová velikost objektu. Při neúspěchu se vrátí `NULL`, při úspěchu ukazatel na objekt – pokud při zvětšování objektu není “za” objektem dostatek místa, může se vynutit přesun celého objektu někam jinam. Funkce `realloc()` je definována v hlavičkovém souboru `stdlib.h,` který nesmíme zapomenout zahrnout, pokud ji chceme používat.

Poznámka:

Už bylo řečeno, že `malloc()` vrací ukazatel na `void.` Co se ale stane v případě, že se paměť nepovede naalokovat? `malloc()` vrátí `NULL.` Tyto chyby je třeba ošetřit a dát uživateli na vědomí, co že se stalo. Jednoduché řešení je uvedeno v referenční příručce GNU C Library – napsat si vlastní funkci `xmalloc()` a poté místo `malloc()` volat jen ji. Ve funkci `xmalloc()` rovnou chybový případ vyřešíme, nebo zavoláme nějakou jinou funkci, která si s tím poradí (vypíše hlášku a skončí program):

```
void *xmalloc(size_t size)
{
    register void *value = malloc (size);
    if (value == 0) fatal("virtual memory exhausted");
    return value;
}
```

Alokovat už umíme, nyní jak paměť zase uvolnit.

Pokud máme nějaký paměťově náročný program, který neustále alokuje další a další paměť a neuvolňuje tu, co nepotřebuje, mohli bychom se brzy dostat do stavu, kdy paměť dojde. Například

```
int *uk_int;
uk_int = (int *) malloc(sizeof(int));

/* později v programu int nepotřebujeme
 * ale potřebujeme jich 10 a tak
```

```
 */
uk_int = (int *) malloc(sizeof(int)*10);

/* aniž bychom původně naalokovanou paměť
 * nejprve vrátili
 */
```

V tomto příkladě se jednalo jen o jeden `int`, ale jindy může jít o víc.

Správně bychom měli ten zapomenutý `int` dealokovat pomocí funkce `free()`:

```
void free(void *ptr);
```

Funkce uvolní operační paměť na kterou ukazuje *ptr*. Paměť by měla být dříve naalokována pomocí `malloc()`, protože jinak není definováno, co `free()` udělá[34].

Nemá smysl uvolňovat všechny dynamicky alokované bloky paměti před ukončením programu, protože po skončení bude paměť stejně vrácena operačnímu systému.

Ještě co se uvolňování paměti týče – při použití `free()` se paměť sice uvolní, ale není vrácena OS, takže ji nemohou používat jiné programy.

Je možno ji naalokovat pouze ve stejném programu pomocí další `malloc()`. Problém je trošku složitější, řekněme že ve způsobu alokace pomocí `malloc()`.

Pokud chcete použít velké množství paměti a poté jej vrátit OS, použijte `mmap()`[35]. Ale abych moc `malloc()` nepomlouval, v GNU C sama při překročení určitého množství paměti (když chce program velké sousto), použije `mmap()` a vše je jak má být.

Chování `malloc()` lze samozřejmě ovlivnit a to pomocí `mallopt()` – viz manuálová stránka a dokumentace. `malloc()` dokonce provádí nějakou optimalizaci a vyhledávání paměťových děr atp. ale ty a další věci (*garbage collector*), jsou opravdu hodně mimo rámec kurzu. Viz dokumentace ke GNU C.

Poznámka:

Jak si někteří již uvědomili, používání dynamických struktur je podstatně pomalejší než používání statických, ale nese sebou mnoho možností, které statickými datovými strukturami nenahradíme ani kdybychom se na hlavu postavili. Krátce a stručně, je to skvělá věc a je nutno ji ovládat.

Tak ještě jedna *poznámka*. Zapomněl jsem zmínit funkci `alloca()`, ale jen krátce, poněvadž ve standardu ANSI jsem ji nějak nenašel, takže to bude asi zvláštnost UNIXů a GNU[36].

```
void *alloca(size_t size);
```

Funkce `alloca()` vrátí ukazatel na alokovaný prostor, podobně jako `malloc()`. Podstatný rozdíl ovšem je, že paměť se alokuje na zásobníku volajícího a to znamená, že po skončení funkce, ve které je paměť alokována, je také automaticky uvolněna[37] – uvolní se při provádění

[34] Stejně tak při volání `free()`, pokud již bylo `free()` voláno na stejném objektu dříve.
[35] Týka se pouze GNU OS a UNIXů.
[36] Není tak docela pravda. Například v MSVC++ 6.0 je funkce `_alloca()` která funguje jako `alloca()`
[37] Nemusíme použít `free()`.

`return`. Uvolní se také při provádění `longjmp()`, což u paměti alokované pomocí `malloc()` nemůžeme čekat, ta by v takovém případě zůstala alokovaná až do konce programu.

Navíc je `alloca()` podstatně rychlejší než `malloc()`. Její nevýhodou je ale případ, kdy se paměti nedostává – nevrátí totiž chybu, ale pravděpodobně způsobí nějaký fatální signál, nejspíše SIGSEGV[38]. Více viz GNU dokumentace.

Cvičení: Projděte si dříve naprogramované prográmky a jejich používání `malloc()`.

3.6 Dynamické pole a struktury

Alokovat už umíme, nyní se podíváme na způsob práce s dynamicky naalokovanými datovými typy.

Nejprve pole.

Na prvky pole se odkazujeme například

```
pole[0], pole[150], pole[x-y],
*uk_pole[MAX], *uk_pole[10]
```
[39]

a tak podobně, ale co když napíšeme prostě

```
pole;
```

Co to je? Ok, podělím se s vámi o malé tajemství, je to ukazatel na pole, konkrétně na jeho začátek – na první prvek.

Zkusíme se nad tímto skandálním odhalením trošičku zamyslet.

Pokud `pole` je pointer, a `pole[1]` jeho druhý prvek, způsobují hranaté závorky dereferenci pointeru, nebo ne?

Samozřejmě ano, protože `pole[1]` už pointer není, je to vlastně `*(pole + 1)`, pokud se pořádně podíváte na předchozí kapitolku.

Zkuste nad tím přemýšlet. V těch několika větách a znalosti operací s ukazateli se ukrývá všechno co se polí, souvislých bloků paměti stejných elementů, týče, včetně vztahu k ukazatelům. Za chvilku přijdete na to, že je to všechno naprosto jednoduché, logické a elegantní a pole v C se pro vás stanou téměř přirozeným prostředkem nad jehož používáním nebudete muset nijak dumat.

Ještě jedna poznámka k přemýšlení, `uk_pole` – což je ukazatel na `pole` – je vlastně ukazatel na ukazatel, je to tak? Jeho dereferencováním

```
*uk_pole
```

dostaneme `pole`, tedy stále ukazatel a

```
*uk_pole[10]
```

provede dereferencování na ukazatel na `pole` a poté ještě dereferencování ukazatele `pole` a přičtení příslušného počtu prvků, je to tedy totéž co

```
*(*uk_pole + 10),
```

nebo ne? (vyzkoušejte si). Tento poznatek – ukazatel na ukazatele – se nám bude hodit dále.

Cvičení: Těchto několik odstavců si pročtěte ještě jednou a pozorně.

[38] *segmentation violation*

[39] *uk_pole* je ukazatel na pole.

3.6.1 Dynamické pole

Nejprve musím zdůraznit, že pokud chápete pointerovou aritmetiku a pointery obecně, nebudete mít s dynamickými poli žádné problémy, protože práce s nimi je naprosto stejná jako s poli statickými.

Dobře, a proč o tom vlastně mluvím?

Je nutno si uvědomit malý rozdíl mezi statickým a dynamickým polem. Pokud definujeme statické pole

```
int pole[20];
```

Dostaneme okamžitě naalokovanou datovou strukturu o potřebné délce, pokud ovšem chceme definovat pole dynamicky

```
int *uk_pole;
```

Nemáme naalokováno nic, jen ukazatel na pole, pro alokování obsahu pole musíme použít `malloc()` či podobnou funkci, nebo prostě provést přiřazení

```
uk_pole=pole;
```

Z toho vyplývá, že pokud máme pole statické, nemůžeme měnit jeho místo v paměti, neboť pole je neměnitelný ukazatel – paměť byla alokována při startu programu.

A tedy ani dost dobře nemůžeme měnit velikost pole (funkcí `realloc()`), pouze tak, že pole překopírujeme do pole většího. Jenže tím bychom nic nezískali, neboť paměť pole by byla uvolněna až po konci programu. Pokud tedy předpokládáme, že budeme měnit velikosti pole, pracujeme s poli dynamicky alokovanými. Je to obecně lepší, protože jsme schopni ošetřit případy, kdy se pole nedokáže naalokovat, měnit velikosti pole a uvolňovat paměť.

Ještě je třeba pamatovat na zjišťování velikostí statického a dynamického pole. Pokud zjišťujeme pomocí `sizeof()` velikost statického pole, dostaneme správnou hodnotu – dle definice pole. Pokud ovšem zjišťujeme velikost dynamického pole, dostaneme velikost pointeru na pole, což jsme nechtěli. Proto si budeme vždy velikost pamatovat v nějakém makru (viz kapitola 4.4), případně proměnné.

3.6.2 Vícerozměrná pole

Někdy se dostaneme do situace, kdy budeme potřebovat vícerozměrné pole. Bude například potřeba reprezentovat matice, tří dimenzionální prostory, mapy vrstev nebo obecně modely n-dimenzionálního prostoru. Nebo budeme potřebovat dělat "zubaté" dvojrozměrné pole – takový typ křížovky, nebo už nevím co, prostě budeme potřebovat vícerozměrné pole.

Fajn. Co to je? Pole, které jsme dosud probírali, bylo pole jednorozměrné. Dá se představit jako několik prvků, postavených v řadě za sebou, s jistou formou abstrakce bychom to mohli považovat za přímku, respektive úsečku.

Dvourozměrné pole si můžeme představit jako plochu, jako síť elementů, čili spoustu řádků naskládaných pod sebe. Opět, s jistou formou abstrakce, jako rovinu.

Třírozměrné pole si zase představíme složením mnoha dvojrozměrných polí vedle sebe, konkrétně v lidské interpretaci nad sebe, čímž nám vznikne normální třírozměrný prostor.

Čtyřrozměrné pole je spousta polí třírozměrných, poskládaných vedle sebe. Už nám asi chybí směr, zkuste si to představit jako třírozměrné prostory uskládané za sebou[40].

Mohli bychom pokračovat dál a dále, ale myslím, že už velice dobře víme, co to vícerozměrné pole je.

Takže, zbývá nám probrat, jak se takové pole tvoří a jak se s ním pracuje v jazyce C.

Na začátek mám příjemnou zprávu. Vše co jsme se z pointerové logiky naučili, nadále platí a vše se řídí intuitivními pravidly. Pro ty, kteří pochopili výše uvedené poznatky o ukazatelích a o polích, bude následující přirozeným rozšířením, které pochopí velice lehce a budou používat s přirozenou elegancí.

Pojďme od toho nejjednoduššího – aneb vícerozměrné statické pole.

```
int pole_2_dim[5][10];
```

A máme dvojrozměrné pole 5x10 elementů typu `int`.

Hned na začátku musíme rázně vyřešit problém, který z rozměrů je větší, důležitější, atp.

První rozměr je tím určujícím, jak se dalo čekat. Neboli, druhý rozměr roste rychleji.

Zmateni? Ale ne, když čtete stránku knihy, větším, určujícím rozměrem jsou řádky a přitom jsou to znaky, kde index roste rychleji (elementem je znak) – jakmile se podřízený rozměr, znaky v řádce, dostane do svého maximálního indexu, teprve poté se zvětší index určující řádku, což zároveň vynuluje index u znaků.

Například knihu jako takovou můžeme chápat jako třírozměrné pole:

```
char kniha[str][rad][znak];
```

nebo klidně jako čtyřrozměrné, pokud přidáme kapitolu[41], knihovnu potom jako pětirozměrný prostor znaků[42] a knihovny na celém světě jako šestirozměrný prostor znaků, "svět knih":

```
char svetknih[knihovna][kniha][kap][str][rad][znak];
```

Definovali jsme si pole, pomocí kterého jsme schopni jednoznačně adresovat jakýkoliv znak v jakékoliv knize v kterékoliv knihovně světa[43], no, řekněte, není to hezké?

Statické pole je samozřejmě alokováno při startu programu[44] a je v paměti uloženo jako souvislý blok.

Vraťme se zpět k `pole_2_dim`. Máme tím definovánu například matici o pěti řádcích a deseti sloupcích (celočíselnou).

Třetí prvek na druhém řádku adresujeme jako

```
pole_2_dim[1][2]
```

Nesmíme totiž zapomínat na to, že se indexy začínají od nuly.

Už vás asi napadá, že `pole_2_dim` je vlastně ukazatel na pole. Samozřejmě. A co je `pole_2_dim[1]`? Samozřejmě ukazatel na druhý řádek pole.

[40] Čtvrtým rozměrem může být například čas, opět dostaneme dva směry které vytvoří co potřebujeme – rozměr. V jedné povídce jsem narazil na pěkné pojmy pro popisy směrů v novém rozměru – "dahoru" a "nolů".

[41] Kniha je souhrnem kapitol, ty jsou souhrnem řádek, ty zase znaků.

[42] Knihovna bude souhrnem knih.

[43] Pro vrtaly: jsme schopni rozlišit názvy knihoven v případě shodnosti – prostě je očíslujeme. Případné stejné knihy v knihovně (někdy mají více exemplářů) opět rozlišíme, ať již podle evidenčního čísla nebo jakkoliv jinak.

[44] V případě světa knih to asi na každém HW nerozchodíme, pokud budeme toto pole rovnou inicializovat hodnotami – oj, to by byl velký zdrojový text!

Pojďme si trochu zahrát:

```
int gamesa[4][4]={
   { 4,5,6,7 },
   { 1,2,3,1 },
   { 2,2,2,2 },
   { 7,6,7,8 }
};
```

Máme definováno a inicializováno (a vida, tak takhle se to dělá!) dvourozměrné pole 4x4 prvky. Já teď uvedu několik prvků a vy budete hádat, které to jsou a jestli uvedené vztahy platí.

Nejprve pár triviálních příkladů:

```
gamesa[1][0];
gamesa[0][4];
```

Teď něco těžšího:

```
(void *) gamesa == (void *) gamesa[0] ?
(void *) (gamesa + 1) == (void *) gamesa[1] ?
(void *) (gamesa[0] + 1) == (void *) gamesa[1] ?
gamesa[2]-gamesa[0] = ???
```

A nakonec opravdový záhul – nepovinně:

```
*(gamesa[1]+2)+1==*(&((*(gamesa+3))[2])-14) ?
```

Řešení:

***trivial*:** oba ukazují na první prvek v druhé řádce, tedy výsledkem je 1. Pamatujte, že "přetečení" indexu je možno použít pouze u statického pole, kde jsou prvky hned za sebou[45].

***medium*:** pravda, pravda, nepravda[46], 8

***hard*:** pravda. Uzávorkování je nutné – priorita by s tím udělala kouzla. Vyzkoušejte si.

Nyní bychom měli být procvičeni a dostatečně "in", takže si rozebereme možnosti definice dynamických vícerozměrných polí.

Nejvíce používaným polem je pole dvourozměrné, kterým můžete reprezentovat téměř vše, co budete kdy potřebovat, včetně textových souborů, načítaných právě jako pole řetězců. Protože máme ukazatele ještě dostatečně v krvi, proč se nedívat na dvourozměrné pole jako na pole pointerů, kde pak každý pointer ukazuje na jednu řádku?

```
char *super_pole[2];
```

[45] Kolem a kolem, je možno definovat i dynamické vícerozměrné pole, kde by to fungovalo, ale radši si takto nehrajte.

[46] Po dereferencování `gamesa[0]`, tedy ukazatele na první řádek, zůstávají elementem jenom znaky, tedy přičteme jeden znak, nikoliv řádek.

Definovali jsme si (statické) pole dvou ukazatelů.

Každý řádek musíme samozřejmě před použitím naalokovat, tedy:

```
super_pole[0] = (char *) malloc(sizeof(char)*BUFF1);
super_pole[1] = (char *) malloc(sizeof(char)*BUFF2);
```

Paměť můžeme kdykoliv uvolňovat a znovu alokovat. Všimli jste si, že každý řádek může být jinak dlouhý? Tuto možnost v případě matice asi nepoužijeme, ale v případě nestejně dlouhých řetězců zřejmě ano.

Tento typ pole je často používán, je s ním však spojen jeden problém – jeden jeho rozměr je statický, takže i když můžeme délku řádků měnit, počet řádků nikoliv.

Pokud tedy chceme "plně" dynamické pole, musíme definovat ukazatel na pole ukazatelů, tedy spíše ukazatel na ukazatel – neboť pole je ukazatelem na první prvek – například:

```
int **pole_jak_ma_byt;
```

Potom si můžeme napsat krásnou funkci na dynamickou alokaci paměti potřebnou pro čtvercovou matici:

```
int **ctv_matice(int rozmer)
{
   int i;
   int **neco1;
   neco1 = (int **) malloc(sizeof(*int)*rozmer);
   for (i=0;i<rozmer;i++)
      neco1[i] = (int *) malloc(sizeof(int)*rozmer);
   return neco1;
}
```

Ještě existuje jeden typ alokace dynamického pole. Stejně jako jsme v prvním případě fixovali velikost jednoho rozměru, můžeme místo toho fixovat velikost v druhém rozměru[47].

Měli jsme pevný počet řádků a pohyblivou délku řádky, možno říci pohyblivý počet sloupců. Nyní uděláme pole s pohyblivým počtem řádků o pevné délce.

Specifikujeme délku řádku:

```
int (*uk_pole)[M_DELKA];
```

A nyní si pomocí `malloc()` naalokujeme tolik řádků, kolik potřebujeme. V budoucnu samozřejmě můžeme použít `realloc()` ke změně počtu řádků.

```
uk_pole = (int *[M_DELKA])
      malloc(sizeof(int)*M_DELKA*M_POCET);
```

M_DELKA je délka jednoho řádku a M_POCET je počet alokovaných řádků.

Do pole přistupujeme jak jsme zvyklí, v případě dvourozměrného pole pomocí dvou indexů. Pořadí indexů se nemění, první zůstává větší rozměr, čili číslo řádky.

Cvičení: Implementujte načítání textového souboru do paměti a listování v tomto souboru. Uživatel zadá jméno souboru, vy jej načtete do dynamického pole, vytisknete první řádku a dovolíte uživateli pohybovat se mezi řádky.

Pomoc: Můžete alokovat bloky paměti o pevné délce.

[47] Pro vícerozměrné pole v největším a v nejmenším.

3.6.3 Struktury

Ke strukturám pouze krátkou zmínku.

```
struct zamestnanec {
   char jmeno[40];
   char prijmeni[60];
   char adresa[150];
   float plat;
};
struct zamestnanec *uk_zamest;
```

Se definuje ukazatel na strukturu. Přístup k jednotlivým položkám přes ukazatel je samozřejmě přes dereferenční operátor, tedy

```
(*uk_zamest).jmeno
```

Kde ovšem nesmíme zapomenout na závorky, neboť `*` má nižší prioritu než tečka.

Máme k dispozici šikovný operátor `->` , který používáme právě při přístupu ke složkám struktury přes ukazatel a to následujícím způsobem:

```
uk_zamest->jmeno
```

Použití tohoto typu zápisu je vhodné, jednak pro orientaci, jednak proto, abychom dělali méně chyb.

3.7 Operace s řetězci a znaky

V této kapitolce si ujasníme co je vlastně řetězec a jak se s ním pracuje a poté se podíváme na operace s jeho elementy – znaky.

3.7.1 Řetězce

Co je to vlastně řetězec?

Jednou říkám že je to pole znaků, podruhé zase tvrdím, že je ukončen znakem `\0`. Tak jak to je?

Vážení přátelé, platí oboje. Jasně řečeno, řetězec je, jak již vyplývá z definice, pole znaků.

```
char konst_retez[]="Konstantni retezec";
```

Není to tak dávno, co jsme probírali pole, takže si jistě vzpomenete, že je indexováno od nuly, čili `konst_retez[0]=='K';`

Řetězec je pole, tedy `konst_retez` je ukazatel na něj, konkrétně na první prvek tohoto pole. Výše uvedené pole je polem statickým. Co nám brání definovat dynamické řetězce? Nic, klidně to dělejte. nesmíte ale zapomenout na velice důležitou věc, a to ukončení pole znakem `\0`. Lépe řečeno, pokud definujeme

```
char retez[15];
```

Pak

`retez[0]` je prvním znakem řetězce

`retez[13]` je posledním možným rozumným znakem řetězce

`retez[14]` je nejposlednější místo, kde muže být uložen znak `\0`

Teď se mě ptáte, proč tam musí být ten prapodivný znak. Protože všechny funkce pracující s řetězci tak poznají konec řetězce. To znamená, že pokud ve výše definovaném řetězci bude znak `\0` na prvním místě, dostaneme prázdný řetězec. Proto opakuji, pamatujte na ukončovací znak, neboť v případě výpadku vaší paměti se mohou dít mysteriózní věci.

Ještě *poznámečka*. V definici `konst_retez` jsem žádný znak `\0` neuváděl, jak to?

Odpověď: V případě definice konstantních řetězců doplní překladač `\0` sám.

V jazyce C máme spoustu funkcí pro práci jak s řetězci, tak se souvislými bloky paměti. Funkce pro práce s bloky paměti začínají slůvkem *mem* a ty pro práci s řetězci zase *str*. Většinou existují pro nějakou operaci obě varianty, ve skutečnosti se liší snad jen tím, že *str* počítají s ukončením řetězce `\0`[48].

Tyto funkce nalezneme v hlavičkovém souboru `string.h`

Uvedu zde všechny nejpodstatnější.

`strcpy()` – *string copy*[49],

```
char *strcpy(char *dest, const char *src);
```

zajišťuje kopírování řetězce *src* na pozici *dest*. Slůvko *const* znamená, že obsah paměti, kam ukazuje pointer *src* nebude měněn. Funkce zkopíruje všechny znaky (včetně `\0`) ze *src* do *dest*. Návratová hodnota je *dest*. V případě překrývání paměti *dest* a *src*, není chování funkce definováno. *dest* musí být dostatečně velké, aby pojmulo *src*, to si pořádně hlídejte.

Uvádím doslovnou citaci z “Linux programmer’s manual”, lépe řečeno, manové stánky pro `strcpy(3)`, část BUGS:

> If the destination string of a `strcpy()` is not large enough (that is, if the programmer was stupid/lazy, and failed to check the size before copying) then anything might happen. Overflowing fixed length strings is a favourite cracker technique.

Pokud někoho zajímá překlad:

> Pokud není cílový řetězec funkce `strcpy()` dost velký (to je tehdy, když byl programátor tupý/líný, a nezkontroloval velikost před kopírováním) pak se může stát cokoliv. Přeplňování řetězců pevné délky je jedna z oblíbených crackovacích technik.

K dalším standardním funkcím pro práci s řetězci patří funkce `strlen()`

```
size_t strlen(const char *s);
```

[48] To je taková polopravda že by se dala nazvat lží. Ale nebojte, vše se v pravý čas dovíte.

[49] Pro ty, co si rádi hrají se slovíčky, také string cpi (nacpání řetězce do paměti).

Která počítá znaky v zadaném řetězci, nepočítaje v to ukončovací znak \0. Návratová hodnota je počet znaků.

Jiná funkce pro kopírování řetězců se jmenuje `strncpy()`, jediný rozdíl oproti `strcpy()` je ten, že kopíruje právě n znaků.

```
char *strncpy(char *dest, const char *src, size_t n);
```

Pokud mezi kopírovanými znaky není ukončovací znak, nezkopíruje se, na což je třeba dávat pozor. Naopak, jestliže je kopírovaný řetězec (*src*), kratší než n znaků, je zbytek znaků do n doplněn znaky \0.

Další z často využívaných funkcí pro kopírování řetězců je funkce `strdup()`, duplikující řetězec.

```
char *strdup(const char *s);
```

Nový paměťový prostor je alokován automaticky s použitím `malloc()`. Pokud funkce uspěla, vrátí ukazatel na nový řetězec, jinak `NULL`.

GNU C má další rozšíření – funkci `strndup()`. Funkce má poněkud odlišné chování od `strdup()` i od `strcpy()` – řekl bych, že to nejrozumnější jaké bychom chtěli, ale nebudu ji tady popisovat, poněvadž nepatří do standardu ANSI C. Viz dokumentace GNU C.

Existují ještě `strdupa()` a `strndupa()`, které pro alokaci paměti používají `alloca()` místo `malloc()` a jsou tedy rychlejší, ale to si nastudujte sami[50].

Od kopírování řetězců přejdeme k jejich zřetězování.

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Funkce pracují přesně jak očekáváme – k řetězci *dest* je přikopírován řetězec *src* a to tak, že je přepsán ukončovací znak řetězce *dest*. Řetězce se nesmí překrývat a *dest* musí být dost velké, jinak je výsledek funkce nedefinován. Funkce vrací ukazatel na dest. `strncat()` se liší pouze tím, že kopíruje pouze *n* znaků – opět pozor na to, aby byl výsledný řetězec ukončen \0!

Pro vyhledávání znaku v řetězci máme funkce

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

Parametr *c* je samozřejmě hledaný znak a parametr *s* řetězec, ve kterém vyhledáváme. Funkce vracejí pointer na první nalezený znak, pokud se znak v řetězci nevyskytuje, vrací `NULL`. Funkce `strrchr()` prohledává řetězec odzadu[51].

Pokud chceme vyhledat podřetězec v řetězci, můžeme použít

```
char *strstr(const char *haystack, const char *needle);
```

[50] Hlavně si předtím najděte, co je to `alloca()`!

[51] r=*reverse*

Vrací ukazatel na první výskyt řetězce *needle* v řetězci *haystack*, nebo `NULL` pokud needle není podřetězcem v řeť. *haystack*. Pokud je *needle* prázdný, vrátí celý *haystack*[52].

K podobnému účelu – prohledávání řetězců – slouží i funkce `strspn()`, `strcspn()`, `strpbrk()`.

Co se týče porovnávání řetězců, můžeme použít

```
int strcpm(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

s1 a *s2* jsou porovnávané řetězce, funkce vracejí 0 pokud se řetězce shodují, zápornou hodnotu pokud je řetězec *s1* lexikograficky menší a kladné číslo pokud je menší *s2*.

Podobně se chová i `strcoll()` pomocí které lze nastavovat porovnávání skrze národní abecedy[53].

Pokud budete někdy potřebovat parsovat řetězce, vzpomeňte si na funkce `strtok()` a `strsep()`[54].

Tak a teď zbystřete pozornost, protože přijde něco opravdu důležitého.

Uvedli jsme si zde spoustu funkcí, které se hojně používají.

Nevím jak se přesně vyjádřit, abyste mě nepochopili špatně, ale to, že se používají tak často, je špatné.

Co chci říci? I když máme takovéto funkce pro práci s řetězci, v drtivé většině případů můžeme pracovat s řetězci jiným způsobem a, co je nejdůležitější, rychleji.

Musíte si uvědomit, že každá z uvedených funkcí musí před tím, než začne s řetězcem pracovat, zjistit jeho délku. Takové zjišťování není žádná sranda, protože musí procházet celý řetězec než najde ukončovací znak. Teprve poté, co je délka řetězce známa, může začít samotná práce, třeba kopírování.

Zamyslete se nad tím, jak probíhá zřetězení dvou řetězců. Vyhledávání konce proběhne dvakrát! (pro první a pro druhý řetězec)

To nejhorší ovšem je, že při používání těchto funkcí si vůbec neuvědomíte, že se délka řetězce zjišťuje vždy znovu, pokud tedy provádíte s řetězci mnoho operací, zbytečně plýtváte výpočetním časem. Snad nejhorším takovým příkladem je zřetězování několika řetězců do jednoho.

Rozeberme si nějaký konkrétní příklad: zjišťujeme délku nějakého řetězce, počítáme kolik má být místa navíc[55], alokujeme potřebné místo, řetězec tam zkopírujeme, pak k němu např. něco přiřetězíme, najdeme nějaký znak a řetězec duplikujeme.

A teď se na to celé podívejme: délka, kterou jste zjistili pomocí `strlen()` se znovu zjišťovala při kopírování, znovu při přiřetězování a znovu při prohledávání – pardon, při duplikování také. A ono zjišťování znamená procházení celé paměti. I když máme *cache*-paměť, plno cyklů procesoru se proplýtvalo zbytečně. Nebylo by mnohem efektivnější použít již jednou zjištěnou délku znovu a znovu a znovu, aniž by se musela zjišťovat[56]?

Určitě bylo, a jak na to?

Není tomu tak dávno, co jsem se zmínil o funkcích pro práci s bloky paměti, začínajícími slůvkem mem. Máme například

[52]Pro zajímavost: *needle* je česky jehla a *haystack* kupka sena.
[53]Co se týče národních abeced, prozkoumejte si `locale.h` a speciálně `setlocale()`.
[54]Tato funkce není ve standardu ANSI C ani ISO C.
[55]Například chceme řetězec nějak upravovat.
[56]Opakuji, zbytečně se zjišťuje délka naprosto stejného řetězce.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Která kopíruje *n byte* z pozice *src* na pozici *dest*. Funkci použijeme naprosto stejně jako `strcpy()`. Cílové místo alokujeme pomocí `malloc()` nebo podobné funkce a pak provedeme `memcpy()`, kde jako *n* uvedeme délku řetězce zvětšenou o 1.

Proč? Protože `strlen()` nepočítá do délky řetězce nulový znak, ale my jej samozřejmě zkopírovat chceme.

V případech, kdy délku řetězce známe, samozřejmě ani nemusíme volat `strlen()`.

Funkce vrací pointer na *dest*. Pokud vám dělá starost pointer na `void` místo na `char`, vzpomeňte na přetypování, které používáte u `malloc()`, ta přece také vrací ukazatel na `void`.

Pokud chcete kopírovat mezi místy, které se v paměti překrývají, můžete použít `memmove()`.

Mezi další funkce patří

```
void *memchr(const void *s, int c, size_t n);
```

Která vyhledává znak c v bloku paměti s o velikosti n.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Porovnává blok paměti jako `strcmp()`, návratová hodnota je totožná.

Pro nastavení bloku paměti na určitou hodnotu (např každý byte na hodnotu 0) slouží funkce `memset()`.

Na závěr zmíním funkci `memmem()`, která provádí vyhledávání bloku v bloku (podřetězce v řetězci). Tato funkce nepatří do standardu ANSI C, je to rozšíření GNU C.

Cvičení: Napište program s jednoduchým textovým rozhraním umožňující zadat libovolný počet vět, pak jimi procházet, přehazovat je, zřetězovat, rozdělovat a hledat v nich. Přehazování je myšleno dvě mezi sebou a také posunutí věty na konec. Zřetězuje se vždy druhá k první. Při rozdělení je možno oddělenou část uložit hned za dělenou větu nebo jako poslední větu. Hledat je možno ve všech větách, zobrazují se čísla vět, kde se daný řetězec nachází. Po konci žádné operace nesmí být mezi větami žádná "prázdná" věta.

Návod:

- Horší postup – kompletně dynamické pole řetězců, používají se funkce s prefixem *str*;
- Lepší postup – kompletně dynamické pole řetězců, používají se funkce s prefixem *mem* a funkce `strlen()`;
- Varianta lepšího postupu – kompletně dynamické pole, element – struktura řetězec+délka, používají se funkce s prefixem *mem*, funkce `strlen()` jen při zadání věty.

3.7.2 Řetězce versus řetězce versus čísla

Jak načíst řetězce nebo čísla z klávesnice, popřípadě ze souboru, to již víme. Jak je tisknout už jsme si také probrali. Ale jak je převádět mezi sebou, když už je jednou máme načtené?

Jednou z možností je použít funkcí

```
int atoi(char *nptr);
long int atol(char *nptr);
```

Definovaných ve `stdlib.h`. Odtamtud pochází i ostatní funkce.

`atoi()` a `atol()` převádí řetězec *nptr* na číslo. Nebudu je ale rozepisovat, poněvadž není doporučeno je používat – nedetekují chyby přetečení. Místo toho vřele doporučuji funkci

```
long int strtol(char *nptr, char **endptr, int base);
```

Převádějící řetězec na `long int`. Funkce je značně variabilní – pomocí parametru *base* můžeme určit z jaké soustavy převádíme. Pokud tedy chceme pouze nahradit `atoi()`, použijeme

```
strtol(ukaz, NULL, 10);
```

Pokud je v řetězci uloženo např binární číslo, použijeme 2 jako parametr *base*. *Base* může nabývat hodnot 2 až 36[57]. *nptr* je samozřejmě ukazatel na řetězec.

Pokud není *endptr* `NULL`, do `*endptr` je uložena adresa prvního nepřevedeného znaku. Pokud nejsou nalezeny žádné číslice, `*endptr` je rovno *nptr*. Pokud po skončení funkce je na adrese `**endptr` `\0`[58], byl převeden celý řetězec.

Konečně se dostaneme k pravidlům převodu.

Ať již vezmeme kteroukoliv funkci, pravidla jsou stejná.

Počáteční bílé znaky jsou přeskočeny, poté je vzato v úvahu znaménko + nebo - a následující číslice se konvertují tak dlouho, dokud se nenarazí na nekonvertovatelný znak.

`strtol()` vrací při podtečení konstantu LONG_MIN, při přetečení LONG_MAX. V obou případech nastaví *errno* na ERANGE.

Co se týče bází větších než deset, za číslici představující desítku je považováno *a*[59], za jedenáctku *b* a tak dále až *z* za 35.

Pokud je báze nula nebo šestnáct, řetězec může mít prefix `0x` a poté je čten v šestnáctkové soustavě. Specifikujeme-li nulu, číslo je čteno jako decimální, pokud ovšem nezačíná nulou – v tom případě je považováno za číslo v osmičkové soustavě.

Jako další převodní funkce máme například

```
double strtod(const char *nptr, char **endptr);
float strtof();[60]
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```

Jimiž se ale nebudeme blíže zabývat, neboť jsou funkci `strtol()` velice podobné.

Jako další možnost můžeme použít funkci

```
int sscanf(const char *s, const char *format, ...);
```

která pracuje stejně jako stará známá `scanf()`, jen místo ze vstupu čte z řetězce *s*. Návratovou hodnotou je opět počet úspěšně načtených položek.

Možností převodu řetězce na číslo máme již opravdu hodně, nyní se podíváme na opačný směr – převod čísla na řetězec.

[57]Ve verzi dokumentace GNU C, kterou mám k dispozici, je uvedeno 2 až 35, to je ale zřejmě chyba – však si vyzkoušejte.

[58]Jinými slovy, pokud je `*endptr == NULL`

[59]Nezávisle na tom, zda je malé či velké.

[60]Tato funkce je GNU rozšířením, v normě ANSI se nevyskytuje.

Když máme příkaz, který bere jakákoliv čísla a vůbec spoustu věcí a tiskne je ve formě řetězce na standardní výstup (funkce `printf()`), proč bychom takový krásný nástroj nevyužili i nyní? Existuje varianta funkce známá jako `sprintf()`:

```
int sprintf(char *str, const char *format, ...);
```

Funkce se chová stejně jako `printf()`, jen zapisuje řetězec na adresu *str*. Vrací počet znaků, zapsaných do *str*. Řetězec sama ukončí ukončovacím znakem, který se nepočítá do počtu zapsaných znaků.

Samozřejmě některé z vás určitě napadlo, že i tato funkce může být velice nebezpečná – pokud budeme do řetězce vpisovat i nějaký jiný řetězec, může se stát, že zapíšeme více než jsme naalokovali. Proto existuje hojně používaná varianta, samo sebou se jménem `snprintf()`:

```
int snprintf(char *str, size_t size, char *format, ...);
```

Funkce neuloží více než *size* znaků (počítaje v to i znak `\0`), je velice vhodné a jednoduché volat ji s parametrem *size* rovným množstvím paměti alokované pro *str*. Teď jsem zjistil, že `snprintf()` nepatří do standardu ANSI C, ale splňuje normu ISO C99.

Návratová hodnota `snprintf()` je počet znaků, které by se zapsaly při dostatečné velikosti bufferu[61].

Cvičení: Napište program který uživateli dovolí vložit dvě čísla ve formátu jaký si vybere (číslo-decimálně, číslo-hexa, číslo-okta, text-decim, text-hexa, text-okta). Tato dvě čísla porovnejte a na výstup dejte jejich rozdíl v dec, hexa a okta.

Poznámky:

- program má porovnávání provádět pouze jednou, ne třikrát
- v případě jakékoliv chyby v zadání čísla tuto chybu uživateli ohlaste, ale pracujte dál s největší částí čísla, kterou jste ze vstupu schopni dostat (zleva) – pokud nevytáhnete nic, počítejte s nulou. V těchto případech (kdy neberete celé číslo) ale na výstup ještě dejte čísla, která ve skutečnosti porovnáváte.

3.7.3 Znaky

Velice krátce navštívíme užitečné funkce v `ctype.h`, týkající se klasifikace a konverze znaků. Ještě než se do nich pustíme, bych chtěl zdůrazniti, že všechny funkce se dají přinutit k práci s národní abecedou.

Stručně:

`int islower(int c);` Vrací *true* pokud je znak malé písmeno.

`int isupper(int c);` *true* při velkém písmenu

`int isalpha(int c);` *true* při znaku abecedy

`int isdigit(int c);` *true* pro číslici

[61] Je vhodné testovat, zda nebylo *size* menší než toto číslo.

`int isxdigit(int c);` *true* pro šestnáctkovou číslici (0-9, a-f, a-F)

`int isalnum(int c);` *true* pro znak abecedy nebo číslici (alfanumerický znak)

`int isgraph(int c);` *true* pro grafický znak (v ASCII od mezery až do poloviny)

`int isprint(int c);` *true* pokud je znak tisknutelný – grafický nebo mezera

`int ispunct(int c);` *true* pro tisknutelný znak, který není alfanumerický

`int isspace(int c);` *true* pokud je znak bílým místem (`\t`, `\v`, `\f`, `\n`, `\r`, `0x20` - mezera)

`int iscntrl(int c);` true pokud je znak kontrolním znakem (není tisknutelný)

Dále máme dvě funkce

```
int tolower(int c);
```

vrací malé písmeno z velkého. Pokud argument není velkým písmenem, vrací znak beze změny.

```
int toupper(int c);
```

obdobně jako `tolower()`, jen převádí malé na velké.

Cvičení: Napište funkce na konvertování řetězce na malá písmena, na velká písmena, na prohození velkých/malých, na vyparsování mezer a na vytáhnutí všech číslic z řetězce. Tyto funkce dejte k dispozici uživateli (přes jednoduché textové rozhraní). Uživatel zadá jako vstup jednu větu a pak ho s ní nechte manipulovat jak dlouho bude chtít.

Přídavek: Přidejte ještě funkce pro návrat k počátečnímu stavu věty a pro *undo* poslední operace (stačí jen jedné poslední operace).

Široké znaky

Pokud byste někdy potřebovali používat široké znaky, existují pro ně ekvivalentní funkce, například `iswalnum()` jako `isalnum()` a podobně. Viz `wctype.h`

Stejně tak pro práci s řetězci tvořenými širokými znaky existuje `wchar.h` - například `fgetwc()`, `wprintf()` a podobně.

Kapitola 4

Ovládání a struktury hlouběji

V této kapitole si prohloubíme naše znalosti o předávání parametrů funkcí, zpracování parametrů z příkazové řádky, projdeme druhy proměnných co se uložení v paměti a původu týče, konečně si vysvětlíme asi stokrát použitý pojem makro a nakonec se podíváme co a k čemu je Makefile.

4.1 Argumenty funkcí

Nejprve syntakticky, abychom si rozuměli. Co je to argument a co parametr? V tomto textu používám tato slova jako jedno, lépe řečeno, myslím jimi to samé. Funkce má vždy ovšem dva typy argumentů (nebo parametrů) a to

Formální parametr ... definice proměnné, uvedená v hlavičce funkce

Skutečný parametr ... skutečná hodnota, která je funkci předána za běhu

Celou tuto kapitolku se budeme bavit pouze o formálních parametrech.

4.1.1 Zápis funkce

Jen pro připomenutí, hlavička funkce vypadá

```
type_of_ret_value name(parameters);
```

Kde *parameters* je seznam parametrů oddělených čárkami a popřípadě bílými místy. Seznam může být prázdný. Každý parametr se skládá z typu parametru a jeho názvu, například

```
int function(int something);
```

nebo

```
float just_another_function(char **string,
          struct body definition_above, ...);
```

nebo něco dvacetinásobně složitějšího.

Tři tečky v deklaraci funkce vás nesmí zarazit, znamená to pohyblivý počet argumentů, o tom ale až později[1].

[1] A vy jste si mysleli že jsem chtěl říct, že tam ještě cosi je, chachá.

4.1.2 Definice, deklarace a prototyp funkce

Měli bychom si ujasnit, co je definice a deklarace funkce.

Definice funkce znamená kompletní hlavička funkce a tělo funkce. Naproti tomu deklarace funkce je jen její jméno a specifikace typu návratové hodnoty, případně i typy a názvy parametrů.

Ha, a na co nám je deklarace funkce dobrá?

Někdy se stane, že funkce volá jinou funkci, která je ovšem definována až níže ve zdrojovém kódu. A tato druhá funkce zase v některých případech volá funkci první. Jednoduše řečeno, ať dáme kteroukoliv dopředu, nikdy to nebude dělat dobrotu a vždy se objeví volání neznámé funkce.

Deklarování funkce se nazývá funkční prototyp – je to vlastně hlavička funkce.

Prototypy dále použijeme pro specifikaci funkcí v hlavičkových souborech.

Prototyp funkce vypadá:

```
int funky_function(int f1, float f2);
```

Povšimněte středníku, který je nutnou součástí tohoto funkčního prototypu[2].

Tento funkční prototyp můžeme napsat uvnitř nějaké funkce, ale, co je ještě lepší, uvést jej za direktivy `include` a `define` (viz 2.1.1.3 a 4.4), předtím, než začneme psát definici jakékoliv funkce. Alespoň si tak udržíme přehled.

4.1.3 Volání funkce

Postup volání funkce v C je následující:

- Volající uloží na zásobník hodnoty parametrů (skutečné parametry) a návratovou adresu. Poté předá řízení volanému[3].
- Volaný alokuje místo pro své lokální proměnné a po provedení svého těla je opět zruší a předá řízení na adresu návratové hodnoty.
- Volající ze zásobníku uklidí hodnoty parametrů, aniž by je nějak bral v potaz a pokračuje ve vykonávání svého těla.

Z toho vyplývá několik věcí:

1. Funkce může měnit hodnoty svých parametrů ale po jejím konci jsou tyto změny ztraceny – tedy je změnit nemůže.
2. Všechny parametry se kopírují a proto nebudeme předávat žádné velké datové struktury, ale pouze ukazatele na ně.

[2] Ano, je to přesně ten typ popisu, který používám již od začátku skript.

[3] A dostáváme se k jedné z nejčastějších bezpečnostních děr. Takový hacker, který zná adresu volané funkce v paměti, a odhadne dobře obsah zásobníku, má zelenou pokud je funkce špatně napsána. Například čte řetězec a ukládá jej na zásobník, přičemž nekontroluje jeho délku. Správná posloupnost hackerových akcí vede k tomu, že se řetězec na zásobníku protáhne a přepíše návratovou hodnotu. Pokud má hacker štěstí nebo potřebné znalosti o zásobníku, povede se mu uložit jako návratovou adresu adresu své vlastní funkce a po skončení funkce volané přebírá řízení ona. HA!

Ještě ke druhému bodu: Nemusíte se bát, že se v případě předávání pole zapomenete – pole je vlastně ukazatel na první položku, tedy pouhých několik *byte* – nikdy se nekopíruje jeho obsah.

Co máme ale udělat v případě kdy chceme předat velkou datovou strukturu, nekopírovat ji a nedovolit uvnitř funkce modifikaci této struktury?

Odpověď – v hlavičce funkce napíšeme místo

```
typ *jmeno
```

tohle:

```
const typ *jmeno
```

A pak nemůžeme obsah paměti, kam pointer ukazuje, uvnitř funkce změnit ani omylem. Pokud se podíváte například na `string.h`, zjistíte, že spousta funkcí v sobě obsahuje jako parametry ukazatele na řetězce, uvozené slůvkem *const*.

Cvičení: Projděte si předchozí příklady (funkce z předchozích příkladů) a rozhodněte, kde všude byste mohli použít slovo *const*. Poté si vyzkoušejte, abyste zjistili, kolik jste udělali chyb.

4.2 Argumenty funkce `main()`

Stejně jako mají funkce své parametry, může je mít i funkce `main()`.

Z historických důvodů se vždy nazývají *argc* a *argv*:

```
int main(int argc, char **argv);
```

Kde *argc* je počet parametrů představující počet parametrů na příkazové řádce[4] a *argv* je pole řetězců, tedy `argv[0]` je vždy jméno, pod kterým jsme program spustili.

Tyto parametry jsou nepovinné, nemusíme je v hlavičce funkce uvést, ovšem pak se nedostaneme k argumentům programu.

Cvičení: Napište program který na chybový výstup vypíše své parametry.

Cvičení 2: Napište jednoduchou kalkulačku (+-*/), která dostane zápis výpočtu jako parametry. Vyhrajte si s parsováním a sestavováním výrazu (platné je jak 2+5 tak 2 + 5). *Doplněk*: V případě chyby spočítejte co můžete.

4.3 Paměťové třídy

Už víme co je to proměnná, její typ a to, že máme několik druhů paměti, kde mohou být proměnné uloženy. Dále to, že některé mohou mít omezenou životnost. Nyní se na tento problém podíváme trochu blíže.

Při startu programu se v paměti vyhradí místo pro text programu[5], pro data a zásobník.

[4] Vždy minimálně jeden – jméno programu, kterým jsme jej spustili.

[5] Instrukce, obvykle pro čtení a provádění, nikoliv pro zápis.

4.3.1 auto

Neinicializované proměnné definované bez jakéhokoliv klíčového slova, např.

```
int a;
```

se nachází na zásobníku[6].

Tyto proměnné vždy vznikají se vstupem do funkce a zanikají s ukončením funkce, proto se jim říká automatické a mají paměťovou třídu "auto".

Tato třída je implicitní pro všechny proměnné ve funkcích (jak už bylo řečeno) a pro argumenty funkcí – o tom jsme již také mluvili – viz 4.1.3.

Slovo určující paměťovou třídu můžeme uvést před typem:

```
auto int b;
```

a tím jí typ explicitně přiřadíme. Třída *auto* je ovšem implicitní a tak to asi nebudeme potřebovat.

4.3.2 register

Další paměťovou třídou je třída *register*. Např.

```
register long int lreg;
```

Takováto proměnná je považována za automatickou, je však požadováno, aby byla umístěna v nějakém *registru* procesoru. Přístup k takové proměnné je nejrychlejší ze všech. Na proměnnou nelze definovat ukazatel. Může se stát, že proměnná do registru umístěna nebude – a to v případě že není *registr* k dispozici[7].

Typické použití této třídy je pro uložení adresy která bude následně nějak použita, nebo proměnnou přes kterou iterujeme cyklus. Různé optimalizace jsou ale prováděny překladačem a tak se například právě při iterování jednoduchého cyklu do *registru* proměnná pravděpodobně dostane sama, aniž bychom ji museli specifikovat jako *register*.

4.3.3 static

Další paměťovou třídou je *static*, která má dva významy. Jak jsme si řekli, globální proměnné jsou uloženy v části pro data. Pokud před globální proměnnou předřadíme slovo *static*, způsobíme tím, že proměnná bude viditelná pouze v modulu (souboru), který právě píšeme a můžeme beze strachu v jiných modulech definovat globální proměnnou stejného jména – takovéto proměnné budou nezávislé.

Pokud použijeme slovo *static* u lokální proměnné, která by měla být uložena na zásobníku, způsobíme tím opět její uložení v datové oblasti. Má to jednu výhodu – proměnná existuje jen jednou a hodnota v ní zůstává uchována i pokud funkce skončí. Tj. při dalším spuštění funkce bude proměnná mít takovou hodnotu, jakou měla při skončení předchozího běhu funkce. Můžeme si takto uchovávat nebo předávat nějaká data z jednoho spuštění funkce do druhého.

[6] Pokud ovšem nejsou globální – nepředchází definicím funkcí, v takovém případě by byly v sekci pro data.

[7] Např. proměnná je moc velká.

4.3.4 extern

Další z tříd je *extern*. Tuto třídu využijeme, když potřebujeme sdílet jednu globální proměnnou mezi několika moduly.

V modulu, kde proměnnou definujeme, ji napíšeme jednoduše jako obyčejnou globální proměnnou, v ostatních modulech ji uvodíme slovy `extern typ jméno`. Stejně tak můžeme označit slovem funkci – v tom případě označujeme, že funkce je definována někde jinde a neuvádíme její tělo – pouze ji deklarujeme.

4.3.5 Další ovlivnění

Další specifikátory, které již přímo neovlivňují způsob uložení proměnné ani její viditelnost, nýbrž zacházení s ní, jsou *const* a *volatile*.

const má opět dvojí použití. Pokud tímto slovem uvodíme inicializovanou definici proměnné, znamená to, že hodnota proměnné se nesmí změnit. Druhé použití je u pointeru jako parametru funkce a zajišťuje, že obsah paměti, kam pointer ukazuje, se nebude měnit, o tom jsme již ale také mluvili – viz 4.1.3.

volatile znamená, že se hodnota proměnné může měnit asynchronně, to jest, že ji může měnit něco vně programu, například systém, přerušení, jiný *thread* procesu atp. Typickým příkladem na použití slova *volatile* jsou právě ovladače přerušení, ovladače signálů nebo proměnné ve funkci, do níž je možno skočit pomocí `longjmp()`. Tato třída znamená, že překladač nemá provádět optimalizace týkající se čtení proměnné – například zkopírovat její hodnotu do *registru* a pracovat s ní tam. Má předpokládat, že to je taková "horká" proměnná a radši ji při každém použití vždy znovu přečíst.

Cvičení: Využijte předchozího příkladu s kalkulačkou – předělejte ji tak, že uživatel bude zadávat výpočet v průběhu programu – neustále další a další výpočty – a bude mít k dispozici výsledek minulý, pod nějakým symbolem (dejte mu to patřičně na vědomí). Tento minulý výsledek implementujte jako statickou proměnnou ve funkci.

Například: 4+6*8 a potom -3+PREV/26, tedy výsledek bude -1

4.4 Makra

Co je to makro?

V jazyce C není makro to, co známe z textových editorů – tedy jakási posloupnost akcí.

Makro v podstatě znamená pouze jakousi textovou náhradu.

Podíváme se na dva různé typy maker v C.

4.4.1 Běžné makro

"Klasické" makro je makro bez parametrů, může být považováno za jakousi konstantu[8]. Definujeme jej

```
#define name value
```

konkrétně třeba

[8]Někdy se mu říká symbolická konstanta.

```
#define BUFFER 500
```

Jméno makra je standardní identifikátor v C, ale z důvodu přehlednosti je psáno velkými písmeny a je více než vhodné tuto konvenci dodržovat.

K čemu takové makro slouží?

Jedna z notorických chyb při programování je psát přímo do kódu jakási čísla, určující chování programu[9]. Třeba čísla určující kolik paměti se alokuje atp.

Proč je to chyba? Nevíte? To se vám ještě nestalo, že jste jednou to číslo napsali, pak jste někdy potřebovali např. stejné množství paměti ještě jinde a tak jste ho napsali na ono druhé místo a někdy později na prvním místě hodnotu změnili, přičemž na změnu v druhém místě jste zapomněli a pak se divili že něco nefunguje tak jak má? Nebo například nehledali někde v textu to číslo, které určuje kolikrát se nějaká operace provede a ne a ne ho najít?

Takovéto věci se řeší právě zavedením maker a definováním makra na začátku programu. Nestane se vám, že čísla se na několika místech liší a konfigurace programu je pak přehlednější, neboť všechna tato čísla máte na jednom místě.

Pravda, nesmíte zapomenout na vhodné pojmenování maker, případně na jejich okomentování. Například velice vhodné je začínat svá makra stejným písmenem a podtržítkem, či snad shromáždit související makra do výčtového typu.

Jakýmsi pravidlem je nenechat v programu ani jedno zatoulané "magické" číslo.

4.4.2 "Funkční" makro

Někdy nazývané jako makro s parametry.

Nejprve si ho přiblížíme a až poté ukážeme, k čemu je a k čemu není.

```
#define name(parameters) value
```

konkrétně například

```
#define greater(a, b) ( (a>b) ? a : b )
```

Definuje se tedy opět pomocí klíčového slova, jména (s parametry) a hodnoty.

Funguje také velice podobně, jen volání

```
b=greater(5, a);
```

preprocesor expanduje na `b=( (5>a) ? 5 : a );` a teprve poté se do kódu pustí překladač. Čili makro se nám opět expanduje do své hodnoty, ovšem s tím, že výskyty parametrů jsou také nahrazeny.

Ok, vysvětlili jsme si, že i tento typ makra je pouhou textovou náhradou. Proč mu ale říkáme "funkční"? Jde o způsob použití makra. Při volání funkce se musí uložit parametry na zásobník, zavolat funkce a poté, co se provede, skákat na návratovou hodnotu a opět zásobník vypouštět. Pokud se jedná o velice krátkou funkci, je to zbytečně proplýtvaný výpočetní čas. Kdyby se tělo funkce napsalo přímo namísto jejího volání, bylo by vše značně jednodušší.

Jenže, psát stejnou věc na (například) třech místech v programu se nikomu nechce. Jednak proto, že když ji bude měnit, bude tak muset dělat na třech místech a nejspíš na nějaké

[9] Říká se jim "magická" čísla.

zapomene a navíc se člověk zbytečně upíše. Proto je tato věc nechána na preprocesoru použitím makra. Programátor se neupíše a kód je ve výsledku rychlejší.

Sice rychlejší, ale na druhou stranu delší. A navíc, co je nejpodstatnější, makra neprovádějí kontrolu typů argumentů, proto se v nich vyskytují častěji chyby. Navíc, makra se vyhodnocují před překladem a proto se chyby v nich překladačem nedají rozumně odchytit. Proto, pokud budete psát nějaké makro, jako že určitě budete, protože to je hezká věc, dávejte si pozor při každém jeho volání na typ parametrů se kterými ho voláte a operace které s nimi provádíte. Navíc, makra v sobě ukrývají nečekané zrady, třeba volání

```
greater(++pro1, pro2);
```

se expanduje na

```
( (++pro1>pro2) ? ++pro : pro2 );
```

Tedy *pro1* se může klidně dvakrát inkrementovat a to jsme asi nechtěli[10].

Pokud píšeme makro, snažíme se předvídat. Nezní to sice moc hezky, ale minimem, které můžeme udělat, je všechno uzávorkovat, abychom si byli jisti prioritou vyhodnocení, tedy například

```
#define greater(a, b) ( ((a)>(b)) ? (a) : (b) )
```

Nevypadá to tak hezky, ale je to bezpečnější[11].

Ještě k onomu "funkčnímu" typu makra. Existuje spousta céčkovských knihovních funkcí, které jsou ve skutečnosti pouhými makry, např. téměř všechny funkce `isalnum()`, `isdigit()` atp. jsou ve skutečnosti makra, např. funkce `tolower()` je v mém systémovém `ctype.h` definována:

```
#define tolower(c) __tobody(c, tolower, __ctype_tolower)
```

Čili pokud použiji v programu funkci `tolower()`, ještě před překladem se mi toto makro rozvine ve výše uvedenou stvůru. Nutno podotknout, že ona stvůra určitě kontrolu správnosti typů předávaných argumentů provádí[12], protože se pouze použije jiná funkce, čili je vše tak jak má být.

Co se týče funkcí, které jsou ve skutečnosti makra, asi byste byli překvapeni, kolik maker používáte.

4.4.3 Práce s makry.

Pokud se nám makro nehodí a chceme jej předefinovat, použijeme

```
#undef name
#define name new_value
```

[10] Jednou v testu a podruhé při vyhodnocování vraceného výrazu.

[11] Hlavní rozdíl makra a funkce spočívá v tom, že parametry funkce musí být známy už před vnořením do funkce, neboť se kopírují na zásobník – tedy se vyhodnocují jenom jednou. Kdežto u makra se parametry textově dosadí do textu makra a může tedy dojít k vícenásobnému vyhodnocení – pokud nezadáváme pouze samotná data, ale i operace s nimi.

[12] Na rozdíl od našeho *greater*.

Stejně tak můžeme zjistit, zda je makro nadefinováno a podle toho se chovat:

```
#ifdef name
commands
#endif
```

nebo případě že není definováno:

```
#ifndef name
commands
#endif
```

Pro případ zapeklitější definice můžeme použít `#elif` a `#else`, které se chovají jako `else if` a `else`. Pravidla jsou stejná jako pro klasické `else`. Výše zmíněným věcem se říká podmíněný překlad a ještě se k němu v následující kapitolce vrátíme.

Cvičení: Ze všech svých cvičení odstraňte "magická" čísla.

4.5 Dělení programu, dílčí překlad

V této kapitolce si nastíníme, jak se dělá nějaký trošičku větší program. Budeme mluvit hlavně o dělení zdrojového kódu a to do několika souborů – jak hlavičkových, tak klasických `.c`

Nejprve si vysvětlíme, k čemu to vlastně potřebujeme a pak se pustíme do popisu jak na to.

Proč bychom program měli dělit?

Důvodů je několik.

Pokud například dělá na programu několik lidí, těžko budou zpracovávat jeden soubor, místo toho si práci rozdělí.

Pokud nacpeme program do jednoho velkého souboru, bude velice náročný na překlad. V případě dělení do několika souborů se může překládat každý soubor zvlášť, také v případě malé úpravy bude třeba překládat jen jeden z nich, který se pak slinkuje s ostatními, které zůstaly nezměněny.

Dalším důvodem je přehlednost programu. Lépe řečeno, program lze logicky rozdělit do několika dílů, ať již podle funkcionality[13] nebo podle jiného dělení[14].

Podívejme se nyní, jak se program dělí, lépe řečeno, podívejme se na pravidla, která bychom při dělení mohli dodržovat.

4.5.1 Rozdělení programu

Předpokládejme, že jsme si zdrojový text rozdělili (rozumně) do několika souborů.

- Funkci `main()` máme jenom v jednom z nich[15].

[13] Např. funkce pro práci se souborem, funkce pro kryptování, funkce pro databáze, funkce pro...

[14] Funkce k sobě patří podle nějakého hlediska, např inicializační funkce zvlášť, zvlášť hlavní funkce a zvlášť funkce používané pro vstup a výstup atp.

[15] To nemusí být pravda, zvláště pokud chceme moduly testovat samostatně, ale o tom později.

- Všechny globální symbolické konstanty – makra – uložíme do hlavního hlavičkového souboru, který pak zahrneme (`include`) ve všech zdrojových souborech.
- V tomto souboru uvedeme také všechny funkce které mají být viditelné ve všech modulech – pomocí slova *extern*, ať již budou pocházet z jakéhokoliv modulu.
- Je vhodné udržovat ke každému souboru `.c` hlavičkový soubor `.h`, kde jsou uvedeny všechny prototypy funkcí, případně včetně lokálních maker a to z důvodu přehlednosti. Toto však není nutným pravidlem.
- V žádném ze souborů nezahrnujeme žádný jiný soubor než `.h` (nezahrnujeme `.c` soubory).
- V souborech `.h` nepřidělujeme žádnou paměť – nic nedefinujeme.
- Všechny globální proměnné v nějakém souboru[16], které nemají být nikde jinde používány, uvodíme slovem *static*, aby nedocházelo ke zmatkům ve jménech.
- Takto rozdělené `.c` soubory můžeme velice snadno odděleně překládat do objektového tvaru – stačí nám jen zahrnout hlavičkové soubory. Funkce, které píše někdo jiný, jako bychom měli.

4.5.2 Testování částí

Ještě pár slov k tomu, jak otestovat interní funkce (nevyužívající žádné externí funkce) v samostatném modulu.

Na co to? V případě, kdy ostatní moduly ještě nejsou hotové a my si chceme tuto část projektu otestovat ještě když ji máme čerstvě v paměti – kdybychom chyby museli hledat až při konečné kompilaci, asi bychom se z jejich množství zbláznili.

Využijeme podmíněného překladu – například:

```
#ifdef SELF
int main()
{ ... }
#endif
```

V takto definované funkci `main()` si uděláme pořádné testy, které budeme moci za běhu ovládat – například spouštět libovolnou funkci na pokyn z klávesnice a dávat jim jaké chceme parametry a sledovat jejich výsledky.

Pokud potom překládáme s definovaným makrem SELF(viz 4.5.3), dovedeme překlad až na spustitelný soubor – máme funkci `main()` a vše si řádně odzkoušíme.

4.5.3 Definice makra pro překlad

Samozřejmě, jedna možnost je napsat makro do zdrojového kódu nebo do inkludovaného souboru, z toho ale vyplývá nutnost měnit zdrojový kód.

[16] A zároveň neexportované funkce.

Máme však ještě jinou cestu jak makro definovat – dát ho rovnou překladači[17]. Použijeme k tomu vlajku -D[18]. Některá prostředí umožňují zapsat makro někam do menu, což ale není nejlepší řešení, protože pak na něj určitě zapomenete. Podle mě ale bude vlajka -D fungovat i na mnoha jiných překladačích a systémech. Příklad:

```
bazil@fenix:~$ gcc -DSELF project_part_1.c
```

Čímž jsem definoval makro SELF – pomocí `-DSELF`.

4.6 Makefile

Zde si dovolím na trošku větší chvilku odbočit od standardů a zmíním se o programu *make* a jeho používání. Tato utilita je k dispozici na všech UNIX-like systémech. Pokud používáte jiný systém, nezoufejte, klidně na něj může být portována[19], stačí ji najít. Jestliže používáte nějaký grafický klikátor, sám dělá pravděpodobně[20] podobnou práci za vás, ale vy tím téměř jistě ztrácíte některé možnosti ovlivňování tohoto procesu. Dosti už, pojďme se podívat na *make*.

4.6.1 *make*

Volně překládám z manové stránky `make(1)`:

> Cílem utility *make* je automaticky rozpoznat, které části velkého programu je třeba přeložit a poté produkovat příkazy nutné k tomuto překladu...
>
> ...jako přípravu k používání *make* musíte napsat program popisující vztahy mezi soubory ve vašem programu. Tento soubor se nazývá makefile...

Jak *make* pozná, že má nějakou část programu přeložit a jinou nemusí?

Velice jednoduše – modifikované soubory (zdrojové) se poznají podle toho, že mají mladší datum modifikace než spustitelný program. A kdyby to z nějakého důvodu nezafungovalo, překlad je možno si vynutit pomocí nějaké volby *make*[21].

Ok, jak vypadá makefile?

Z důvodů konvence jej budeme nazývat Makefile a pokud to jen bude možné, pojmenujte jej opravdu velkým M.

4.6.2 Makefile-proměnné

V Makefile můžeme provádět definici Makefile maker, chovajících se podobně jako standardní shellovské proměnné, jinými slovy, definice makra vypadá

```
JMENOMAKRA=HODNOTAMAKRA
```

[17]Nebojte, on si ho vezme už preprocesor řešící podmíněný překlad, kompilátoru by makro bylo samozřejmě k ničemu.

[18]Na Linux/Unix, na ostatních překladačích se podíváme do dokumentace/helpu k překladači.

[19]Například pro MS-DOS *djgpp* je portována přímo GNU verze *make*.

[20]Pod nějakým klikadlem nazvaným *project*.

[21]Najděte si v dokumentaci/helpu.

například

```
FLAGS=-DSELF -DDEBUG
```

a rozvoj makra se zajistí pomocí

`$()` – tedy `$(FLAGS)` se rozvine na `-DSELF -DDEBUG`

Makro definované pomocí rovnítka se vypočítává "rekurzivně", tedy pokud definujeme

```
makro1=$(makro2)
makro2=$(makro3)
makro3=gulaasch
```

Poté, když někde v Makefile napíšeme `$(makro1)`, rozbalí se do *makro2*, to si vynutí vyhodnocení *makro3* a výsledek tedy bude `gulaasch`. Jinými slovy, makra se vyhodnocují na každém místě znovu a rekurzivně.

Nevýhodou je pomalejší běh Makefile[22] a nemožnost napsání něčeho jako

```
makro=$(makro) ahoj
```

Protože by rozvoj zabředl do nekonečné smyčky. Máme ještě druhý typ definování maker[23]:

```
makro1:=ahoj
makro2:=$(makro1) svete
makro1:=cau
makro4:=$(makro1) jak se mas
```

Pomocí operátoru `:=` se makro vyhodnocuje pouze jednou a pouze pomocí maker v té chvíli definovaných a jejich okamžitých hodnot. Výhodou je rychlejší vyhodnocování a možnost změny obsahu maker.

Nikdo neříká, že není možno v Makefile použít oboje.

V Makefile můžeme použít komentáře – uvodíme řádku znakem `#` a *make* ji bude ignorovat.

Pokud potřebujeme napsat nějakou delší řádku a chceme ji zpřehlednit, ale nesmí být rozdělena znakem konce řádky[24], rozdělíme ji pomocí zpětného lomítka (\). Obsah řádky pak může pokračovat na řádce další, *make* si je před zpracováním spojí na původní dlouhou řádku.

4.6.3 Makefile-pravidla

To nejpodstatnější, co se v Makefile vyskytuje, jsou pravidla. Pravidlo má následující podobu:

```
target : prerequisites
    command
...
```

Kde

[22] Pokud ale není extrémně velké, tak to nepoznáte.
[23] Já pořád o makrech a jak tak koukám, v dokumentaci tomu říkají proměnné.
[24] Například příkaz pro překladač.

target je jméno, česky cíl. Nejčastěji je toto jméno porovnáváno s parametrem příkazu *make* a představuje akci, která se má udělat. Program *make* ale spouštíme v drtivé většině (jednodušších) případů bez parametrů a proto první cíl, který v Makefile uvedeme, je vždy považován za implicitní. V jednodušších Makefile použijeme za *target* jméno, které má nést výsledný program. Při větších projektech je rozumné uvést jako první cíl `all:jméno` následováno prázdnou řádkou, tedy jakýsi cíl bez pravidel, kde jméno je jiný cíl. Je tak jasně řečeno, který cíl se stává implicitním a nemusíme si dělat starosti. Za takovým cílem pak zahrnujeme další Makefile (níže).

prerequisites jsou obyčejně soubory potřebné k sestavení programu, přesněji k dosažení cíle. Ve složitějších Makefile můžeme často vidět jako tyto prerekvizity i další cíle, které je třeba udělat.

command je akce, kterou make provede – říká, jakým způsobem má být cíl překompilován v případě neaktuálnosti. Je od začátku řádku oddělena tabulátorem[25]. Příkazů může být více (proto ty tři tečky), každý se uvádí na zvláštním řádku – každý je oddělen jedním tabulátorem.

Co se týče provádění akcí – pokud je cíl souborem – akce pro nějaký cíl se musí provést, pokud se některá prerekvizita změnila, nebo pokud soubor zatím neexistuje.

Uvedeme si nějaké jednoduché pravidlo, třeba:

```
program : program.o
    gcc -o program program.o
```

Cíl "program" se provede, pokud bude soubor `program.o` mladší než soubor `program`. Pro update souboru `program` se použije uvedený příkaz.

O Makefile by se dalo vykládat dlouho a já bych to rád zkrátil, takže letem světem a nakonec malý příklad, který snad vše objasní.

Pro psaní pravidel máme spoustu fíglů, například můžeme napsat jakési jednoduché multipravidlo:

```
ubs%.o:ubs%.c
    gcc -c $< -o $@
```

Tím specifikujeme, že každý `ubs*.o` má jako prerekvizitu `ubs*.c`, kde se ovšem části zastoupené za hvězdičku shodují. V případě neaktuálnosti se pustí příkaz

```
gcc -c ubs*.c -o ubs*.o
```

`$<` znamená první prerekvizitu a `$@` zase cíl. Velice užitečné je `$^` znamenající všechny prerekvizity.

Jakmile bude program růst, zjistíme, že nám ani takové finty nestačí. Dobrá, stejně jako jsme rozdělili zdrojový kód na různé soubory, rozdělíme i Makefile. Ovšem trochu jiným způsobem. Makefile budeme mezi sebou zahrnovat. Budeme mít jeden *top-level* Makefile. V něm

[25] Ne několika mezerami, opravdu tabulátorem.

definujeme potřebná makra, specifikujeme (výše uvedeným aliasem *all*) hlavní cíl, zahrneme všechny Makefile první úrovně a specifikujeme cíle hlavního Makefile.

V Makefile první úrovně[26] opět definujeme makra, specifikujeme hlavní cíl, případně cíle a zahrneme Makefile druhé úrovně. Tak to pokračuje dále až se dostaneme k úrovni poslední, která odpovídá buďto velice souvisejícím skupinkám zdrojových souborům, nebo v případě krize jednotlivým souborům.

Zahrnování se provádí `include název_makefile`, můžeme použít zástupné znaky. Probíhá tak, že *make* přestane číst současný Makefile, ponoří se do zahrnutého, a až jej přečte, vrátí se a pokračuje.

Opravdu složité projekty pouští *make* rekurzivně na jiné Makefile, ale to už se podívejte sami třeba na Makefile jádra Linuxu. Pokud vám z toho naskočí husí kůže, nic si z toho nedělejte, já to taky někdy viděl poprvé.

4.6.4 Makefile-příklad

Slibovaný příklad[27]:

```
#hlavni makefile pro project1
#

#cesta k formularum
FORM_PATH=forms

#kompilator a make
COMPILER=gcc
FLAGS=-DMAXL=10 -O3 -I$(FORM_PATH)
cc=$(COMPILER) $(FLAGS)
MAKE=make

#hlavni cil
all: project1

#ostatni makefile
include help.mk
include io.mk

#specifikace cilu, help a io mame importovane
#nekde z help.mk a io.mk
FILES=user.o $(FORM_PATH)/form.o help/help.o io/io.o
BUILD=forms help io

project1: $(BUILD) $(FILES)
    $(cc) $(FILES) -o projekt
```

[26]Rozděleno podle skupin zdrojových kódů spolu souvisejících.

[27]Parametr "-o text" u *gcc* znamená, že výstupní soubor se bude jmenovat text. Parametr "-C cesta" u *make* znamená, že předtím než vůbec začne něco make dělat, změní adresář na cesta.

```
user.o: user.c userdef.c lang.c
    $(cc) $^ -o $@

forms:
    $(MAKE) -C $(FORM_PATH)
# ^^^ změní adresář a spustí v něm make bez
#parametrů - chytne se Makefile v daném
#adresáři a implicitní cíl


clean:
    $(FORMS) -C forms clean
    rm project1 user.o
#     ^^^ podobně jako při cíli forms,
#jen se v Makefile provádí cíl clean
```

Ostatní Makefile už uvádět nebudu. Když tak se podívejte třeba na to jádro[28]. Opravdu to umí lépe než já.

[28] *ftp.kernel.org, www.linux.cz, www.root.cz, ftp.fi.muni.cz* a mnohé další.

Kapitola 5

Stoupáme do výšin

V této kapitole si ukážeme poslední jazykové prvky, které se v tomto kurzu naučíme. Nemusíte být tak smutní, už jsme si toho ukázali opravdu moc, dá se říct že drtivou většinu věcí.

Co jsme moc neprobírali, byly vlastní funkce, vlastně jsme se nedostali ani k jedné aritmetické funkci. Najdete je popsané snad všude, dokonce ani nepotřebujete žádnou příručku, stačí jen se podívat do hlavičkových souborů `math.h`, `stdlib.h` a na funkce pro práci s časem do `time.h`.

Tak úplně se s jazykem C ale neloučíme, protože si ukážeme ještě nějaké "fígle" jak a co dělat když se něco nevede a provádět nějakou minimální optimalizaci, ale to už opravdu se samotným jazykem moc společného nemá.

Navíc se s C potkáme v druhé části kurzu, neboť jazyk C++ z jazyka C vzešel a má s ním společné neuvěřitelné množství prvků a dosud získané znalosti účelně využijeme.

Velice dlouho jsem přemýšlel, zda se mám vrhnout na `fork()`, ale nakonec jsem rozhodl že ne, protože vlastně do žádného standardu jazyka C tato věcička nepatří. Koneckonců už Makefilem jsem se dosti odchýlil od standardů.

Stejně tak nebudu probírat žádné síťové věci (*socket* atp.). Pokud něco takového potřebujete, pravděpodobně děláte na LINUXu nebo podobném OS a tam máte plno dokumentace a manových stránek. Podívejte se také na seznam literatury na konci textu. Kdyby vám to nestačilo, podívejte se na internet, najdete přehršel zdrojů.

5.1 Pointer na funkci

I funkce má svou adresu v paměti. Můžeme ji využívat, ale k čemu? Pokud máme funkci, která volá nějakou jinou funkci, může tak učinit přes ukazatel na volanou funkci a to tak, že si tento ukazatel vezme jako parametr. Tím jí vlastně parametrem určujeme, kterou funkci má použít.

5.1.1 Použití

Můžeme tak velice snadno udělat nějakou převodní funkci která používá jiné elementární funkce nebo funkci pro nějaký složitější výpočet, kde se v závislosti na okolnostech použije pro dílčí výpočet funkce zadaná parametrem.

Dále můžeme definovat pole funkcí, abychom je měli hezky uspořádané. Tedy, vlastně, pole funkcí přímo definovat nemůžeme, ale můžeme definovat pole ukazatelů na tyto funkce.

Dobrá, hezky popořádku.

5.1.2 Deklarace

Nadefinujeme si funkci vracející pointer na `int`:

```
int *func(int a);
```

Nyní si nadefinujeme pointer na funkci vracející `int`

```
int (*uk_f_int)();
```

Tím je snad jasná funkce závorek.

Pokud chceme konkrétně na funkci s dvěma parametry, prvním typu `int` a druhým `float`:

```
int (*uk_f_int)(int, float);
```

To nám ale nestačí, my samozřejmě potřebujeme pointer na funkci vracející pointer na `int`, tedy:

```
int *(*uk_f_poi_int)();
```

Pointeru přiřadíme adresu funkce takto:

```
uk_f_poi_int=func[1];
```

Tedy nic nedeferencujeme, neboť jméno funkce znamená i její adresu.

Ještě potřebujeme vědět, jak funkci pomocí pointeru zavolat:

```
uk_f_poi_int(param);
```

nebo také

```
(*uk_f_poi_int)(param);
```

Druhý způsob je méně hezký, ale je z něj okamžitě znát, že voláme funkci přes ukazatel na ní a to nám při čtení kódu (třeba někdy později) napoví, že se může v různých případech jednat o různé funkce a pokud hledáme nějakou chybu, tuto řádku si pravděpodobně musíme vyzkoušet při všech možných funkcích.

5.1.3 Pole pointerů

Jak dělat pole ukazatelů na funkce? Ale to přece umíme, úplně stejně jako každé jiné pole, prostě se za proměnnou přidají hranaté závorky a tím se z ní udělá pole. Budeme pak ale potřebovat specifikovat rozměr, anebo uděláme pole totálně dynamické tím, že definujeme ukazatel na ukazatele, ale i to už přece umíme. Ukázka pole bude v další kapitole, spřáhneme to s operátorem typedef – kap. 5.2.1.

Cvičení: Vyzkoušejte si funkčnost ukazatelů na funkce. Nadefinujte jeden ukazatel a nechte uživatele vybírat si mezi funkcemi – použití funkce realizujte přehozením ukazatele a volejte funkci přes ukazatel. Příkladem takového programu může být jednoduchá kalkulačka.

[1] Všimněte, za jménem funkce nejsou závorky – tím jednoznačně určujeme, že funkci nevoláme, nýbrž bereme její adresu!

5.2 Vlastní typy

Ve většině materiálů se tato možnost popisuje poměrně brzy, já si ji ale záměrně nechal až na konec, neboť teprve teď ji řádně oceníte a budete si pamatovat a používat.

Máme spoustu typů, `int`, `float` ... a můžeme si definovat vlastní.

Jak?

5.2.1 Operátor typedef

Pomocí operátoru `typedef` s následující syntaxí:

```
typedef specifikace nazev;
```

specifikace je tatáž, jako bychom specifikovali proměnnou, tedy nejjednodušší typ je pouhá náhrada nějakého typu vlastním slovem.

```
typedef int hegelemegele;
```

Od tohoto bodu můžeme proměnnou typu `int` definovat

```
int a,b;
```

nebo

```
hegelemegele a,b;
```

Asi je vám jasné, že to není to pravé ořechové – nikoliv kvůli názvu, ale kvůli tomu, že když to pak po vás bude někdo číst, bude si muset po celou dobu procházení zdrojového textu pamatovat že to a to slovo znamená `int`. A co teprve když si takto přejmenujete více typů – ach, radši ani neuvažovat.

Dobrá, řekl jsem, že používáme `typedef` stejně, jako bychom definovali proměnnou, pokud tedy definujeme

```
typedef struct {
   char puvod[10];
   int odruda;
   int hmotnost_kusu;
   long int mnozstvi;
} BRAMBORY;
```

Můžeme kdykoliv v programu tento typ použít, například

```
BRAMBORY pytel1, pytel2;
```

Pokud chceme, aby se struktura odkazovala sama na sebe (zřetězený seznam prvků), pozměníme definici

```
typedef struct brambory {
   char puvod[10];
   int odruda;
   int hmotnost_kusu;
   long int mnozstvi;
   struct brambory *dalsi_pytel;
} BRAMBORY;
```

Čímž dostaneme dynamický seznam (viz cvičení). Slovo brambory v definici typu je nutné, jinak bychom nemohli uvnitř struktury tvořit ukazatel – nemůžeme udělat ukazatel na typ BRAMBORY, protože tento typ ještě není definován, kdežto "struct brambory" už překladač má.

Jak správně tušíte, můžeme si klidně definovat typ odpovídající nějaké funkci (viz kapitolka 5.1), například definicí

```
typedef int *(UK_F_POI_INT)();
```

Dostaneme typ funkce vracející ukazatel na `int`, čili proměnnou *uk_f_poi_int* z předchozí kapitoly nyní definujeme jako

```
UK_F_POI_INT uk_f_poi_int;
```

A nakonec to slíbené pole funkcí, pardon, pole ukazatelů na funkce:

```
UK_F_POI_INT pole_uk_f[20]; (pozor, pozor, "magické" číslo!)
```

Anebo dynamicky:

```
UK_F_POI_INT *pole_uk_f;
```

Dále jako s klasickým plně dynamickým polem.

5.3 Funkce s proměnlivým počtem parametrů

Přichází k nám třešnička na dortíčku, pro ty, co nemají rádi třešničky, tak třeba knedlíček.

Jak už víme, parametry funkce jsou určeny v její hlavičce nebo prototypu, a to jejich typy a jmény. Před nedávnem jsem někde uvedl místo parametru tři tečky a řekl, že to je jakási věc jak říci "a tato funkce má ještě jakési další parametry". Ano, opravdu si můžeme udělat funkci s proměnlivým počtem parametrů. V C už to je[2], tak proč bychom si je nemohli definovat i my.

5.3.1 Definice funkce

```
int happypuppy(int whatever, ...);
```

Přičemž (podle normy ISO) platí, že před ... musí funkce mít alespoň jeden parametr.

[2]Například funkce `printf()`.

5.3.2 Práce s argumenty

Jak se uvnitř funkce přistupuje k argumentům? Nemají přece žádná jména.

Je třeba zahrnout soubor `stdarg.h`, tím dostaneme k dispozici několik maker sloužících právě pro operace s "proměnlivými" argumenty.

Nejprve musíme nadefinovat proměnnou typu `va_list`[3], která je ukazatelem na argumenty.

Poté nainicializujeme ukazatel voláním `va_start`.

Pomocí makra `va_arg` dostaneme parametry jeden po druhém – resp. první volání `va_arg` nám vrátí první argument, druhé volání druhý a tak dále.

Vůbec nemusíme projít argumenty až do konce, ale když jich chceme vzít více než jich je, dostaneme nějaké nesmysly.

Konec práce s argumenty označíme voláním `va_end`. U drtivé většiny rozumných překladačů toho není třeba, ale už kvůli standardu bychom to měli dodržovat.

Pokud z nějakého důvodu potřebujeme nějaký argument znovu a ukazujeme už "za něj", jednoduše zavoláme `va_end` následován `va_start` – a `va_arg` nám opět vrátí první argument a tak dále.

Můžeme mít více proměnných typu `va_list` a každý list přesouvat pomocí `va_arg` zvlášť[4].

5.3.3 Počet parametrů I

První pevný parametr funkce se velice často používá k předání informace o počtu proměnlivých argumentů – pokud voláte funkci s 10 parametry, ona má jako první parametr hodnotu určující počet variabilních parametrů, tak jí dáte jako první skutečný parametr proměnnou s hodnotou 9, aby věděla, kolik parametrů má číst. Tato konvence se používá, pokud mají mít parametry stejný typ.

Rozšířením této možnosti je následující případ. Předpokládejme, že funkce bude pracovat s mnoha objekty a my máme specifikovat vždy jednu vlastnost takového objektu.

Funkce neví, s kolika objekty bude pracovat – to jí řekneme pomocí proměnného počtu parametrů a zároveň jí předáme iniciální hodnoty specifikované vlastnosti objektů. Řada objektů má předdefinovaný způsob inicializace vlastnosti (pokud vlastnost nezadáme). Těchto objektů má být opravdu hodně, proto chceme specifikovat jen některé z nich. Použijeme bitmasku jako druhý pevný parametr a tím určíme které objekty mají být přeskočeny a inicializovány předdefinovaným způsobem.

Jak si to po sobě čtu, asi to nikdo nepochopí, takže tento příklad trochu zkonkretizuji – můžete totiž něco takového klidně někdy potřebovat.

Ještě předtím, asi jsem nerozebíral bitové pole, takže použiji bytepole – bitové pole si najděte v literatuře. Není to nijak převratná věc, dá se nahradit bytepolem, přičemž spotřebujeme osmkrát více místa, ovšem v příkladu nám to nevadí.

```
#define M_o_alloc(obj) \
   obj=(OBJ *) malloc(sizeof(OBJ));
#define M_GROW 10

int obj_sum(int num, int *mask, int mask_size, ...)
```

[3] *Variadic argument list.*

[4] Nebojte se, nic se v paměti nekopíruje, ukazují na stejné argumenty.

```
{
   va_list argums;
   typedef struct object {
      int value;
      struct object *next;
   } OBJ;
   OBJ *o_catalog=NULL;
   OBJ *o_start;
   int i;

   va_start(argums, mask_size);
   for (i=0;i<mask_size;i++) {
      if (o_catalog==NULL) {
         M_o_alloc(o_catalog);
         o_start=o_catalog;
      } else {
         M_o_alloc(o_catalog->next);
         o_catalog=o_catalog->next;
      }
      o_catalog->next=NULL;
      o_catalog->value = (mask[i]) ? \
      va_arg(argums, int) : M_GROW*i ;
   }

   for (o_catalog=o_start,i=0;
        o_catalog->next!=NULL;
        o_catalog=o_catalog->next)
              i+=o_catalog->value;

   i+=o_catalog->value;
   va_end(argums);
   return i;
}
```

Jako krásné **cvičení** si funkci projděte a zjistěte co vlastně dělá.

Už jste to zjistili? Dobrá, tak pro jistotu:

Funkce alokuje struktury které v sobě mají nějakou hodnotu typu `int` a ukazatel na následující objekt stejného typu v paměti – v `typedef struct object`. Uchovává si ukazatel na první objekt, aby se po alokování mohla vrátit a provést nějakou akci.

Při alokování každého objektu se podívá do příslušného místa v poli a pokud je pole obsazené 1 (*true*), přečte hodnotu do objektu z "proměnlivého argumentu". Pokud je položka v poli nepravdivá, přiřadí implicitní hodnotu, která závisí na pořadí objektu v řadě.

Po skončení alokace (je na konci pole) se funkce vrátí a začne hodnoty objektů sčítat.

Poznámky:

1. `malloc()` v makru, hmm hmm, nic moc, ale nechtělo se mi to vypisovat. Vypadá to sice hezky, ale v alokaci se často dělají chyby a dávat něco takového do makra je riziko. Je

to v malé funkci, tak budiž. Co je horší – není prováděn test na úspěšnost alokace – zda malloc() nevrátil NULL!

2. Počet proměnlivých parametrů je v tomto případě předáván úplně zbytečně, protože stejně závisí na poli. Dá se z něj spočítat.

3. Pro druhý průchod se dá lehce použít *o_start* ale za ušetření jednoho přiřazení bychom ztratili počáteční adresu.

4. "Díky" podmínce ve `for` se musí přičítání posledního objektu dodělávat. Často se na to zapomíná. Řešením je použít cyklus s podmínkou na konci ale potom nemáme logiku cyklu tak hezky kompaktní jako v závorkách za `for`.

5. Zvolená akce – sčítání – vůbec neukazuje jak takovouto věc využít. Promiňte, zrovna mě nic vhodného (dostatečně jednoduchého) nenapadlo.

6. Z hlavičky funkce není *vůbec* patrné, že velikost pole nemá být velikost, ale počet elementů a tak je to potřeba někde zdokumentovat, protože pokud to někdo zavolá s velikostí pole, funkce bude číst někde daleko za koncem pole.

7. Vůbec chybí komentáře, ale to je schválně – abyste se trochu zapotili. Má to mít výchovný charakter – vy snad po tomhle komentáře používat budete.

Příklad volání je uveden buďto v dodatcích nebo je dostupný někde na stránce, kde jste text stáhli[5].

5.3.4 Počet parametrů - II

Jedna z dalších konvencí – nepoužijeme jako pevný parametr počet proměnlivých parametrů ale hodnotu prvku, která bude proměnlivé hodnoty uzavírat. Jinými slovy, až procedura narazí na specifikovanou hodnotu, pozná, že je na posledním parametru. Problémem je, pokud se taková hodnota vyskytuje v parametrech dvakrát. Dejte na to pozor, ať vám funkce neskončí na prvním z nich. Typické použití je při předávání ukazatelů – poslední nastavíme na NULL a pak ve funkci testujeme na toto makro. Jakmile odchytíme NULL, víme, že jsme na konci.

5.3.5 Počet parametrů - III

Další možnost je použít řešení podobné `printf()` – pevný parametr bude formátovacím řetězcem specifikujícím kolik a jaké parametry jsme předali – je to asi jediné rozumné řešení v případě předávání parametrů různých typů.

Existují funkce pro vstup a výstup na základě proměnlivých argumentů – `vscanf()`, `vprintf()` a také další varianty. To už je ale opravdu všechno.

Cvičení: Zkuste si nadefinovat nějakou (třeba úplně jednoduchou) funkci používající proměnlivý počet parametrů. Stačí, když je budete postupně vypisovat na obrazovku.

[5]Poznámka: zatím tam není nic (25.7.2001)

Kapitola 6

Možnosti ladění a hledání chyb

Tahle kapitola je myšlena jako takový malý přídavek za to, že jste byli hodní. Rozeberu zde problémy, se kterými byste se mohli setkat. Tedy rozeberu ty, co mě napadnou. Pokud budete mít nějaký problém, který bude na úrovni zde popsaných problémů, napište mi a já sem jeho řešení přidám. Taktéž pokud budete mít nějaký již vyřešený problém, napište a bude sem přidán (společně se jménem autora).

6.1 Chybová hlášení od překladače

Poněvadž používám GNU C Compiler (*gcc*), rozeberu zde některé jeho nejčastější hlášky.

Chybová hlášení *gcc* se dělí na chyby (*errors*) a varování (*warnings*). Pokud překladač zahlásí na nějaké části kódu *warning*, znamená to, že danou část kódu pokládá za chybnou, ale je schopen se s touto chybou vyrovnat – v některých případech je varování myšleno hlavně tak, aby si programátor uvědomil, že použitý postup nemusí za všech okolností vyjít[1]. Po *warning* tedy překladač v překladu pokračuje a je na nás, jestli s kódem něco uděláme nebo ne. Doporučuji alespoň krátké zamyšlení nad kterýmkoliv *warningem* – v žádném případě je překladač nedává jen tak pro legraci. *Warning* poznáte tak, že je uvozen "warning:".

Naopak, pokud dojde při překladu k chybě, překladač nevydá výslednou binárku. Při některých chybách sice ještě chvilku pokračuje dál, ale to z toho důvodu, aby při jednom průchodu odhalil co nejvíce chyb a ne jen jednu. V každém případě je třeba si uvědomit, že kterákoliv chyba může překladač ovlivnit a tudíž další nalezené chyby chybami být nemusí, nebo naopak[2]. Prostě chybu opravíme a překlad spustíme znovu.

Snažte se nerozčilovat. Pokud vám překladač označí něco jako chybu, opravdu to chyba je, i když to někdy není na první pohled vidět. Čím déle budete překladač používat, tím více zkušeností poberete a budete chyby vyhledávat a odstraňovat rychleji.

V předchozím odstavci jsem uvedl vyhledávat, protože objevení chyby na 45. řádku neznamená, že její příčina nebude skryta třeba o deset řádků výš.

Hlášky *gcc* obsahují vždy soubor, ve kterém se chyba vyskytla, dále funkci, ve které se tato chyba vyskytla, příslušný řádek a konečně i chybovou hlášku, například:

```
cosig.c: In function ‘main’:
```

[1] T.j. v některých situacích může být chybný.

[2] Některé nemusí být odhaleny.

```
cosig.c:5: warning: implicit declaration of function ‘printf’
```

Někdy se mohou hlášky omezit jen na druhou část výše uvedeného hlášení[3] – to tehdy, když se chyba nevyskytla v žádné funkci, ale např. ještě před vstupem do funkce, v tom případě se před nimi zmíněný první řádek buďto vůbec nenachází[4], nebo je nahrazen řádkem “At top level:”

Příklad: (kompletní zdrojový kód souboru `cosig.c`)

```
-----začátek souboru---
<volná řádka>
main()
{
    int *cosi;
    if (cosi = 10) ;
    printf("ahoj\n");
    wer=12;
}

marinqw()
{
}
----konec souboru---
```

Nám po příkazu `gcc -Wall cosig.c` vyplázne:

```
cosig.c:3: warning: return-type defaults to ‘int’

cosig.c: In function ‘main’:
cosig.c:5: warning: assignment makes pointer
           from integer without a cast

cosig.c:5: warning: suggest parentheses around
           assignment used as truth value

cosig.c:6: warning: implicit declaration of
           function ‘printf’

cosig.c:7: ‘wer’ undeclared (first use in this
           function)
cosig.c:7: (Each undeclared identifier is
            reported only once
cosig.c:7: for each function it appears in.)

cosig.c:8: warning: control reaches end of
```

[3] Je vypuštěna první řádka se jménem funkce.

[4] Na začátku překladu, ještě nebyly ohlášeny chyby z žádné funkce.

```
              non-void function

cosig.c: At top level:
cosig.c:11: warning: return-type defaults to ‘int’

cosig.c: In function ‘marinqw’:
cosig.c:12: warning: control reaches end of
            non-void function
```

Dobrá, podívejme se na některá nejběžnější varování:

unused variable 'něco' ... upozornění na zbytečně definovanou proměnnou (proměnná není použita)

comparison between 'nějaký typ' and 'nějaký jiný typ' ... porovnáváte něco, co se typově liší, například `signed` číslo s `unsigned`, nebo pointer s `int` atp.

conversion from NaN to 'číselný typ' ... převádíte ne-číslo (*not-a-number*) na číslo

dereferencing 'void *' pointer ... jasné

implicit declaration of function 'jméno funkce' ... používáte funkci, kterou nemáte. Pravděpodobně jste zapomněli zahrnout nějaký hlavičkový soubor[5], nebo voláte vlastní funkci, vyskytující se níže v kódu – v takovém případě dopředu předřaďte její prototyp.

return-type defaults to 'int' ... neuvedli jste návratový typ funkce, překladač tedy počítá s `int`.

control reaches end of non-void function ... funkce, která vracela typ jiný než `void`, skončila předtím, než něco vrátila (měla něco vrátit a nevrátila nic).

assignment makes pointer from integer without a cast ... kvůli přiřazení do pointeru byl překladač nucen přetypovat `int` na pointer bez vašeho vědomí. Přetypujte. Podobně opačný příklad.

suggest parentheses around assignment used as truth value ... asi chcete provést test (`==`), ale použili jste jen jedno rovnítko – v testu. Pokud jste chtěli opravdu použít přiřazení a testovat jeho návratovou hodnotu, uzávorkujte přiřazení.

Teď ještě nějaké časté chyby:

'něco' undeclared (first used in this function)

(Each undeclared identifier is reported only once for each

function it appears in.) ... používáte proměnnou, kterou překladač nezná (zapomněli jste ji nadeklarovat nebo jste udělali překlep)

[5] Viz výše uvedený příklad, používá `printf()`, aniž by zahrnul `stdio.h`.

parse error before 'něco' ... špatný zápis. Selhal některý z parserů provádějících syntaktickou analýzu. Trochu lidštěji: máte někde překlep – např. na předchozím řádku jste zapomněli udělat středník a tak podobně.

'nějaká strašná nečitelnost': undefined reference to

'jméno funkce' collect2: ld returned 1 exit status ... voláte jméno funkce, která nebyla nalezena. Nejspíš jste udělali chybu ve jméně funkce (`printff()` atp.), v případě vaší vlastní funkce zkontrolujte definici.

conflicts with previous function/built-in declaration ... snažíte se pod již použitým jménem nadeklarovat druhou věc, což vám C nedovolí.

confused by earlier errors, bailing out ... *gcc* má dost a balí další procházení programu – t.j. za poslední ohlášenou chybu se v kódu nepouští, nejprve opravte vypsané chyby.

continue statement not within a loop ... použili jste příkaz `continue` mimo tělo cyklu

incorrect number of parameters ('číslo', should be 'číslo') ... voláte funkci s jiným počtem parametrů než je třeba.

too few arguments to function 'jméno funkce' NEBO

too many arguments to function 'jméno funkce' ... viz předchozí případ

Vyskytuje se ještě několik běžných hlášek, ale ty rozluštíte okamžitě z jejich znění.

6.2 Ladění

V první řadě se podíváme na některé nejčastější ladicí techniky.

V řadě druhé pak na hlavní ladicí prostředky v GNU systémech (Linux) a to *gcc*, *lint*, *gdb* a *ddd*.

6.2.1 Hlášení na `stderr`

Jedna z nejrychlejších a docela spolehlivých technik je výpis na standardní chybový výstup. Nelze použít vždy, ale v menších prográmcích se často hodí.

Myšlenka je zcela prostá. Odhadnete, kde by mohla být chyba, a vložíte (před ni nebo za ni) výpis pomocí `fprintf(stderr, ...)`. V tomto výpisu si necháte vytisknout hodnotu nějaké proměnné.

Typickým příkladem je kopírování kusů paměti, používání ukazatelů, cykly a větvení programu.

Musíte si ale uvědomit, že tato technika není v mnoha případech nejlepší. Například asi těžko budete vypisovat v každém průchodu cyklu iterační proměnnou, pokud se cyklus provádí 500x nebo více.

Třeba v cyklu (obecně v bloku příkazů) mnohokrát potřebujeme zjistit stavy několika proměnných v případě, že nějaká jiná (rozhodovací) proměnná nabyla určité hodnoty. Vložíme tedy `if`, a když je podmínka splněna, provedeme výpis sledovaných proměnných na `stderr`.

Tuto metodu lehce použijeme i k odhalení místa pádu programu[6]. Čas od času se stane,

[6] Pokud není moc složitý :-(

že váš program naprosto neočekávaně padá a vy nevíte kde. Jednou z možností odhalení místa pádu je vložení několika hlášek do programu. Za běhu potom víte, mezi kterými dvěma hláškami program popadal[7]. Samozřejmě k tomu musíte být schopni odhadnout průběh programu[8]. Druhá možnost je – alespoň u *gcc* – použít některé podpůrné prostředky, které vám i při pádu programu jsou schopny říci, na kterém řádku nastala chyba.

Tím, že znáte řádek chyby, ještě neznáte její příčinu. Důvod, proč program spadl na té které řádce může být skryt o mnoho řádek "výše" ve zdrojovém kódu, nebo třeba v datech. Právě proto si po odhalení kritického místa necháte před pádem vypsat obsah podezřelých proměnných na `stderr`.

Pokud vám program padá při výpisu proměnných, bude pravděpodobně chyba při jejich čtení[9].

Pokud nemáte možnost chybový výstup přímo sledovat[10], máte možnost si výstup přesměrovat, a to buďto přímo v programu volajícím – příkaz přesměrování chybového výstupu je `2>soubor` – nebo v programu vašem. Uděláte to naprosto jednoduše. Zavřete chybový výstup[11] a znovu jej otevřete např. do souboru. Touto obměnou se dostáváme k nejjednodušší verzi následující techniky.

6.2.2 Logovací soubory

Tato technika je rozšířením techniky předchozí. Nabízí několik možností navíc, ovšem dá daleko více práce ji využívat. Ve větších programech se vám ale jen vyplatí, neboť ji využijete k více účelům. Můžete ji použít i v případě forkování, komunikace přes sockety, pseudoterminály, s potomky, přes *www* rozhraní, ...

Slovo *log* znamená česky záznam. Logovací soubory jsou tedy záznamem o průběhu programu.

Půjdeme na to od nejjednodušší verze.

Vytvoříte si funkce na vytvoření logovacího souboru a přidání zprávy na konec. V této jednodušší verzi vytvářecí funkce soubor vytvoří a uzavře, zapisovací funkce jej otevře, přidá zprávu předanou parametrem na konec, a poté soubor opět zavře.

Na začátku programu zavoláte vytvářecí funkci a za běhu budete místo výpisu na chybový výstup volat funkci zápisovou s řetězcem jako parametrem.

Neustálému otevírání a zavírání souboru se lze vyhnout. Vytvářecí funkci upravíte tak, aby soubor neuzavírala a potom nemusíte při zápisu do souboru ani otevírat ani zavírat. Aby se nestalo, že při skončení programu se soubor neuzavře, vytvoříte krátkou funkci, která bude tento soubor zavírat a pomocí funkce `atexit()` ji nainstalujete – při konci programu by se vaše funkce měla zavolat.

Problém je, pokud váš program opravdu spadne[12], v takovém případě se funkce instalovaná pomocí `atexit()` nezavolá. Například po neoprávněném přístupu do paměti jsou nezapsané diskové buffery ztraceny. Provizorním řešením je volat funkci `fflush()` po každém zápisu do logovacího souboru, tím ale zase nezískáte téměř žádné zrychlení oproti otevírání/zavírání.

[7] Poslední, která se vytiskla a první, která se nevytiskla.

[8] Kudy ve zdrojovém kódu šel výpočet.

[9] Typicky ukazatel, který ukazuje kamsi kde nic není – špatně nebo vůbec inicializovaný.

[10] Program není pouštěn interaktivně, nechcete, aby chybový výpis viděl uživatel atp.

[11] Což je normální datový proud.

[12] *Abnormal program termination.*

Pro každý výstup do logovacího souboru voláte funkci. Než namítnete, že to je velké zpomalení, uvědomte si, že toho zase tak moc do souboru psát nebudete a navíc daleko pomalejší než volání funkce bude samotný zápis.

O rychlosti jsme si popovídali dost, teď k rozšíření výše uvedené metody.

Můžeme zavést více logovacích souborů a k volání zápisové funkce přidat parametr určující, kam se daná poznámka zapíše. Větší množství logovacích souborů zaručuje nejen přehlednost – při forkování procesu může potomek zapisovat do svého logovacího souboru[13] – nemusíte se bát, že se některé hlášky ztratí. Pokud budete potřebovat např. jeden logovací soubor pro (jednoho) rodiče a jeden pro všechny potomky, rodič může zapisovat beze strachu a mezi potomky musíte implementovat zámky – to už se ale sami podívejte např na funkci `flock()`. Dopředu vás varuji, že se systém od systému liší.

Více logovacích souborů se také využívá pro různé účely. Do jednoho jdou například hlášení programu a do druhého se eviduje činnost uživatelů apod.

Pro větší program je velice vhodné definovat makro např. M_LOGLEVEL, které nabývá předem stanovených hodnot – několika málo přirozených čísel. Tato čísla vyjadřují jak moc, nebo co se bude logovat. Např. 0 může znamenat žádný log, 1 log nejdůležitějších hlášek programu, 2 všechny hlášky programu a 3 hlášky + logfile pro činnost uživatelů. V průběhu programu, vždy když chcete nějakou hlášku zapsat, otestujete prvně *loglevel*. Například pro zápis důležité programové hlášky testujete, zda je *loglevel* roven nebo větší než jedna[14].

6.2.3 Závislé zprávy

Někdy možná budete potřebovat, aby se k vám informace o stavech proměnných dostaly v případech, které nebudete schopni předem odhadnout pomoci stavu nějaké proměnné či čehokoliv jiného. Budete potřebovat, aby vám informace k ladění potřebné dodal program za běhu, "na požádání".

Ať již je budete chtít na chybový výstup, do nějakého souboru, či snad do připraveného *socketu* nebo kamkoliv, musíte svému programu dát vědět, že je potřebujete.

Nejjednodušší způsob, bohužel ale nejméně použitelný, je pomocí specifických dat, která mu předáte mezi daty normálními. Pokud takovouto cestu nemáte, musíte mu dát vědět jinak.

Jednou z elegantních možností je výše zmíněný *socket*, nebo třeba roura (*pipe*), v případě *socketu* ale musíte do programu přidat funkce pro práci se *sockety*, navíc musíte mít další program, jak zprávy přes *sockety* posílat... prostě to není jednoduché. Roura je zase značně nespolehlivá, takže by to chtělo ještě něco jiného.

V GNU OS (a UNIXech) máte k dispozici techniku signálů a jejich ovladačů, která je dostatečně jednoduchá na implementaci ve vašich programech. Napíšete si ovladač signálu. Jeho jedinou funkcí bude změna nějaké proměnné, která bude ve vašem programu deklarovaná jako *volatile*. Instalujete tento ovladač. Ve chvíli, kdy váš program dostane signál, spustí se ovladač a změní výše uvedenou proměnnou.

Na programu zbývá občas si proměnnou testovat a zachovat se podle její hodnoty. Samozřejmě, mohli byste informace vypsat/zapsat do souboru už v ovladači signálu, jenže ovladač signálu by měl být co nejmenší, aby se snížila pravděpodobnost že během něj bude zavolán ovladač jiného signálu. Ale o tom si počtěte jinde.

[13] Pokud jej s nikým nesdílí.

[14] Tato hláška se musí zapsat v loglevelech 1, 2, 3.

Signály jsou jednoduchou a docela spolehlivou cestou jak takové “informace na požádání” poskytnout. Tuto možnost oceníte, když si uvědomíte, že program můžete takto dolaďovat za běhu a v situacích, kdy se k němu jinak absolutně nemůžete dostat (ani *debuggerem*). Uvedu příklad:

```
#include <signal.h>
/* atomicka promenna (globalni), muzeme
 * inicializovat
 */
   volatile sig_atomic_t printdebug;
/* v pripade nouze pouzijte misto typu
 * sig_atomic_t treba int
 */

   ... program ...

/* ovladac signalu */
void sig_debug(int signumber)
{
   printdebug = 1;
}

... program ...

/* instalace ovladace */
   signal(SIGUSR1, sig_debug);
/* tim jsme ovladac nastavili na signal
 * sigusr1 (lepsi je pouzit konstantu
 * nez tam davat cislo - na ruznych OS se lisi).
 * Cisla zjistite kill -l
 * nebo man signal -> man 7 signal
 * vyse uvedena funkce vracela typ
 * sighandler_t (meli bychom jej pouzit i jako
 * navratovy typ naseho ovladace) - a to
 * predchozi ovladac.
 * Kazdopadne vam doporucuji neco si pred
 * pouzivanim signalu precist
 * napriklad manualove stranky
 */
... program ...
```

Ještě pár poznámek k signálům. Doporučuji v programu použít signály SIGUSR1 a SIGUSR2, ostatní signály již přidělený význam mají. Mohli byste je sice využít, ale držte se výše uvedených. Signál programu zašlete pomocí příkazu `kill`, např. pokud chcete poslat signál 10 (což je většinou SIGUSR1), napíšete

```
kill -10 <PID>
```

Kde <PID> je identifikační číslo procesu – to zjistíte pomocí příkazu `ps`. Ten vám už ale vysvětlovat nebudu, podívejte se na manovou stránku.

6.2.4 *gcc*

Ne, nezbláznil jsem se, vím, že *gcc* je překladač. Chci k němu přidat jen malou poznámku. *Gcc* má hafo přepínačů a jeden z nich bych vyvýšil nad ostatní a doporučil vám k neustálému používání – je to přepínač *Wall* a znamená vypiš všechna standardní varování. Používá se:

```
gcc -Wall program.c
```

6.2.5 *lint*

Nebudu zde příliš popisovat používání *lint*, protože existuje spousta velice dobré dokumentace, pouze stručně nastíním, co je a co umí.

Lint[15] se používá pro kontrolu zdrojových textů ještě před tím, než jdou do překladače. Je schopen najít drtivou většinu běžných problémů – a to jak těch větších, tak dost bezvýznamných. Má spoustu parametrů, které určují jak se má chovat, tedy říct, na které věci vás má upozorňovat.

Jednoduše – upozorní vás na (pravděpodobně špatně) zapsané operace s pointery, na zahazování výsledků funkcí, na neodalokování dynamické paměti[16], na nevyužité exportované proměnné a funkce, na podivné předávání parametrů funkcím, na používání neinicializovaných proměnných a daleko, daleko více věcí.

Příklad používání:

```
lclint +forcehints moje.c
```

Můžete jej samozřejmě pustit na více souborů najednou. No a pokud ho použijete poprvé a na ne úplně malý prográmek, vypíše vám chyb více než se vejde na jednu obrazovku, takže si zapněte "slabší" kontrolu a výstup přesměrujte do souboru:

```
lclint +forcehints -weak moje.c >lint_errors_moje.txt
```

Víc dokumentace.

6.2.6 *gdb*

Gdb = GNU *Debugger* (GNU odvšivovač)

Gdb je velice silný prostředek pro ladění programů. Opět, má velice dobrou dokumentaci a proto zmíním jen stručně jeho funkce.

Gdb umí pouštět, zastavovat a krokovat váš program. Umí zobrazovat a měnit hodnoty různých proměnných...

Gdb umí jak pracovat s "čerstvým" programem, tak zjistit jeho stav když spadl[17] za pomoci *core* souboru. Dokonce umí pracovat i s běžícím procesem – napojit se na něj – ale to jsem zatím nikdy nezkoušel.

[15] Nověji *lclint*.
[16] Např. pomocí `malloc()` ve funkci, pointer na paměť se nevrací.
[17] Pokud program spadne a máte dobře nastavené limity – pro *core* je to `ulimit -c`.

Pokud chceme nějaký program ladit pomocí *gdb*, musíme do binárního souboru přidat při překladu ladicí informace – například

```
gcc -o vystup -g vstup.c
```

Vytvoří binárku se jménem “vystup”, která v sobě má zahrnuty ladicí informace – děje se to právě parametrem *g*.

Gdb je řádkově orientovaný nástroj – to znamená, že po jeho spuštění obdržíte *prompt* a píšete příkazy.

K dispozici je docela slušný help – pod příkazem “help”.

Soubor specifikujeme většinou už při spuštění a to jako

```
gdb binarka
```

Gdb si sám zjistí potřebný zdrojový program – pokud je dostupný. Jinak jej přidáte na příkazovém řádku.

Nejzákladnější příkazy jsou:

list n ... vylistuje +- 5 řádků kolem řádky n ve zdrojovém kódu

b <číslo> ... nastaví *breakpoint* na řádku <číslo>

b fce ... nastaví *breakpoint* po vstupu do funkce fce

run ... začne vykonávat program

s ... vykoná následující příkaz, pokud je to volání funkce, zanoří se dovnitř

n ... vykoná následující příkaz, pokud je to volání funkce, vykoná ji najednou

delete ... vymaže *breakpointy*

info break ... vypíše *breakpointy*

print <var> ... vypíše hodnotu proměnné <var>

set <var>=<value> ... nastaví hodnotu proměnné <var> na <value>

quit ... ukončí *gdb*

Gdb umí spoustu dalšího, například co se *breakpointů* týče, umí se na nich zastavit pokud je splněna podmínka, dočasně je vypnout... umí sledovat změny proměnných a další a další. Tohle všechno byla jen motivace, další viz dokumentace.

6.2.7 *ddd*

Ddd je grafický *front-end* pro *gdb* a mnohé další *debuggery.* Nejprve bude kritika a poté smetana.

Pro ty, kteří jsou zlenivělí a vyhovuje jim myškoidní grafický klikátor je tento program ideální. Nechci jej nijak ponižovat, v C umí přesně tolik co *gdb*. Pokud jej chcete používat pro jazyk C, musíte samozřejmě mít nainstalován *gdb*.

Jeho výhodou oproti *gdb* je možnost dělání okýnek a také větší uživatelská přítulnost. Umí například zobrazit hodnotu nějaké proměnné tím, že na ni najedete kurzorem myšky a hodnota se vám zobrazí. Dá se v něm také lépe orientovat. Příkazy jsou uspořádané v menítkách a je k dispozici i několik ikon. Drtivá většina příkazů je k dispozici na klávesovou zkratku, což považuji za důležitý kladný rys.

Co je (podle mě) ovšem nejdůležitější, jsou dvě věci:

1. Máte k dispozici příkazový řádek z *gdb*, kde nejen vidíte jaké příkazy *ddd gdb* předhazuje, ale můžete je zde přímo psát, čímž máte možnost dostat se tam, kam to pomocí menu/ikonek nejde.

2. Pokud se naučíte s *ddd* pracovat přes menítka a horké klapky v C, budete umět *debuggovat* jakýkoliv jazyk který *ddd* zvládá[18].

Ddd samotný se dá tak krásně a intuitivně používat a navíc má tak dobrou dokumentaci a help, že se o něm nebudu více rozepisovat. Samozřejmě je to GNU program a tak je k dispozici bezplatně.

6.3 Co dělat když...

Tato kapitolka podává návod jak něco (jednoduchého) udělat. Neříkám, že je to vždy jediná nebo nejlepší cesta, ale bude fungovat. Rozeberu nejčastější potřeby.

Má fungovat jako rychlá pomoc – nemusíte prohledávat celý text.

6.3.1 soubory

Kapitola o souborech – 3.2

Používáme hlavičkový soubor `stdio.h`

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

Proměnné, pomocí kterých pracujeme se soubory jsou ukazateli na typ `FILE`:

```
FILE *soub1, *soub2;
```

[18] Nebo spíše jakým *debuggerům* může sloužit jako *front-end* – tedy *gdb, python debugger, perl debugger, dbx, xdb, ladebug, jdb.*

6.3.1.1 Otevření souboru

(kap. 3.2.3):

```
soub1 = open(cesta, mod);
```

cesta ... řetězec – může být jenom jméno souboru

mod ... řetězec – určuje v jakém módu budeme soubor otevírat (co s ním budeme provádět):

- **a** ... append – budeme psát za konec (přidávat), ukaz. na konec
- **a+** ... append + číst
- **w** ... zapisovat od začátku – přepíše existující soubor
- **w+** ... w + číst
- **r** ... pro čtení, ukazatel na začátek
- **r+** ... r + možnost psaní

Při neúspěchu vrací funkce `NULL` pointer.

Bezpečné otevření souboru:

```
if ( (soub2 = fopen("ahooj.txt","r")) == NULL ) {
   printf("Soubor se nepodařilo otevřít!\n");
   exit(-1);
}
```

6.3.1.2 Uzavření souboru

```
int fclose(FILE *ukaz);
```

Při neúspěchu vrací nulu. Použití:

```
if ( fclose(soub2) != 0 ) {
   printf("Soubor se nepodařilo uzavřít!\n");
   exit(-1);
}
```

6.3.1.3 Vstup a výstup souborů

(kap. 3.2.4)

Znakový I/O Funkce `fputc()` a `fgetc()`.

Mají dva parametry. Prvním je znak a druhým ukazatel na soubor. Při neúspěchu vrací `EOF`.

Přečtení celého souboru pomocí `fgetc()`:

```
char znak;
FILE *soubor;
soubor = fopen("ahooj.txt","r");
while (fgetc(znak, soubor) != EOF) ;
```

Formátovaný I/O Funkce `fprintf()` a `fscanf()`.

Prvním parametrem je ukazatel na soubor, potom formátovací řetězec a poté další parametry. Vrací počet zapsaných položek. Formátovací řetězce jsou stejné jako u `printf()` a `scanf()`.

```
fprintf(soubor, "%s\t%d\n", "5*8 je", 5*8);
```

Řádkový I/O Funkce pro vstup a výstup řádku.

```
int fputs(const char *s, FILE *stream);
```

Výstup jednuho řádku do souboru. Vrací 0 při úspěchu, použití:

```
fputs("To je ale věta.", soubor);
```

```
char *fgets(char *s, int size, FILE *stream);
```

Vstup řádku ze souboru do bufferu *s*. *size* je velikost bufferu. Načte nejvíce `size-1` znaků, čtení je tedy bezpečné. Vrací pointer na *s*, v případě neúspěchu – nebyl načten ani jeden znak – `NULL`. Použití:

```
#define BUFFER 20
   char *ret;
   ret = (char *) malloc((BUFFER+1)*sizeof(char));
   if ( fgets(ret, (BUFFER+1), soubor) == NULL ) {
      fprintf(stderr, "Chyba při čtení ze souboru\n");
      exit(-1);
   }
```

6.3.1.4 Další operace

(kap. 3.2.8)

Testování konce souboru – `feof()`. Parametrem je ukazatel na soubor, pokud je ukazatel na konci, vrací nenulovou hodnotu – vrací `int`.

Testování chybového příznaku – `ferror()`. Podobně jako `feof()`.

Čištění chybového příznaku `clearerr()`. Parametrem je ukazatel na soubor, nevrací nic (`void`).

Vylití bufferů na disk – `fflush()`. Parametrem je ukazatel na soubor. Vrací int, 0=úspěch. Pokud jí předáme `NULL`, vylije všechny proudy.

Pohyb v souboru – funkce `fseek()`, `fgetpos()` a `fsetpos()`. Podívejte se na ně blíže v kapitolce 3.2.8, tady jen nejjednodušší příklady:

Na začátek:

```
fseek(soubor, 0, SEEK_SET);
```

Na konec:

```
fseek(soubor, 0, SEEK_END)
```

6.3.2 alokace

(kap. 3.5)

Pokud potřebujeme za běhu programu získat paměť, použijeme většinou funkci `malloc()` – kap. 3.5.5. Vrací ukazatel na typ `void`, který si vždy přetypujeme podle potřeby. Jejím argumentem je počet alokovaných byte, který lehce spočítáme – vynásobíme počet našich objektů počtem *byte*, které každý z nich zabírá.

Alokujeme pole dvaceti struktur *moje_structura*[19]:

```
moje_structura *pole;
pole = (moje_structura *) malloc(20*sizeof(moje_structura));
```

Řetězec o délce DELKA:

```
char *ret;
ret = (char *) malloc((DELKA+1)*sizeof(char));
```

Operátor `sizeof()` vrací velikost daného objektu.

Uvolnění paměti – funkce `free()`. Vrací `void` a jejím parametrem je ukazatel na `void`. Pokud jí předložíme ukazatel na něco jiného než `void`, zvládne to také.

`free(ret);` nebo `free((void)ret);`

6.3.3 makra

(kap. 4.4)

Typicky nadefinujeme makro jako

```
#define NAZEV_MAKRA HODNOTA_MAKRA
#define M_POCET 50
#define M_TEXT "Sel do obchodu."
```

Typ se nikde nespecifikuje. Makro použijeme například:

```
cosi=malloc(sizeof(int)*M_POCET);
printf(M_TEXT);
```

Funkční makro definujeme

```
#define NAZEV_MAKRA(PAR_A,...) POPIS_MAKRA
#define op(a, b) ( ((a)+(b)) * ((a)-(b)) / (b) )
```

Typ opět nikam nepíšeme. Použití:

```
c=op(50,n);
```

Takový zápis se rozvine do

```
c=((50)+(n)) * ((50)-(n)) / (n);
```

za pomoci preprocesoru ještě dříve, než se k němu dostane překladač. Je třeba dávat si velký pozor na typy a vůbec na to, co děláme.

[19] Čili ukazatel na typ *moje_structura*.

6.3.4 řetězce

(kap. 3.7.1)

Zápis konstantního řetězce:

```
char konst[] = "Opice na opici";
```

Alokace prostoru pro neinicializovaný řetězec:

```
char retez[20];
```

19 znaků plus znak `\0`;

Dynamicky - ukazatel na znak:

```
char *ret;
```

Alokace viz 6.3.2.

6.3.4.1 Standardní I/O

(kap. 2.7.2)

Funkce `getchar()` a `putchar()`.

`getchar()` vrací jeden znak, je bez parametrů, naproti tomu `putchar()` znak vypisuje a má parametr typu `char`.

`getchar()` čeká na <enter>. Tomuto čekání se lze vyhnout několika způsoby, jsou však dost silně závislé na OS. V GNU OS máte tři možnosti:

- použít *ncurses* (nejpohodlnější) – `curses.h`,
- použít *ioctl* (nedoporučuji),
- použít *termios* – `termios.h` (taky docela dost práce).

Doporučuji *ncurses*, váš program bude 100% přenositelný na systémy s *ncurses*.

Použití `getchar()` a `putchar()`:

```
char a;
a = getchar();
putchar(a);
```

6.3.4.2 Funkce `printf()` a `scanf()`.

(kap. 2.7.3)

Slouží pro formátovaný výstup. Funkce vracejí počet položek, které se podařilo vytisknout/přečíst.

```
printf("%d:%d", 10, 15);
```

Vytiskne 10:15

Základní specifikace proměnných v řetězci:

%d ... decimální celé číslo (int)

%f ... reálné číslo

%s ... řetězec

```
scanf("%d %f", &a, &b);
```

Načte celé číslo do proměnné “a”, přeskočí mezeru[20] a poté načte reálné číslo do proměnné “b”.

Formátovací řetězec je podobný jako u `printf()`, další parametry nejsou proměnné, ale ukazatele na proměnné.

Funkce `sprintf()` má oproti `printf()` přidán jeden parametr, a to ukazatel na řetězec. Do tohoto řetězce se zapíše výsledek formátování (který by se při `printf()` vypsal na standardní výstup). Přidaný parametr je na začátku:

```
char *novy_ret;
novy_ret = (char *) malloc(DELKA*sizeof(char));
sprintf(novy_ret, "%d:%d", 10, 15);
/* vytiskne 10:15 do retezce novy_ret */
```

6.4 Nejčastější chyby

Souhrn nejčastěji vyráběných chyb.

6.4.1 Syntaktické chyby

Syntaktické chyby jsou relativně příjemné, protože je odhalíme již při překladu. Překladač vypíše něco jako “syntax error” nebo “parse error before” a číslo řádky, kde byla chyba odhalena.

Patří sem všechny překlepy. Doslova typickým příkladem je zapomenutí uzavírací uvozovky v řetězci. Pokud používáte nějaký editor, který umí *syntax highliting*[21], pravděpodobně takovou chybu ani neuděláte.

Dalším notorickým příkladem jsou závorky veškerých typů – například je více otevíracích závorek než zavíracích apod. Pokud použijete editor, který umí “inteligentní” automatické zarovnávání, drtivé většiny takovýchto chyb se také nedočkáte[22].

Uhodli jste. Doporučuji vám nějaký vhodný editor, například “vim”[23].

Víc není co řešit. Syntaktické chyby jsou jasné.

6.4.2 Porovnávání

= a ==

Samostatné rovnítko je přiřazovací příkaz a pokud jej použijeme v testu, dostaneme něco jiného než jsme chtěli. Některé kompilátory mají volbu pro vyhledávání takovýchto chyb. Problémem je, že ne vždy je přiřazení v testu chybou[24].

---

[20] Bílé znaky.

[21] Barevně odlišovat určité kusy programu – např. editor “vim”.

[22] Např. editor “vim”, umí i vyhledávat závorku odpovídající té, na které je kurzor.

[23] Víte, ve kterém editoru jsem psal původní text téhle příručky? Já vim.

[24] Pokud chcete provést přiřazení a nechcete vidět *warning,* přiřazení ještě zvlášť uzávorkujte.

6.4.3 Chyby v deklaracích/definicích, přetypování.

Zde se vyskytují nejčastěji zapomenuté středníky, ale to jsou syntaktické chyby.

Jednou z chyb je definice nového typu pouhým přejmenování typu existujícího. Samozřejmě to bude fungovat, ale dostaneme pouze zmatek.

Velké množství chyb se vyskytuje také v makrech (funkčních). Je třeba poctivě zkontrolovat každé makro a uvážit případy, kdy selže a dá špatný výsledek. Pokud si myslíte, že byste jej teoreticky mohli během psaní programu zavolat s takovými "špatnými" parametry, raději makro nepoužijte – určitě si tyto případy nezapamatujete.

Přetypovat můžete v podstatě cokoliv na cokoliv, ovšem pouze tehdy, když tuto hodnotu odněkud taháte. Nemůžete jen tak přetypovat celou proměnnou – změnit její typ – ten byl určen již při definici proměnné.

Když přetypováváte na ukazatel na typ, nezapomeňte na hvězdičku.

6.4.4 Chyby ve struktuře programu

Chyby první odrůdy tohoto typu vznikají už při návrhu programu – špatné rozdělení problému na dílčí problémy. Je lepší volat padesát funkcí než mít molochovitou funkci `main()` na sto obrazovek.

Druhá odrůda se týká zapomenutých bloků za příkazem `if` a příkazy cyklů. Při psaní programu napíšete za `if` jen jeden příkaz, takže nepotřebujete blok – tedy závorky `{, }` vynecháte. Při pozdější úpravě přidáte ještě jeden příkaz, ale blok ne a program se najednou chová podivně, neboť druhý příkaz se provede nezávisle na podmínce. Pokud nevěříte vlastní paměti, vždy použijte blok.

6.4.5 Funkce

Funkce naštěstí provádějí typovou kontrolu a tou je většina běžných problémů odhalena již za překladu. Pamatujte také, že při předávání můžete předávané hodnoty přetypovávat.

Lokální proměnné ve funkcích zakrývají globální proměnné. Pokud máte lokální proměnnou "a" a globální proměnnou "a", budete mít přístup pouze k lokální proměnné. Ve většině případů jsou globální proměnné zlo[25] a proto se jim vyhýbejte co to půjde.

Při volání funkce se hodnoty parametrů kopírují z paměti na zásobník. V případě větších množství dat proto použijte ukazatel, volání funkce nebude tak pomalé.

Funkce nemůže měnit hodnoty svých parametrů. Pokud chcete, aby nějaká data změnila, předejte ukazatel na ně.

Pokud chcete předávat ukazatelem a mít jistotu, že se data nezmění, uvoďte parametr v hlavičce funkce klíčovým slovem *const* – viz 4.1.3.

6.4.6 +1 chyby

Pamatujte, že pole je indexováno od nuly a proto když definujete

```
int a[20];
```

[25]Nebudu to blíže rozepisovat, prostě mi pro jednou věřte.

tak poslední prvek je `a[19]`. V případě řetězců navíc máte v posledním prvku `\0`.

Jazyk C neprovádí kontrolu indexů (kvůli rychlosti), a proto jsou takovéto chyby hodně nepříjemné.

6.4.7 Ukazatele a paměť

Hvězdička má docela nízkou prioritu, nezapomeňte závorkovat. Při přístupu do struktury – přes ukazatel – použijte operátor `->`.

Před porovnáním ukazatelů je přetypujte na (`void *`).

`malloc()` vrací (`void *`), přetypujte.

Dynamicky naalokovaná paměť ve funkci se neuvolní s jejím koncem. Taktéž pokud přiřadíte ukazateli adresu jiné paměti a předchozí paměť neodalokujete, adresu původní paměti ztratíte a nebudete ji schopni uvolnit jinak než ukončením programu.

Index

Literatura

[1] *Učebnice jazyka C*, Pavel Herout, Kopp, ISBN 80-85828-21-9, II.díl: 80-85828-50-2

[2] *Jazyky C a C++ podle normy ANSI/ISO* – kompletní kapesní průvodce, Dirk Louis, Petr Mejzlík, Miroslav Virius, Grada, ISBN 80-7169-631-5

[3] Dokumentace GNU C library (glibc) – *The GNU C Library Reference Manual*

[4] Manuálové stránky glibc