Úvod do programování

Úvod

Tento text vznikl pro potřeby výuky předmětu Úvod do programování na FEI VŠB-TUO. Slouží k získání přehledu o základních konceptech programovacího jazyka C. Není však plnohodnotnou náhradou za poslechy přednášek a návštěvy cvičení a programovat vás (stejně jako žádný jiný text) nenaučí, toho lze dosáhnout pouze opakovaným zkoušením a řešením různých úloh. Studentům tedy silně doporučujeme, aby přednášky a cvičení navštěvovali a hlavně aby se věnovali programování doma, alespoň několik hodin týdně.

V tomto textu naleznete stručný úvod o programování, překladu a ladění programů, nastavení prostředí k editaci zdrojového kódu, a zejména popis základních konstrukcí jazyka *C* (proměnné, podmínky, cykly, funkce, ukazatele, pole, řetězce, struktury atd.) spolu se sadou úloh k procvičení jednotlivých témat. Pomocí ikony vlevo nahoře můžete v textu rychle vyhledávat, pokud potřebujete najít informace o konkrétním tématu.

Několik poznámek k textu:

- Tento text neslouží jako kompletní průvodce jazyka C. Pro takovýto účel lze doporučit některý knižní titul, např. Učebnice jazyka *C* od Pavla Herouta nebo přímo standard jazyka C99 .
- Jelikož je předmět UPR zaměřen na vývoj v operačním systému Linux, tak ukázky kódu a příkazů v terminálu předpokládají použití tohoto operačního systému (konkrétně distribuce `Ubuntu`).
- Tento text je psán česky, nicméně primárním jazykem programování (celosvětově) je angličtina. Přeložené pojmy, které mají zavedené anglické názvy, budou v tomto textu uvedeny v závorce *kurzívou*.
- V tomto textu naleznete různé ukázky *C* kódu. Některé z nich můžete sami upravovat a dokonce i spustit rovnou v prohlížeči pomocí ikony v pravém horním rohu kódu. Ukázky budou pro zjednodušení používat názvy v češtině, nicméně jakmile už nebudete v programování úplní nováčci, silně vám doporučujeme psát zdrojové kódy v angličtině.
- Pokud v textu naleznete gramatickou či faktickou chybu nebo budete mít jakoukoliv zpětnou vazbu k obsahu či formě textu, dejte nám prosím vědět na tento e-mail nebo vytvořte issue na GitHubu .

Autory textu jsou Jan Gaura , Dan Trnka a Kuba Beránek.

Historii změn tohoto studijního textu můžete naleznout v jeho GitHub repozitáři.

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Programování

j. sady příkazů pro počítač, který slouží k vyřešení nějakého konkrétního **problému**. Problémem se zde myslí nějaká úloha, kterou chceme vyřešit. Takovéto úlohy obsahují nějaký (počítačem zpracovatelný) vstup, například:

- pohyb myši
- stisk klávesy
- zvuk z mikrofonu
- textový soubor na disku

a k nim určený výstup, například:

- vykreslení obrazce či textu na monitoru
- zapsání dat do souboru na disku
- odeslání informací přes síť

Aby počítačový program korektně řešil nějakou úlohu, tak musí na všechny validní vstupy vrátit správný výstup. Pokud vstup neodpovídá zadání, tak by měl program vrátit rozumnou chybovou hlášku (a ne spadnout anebo ještě hůře pracovat s nevalidními daty). Postup pro řešení nějaké úlohy daný jasně definovanými kroky se nazývá **algoritmus**. Zápisu (algoritmu) v nějakém konkrétním programovacím jazyce se pak říká **implementace**.

Zde je příklad úloh, které se během semestru naučíte vyřešit pomocí jazyka *C*:

```
Spočítej průměr seznamu čísel.
```

nebo

```
Načti obrázek z disku, změň jeho velikost a ulož ho do jiného souboru.
```

Řešením podobných úloh si osvojíte základy programování a budete poté moct řešit zajímavější úlohy, jako je například tvorba počítačové hry nebo aplikace komunikující přes internet.

Programovací jazyky

Z pohledu počítače je program sekvence příkazů (nazývaných **instrukce**), které může počítač vykonat k vyřešení nějakého problému. Abychom mohli počítači říct, co má vykonávat, potřebujeme mu příkazy zadat ve formě, které bude rozumět. Ač se to možná nezdá, tak počítače umí vykonávat pouze velmi jednoduché příkazy. V podstatě umí pouze provádět aritmetické a logické operace (sečti/odečti/vynásob/vyděl) s čísly a manipulovat (ukládat, kopírovat, přesouvat) s těmito čísly v paměti. Veškeré složitější úkoly, jako třeba vykreslení obrázku na obrazovku, zapsání textu do dokumentu nebo simulace světa v počítačové hře je výsledkem kombinací tisíců či milionů takovýchto jednoduchých instrukcí.

Zde je ukázka jednoduchého programu, který zdvojnásobí číslo pomocí příkazů `MOV` a `ADD`:

```
MOV EAX, 8
ADD EAX, EAX
```

Pokud bychom programy psali pomocí těchto jednoduchých příkazů, tak by bylo složité se v nich vyznat, obzvláště, pokud by obsahovaly stovky, tisíce nebo dokonce miliony takovýchto příkazů. Ideálně bychom chtěli programy zapisovat v přirozeném jazyce (`Vykresli čtverec na obrazovku`, `Zapiš text do dokumentu`), nicméně tomu počítače nerozumí a je velmi náročné jej převést na správnou sekvenci příkazů pro počítač, protože jazyky, které používáme, jsou často nejednoznačné a nemají jednotnou strukturu.

Jako kompromis tak vznikly **programovací jazyky**, které umožňují zápis programů ve formě, která je lidem srozumitelná, ale zároveň ji lze relativně jednoduše převést na příkazy, které je schopen počítač provést. Převodu programu zapsaného v programovacím jazyce na počítačové instrukce se říká **překlad** (*compilation*) a programy,

které tento překlad provádějí, se nazývají **překladače** (*compilers*). Později si ukážeme, jak takovýto překladač použít k překladu kódu.

Zde je ukázka části programu v jazyce *C*:

```
while (je_tlacitko_zmacknuto(MEZERNIK)) {
    posun_nahoru(postava);
}
```

I někdo, kdo se s jazykem *C* nikdy nesetkal, může z tohoto kusu kódu zhruba odvodit, co asi dělá, pokud ho přečte jako větu. Tento program však může být převeden na stovky až tisíce počítačových instrukcí a z takového množství příkazů už by bylo složité odvodit, k čemu je program určen.

Jazyk *C*

Existuje nespočet programovacích jazyků, například Python, Java, C#, PHP či Javascript. Každý z nich má své výhody a nevýhody a záleží na konkrétním problému, který je třeba vyřešit, pro zvolení vhodného programovacího jazyka.

V tomto kurzu se budeme zabývat pouze programovacím jazykem **C**. Tento jazyk vytvořili Dennis Ritchie a Ken Thompson v laboratořích firmy Bell v roce 1972, tedy již před téměř 50 lety, a za tu dobu nedočkal mnoha výrazných změn.

I když pro něj v dnešní době asi nenaleznete mnoho pracovních nabídek a není primární volbou pro tvorbu webových či mobilních aplikací, vyplatí se mu rozumět a umět ho používat, a to hned z několika důvodů:

- Jazyk *C* lze použít na téměř všech existujících platformách a je tak velmi univerzálním jazykem. Téměř veškerý existující software obsahuje kusy kódu v jazyce *C*. Operační systémy (Linux, OS X, Windows, Android, iOS), prohlížeče (Chrome, Firefox, Edge), multimediální programy (Photoshop, Powerpoint, Word, BitTorrent), hry (World of Warcraft, Quake, Doom, Call of Duty, League of Legends, DOTA 2, Fortnite), vestavěná zařízení (mikročipy, pračky, řídící jednotky vesmírných letadel nebo aut). Všechny tyto věci jsou buď z části anebo zcela poháněné jazykem *C*.
- Je to jednoduchý jazyk, který neobsahuje velké množství funkcionalit, které lze naleznout ve většině modernějších jazyků. Díky tomu se dá naučit za jeden semestr.
- Jeho úroveň abstrakce není o mnoho výše než základní počítačové instrukce. Při výuce *C* tak lze zároveň pochopit, jak funguje počítač a operační systém. Díky tomu lze také při správném zacházení psát velmi efektivní programy (to ale nicméně není obsahem tohoto kurzu).
- **Syntaxe** (způsob zápisu) jazyka *C* ovlivnila velké množství jazyků, které vznikly po něm. Jakmile se ji naučíte, tak budete schopni rozumět syntaxi většiny současných nejpoužívanějších jazyků (C++, C#, Java, Kotlin, Javascript, PHP, Rust, ...).

Jazyk *C* má samozřejmě také řadu nevýhod. Vzhledem k jeho stáří a omezené sadě funkcionalit je často značně pracnější a zdlouhavější pomocí něho dosáhnout stejného výsledku než u modernějších programovacích jazyků. Nevede také programátory za ručičku – při psaní programu v jazyce *C* je velmi jednoduché udělat chybu, která může způsobit v lepším případě pád programu, v horším případě může běžící program poškodit tak, že začne vydávat chybný výstup nebo se začně chovat nepředvídatelně.

Tyto chyby se můžou projevit jen někdy, nebo jenom na určité kombinaci hardwaru či operačního systému, a programátor na ně není často nijak upozorněn a musí je najít ručně zkoumáním zdrojového kódu. Podobný typ chyb je také nejčastějším zdrojem bezpečnostních děr ve všech možných softwarech, které (jak už víme) téměř vždy obsahují alespoň část kódu napsaného v "Céčku".

Zde je vybraný seznam populárních programů napsaných v jazyce *C*, které jsou **open-source**, takže si jejich zdrojový kód můžete prohlédnout a v případě potřeby i modifikovat:

- Linux (operační systém)
- Quake III (počítačová hra)
- git (verzovací systém)
- PHP (překladač/interpret jazyka PHP)
- OBS Studio (streamovací software)

Paměť

Počítače si potřebují ukládat výsledky výpočtů do paměti, aby je později mohly opět načíst a pracovat s nimi. Je mnoho typů paměti, s kterými lze pracovat, nejběžněji se setkáme s tzv. operační pamětí (**RAM**). RAM znamená Random-access Memory, tedy paměť s náhodným přístupem. To znamená, že počítač může do paměti šahat v libovolném pořadí a na libovolném místě, kde je to potřeba.

Reprezentace hodnot v paměti

Počítačová paměť uchovává informace v buňkách, které obsahují jedno číslo, které může obsahovat 256 různých hodnot. To vychází z toho, že informace je reprezentována **bity**, jednotkou informací, která může nabývat pouze dvě hodnoty - pravda (*true*) nebo nepravda (*false*). Každá buňka paměti obsahuje jeden **byte**, neboli 8 bitů.

Pracuje se zde s dvojkovou (binární) soustavou, pokud tedy máme k dispozici n bitů, tak pomocí nich můžeme reprezentovat 2^n hodnot. Např. s dvěma bity můžeme reprezentovat 4 různé hodnoty (00, 01, 10, 11). Více o binární soustavě a bytech se dozvíte v předmětu Základy číslicových systémů (ZDS).

I když paměť vždy obsahuje čísla v dvojkové soustavě, je důležité si uvědomit, že význam těmto číslům přiřazujeme my, tedy programátoři a uživatelé počítače. Pokud je v paměti číslo **65**, tak může reprezentovat například:

- počet získaných bodů studenta (interpretujeme jej jako celé nezáporné číslo)
- písmeno `A` v nějakém dokumentu (interpretujeme jej jako znak v kódování ASCII)
- tmavě šedý pixel (interpretujeme jej jako barvu)

Hodnotu **255** uloženou v *bytu* paměti můžeme například vnímat jako celé nezáporné číslo (*unsigned integer*) **255**, anebo jako celé číslo se znaménkem (*signed integer*) **-1** v dvojkovém doplňku.

Čísla v paměti sama o sobě nemají žádný význam, záleží pouze na tom, jak je my, a obzvláště naše programy, interpretují a jaké operace nad nimi provádějí.

Adresování paměti

Abychom se mohli odkazovat na hodnoty v paměti, tak musíme mít možnost rozlišit jednotlivé buňky od sebe. Toho dosáhneme pomocí **adresy**. Paměť je adresována tak, že každá paměťová buňka (každý *byte*) má číselnou adresu od 0 do velikosti paměti (nevčetně). Velmi zjednodušeně řečeno, pokud máte RAM paměť o velikosti 8 GiB (8589934592 "bajtů"), tak můžete adresovat buňky od 0 do 8589934591[1].

[1] Programy běžně nemají přístup k celé paměti počítače (mimo jiné z bezpečnostních důvodů). Váš operační

systém používá tzv. **virtuální paměť**, která každému běžícímu programu přiděluje určité rozsahy paměti, s kterými může pracovat. Více se dozvíte v předmětu Operační systémy.

Pokud byste programovali počítač přímo pomocí instrukcí, tak mu můžete dát například instrukci `Nastav byte na adrese 58 na hodnotu 5` nebo `Přečti 4 byty začínající na adrese 1028`. Při programování v *C* ovšem často budou adresy skryté na pozadí a bude se o ně starat překladač, my se budeme na konkrétní úsek paměti obvykle odkazovat jménem, které mu přiřadíme.

Nastavení prostředí

Abyste mohli programovat v *C*, musíte si nainstalovat, nakonfigurovat a naučit se používat sadu programů. V této kapitole je stručný popis toho, jak si nastavit operační systém Linux, textový editor k psaní programů, překladač pro překlad z jazyka *C* do spustitelného souboru a také jak řešit chyby při psaní programů.

Linux

Jak už bylo zmíněno v úvodu , v UPR budeme psát a spouštět programy v operačním systém Linux. Je tak nutné, abyste si na svém počítači tento operační systém zprovoznili.

Pokud používáte operační systém OS X, tak teoreticky Linux instalovat nemusíte, stačí si nastavit překladač `gcc`.

Pokud používáte operační systém Windows, tak pro použití Linuxu můžete využít jeden z následujících tří možností.

Návod pro práci s terminálem na Linuxu můžete najít např. zde . Tahák pro příkazy terminálu najdete zde .

Windows Subsystem for Linux (doporučeno)

WSL je systém, který umožňuje nainstalovat Linux pod operačním systémem Windows. Jakmile jej nainstalujete, budete mít k dispozici Linuxový terminál (`bash`) a budete moct používat překladač gcc a editor Visual Studio Code . Výhoda tohoto řešení je, že pro použití Linuxu nemusíte restartovat počítač ani zapínat virtuální stroj, Linux je v podstatě jenom "další aplikace" pod Windows.

Návod pro zprovoznění WSL spolu s prostředím pro vývoj v jazyce *C* naleznete zde . Při instalaci WSL používejte distribuci `Ubuntu 20.04`.

Virtualizovaný Linux

Linux můžete také používat ve virtualizované podobě pomocí virtuálního stroje . Připravili jsme pro vás tzv. obraz virtuálního stroje, který obsahuje již nastavený Linux, konkrétně `Ubuntu 20.04`, se vším potřebným pro předmět UPR.

Abyste jej mohli použít, tak si nejprve musíte nainstalovat virtualizační program VirtualBox. Poté si předpřipravený obraz stáhněte, otevřete ho ve VirtualBoxu a potvrďte import s výchozím nastavením.

Virtuální počítač poté bude možné spustit z programu VirtualBox. Uživatelské jméno i heslo je `student`.

Nativní instalace Linuxu

Nejspolehlivější variantou použití Linuxu je nainstalovat si ho přímo "na železo", tj. bez virtualizace. Můžete jej například nastavit v režimu dual boot , kdy se při startu počítače můžete rozhodnout, zdali se nabootuje do Windows (či jiného operačního systému) nebo do Linuxu. Pokud jste s Linuxem nikdy nepracovali, tak doporučujeme použít Linuxovou distribuci Ubuntu ve verzi `20.04`.

Vývojové prostředí

Abychom mohli přeložit a spustit nějaký program, musíme ho obvykle nejprve zapsat do jednoho nebo více souborů ve formě tzv. **zdrojového kódu** (*source code*). K usnadnění tohoto procesu existují **textové editory** a **vývojová prostředí** jako například `MS Visual Studio`, `QtCreator`, `JetBrains CLion`, `CodeBlocks`, `Visual Studio Code`, `vim`, `emacs` apod. Tyto programy usnadňují psaní kódu pomocí zvýrazňování syntaxe, automatizace překladu, spouštění a testování programů a také správy projektů.

Na cvičeních UPR budeme používat editor `Visual Studio Code`, který je dostupný zdarma. Níže je stručný návod k jeho použití. Při programování se hodí detailně znát a efektivně využívat editor, který používáte, ale pro začátek nám budou stačit naprosté základy.

Instalace potřebných rozšíření (pomocí terminálu)

VSCode podporuje programovací jazyky pomocí rozšíření, po první instalaci VSCode tak nejprve musíme nainstalovat potřebná rozšíření pro jazyk C. V terminálu spusťte tyto příkazy:

```
$ code --install-extension ms-vscode.cpptools
```

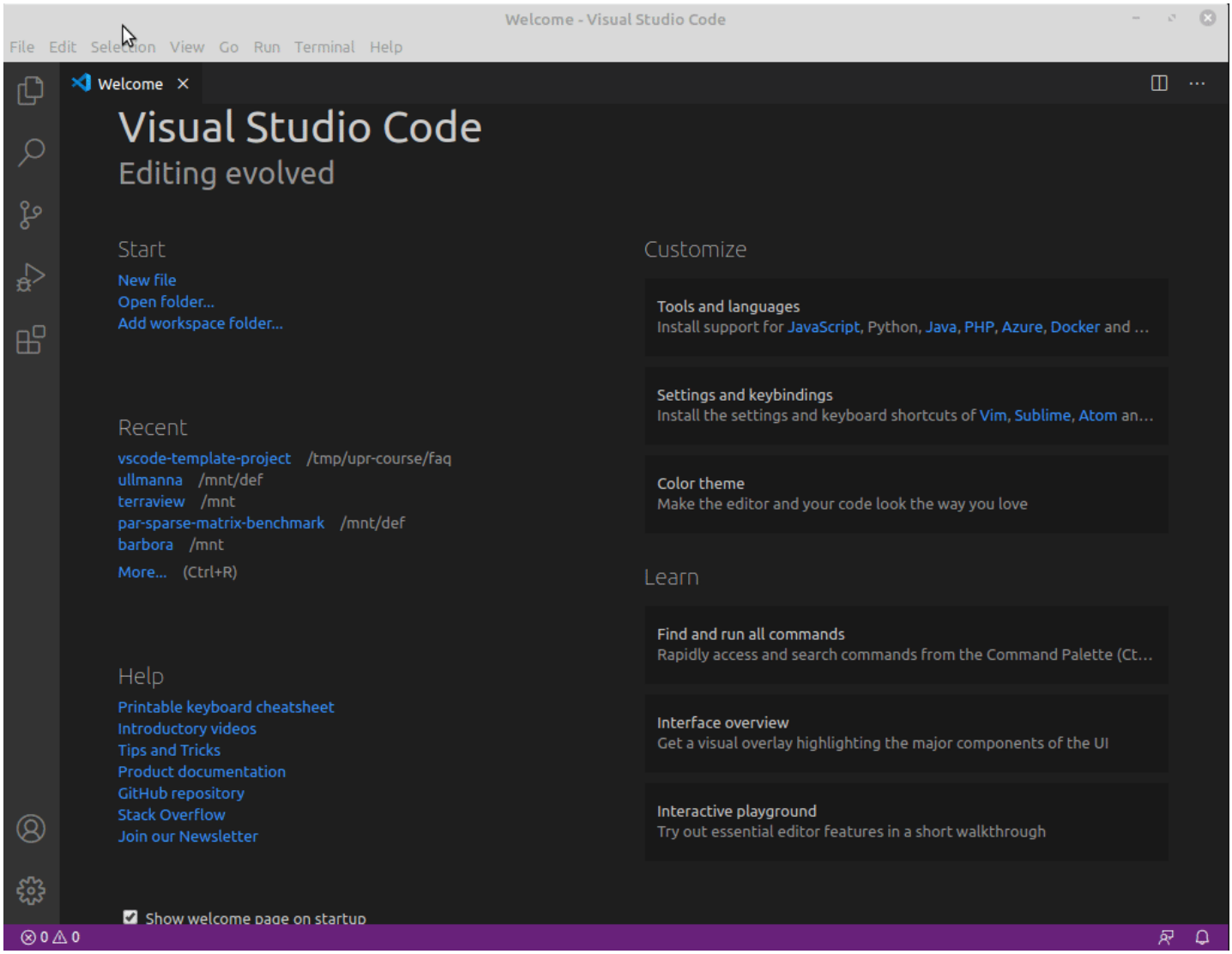
> Návod pro práci s terminálem na Linuxu můžete najít např. zde . Tahák pro příkazy terminálu najdete zde .

Instalace potřebných rozšíření (pomocí uživatelského rozhraní)

1. Otevřete obrazovku rozšíření (`Ctrl+Shift+X` nebo spusťte akci `Install Extensions`)
2. Vyhledejte rozšíření `C/C++` a nainstalujte ho

Ukázka nastavení projektu

Jako vzorový projekt můžete použít tuto šablonu.



Užitečné zkratky

- Spustit program - `F5`
- Naformátovat kód - `Ctrl + Shift + I`
- Zobrazit vyhledávač akcí - `Ctrl + Shift + P`

Překlad programu

Pro překlad programu z jazyka *C* do **spustitelného** (*executable*) souboru budeme používat jiný program, kterému se říká překladač. Překladačů jazyka *C* existuje celá řada, my budeme využívat asi nejpoužívanější překladač pro Linuxové systémy s názvem **GCC** (GNU Compiler Collection).

Překladač `gcc`, spolu s dalšími potřebnými nástroji, můžete na Ubuntu v terminálu nainstalovat pomocí následujícího příkazu:

```
$ sudo apt install build-essential
```

Překlad prvního programu

Ještě než si ukážeme, jak vlastně programovací jazyk *C* funguje, tak zkusíme přeložit velmi jednoduchý *C* program do spustitelného souboru a spustit jej. Vytvořte soubor s názvem `main.c` a nakopírujte do něj následující *C* kód (později si vysvětlíme, jak tento kód funguje):

```
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

> Tento program se nazývá `Hello world`, jelikož tento text vypíše na obrazovku. Podobný jednoduchý program je obvykle tím prvním, co programátor v nějakém novém programovacím jazyce vytvoří.

Nyní otevřete terminál (`Ctrl + Alt + T` v Ubuntu) ve složce s tímto souborem, spusťte program `gcc` a předejte mu cestu k tomuto souboru:

```
$ gcc main.c -o program
```

Tímto příkazem řeknete "Gécécéčku", aby přeložil zdrojový soubor `main.c` a uložil výsledný spustitelný soubor do souboru `program`. Pokud byste přepínač `-o <nazev souboru>` nepoužili, tak se vytvoří spustitelný soubor s názvem `a.out`.

> Na Windowsu spustitelné soubory mají obvykle příponu `.exe`, na Linuxu to však není běžnou praxí a spustitelné soubory typicky žádnou příponu nemají.

Pokud chcete nyní program spustit, stačí v terminálu zadat (relativní) cestu k danému spustitelnému souboru.

```
$ ./program
Hello world!
```

Program by měl na výstup vytisknout text `Hello world!`.

Tipy pro práci s příkazovou řádkou

Pro představu je k dispozici ještě shrnující video:

Jak překlad probíhá?

Překlad programu bude detailně vysvětlen v sekci o linkeru.

Ve zkratce: Překlad programů probíhá ve dvou hlavních fázích: **překlad** (*translation*) a **linkování** (*linking*). Dohromady se oboum těmto krokům také říká **kompilace** (*compilation*).

Při překladu překladač vezme každý *C* zdrojový soubor, který mu předložíme, a samostatně jej přeloží do tzv. **objektového souboru** (*object file*). Takovýto soubor obsahuje již přeložené instrukce pro procesor, ale není sám o sobě spustitelný, tj. nejedná se o program, ale pouze o přeložený binární kód.

Jakmile jsou všechny zdrojové soubory přeloženy do objektových souborů, tak přichází na řadu další program, tzv. **linker**, který tyto objektové soubory spojí dohromady, propojí je dle potřeby, případně k nim připojí externí knihovny a na konci vytvoří finální spustitelný soubor, který lze poté spustit.

Když použijete program `gcc` způsobem, jaký jsme si ukázali výše, tak se na pozadí spustí překladač a poté i linker a oba dva tyto kroky se tak provedou automaticky. Je ale možné provést je i separátně:

```
$ gcc -c main.c      # vytvoří objektový soubor main.o
$ gcc main.o -o main # slinkování souboru main.o
```

Ladění programů

Tato sekce slouží k řešení často se vyskytujících problémů při programování v C. Pokud váš program padá při běhu nebo se nechová tak, jak má, tak v něm nejspíše máte nějakou chybu (tzv. **bug**). Proces hledání chyby, která způsobuje pád nebo špatné chování programu se pak nazývá **ladění** (*debugging*).

Chyby při překladu programu

Pokud váš program nelze přeložit a překladač vypisuje nějakou chybovou hlášku, tak máte v zápisu programu nějakou chybu, obvykle v syntaxi, tedy zápisu kódu. Je dobré si danou chybovou hlášku pořádně přečíst, obvykle se odkazuje na relativně přesné místo, kde máte kód špatně, a někdy dokonce i nabízí řešení, jak problém vyřešit.

Při překladu můžete dostat například následující chybovou hlášku:

```
main.c: In function ‘main’:
main.c:2:2: error: ‘a’ undeclared (first use in this function)
    2 |  a = 0;
```

Tato konkrétní chyba byla způsobena tím, že byla použitá proměnná bez její předchozí deklarace. Pokud chybě nerozumíte, zkuste ji nejprve vygooglit, ideálně pouze část, která není konkrétně závislá na podobě vašeho projektu. Nemá cenu googlit `main.c:2:2`, protože tento text je závislý na tom, jak jste si pojmenovali své soubory, ostatní programátoři nejspíše mají jiné názvy souborů. V případě této chyby by tedy bylo lepší googlit text `error: undeclared (first use in this function)`.

Může se stát, že překladač vypíše více chybových hlášek zároveň, i když chyba v programu je pouze jedna. Zkuste scrollovat výstupem hlášek nahoru, abyste zjistili, která chyba byla vypsána jako první, zbytek výpisu může být "planý poplach".

Pokud se vám nedaří chybu vygooglit, tak kontaktujte svého cvičícího.

Při překladu můžete použít dodatečné přepínače, při jejichž použití vydá překladač více varování o možných problémových místech ve vašem kódu:

```
$ gcc -Wall -Wextra -pedantic main.c -o program
```

Chyby při běhu programu

Pokud váš program padá při běhu, můžete zkusit následující způsoby ladění:

Address sanitizer

Tento nástroj modifikuje váš program tak, aby dokázal detekovat značné množství chyb při jeho běhu, a pokud nějakou chybu najde, tak váš program okamžitě ukončí a popíše, k jakému problému došlo.

```
$ gcc -g -fsanitize=address main.c -o program
```

Jakmile takto přeložený program spustíte a dojde k nějaké chybě, tak bude její popis vypsán na výstup.

Pokud se chyba opraví těsně po svém vzniku, je to mnohem jednodušší, než když se chyba projeví až později v úplně jiné části kódu. **Doporučujeme tak vždy používat Address Sanitizer při vývoji programů v C**. Ušetříte si tak spoustu času a námahy při ladění chyb.

Logování

Jedním z nejjednodušších způsobů, jak se dozvědět, co se v programu děje, je jednoduše tisknout hodnoty zajímavých proměnných na výstup programu. Pokud přidáte takovýto výstup na různá místa v kódu, můžete pak podle výstupu zpětně rekonstruovat, co se při běhu programu dělo.

Krokování

Pro interaktivnější zkoumání chování programů je možné je tzv. **krokovat**. K tomu je potřeba nástroj, který umí program pozastavit při jeho běhu a zobrazit uživateli, co se v něm děje. Takovéto nástroje se nazývají **debuggery**. Při krokování se program zastaví na určitém místě (řádku) v kódu, a programátor pak může zkoumat hodnoty proměnných a spouštět program řádek po řádku.

Pro vás je nejjednodušší použít krokování integrované ve VSCode:

- Klikněte na sloupeček vlevo od čísla řádku, na kterém chcete, aby se program zastavil. Objeví se tam červené kolečko (tzv. **breakpoint**).
- Spusťte program s laděním (`F5`). Program by se na řádku s breakpointem měl zastavit.
- Ve sloupci `Variables` v levé části VSCode můžete prozkoumat hodnoty proměnných.
- Pomocí příkazu `Step Over` (`F10`) program vykoná následující řádek a poté se opět zastaví. Pokud nechcete přeskakovat volání funkcí, použijte `Step Into` (`F11`).

VSCode používá pro ladění vašeho programu debugger `gdb`. Pokud ho chcete použít manuálně, návod můžete najít například zde .

Programování v *C*

V této kapitole naleznete popis základních konstrukcí jazyka *C*, které jsou základními stavebními kameny pro tvorbu programů. Ke každému tématu je k dispozici také úloh. Pokud úlohy zvládnete vypracovat, tak budete mít jistotu, že jste dané téma pochopili a můžete se posunout dále. Pokud nezvládnete úlohy splnit, tak můžete mít s dalšími koncepty problém. Pokud nebudete stíhat, tak kontaktujte svého cvičícího.

Před přečtením této kapitoly si přečtěte sekci o paměti.

Zde je přibližný seznam témat, které si během semestru ukážeme. Pořadí témat probíraných na cvičení a přednáškách se může od tohoto seznamu lišit, tento text je určen spíše jako "kuchařka", ve které se můžete k jednotlivým tématům vracet, abyste si je připomněli. Text je nicméně psaný tak, aby se dal zhruba číst v uvedeném pořadí bez toho, aby používal pojmy, které zatím nebyly vysvětleny.

Základní témata

- Úvod - základní syntaxe a komentáře
- Příkazy a výrazy - jak provádět výpočty
- Proměnné - jak něco uložit a načíst z paměti paměti
- Datové typy - jak pracovat s daty v paměti
- Řízení toku - jak se rozhodovat a provádět akce opakovaně
- Funkce - jak opakovaně využít a parametrizovat opakující se kód
- Ukazatele - jak sdílet data v paměti a pracovat s adresami
- Pole - jak jednotně pracovat s velkým množstvím dat
- Text - jak v programech pracovat s texten
- Struktury - jak vytvořit vlastní datové typy
- Soubory - jak číst a zapisovat soubory
- Modularizace - jak rozdělit program do více zdrojových souborů
- Knihovny - jak využít existující kód od jiných programátorů

Všechny tyto koncepty jsou velmi univerzální a v tzv. imperativních programovacích jazycích jsou v podstatě všudypřítomné. Jakmile se je jednou naučíte, tak je budete moct využívat téměř v libovolném populárním programovacím jazyku (Java, C#, Kotlin, Python, PHP, Javascript, Rust, C++ atd.).

Navazující aplikovaná témata

- TGA - jak vytvořit obrázek
- GIF - jak vytvořit animací
- SDL - jak vytvořit interaktivní grafickou aplikaci či hru
- Chipmunk - jak simulovat fyzikální procesy

Základy syntaxe

C je (programovací) jazyk a jako každý jazyk má svá pravidla, které je nutno dodržovat. Například v češtině musíme dodržovat určitá pravidla a zvyklosti, abychom byli schopni výsledný text pochopit. Věty `jsme, M y máma, táta a` nebo `.o dku d! ty z, jsi` nedávají smysl, protože obsahují interpunkční znaménka na špatných místech, větné členy jsou ve špatném pořadí a některá slova obsahují mezery na místech, kam nepatří. Stejně tak v jazyce *C* můžete velmi jednoduše napsat program, kterému překladač nebude rozumět a překlad poté skončí se syntaktickou chybou

(*syntax error*). Na syntax *C* si musíte postupně zvyknout, poté už podobné chyby budete schopni snadno vyřešit.

Zde je asi nejkratší možný program v jazyce *C*:

```
int main() {
    return 0;
}
```

V programu výše je pouze funkce s názvem `main`. Funkce si popíšeme později, prozatím budeme psát kód vždy do funkce `main` (před `return 0;`). Jednotlivé prvky programu si postupně vysvětlíme v následujících sekcích, prozatím si však všimněte, že **bílé znaky** (*whitespace*)[1] jsou obvykle překladačem ignorovány. Například

[1] Bílé znaky jsou (neviditelné) znaky, které reprezentují mezery v textu, tj. odřádkování, mezerník, tabulátor atd.

```
int


main()                {


    return         0;
}
```

reprezentuje úplně stejný program. Nicméně asi sami uznáte, že pokud bychom s bílými znaky nakládali takto nerozvážně, tak by zdrojový kód byl pro lidi špatně čitelný. Proto doporučujeme formátování provádět automaticky ve VSCode pomocí zkratky `Ctrl + Shift + I`, ať nad ním nemusíte přemýšlet.

Bílé znaky nicméně nejsou ignorovány úplně na všech místech. Například v řetězcích jsou bílé znaky brány jako součást textu. Nemůžete také rozdělovat mezerami názvy (např. `in t` nebo `ma in` z programu výše by způsobily chybu při překladu).

Komentáře

Abychom mohli v následujících sekcích popisovat kusy kódu, ukážeme si teď **komentáře**. Jedná se o text ve zdrojovém kódu, který je určen pro programátory, ne pro překladač, který je zcela ignoruje. Bez komentářů bychom nemohli do zdrojového kódu dodávat poznámky, protože překladač by jinak měl snahu je interpretovat jako *C* kód. Komentáře v kódu obvykle poznáte snadno, protože je váš editor bude vykreslovat jinou barvou než zbytek kódu.

V *C* existují dva typy komentářů:

- Řádkové komentáře - pokud do kódu napíšete `//`, tak vše za těmito lomítky až do konce řádku se bude brát jako komentář.

  ```
  // komentář 1
  int main() {
      // komentář 2
      return 0; // komentář 3
  }
  ```

- Blokové komenáře - pokud do kódu napíšete `/*`, tak bude jako komentář označen všechen následující text, dokud nedojde k ukončení komentáře pomocí `*/`.

```
int main() {
    /* zde je komentář
zde taky
a tady taky */
    return 0;
}
```

Ze začátku je asi jednodušší používat řádkové komentáře, ve VSCode můžete použít klávesovou zkratku `Ctrl + /` pro zakomentování/odkomentování řádku kódu. Pokud vám přijde nějaký kus kódu komplikovaný, tak si k němu zkuste dopsat komentář, který vysvětlí, co má daný kód dělat. Porozumíte tak kódu snáž, až se k němu např. za měsíc vrátíte.

Klíčová slova

Klíčová slova (*keywords*) jsou vestavěné názvy, kterým překladač přiřazuje speciální význam. V textovém editoru je typicky poznáte tak, že budou zabarvená jinou barvou. Například v tomto kódu jsou `int` a `return` klíčová slova:

```
int main() {
    return 0;
}
```

Během semestru se postupně naučíte, k čemu se jednotlivá klíčová slova používají. Jejich kompletní seznam můžete najít například zde .

Speciální znaky

Při programování (jak už v *C*, tak i v jiných jazycích) budete používat spousty symbolů, které běžně asi často nevyužíváte (například `[`, `]`, `{`, `}`, `<`, `>`, `=`, `%`, `#`, `&`, `*`, `;`, `\`, `"`, `'`). Obzvláště pokud pro programování budete používat českou klávesnici, je dobré si ze začátku najít nějaký tahák (např. tento), abyste nemuseli pokaždé zdlouhavě vzpomínat, na které klávese se daný znak nachází.

Formátování kódu

Už víme, že překladač ignoruje bílé znaky a celkové formátování kódu. Nicméně programátorům obvykle velmi záleží na tom, jaké má kód odsazení, zarovnání, závorkování atd. Existuje mnoho stylů, pomocí kterých můžete kód formátovat. Například programátoři se dokážou běžně pohádat o tom, zda složené závorky na začátku bloku psát na stejném:

```
if (...) {

}
while (...) {

}
```

nebo novém řádku:

```
if (...)
{
}
while (...)
{
```

```
}
```

Jaký styl formátování použijete je na vás, nicméně obecně platným pravidlem je, že byste se měli držet ve svých programech jednotného stylu a nemíchat více stylů dohromady.

Vykonávání programů

Jak už víme, programy jsou sekvence příkazů pro počítač, který je provádí instrukci po instrukci (resp. řádek po řádku). Jakmile počítač vykoná jeden řádek vašeho programu, tak skočí na řádek níže, dokud nedojde na konec programu. Aby počítač věděl, kterou instrukci má provést jako první, tak mu musíme říct, kde má začít. K tomu přesně slouží funkce (pojmenovaný blok kódu) se speciálním názvem `main`:

```
int main() {
    // ZDE
    return 0;
}
```

Výše zmíněný program se po překladu a spuštění začne vykonávat na prvním řádku funkce `main`, a jakmile provede všechny řádky, tak program skončí. Tento program je v podstatě prázdný, takže se pouze zapne a vypne. Prozatím budeme veškerý kód psát dovnitř funkce `main`, mezi složené závorky (`{`, `}`) a před řádek `return 0;` (tedy na místo komentáře `ZDE`). Později si vysvětlíme, jak tato funkce funguje, prozatím to berte tak, že v programu vždy musí funkce `main` být, aby počítač věděl, odkud začít vykonávání kódu.

Příkazy

Programy v *C* se skládají z **příkazů** (*statements*). Příkaz říká počítači, co má provést, na mnohem vyšší úrovni než instrukce - jeden C příkaz může být přeložen překladačem na desítky instrukcí pro procesor. Existuje mnoho různých typů příkazů, které naleznete v následujících sekcích.

Výrazy

Jak už vyplává z jeho názvu, nejpřirozenější a hlavní funkcí počítače je něco počítat. Jedním ze základních konstrukcí jazyka *C* (i jiných programovacích jazyků) tak je možnost počítat různé hodnoty. Něco, co se dá vypočítat (tak, aby výsledkem byla nějaká hodnota), se nazývá **výraz** (*expression*). Příkladem asi nejjednoduššího výrazu je číslo, např. `5`. Takovýto výraz již není nutné dále vyhodnocovat, jeho hodnota je prostě `5`. Pokud v programu použijete přímo hodnotu nějakého čísla (popř. něčeho jiného, jak uvidíme později), tak se takový výraz označuje jako **literál** (*literal*).

V *C* můžeme s výrazy provádět různé operace pomocí **operátorů**. Můžeme například použít operátor `+` s dvěma výrazy, čímž vznikne složitější výraz: `5 + 5`, který se v programu vyhodnotí na hodnotu `10`.

Výpis výrazů

Abyste si ze začátku mohli jednoduše zobrazit hodnoty výrazů, tak si ukážeme kód, pomocí kterého můžete vypsat text na výstup programu (do terminálu). K výpisu můžete použít příkaz

```
printf("<text>");
```

Text, který vložíte mezi uvozovky (`"`) se vypíše na výstup programu [2]:

[2]Tento kód můžete modifikovat i spustit přímo v prohlížeči. Stačí kliknout na ikonu vpravo nahoře nebo stisknout `Ctrl+Enter`.

```
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

Abyste `printf` mohli použít, musíte na začátek programu vložit řádek `#include <stdio.h>`. Tento řádek i `printf` zatím berte jako "black box", později si vysvětlíme, jak přesně fungují.

V zadaném textu můžete používat určité speciální znaky. Například sekvence znaků `\n` způsobí, že na výstupu dojde k **odřádkování** (*newline*), po kterém se text začne vypisovat na dalším řádku:

```
#include <stdio.h>

int main() {
    printf("Prvni radek\nDruhy radek");
    return 0;
}
```

Abyste mohli tisknout hodnoty výrazů, můžete použít **zástupné znaky** (*placeholders*). Pokud chcete vypsat *číselnou* hodnotu na výstup programu, stačí v textu použít zástupný znak `%d`, za uvozovky přidat čárku a doplnit výraz na místo určené komentářem:

```
#include <stdio.h>

int main() {
    printf("Cislo: %d\n", /* Hodnota tohoto výrazu se vypíše na výstup */ 1);
    return 0;
}
```

Když chcete vypsat například výsledek vyhodnocení výrazu `10 + 5`, tak stačí napsat: `printf("%d\n", 10 + 5);` a na výstup programu by se měl vypsat text `15`.

Pokud chcete vytisknout více hodnot, tak prostě řádek s `printf(...);` zkopírujte a na uvedené místo vložte jiný výraz. Počítač provádí programy řádek po řádku, odshora dolů. Uhodnete, co se vypíše na výstup po přeložení a spuštění následujícího programu?

```
#include <stdio.h>

int main() {
    printf("%d\n", 1);
    printf("%d\n", /* tady vložte výraz */);
    return 0;
}
```

Cvičení: Zkuste si na místo komentáře doplnit několik výrazů (např. `5 + 8`, `8 * 3`, `12 * (2 + 3)`), přeložit program, spustit ho a podívat se, co vypíše na výstup, abyste si vyzkoušeli vyhodnocování výrazů.

Datové typy

Každý výraz má svůj datový typ, který udává, jak je hodnota výrazu v programu interpretována a také jaké operace má smysl nad výrazem dělat. Více o datových typech a operátorech se dozvíte v sekci Datové typy .

Vedlejší efekty

Pokud chcete pouze vypočítat výraz ("jen tak"), mimo nějaký příkaz, stačí za něj dát středník. Tím ze samostatného výrazu uděláte příkaz:

```
1 + 1; // vypočte se `2`, výsledek se na nic nepoužije
```

Toto má smysl dělat pouze u výrazů, které mají nějaký **vedlejší efekt** (*side effect*), který způsobí, že při provádění výrazu se v programu něco změní. Jinak by výraz sám o sobě byl vypočten, ale nic dalšího by se nestalo. O výrazech, které umí produkovat vedlejší efekty, se dozvíte v pozdějších sekcích.

Příkazy vs výrazy

Jakmile se budete postupně učit o jednotlivých konstrukcích jazyka C, je důležité uvědomit si, jaký je rozdíl mezi výrazem (něco, co se dá vypočítat) a příkazem, pomocí kterého počítači říkáme, aby něco (s nějakým výrazem) udělal (například vypsal ho na výstup, zapsal do paměti atd.).

Proměnné

Aby programy mohly řešit nějaký úkol, tak si téměř vždy musí umět něco zapamatovat. K tomu slouží tzv. **proměnné** (*variables*). Proměnné nám umožňují pracovat s pamětí intuitivním způsobem - část paměti si pojmenujeme nějakým jménem a dále se na ni tímto jménem odkazujeme. Proměnné můžou uchovávat libovolnou hodnotu a také ji v průběhu programu měnit. Příklady použití proměnných:

- Ve webové aplikaci si číselná proměnná pamatuje počet návštěvníků. Při zobrazení stránky se hodnota proměnná zvýší o 1.
- Ve hře si číselná proměnná pamatuje počet životů hráčovy postavy. Pokud dojde k zásahu postavy nepřítelem, tak se počet životů sníží o zranění (*damage*) nepřítelovy zbraně. Pokud hráč sebere lékárníčku, tak se počet jeho životů opět zvýší.
- V terminálu si proměnná reprezentující znaky pamatuje text, který byl zadán na klávesnici.

Proměnné jsou jedním z nejzákladnějších a nejčastějších stavebních kamenů většiny programů, během semestru se s nimi budeme setkávat neustále. Není tak náhodou, že jedním z nejzákladnějších příkazů v *C* je právě vytvoření proměnné. Tím řekneme počítači, aby vyčlenil (tzv. **naalokoval**) místo v paměti, které si v programu nějak pojmenujeme a dále se na něho pomocí jeho jména můžeme odkazovat[1].

[1] O tom, jak přesně tato alokace paměti probíhá, se dozvíte později v sekci o ukazatelích .

Definice a platnost

Takto vypadá **definice** (vytvoření) jednoduché proměnné s názvem `vek`:

```
int vek;
```

Jakmile proměnnou nadefinujeme, tak z ní můžeme buď číst anebo zapisovat paměť, kterou tato proměnná

reprezentuje, pomocí jejího názvu (zde `vek`).

Proměnná je platná (lze ji používat) vždy od místa definice do konce **bloku**, ve kterém byla nadefinována. Bloky jsou kusy kódu ohraničené složenými závorkami (`{ a }`):

```
int main() {
    int a;

    {
        // zde je platné pouze `a`
        int b;
        // zde je platné `a` i `b`
    } // zde končí platnost proměnné `b`

    // zde je platné pouze `a`

    return 0;
} // zde končí platnost proměnné `a`
```

Oblast, ve které je proměnná validní, se nazývá *(variable) scope* .

Datový typ

`int` před názvem proměnné udává její datový typ, o kterém pojednává následující sekce . Prozatím si řekněme, že `int` je zkratka pro `integer`, tedy celé číslo. Tím říkáme programu, že má tuto proměnnou (resp. paměť, kterou proměnná reprezentuje) interpretovat jako celé číslo se znaménkem.

Inicializace

Do proměnné bychom měli při jejím vytvoření rovnou uložit nějaký *výraz* , který musí být stejného datového typu jako je typ proměnné:

```
int a = 10;
int b = 10 + 15;
```

Obecná syntaxe pro definici proměnné je

```
<datový typ> <název>;
```

popřípadě

```
<datový typ> <název> = <výraz>;
```

pokud použijeme inicializaci.

> Všimněte si, že na konci definice proměnné vždy musí následovat středník (**;**). Opomenutí středníku na konci příkazu je velmi častá chyba, která často končí těžko srozumitelnými chybovými hláškami při překladu. Dávejte si tak na středníky pozor, obzvláště ze začátku.

Vždy inicializujte proměnné!

Je opravdu důležité do proměnné vždy při její definici přiřadit nějakou úvodní hodnotu. Pokud to neuděláme, tak její hodnota bude **nedefinovaná** (*undefined*), což v praxi znamená, že může být jakákoliv a při každém spuštění

programu se může lišit. Čtení hodnoty takovéto nedefinované proměnné způsobuje **nedefinované chování** (*undefined behaviour*)[2] programu. Pokud k tomu dojde, tak si překladač s vaším programem může udělat, co se mu zachce, a váš program se poté může chovat nepředvídatelně.

[2]Situace, které můžou způsobit nedefinované chování, budou dále v textu označené pomocí ikony .

Proto vždy dávejte proměnným iniciální hodnotu!

Čtení

Pokud v programu použijeme název platné proměnné, tak dojde k načtení její hodnoty. Pokud použijeme název proměnné v programu na místě, kde je očekáván výraz, tak se vyhodnotí jako současná hodnota proměnné:

```
int main() {
    int a = 5;
    int b = a;  // hodnota `b` je 5
    int c = b + a + 1;  // hodnota `c` je 11

    return 0;
}
```

Kdekoliv tak můžete použít výraz, můžete použít i proměnnou (pokud sedí datové typy). Pro výpis hodnot proměnných na výstup programu můžete `printf`. Hodnoty proměnných můžete zkoumat také krokováním pomocí debuggeru .

Zápis

Pokud by proměnná měla pouze svou původní hodnotu, tak by nebyla moc užitečná. Hodnoty proměnných naštěstí jde měnit. Můžeme k tomu použít další typ *C* výrazu **přiřazení** (*assignment*):

```
int main() {
    int a = 5;  // hodnota `a` je 5
    a = 8;      // hodnota `a` je 8

    return 0;
}
```

Obecná syntaxe pro přiřazení do proměnné je

```
<název proměnné> = <výraz>
```

Opět musí platit, že výraz musí být stejného typu[3], jako je proměnná, do které přiřazujeme. Na konci řádku také nesmí chybět středník. Přiřazení je příklad výrazu, který má vedlejší efekt, proto se obvykle používá jako samostatný příkaz (tj. dává se za něj středník).

[3]*C* umožňuje automatické (tzv. **implicitní**) konverze mezi některými datovými typy, takže typ výrazu nemusí být nutně vždy stejný. Tyto konverze se nicméně často chovají neintuitivně a překladač vás před nimi obvykle nijak nevaruje, i když vrátí výsledek, který nedává smysl. Snažte se tak ze začátku opravdu vždy používat odpovídající typy. Více se dozvíte v sekci o datových typech .

Jak přiřazení funguje? Počítač se podívá, na jaké adrese v paměti daná proměnná leží, a zapíše do paměti hodnotu

výrazu, který do proměnné zapisujeme, čímž změní její hodnotu v paměti. Z toho vyplývá, že dává smysl zapisovat hodnoty pouze do něčeho, co má adresu v paměti (prozatím známe pouze proměnné, později si ukážeme další věci, do kterých lze zapisovat). Například příkaz `5 = 8;` nedává smysl. `5` je výraz, číselná hodnota, která nemá žádnou adresu v paměti, nemůžeme tak do ní nic zapsat. Stejně tak nedává moc smysl říct `Číslo 5 odteď bude mít hodnotu 8`.

Zatím známe pouze proměnné, později si však ukážeme další možnosti , jak vytvořit "něco, co má adresu v paměti", a co tak půjde použít na levé straně operátoru zápisu `=`.

Cvičení: Zkuste napsat program, který vytvoří několik proměnných, přečte a změní jejich hodnoty a pak je vypíše na výstup programu (k výpisu využijte `printf`, který jsme si již ukázali dříve).

Definice více proměnných najednou

Pokud potřebujete vytvořit více proměnných stejného datového typu, můžete použít více názvů oddělených čárkou za datovým typem proměnné. Takto například lze vytvořit tři celočíselné proměnné s názvy `x`, `y` a `z`:

```
int x = 1, y = 2, z = 3;
```

Doporučujeme však tento způsob tvorby více proměnných spíše nepoužívat, aby byl kód přehlednější.

Globální proměnné

Proměnné, které jsme si ukázali, byly vytvářeny uvnitř funkcí (tj. ne na nejvyšší úrovni souboru). Takovéto proměnné se nazývají **lokální proměnné**. Pokud chceme, aby k nějaké proměnné byl přístup odkudkoliv v programu, tak můžeme vytvořit proměnnou na úrovni souboru. Takovéto proměnné se nazývají **globální**.

V rámci jednoho souboru lze globální proměnnou použít od místa, kde je definována, až po konec souboru:

```
#include <stdio.h>

// zde nelze použít proměnnou `globalni_promenna`

int globalni_promenna = 1;

int main() {
    globalni_promenna += 1;
    printf("%d\n", globalni_promenna);

    return 0;
}
```

Iniciální hodnota

Narozdíl od lokálních proměnných, globální proměnné se nainicializují na hodnotu `0`[1], i když jim žádnou úvodní hodnotu nedáte. I tak je ale dobrým zvykem úvodní hodnotu takovýmto proměnným dát, aby šlo jasně vidět, že absence úvodní hodnoty není pouze nedopatřením ze strany programátora.

[1]Je to zajištěno tím, že jsou uloženy v sekci spustitelného souboru nazývané `.bss`. Po spuštění programu jsou tak automaticky vynulovány.

(Ne)používání globálních proměnných

Globální proměnné jsou zde zmíněny pro úplnost, nicméně doporučujeme je používat spíše zřídka, obzvláště pokud půjde o globální proměnné, které půjde měnit (tj. pokud to nebudou konstanty). Obecně řečeno, na čím více místech je proměnná dostupná, tím složitější je přemýšlení nad tím, jak přesně s ní pracovat, proto je lepší používat proměnné lokální, pokud to jde.

Když je proměnná globální, tak je k ní přístup v podstatě odkudkoliv v programu. To sice zní neškodně, ba i užitečně, nicméně přináší to s sebou značné nevýhody, pokud lze proměnnou zároveň měnit. Jakmile totiž lze proměnnou odkudkoliv změnit, snadno se vám může stát, že nějaký kus programu vám bude hodnotu takovéto proměnné měnit "pod rukama", a bude obtížné najít kód, který danou proměnnou změnil (a také důvod, proč ji změnil).

> Globální proměnné také mohou způsobovat problémy, pokud ve vašem problému budete využívat více jader procesoru. Tzv. paralelní programy nicméně nebudeme v tomto předmětu řešit, více se o nich dozvíte například v předmětu Architektury počítačů a paralelních systémů.

Konstanty

V určitých případech můžeme chtít mít proměnné s konstantní hodnotou, které by se neměly v průběhu programu měnit. Takové proměnné se nazývají **konstanty** (*constants*).

Abychom zamezili nechtěné změně hodnoty konstanty, můžeme datový typ proměnné označit klíčovým slovem `const`, který umístíme před[1] název datového typu. Pokud bychom se snažili o změnu proměnné s takovýmto datovým typem, překladač nám to nedovolí.

> [1]Modifikátor `const` lze umístit i za datový typ. Někteří programátoři o umístění tohoto modifikátoru vedou vášnivé diskuze . Důležité hlavně je, abyste ve volbě umístění modifikátorů byli konzistentní a používali je na všech místech stejně.

```
int main() {
    const int a = 5;
    a += 1; // chyba

    return 0;
}
```

Použití konstant může mít několik důvodů:

- V programech někdy opakovaně používáme konstantní hodnoty, které mají pevně danou hodnotu. Při čtení zdrojového kódu nemusí být jasné, co takového hodnoty znamenají (v takovém případě se hanlivě označují jako "magické konstanty"). Abychom takového hodnoty pojmenovali, můžeme je uložit do konstantní proměnné. Při čtení programu pak bude zřejmé, co reprezentují. Porovnejte variantu s nepopsanými číselnými hodnotami:

  ```
  float vypocti_cenu(float cena) {
      return cena * (1 + 0.21);
  }
  float vypocti_odvod(float celkova_cena, bool dph) {
      if (dph) {
          return celkova_cena * 0.21;
      } else {
  ```

```
            return 0;
        }
    }
```

s variantou využívající pojmenované konstanty:

```
const float DPH = 0.21f;

float vypocti_cenu(float cena) {
    return cena * (1 + DPH);
}
float vypocti_odvod(float celkova_cena, bool dph) {
    if (dph) {
        return celkova_cena * DPH;
    } else {
        return 0;
    }
}
```

Druhá varianta kódu je jistě čitelnější.

- V určitých případech, například u konstantních řetězců, jsou data uložena v oblasti paměti, kterou nelze měnit. Pomocí `const` si můžeme pohlídat, že se takováto paměť opravdu nezmění.

Složený zápis

Často potřebujeme hodnotu proměnné pouze trochu poupravit, a ne do ní vyloženě zapsat novou hodnotu. Běžná je například operace zvýšení hodnoty proměnné o `1` (tzv. **inkrementace** proměnné). K tomu můžeme použít tento příkaz:

```
pocet = pocet + 1; // zvýšení hodnoty proměnné `pocet` o 1
```

nicméně to je docela zdlouhavé. Proto *C* nabízí tzv. operátory **složeného zápisu** (*compound assignment*). Tyto operátory jsou spojené z normálního operátoru (např. `+`) a operátoru `=`: `+=`, `-=`, `*=`, atd. Složený zápis

```
<proměnná> <operátor>= <výraz>;
```

je ekvivalentní příkazu

```
<proměnná> = <proměnná> <operátor> <výraz>;
```

Například:

```
int pocet = 0;
pocet += 1;   // stejné jako pocet = pocet + 1;
pocet *= 3;   // stejné jako pocet = pocet * 3;
```

Stejně jako zápis je složený zápis příkladem výrazu s vedlejším efektem.

Inkrementace a dekrementace

Speciálním případem složeného zápisu je tzv. **inkrementace** (zvýšení hodnoty proměnné o jedničku) a

dekrementace (snížení hodnoty proměnné o jedničku). Tyto operace jsou tak časté, že *C* obsahuje speciální "zkratky" pro jejich provedení. Aby to nebylo tak jednoduché, tak tyto zkratky existují ve dvou variantách:

- *Postfixová*: `<proměnná>++`. Tento výraz se vyhodnotí jako hodnota dané proměnné, a **poté** zvýší hodnotu proměnné o jedničku. Zkuste uhodnout, co vypíše následující program:

```
#include <stdio.h>

int main() {
    int a = 1;
    int b = a++;
    printf("%d\n", a);
    printf("%d\n", b);

    return 0;
}
```

- *Prefixová*: `++<proměnná>`. Tento výraz **nejprve** zvýší hodnotu proměnné, a až poté se vyhodnotí jako (nová, již zvýšená) hodnota dané proměnné. Zkuste uhodnout, co vypíše následující program:

```
#include <stdio.h>

int main() {
    int a = 1;
    int b = ++a;
    printf("%d\n", a);
    printf("%d\n", b);

    return 0;
}
```

Dekrementace se chová totožně jako inkrementace, pouze s tím rozdílem, že snižuje hodnotu proměnné o `1` a místo `++` používá `--`.

Inkrementace a dekrementace jsou příklady výrazů s vedlejším efektem.

> Tyto zkratky jsou sice užitečné, ale také můžou vyústit v překvapivé chování díky způsobu, kterým jsou vyhodnocovány. Ze začátku je radši využívejte pouze v situacích, kdy budou použity jako příkaz, který změní hodnotu proměnné (`i++;`). Jinak řečeno, raději se moc nespoléhejte na hodnotu, ve kterou se inkrementace/dekrementace vyhodnotí.

Pojmenovávání proměnných

V *C* existují určitá pravidla pro pojmenování proměnných:

- Proměnné se nesmí jmenovat stejně jako klíčová slova, jinak by překladač neuměl rozlišit, co je název proměnné a co klíčové slovo (například u `int int;`).
- Název proměnné může obsahovat pouze malá (`a-z`) a velká (`A-Z`) písmena anglické abecedy, číslice (`0-9`) a podtržítko (_).

- Název proměnné nesmí začínat číslicí, tj. `5x` není validní název proměnné.

V programech je nutné neustále přiřazovat něčemu název, což zdaleka není tak jednoduché, jak se může na první pohled zdát. Kromě výše zmíněných pravidel je zároveň vhodné volit názvy tak, aby byly přehledné pro vás (a ostatní programátory, kteří váš zdrojový kód budou číst). Názvy proměnných jako `a` nebo `x` jsou nicneříkající a kód s podobnými názvy je pak složitější pochopit. Porovnejte následující dva úseky kódu, které se liší pouze v použitých názvech proměnných:

```
int c = 1337;
int x = c - y;
int d = x * z;

// nebo

int zakladni_cena = 1337;
int zlevnena_cena = zakladni_cena - sleva;
int finalni_cena = zlevnena_cena * dph;
```

I když je druhá varianta delší, tak jde okamžitě poznat, co program počítá, narozdíl od první varianty.

Víceslovné názvy

Existuje několik zaběhlých stylistických způsobů pro zápis názvů v *C*, které obsahují více slov. Zde je seznam nejpoužívanějších konvencí:

- `Camel case`: `mujUcet`, `prvniKlikUzivatele`
- `Pascal case`: `MujUcet`, `PrvniKlikUzivatele`
- `Snake case`: `muj_ucet`, `prvni_klik_uzivatele`
- `Screaming snake case`: `MUJ_UCET`, `PRVNI_KLIK_UZIVATELE`

Různé konstrukce *C* můžou využívat různé styly, například častá konvence je použití `snake_case` pro názvy proměnných a funkcí a `PascalCase` pro názvy struktur . Který styl budete používat záleží na vaší osobní preferenci, nicméně důležité je zejména držet se jednotného stylu a nekombinovat různé styly (pro jednotlivé typy konstrukcí) v jednom programu.

Čeština nebo angličtina?

Pokud vám to přijde přehlednější, tak ze začátku můžete používat české názvy[1] pro názvy proměnných a dalších konstrukcí. Může tak pro vás být snadnější odlišit, kterou část kódu jste vytvořili vy (ta bude mít český název), a co je naopak vestavěná součást *C* (např. `int`).

[1] Bez diakritiky.

Nicméně, jak už bylo uvedeno v úvodu , primárním jazykem programování je angličtina. Pokud byste se někdy setkali s cizím kódem a museli ho pochopit či upravit, určitě oceníte, když bude v angličtině, než kdyby byl například ve finštině. Stejně tak pokud budete sdílet svůj kód online, můžete s ním oslovit mnohem širší skupinu programátorů, když bude v angličtině, než kdyby byl v češtině.

Jakmile se tedy v programování trochu aklimatizujete, používejte ve všech svých programech raději anglické názvy.

Datové typy

Paměť počítače pracuje s jednotlivými *byty*, nicméně pro lidi je žádoucí používat popis dat v paměti na mnohem vyšší úrovni abstrakce, aby se nám o datech jednoduššeji přemýšlelo. Pokud programujeme textový editor, chceme se bavit o znacích, odstavcích, fontech či barvách, pokud programujeme počítačovou hru, chceme se bavit o zbraních, brnění, kouzlech či pixelech.

Přesně k tomu slouží **datové typy**, které popisují, jak budeme interpretovat konkrétní hodnoty daného typu v paměti, kolik bytů budou zabírat a jaké operace nad nimi budeme moct provádět. Jazyk *C* obsahuje několik vestavěných datových typů, později si ukážeme, jak vytvořit své vlastní.

Celočíselné datové typy

Asi nejpřirozenějším a nejpoužívanějším datovým typem ve většině programovacích jazyků jsou (celá) čísla. Tyto číselné datové typy nám umožňují pracovat s celými čísly, které mají typicky jednotky (1 - 8) bytů[1]. Počet bytů udává, jak velký rozsah mohou hodnoty daného typu obsahovat. Například číslo s 2 byty (16 bity) bez znaménka může obsahovat hodnoty 0 až 2^{16}-1. Čím více bytů, tím více zabere hodnota daného typu místa v paměti.

[1] I když 8 bytů (64 bitů) může znít jako málo, tak pomocí takového čísla můžeme vyjádřit 2^{64} (neboli `18 446 744 073 709 551 616`) různých hodnot, což pro naprostou většinu běžného použití čísel bohatě stačí.

U celých číselných typů se rozlišuje, zda jsou **signed** (se znaménkem) nebo **unsigned** (bez znaménka, nezáporné). Tato vlastnost udává, jaké hodnoty může typ nabývat (tj. jestli mohou být i záporné nebo ne). Například číslem o velikosti jednoho bytu můžeme reprezentovat 256 různých hodnot. Pokud budeme interpretovat toto číslo se znaménkem, tak může uchovávat hodnoty -128 až 127. Pokud ho budeme interpretovat bez znaménka, tak může uchovávat hodnoty 0 až 255.

C obsahuje několik základních typů celočíselných proměnných, které se liší v tom, kolik mají bytů a jestli jsou znaménkové nebo ne. Pokud před název typu napíšeme `signed`, bude se jednat o znaménkový typ, pokud použijeme `unsigned`, tak použijeme typ bez znaménka. Většina typů je implicitně se znaménkem, tj. `int` je to samé jako `signed int`. V následující tabulce je seznam nejčastějších celočíselných typů[2]:

[2] Počet bytů (a znaménkovost u typu `char`) záleží na kombinaci použitého hardwaru, operačního systému a překladače. Zde jsou uvedeny hodnoty, se kterými se můžete nejčastěji setkat na 64-bitovém x86 Linuxovém systému s překladačem GCC při použití dvojkového doplňku.

Název	Počet bytů	Rozsah hodnot	Se znaménkem
`char` nebo `signed char`	1	[-128; 127]	☐
`unsigned char`	1	[0; 255]	☐
`short` nebo `signed short`	2	[-32 768; 32 767]	☐
`unsigned short`	2	[0; 65 535]	☐
`int` nebo `signed int`	4	[-2 147 483 648; 2 147 483 647]	☐
`unsigned int`	4	[0; 4 294 967 295]	☐

`long` nebo `signed long`	8	[-9 223 372 036 854 775 808; 9 223 372 036 854 775 807]	□
`unsigned long`	8	[0; 18 446 744 073 709 551 615]	□

Každý vestavěný datový typ (`char`, `short`, `int`) a modifikátor znaménkovosti (`signed`, `unsigned`) je zároveň klíčovým slovem.

Pokud ze začátku nebudete vědět, který typ zvolit, tak pro základní aritmetické operace používejte ze začátku typy se znaménkem s 4 byty, tedy `int`. Tento typ je také implicitně použit, když v programu použijete číselný výraz, například výraz `1` má datový typ `int`[3].

[3]Pouze pokud by výraz nešel reprezentovat typem `int`, použije se číselný typ s více byty.

Typ `char` je speciální v tom, že zároveň běžně reprezentuje textové znaky v ASCII kódování. Více o reprezentaci textu v programech se dozvíte v sekci o řetězcích.

Operace s číselnými typy

C umožňuje provádět operace nad vestavěnými datovými typy pomocí tzv. **operátorů**. Při práci s výrazy celočíselných typů lze provádět běžné aritmetické operace `+`, `-`, `/`, `*` nebo `%` (zbytek po dělení). Například `5 + 8` nebo `2 * 16` tak bude obvykle fungovat tak, jak byste očekávali. Je si ale třeba dát pozor na několik zrádných věcí:

- Při dělení dvou celočíselných čísel pomocí operátoru `/` dochází k celočíselnému dělení, tj. například výsledek výrazu `5 / 2` je `2`, a ne `2.5`. Pokud chcete provádět dělení desetinných čísel, musíte použít odpovídající datový typ. Zkuste si to:

```
#include <stdio.h>
int main() {
    printf("%d\n", 5 / 2);
    return 0;
}
```

- Jelikož mají čísla v počítači omezenou přesnost (typicky několik jednotek bytů), tak může při matematických operacích dojít k tzv. **přetečení** (*overflow*). Například pokud vynásobíme jednobytové číslo `50` desíti, tak bychom očekávali výsledek `500`, nicméně tak velké číslo nelze v jednom bytu reprezentovat. Výsledkem místo toho bude `244` (`500 % 256`), pokud se jedná o číslo bez znaménka, nebo `-12`, pokud jde o číslo se znaménkem. Podobné výsledky jsou silně neintuitivní, pokud tedy váš program vrací zvláštní číselný výsledek, zkontrolujte si, jestli neprovádíte operace, při kterých mohlo dojít k přetečení.
- *C* provádí implicitní konverze mezi datovými typy, které mohou změnit datový typ výrazů, které používáte, bez vašeho vědomí. Je tak (obzvláště ze začátku) vhodné ujistit se, že provádíte operace mezi stejnými datovými typy.
- Stejně jako v matematice, tak i v *C* záleží u operátorů na jejich prioritě a asociativitě. Seznam všech operátorů spolu s jejich prioritiou naleznete zde . Například výsledek výrazu `1 + 2 * 3` je `7`, a ne `9`. Pokud budete chtít prioritu ovlivnit, můžete výrazy **uzávorkovat**, abyste jim dali větší přednost: `(1 + 2) * 3` se vyhodnotí jako `9`.

Kromě základních aritmetických operací *C* podporuje také bitové operace :

- AND: operátor `&`
- OR: operátor `|`
- XOR: operátor `^`

Tabulka aritmetických operátorů

Zde je pro přehlednost tabulka se základními aritmetickými operátory. Datový typ výsledku těchto operátorů záleží na datovém typu jejich parametrů.

Operátor	Popis	Příklad
`+`	Sečtení	`1 + 5`
`-`	Odečtení	`2.3 - 4.8`
`*`	Násobení	`3 * 8`
`/`	Dělení	`4 / 2`
`%`	Zbytek po dělení (modulo)	`5 % 2`
`&`	Bitový součin	`12 & 4`
`|`	Bitový součet	`12 | 4`
`~`	Bitová negace	`~8`
`^`	Bitový XOR	`14 ^ 18`
`<<`	Bitový posun doprava	`137 << 2`
`>>`	Bitový posun doleva	`140 >> 3`

O dalších typech operátorů se postupně dozvíte během semestru. Plný seznam *C* operátorů naleznete zde .

Explicitní konverze

Někdy potřebujete převést hodnoty mezi různými datovými typy. K tomu slouží **operátor přetypování** (*cast operator*), který má syntaxi `(<datový typ>) <výraz>` a převede výraz na daný datový typ. Například `(short) 1` převede výraz `1` z typu `int` na `short`. Je dobré si uvědomit, k čemu může dojít při převodu mezi různými datovými typy:

- Pokud je cílový datový typ menší a převáděnou hodnotu v něm nelze reprezentovat, tak dojde k oseknutí hodnoty. V důsledku způsobu reprezentace hodnot v počítači takováto operace odpovídá zbytku po dělení:

  ```
  unsigned short a = 256;
  (unsigned char) a // hodnota tohoto výrazu je 0 (256 % 256)
  ```

- Pokud převádíte znaménkový typ na bezznaménkový a hodnota převáděného výrazu je záporná, tak nedojde k intuitivnímu použití absolutní hodnoty[4]. V důsledku způsobu reprezentace hodnot v počítači takováto operace odpovídá přičtení dané hodnoty k maximální možné hodnotě cílového typu:

  ```
  signed char c = -50;
  (unsigned char) c // hodnota tohoto výrazu je 206 (256 - 50)
  ```

[4] K tomu můžete použít například funkci abs .

Pokud se chcete dozvědět více o tom, proč konverze mezi typy fungují tak, jak fungují, tak se podívejte na to, jak

funguje dvojkový doplněk.

Hexadecimální a oktální zápis čísel

V *C* můžete zapisovat číselné hodnoty také pomocí oktální (osmičkové) či hexadecimální (šestnáctkové) soustavy. Čísla začínající na `0` budou interpretována jako osmičková soustava, čísla začínající na `0x` budou interpretována jako šestnáctková soustava:

```
#include <stdio.h>

int main() {
    int a = 13;     // hodnota 13
    int b = 015;    // hodnota 13
    int c = 0xD;    // hodnota 13
    printf("%d\n", a);
    printf("%d\n", b);
    printf("%d\n", c);

    return 0;
}
```

Desetinné číselné typy

Pokud budete chtít provádět výpočty s desetinnými čísly, tak můžete využít datové typy s tzv. **plovoucí řádovou čárkou** (*floating point numbers*). Hodnoty těchto datových typů umožňují udržovat čísla sestávající se z celé a z desetinné části. Díky tomu, jak jsou navržena , tato čísla dokáží reprezentovat jak velmi malé, tak velmi velké hodnoty (za cenu přesnosti desetinné části).

V *C* jsou dva základní vestavěné datové typy pro práci s desetinnými čísly, liší se pouze velikostí (a tedy i tím, jak přesně dokáží desetinná čísla reprezentovat). Oba dva typy jsou znaménkové:

Název	Počet bytů	Rozsah hodnot	Přesnost	Se znaménkem
`float`	4	[-3.4e+38; 3.4e+38]	~7 des. míst	☐
`double`	8	[-1.7e+308; 1.7e+308]	~16 des. míst	☐

Název `double` pochází z "double precision", tedy dvojitá přesnost (typ `float` se také někdy označuje pomocí "single precision").

Pokud chcete v programu vytvořit výraz datového typu `double`, stačí napsat desetinné číslo (jako desetinný oddělovač se používá tečka, ne čárka): `10.5`, `-0.73`. Pokud chcete vytvořit výraz typu `float`, tak za toto číslo ještě přidejte znak `f`: `10.5f`, `-0.73f`.

Pokud chcete vytisknout na výstup hodnotu datového typu `float` nebo `double`, můžete použít zástupný znak `%f`:

```
printf("Desetinne cislo: %f\n", 1.0);
```

Přesnost desetinných čísel

Je třeba si uvědomit, že desetinná čísla v počítači mají pouze konečnou přesnost a jsou reprezentována v dvojkové soustavě:

- V počítači nelze reprezentovat iracionální čísla s nekonečnou přesností. Pokud tedy chcete do paměti uložit například hodnotu π, budete ji muset zaokrouhlit.
- Kvůli použití dvojkové soustavy některé desetinné hodnoty nelze vyjádřit přesně. Například číslo $\frac{1}{3}$ lze v desítkové soustavě vyjádřit zlomkem, ale v dvojkové soustavě toto číslo má nekonečný desetinný rozvoj (`0.010101...`) a opět tedy nelze vyjádřit přesně:

```
#include <stdio.h>
int main() {
    printf("%f\n", 1.0 / 3.0);
    return 0;
}
```

Konverze na celé číslo

Pokud budete konvertovat desetinné číslo na celé číslo, tak dojde k "useknutí" desetinné části:

```
#include <stdio.h>
int main() {
    printf("%d\n", (int) 1.6);
    printf("%d\n", (int) -1.6);
    return 0;
}
```

Toto chování odpovídá zaokrouhlení k nule, tj. kladná čísla se zaokrouhlí dolů a záporná čísla nahoru.

Pravdivostní typy

Posledním základním datovým typem, který si ukážeme, je pravdivostní typ **Booleovské logiky**. Hodnoty tohoto datového typu mají pouze dvě možné varianty - **pravda** (*true*) nebo **nepravda** (*false*). Tento typ se hodí zejména pro různé logické operace, například porovnávání hodnot (`Je a menší než b?` - ano/ne).

V *C* se Booleovský datový typ nazývá `_Bool`. Nicméně tento název je docela krkolomný, obvykle se proto používá místo něho typ `bool`. Abyste ho mohli použít, tak na začátek programu musíte vložit řádek `#include <stdbool.h>`. Později si vysvětlíme, co tento řádek dělá.

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

int main() {
    bool venku_je_hezky = true;
    bool upr_je_slozite = false;

    printf("%d\n", venku_je_hezky);
    printf("%d\n", upr_je_slozite);

    return 0;
}
```

Jak lze v ukázce výše vidět, `true` reprezentuje pravdivý Booleovský výraz a `false` nepravdivý Booleovský výraz a `bool` hodnoty lze vytisknout na výstup stejným způsobem jako celočíselné hodnoty.[1] Hodnoty Booleovského typu obvykle zabírají v paměti jeden *byte*.

[1]Při výpisu dojde ke konverzi `boolu` na celé číslo.

Logické operace

V (Booleovské) logice existují tři základní operátory:

- **logický součin** (*AND*): `X a zároveň Y`
- **logický součet** (*OR*): `X nebo Y`
- **logická negace** (*NOT*): `neplatí X`

Tyto logické operace lze v *C* použít pomocí následujících operátorů:

- **AND**: `&&`
- **OR**: `||`
- **NOT**: `!`

Tyto operátory můžete použít mezi dvěmi výrazy datového typu `bool`. Například:

```
bool je_muz = true;
bool je_zena = false;
bool je_clovek = je_muz || je_zena; // true || false -> true

bool je_rodic = true;
bool je_otec = je_rodic && je_muz;  // true && true -> true
bool je_matka = je_rodic && ~je_otec; // true && ~true -> true && false -> false
```

Pro připomenutí, zde je pravdivostní tabulka těchto logických operátorů:

X	Y	X && Y	X || Y	~X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Porovnávání hodnot

Při programování často potřebujete porovnat hodnoty mezi sebou:

- `Má Jarda více bodů než Kamil?`
- `Má uživatelovo heslo více než 5 znaků?`
- `Má Lenka na účtu alespoň 100 dolarů?`

K tomu slouží šest základních porovnávacích operátorů:

- **Rovná se**[2]: `==`

  [2]Zde si dávejte velký pozor na rozdíl mezi `=` (přiřazení hodnoty) a `==` (porovnání dvou hodnot). Záměna těchto dvou operátorů je častou začátečnickou chybou a vede k obtížně nalezitelným chybám.
- **Nerovná se**: `!=`

Větší: `>`

- **Větší nebo rovno**: `>=`
- **Menší**: `<`
- **Menší nebo rovno**: `<=`

Porovnávat mezi sebou můžete libovolné hodnoty dvou stejných datových typů. Výsledkem porovnání je výraz datového typu `bool`:

```
int jarda_body = 10;
int kamil_body = 13;

bool remize = jarda_body == kamil_body; // false
bool vyhra_jardy = jarda_body > kamil_body; // true

int delka_hesla = 8;
bool heslo_moc_kratke = delka_hesla <= 5; // false
```

Dávejte si ovšem pozor na to, že pouze operátory `==` a `!=` lze použít univerzálně na všechny datové typy. Například použít `<` pro porovnání dvou Booleovských hodnot obvykle nedává valný smysl, operátory `<`, `<=`, `>` a `>=` jsou obvykle využívány pouze pro porovnávání čísel.

Porovnávání hodnot můžete zkombinovat s logickými operátory pro vyhodnocení komplexních pravdivostních výrazů:

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

int main() {
    int delka_hesla = 8;
    bool email_overen = false;
    int rok_narozeni = 1994;

    bool uzivatel_validni = delka_hesla >= 9 && (email_overen || rok_narozeni > 1990); // false
    bool uzivatel_validni2 = delka_hesla >= 9 && email_overen || rok_narozeni > 1990; // true

    printf("%d\n", uzivatel_validni);
    printf("%d\n", uzivatel_validni2);

    return 0;
}
```

Zde je opět třeba dávat si pozor na prioritu operátorů (například `&&` má větší prioritu než `||`) a v případě potřeby výrazy uzávorkovat. Pokud si zkusíte přeložit tento program, tak vás dokonce překladač bude varovat před tím, že jste výraz neuzávorkovali a může tak vracet jiný výsledek, než očekáváte.

Tabulka logických operátorů

Zde je pro přehlednost tabulka s logickými operátory. Datový typ výsledku je u těchto operátorů vždy `bool`.

Operátor	Popis	Příklad
`&&`	Logický součin (AND)	`a == b && c >= d`
`||`	Logický součet (OR)	`a < b || c == d`
`!`	Logická negace (NOT)	`!(a > b && c < d)`
`==`	Rovná se	`a == 5`

`!=`	Nerovná se	`a != 5`
`>`	Větší než	`a > 5`
`>=`	Větší nebo rovno než	`a >= 5`
`<`	Menší než	`a < 5`
`<=`	Menší nebo rovno než	`a <= 5`

Zkrácené vyhodnocování

Při vyhodnocování Booleovských výrazů s logickými operátory se v *C* používá tzv. **zkrácené vyhodnocování** (*short-circuit evaluation*). Například pokud se vyhodnocuje výraz `a || b`, tak může dojít k následující situaci:

- Počítač vše provádí v sekvenčních krocích, tj. nejprve vyhodnotí `a`.
- Pokud má výraz `a` hodnotu `true`, tak už je jasné, že celý výraz `a || b` bude mít hodnotu `true`.
- K vyhodnocení výrazu `b` tak už nedojde, protože je to zbytečné.

Toto chování může urychlit provádění programu, protože přeskakuje provádění zbytečných příkazů, nicméně může také způsobit nečekané chyby. Pokud by například vyhodnocení výrazu `b` obsahovalo nějaké vedlejší efekty, které se projeví při jeho provedení (například změna hodnoty v paměti), tak může být problematické, pokud se vyhodnocení tohoto výrazu zcela přeskočí. Pokud si pamatujete na inkrementaci, tak ta je jedním z případů výrazů, které mají vedlejší efekt (změnu hodnoty proměnné).

Konverze

Pokud se pokusíte o převod celého či desetinného čísla na `bool`, tak můžou nastat dvě varianty:

- Pokud je číslo nenulové, výsledkem bude `true`.
- Pokud je číslo nula, výsledkem bude `false`.

V opačném směru (konverze `bool` u na číslo) dojde k následující konverzi:

- `true` se převede na `1`
- `false` se převede na `0`

Řízení toku

Pokud by počítače program vždy pouze vykonaly od začátku do konce a provedly pokaždé ty stejné operace, tak by nebyly moc užitečné. Sice by zvládly něco rychle vypočítat, ale už ne se rozhodovat, jakou operaci mají provést, nebo nějakou operaci provádět opakovaně, což jsou velmi užitečné vlastnosti.

Instrukce programu se běžně vykonávají ("tečou") jedna po druhé ("odshora dolů"). *C* obsahuje příkazy pro tzv. **řízení toku** (*control flow*), které můžou toto vykonávání instrukcí ovlivnit:

- Podmínky umožňují vykonat kus kódu, pouze pokud platí nějaký výraz. Díky tomu se můžeme rozhodnout, zda nějakou operaci provést, nebo ne, v závislosti na vstupu programu.
- Cykly umožňují vykonávat kus kódu opakovaně. Díky tomu můžeme například provést nějakou operaci pro všechny prvky ze vstupu programu anebo ji provádět, dokud nedojde ke splnění nějaké podmínky.

Ač se to možná nezdá, proměnné, podmínky a cykly bohatě stačí k tomu, abyste byli schopni napsat libovolný počítačový program. Pomocí těchto tří jednoduchých konstrukcí byste tak teoreticky mohli vytvořit třeba textový editor, hru nebo i celý operační systém. Nicméně, pokud bychom využívali pouze tyto konstrukce, tak ve větších programech by bylo složité se zorientovat a byly by také dost neefektivní. Proto si v příštích sekcích ukážeme několik dalších konstrukcí, které vám můžou programování usnadnit.

Podmínky

V programech se často potřebujeme rozhodnout, co by se mělo provést, v závislosti na hodnotě nějakého výrazu:

- Pokud uživatel nakoupil zboží v posledním týdnu, odešli mu e-mail.
- Zadal uživatel správné heslo? Pokud ano, tak ho přesměruj na jeho profil. Pokud ne, tak zobraz chybovou hlášku.
- Jaké má uživatel konto? Pokud kladné, tak ho vykresli zelenou barvou, pokud záporné, tak červenou a pokud nulové, tak černou.

V *C* můžeme provádět takováto rozhodnutí pomocí **podmínek** (*conditions*). Základním příkazem pro tzv. **podmíněné vykonání** kódu je podmínka `if`:

```
if (<výraz typu bool>) {
    // blok kódu
}
```

Pokud se výraz předaný `if`u vyhodnotí jako `true` (pravda), tak se provede blok kódu uvnitř `if`u tak, jak jste zvyklí, a program dále bude pokračovat za příkazem `if`. Pokud se však výraz vyhodnotí jako `false` (nepravda), tak se blok kódu uvnitř `if`u vůbec neprovede. V následujícím programu zkuste změnit výraz uvnitř závorek za `if` tak, aby se blok v podmínce vykonal:

```
#include <stdio.h>

int main() {
    int delka_hesla = 5;

    printf("Kontroluji heslo...\n");
    if (delka_hesla > 5) {
        printf("Heslo je dost dlouhe\n");
    }
    printf("Kontrola hesla dokoncena\n");

    return 0;
}
```

> Anglické slovo `if` znamená v češtině `Jestliže`. Všimněte si tak, že kód výše můžete přečíst jako větu, která bude mít stejný význam jako uvedený *C* kód: `Jestliže je délka hesla větší než pět, tak (proveď kód v bloku)`.

Provádění alternativ

Často v programu chceme provést jednu ze dvou (nebo více) alternativ, opět v závislosti na hodnotě nějakého výrazu. To sice můžeme provést pomocí několika `if` příkazů za sebou:

```
if (body > 90) { znamka = 1; }
```

```
if (body <= 90 && body > 80) { znamka = 2; }
if (body <= 80 && body > 50) { znamka = 3; }
...
```

Nicméně to může být často dost zdlouhavé. *C* tak umožňuje přidat k příkazu `if` příkaz, který se provede v případě, že výraz v podmínce `if` není splněn. Takto lze řetězit více podmínek za sebou, kdy v každé následující podmínce víme, že žádná z předchozích nebyla splněna. Dosáhneme toho tak, že za blokem podmínky `if` použijeme klíčové slovo `else` ("v opačném případě"):

```
if (<výraz typu bool>) {
    // blok kódu
} else ...
```

Pokud za blok podmínky `if` přidáte `else`, tak se program začne vykonávat za `else`, pokud výraz podmínky není splněn. Za `else` pak může následovat:

- Blok kódu, který se rovnou provede:

  ```
  if (body > 90) {
      // blok A
  } else {
      // blok B
  }
  // X
  ```

  Pokud platí `body > 90`, provede se blok A, pokud ne, tak se provede blok B. V obou případech bude dále program vykonávat kód od bodu `X`.

- Další `if` podmínka, která je opět vyhodnocena. Takovýchto podmínek může následovat libovolný počet:

  ```
  if (body > 90) {
      // blok A, více než 90 bodů
  } else if (body > 80) {
      // blok B, méně než 91 bodů, ale více než 80 bodů
  } else if (body > 70) {
      // blok C, méně než 81 bodů, ale více než 70 bodů
  }
  // X
  ```

  Takovéto spojené podmínky se vyhodnocují postupně shora dolů. První podmínka `if`, jejíž výraz je vyhodnocen jako `true`, způsobí, že se provede blok této podmínky, a následně program pokračuje za celou spojenou podmínkou (bod `X`).

  Na konec spojené podmínky můžete opět vložit i `else` s blokem bez podmínky. Tento blok se provede pouze, pokud žádná z předchozích podmínek není splněna:

  ```
  if (body > 90) {
      // blok A, více než 90 bodů
  } else if (body > 80) {
      // blok B, méně než 90 bodů, ale více než 80 bodů
  } else {
  ```

```
        // blok C, méně než 81 bodů
    }
```

Všimněte si, že tento kód opět můžeme přečíst jako intuitivní větu. Pokud je počet bodů vyšší, než 90, tak proveď A. V opačném případě, pokud je vyšší než 80, tak proveď B. Jinak proveď C.

Cvičení: Upravte následující program, aby vypsal:

- `Student uspel s vyznamenanim`, pokud je hodnota proměnné `body` větší než `90`.
- `Student uspel`, pokud je hodnota proměnné `body` v (uzavřeném) intervalu `[51, 90]`.
- `Student neuspel`, pokud je hodnota proměnné `body` menší než `51`.

```
#include <stdio.h>

int main() {
    int body = 50;

    printf("Student uspel\n");

    return 0;
}
```

Vnořování podmínek

Někdy potřebujeme zkontrolovat složitou podmínku (nebo sadu podmínek). Jelikož podmínky jsou *příkazy* a bloky kódu můžou obsahovat libovolné příkazy, tak vám nic nebrání v tom podmínky *vnořovat*:

```
#include <stdio.h>

int main() {
    int delka_hesla = 4;
    int delka_jmena = 3;
    if (delka_hesla > 5) {
        if (delka_jmena > 3) {
            printf("Uzivatel byl zaregistrovan\n");
        } else {
            printf("Uzivatelske jmeno neni dostatecne dlouhe\n");
        }
    } else {
        printf("Heslo neni dostatecne dlouhe\n");
    }

    return 0;
}
```

Cvičení: Upravte hodnotu proměnných v programu výše tak, aby program vypsal `Uzivatel byl zaregistrovan`. Neměňte v programu nic jiného.

Vynechání složených zárovek

Za `if` nebo `else` můžete vynechat složené závorky (`{`, `}`). V takovém případě se bude podmínka vztahovat k (jednomu) příkazu následujícímu za `if/else`:

```
if (body > 80) printf("Student uspel\n");
else printf("Student neuspel\n");
```

Zejména ze začátku za podmínkami vždy však raději používejte složené závorky, abyste předešli případným chybám a učinili kód přehlednějším.

Ternární operátor

Občas chcete použít jeden ze dvou výrazů v závislosti na hodnotě nějaké podmínky. Například pokud byste chtěli přiřadit minimum ze dvou hodnot do proměnné:

```
int a = 1;
int b = 5;

int c = 0;
if (a < b) {
    c = a;
} else {
    c = b;
}
```

Toto lze provést zkráceně pomocí výrazu **ternárního operátoru** (*ternary operator*). Tento výraz má následující syntaxi:

```
<výraz X typu bool> ? <výraz A> : <výraz B>
```

Pokud je výraz `X` pravdivý, tak se ternární operátor vyhodnotí jako hodnota výrazu `A`, v opačném případě se vyhodnotí jako hodnota výrazu `B`. Uhodnete, co vypíše následující program?

```
#include <stdio.h>

int main() {
    int a = 1;
    int b = 5;
    int c = (a >= b) ? a - b : a + b;
    printf("%d\n", c);

    return 0;
}
```

Příkaz `switch`

V případě, že byste chtěli provést rozlišný kód v závislosti na hodnotě nějakého výrazu, a tento výrazu (např. proměnná) může nabývat více různých hodnot, tak může být zdlouhavé použít spoustu `if`ů:

```
if (a == 0) {
    ...
}
else if (a == 1) {
    ...
}
else if (a == 2) {
    ...
}
...
```

Jako jistá zkratka může sloužit příkaz `switch`. Ten má následující syntaxi:

```
switch (<výraz>) {
    case <hodnota A>: <blok kódu>
    case <hodnota B>: <blok kódu>
    case <hodnota C>: <blok kódu>
    ...
}
```

Tento příkaz vyhodnotí předaný výraz, a pokud se ve `switch`i nachází klíčové slovo `case` následované hodnotou odpovídající hodnotě výrazu, tak program skočí na blok, který následuje za `case`. Dále se program bude vykonávat sekvenčně až do konce `switch`e (pak už se `case` ignoruje).

Tento program vypíše `52`, protože předaný výraz má hodnotu `5`, takže program skočí na blok za `case 5` a dále pokračuje sekvenčně až do konce bloku `switch` příkazu.

```
#include <stdio.h>

int main() {
    switch (5) {
        case 0: printf("0");
        case 1: printf("1");
        case 5: printf("5");
        case 2: printf("2");
    }

    return 0;
}
```

Do `switch`e lze předat i blok pojmenovaný `default`, na který program skočí v případě, že se nenalezne žádný `case` s odpovídající hodnotou:

```
#include <stdio.h>

int main() {
    switch (10) {
        case 0: printf("0");
        case 1: printf("1");
        case 5: printf("5");
        case 2: printf("2");
        default: printf("nenalezeno");
    }

    return 0;
}
```

Velmi často chcete provést pouze jeden blok kódu u jednoho `case` a nepokračovat po něm až do konce celého `switch` bloku. Běžně se tedy za každým `case` blokem používá příkaz `break`, který ukončí provádějí celého `switch` příkazu:

```
#include <stdio.h>

int main() {
    switch (1) {
        case 0: printf("0"); break;
        case 1: printf("1"); break;
        case 2: printf("2"); break;
        default: printf("nenalezeno");
    }

    return 0;
}
```

Cykly

V programech chceme často provádět nějakou operaci opakovaně, například:

- Pro každý záznam v databázi vypiš řádek do souboru.
- Pošli zprávu každému účastníkovi chatu.
- Načítej řádky ze souboru, dokud nedojdeš na konec souboru.

Pokud bychom používali pouze sekvenční zápis příkazů, tak bychom museli neustále kopírovat ("copy-pastovat") kód:

```
printf("0\n");
printf("1\n");
printf("2\n");
...
```

což by vedlo k nepřehledným programům[1]. Pokud bychom navíc našli v programu chybu, museli bychom ji opravit na všech místech, kam jsme kód zkopírovali.

[1] Představte si, že chcete na výstup programu nebo do souboru vypsat třeba tisíc různých řádků textu.

Ani s kopírováním kódu bychom si však nevystačili, pokud bychom potřebovali provádět kód opakovaně v závislosti na vstupu programu. Představte si situaci, kdy nám uživatel na vstup programu zadá číslo, kolikrát má náš program vypsat nějaký řádek textu na výstup. Uživatel se při každém spuštění programu může rozhodnout pro jiné číslo, `0`, `1`, `42`, `1000`. Program však zůstává stále stejný - při jeho psaní se musíme rozhodnout, kolik příkazů pro výpis v něm použijeme, a už poté nemůžeme jednoduše tuto volbu změnit, když program běží. Takovýto program bychom tedy zatím (pouze pomocí proměnných a podmínek) neměli jak naprogramovat.

Proto programovací jazyky nabízí tzv. **cykly** (*loops*), pomocí kterých můžeme jednoduše říct počítači, aby určitý blok kódu opakoval, kolikrát budeme chtít. Díky tomu může program i s pouze několika málo řádky kódu říct počítači, aby provedl spoustu instrukcí. Jazyk *C* nabízí dva základní typy cyklů, while a for.

> Další motivací pro využití cyklů je to, že moderní procesory počítačů mají běžně frekvence od 1 do 4 GHz, takže za vteřinu zvládnou provést několik miliard taktů a během každého taktu navíc až desítky různých operací. Jistě si dovedete představit, že s pouze sekvenčním zápisem kódu bychom tento potenciál nemohli naplno využít. I když jeden řádek *C* kódu může být přeložen až na desítky procesorových instrukcí, tak i kdybychom zvládli napsat program se stovkami milionů řádek, pořád bychom takovýmto programem "zabavili" procesor na pouhou vteřinu. Běžící programy tak obvykle tráví většinu času prováděním nějakého cyklu.

Cyklus `while`

Nejjednodušším cyklem v *C* je cyklus `while` ("dokud"):

```
while (<výraz typu bool>) {
    // blok cyklu
}
```

Funguje následovně:

1. Nejprve se vyhodnotí (Booleovský) výraz v závorce za `while`.

2. Pokud výraz není pravdivý, tak se provede bod 3. Pokud je výraz pravdivý, tak se provede blok[1] cyklu a dále se pokračuje bodem 1.

   [1]Blok cyklu se také často nazývá jako **tělo** (*body*) cyklu.

3. Program pokračuje za cyklem `while`.

Jinak řečeno, dokud bude splněná podmínka za `while`, tak se bude opakovaně provádět tělo cyklu. Vyzkoušejte si to na následujícím příkladu:

```
#include <stdio.h>

int main() {
    int pocet = 0;
    while (pocet < 5) {
        printf("Telo cyklu se provedlo\n");
        pocet = pocet + 1;
    }
    return 0;
}
```

Tento kód opět můžeme přečíst jako větu: `Dokud je hodnota proměnná menší než pět, prováděj tělo cyklu`. Jedno vykonání těla cyklu se nazývá **iterace**. Cyklus v ukázce výše tedy provede pět iterací, protože se tělo cyklu provede pětkrát.

Pokud výraz za `while` není splněn, když se `while` začne vykonávat, tak se tělo cyklu nemusí provést ani jednou (tj. bude mít nula iterací).

Je důležité dávat si pozor na to, aby cyklus, který použijeme, nebyl nechtěně **nekonečný** (*infinite loop*), jinak by náš program nikdy neskončil. Zkuste v kódu výše zakomentovat nebo odstranit řádek `count = count + 1;` a zkuste program spustit. Jelikož se hodnota proměnné `count` nebude nijak měnit, tak výraz `count < 5` bude stále pravdivý a cyklus se tak bude provádět neustále dokola. Této situaci se lidově říká "zacyklení"[2]. Po spuštění nekonečného cyklu v prohlížeči radši restartujte tuto stránku :)

[2]Pokud program spouštíte v terminálu a zacyklí se, můžete ho přerušit pomocí klávesové zkratky `Ctrl + C`.

Řídící proměnná

Často chceme provést v těle cyklu jinou operaci v závislosti na tom, která iterace se zrovna vykonává. K tomu obvykle slouží tzv. **řídící proměnná** (*index variable*), která udává, v jaké iteraci cyklu se nacházíme, a podle ní se poté provede odpovídající operace. Například pokud bychom chtěli vypsat na výstup řadu čísel `0` až `4`, tak to můžeme provést s `while` cyklem následovně:

```
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 5) {
        printf("%d\n", i);
        i += 1;
    }
    return 0;
}
```

Řídící proměnná je zde `i` - tento název se pro řídící proměnné pro jednoduchost často používá.

Řízení toku cyklu

V cyklech můžete využívat dva speciální příkazy, které fungují pouze v těle nějakého cyklu:

- Příkaz `continue;` způsobí, že se přestane vykonávat tělo cyklu, a program se vrátí na začátek cyklu (tedy u `while` na vyhodnocení výrazu). `continue` lze chápat jako skok na další iteraci cyklu. Zkuste uhodnout, co vypíše následující kód:

```
#include <stdio.h>

int main() {
    int pocet = 0;
    while (pocet < 10) {
        pocet = pocet + 1;

        if (pocet < 5) continue;

        printf("Hodnota: %d\n", pocet);
    }

    return 0;
}
```

- Příkaz `break;` způsobí, že se cyklus přestane vykonávat a program začne vykonávat kód, který následuje za cyklem. Cyklus se tak zcela přeruší. Zkuste uhodnout, co vypíše následující kód:

```
#include <stdio.h>

int main() {
    int pocet = 0;
    while (pocet < 10) {
        if (pocet * 2 > 12) break;

        printf("Hodnota: %d\n", pocet);
        pocet = pocet + 1;
    }

    return 0;
}
```

Tip pro návrh cyklů while

Vnořování cyklů

Stejně jako podmínky, i cykly jsou příkazy, a můžete je tak používat libovolně v blocích *C* kódu a také je vnořovat. Chování vnořených cyklů může být ze začátku trochu neintuitivní, proto je dobré si je procvičit. Zkuste si pomocí debuggeru krokovat následující kód, abyste pochopili, jak se provádí, a zkuste odhadnout, jakých hodnot budou

postupně nabývat proměnné `i` a `j`. Poté odkomentujte výpisy `printf` a ověřte, jestli byl váš odhad správný:

```
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 3) {
        // printf("i: %d\n", i);
        int j = 0;
        while (j < 4) {
            // printf("  j: %d\n", j);
            j = j + 1;
        }

        i = i + 1;
    }

    return 0;
}
```

Pro každou iteraci "vnějšího" `while` cyklu se provedou čtyři iterace "vnitřního" `while` cyklu. Dohromady se tak provede celkem `3 * 4` iterací.

Cyklus `do while`

Cyklus `while` má také alternativu zvanou `do while`. Tento cyklus má následující syntaxi:

```
do {
    // tělo cyklu
}
while (<výraz typu bool>);
```

Tento kód můžeme číst jako `Dělej <tělo cyklu>, dokud platí <výraz>`.

Jediný rozdíl mezi `while` a `do while` je, že v cyklu `do while` se výraz, který určuje, jestli se má provést další iterace cyklu, vyhodnocuje až na konci cyklu. Tělo cyklu tak bude pokaždé provedeno alespoň jednou (i kdyby byl výraz od začátku nepravdivý).

Pokud pro to nemáte zvláštní důvod, asi není třeba tento typ cyklu používat.

Cyklus `for`

V programech velmi často potřebujeme vykonat nějaký blok kódu přesně `n`-krát:

- Projdi `n` řádků ze vstupního souboru a sečti jejich hodnoty.
- Pošli zprávu všem `n` účastníkům chatu.
- Vystřel přesně třikrát ze zbraně.

I když pomocí cyklu `while` můžeme vyjádřit provedení `n` iterací, je to relativně zdlouhavé, protože je k tomu potřeba alespoň tří řádků:

- Inicializace cyklu: vytvoření řídící proměnné, která se bude kontrolovat v cyklu
- Kontrola výrazu: kontrola, jestli už proměnná nabrala požadované hodnoty

Operace na konci cyklu: změna hodnoty řídící proměnné

```
int i = 0; // inicializace
while (i < 10) { // kontrola výrazu
    // tělo cyklu
    i += 1; // změna hodnoty řídící proměnné
}
```

Cyklus `for` existuje, aby tuto častou situaci zjednodušil. Kód výše by se dal pomocí cyklu `for` přepsat takto:

```
for (int i = 0; i < 10; i += 1) {
    // tělo cyklu
}
```

Jak lze vidět, `for` cyklus v sobě kombinuje inicializaci cyklu, kontrolu výrazu a provedení příkazu po každé iteraci. Obecná syntaxe tohoto cyklu vypadá takto:

```
for (<příkaz A>; <výraz typu bool>; <příkaz B>) {
    // tělo cyklu
}
```

Takovýto cyklus se vykoná následovně:

1. Jakmile se cyklus začne vykonávat, nejprve se provede příkaz `A`. Zde se typicky vytvoří řídící proměnná s nějakou počáteční hodnotou.
2. Zkontroluje se výraz. Pokud není pravdivý, cyklus končí a program pokračuje za cyklem `for`. Pokud je pravdivý, provede se tělo cyklu a program pokračuje bodem 3.
3. Provede se příkaz `B` a program pokračuje bodem 2.

Cvičení: Napište program, který pomocí cyklu `for` na výstup vypíše čísla od 0 do 9 (včetně).

Funkce

Zatím jsme veškerý kód psali pouze na jedno místo v programu, do "mainu". Jakmile programy začnou být větší a větší, tak začne také být neustále těžší a těžší se v nich zorientovat a udržet je celé v hlavě, abychom nad nimi mohli přemýšlet. Zároveň se nám v programu brzy začnou objevovat úseky kódu, které jsou téměř totožné, ale liší se v drobných detailech. Chtěli bychom tak mít možnost takovýto kód napsat pouze jednou a tyto měnící se detaily do něj pouze "dosadit". K rozdělení kódu programu do sady ucelených částí a jejich parametrizaci slouží **funkce** (*functions*).

Funkce je pojmenovaný blok kódu, na který se můžeme odkázat v jiné části programu a vykonat tak kód, který se ve funkci nachází. S jednou funkcí už jsme se setkali. Jedná se o funkci `main`, jejíž kód je proveden při spuštění programu. My si nicméně můžeme vytvořit vlastní funkce. Zde je příklad vytvoření, tj. **definice** (*definition*) jednoduché funkce s názvem[1] `vypis_text`:

[1]Pravidla pro pojmenovávání funkcí jsou totožná s pravidly pro pojmenovávání proměnných.

```
void vypis_text() {
    printf("Ahoj\n");
}
```

Před názvem funkce je nutné uvést datový typ (zde je uveden typ `void`). Níže bude vysvětleno, k čemu tento typ slouží.

Tento blok [2] kódu se přeloží na instrukce a bude existovat v přeloženém programu stejně jako funkce `main`, nicméně sám o sobě se nezačne provádět. Abychom kód této funkce provedli, musíme ji tzv. **zavolat** (*call*). To provedeme tak, že napíšeme název této funkce a za něj dáme závorky (`()`):

[2] Stejně jako u cyklů se bloku kódu funkce často říká **tělo funkce** (*function body*).

```
#include <stdio.h>

void vypis_text() {
    printf("Ahoj\n");
}
int main() {
    vypis_text(); // zavolání funkce vypis_text
    return 0;
}
```

Zavolání funkce je výraz, při jehož vyhodnocení dojde k provedení kódu funkce, která se volá. Když se v programu nahoře ve funkci `main` vykoná řádek `vypis_text();`, tak se začne vykonávat kód funkce `vypis_text`. Jakmile se příkazy z této funkce vykonají, tak program bude pokračovat ve funkci `main`.

Pomocí volání funkcí můžeme mít kus kódu v programu zapsán pouze jednou ve funkci, a poté ho můžeme spouštět z různých částí programu, podle toho, kdy se nám to zrovna bude hodit.

Parametrizace funkcí

Funkcím lze dávat vstupy zvané **parametry** (*parameters*). Parametry jsou proměnné uvnitř funkce, jejichž hodnotu nastavujeme při zavolání dané funkce. Například následující funkce `vypis_cislo` má parametr `cislo` s datovým typem `int`.

```
#include <stdio.h>

void vypis_cislo(int cislo) {
    printf("Cislo: %d\n", cislo);
}
int main() {
    vypis_cislo(5);
    return 0;
}
```

Při zavolání funkce musíme pro každý její parametr do závorek dát hodnotu odpovídajícího datového typu. Zde je jediný parameter typu `int`, takže při zavolání této funkce musíme do závorek dát jednu hodnotu datového typu `int`: `vypis_cislo(5)`. Před spuštěním příkazů ve funkci dojde k tomu, že hodnota každého parametru se nastaví na hodnotu předanou ve volání funkce[3]. Při zavolání `vypis_cislo(5)` si tak můžete představit, že se vykoná následující kód:

[3] Hodnoty (výrazy) předávané při volání funkce se nazývají **argumenty** (*arguments*). Při volání `vypis_cislo(5)` se tedy do parametru `cislo` nastaví hodnota argumentu `5`.

```
{
    // nastavení hodnot parametrů
    int cislo = 5;
```

```
    // tělo funkce
    printf("Cislo: %d\n", cislo);
}
```

Parametrů mohou funkce brát libovolný počet, nicméně obvykle se používá maximálně cca 5 parametrů, aby funkce a její používání (volání) nebylo příliš složité. Jednotlivé parametry jsou odděleny v definici funkce i v jejím volání čárkami:

```
#include <stdio.h>

void vypis_cisla(int a, int b) {
    printf("Cislo a: %d\n", a);
    printf("Cislo b: %d\n", b);
}
int main() {
    vypis_cisla(5 + 5, 11 * 2);
    return 0;
}
```

Pomocí parametrů můžeme vytvořit kód, který není "zadrátovaný" na konkrétní hodnoty, ale umí pracovat s libovolnou hodnotou vstupu. Díky toho lze takovou funkci využít v různých situacích bez toho, abychom její kód museli kopírovat. Příklady použití parametrů funkcí:

- Funkci `vypis_ctverec`, která přijme jako parametr číslo `n` a vypíše na výstup čtverec tvořený znaky `x` o straně `n`.
- Funkci `vykresli_pixel`, která přijme jako parametry souřadnici na obrazovce a barvu a vykreslí na obrazovce na dané pozici pixel s odpovídající barvou.

Cvičení: Zkuste naprogramovat funkci `vypis_ctverec`.

Návratová hodnota funkcí

Nejenom, že funkce můžou přijímat vstup, ale umí také vracet výstup. Datový typ uvedený před názvem funkce udává, jakého typu bude tzv. **návratová hodnota** (*return value*) dané funkce. V příkladech výše jsme viděli datový typ `void`. Tento datový typ je speciální, protože říká, že funkce nebude vracet *nic*. Pokud funkce má návratový typ `void`, tak nevrací žádnou hodnotu - pokud zavoláme takovouto funkci, tak se sice provede její kód, ale výraz zavolání nevrátí žádnou hodnotu:

```
void funkce() {}

int main() {
    // chyba při překladu, funkce nic nevrací
    int x = funkce();
    return 0;
}
```

Často bychom nicméně chtěli funkci, která přijme nějaké hodnoty (parametry), vypočte nějakou hodnotu a poté ji vrátí. Toho můžeme dosáhnout pomocí příkazu `return <výraz>;`. Při provedení tohoto výrazu se přestane funkce vykonávat a její volání se vyhodnotí hodnotou předaného výrazu. Zde je příklad funkce, která bere jako vstup jedno číslo a spočítá jeho třetí mocninu:

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

```
int treti_mocnina(int cislo) {
   return cislo * cislo * cislo;
}
int main() {
   printf("%d\n", treti_mocnina(5 + 1));
   return 0;
}
```

Příkazů `return` může být ve funkci více:

```
int absolutni_hodnota(int cislo) {
    if (cislo >= 0) {
        return cislo;
    }
    return -cislo;
}
```

Nicméně je důležité si uvědomit, že po provedení příkazu `return` už funkce dále nebude pokračovat:

```
int zvetsi(int cislo) {
    return cislo + 1;
    printf("Provadi se funkce zvetsi\n"); // tento řádek se nikdy neprovede
}
```

Pokud má funkce jakýkoliv jiný návratový typ než `void`, tak v ní musí být vždy proveden příkaz `return`! Pokud k tomu nedojde, tak program může začít vykazovat nedefinované chování a může se tak chovat nepředvídatelně. Například následující funkce je špatně, protože pokud hodnota parametru `cislo` bude nezáporná, tak se ve funkci neprovede příkaz `return`:

```
int absolutni_hodnota(int cislo) {
    if (cislo < 0) {
      return -cislo;
    }
}
```

Pokud má funkce návratový typ `void`, tak její provádění můžeme ukončit pomocí příkazu `return;` (zde nepředáváme žádný výraz, protože funkce nic nevrací).

Syntaxe

Syntaxe funkcí v *C* vypadá takto:

```
<datový typ> <název funkce>(<dat. typ par. 1> <název par. 1>, <dat. typ par. 2> <název par. 2>,
...) {
    // blok kódu
}
```

Datovému typu, názvu funkce a jejím parametrům se dohromady říká **signatura** (*signature*) funkce. Tato informace je důležitá, abychom věděli, jak s danou funkcí pracovat (jak ji volat), k tomu není nutné znát obsah těla funkce.

Výhody funkcí

Zde je pro zopakování uveden přehled výhod používání funkcí:

- Znovupoužitelnost kódu: pokud chcete stejný kód použít na více místech programu, nemusíte ho "copy-pastovat". Stačí ho vložit do funkce a tu poté zavolat.
- Parametrizace kódu: pokud chcete spouštět stejný kód nad různými vstupními hodnotami, stačí udělat funkci, která dané hodnoty přijme jako parametry (a případně vrátí výsledek výpočtu jako svou návratovou hodnotu).
- Abstrakce: když rozdělíte logiku programu do sady funkcí, tak si značně usnadníte přemýšlení nad celým programem. Jednotlivé funkce budete moct testovat a přemýšlet nad nimi separátně, nezávisle na zbytku programu. Pomocí používání funkcí také bude mnohem přehlednější čtení programu, protože bude stačit číst, co se provádí (která funkce se volá) a ne jak se to provádí (jaké příkazy jsou v těle funkce). Takovýhle kód pak lze číst téměř jako větu v přirozeném jazyce:

  ```
  int zivot = vrat_zivoty_hrace(id_hrace);
  zivot = zivot - vypocti_zraneni_prisery(id_prisery);
  nastav_zivory_hrace(id_hrace, zivor);
  ```

- Sdílení kódu: pokud budete chtít použít kód, který napsal někdo jiný, tak toho dosáhnete právě používáním funkcí, které vám někdo připraví.

Umístění funkcí

Funkce v *C* musíme psát vždy na nejvyšší úrovni souboru. V *C* tedy například není možné definovat funkci uvnitř jiné funkce:

```
int main() {
    int test() { }
}
```

Proč název "funkce"?

Možná vás napadlo, že název funkce zní podobně jako funkce v matematice. Není to náhoda, funkce v programech se tak opravdu dají částečně chápat – berou nějaký vstup (parametry) a vracejí výstup (návratovou hodnotu). Například následující matematickou funkci:

$f(x) = 2 \quad x$

můžeme v *C* naprogramovat takto:

```
int f(int x) {
    return 2 * x;
}
```

Aby ale funkce v *C* splňovala požadavky matematické funkce, musí být splněno několik vlastností:

- Funkce nesmí mít žádné vedlejší efekty. To znamená, že by měla pouze provést výpočet na základě vstupních parametrů a vrátit vypočtenou hodnotu. Neměla by číst nebo modifikovat globální proměnné nebo například interagovat se soubory na disku.
- Funkce musí mít návratový typ jiný než `void`, aby vracela nějakou hodnotu. Z toho také vyplývá, že funkce s návratovým typem `void` musí mít nutně nějaké vedlejší efekty, jinak by totiž nemělo cenu ji volat (protože nic nevrací).
- Pokud je funkce zavolána se stejnými hodnotami parametrů, musí vždy vrátit stejnou návratovou hodnotu. Této

vlastnosti se říká *idempotence*. Jelikož jsou počítače deterministické, tato vlastnost by měla být triviálně splněna, pokud funkce neobsahuje žádné vedlejší efekty.

Funkce splňující tyto vlastnosti se nazývají *čisté* (*pure*). S takovýmito funkcemi je jednodušší pracovat a přemýšlet nad tím, co dělají, protože si můžeme být jistí, že nemodifikují okolní stav programu a pouze spočítají výsledek v závislosti na svých parametrech. Pokud to tedy jde, snažte se funkce psát tímto stylem (samozřejmě ne vždy je to možné).

V předmětu Funkcionální programování budete pracovat s funkcionálními programovacími jazyky, ve kterých je většina funkcí čistých.

Rekurze

Pokud funkce obsahuje volání sama sebe, tak tuto situaci nazýváme **rekurzí** (*recursion*). Pro řešení některých problémů může být přirozené rozdělovat je na čím dál tím menší podproblémy, dokud se nedostaneme k podproblému, který je dostatečně jednoduchý, abychom ho vyřešili rovnou. Toto můžeme modelovat právě rekurzí, kdy voláme stejnou funkci s jinými parametry, dokud se nedostaneme k parametrům, pro které umíme problém vyřešit jednoduše, a v ten moment rekurzi ukončíme.

Jedním z jednoduchých problémů, na kterém můžeme rekurzi demonstrovat, je výpočet faktoriálu . Faktoriál lze nadefinovat například takto:

$n! = n \quad (n-1)!$

Vidíme, že tato samotná definice je "rekurzivní": pro výpočet faktoriálu `n` musíme znát hodnotu faktoriálu `n - 1`. Výpočet faktoriálu můžeme provést například následující funkcí:

```
int faktorial(int n) {
    if (n < 1) return 1;
    return n * faktorial(n - 1);
}
```

Pokud je parametr `n` menší než `1`, umíme faktoriál vypočítat triviálně. Pokud ne, tak spočteme faktoriál `n - 1` a vynásobíme ho hodnotou `n`. Je důležité si uvědomit, v jakém pořadí zde probíhá výpočet. Například při volání `factorial(4)`:

1. Zavolá se `factorial(4)`.
2. `factorial(4)` zavolá `factorial(3)`.
3. `factorial(3)` zavolá `factorial(2)`.
4. `factorial(2)` zavolá `factorial(1)`.
5. `factorial(1)` vrátí `1`.
6. `factorial(2)` vrátí `2 * 1`.
7. `factorial(3)` vrátí `3 * 2 * 1`.
8. `factorial(4)` vrátí `4 * 3 * 2 * 1`.

Nejprve tak dojde k vypočtení `factorial(1)`, poté `factorial(2)` atd. Výpočet je tak v jistém smyslu "otočen". Zkuste si výpočet faktoriálu odkrokovat , abyste si ujasnili, jak výpočet probíhá.

Přetečení zásobníku

Je důležité dávat si pozor na to, abychom vždy ve funkci měli podmínku, která rekurzi ukončí. Jinak by se funkce volala "donekonečna", dokud by nakonec nedošlo k přetečení zásobníku.

Funkce standardní knihovny

Když už nyní víme, co jsou to funkce, tak si můžeme vysvětlit, odkud se berou některé funkce, které jsme doposud používali, i když jsme je sami nenapsali.

Například příkaz `printf("...")` je volání funkce s názvem `printf`. Tato funkce pochází ze **standardní knihovny C** (*C standard library*). Jedná se o sadu užitečných funkcí, které jsou tak často využívané, že jsou implicitně překladačem přidány k vašemu programu, abyste je mohli využít a nemuseli ztrácet čas jejich psaním v každém programu od nuly.

Tyto funkce se starají například o následující oblasti:

- Čtení ze vstupu programu a zápis na výstup programu (například funkce `printf`)
- Dynamická alokace paměti
- Čtení a zápis souborů na disku
- Generování náhodných čísel
- Práce s textem
- Práce s časem a datem

a mnoho dalších.

Abychom mohli tyto funkce používat, potřebujeme v našem programu vložit kód, který obsahuje signatury těchto funkcí. Toho dosáhneme pomocí použití preprocesoru – zde se dozvíte, jak funguje příkaz `#include <...>`, který jsme doposud používali jako "black box".

Seznam funkcí dostupných v standardní knihovně můžete naleznout například zde [1].

[1] Dívejte se pouze na funkce pro `C99`, a ne na funkce `C++`. Jedná se o jiný jazyk.

Jak je standardní knihovna *C* připojena k vašim programům a jak si vytvořit vlastní knihovnu se dozvíme později v sekci o knihovnách.

Preprocesor

Než je váš zdrojový soubor přeložen na strojové instrukce, tak jej překladač nejprve prožene tzv. **preprocesorem** (*preprocessor*). Tento program nedělá nic jiného, než že projde váš zdrojový kód a zpracuje příkazy začínající na `#`. V podstatě jediné, co takovéto příkazy dělají, je kopírování ve vašem zdrojovém kódu.

Ukážeme si dva typy příkazů, které preprocesor umí zpracovávat:

- Vkládání souborů do vašeho kódu (`#include`)
- Vytváření maker (`#define`)

Pokud si chcete ověřit, jak vypadá váš zdrojový soubor poté, co jej zpracuje preprocesor, ale předtím, než je přeložen na strojové instrukce, můžete k tomu použít tento příkaz:

```
$ gcc -P -E main.c
```

Vkládání souborů

Příkaz `#include` slouží ke vložení obsahu jiného souboru do vašeho zdrojového kódu. Tento příkaz existuje ve dvou variantách:

```
#include <cesta k souboru>
#include "cesta k souboru"
```

Rozdíl mezi nimi je popsán níže.

Jakmile preprocesor narazí na tento příkaz, tak se pokusí najít soubor na uvedené cestě, zpracuje jeho obsah (tj. vyhodnotí případné další příkazy jako `#include`, které v něm mohou být) a poté jeho obsah vloží na místo, kde je `#include` použit. Jedná se o prosté textové nahrazení (`Ctrl+C -> Ctrl+V`).

Tento příkaz slouží k tomu, abychom mohli používat stejný kód ve více souborech bez toho, abychom jej museli neustále ručně kopírovat. Prozatím budeme vkládat do našeho kódu zejména soubory obsahující různé funkce standardní knihovny *C*. Později si pak ukážeme, jak vytvářet *C* programy sestávající se z více zdrojových souborů.

Zkuste si například tento zdrojový soubor pojmenovat jako `main.c` a pomocí příkazu `gcc -P -E main.c` v terminálu zjistit, jak vypadá poté, co na něj byl aplikován preprocesor:

```
#include <stdio.h>
int main() {
    printf("Hello world\n");
    return 0;
}
```

Asi je zřejmé, že by nebylo praktické kopírovat ručně všechen tento kód pokaždé, když bychom chtěli něco vytisknout na výstup programu.

Relativní cesta

Cesta k souboru zadávaná v `#include` by měla být relativní, tj. není dobrý nápad používat něco podobného:

```
#include "C:/Users/Kamil/Desktop/upr/muj_soubor.h"
```

Takovýto program by totiž jistě nefungoval na jiném než vašem počítači. Z jakého bodu se tato relativní cesta vyhodnotí je popsáno níže.

Rozdíl mezi `#include <...>` a `#include "..."`

Rozdíl mezi těmito variantami není pevně definován, nicméně většina preprocesorů (resp. překladačů) funguje takto:

- `#include <...>` nejprve vyhledá zadanou cestu v tzv. systémových cestách. Jedná se o známé adresáře, ve kterých jsou uloženy jednak soubory standardní knihovny *C*, a také dalších knihoven, které máte v systému nainstalované. Pouze pokud se zde daný soubor nenalezne, tak se cesta vyhodnotí relativně ke zdrojovému souboru, ve kterém byl `#include` použit.

Seznam systémových cest si můžete vypsat pomocí příkazu `echo | gcc -E -Wp,-v -` v Linuxovém terminálu. Do tohoto seznamu můžete také přidat dodatečné adresáře, když `gcc` předáte parametr `-I`. Více se dozvíte v sekci o knihovnách .

Pokud soubor, který chcete do vašeho kódu vložit, se nachází v externí knihovně, která nepatří do vašeho projektu, je běžné používat právě `#include <>`.

- `#include "..."` se nedívá do systémových cest, ale rovnou hledá zadanou cestu relativně k souboru, ve kterém byl `#include` použit. Tuto formu používejte, pokud budete vkládat soubory z vašeho projektu.

Makra

Občas můžeme chtít v programech použít stejnou hodnotu na více místech. V takovém případě se hodí danou hodnotu pojmenovat, aby bylo zřejmé, co reprezentuje. Zároveň by bylo užitečné ji nadefinovat pouze na jednom místě, abychom její hodnotu mohli jednoduše manuálně změnit bez toho, abychom při tom museli upravovat všechna místa, kde danou hodnotu používáme.

Pomocí příkazu `#define <název> <hodnota>` můžeme vytvořit **makro** (*macro*) s daným názvem a hodnotou. Pokud preprocesor v kódu od řádku s `#define` do konce zdrojového kódu narazí na název makra, tak tento název nahradí hodnotou makra (opět se jedná o prosté textové nahrazení, tedy `Ctrl+C -> Ctrl+V`). Zkuste si například, co vypíše tento program:

```
#include <stdio.h>

#define CENA 25

int main() {
    printf("Cena je %d\n", CENA);
    printf("Dvojnasobek ceny je %d\n", CENA * 2);

    return 0;
}
```

Představte si, že hodnotu tohoto makra používáme v programu na stovkách míst. Pokud bychom ji potřebovali změnit, tak stačí změnit jeden řádek s `#define` a preprocesor se poté postará o to, že se hodnota aktualizuje na všech použitých místech.

Makra jsou dle konvence obvykle pojmenována "caps-lockem", tedy velkými písmeny (respektive stylem screaming snake case).

Je třeba brát na vědomí, že preprocesor opravdu dělá pouhé textové nahrazení. Například následující kód tak nedává smysl:

```
#define CENA 25
int main() {
    CENA = 0;
    return 0;
}
```

protože po spuštění preprocesoru se z něj stane tento (nesmyslný) kód:

```
int main() {
    25 = 0;
```

```
    return 0;
}
```

Makra s parametry

Makra můžou také obsahovat parametry:

```
#define <název_makra>(<param1>, <param2>, ...) <hodnota_makra>
```

Tyto parametry můžete použít pro definici hodnoty. Nicméně je opět třeba dát pozor na to, že preprocesor pracuje pouze s textem, nerozumí jazyku *C*. Parametry tak jsou předávány čistě jako text, je tak potřeba dávat si pozor na několik věcí:

- Priorita operátorů - pokud bychom chtěli vytvořit makro pro výpočet druhé mocniny, můžeme ho napsat například takto:

  ```
  #define MOCNINA(a) a * a
  ```

  Pokud však takovéto makro použijeme s nějakým komplexním výrazem, nemusíme dosáhnout kýženého výsledku kvůli priority operátorů:

  ```
  #include <stdio.h>

  #define MOCNINA(a) a * a

  int main() {
      printf("%d\n", MOCNINA(1 + 1));
      return 0;
  }
  ```

  Řádek s `printf` totiž preprocesor změní na `printf("%d\n", 1 + 1 * 1 + 1);`, což jistě není to, co jsme chtěli. Proto je dobré při použití maker s parametry obalovat jednotlivé parametry závorkami:

  ```
  #define MOCNINA(a) (a) * (a)
  ```

  Pak by zde již došlo k úpravě na `printf("%d\n", (1 + 1) * (1 + 1));`, což vrátí druhou mocninu hodnoty `1 + 1`, tedy `4`.

- Vedlejší efekty - pokud mají argumenty předávané do makra nějaké vedlejší efekty , je třeba si dávat pozor na to, že makro může jednoduše takovýto argument rozkopírovat a tím pádem vedlejší efekt provést vícekrát. Například při použití makra `MOCNINA` výše by zde došlo k dvojnásobené inkrementaci proměnné `x`:

  ```
  #include <stdio.h>

  #define MOCNINA(a) a * a

  int main() {
      int x = 0;
      int mocnina = MOCNINA(x++);
      printf("%d\n", x);
  ```

```
    return 0;
}
```

Do maker tak radši nedávejte argumenty, které způsobují vedlejší efekty.

Makra vs globální proměnné

Globální proměnné jsou také pojmenované hodnoty definované na jednom místě, proč tedy potřebujeme makra? Je to z několika důvodů:

- Globální proměnné zabírají místo v paměti programu a zároveň zvyšují velikost spustitelného souboru, protože v něm musí být uložena jejich iniciální hodnota (pokud to tedy není `0`). Makra se pouze textově nahradí během překladu programu, takže samy o sobě žádnou paměť nezabírají.
- Makra s parametry umožňují definici hodnot či textu závislou na použitých parametrech, což globální proměnné neumožňují.
- Konstantní globální proměnné nelze použít například pro určení velikosti statických polí.

Nicméně, makra jsou občas problémová kvůli toho, že se nahrazují čistě jako text. Pokud je to tedy možné, zkuste raději použít pro definici konstant v kódu konstantní globální proměnné.

Podmíněný překlad

Makra mohou také být použity k tzv. **podmíněnému překladu** (*conditional compilation*). Pomocí příkazů preprocesoru jako `#ifdef` nebo `#if` můžete přeložit kus kódu pouze, pokud je nadefinované určité makro (popřípadě pouze pokud má určitou hodnotu). Toho se běžně využívá například pro tvorbu programů, které jsou kompatibilní s více operačními systémy (např. funkce může mít jinou implementaci pro Linux a jinou pro Windows).

V UPR se s podmíněným překladem nesetkáme, více se o něm můžete dozvědět například zde .

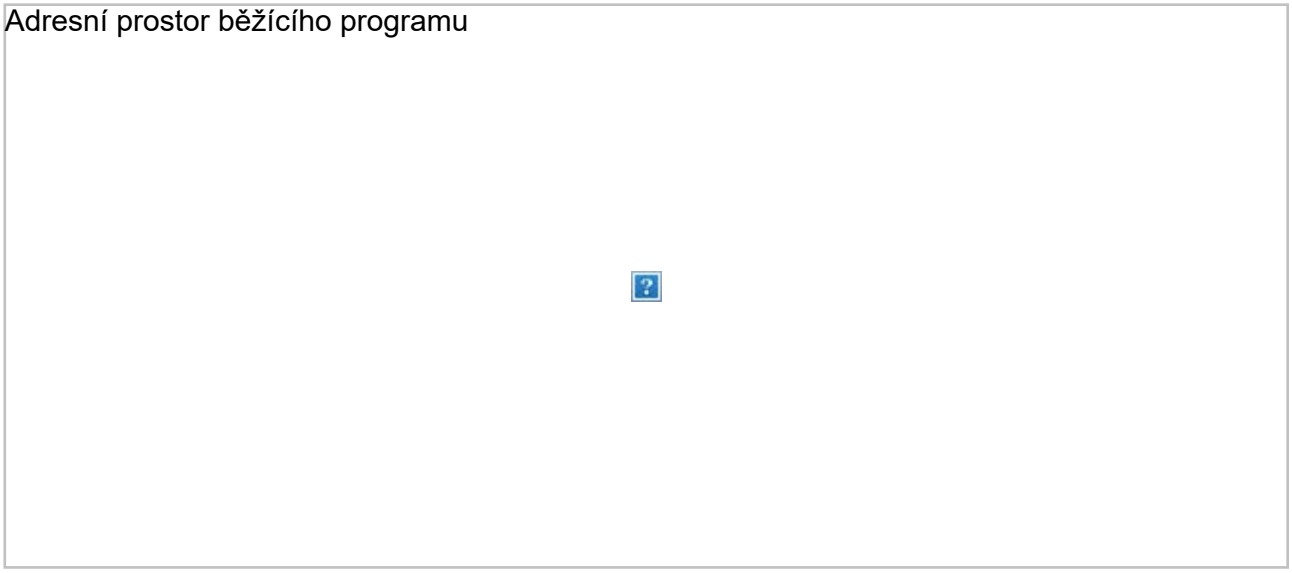
Práce s pamětí

V sekci o paměti jsme se dozvěděli, že operační paměť počítače lze adresovat pomocí číselných adres. Prozatím jsme nicméně v našich programech s žádnými adresami explicitně nepracovali, pouze jsme vytvářeli proměnné, jejichž paměť byla spravována automaticky. V této sekci se dozvíte základy toho, jak tzv. **správa paměti** (*memory management*) funguje.

Adresní prostor programu

Když spustíte svůj program, tak pro něj operační systém vytvoří tzv. **adresní prostor** (*address space*), což je oblast paměti, se kterou program může pracovat.[1] Tato oblast je typicky rozdělena na několik částí, z nichž každá slouží pro různé typy dat:

[1] Díky mechanismu virtuální paměti je tento prostor soukromý pro váš běžící program - ostatní běžící programy do něj nemají přístup, pokud jim to explicitně nepovolíte.



- **Instrukce programu**: do této části paměti se při spuštění programu zkopírují jeho instrukce ze spustitelného souboru na disku. Procesor poté čte instrukce, které má vykonat, právě z této části paměti. Tato paměť je obvykle chráněna proti zápisu a slouží pouze pro čtení.
- **Zásobník**: tato část uchovává automaticky spravovaná data, zejména lokální proměnné a parametry funkcí. Tuto oblast popisuje sekce o automatické paměti.
- **Halda**: tato část můžete využít k manuální alokaci paměti. To nám umožňují ukazatele , díky kterým můžeme explicitně pracovat s adresami v paměti. Tuto oblast adresního prostoru popisuje sekce o dynamické paměti.
- **Globální data**: tato část obsahuje globální proměnné, které žijí po celou dobu trvání programu.

Automatická paměť

Zatím jsme používali (lokální) proměnné, které vznikají a zanikají uvnitř funkcí. Nemuseli jsme se tedy nijak starat o to, kde existují v paměti. Lokální proměnné se ukládají do oblasti v paměti, kterou nazýváme **zásobník** (*stack*). Každý běžící program má vyhrazen určitou oblast adresovatelné paměti, která je použita právě jako zásobník.

Při každém zavolání funkce vznikne na zásobníku tzv. **zásobníkový rámec** (*stack frame*). V tomto rámci je vyhrazena paměť pro lokální proměnné volané funkce a také pro její parametry . Rámec vzniká při zavolání funkce, v jednu chvíli tak na zásobníku může existovat více rámců (s různými hodnotami proměnných a parametrů) pro stejnou funkci. Rámce vznikají v paměti jeden za druhým, a jsou uvolněny v momentě, kdy se jejich funkce dokončí.[1]

[1]Rámce tak mohou vznikat nebo zanikat pouze na konci zásobníku, ne uprostřed. Proto se tato oblast nazývá zásobník, podle datové struktury , která má tuto vlastnost.

Při zavolání funkce se do paměti určené pro jednotlivé parametry v rámci nakopírují hodnoty argumentů předaných při volání funkce. Jakmile funkce skončí, tak je rámec, spolu s pamětí lokálních proměnných, uvolněn[2].

[2]Uvolnění zde znamená pouze to, že program bude pokládat danou paměť za volnou k dalšímu použití. Pokud tak například funkce bude mít lokální proměnnou s hodnotou `5` a vykonání funkce skončí, tato hodnota v paměti zůstane, dokud nebude přepsána příštím zavoláním funkce.

V následující animaci můžete vidět sekvenci volání funkcí. Ve sloupci vpravo je zobrazen stav zásobníku při provádění tohoto programu. Modře jsou v něm znázorněny hodnoty parametrů a červeně hodnoty lokálních proměnných. Můžete si všimnout, že lokální proměnné mají nedefinovanou hodnotu , dokud do nich není nějaká

hodnota zapsána, nicméně paměť pro ně již existuje od začátku provádění funkce.

V animaci si můžete všimnout, že rámce vždy vznikají a zanikají pouze na konci zásobníku.[3] Uhodnete, jaké číslo tento program vypíše?

[3] Z historických důvodů zásobník roste "dolů", tj. nové rámce se vytvářejí na nižší adrese v paměti.

Výhody automatické paměti

Používání automatické paměti má značné výhody:

- Nemusíme se starat o to, jak je paměť alokována a uvolňována, vše za nás řeší překladač, který generuje instrukce pro vytváření a uvolňování rámců při volání/dokončení provádění funkce.
- Alokace i uvolnění paměti je velmi rychlá. Jde v podstatě o provedení jediné instrukce, která si pamatuje, kde zrovna zásobník "končí" v paměti.

Pokud tedy nepotřebujete žádnou složitější funkcionalitu, první volbou by mělo být právě použití automatické paměti (tedy lokálních proměnných).

Nevýhody automatické paměti

Automatická paměť je sice velmi užitečná, nicméně někdy potřebujeme použít i jiné typy paměti, protože automatická paměť má i určité nedostatky:

- Maximální velikost zásobníku je omezena [4]. Nemůžeme tak na něm naalokovat větší množství paměti.

  [4] Obvykle jde o jednotky KiB/MiB.
- Počet a velikost lokálních proměnných je "zadrátována" do programu během jeho překladu. Nemůžeme tak naalokovat paměť s velikostí závislou na vstupu programu. Například pokud uživatel zadá číslo `n` a my bychom chtěli vytvořit paměť pro `n` čísel, tak nestačí použití zásobníku.
- Paměť lokálních proměnných a parametrů je uvolněna při dokončení provádění funkce. Jediným způsobem, jak předat hodnotu z volání funkce, je pomocí návratového typu, lze takto tedy vrátit pouze jednu hodnotu. Nelze tak jednoduše sdílet hodnoty mezi funkcemi, protože paměť lokálních proměnných je po dokončení volání funkce uvolněna a nelze ji tak použít z volající funkce.
- Argumenty předávané do funkcí se kopírují do zásobníkového rámce volané funkce a návratová hodnota se zase kopíruje zpět do rámce volající funkce. Toto kopírování může být zbytečně pomalé pro hodnoty zabírající velký počet bytů.

Abychom mohli alokovat větší množství paměti či jednoduššeji sdílet hodnoty proměnných mezi funkcemi, tak musíme mít možnost alokovat a uvolňovat paměť manuálně. Nejprve ale potřebujeme způsob, jak pracovat přímo s adresami v paměti, k čemuž slouží ukazatele .

Ukazatele

Abychom v *C* mohli manuálně pracovat s pamětí, potřebujeme mít možnost odkazovat se na jednotlivé hodnoty v paměti pomocí adres . Adresa je číslo, takže bychom mohli pro popis adres používat například datový typ `unsigned`

`int`[1]. To by ale nebyl dobrý nápad, protože tento datový typ neumožňuje provádět operace, které bychom s adresami chtěli dělat (načíst hodnotu z adresy či zapsat hodnotu na adresu), a naopak umožňuje provádět operace, které s adresami dělat nechceme (například násobení či dělení adres obvykle nedává valný smysl).

[1]Nejnižší možná adresa je `0`, takže záporné hodnoty nemá cenu reprezentovat.

Z tohoto důvodu *C* obsahuje datový typ, který je interpretován jako adresa v paměti běžícího programu. Nazývá se **ukazatel** (*pointer*). Kromě toho, že reprezentuje adresu, tak každý datový typ ukazatele také obsahuje informaci o tom, jaký typ hodnoty je uložen v paměti na adrese obsažené v ukazateli. Poté říkáme, že ukazatel "ukazuje na" daný datový typ.

Abychom vytvořili datový typ ukazatele, vezmeme datový typ, na který bude ukazovat, a přidáme za něj hvezdičku (`*`). Takto například vypadá proměnná datového typu "ukazatel na `int`"[2]:

[2]Je jedno, jestli hvězdičku napíšete k datovému typu (`int* p`) anebo k názvu proměnné (`int *p`), bílé znaky jsou zde ignorovány. Pozor však na vytváření více ukazatelů na jednom řádku.

```
int* ukazatel;
```

Je důležité si uvědomit, co tato proměnná reprezentuje. Datový typ `int*` zde říká, že v proměnné `ukazatel` bude uloženo číslo, které budeme interpretovat jako adresu. V paměti na této adrese poté bude ležet číslo, které budeme interpretovat jako datový typ `int` (celé číslo se znaménkem).

Ukazatele lze libovolně "vnořovat", tj. můžeme mít například "ukazatel na ukazatel na celé číslo" (`int**`). Ukazatel ale i tehdy bude prostě číslo, akorát ho budeme interpretovat jako adresu na adresu. Pro procvičení je níže uvedeno několik datových typů spolu s tím, jak je interpretujeme.

- `int` - interpretujeme jako celé číslo
- `int*` - interpretujeme jako adresu, na které je uloženo celé číslo
- `float*` - interpretujeme jako adresu, na které je uloženo desetinné číslo
- `int**` - interpretujeme jako adresu, na které je uložena adresa, na které je uloženo celé číslo

Někdy chceme použít "univerzální" ukazatel, který prostě obsahuje adresu, bez toho, abychom striktně určovali, jaká hodnota na dané adrese bude uložena. V tom případě můžeme použít datový typ `void*`.

> Velikost všech ukazatelů v programu je stejná a je daná použitým operačním systémem a překladačem. Ukazatele musí být dostatečně velké, aby zvládli reprezentovat libovolnou adresu, která se v programu může vyskytnout. Na vašem počítači to bude nejspíše 8 bytů, protože pravděpodobně používáte 64-bitový systém.

Inicializace ukazatele

Jelikož před spuštěním programu nevíme, na jaké adrese budou uloženy hodnoty, které nás budou zajímat, tak obvykle nedává smysl inicializovat ukazatel na konkrétní adresu (např. `int* p = 5;`). Pro inicializaci ukazatele tak existuje několik standardních možností:

- **Inicializace na nulu**: Pokud chceme vytvořit "prázdný" ukazatel, který zatím neukazuje na žádnou validní adresu, tak se dle konvence inicializuje na hodnotu `0`. Takovému ukazateli se pak říká **nulový ukazatel** (*null pointer*). Jelikož datový typ výrazu `0` je `int`, tak před přiřazením této hodnoty do ukazatele jej musíme

přetypovat na datový typ cílového ukazatele:

```
float* p = (float*) 0;
```

Jelikož tento typ inicializace je velmi častý, standardní knihovna *C* obsahuje makro `NULL`, které konverzi nuly na ukazatel provede za vás. Můžete jej najít například v souboru `stdlib.h`:

```
#include <stdlib.h>
float* p = NULL;
```

- **Využití alokační funkce**: Pokud budete alokovat paměť manuálně, tak použijete funkce, které vám hodnotu ukazatele vrátí jako svou návratovou hodnotu.
- **Využití operátoru adresy**: Pokud chcete ukazatel nastavit na adresu již existující hodnoty v paměti, můžete použít **operátor adresy** (*address-of operator*). Ten má syntaxi `&<proměnná>`. Tento operátor se vyhodnotí jako adresa předané proměnné[3]:

[3]Všimněte si, že pro výpis ukazatelů ve funkci `printf` se používá `%p` místo `%d`.

```
#include <stdio.h>

int main() {
    int x = 1;
    int* p = &x;

    printf("%d\n", x);  // hodnota proměnné x
    printf("%p\n", p);  // adresa v paměti, kde je uložena proměnná x

    return 0;
}
```

Výraz předaný operátoru `&` se musí vyhodnotit na něco, co má adresu v paměti (většinou to bude proměnná). Nedává smysl použít něco jako `&5`, protože 5 je číselná hodnota, která nemá žádnou adresu v paměti.

Při použití tohoto operátoru je také třeba dávat si pozor na to, aby hodnota v paměti, jejíž adresu použitím `&` získáme, stále existovala, když se budeme později snažit k této adrese pomocí ukazatele přistoupit. V opačném případu by mohlo dojít k paměťové chybě .

Přístup k paměti pomocí ukazatele

Když už máme v ukazateli uloženou nějakou (validní) adresu v paměti, tak k této paměti můžeme přistoupit pomocí operátoru **dereference**. Ten má syntaxi `*<výraz typu ukazatel>`. Při použití tohoto operátoru na ukazateli program přečte adresu v ukazateli, podívá se do paměti a načte hodnotu uloženou na této adrese. Podle toho, na jaký datový typ ukazatel ukazuje, se načte odpovídající počet bytů z paměti:

```
#include <stdio.h>

int main() {
    int cislo = 1;
```

```
    int* ukazatel = &cislo;

    printf("%p\n", ukazatel);
    printf("%d\n", *ukazatel);
    printf("%d\n", cislo);

    return 0;
}
```

V tomto programu se do proměnné `ukazatel` uloží adresa proměnné `cislo`, a poté dojde k načtení hodnoty (`*ukazatel`) této proměnné z paměti přes adresu uloženou v ukazateli.

Pokud chceme do adresy uložené v ukazateli naopak nějakou hodnotu zapsat, tak můžeme operátor dereference použít také na levé straně operátoru zápisu . Uhodnete, co vypíše tento program?

```
#include <stdio.h>

int main() {
    int cislo = 1;
    int* ukazatel = &cislo;
    *ukazatel = 5;

    printf("%d\n", cislo);

    return 0;
}
```

Pokud provádíte operace s přímo s proměnnou ukazatele, budete vždy pracovat "pouze" s adresou, která je v něm uložena. Pokud chcete načíst nebo změnit hodnotu, která v paměti leží na adrese uložené v ukazateli, musíte použít operátor dereference.

Aritmetika s ukazateli

Abychom se mohli v paměti "posouvat" o určitý kus dopředu či dozadu (relativně k nějaké adrese), můžeme k ukazatelům přičítat či odčítat čísla. Toto se označuje jako **aritmetika s ukazateli** (*pointer arithmetic*). Tato aritmetika má důležité pravidlo – pokud k ukazateli na konkrétní datový typ přičteme hodnotu `n`, tak se adresa v ukazateli zvýší o `n`-násobek velikosti datového typu, na který ukazatel ukazuje. Při aritmetice s ukazateli se tak neposouváme po jednotlivých bytech, ale po celých hodnotách daného datového typu[4].

[4] Z toho vyplývá, že aritmetiku nemůžeme provádět nad ukazateli `void*`, protože ty neukazují na žádný konkrétní datový typ.

Například, pokud bychom měli ukazatel `int* p` s hodnotou `16` (tj. "ukazuje" na adresu `16`) a velikost `intu` by byla `4`, tak výraz `p + 1` bude ukazatel s hodnotou `20`, výraz `p + 2` bude ukazatel s adresou `24` atd.

Je důležité rozlišovat , jestli při použití sčítání/odčítání pracujeme s hodnotou ukazatele anebo s hodnotou na adrese, která je v ukazateli uložena:

```
int x = 1;
int* p = &x;

*p += 1;    // zvýšili jsme hodnotu na adrese v `p` (tj. proměnnou `x`) o `1`
p += 1;     // zvýšili jsme adresu v `p` o `4` (tj. p nyní už neukazuje na `x`)
```

K čemu je aritmetika s ukazateli užitečná se dozvíte v sekci o práci s více proměnnými zároveň.

Kromě dereference a aritmetiky lze s ukazateli provádět také porovnávání (klasicky pomocí operátorů == nebo >). Díky toho můžeme například zjistit, jestli se dvě adresy rovnají.

Využití ukazatelů

Jak se dozvíte v následující sekci, ukazatele jsou nezbytné pro manuální alokaci paměti. Hodí se také při práci s více proměnnými zároveň. Kromě toho je ale lze použít také například v následujících situacích, které všechny souvisí s předáváním adres (ukazatelů) do funkcí:

- **Změna vnějších hodnot zevnitř funkce** - hodnoty argumentů předávaných při volání funkcí se do funkce kopírují, nelze tak jednoduše zevnitř funkce měnit hodnoty proměnných, které existují mimo danou funkci. To je sice samo o sobě vhodná vlastnost, protože pokud bude funkce měnit pouze své lokální proměnné, případně parametry, tak bude jednodušší se v ní vyznat. Nicméně, někdy opravdu chceme ve funkci změnit hodnoty externích proměnných. Toho můžeme dosáhnout tak, že si do funkce místo hodnoty proměnné pošleme její adresu v ukazateli, a pomocí této adresy pak hodnotu proměnné změníme. Takto například můžeme vytvořit funkci, která vezme adresy dvou proměnných a prohodí jejich hodnoty:

```
#include <stdio.h>
void vymen(int* a, int* b) {
    int docasna_hodnota = *a;
    *a = *b;
    *b = docasna_hodnota;
}
int main() {
    int x = 5;
    int y = 10;
    vymen(&x, &y);
    printf("Po prehozeni: x=%d, y=%d\n", x, y);
    return 0;
}
```

- **Vrácení více návratových hodnot** - posílání adres proměnných do funkce můžeme využít také k tomu, abychom z funkce vrátili více než jednu návratovou hodnotu (do adres uložených v parametrech totiž můžeme zapsat "návratové" hodnoty). Toho bychom však měli využívat pouze, pokud je to opravdu nezbytné. Takovéto funkce je totiž složitější volat a nejsou čisté, protože obsahují vedlejší efekt - mění externí stav programu.
- **Sdílení hodnot bez kopírování** - pokud bychom měli proměnné, které v paměti zabírají velké množství bytů (například struktury), a předávali je jako argumenty funkci, tak může být zbytečně pomalé je pokaždé kopírovat. Pokud do funkce pouze předáme jejich adresu, tak dojde ke kopii pouze jednoho čísla s adresou, nezávisle na tom, jak velká je proměnná, která je na dané adrese uložena. Ukazatele tak můžeme použít ke sdílení hodnot v paměti mezi funkcemi bez toho, abychom je kopírovali.

Konstantní ukazatele

Pokud použijeme klíčové slovo `const` v kombinaci s ukazateli, je potřeba si dávat pozor na to, k čemu se tohle klíčové

slovo váže. To závisí na tom, zda je `const` v datovém typu před nebo za hvězdičkou. Zde jsou možné kombinace, které můžou vzniknout u jednoduchého ukazatele:

- `int*` - ukazatel na celé číslo. Adresu v ukazateli lze měnit, hodnotu čísla na adrese v ukazateli také lze měnit.
- `const int*` - ukazatel na konstantní celé číslo. Adresu v ukazateli lze měnit, hodnotu čísla na adrese v ukazateli nikoliv.
- `int const *` - konstantní ukazatel na celé číslo. Adresu v ukazateli nelze měnit, hodnotu čísla na adrese v ukazateli lze měnit.
- `const int const *` - konstantní ukazatel na konstantní celé číslo. Adresu v ukazateli nelze měnit, hodnotu čísla na adrese v ukazateli také nelze měnit.

Definice více ukazatelů najednou

Pokud byste chtěli vytvořit více ukazatelů najednou , musíte si dát pozor na to, že v tomto případě se hvězdička vztahuje pouze k jednomu následujícímu názvu proměnné. Tento kód tak vytvoří ukazatel s názvem `x`, a dvě celá čísla s názvy `y` a `z`:

```
int* x, y, z;
```

Pokud byste chtěli vytvořit tři ukazatele, musíte dát hvězdičku před každý název proměnné:

```
int* x, *y, *z;
```

Dynamická paměť

Už víme, že pomocí automatické paměti na zásobníku nemůžeme alokovat velké množství paměti a nemůžeme ani alokovat paměť s dynamickou velikostí (závislou na velikosti vstupu programu). Abychom tohoto dosáhli, tak musíme použít jiný mechanismus alokace paměti, ve kterém paměť alokujeme i uvolňujeme manuálně.

Tento mechanismus se nazývá **dynamická alokace paměti** (*dynamic memory allocation*). Pomocí několika funkcí standardní knihovny *C* můžeme naalokovat paměť s libovolnou velikosti. Tato paměť je alokována v oblasti paměti zvané **halda** (*heap*). Narozdíl od zásobníku, prvky na haldě neleží striktně za sebou, a lze je tak uvolňovat v libovolném pořadí. Můžeme tak naalokovat paměť libovolné velikosti, která přežije i ukončení vykonávání funkce, díky čemuž tak můžeme sdílet (potenciálně velká) data mezi funkcemi. Nicméně musíme také tuto paměť ručně uvolňovat, protože (narozdíl od zásobníku) to za nás nikdo neudělá.

Alokace paměti

K naalokování paměti můžeme použít funkci `malloc` (*memory alloc*), která je dostupná v souboru `stdlib.h` ze standardní knihovny *C*. Tato funkce má následující signaturu[1]:

[1]Datový typ `size_t` reprezentuje bezznaménkové celé číslo, do kterého by měla jít uložit velikost největší možné hodnoty libovolného typu. Často se používá pro indexaci polí.

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

Velikost alokované paměti

Parametr `size` udává, kolik bytů paměti se má naalokovat. Tuto velikost můžeme "tipnout" manuálně, nicméně to není moc dobrý nápad, protože bychom si museli pamatovat velikosti datových typů (přičemž jejich velikost se může lišit v závislosti na použitém operačním systému či překladači!). Abychom tomu předešli, tak můžeme použít operátor `sizeof`, kterému můžeme předat datový typ[2] a tento výraz se poté vyhodnotí jako velikost daného datového typu:

[2]Případně výraz, v tom případě si `sizeof` vezme jeho datový typ.

```
#include <stdio.h>
int main() {
    printf("Velikost int je: %lu\n", sizeof(int));
    printf("Velikost int* je: %lu\n", sizeof(int*));
    return 0;
}
```

Návratový typ `void*` reprezentuje ukazatel na libovolná data. Funkce `malloc` musí fungovat pro alokaci libovolného datového typu, proto musí mít návratový typ právě univerzální ukazatel `void*`. Při zavolání funkce `malloc` bychom měli tento návratový typ přetypovat na ukazatel na datový typ, který alokujeme.

Při zavolání `malloc`u dojde k naalokování `size` bytů na haldě. Adresa prvního bytu této naalokované paměti se poté vrátí jako návratová hodnota `malloc`u. Zde je ukázka programu, který naalokuje paměť pro jeden `int` ve funkci, adresu naalokované paměti poté vrátí jako návratovou hodnotu a naalokovaná paměť je poté přečtena ve funkci `main`:

```
#include <stdlib.h>

int* naalokuj_pamet() {
    int* pamet = (int*) malloc(sizeof(int));
    *pamet = 5;
    return pamet;
}
int main() {
    int* pamet = naalokuj_pamet();
    printf("%d\n", *pamet);
    return 0;
}
```

Iniciální hodnota paměti

Stejně jako u lokálních proměnných platí, že hodnota naalokované paměti je nedefinovaná. Než se tedy hodnotu dané paměti pokusíte přečíst, musíte jí nainicializovat zápisem nějaké hodnoty! Jinak bude program obsahovat nedefinované chování .

Pokud byste chtěli, aby naalokovaná paměť byla rovnou při alokaci vynulována (všechny byty nastavené na hodnotu `0`), můžete místo funkce `malloc` použít funkci `calloc`[3].

[3]Pozor však na to, že tato funkce má jiné parametry než `malloc`. Očekává počet hodnot, které se mají naalokovat, a velikost každé hodnoty.

Případně můžete použít užitečnou funkci `memset`, která vám vyplní blok paměti zadaným bytem.

Uvolnění paměti

S velkou mocí přichází i velká zodpovědnost, takže při použití dynamické paměti sice máme více možností než při použití zásobníku, ale zároveň **MUSÍME** tuto paměť korektně uvolňovat (což se u automatické paměti provádělo automaticky). Pokud bychom totiž paměť neustále pouze alokovali a neuvolňovali, tak by nám brzy došla .

Abychom paměť naalokovanou pomocí funkcí `malloc` či `calloc` uvolnili, tak musíme použít funkci `free`:

```
#include <stdlib.h>

int main() {
    int* p = (int*) malloc(sizeof(int)); // alokace paměti
    *p = 0;                              // použití paměti
    free(p);                             // uvolnění paměti

    return 0;
}
```

Jako argument této funkci musíme předat ukazatel navrácený z volání `malloc`/`calloc`. Nic jiného do této funkce nedávejte, uvolňovat můžeme pouze dynamicky alokovanou paměť! Nevolejte `free` s adresami např. lokálních proměnných[4].

[4]Je však bezpečné uvolnit "nulový ukazatel", tj. `free(NULL)` je validní (v tomto případě funkce nic neudělá).

Jakmile se paměť uvolní, tak už k této paměti nesmíte přistupovat! Pokud byste se pokusili přečíst nebo zapsat uvolněnou paměť, tak dojde k nedefinovanému chování . Nesmíte ani paměť uvolnit více než jednou.

Při práci s dynamickou (manuální) pamětí tak dbejte zvýšené opatrnosti a ideálně používejte při vývoji Address sanitizer. (Neúplný) seznam věcí, které se můžou pokazit, pokud kombinaci manuální alokace a uvolňování paměti pokazíte, naleznete zde .

Alokace více hodnot zároveň

Jak jste si mohli všimnout ze signatury funkce `malloc`, můžete jí dát libovolný počet bytů. Nemusíte se tak omezovat velikostí základních datových typů, můžete například naalokovat paměť pro 5 `int`ů zároveň, které poté budou ležet za sebou v paměti a bude tak jednoduché k nim přistupovat v cyklu. Jak tento koncept funguje se dozvíte v sekci o dynamických polích .

Globální paměť

Posledním základním typem paměti je tzv. globální (nebo také statická) paměť. Tato paměť je specifická tím, že vzniká při spuštění programu a zaniká při jeho ukončení, lze ji tak používat během celé délky běhu programu.

Globální proměnné jsou umístěny v globální paměti. Je dobré si uvědomit, že tyto proměnné zároveň zabírají místo ve spustitelném souboru na disku, protože v něm musí být uložena jejich iniciální hodnota[1].

[1]Pokud tedy nejsou inicializované na nulu).

Pole

Počítače slouží k (rychlému) zpracování velkého objemu dat, běžně tak v programech potřebujeme zpracovávat mnoho proměnných najednou. Například:

- V dokumentu otevřeném ve Wordu můžete mít uložené tisíce různých znaků.
- Na server v online hře může v danou chvíli být připojené velké množství hráčů a všem musíme posílat informace o stavu hry.
- Obrázky se běžně v programech reprezentují jako 2D mřížka pixelů. Například černobílý obrázek s rozměry `1024x1024` vyžaduje držet v paměti `1048576` bytů (čísel) reprezentujících jednotlivé pixely.

Asi si dovedete představit, že například pro reprezentaci obrázku bychom si s proměnnými, které jsme používali doposud, nevystačili. Pokud bychom po jedné vytvářeli proměnné `pixel1`, `pixel2`, `pixel3`, tak by jednak byl náš zdrojový kód obrovský a nedalo by se v něm vyznat, a také bychom nemohli mít velikost obrázku závislou na vstupu programu, protože počet proměnných by byl "zadrátovaný" ve zdrojovém kódu programu. Chtěli bychom tak mít možnost napsat kód, který bude umět zpracovat 1, 2, 100 nebo 1000 hodnot bez toho, abychom tento kód museli jakkoliv měnit.

Asi nejběžnějším a nejjednodušším způsobem, jak v paměti počítače uchovávat větší množství hodnot, je uložit všechny hodnoty jednu po druhé za sebou v paměti[1]. Tento koncept uložení dat se nazývá **pole** (*array*) a je tak běžný, že ho programovací jazyky obvykle přímo podporují, a jazyk *C* není výjimkou.

[1]Způsoby, jak v paměti počítače uchovávat komplexní a rozsáhlá data, se nazývají datové struktury . Pole je jednou z nejjednodušších datových struktur.

V následujících sekcích se dozvíte, jak s poli pracovat, jak je vytvořit v automatické a dynamické paměti a jak lze v počítači reprezentovat vícerozměrná pole.

Statické pole

Pole v automatické paměti[1] (na zásobníku) se označují jako **statická pole** (*static arrays*). Můžeme je vytvořit tak, že za název proměnné přidáme hranaté závorky s číslem udávající počet prvků v poli. Takto například vytvoříme pole celých čísel s třemi prvky:

[1]Pole můžeme tímto způsobem vytvořit také v globální paměti.

```
int pole[3];
```

Takováto proměnná bude obsahovat paměť pro 3 celá čísla (tedy nejspíše na vašem počítači dohromady 12 bytů). Počet prvků v poli se označuje jako jeho **velikost** (*size*).

> Pozor na to, že hranaté závorky se udávají za název proměnné, a ne za název datového typu. `int[3] pole;` je tedy špatně.

Čísla takového pole budou v paměti uložena jeden za druhým[2]:

[2]Každý zelený čtverec na tomto obrázku reprezentuje 4 bytů v paměti (velikost jednoho `intu`).

V jistém smyslu je tak pole pouze zobecněním normální proměnné. Pokud totiž vytvoříte pole o velikosti jedna (`int a[1]`), tak v paměti bude reprezentováno úplně stejně jako klasická proměnná (`int a`).

> Pole lze vytvořit také na haldě pomocí dynamické alokace paměti. Všechny níže popsané koncepty jsou platné i pro dynamická pole, nicméně budeme je demonstrovat na statických polích, protože ty je jednodušší vytvořit.

Počítání od nuly

Pozice jednotlivých prvků v poli se označují jako jejich **indexy** (*array indices*). Tyto pozice se číslují od hodnoty `0` (tedy ne od jedničky, jak můžete být jinak zvyklí). První prvek pole je tedy ve skutečnosti na nulté pozici (indexu), druhý na první pozici, atd. (viz obrázek nahoře). **Počítání od nuly** (*zero-based indexing*) je ve světě programování běžné a budete si na něj muset zvyknout. Jeden z důvodů, proč se prvky počítají právě od nuly, se dozvíte níže .

Z tohoto vyplývá jedna důležitá vlastnost - poslední prvek pole je vždy na indexu `<velikost pole> - 1`! Pokud byste se pokusili přistoupit k prvku na indexu `<velikost pole>`, budete přistupovat mimo paměť pole, což pravděpodobně způsobí paměťovou chybu.

Konstantní velikost statického pole

Hodnota zadaná v hranatých závorkách by měla být konstantní (tj. buď přímo číselná hodnota anebo konstantní proměnná). Pokud budete potřebovat pole dynamické velikosti, tak byste měli použít manuální alokaci paměti.

Jazyk *C* od verze *C99* již sice povoluje dávat do hranatých závorek i "dynamické hodnoty":

```
int velikost = ...; // velikost se načte např. ze souboru
int pole[velikost];
```

Nicméně tuto funkcionalitu není vhodné používat. Zásobník má omezenou velikost a není určen pro alokaci velkého množství paměti[3]. Pokud navíc velikost takovéhoto pole může ovlivnit uživatel programu (např. zadáním vstupu), může váš program jednoduše "shodit", pokud by zadal velké číslo a došlo by k pokusu o vytvoření velkého pole na zásobníku. Zkuste se tak vyvarovat používání dynamických hodnot při vytváření polí na zásobníku.

[3]Můžete si například zkusit přeložit následující program:

```
int main() {
    int pole[10000000];
    return 0;
}
```

Při spuštění by měl program selhat na paměťovou chybu, i když váš počítač má pravděpodobně více než `10000000 * 4` (cca `38` MiB) paměti. Pokud chcete alokovat více než několik stovek bytů, použijte raději dynamickou alokaci na haldě.

Inicializace pole

Stejně jako u normálních lokálních proměnných platí, že pokud pole nenainicializujete, tak bude obsahovat nedefinované hodnoty. V takovém případě z pole nesmíte jakkoliv číst, jinak by došlo k nedefinovanému chování ! K inicializaci hodnoty můžete použít složené závorky se seznamem hodnot (oddělených čárkou), které budou do pole

uloženy. Pokud nezadáte dostatek hodnot pro vyplnění celého pole, tak zbytek hodnot bude nulových.

```
int a[3];               // pole bez definované hodnoty, nepoužívat!
int b[3] = {};          // pole s hodnotami 0, 0, 0
int c[4] = { 1 };       // pole s hodnotami 1, 0, 0, 0
int d[2] = { 2, 3 };    // pole s hodnotami 2, 3
```

Hodnot samozřemě nesmíte zadat více, než je velikost pole.

Pokud využijete inicializaci statického pole, můžete vynechat velikost pole v hranatých závorkách. Překladač v tomto případě dopočítá velikost za vás:

```
int p[] = { 1, 2, 3 }; // p je pole s třemi čísly
```

Přístup k prvkům pole

K přístupu k jednotlivým prvkům pole můžeme využít ukazatelů. Proměnná pole se totiž chová jako ukazatel na první prvek (prvek na nultém indexu) daného pole, pomocí operátoru dereference tak můžeme jednoduše přistoupit k prvnímu prvku pole:

```
#include <stdio.h>

int main() {
    int pole[3] = { 1, 2, 3 };
    printf("%d\n", *pole);
    return 0;
}
```

Abychom přistoupili i k dalším prvkům v poli, tak můžeme využít aritmetiky s ukazateli. Pokud chceme získat adresu prvku na `i`-tém indexu, stačí k ukazateli na první prvek přičíst `i`[4]:

[4]Všimněte si, že při použití operátoru dereference zde používáme závorky. Je to z důvodu priority operátorů. Výraz `*pole + 2` by se vyhodnotil jako první prvek z pole `pole` plus `2`, protože `*` (dereference) má větší prioritu než sčítání.

```
#include <stdio.h>

int main() {
    int pole[3] = { 1, 2, 3 };
    printf("%d\n", *(pole + 0));   // první prvek pole
    printf("%d\n", *(pole + 1));   // druhý prvek pole
    printf("%d\n", *(pole + 2));   // třetí prvek pole
    return 0;
}
```

Nyní už možná tušíte, proč se při práci s poli vyplatí počítat od nuly. Prvek na nultém indexu je totiž vzdálen nula prvků od začátku pole. Prvek na prvním indexu je vzdálen jeden prvek od začátku pole atd. Pokud bychom indexovali od jedničky, museli bychom při výpočtu adresy relativně k ukazateli na začátek pole vždy odečíst jedničku, což by bylo nepraktické.

Operátor přístupu k poli

Jelikož je operace přístupu k poli ("posunutí" ukazatele a jeho dereference) velmi běžná (a zároveň relativně krkolomná), *C* obsahuje speciální operátor, který jej zjednodušuje. Tento operátor se nazývá *array subscription operator* a má syntaxi `<výraz a>[<výraz b>]`. Slouží jako zkratka [5] za `*(<výraz a> + <výraz b>)`. Například `pole[0]` je ekvivalentní výrazu `*(pole + 0)`, `pole[5]` je ekvivalentní výrazu `*(pole + 5)` atd:

[5] Takovéto "zkratky", které v programovacím jazyku nepřináší novou funkcionalitu, pouze zkracují či zjednoduššují často používané kombinace příkazů, se označují jako **syntax sugar**.

```
int pole[3] = { 1, 2, 3 };
pole[0] = 5;       // nastavili jsme první prvek pole na hodnotu `5`
int c = pole[2];   // nastavili jsme `c` na hodnotu posledního prvku pole
```

Jelikož je používání hranatých závorek přehlednější než používání závorek a hvězdiček, doporučujeme je používat pro přistupování k prvkům pole, pokud to půjde.

> Pozor na rozdíl mezi tímto operátorem a definicí pole. Obojí sice používá hranaté závorky, ale jinak spolu tyto dvě věci nesouvisejí. Podobně jako se `*` používá pro definici datového typu ukazatele a zároveň jako operátor dereference (navíc i jako operátor pro násobení). Vždy záleží na kontextu, kde jsou tyto znaky použity.

Použití polí s cykly

Pokud bychom k polím přistupovali po individuálních prvcích, tak bychom nemohli využít jejich plný potenciál. I když umíme jedním řádkem kódu vytvořit například 100 různých hodnot (`int pole[100];`), pokud bychom museli psát `pole[0]`, `pole[1]` atd. pro přístup k jednotlivým prvkům, tak bychom nemohli s polem efektivně pracovat. Smyslem polí je zpracovat velké množství dat jednotným způsobem pomocí malého množství kódu. Jinak řečeno, chtěli bychom mít stejný kód, který umí zpracovat pole o velikosti `2` i `1000`. K tomu můžeme efektivně využít cykly.

Velmi často je praktické použít řídící proměnnou cyklu k tomu, abychom pomocí ní indexovali pole. Například, pokud bychom měli pole s velikostí `10`, tak ho můžeme "projít" pomocí cyklu `for`:

```
#include <stdio.h>

int main() {
    int pole[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    for (int i = 0; i < 10; i++) {
        printf("%d ", pole[i]);
    }
    return 0;
}
```

Situace, kdy pomocí cyklu procházíte pole je velmi častý a určitě se s ním mnohokrát setkáte a použijete jej. Zkuste si to procvičit například pomocí těchto úloh.

Předávání pole do funkcí

Při předávání polí do funkcí si musíme dávat pozor zejména na dvě věci.

Převod pole na ukazatel

Už víme, že když předáváme argumenty do funkcí, tak se jejich hodnota zkopíruje. U statických polí tomu tak ovšem není, protože pole můžou být potenciálně velmi velká a provádění kopií polí by tak potenciálně mohlo trvat dlouhou dobu. Když tak použijeme proměnnou pole jako argument při volání funkce, dojde k tzv. **konverzi pole na ukazatel** (*array to pointer decay*). Pole se tak vždy předá jako ukazatel na jeho první prvek:

```
#include <stdio.h>

void vypis_pole(int* pole) {
    printf("%d\n", pole[0]);
}

int main() {
    int pole[3] = { 1, 2, 3 };
    vypis_pole(pole);
    return 0;
}
```

Pro parametry sice můžete použít datový typ pole:

```
void vypis_pole(int pole[3]) { ... }
```

nicméně i v tomto případě se bude takovýto parametr chovat stejně jako ukazatel (v tomto případě tedy `int*`). Navíc překladač ani nebude kontrolovat, jestli do takového parametru opravdu dáváme pole se správnou velikostí. Pro parametry reprezentující pole tak radši používejte ukazatel.

Předávání velikosti pole

Když ve funkci přijmeme jako parametr ukazatel na pole, tak nevíme, kolik prvků v tomto poli je. Tato informace je ale stěžejní, bez ní totiž nevíme, ke kolika prvkům pole si můžeme dovolit přistupovat. Pokud tedy ukazatel na pole předáváme do funkce, je obvykle potřeba zároveň s ním předat i délku daného pole:

```
int secti_pole(int* pole, int velikost) {
    int soucet = 0;
    for (int i = 0; i < velikost; i++) {
        soucet += pole[i];
    }
    return soucet;
}
```

Výpočet velikosti pole

Abyste při změně velikosti statického pole nemuseli ručně jeho velikost upravovat na více místech v kódu, tak můžete ve funkci, kde definujete statické pole, vypočítat jeho velikost pomocí operátoru `sizeof`:

```
int pole[3] = { 1, 2, 3 };
printf("Velikost pole v bytech: %lu\n", sizeof(pole));
```

Abyste zjistili počet prvků ve statickém poli, můžete velikost v bytech vydělit velikostí každého prvku v poli:

```
int pole[3] = { 1, 2, 3 };
printf("Pocet prvku v poli: %lu\n", sizeof(pole) / sizeof(pole[0]));
```

> Operátor `sizeof` bude pro toto použití fungovat pouze pro statické pole a pouze ve funkci, ve které statické pole vytváříte! Pokud pole pošlete do jiné funkce, už z něj bude pouze ukazatel, pro který `sizeof` vrátí velikost

ukazatele (což bude na vašem PC nejspíše `8` bytů).

Dynamické pole

Pole alokovaná na zásobníku by měly mít velikost danou při překladu programu, často ale potřebujeme vytvářet pole v závislosti na vstupu programu (například když načítáme soubor, tak dopředu nevíme, kolik bude mít řádků). Ze sekce o dynamické paměti již víme, jak alokovat libovolné množství paměti na haldě pomocí funkce `malloc`. Pro vytvoření **dynamického pole** (*dynamic array*) tak stačí použít funkci `malloc`. Například pro vytvoření dynamického pole pro `5` celých čísel potřebujeme naalokovat `5 * sizeof(int)` bytů:

```
int* pole = (int*) malloc(5 * sizeof(int));
```

S takovouto pamětí pak můžeme pracovat jako s polem `int`ů o velikosti `5`. Jakmile již takovéto pole nepotřebujeme, nesmíme jej samozřejmě zapomenout uvolnit.

Změna velikosti pole

Občas potřebujeme velikost dynamického pole změnit (obvykle zvětšit). Například pokud vám uživatel zadává na vstupu seznam čísel, na začátku můžete vytvořit paměť pro 10 čísel, ale při zadání 11. čísla musíte tuto paměť zvětšit, jinak byste neměli nové číslo kam zapsat. Tento proces se nazývá **realokace** (*reallocation*) a lze jej provést například následujícím způsobem:

1. Naalokujeme nové dynamické pole o požadované velikosti
2. Zkopírujeme obsah původního pole do nového pole
3. Uvolníme paměť původního pole
4. Upravíme odpovídající ukazatel(e) v programu, aby ukazoval(y) na nově naalokované pole

Pokud se vám toto nechce programovat ručně, tak můžete také použít funkci `realloc` ze standardní knihovny *C*, která to udělá za vás. Tato funkce očekává původní adresu alokace z `malloc`/`calloc` a počet bytů nové alokace.

Cvičení: Zkuste si naprogramovat funkci, která obdrží pole a jeho původní velikost a realokuje ho na novou velikost.

Vícerozměrné pole

Někdy potřebujeme v programech reprezentovat věci, které jsou přirozeně vícerozměrné. Typickým příkladem jsou obrázky, které lze reprezentovat jako dvourozměrnou mřížku pixelů (jeden rozměr udává řádek a druhý sloupec).

Paměťové adresy však mají pouze jeden rozměr, jelikož jsou reprezentovány jedním číslem. Jak tedy můžeme do jednorozměrné paměti uložit vícerozměrnou hodnotu? Způsobů je více, nicméně asi nejjednodušší je prostě "vyskládat" jednotlivé rozměry (dimenze) v paměti za sebou, jeden rozměr za druhým. Pokud bychom například měli dvojrozměrnou mřížku[1] s rozměry `5x5`, můžeme ji reprezentovat tak, že nejprve do paměti uložíme první řádek, poté druhý řádek atd.:

[1] Reprezentující například obrázek či matici.

Tento koncept se označuje jako **vícerozměrné pole** (*multidimensional array*).

Inicializace vícerozměrných polí

Vícerozměrné pole můžete nainicializovat stejně jako klasické pole. Pro zpřehlednění kódu však také můžete použít složené závorky pro oddělení jednotlivých dimenzí:

```
int pole_2d[3][4] = {
   {0, 1, 2, 3},    // hodnoty pro první řádek
   {4, 5, 6, 7},    // hodnoty pro druhý řádek
   {8, 9, 10, 11}   // hodnoty pro třetí řádek
};
```

Způsob vyskládání dimenzí

Je na nás, v jakém pořadí jednotlivé dimenze do paměti uložíme. Pokud bychom se bavili o 2D poli, tak můžeme do paměti uložit řádek po řádku (viz obrázek výše), toto je nazývané jako **row major ordering**. Můžeme ale také do paměti vyskládat sloupec po sloupci, což se nazývá **column major ordering**. Je víceméně jedno, který způsob použijeme, je ale důležité se držet jednoho přístupu, jinak může dojít k záměně indexů. Indexování totiž záleží na tom, jaký způsob vyskládání použijeme. Níže předpokládáme pořadí *row major* .

Indexování

Při práci s dvourozměrným polem bychom chtěli pracovat s dvourozměrným indexem (řádek `i`, sloupec `j`), nicméně při samotném přístupu do paměti pak musíme tento vícerozměrný index převést na 1D index. A naopak, z 1D indexu bychom chtěli mít možnost získat zpět 2D index. Pro výpočet indexů 2D pole s `vyska` řádky a `sirka` sloupci můžeme použít tyto jednoduché vzorce:

- **Převod z 2D do 1D** - abychom se dostali na cílovou pozici, musíme přeskočit `radek` řádků, kde každý řádek má `sirka` prvků, a poté ještě musíme přičíst pozici sloupce (`sloupec`).

  ```
  int index_2d_na_1d(int radek, int sloupec, int sirka) {
      return radek * sirka + sloupec;
  }
  ```

- **Převod z 1D do 2D** - pro převod z 1D indexu zpět na 2D index stačí aplikovat opačný postup. Nejprve vydělíme 1D index počtem sloupců, abychom zjistili, na jakém jsme řádku, a poté použijeme zbytek po dělení, abychom zjistili, na jakém jsme sloupci.

  ```
  void index_1d_na_2d(int index, int sirka, int* radek, int* sloupec) {
      *radek = index / sirka;
      *sloupec = index % sloupec;
  }
  ```

Tento koncept lze zobecnit na libovolně rozměrné pole (3D, 4D, ...).

Vícerozměrné pole v *C*

C obsahuje základní podporu pro vytváření vícerozměrných statických polí . Při vytváření pole stačí použít hranaté závorky pro každou dimenzi pole. Například takto lze vytvořit 2D pole s rozměry `3x3` na zásobníku:

```
int pole[3][3];
```

Výhoda takovýchto polí je, že překladač provede převod z 2D indexu na 1D index za vás, a můžete tak toto pole přímo indexovat vícerozměrným indexem. Například první prvek pole z kódu výše lze nalézt na pozici `pole[0][0]`, poslední na pozici `pole[2][2]`.

Takováto pole jsou v paměti vyskládána postupně dle jednotlivých dimenzí zleva. Nejprve tedy v paměti leží prvek `pole[0][0]`, poté `pole[0][1]`, ..., `pole[1][1]`, `pole[1][2]` atd. Pokud bychom měli 2D pole a první index bychom pokládali za index řádku, tak toto vyskládání odpovídá *row major* pořadí.

Vícerozměrná pole v *C* lze zobecnit do vyšších dimenzí (můžete tak použít například `int pole[3][3][3]` atd.), nicméně je dobré to nepřehánět, aby kód zůstal přehledný.

Vícerozměrné dynamické pole

Pokud potřebujete vícerozměrné pole s dynamickou velikostí , stačí při volání funkce `malloc` vytvořit dostatek paměti pro všechny rozměry. Pokud bychom například chtěli naalokovat paměť pro 2D obrázek s `vyska` řádky a `sirka` řádky, můžeme použít následující volání funkce `malloc`:

```
int* pamet_obrazku = (int*) malloc(vyska * sirka * sizeof(int)));
```

Text

Doposud jsme pracovali zejména s čísly, nyní se podíváme na to, jak můžeme v počítači reprezentovat a pracovat se znaky a obecně s textem. Zpracování textu je obsaženo téměř v každém programu – načítání konfiguračních souborů, zadávání příkazů z terminálu, práce s dokumenty či tabulkami, komunikace po síti a mnoho dalších činností vyžaduje zpracovávat text.

Nejprve si ukážeme, jak v počítači reprezentovat jednotlivé znaky , dále jak z nich vytvořit delší sekvence textu a konečně jak text načítat a vypisovat.

Znaky

Už víme, že v paměti počítače je nakonec vše reprezentováno číslem, a ani textové znaky nejsou výjimkou. Přirozeným způsobem, jak od sebe znaky odlišit, je přiřadit každému znaku jiné číslo, například znak `A` můžeme reprezentovat číslem `0`, znak `B` číslem `1` atd. Kdyby si však každý program(átor) definoval vlastní způsob, jak převádět znaky na čísla, tak by mezi sebou programy nemohly rozumně komunikovat, protože by si nerozuměly.

Z toho důvodu vzniklo za poslední desítky let mnoho **textových kódování** (*character encoding*), které definují, jaká čísla přiřadit jednotlivým znakům. Dnešním de-facto standardem je kódování Unicode , které obsahuje přes sto tisíc různých znaků, od dávných hieroglyfů, přes českou či anglickou abecedu, až po všelijaké emoji. Práce s kódováním Unicode však není v jazyce *C* přímočará, navíc pro naše potřeby vůbec není potřeba[1].

[1]Pokud byste se o kódování znaků a Unicode chtěli dozvědět více, přečtěte si tento článek.

V rámci předmětu UPR si tak vystačíme s kódováním ASCII (American Standard Code for Information Interchange). Toto kódování sice obsahuje pouze 128 znaků (čísla, malá a velká písmena anglické abecedy, interpunkce apod.), nicméně práce s ním je díky tomu velmi jednoduchá. Je navíc podmnožinou Unicode, takže programy, které podporují Unicode kódování, si s ASCII hravě poradí. Tabulku, která uvádí, jak ASCII mapuje jednotlivé znaky na čísla, naleznete např. zde [2].

[2]V tabulce si můžete všimnout, že čísla nejsou znakům přiřazena zcela náhodně, například znaky reprezentující číslice `0` až `9` mají přiřazena čísla ležící za sebou (`48 - 57`), a stejně je tomu i u písmen anglické abecedy. Této vlastnosti můžeme využít pro usnadnění některých textových operací .

ASCII znaky v *C*

Jelikož ASCII "kóduje" pouze 128 znaků, tak pro reprezentaci ASCII znaku by nám stačilo 7 bitů. Nicméně pracovat se sedmibitovými hodnotami by bylo poněkud nepraktické, proto se běžně ASCII znak ukládá do jednobytového (osmibitového) čísla. V *C* se pro reprezentaci jednoho ASCII znaku používá datový typ `char`[3], s kterým jsme se již setkali.

[3]*C* neobsahuje specializovaný typ pro jednobytové celé číslo, `char` tak reprezentuje jak ASCII znak, tak i celé číslo s jedním bytem. Záleží pak na nás, jak budeme hodnotu v `char`u interpretovat - jestli jako celé číslo nebo jako ASCII znak.

Pokud bychom chtěli do proměnné s typem `char` nějaký znak uložit, tak bychom mohli použít přímo jeho číslo z ASCII tabulky :

```
char znak = 65; // tento znak bude reprezentovat písmeno A
```

Nicméně takto by si každý programátor musel nazpaměť pamatovat ASCII tabulku, což je dost nepraktické. *C* tak nabízí zkratku v podobě **znakového literálu** (*char literal*). Pokud napíšete jeden ASCII znak do apostrofů (`'`), tento výraz se vyhodnotí jako ASCII číselná hodnota daného znaku s datovým typem `char`. Obvykle tak znaky v programech zadáváme v apostrofech pro zjednodušení:

```
char znak = 'A'; // tento znak bude reprezentovat písmeno A
```

Pokud bychom si chtěli ověřit, že hodnota tohoto znaku je opravdu `65`, jak udává ASCII, můžeme si ho vypsat na výstup programu jako číslo:

```
#include <stdio.h>

int main() {
    char znak = 'A';
    printf("%d\n", (int) znak);
    return 0;
}
```

Do apostrofů nikdy nedávejte více než jeden znak! Překladač by se snažil takovýto zápis interpretovat jako vícebytový znak, což téměř jistě není to, čeho chcete dosáhnout. Pro práci s textem (více znaky najednou) slouží řetězce. Jedinou výjimkou jsou speciální znaky, které se zapisují pomocí zpětného lomítka, například:

- `'\n'` reprezentuje znak `LF`, který udává, že má dojít k přechodu kurzoru na nový řádek.[4]

  [4]Nepleťte si ho se znakem `'n'`, který reprezentuje klasické písmeno `n` z abecedy.
- `'\t'` reprezentuje znak `TAB`, který reprezentuje tabulátor.
- `'\0'` reprezentuje znak `NUL` s číselnou hodnotou `0`.

Čísla vs znaky

Při používání apostrofů je mimo jiné třeba si dávat pozor na to, jestli pracujeme s číselnou hodnotou nebo se znakem, který reprezentuje nějakou číslici. Například zde:

```
char znak = 9;
```

Nedojde k uložení znaku `9` do proměnné. Bude do ní uložen znak `TAB`, který má v ASCII hodnotu `9` a pomocí apostrofů ho lze zapsat jako `'\t'`. Pokud bychom do znaku chtěli zapsat znak reprezentující číslici `9`, musíme použít buď literál `'9'` nebo číselnou hodnotu `57`, která devítku v ASCII reprezentuje.

Pokud byste chtěli převést ASCII znak číslice na její číselnou hodnotu, stačí od něho odečíst hodnotu `48`, neboli znak `'0'`. `'0' - '0'` je `0`, `'5' - '0'` je `5` atd. To je způsobeno tím, že číslice v ASCII mají přiřazeny sekvenční číselné hodnoty.

Řetězce

Nyní už víme, jak můžeme v *C* pracovat s jednotlivými (ASCII) znaky. Obvykle však chceme pracovat s delšími sekvencemi textu - řádky, větami, odstavci atd. Sekvence textu se v programovacích jazycích obvykle označují jako **řetězce** (*strings*).

Dobrá zpráva je, že pro použití řetězců v *C* už známe vše potřebné – řetězce nejsou nic jiného než pole znaků!

Řetězce v *C*

Teoreticky bychom si mohli navrhnout vlastní způsob, jak řetězce v paměti reprezentovat a jak s nimi pracovat. Nicméně zaběhlým způsobem, jak s ASCII textem v *C* pracovat, a pro který *C* nabízí různé funkce a základní syntaktickou podporu, je použití takzvaných **řetězců zakončených nulou** (*null-terminated strings*). Takto reprezentovaný řetězec není nic jiného než pole znaků, které obsahuje na svém posledním indexu znak `'\0'` (s číselnou hodnotou `0`), který značí konec řetězce. Například řetězec `UPR` by tedy v paměti počítače byl reprezentovaný takto:

Vytvoření řetězce

Pokud bychom chtěli vytvořit řetězec na zásobníku, můžeme vytvořit statické pole, umístit do něho jednotlivé znaky řetězce a za ně přidat znak `'\0`[1]:

[1]Pro výpis řetězce pomocí funkce `printf` můžeme použít `%s`.

```
#include <stdio.h>

int main() {
    char text[4] = {'U', 'P', 'R', '\0'};
    printf("%s\n", text);
    return 0;
}
```

Pokud bychom potřebovali řetězec s dynamickou nebo velkou délkou, můžeme pro vytvoření řetězce samozřejmě použít také dynamickou paměť.

Řetězcový literál

Vytváření řetězců tímto způsobem je nicméně značně zdlouhavé a nepřehledné. Často chceme v programu jednoduše a rychle zapsat krátký textový řetězec tak, aby šel přehledně přečíst. K tomu můžeme využít tzv. **řetězcový literál** (*string literal*). Pokud napíšeme v *C* text do uvozovek, například `"UPR"`, tak se stane následující:

1. Překladač při překladu uloží do výsledného spustitelného souboru pole reprezentující daný řetězec. V tomto případě půjde o pole velikosti 4 s hodnotami `'U'`, `'P'`, `'R'` a `'\0'`. Při spuštění programu se toto pole načte do globální paměti v sekci adresního prostoru, která je určena pouze pro čtení. Do takto vytvořeného řetězce tak nelze zapisovat, lze jej pouze číst[2].

   [2]Tyto řetězce jsou pouze pro čtení zejména z toho důvodu, aby je šlo sdílet. Pokud například v programu použijete třikrát stejný řetězcový literál, překladač může v paměti pole pro tento literál vytvořit pouze jednou, aby ušetřil paměť. Kvůli toho ale musí být řetězce pouze pro čtení, pokud bychom totiž takto sdílený řetězec změnili, změnilo by to i hodnotu všech ostatních literálů, které se vyhodnotí na jeho adresu, což by bylo dost neintuitivní.
2. Samotný výraz literálu se při běhu programu vyhodnotí jako adresa prvního znaku řetězce uloženého v globální paměti.
3. Datový typ literálu bude ukazatel na konstantní znak, tedy `const char*`. Tento datový typ říká, že hodnotu znaku na dané adrese nelze měnit.

Pomocí řetězcového literálu si tak můžeme značne usnadnit zápis řetězců v programech, jelikož nemusíme přemýšlet nad délkou pole, nemusíme pamatovat na umístění znaku `'\0'` na konec řetězce a ani nemusíme obalovat jednotlivé znaky do apostrofů:

```
#include <stdio.h>

int main() {
    const char* text = "UPR";
    printf("%s\n", text);
    return 0;
}
```

Je však třeba pamatovat na to, že takto vytvořené řetězce jsou opravdu pouze pro čtení, a nesmíme tak do nich zapisovat. Pokud je budete ukládat do proměnné, tak použijte datový typ `const char*`, díky kterému vás překladač bude hlídat, abyste se do takovéhoto řetězce omylem nesnažili něco zapsat.

Pokud byste chtěli použít řetězcový literál pro vytvoření řetězce, který lze měnit, můžete ho uložit do proměnné typu `char[]` (tj. pole znaků, které lze měnit):

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

```
int main() {
    char text[] = "UPR";
    text[0] = 'A';
    printf("%s\n", text);
    return 0;
}
```

V takovémto případě se hodnota z literálu překopíruje do proměnné pole znaků na zásobníku.

> Pokud jsou vám řetězcové literály povědomé, je to kvůli toho, že jsme je již mnohokrát využili při volání funkce `printf`.

Víceřádkové řetězcové literály

Pokud budete chtít zapsat řetězcový literál na více řádků kódu, můžete buď na konci každého neukončeného řádku použít znak `\`:

```
const char* veta = "Ahoj \
jmenuji \
se \
Karel";
```

nebo každý řádek samostatně uzavřít v uvozovkách:

```
const char* veta = "Ahoj"
"jmenuji"
"se"
"Karel";
```

Pozor však na to, že v ani jednom ze zmíněných případů nebude součástí řetězce znak odřádkování. Ten musíte vždy přidat explicitně:

```
const char* radky = "radek1\n\
radek2\n\
radek3\n";

// nebo
const char* radky = "radek1\n"
"radek2\n"
"radek3\n";
```

K čemu slouží nulový znak na konci?

U polí je trochu nepraktické to, že pokud je chceme poslat do nějaké funkce, musíme spolu s ukazatelem na první prvek pole předat také jeho velikost , aby funkce věděla, ke kolika prvkům si může dovolit přistoupit. Jiným způsobem, jak určit velikost pole, je zvolit si speciální hodnotu, která bude značit konec pole. Když kód, který s takovýmto polem bude pracovat, na tuto speciální hodnotu narazí, tak bude vědět, že dále v paměti již pole nepokračuje.

Tento mechanismus je využit právě u řetězců zakončených nulou, kde onou speciální hodnotou je právě tzv. `NUL` znak, který má číselnou hodnotu `0`. Například při procházení řetězce v cyklu tak nemusíme dopředu znát jeho délku, stačí cyklus ukončit, jakmile narazíme na znak `'\0'`. Například funkce pro spočtení délky řetězce by mohla vypadat takto[3]:

[3]Všimněte si, že tato funkce bere ukazatel na konstantní pole znaků. Pokud ve funkci nepotřebujete měnit

hodnoty pole, je obvykle dobrý nápad použít klíčové slovo `const` před datovým typem obsaženým v poli, aby vás překladač ohlídal, že se pole nesnažíte měnit. Do takovéto funkce pak klidně můžete poslat i pole, které ve skutečnosti měnit lze, jinak řečeno např. `char*` lze bez problému převést na `const char*`. V opačném směru konverze není korektní.

```
int delka_retezce(const char* retezec) {
    int delka = 0;

    // dokud není znak na adrese v ukazateli roven znaku NUL
    while (*retezec != '\0') {
        delka = delka + 1;
        retezec = retezec + 1;  // posuň ukazatel o jeden znak dále
    }
    return delka;
}
```

Tato funkce postupně projde všechny znaky řetězce a počítá, kolik jich je, dokud nenarazí na znak `'\0`. Pro procházení řetězce je zde použita aritmetika s ukazateli.

Z toho vyplývá mimo jiné to, že znak `NUL` nemůže být použit "uprostřed" řetězce. Pokud by tomu tak bylo, tak funkce, které by s takovýmto řetězcem pracovaly, by při nalezení tohoto znaku přestaly řetězec zpracovávat, a jakékoliv další znaky za `NUL` by byly ignorovány. Uhodnete tak, co vypíše následující program?

```
#include <stdio.h>

int main() {
    char text[] = {'U', '\0', 'P', 'R', '\0'};
    printf("%s\n", text);
    return 0;
}
```

Řetězce jako pole

S řetězci pracujeme jako s klasickými poli znaků. Například pro získání prvního znaku řetězce můžeme použít operátor hranatých závorek:

```
char vrat_prvni_znak(const char* retezec) {
    return retezec[0];
}
```

Funkce pro práci s řetězci

Standardní knihovna *C* obsahuje řadu funkcí, které umí s řetězci zakončenými nulou pracovat. Zde je seznam několika vybraných funkcí, které pro vás můžou být užitečné:

- **Zjištění délky řetězce**: funkce `strlen` bere jako parametr řetězec a vrací jeho délku. Jedná se o jednu z nejčastěji používaných funkcí při práci s řetězci a vyplatí se jí tak znát.

  Při jejím použití je ovšem nutné si dát pozor na to, že délka provádění této funkce závisí na tom, jak je řetězec dlouhý. Pokud bude mít řetězec milion znaků, tak bude tato funkce muset projít všech milion znaků, dokud nenarazí na znak `NUL`. Dávejte si tak pozor, abyste tuto funkci nevolali zbytečně často. Například pokud použijete funkci `strlen` v podmínce cyklu `for`:

```
for (int i = 0; i < strlen(retezec); i++) {
    ...
}
```

Tak se délka řetězce vypočte při každé iteraci cyklu. Pokud by tak řetězec měl milion znaků, musel by program provést bilion[4] (!) operací pouze pro zjištění délky řetězce. Lepší volbou (pokud se délka řetězce nemění, což je relativně vzácná operace) je tak předpočítat si jeho délku dopředu a uložit si ji do proměnné:

[4] 1 000 000 000 000

```
int delka = strlen(retezec);
for (int i = 0; i < delka; i++) {
    ...
}
```

- **Porovnání dvou řetězců**: běžnou operací, kterou bychom s řetězci chtěli udělat, je porovnat, zdali jsou dva řetězce stejné, popřípadě který z nich je menší[5]. Funkce `strcmp` bere dva řetězce a vrací nulu, pokud se řetězce rovnají, zápornou hodnotu, pokud je první řetězec menší než ten druhý, a kladnou hodnotu, pokud je druhý řetězec menší než první.

  [5] Pro porovnávání řetězců se používá lexikografické uspořádání. Nalezne se první dvojice znaků (zleva), ve kterém se řetězce liší, a tyto dva znaky se porovnají pomocí jejich číselné (ASCII) hodnoty.

- **Vyhledání řetězce v řetězci**: pokud chcete zjistit, jestli se v nějakém řetězci vyskytuje jiný řetězec, můžete použít funkci `strstr`.

- **Převod textu na číslo**: často můžete potřebovat převést textový zápis čísla na jeho číselnou hodnotu. K tomu můžete použít například funkci `strtol` (*string to long*). První parametr funkce je řetězec, který chcete převést, do druhého parametru můžete předat ukazatel na ukazatel na znak, do kterého se uloží pozice ve vstupním řetězci těsně za načteným číslem. Posledním parametrem je soustava, ve které se má číslo načíst (obvykle to bude desítková soustava, tedy hodnota `10`). Návratovou hodnotou funkce je pak načtené číslo.

  Můžete použít také funkci `atoi`, která je trochu jednodušší na použití, ale při jejím použití nelze zjistit, zdali při konverzi nedošlo k chybě (například pokud vstupní řetězec nereprezentoval číslo).

Cvičení: Pro procvičení práce s řetězci si můžete zkusit některé z těchto funkcí sami naprogramovat. Další úlohy pro práci s řetězci můžete nalézt zde .

Vstup a výstup

Už víme, jak v paměti počítače pracovat s (ASCII) znaky a řetězci. Nyní si ukážeme, jak můžou naše programy komunikovat s okolním světem – se soubory na disku, s terminálem, s ostatními programy běžícími na vašem počítači či s úplně jiným počítačem přes síť. Komunikace programů se obecně označuje jako **I/O** (*input/output*).

Komunikace s terminálem, souborem, tiskárnou či přes síť má samozřejmě rozlišná pravidla. Abychom v každém programu nemuseli programovat podporu pro každý vstupní/výstupní kanál od nuly, z velké části se o toto stará operační systém. Ten nám umožňuje komunikovat s okolním světem pomocí tzv. **souborových deskriptorů** (*file descriptors*). Při vytvoření nového komunikačního kanálu (například při otevření souboru) našemu programu operační systém předá nový souborový deskriptor identifikovaný číslem. Když poté náš program chce vypsat nebo načíst data,

tak musí předat operačnímu systému číslo deskriptoru, se kterým chceme komunikovat. Můžeme například říct `Vypiš text "ahoj" do souborového deskriptoru s číslem 5`. Ať už je na tento deskriptor připojen soubor, terminál či něco jiného, operační systém se postará o to, aby k němu data z našeho programu korektně dorazila.

Standardní souborové deskriptory

Každému programu při spuštění přiřadí operační systém tři základní souborové deskriptory:

- **Standardní vstup** (`stdin`): tento deskriptor má číslo `0` a používá se pro čtení vstupu. Pokud váš program spustíte z terminálu, tak do `stdinu` bude přesměrován text, který napíšete v terminálu. Nemusí tomu tak však být vždy. Váš program můžete například spustit z jiného programu, a předat mu vstup přímo z paměti. Nebo můžete například na vstup vašeho programu přesměrovat soubor z disku:

  ```
  $ ./program < soubor.txt
  ```

- **Standardní výstup** (`stdout`): tento deskriptor má číslo `1` a používá se pro výpis dat. Pokud váš program spustíte z terminálu, tak data odeslaná do `stdoutu` se objeví na obrazovce terminálu. Opět to ale není jediná možnost, `stdout` může být například přesměrovaný do souboru na disku:

  ```
  $ ./program > soubor.txt
  ```

  Pokud použijete například funkci `printf`, tak ta pošle svůj výstup právě do deskriptoru `stdout`.

  Pokud toto nastavení nezměníte, tak `stdout` implicitně používá tzv. **bufferování po řádcích** (*line buffering*). To znamená, že pokud zapíšete do `stdout` pomocí některé z funkcí standardní knihovny *C* nějaký text, tak tento text se nejprve zapíše do dočasného pole (*bufferu*) v paměti. Až jakmile na výstup zapíšete znak odřádkování `'\n'`[1], tak dojde k vyprázdnění (*flush*) bufferu, kdy je jeho obsah odeslán na výstup. Jinak řečeno, dokud nevypíšete znak odřádkování, váš výstup se neobjeví např. v terminálu. Bufferování po řádcích se provádí jako optimalizace, výstup (i vstup) programu totiž může být velmi pomalý.

  [1] Nebo jakmile v bufferu dojde paměť.

- **Standardní chybový výstup** (`stderr`): tento deskriptor má číslo `2` a používá se pro výpis chyb a logovacích záznamů. Narozdíl od `stdout` nepoužívá `stderr` implicitně line buffering, takže cokoliv, co do něj zapíšete, se okamžite odešle na výstup deskriptoru.

Mimo těchto standardních deskriptorů můžete ve svých programech vytvářet i další deskriptory, například pomocí otevírání souborů. Více o tom, jak fungují souborové deskriptory a vstup a výstup programu se dozvíte v předmětu Operační systémy.

Interpretace vstupních a výstupních dat

Je dobré si uvědomit, že stejně jako v operační paměti, i při komunikaci vždy pracujeme pouze s čísly (byty), jejichž význam je dán čistě tím, jak je jejich příjemce bude interpretovat. Pokud náš program do souboru zapíše byty `85`, `80`, `82`, a my tento soubor otevřeme v textovém editoru, který jej bude pokládat za ASCII soubor, zobrazí se nám text `UPR`. Pokud jej však otevřeme v binárním editoru, budou to pro něj pouze tři celá čísla. Pro prohlížeč obrázků by tato čísla zase mohla reprezentovat barevné složky RGB pixelu.

Aby tak komunikace dvou stran dávala smysl, musí se obě strany dohodnout na tom, jak budou interpretovat přenášená data.

Ošetření chyb

Zatím jsme předpokládali, že operace, které provádíme v programu, vždy uspějí. Například při zápisu hodnoty do proměnné jsme předpokládali, že se hodnota v paměti na adrese dané proměnné opravdu objeví a když ji pak zpátky načteme, tak se při přenosu nijak neznehodnotí.

Při načítání vstupu či výpisu dat ovšem může velmi často dojít k různým chybovým situacím. Během zápisu souboru na USB "flashku" ji můžeme omylem vytáhnout, při posílání dat přes síť nám může vypadnout připojení k internetu nebo při načítání čísla z terminálu nám může zákeřný uživatel zadat něco, co číslo ani zdaleka nepřipomíná.

Pokud tedy chceme psát robustní programy, které zvládnou korektně reagovat i na nevalidní vstup a na různé chybové situace, které mohou nastat, musíme do našich programů přidat tzv. **ošetření chyb** (*error handling*). Jedná se o obslužný kód, který reaguje na možné problémové situace a snaží se je vyřešit. Jak ošetřovat chyby při komunikaci si ukážeme v jednotlivých sekcích o vstupu a výstupu .

Vstup

Abychom mohli našim programům dávat příkazy nebo parametrizovat jejich chování, téměř vždy v nich potřebujeme přečíst nějaké informace ze vstupu programu. V této sekci si ukážeme několik užitečných funkcí ze standardní knihovny *C*, které nám to umožňují. Pro použití těchto funkcí musíte ve svém programu vložit soubor `<stdio.h>`.

Načtení jednoho znaku

Pro načtení jednoho znaku ze standardního vstupu (`stdin`) můžeme použít funkci `getchar`. Ta nám vrátí jeden znak ze vstupu, popřípadě hodnotu makra `EOF`[1], pokud již je vstup uzavřený a nelze z něj nic dalšího načíst nebo pokud došlo při načítání k nějaké chybě.

[1] End-of-file

Načtení řádku

Načítat vstup po jednotlivých znacích je poměrně zdlouhavé. Velmi často chceme ze vstupu načíst jeden řádek textu. Toho můžeme dosáhnout například použitím funkce `fgets`. Ta jako parametry přijímá ukazatel na řetězec, do kterého zapíše načítaný řádek, maximální počet znaků, který lze načíst[2]. Třetí parametr je soubor , ze kterého se má vstup načíst. O souborech se dozvíte více později, pokud chcete načítat data ze standardního vstupu, tak použijte jako třetí parametr globální proměnnou `stdin`, která je nadefinována v souboru `<stdio.h>`. Pro jednoduché zjištění délky řetězce, do kterého zapisujete, můžete použít operátor `sizeof`:

[2] Tato velikost je včetně znaku `'\0'`, který je vždy zapsán na konec vstupního řetězce. Pokud tak máte řetězec (pole) o délce `10`, předejte do `fgets` hodnotu `10`. Funkce načte maximálně `9` znaků a na konec řetězce umístí znak

'\0'.

```
#include <stdio.h>

int main() {
    char buf[80];
    // načti řádek textu ze vstupu do řetězce `buf`
    fgets(buf, sizeof(buf), stdin);

    return 0;
}
```

Pokud tato funkce vrátí návratovou hodnotu `NULL`, tak při načítání došlo k chybě. Tuto chybu byste tak ideálně měli nějak ošetřit:

```
#include <stdio.h>

int main() {
    char buf[80];
    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        printf("Nacteni dat nevyslo. Ukoncuji program\n");
        return 1;
    }

    return 0;
}
```

Načtení formátovaného textu

Pokud chceme načítat text, který má očekávaný formát, popřípadě chceme text rovnou zpracovat, například jej převést na číslo, můžeme použít formátované načítání vstupu pomocí funkce `scanf`. Této funkci předáme tzv. **formátovací řetězec** (*format string*), který udává, jak má vypadat vstupní text. V tomto řetězci můžeme používat různé zástupné znaky. Za každý zástupný znak ve formátovacím řetězci `scanf` očekává jeden parametr s adresou, do které se má uložit načtená hodnota popsaná zástupným znakem ze vstupu. Například tento kód načte ze vstupu dvě celá čísla:

```
int x, y;
scanf("%d%d", &x, &y);
```

Pomocí formátovacího řetězce můžeme také vyžadovat, co musí v textu být. Například `scanf("x%d", ...)` načte vstup pouze, pokud v něm nalezne znak `'x'` následovaný číslem.

Seznam všech těchto zástupných znaků naleznete v dokumentaci . Načítat můžeme například celá čísla (`%d`), desetinná čísla (`%f`) či znaky (`%c%`).

> Funkce `scanf` načítá data ze standardního vstupu programu (`stdin`). Obsahuje ovšem několik dalších variant, pomocí kterých může načítat formátovaná data z libovolného souboru (`fscanf`) nebo třeba i z řetězce v paměti (`sscanf`).

Funkce `scanf` je jistě užitečná, zejména u krátkých "toy" programů, nicméně má také určité problémy, které jsou popsány níže. Pokud to je tedy možné, pro načítání vstupu raději používejte funkci `fgets`.

Načítání řetězců pomocí `scanf`

Pomocí `scanf` můžeme načítat také celé řetězce pomocí zástupného znaku `%s`. Zde si ovšem musíme dávat pozor, abychom u něj uvedli i maximální délku řetězce, do kterého chceme text načíst[3]:

[3]Narozdíl od funkce `fgets` se zde musí uvést délka o jedna menší, než je délka cílového řetězce, do kterého znaky zapisujeme.

```
char buf[21];
scanf("%20s", buf);
```

Pokud bychom použili zástupný znak `%s` bez uvedené velikosti cílového řetězce, snadno by se mohlo stát, že nám uživatel zadá moc dat, které by funkce `scanf` začala vesele zapisovat i za paměť předaného řetězce, což může vést buď k pádu programu (v tom lepším případě) nebo ke vzniku bezpečnostní zranitelnosti, pomocí které by uživatel našeho programu mohl například získat přístup k počítači, na kterém program běží (v tom horším případě):

```
char buf[21];
// pokud uživatel zadá více než 20 znaků, může svým vstupem začít přepisovat paměť
// běžícího programu
scanf("%s", buf);
```

Zpracování bílých znaků

Funkce `scanf` ignoruje bílé znaky (mezery, odřádkování, tabulátory atd.) mezi jednotlivými zástupnými znaky ve formátovacím řetězci. Například v následujícím kódu je validním vstupem `x8`, `x 8` i `x 8`:

```
int a;
scanf("x%d", &a);
```

I když může toto chování být užitečné, někdy je také celkem neintuitivní. Problém může způsobovat zejména, pokud se pro načítání vstupu kombinuje formátované načítání (`scanf`) s neformátovaným načítáním (např. `fgets`). Funkce `scanf` totiž bílé znaky nechá ve vstupu ležet, pokud je nepotřebuje zpracovat.

Například, následující program načítá číslo pomocí funkce `scanf` a poté se snaží načíst následující řádek textu pomocí funkce `fgets`:

```
int cislo;
scanf("%d", &cislo);

char radek[80];
fgets(radek, sizeof(radek), stdin);
```

Pokud tomuto programu předáme text `5\nahoj`, očekávali bychom, že se v řetězci `radek` objeví `ahoj`. Nicméně funkce `scanf` načte číslo `5` a nechá ve vstupu ležet znak odřádkování, protože nic dalšího načíst nepotřebuje. Funkce `fgets` poté uvidí znak odřádkování, načte jej a skončí své provádění (načte prázdný řádek), což zřejmě není chování, které bychom od programu čekali.

Ošetření chyb

Funkce `scanf` je problematická i co se týče ošetření chyb. Její návratová hodnota sice udává, kolik zástupných znaků ze vstupu se jí podařilo načíst, problémem však je, že pokud se funkce načte třeba pouze polovinu vstupu, tak již nemůžeme zavolat znovu se stejným formátovacím řetězcem, jinak by se snažila načíst data, která již načetla.

Například pokud bychom tomuto programu:

```
int x, y;
scanf("%d%d", &x, &y);
```

předali text `5 asd`, tak funkce vrátí hodnotu `1`, tj. načetla ze vstupu jedno číslo. Nyní ovšem už funkci nemůžeme zavolat znovu (jakmile bychom např. ve vstupu přeskočili nevalidní text), protože v této chvíli už bychom chtěli načíst pouze jedno číslo.

Parametry příkazového řádku

Další možností, jak předat nějaký vstup vašemu programu, je předat mu parametry při spuštění v terminálu:

```
$ ./program arg1 arg2 arg3
```

K těmto předaným řetězcům poté lze přistoupit ve funkci `main`.

Výstup

Stejně jako pro načítání vstupu, i pro výpis textu na výstup nabízí standardní knihovna *C* sadu užitečných funkcí, opět umístěných v souboru `<stdio.h>`. Stejně jako u načítání vstupu bychom měli řešit ošetření chyb. Nicméně, u zápisu to (alespoň u malých programů) není až tak nutné, protože chyby zápisu jsou vzácnější než chyby při vstupu. Zdrojem dat je totiž náš program, a nemusíme tak kontrolovat, jestli jsou data validní. Tato povinnost v jistém smyslu přechází na druhou stranu, s kterou náš program komunikuje.

Vypsání znaku

Pro vypsání jednoho znaku na standardní výstup (`stdout`) můžeme použít funkci `putchar`.

Vypsání řetězce

Pro vypsání celého řetězce na `stdout` můžete použít funkci `puts`. Pozor na to, že v předaném řetězci musí být obsažen ukončovací `NUL` znak! Funkce `puts` se bude snažit číst a vypisovat znaky ze zadané adresy, až dokud na takovýto znak nenarazí. Pokud by tento znak v předaném řetězci nebyl, tak se může funkce pokoušet číst nevalidní paměť i za pamětí řetězci, dokud na `NUL` nenarazí.

Vypsání formátovaného textu

K výpisu formátovaného textu na `stdout` můžeme použít funkci `printf`, s kterou jsme se již mnohokrát setkali. Prvním parametrem funkce je formátovací řetězec, do kterého můžete dávat zástupné znaky. Pro každý zástupný znak funkce očekává jednu hodnotu (argument) za formátovacím řetězcem, které bude zformátován výstup. Například takto můžeme vytisknout číslo a po něm řetězec:

```
const char* text = "Cislo";
int cislo = 5;
```

```
printf("%s: %d\n", text, cislo);
```

Zástupné znaky funkcí `printf` i `scanf` jsou obdobné, jejich seznam a různé možnosti nastavení můžete najít v dokumentaci.

Stejně jako `scanf` má i funkce `printf` různé varianty pro formátovaný výpis do souborů (`fprintf`) či do řetězce v paměti (`sprintf`).

Vlastní datové typy

Nyní už umíme pracovat se základními datovými typy v *C* (celá čísla, desetinná čísla, pravdivostní hodnoty, znaky) a také umíme pracovat s jejich adresami a vytvářet jich více najednou. Doposud jsme však vždy pracovali s každým datovým typem zvlášť.

Představte si, že byste chtěli naprogramovat hru, ve které budete mít nějaké počítačem ovládané příšery[1]. Každá příšera může mít spoustu vlastností – jméno, počet životů, zranění, které uděluje, umístění na mapě, kořist atd. Zároveň bude takových příšer v naší hře určitě více. Mohli bychom tak příšery reprezentovat pomocí pole pro každou jeho vlastnost:

[1] *Non-player character* (NPC)

```
const char* prisera_jmeno[100];
int prisera_zivot[100];
int prisera_zraneni[100];
float prisera_poloha_x[100];
float prisera_poloha_y[100];
...
```

I když by jistě šlo programy tvořit tímto způsobem, asi sami uznáte, že to není ideální, protože to má spoustu nevýhod:

- Pokud bychom například změnili (maximální) počet příšer, museli bychom synchronizovat tuto velikost mezi všemi poli, které reprezentují jednotlivé vlastnosti příšer.
- K názvům proměnných musíme přidávat nějakou předponu (např. `prisera`), abychom dali najevo, že tyto proměnné vlastně patří k jednomu logickému prvku (příšeře).
- Pokud bychom chtěli jednu takovou příšeru poslat do funkce, tak by to vyžadovalo spoustu parametrů:

  ```
  int vypocti_pocet_zkusenosti(
      const char* prisera_jmeno,
      int prisera_zivot,
      int prisera_zraneni,
      float prisera_poloha_x,
      float prisera_poloha_y,
      ...
  ) { }
  ```

  Celou příšeru bychom ani nemohli z funkce přímočaře vrátit, protože funkce můžou vracet pouze jednu hodnotu.
- Pokud bychom chtěli příšeře přidat novou vlastnost, museli bychom přidat novou proměnnou nebo pole na všechna místa, kde s příšerami pracujeme. Například by se musely změnit parametry každé funkce, která by přijímala příšeru.

Co bychom ve skutečnosti chtěli překladači říct, je něco ve smyslu `Příšera je něco, co má jméno, počet životů, zranění, pozici a kořist`, a poté bychom chtěli ve funkci například říct `Vytvoř pole 100 příšer`:

```
Prisera prisery[100];
```

Takto bychom zlepšili úroveň abstrakce našeho kódu – v tomto konkrétním případě bychom se mohli začít v kódu bavit o `příšeře` místo o `jménu, počtu životů, zranění, ...`, které spolu nějak souvisí.

Jinak řečeno, chtěli bychom si vytvořit náš vlastní datový typ. A právě to můžeme v *C* udělat pomocí struktur .

> Struktury jsou posledním syntaktickým prvkem *C*, o kterém se budeme v předmětu UPR bavit. Jazyk *C* sice obsahuje i několik dalších syntaktických prvků, které jsme si neukázali, ty však nejsou nutné pro tvorbu jednoduchých programů. Dále se už pouze budeme bavit o konkrétních aplikacích toho, co jsme se naučili, pro tvorbu různých typů programů.

Struktury

Struktury (*structures*) nám umožňují popsat nový datový typ, který se bude skládat z jednoho či více tzv. **členů** (*members*)[1]. Každému členu musíme určit jeho jméno a datový typ. Novou strukturu můžeme popsat pomocí tzv. *deklarace struktury*:

[1]Můžete se setkat také s názvy **atribut** (*attribute*), **vlastnost** (*property*) nebo *field*. V kontextu struktur *C* označují všechny tyto názvy jedno a to samé - člena struktury.

```
struct <název struktury> {
    <datový typ prvního členu> <název prvního členu>;
    <datový typ druhého členu> <název druhého členu>;
    <datový typ třetího členu> <název třetího členu>;
    ...
};
```

> Při definici struktury nezapomínejte na finální středník za složenými závorkami, je povinný.

Například, pokud bychom chtěli vytvořit datový typ reprezentující `příšeru`, která má své jméno a počet životů, můžeme deklarovat následující strukturu:

```
struct Prisera {
    const char* jmeno;
    int pocet_zivotu;
};
```

Tento kód sám o sobě **nic neprovádí**! Pouze pomocí něho říkáme překladači, že vytváříme nový datový typ s názvem `struct Prisera`. Poté nám překladač umožní dále v programu vytvořit například lokální proměnnou tohoto datového typu:

```
// lokální proměnná s názvem `karel` a datovým typem `struct Prisera`
struct Prisera karel;
```

Pro pojmenovávání struktur používejte v rámci předmětu UPR jmennou konvenci `PascalCase`.

> Struktury jsou plnohodnotnými datovými typy. Můžete tak vytvářet ukazatele na struktury, pole struktur,

můžete použít struktury jako členy jiné struktury atd.

Reprezentace struktury v paměti

Pokud vytvoříme proměnnou datového typu struktury, tak překladač naalokuje paměť pro všechny členy této struktury. V případě výše by proměnná `karel` obsahovala nejprve byty pro ukazatel `const char*` a poté byty pro `int`. Členové struktury budou v paměti uloženy ve stejném pořadí, v jakém byly popsány při deklaraci struktury. Neznamená to ovšem, že musí ležet hned za sebou! Překladač se může rozhodnout mezi členy struktury v paměti vložit mezery (tzv. *padding*) kvůli urychlení provádění programu[2].

[2]Pro některé typy procesorů může být rychlejší přistupovat k adresám v paměti, které jsou například násobkem `4` nebo `8`. Proto překladač mezery do struktur vkládá, aby jednotlivé členy zarovnal (*align*) v paměti. Více se můžete dozvědět například zde .

Z toho vyplývá, že velikost struktury není vždy zcela intuitivní. Například následující struktura s názvem `StiskKlavesy` obsahuje jeden znak (`char`) s velikostí 1 byte a jedno číslo (`int`) s velikostí 4 byty. Kvůli "neviditelným" mezerám vloženým překladačem ovšem velikost struktury nemusí být `5` bytů!

```
#include <stdio.h>

struct StiskKlavesy {
    char klavesa;
    int delka;
};

int main() {
    printf("Velikost znaku: %lu\n", sizeof(char));
    printf("Velikost cisla: %lu\n", sizeof(int));
    printf("Velikost struktury StiskKlavesy: %lu\n", sizeof(struct StiskKlavesy));

    return 0;
}
```

Proto pro zjištění velikosti struktury (například při dynamické alokaci paměti) vždy používejte operátor `sizeof`.

Umístění a platnost struktur

Stejně jako u proměnných platí, že strukturu lze používat pouze v oblasti, ve které je platná (v jejím tzv. *scopu*). Narozdíl od funkcí lze struktury deklarovat i uvnitř funkcí, nicméně nejčastěji se struktury deklarují na nejvyšší úrovni souboru (tzv. *global scope*).

Inicializace struktury

Stejně jako u základních datových typů a polí platí, že pokud lokální proměnné s datovým typem nějaké struktury nedáte počáteční hodnotu, tak bude její hodnota nedefinovaná . Strukturu můžete nainicializovat pomocí složených závorek se seznamem hodnot pro jednotlivé členy struktury:

```
struct Prisera karel = { "Karel", 100 };
```

Stejně jako u polí platí, že hodnoty, které nezadáte, se nainicializují na nulu:

```
struct Prisera karel = {}; // `jmeno` i `pocet_zivotu` bude `0`
struct Prisera karel = { "Karel" }; // `jmeno` bude "Karel", `pocet_zivotu` bude `0`
```

Abyste si nemuseli pamatovat pořadí členů struktury při její inicializaci, můžete jednotlivé členy nainicializovat explicitně pomocí tečky a názvu daného členu:

```
struct Prisera karel = { .pocet_zivotu = 100, .jmeno = "Karel" };
```

Jednotlivé hodnoty členům se přiřazují zleva doprava, takže pokud použijete název nějakého členu více než jednou, "zvítězí" poslední zadaná hodnota. Tomuto se však vyhněte, a ani nekombinujte inicializaci pomocí pořadí a pomocí názvů členů. Takovýto kód by totiž byl značně nepřehledný.

Přístup ke členům struktur

Abychom mohli číst a zapisovat jednotlivé členy struktur, můžeme použít operátor **přístupu ke členu** (*member access operator*), který má syntaxi `<struktura>.<název členu>`:

```
#include <stdio.h>

struct Osoba {
    int vek;
    int pocet_pratel;
};

int main() {
    struct Osoba martina = { .vek = 18, .pocet_pratel = 10 };
    martina.vek += 1;
    martina.pocet_pratel += 20;
    printf("Martina ma %d let a ma %d pratel\n", martina.vek, martina.pocet_pratel);

    return 0;
}
```

Pokud máme k dispozici pouze ukazatel na strukturu, tak je přístup k jejím členům trochu nepraktický kvůli prioritě operátorů. Operátor dereference (`*`) má totiž menší prioritu než operátor přístupu ke členu (`.`). Abychom tak nejprve z ukazatele na strukturu načetli její hodnotu a až poté přistoupili k jejímu členu, museli bychom použít závorky:

```
void pridej_pratele(struct Osoba* osoba) {
    (*osoba).pocet_pratel++;
}
```

Jelikož ukazatele na struktury jsou využívány velmi často, *C* nabízí pro tuto situaci zkratku v podobě operátoru **přístupu k členu přes ukazatel** (*member access through pointer*), který má syntaxi `<ukazatel na strukturu>-><název členu>`:

```
void pridej_pratele(struct Osoba* osoba) {
   osoba->pocet_pratel++;
}
```

Operátor `->` je čistě syntaktickou zkratkou, tj. `*(ukazatel).clen == ukazatel->clen`.

Vytváření nových jmen pro datové typy

Možná vás napadlo, že psát při každém použití struktury klíčové slovo `struct` před jejím názvem je zdlouhavé. *C* umožňuje dávat datovým typům nové názvy, aby se nám s nimi lépe pracovalo. Lze toho dosáhnout pomocí syntaxe `typedef <datový typ> <jméno>;`:

```
typedef int teplota;

int main() {
    teplota venkovni = 24;
    return 0;
}
```

Pomocí `typedef` dáme danému datovému typu nové jméno, pomocí kterého pak tento typ můžeme používat (původní název datového typu toto nijak neovlivní). Opět platí, že takto vytvořené jméno lze použít pouze v oblasti (*scopu*), kde byl `typedef` použit. Obvykle se používá na nejvyšší úrovni souboru.

U struktur si pomocí `typedef` můžeme zkrátit jejich název z `struct <nazev>` na `<nazev>`:

```
struct Osoba {
    int vek;
};

typedef struct Osoba Osoba;

int main() {
    Osoba jiri;
    return 0;
}
```

Toto lze ještě více zkrátit, pokud deklaraci struktury použijeme přímo na místě datového typu v `typedef`:

```
typedef struct Osoba {
    int vek;
} Osoba;
```

A konečně, abychom nemuseli jméno struktury opakovat dvakrát, můžeme vytvořit tzv. **anonymní strukturu** (*anonymous structure*) bez názvu, a jméno jí přiřadit až pomocí `typedef`.

```
typedef struct {
    int vek;
} Osoba;
```

Právě takto se obvykle deklarují struktury v *C*.

Použití struktur ve strukturách

Jelikož deklarace struktury vytvoří nový datový typ, nic vám nebrání v tom používat struktury jako členy jiných struktur[3]:

[3] Lze si můžete všimnout, že vnořené struktury lze inicializovat stejně jako proměnné struktur, tj. pomocí složených závorek `{}`.

```
typedef struct {
```

```
    float x;
    float y;
} Poloha;

typedef struct {
    const char* jmeno;
    int cena;
} Vec;

typedef struct {
    int pocet_zivotu;
    Poloha poloha;
    Vec korist[10];
} Prisera;

int main() {
    Prisera prisera = { .pocet_zivotu = 100, .poloha = { .x = 0, .y = 0 } };

    return 0;
}
```

Díky tomu můžeme vytvářet celé hierarchie datových typů, což může značně zpřehlednit náš program, protože díky tomu náš kód může pracovat na vyšší úrovni abstrakce.

Rekurzivní struktury

Pokud bychom ovšem chtěli použít jako člena struktury tu stejnou strukturu (například struktura `Osoba` může mít člen `matka` opět s datovým typem `Osoba`), nemůžeme takovýto člen uložit ve struktuře přímo, můžeme tam uložit pouze jeho adresu[4]:

[4]Zde si můžete všimnout, že musíme použít `struct Osoba` pro datový typ členu `matka`. Je to z toho důvodu, že v momentě, kdy tento člen definujeme, ještě neproběhl `typedef`, takže datový typ `Osoba` zatím neexistuje. K vytvoření tohoto datového typu dojde až jakmile je struktura zcela nadefinována.

```
typedef struct Osoba {
    int vek;
    struct Osoba* matka;
} Osoba;
```

Je to proto, že pokud by `Osoba` byla definována pomocí `Osoby`, tak by došlo k rekurzivní definici, kterou nelze vyřešit. Nešlo by totiž určit velikost `Osoby` - její velikost by závisela na velikosti jejího členu `matka`, jehož velikost by závisela na velikosti jeho členu `matka` atd. Proto tedy musíme v tomto případě použít ukazatel, který má fixní velikost, ať už ukazuje na jakýkoliv typ.

Struktury a funkce

Pomocí struktur si můžeme vytvořit nový datový typ, což pomáhá přehlednosti programů, protože se díky tomu můžeme v programech vyjadřovat pomocí pojmů z oblasti (tzv. domény), kterou se náš program zabývá (`Student`, `Příšera`, `Munice`, `Letadlo`, `Volant` atd.) a ne pouze pomocí pojmů, kterým rozumí počítač (číslo, znak, pravdivostní hodnota).

Abychom pracovali s ještě vyšší úrovní abstrakce, bylo by užitečné, pokud bychom mohli k vlastním datovým typům nadefinovat také vlastní operace, které by s nimi uměly pracovat. Některé programovací jazyky umožňují provádět tzv. **přetěžování operátorů** (*operator overloading*), pomocí kterého můžeme například umožnit používání operátorů jako je `+` s vlastními datovými typy. *C* toto sice neumožňuje, nicméně chování můžeme k námi vytvořeným strukturám přidat pomocí funkcí.

Často tak k naší struktuře chceme vytvořit sadu funkcí, které s ní budou pracovat. V *C* pro tento koncept neexistuje žádná syntaktická podpora. Obvykle se tak prostě takovéto funkce pojmenují tak, aby začínaly názvem struktury, ke které jsou přidružené, a přebírají ukazatel na tuto strukturu jako svůj první parametr[1]:

[1]Ukazatel se používá, abychom nemuseli struktury při předávání do funkcí kopírovat (mohou být relativně velké) a abychom je mohli případně zevnitř funkcí modifikovat.

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

typedef struct {
    float x;
    float y;
} Poloha;

typedef struct {
    const char* jmeno;
    int skore;
    Poloha poloha;
} Hrac;

void hrac_posun(Hrac* hrac, int x, int y) {
    hrac->poloha.x += x;
    hrac->poloha.y += y;
}
void hrac_pridej_skore(Hrac* hrac, int skore) {
    hrac->skore += skore;
    if (hrac->skore > 100) {
        hrac->skore = 100;
    }
}
bool hrac_vyhral(Hrac* hrac) {
    return hrac->skore == 100;
}

int main() {
    Hrac hrac = { .jmeno = "Jindrich", .skore = 40, .poloha = { .x = 10, .y = 20 } };
    hrac_posun(&hrac, 5, -8);
    hrac_pridej_skore(&hrac, 70);

    printf("Hrac vyhral: %d\n", hrac_vyhral(&hrac));

    return 0;
}
```

Pokud vytvoříme vhodné datové typy (struktury) a budeme s nimi pracovat pomocí funkcí, tak by se naše programy měly přibližovat k tomu, aby je šlo číst jako plynulý a přehledný text.

Vytváření vlastních datových typů, které mají přidružené chování, je základem tzv. Objektově orientovaného programování.

Struktury jako návratový typ funkce

Jelikož struktury mohou obsahovat více datových typů, můžete pomocí nich také obejít fakt, že funkce mohou vracet pouze jednu hodnotu:

```
typedef struct {
    float x;
    float y;
} Poloha;

Poloha vrat_pocatecni_polohu() { ... }
```

Soubory

V sekci o vstupu a výstupu jsme si ukázali, jak pracovat se souborovými deskriptory `stdin` a `stdout` pro základní komunikaci s okolním světem (obvykle s terminálem). Nyní si ukážeme, jak si vytvořit vlastní souborové deskriptory pomocí otevírání souborů na disku. Použijeme k tomu funkce ze standardní knihovny *C*, které se opět nachází v souboru `<stdio.h>`.

Stejně jako u obecného vstupu a výstupu platí, že soubor na disku je pouze seznamem čísel (bytů). Jejich význam je daný čistě tím, kdo a jak je bude interpretovat. Stejný soubor může být například:

- Textovým editorem pokládán za textový dokument
- Prohlížečem obrázků pokládán za obrázek
- Hudebním přehrávačem pokládan za zvukovou nahrávku

Obvykle souborům dáváme přípony (`.txt`, `.jpg`, `.mp3` atd.), abychom dali najevo, jak by se daný soubor měl interpretovat. Samotná přípona však sama o sobě nic neznamená. Změnou přípony z `.txt` na `.jpg` sice můžeme změnit způsob interpretace souboru, samotná data v něm však zůstanou stále stejná – pokud v souboru předtím nebyla data ve formátu JPEG, změna přípony tento stav nijak nezmění.

Nejprve si ukážeme, jak můžeme otevírat soubory na disku, a poté jak do otevřených souborů zapisovat nebo číst data.

Otevírání souborů

Abychom mohli s nějakým souborem začít pracovat, musíme ho nejprve v našem programu otevřít, aby byl vytvořen souborový deskriptor, do kterého pak můžeme zapisovat či z něho číst data. K tomu slouží funkce `fopen`, která má následující signaturu:

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

Cesta k souboru

Jako svůj první parametr funkce `fopen` očekává řetězec s cestou k souboru, který má být otevřen. Cestu můžete zadat dvěma způsoby:

Absolutní cesta (*absolute path*) je cesta, která začíná kořenovým adresářem souborového systému, například `/home/student/upr/soubor.txt`[1]. Aby byla cesta absolutní, musí na Linuxu začínat dopředným lomítkem.

[1]Na Windows by podobná cesta mohla vypadat například jako `C:\Users\student\upr\soubor.txt`.

- **Relativní cesta** (*relative path*) se vyhodnotí relativně k tzv. **pracovnímu adresáři** (*working directory*) běžícího programu. Pokud spustíte váš program z terminálu, tak se pracovní adresář implicitně nastaví na adresář, ze kterého jste program spustili. Pokud tedy například spustíte váš program ze adresáře `/home/student/upr` a funkci `fopen` předáte cestu `soubor.txt`, tak se funkce pokusí otevřít soubor na cestě `/home/student/upr/soubor.txt`.

Při zadávání cesty můžete využít zkratky `.` a `..`, které jsou užitečné zejména u relativních cest:

- Zkratka `.` se odkazuje na současný adresář, `./soubor.txt` je tedy to samé jako `soubor.txt`.
- Zkratka `..` se odkazuje na rodičovský adresář, `../data/abc.txt` tedy říká: `Podívej se do rodičovského adresáře, tam nalezni adresář data a v něm soubor abc.txt`.

Nepokoušejte se však zadávat cesty k neexistujícím adresářům. `fopen` sice umí vytvořit nový soubor (pokud použijete odpovídající mód), neexistující adresář za vás nicméně nevytvoří.

Doposud jsme používali prvky *C*, které byly vesměs nezávislé na použitém operačním systému. Jakmile ale naše programy začnou interagovat se **souborovým systémem** (*file system*), budeme muset začít respektovat zákonitosti operačního systému, na kterém náš program poběží. Proto například u cesty k souborům vždy používejte dopředná lomítka (`/`) pro oddělování adresářů, pokud program budete spouštět na Linuxu.

Mód otevření

Druhým parametrem funkce `fopen` je řetězec, jehož obsah určuje, v jakém **módu** (*mode*) se má soubor otevřít. Kompletní seznam všech kombinací módů naleznete v dokumentaci, zde je seznam běžných variant:

Mód	Možné operace	Co se stane, když už soubor existuje?	Co se stane, když soubor neexistuje?
`"r"`	Čtení		chyba
`"w"`	Zápis	obsah souboru je smazán	soubor je vytvořen
`"a"`	Zápis na konci		soubor je vytvořen
`"r+"`	Čtení, zápis		chyba
`"w+"`	Čtení, zápis	obsah souboru je smazán	soubor je vytvořen
`"a+"`	Čtení, zápis na konci		soubor je vytvořen

Při otevírání souboru si musíte rozmyslet, jestli z něj chcete číst, zapisovat do něj nebo provádět obojí. Zároveň si musíte určit, jestli chcete soubor vytvořit v případě, že neexistuje, popřípadě jestli má být jeho obsah smazán, pokud už existuje. Podle těchto vlastností si pak zvolte odpovídající mód otevření souboru.

Textový vs binární režim

Pokud použijete jeden ze základních módů, soubor se otevře v tzv. **textovém režimu**. V tomto režimu dochází ke

konverzi určitých bytů při čtení a zápisu ze souboru. Asi nejdůležitějším znakem, který je takto konvertován, je `'\n'`, neboli **odřádkování** (*newline*). Různé operační systémy totiž při interpretaci souborů používají různé znaky pro odlišení situace, kdy má dojít k přesunu kurzoru na nový řádek:

- `LF`: Linux a macOS [2] používají pro konec řádku ASCII přímo znak `LF (line feed)`, který lze v *C* zapsat jako `'\n'`.

  [2] V dávných dobách používal Mac OS pro odřádkování pouze znak `CR`.
- `CRLF`: Windows používá pro konec řádku dvojici ASCII znaků `CR (carriage return)` a `LF` (v tomto pořadí). `CR` lze v *C* zapsat jako `'\r'`.

Na Windows tak při zápisu do souborů otevřených v textovém módu dojde ke konverzi znaku odřádkování `\n` na dvojici znaků `\r\n`. Stejně tak při načítání dat ze souboru se dvojice znaků `\r\n` převedou na `\n`. Na Linuxu textový mód v podstatě nic nedělá, protože se zde pro odřádkování používá přímo znak `\n`.

Pokud byste však chtěli mít jistotu, že opravdu k žádné konverzi nedojde, a budete zapisovat data, která nemají být interpretována jako text, můžete na konec módu přidat znak `b`. Poté se soubor otevře v tzv. **binární režimu**, kde k žádné konverzi nedojde. Mód `"rb"` tak například říká `Otevři soubor pro čtení v binárním režimu`.

Ošetření chyb

Jakmile řeknete funkci `fopen` jaký soubor (a v jakém módu) má otevřít, funkce jej otevře a vrátí vám ukazatel na strukturu `FILE`, pomocí které můžete se souborem dále pracovat[3]. Stejně jako u jakékoliv práce se vstupem a výstupem i při práci se soubory však může často docházet k různým chybám.

[3] `FILE` je tzv. **neprůhledná** (*opaque*) struktura deklarovaná ve standardní knihovně *C*. Nebudete přistupovat k žádným jejím členům, pouze ukazatel na ni budete posílat do různých funkcí pro práci se soubory, abyste určili, s jakým (otevřeným) souborem chcete pracovat.

Pokud byste se například pokoušeli otevřít neexistující soubor v módu pro čtení `"r"`, dojde k chybě. V takovém případě vám funkce `fopen` vrátí adresu nula (tzv. `NULL` ukazatel). Po každém pokusu o otevření souboru byste tak měli ověřit, zdali se otevření opravdu podařilo nebo ne. Pokud při otevření došlo k chybě, tak se do globální proměnné `errno` uloží číslo, které identifikuje, o jaký typ chyby šlo[4]. K proměnné budete mít přístup, pokud do svého programu vložíte soubor `<errno.h>`. Pomocí funkce `strerror` ze souboru `<string.h>` pak můžete získat řetězec, který danou chybu popisuje:

[4] Seznam různých chybových hodnot, které se můžou v `errno` objevit, můžete naleznout například zde .

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

int main() {
    FILE* soubor = fopen("soubor.txt", "r");
    if (soubor == NULL) {
        printf("Doslo k chybe pri otevirani souboru: %s\n", strerror(errno));
        return 1; // došlo k chybě, vrátíme 1 jako chybový stav programu
    }
```

```
    return 0;
}
```

Pokud píšete malý program a nechce se vám ručně každou chybu ošetřovat, můžete využít funkci `assert` ze souboru `<assert.h>`. Tato funkce očekává pravdivostní hodnotu a kontroluje, zdali platí (`assert` znamená `ujisti se, že platí ...`). Pokud hodnota neplatí, tj. vyhodnotí se na `0` či `false`, tak dojde k okamžitému ukončení vašeho programu. Vypsanou chybovou hlášku tak nebudete moct ovlivnit, ale ošetření chyby se značně zjednodušší:

```
FILE* soubor = fopen("soubor.txt", "r");
assert(soubor); // pokud je `soubor` roven `NULL`, program se zde ukončí
```

Ošetření chyb je dobré nepodceňovat. Pokud chybu ošetříte okamžitě po jejím možném vzniku (i kdyby to mělo být okamžitým vypnutím programu), tak bude mnohem jednodušší zjistit, kde v kódu a proč vznikla. Jinak se může jednoduše stát, že k chybě sice dojde, ale program bude pokračovat vesele dál. Tato chyba pak může v průběhu programu způsobit kaskádu dalších chyb, které nakonec dříve či později povedou k "spadnutí" nebo špatnému fungování programu. V takové situaci bude mnohem náročnější zjistit, kde vznikla původní chyba, která vše způsobila, protože program může spadnout na úplně jiném místě v kódu.

Zavření souboru

Jakmile se souborem přestanete pracovat, je **nutné** ho zavřít. K tomu slouží funkce `fclose`:

```
FILE* soubor = fopen("soubor.txt", "w");
// zápis/čtení ze souboru...
fclose(soubor);
```

Funkce `fclose` vrací číselnou hodnotu, která oznamuje, zdali funkce proběhla v pořádku nebo ne. Pokud funkce vrátí `0`, tak se soubor úspěšně uzavřel. I u zavírání souborů bychom tedy měli mít alespoň základní ošetření chyb[5]:

[5] Operátor `!` provede logickou negaci. Pokud jej použijeme s hodnotou `0`, vrátí hodnotu `1`. Pokud jej použijeme s jakoukoliv jinou hodnotou, vrátí hodnotu `0`. Pokud tedy funkce `fclose` vrátí cokoliv jiného než `0`, `assert` ukončí program.

```
assert(!fclose(soubor));
```

Pokud bychom soubor nezavřeli, tak se například může stát, že kvůli použitému *bufferování* by se data, která jsme do souboru zapsali, nemusela objevit na souborovém systému.

Práce se soubory

Jakmile jsme se pokusili o otevření souboru, ujistili jsme se, že se to opravdu povedlo a získali jsme ukazatel `FILE*`, můžeme začít do programu zapisovat nebo z něj číst data (podle toho, v jakém módu jsme ho otevřeli).

Pozice v souboru

Struktura `FILE` má vnitřně uloženou **pozici** v souboru, na které probíhají veškeré operace čtení a zápisu. Pro zjednodušení práce se soubory se pozice automaticky posouvá dopředu o odpovídající počet bytů po každém čtení či zápisu. Jakmile tedy přečtete ze souboru `n` bytů, tak se pozice posune o `n` pozic dopředu. Pokud byste tedy dvakrát po

sobě přečetli jeden byte ze souboru obsahující text `ABC`, nejprve získáte znak `A`, a podruhé už znak `B`, protože po prvním čtení se pozice posunula dopředu o jeden byte.

> Tím, že je pozice sdílená pro čtení a zápis, tak se raději vyvarujte současnému čtení i zápisu nad stejným otevřeným souborem. V opačném případě budete muset být opatrní, abyste si omylem nepřepsali data nebo nečetli data ze špatné pozice.

Současnou pozici v souboru můžete zjistit pomocí funkce `ftell`. Pokud byste chtěli pozici ručně změnit, můžete použít funkci `fseek`, pomocí které se také například můžete v souboru přesunout na začátek (např. abyste ho přečetli podruhé) nebo na konec (např. abyste zjistili, kolik soubor celkově obsahuje bytů)[1].

[1]Toho můžete dosáhnout tak, že pomocí `fseek(file, 0, SEEK_END)` přesunete pozici na konec souboru, a dále pomocí `ftell(file)` zjistíte, na jaké pozici jste. To vám řekne, kolik má soubor celkově bytů.

> Při použití módu `"a"` budou veškeré zápisy probíhat vždy na konci souboru. Tento mód se hodí například při zápisu do tzv. **logovacích souborů**, které chronologicky zaznamenávají události v programu (události tak vždy pouze přibývají). Zároveň se však po každém zápisu v tomto módu pozice posune na jeho konec. Raději tak nepoužívejte mód `"a+"`, který umožňuje zápis na konec i čtení. Práce s pozicí při současném zapisování i čtení je v takovémto módu totiž poněkud náročná.

Všimněte si, že při práci se `stdout` a `stdin` jsme s pozicí manipulovat nemohli. Je to proto, že tyto dva deskriptory jsou z jistého pohledu "obecnější" než soubory. Můžou být přesměrované na terminál, do souboru, ale klidně také i do jiného počítače přes síť. Tím, že nevíme, "co jsou zač", tak si s nimi nemůžeme dovolit provádět některé operace, jako je právě manipulace s pozicí. Pokud například odešleme data přes síť, už je nemůžeme "vrátit zpátky" změnou pozice. U souborů však víme, že opravdu pracujeme se souborem, takže pozici pro zápis a čtení měnit můžeme.

Zápis do souboru

Pokud chceme do otevřeného souboru zapsat nějaké byty, můžeme použít funkci `fwrite`:

```
size_t fwrite(
    const void* buffer, // adresa, ze které načteme data do souboru
    size_t size,        // velikost prvku, který zapisujeme
    size_t count,       // počet prvků, které zapisujeme
    FILE* stream        // soubor, do kterého zapisujeme
);
```

Funkce `fwrite` předpokládá, že budeme do souboru zapisovat více hodnot stejného datového typu. Parametr `size` udává velikost tohoto datového typu a parametr `count` počet hodnot, které chceme zapsat. Pokud tuto funkci zavoláme, tak dojde k zápisu `size * count` bytů z adresy `buffer` do souboru `stream`. Návratová hodnota `fwrite` značí, kolik prvků bylo do souboru úspěšně zapsáno. Pokud je tato hodnota menší než `count`, tak došlo k nějaké chybě. Například zápis pěti celých čísel do souboru by mohl vypadat následovně:

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

int main() {
    int pole[5] = { 1, 2, 3, 4, 5 };
```

```
    // otevření souboru
    FILE* soubor = fopen("soubor", "wb");
    assert(soubor);

    // zápis do souboru
    int zapsano = fwrite(pole, sizeof(int), 5, soubor);
    assert(zapsano == 5);

    // zavření souboru
    fclose(soubor);

    return 0;
}
```

Při takovémto použití `fwrite` může dojít k zapsání například pouze `3` čísel, pokud během zápisu dojde k chybě[2]. Pokud bychom chtěli zapsat buď vše nebo nic, můžeme říct, že zapisujeme pouze jeden prvek a parameter `count` nastavit na celkovou velikost všech dat, které chceme zapsat:

[2] V takovémto případě by funkce `fwrite` vrátila hodnotu `3`.

```
int pole[5] = { 1, 2, 3, 4, 5 };
fwrite(pole, sizeof(pole), 1, soubor);
```

Pokud bychom zapsali `pole` do souboru takto, uloží se do něj celkem `20` (`5 * 4`) bytů (čísel), které později můžeme v programu zase načíst zpátky. Pokud bychom se podívali, co v souboru je, nalezli bychom seznam čísel `1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0 5 0 0 0`, což odpovídá paměťové reprezentaci pole pěti `int`ů, které bylo vytvořeno výše.

Textový zápis

Pokud bychom si tato data chtěli přečíst jako text, můžeme čísla z výše zmíněného pole zapsat do souboru pomocí nějakého textového kódování, například ASCII. K tomu můžeme využít funkci `fprintf`, která funguje stejně jako `printf`, s tím rozdílem, že text nevypisuje na `stdout`, ale do předaného souboru:

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

int main() {
    int pole[5] = { 1, 2, 3, 4, 5 };

    // otevření souboru
    FILE* soubor = fopen("soubor.txt", "w");
    assert(soubor);

    // zápis do souboru
    for (int i = 0; i < 5; i++) {
        fprintf(soubor, "%d ", pole[i]);
    }

    // zavření souboru
    fclose(soubor);

    return 0;
}
```

V tomto případě by se do souboru zapsalo deset bytů (čísel) `49 32 50 32 51 32 52 32 53 32`, protože číslice jsou v

ASCII reprezentovány čísly `48` až `57` a mezera je reprezentována číslem `32`. Pokud bychom tento soubor otevřeli v textovém editoru, tak by se nám zobrazil text `1 2 3 4 5` .

Bufferování

Stejně jako při zápisu do `stdout` se i při zápisu do souborů uplatňuje *bufferování*. Data, která do souboru zapíšeme, se tak v něm neobjeví hned. Pokud bychom chtěli donutit náš program, aby data uložená v bufferu opravdu vypsal do souboru, můžeme použít funkci `fflush` [3].

[3] Ani zavolání funkce `fflush` však nezajistí, že se data opravdu zapíšou na fyzické médium (například harddisk). To je ve skutečnosti velmi obtížný problém .

Čtení ze souboru

Pro čtení ze souboru můžeme použít funkci `fread`, která je protikladem funkce `fwrite`:

```
size_t fread(
    void* buffer,   // adresa, na kterou zapíšeme data ze souboru
    size_t size,    // velikost prvku, který načítáme
    size_t count,   // počet prvků, které načítáme
    FILE* stream    // soubor, ze kterého čteme
);
```

Tato funkce opět předpokládá, že budeme ze souboru načítat několik hodnot stejného datového typu. Například načtení pěti celých čísel, které jsme zapsali v kódu výše , by mohlo vypadat následovně:

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

int main() {
    int pole[5] = { 1, 2, 3, 4, 5 };

    // otevření souboru
    FILE* soubor = fopen("soubor", "rb");
    assert(soubor);

    // čtení ze souboru
    int precteno = fread(pole, sizeof(int), 5, soubor);
    assert(precteno == 5);

    // zavření souboru
    fclose(soubor);

    return 0;
}
```

Funkce vrátí počet prvků, které úspěšně načetla ze souboru.

Textové čtení

Pokud bychom chtěli načítat ze souboru ASCII text, opět můžeme použít funkce pro načítání textu, například `fgets`[4]

nebo `fscanf`.

[4]S funkcí `fgets` jsme se setkali již dříve, kdy jsme jí jako poslední parametr globální proměnnou `stdin`. Datový typ proměnné `stdin` je právě `FILE*` – při spuštění programu standardní knihovna *C* vytvoří proměnné `stdin`, `stdout` a `stderr` a uloží do nich standardní vstup, výstup a chybový výstup.

U načítání dat si vždy dejte pozor na to, abyste na adrese, kterou předáváte do `fread` nebo `fgets`, měli dostatek naalokované validní paměti. Jinak by se mohlo stát, že data ze souboru přepíšou adresy v paměti, kde leží nějaké nesouvisející hodnoty, což by vedlo k paměťové chybě .

Rozpoznání konce souboru

Při čtení ze souboru je třeba vyřešit jednu dodatečnou věc – jak rozpoznáme, že už jsme soubor přečetli celý a už v něm nic dalšího nezbývá? Pokud načítáme data ze souboru "binárně", tj. interpretujeme je jako byty a ne jako (ASCII) text, obvykle stačí si velikost souboru předpočítat po jeho otevření pomocí funkcí `ftell` a `fseek` nebo si ji přečíst přímo ze samotného souboru[5].

[5]Spousta binárních formátů (např. `JPEG`) jsou tzv. **samo-popisné** (*self-describing*), což znamená, že typicky na začátku souboru je v pevně stanoveném formátu uvedeno, jak je daný soubor velký. Využijeme toho například při práci s obrázkovým formátem `TGA`.

Co ale dělat, když načítáme textové soubory, jejichž formát obvykle není ani zdaleka pevně daný? Předpočítat si velikost souboru a pak muset po každém načtení např. řádku počítat, kolik znaků jsme vlastně načetli, by bylo relativně komplikované. Při čtení textových souborů se tak obvykle využívá jiná strategie – čteme ze souboru tak dlouho, dokud nedojde k chybě. Způsob detekce chyby záleží na použité funkci:

- `fscanf` vrátí číslo `<= 0`, pokud se jí nepodaří načíst žádný zástupný znak ze vstupu.
- `fgets` vrátí ukazatel s hodnotou `0`, pokud dojde k chybě při čtení.

Jakmile dojde k chybě, tak bychom ještě měli ověřit, jestli jsme opravdu na konci souboru, anebo byla chyba způsobena něčím jiným[6]. To můžeme zjistit pomocí funkcí `feof`, která vrátí nenulovou hodnotu, pokud jsme se před jejím zavoláním pokusili o čtení a pozice již byla na konci souboru, a `ferror`, která vrátí nenulovou hodnotu, pokud došlo k nějaké jiné chybě při práci se souborem.

[6]Například pokud čteme soubor z USB disku, který je během čtení odpojen od počítače.

Program, který by načítal a rovnou vypisoval řádky textu ze vstupního souboru, dokud nedojde na jeho konec, by tedy mohl vypadat například takto:

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

int main() {
    FILE* soubor = fopen("soubor.txt", "r");
    assert(soubor);

    char radek[80];
```

```
    while (1) {
        if (fgets(radek, sizeof(radek), soubor)) {
            // radek byl uspesne nacten
            printf("Nacteny radek: %s", radek);
        }
        else {
            if (feof(soubor)) {
                printf("Dosli jsme na konec souboru\n");
            } else if (ferror(soubor)) {
                printf("Pri cteni ze souboru doslo k chybe: %s\n", strerror(errno));
            }

            break;
        }
    }

    fclose(soubor);

    return 0;
}
```

Modularizace

Prozatím byly naše programy tvořeny pouze jedním zdrojovým souborem. Pro krátké programy do pár stovek řádků to stačí, nicméně asi si dovedete představit, že programy s tisíce či miliony řádky kódu už se do jednoho souboru rozumně "nevlezou". V této sekci si tak ukážeme, jak programy v *C* rozdělit do více zdrojových souborů.

Rozdělení programu do více souborů má spoustu výhod:

- **Větší přehlednost** - pokud by byl veškerý kód v jednom zdrojovém souboru, tak by se v takovém souboru dalo u větších programů jen těžko vyznat. Pokud budou jednotlivé logické části programu umístěny v samostatných souborech či adresářích, bude např. mnohem jednodušší najít část programu, kterou chceme upravit.

  Například u hry bychom mohli rozdělit program do souborů `zvuk.c`, `grafika.c`, `ovladani_priser.c`, `zbrane.c`, `klavesnice.c`, `mys.c` atd. Pokud by některý z těchto souborů byl opět moc velký nebo složitý, můžeme jeho funkcionalitu rozdělit na více souborů.

- **Menší provázanost** - pokud je vše v jednom souboru, znamená to, že z libovolné funkce lze volat všechny ostatní funkce (popř. používat všechny ostatní struktury atd.). Toto vede k situaci, kdy jsou jednotlivé části programu na sobě vzájemně závislé a propojené. To možná zní nevinně, nicméně ve skutečnosti to téměř nevyhnutelně vede k programu, který je velmi obtížné upravit. Pokud totiž změníte jednu věc, často se musí změnit i všechny další věci (funkce, struktury), které na dané věci závisí. Pokud závisí vše se vším, tak i malá změna v jedné části kódu může kaskádově vyvolat nutnost upravit celý zbytek programu, což je náročné.

  Abychom tomu předešli, je vhodné učinit jednotlivé části programu samostatné, sdílet z nich se zbytkem kódu pouze to, co je opravdu potřeba, a zbytek funkcionality učinit "soukromý" pro daný soubor. Změny v těchto soukromých částích pak nemohou ovlivnit zbytek kódu, protože ten na nich nebude závislý.

- **Efektivnější spolupráce v týmu** - rozdělení na více souborů také usnadní týmovou spolupráci. Pokud budou jednotliví programátoři upravovat jiné soubory, bude mnohem menší riziko tzv. "souběhu", kdy by jejich změny ve stejném souboru mohly kolidovat. Tyto problémy pak dále řeší tzv. verzování , o kterém se dozvíte v navazujících předmětech.

- **Znovuvyužití kódu** - pokud by každý program musel implementovat veškerou funkcionalitu od nuly, tak by

bylo programování i jednoduchého programu nesmírně náročné.[1] V rámci jednoho programu s můžeme nějakou ucelenou funkcionalitu (např. sadu funkcí spolu se strukturami) vyčlenit do samostatného souboru, což nám umožní ji opakovaně používat z ostatních souborů v našem programu. Napříč programy pak můžeme sdílet kód pomocí tzv. **knihoven** (*libraries*). Pro obojí musíme umět používat kód, který se nenachází ve zdrojovém souboru, ze kterého ho chceme využít.

[1] Ostatně například bez standardní knihovny *C* bychom v našem programu ani nebyli schopni něco vypsat do terminálu.

V programovacích jazycích se obecně různé samostatné části kódu, které jsou typicky umístěny v adresářích či souborech, a starají se o konkrétní funkcionalitu v programu, nazývají *moduly* . Proto je tato sekce nazvána modularizace. Jedná se však spíše o obecný pojem, v jazyce *C* se přímo s pojmem modul zase tak běžně nesetkáte.

Postupně si ukážeme:

- jak funguje překlad programů s více zdrojovými soubory
- jak používat funkce a proměnné z jiných souborů
- jaké jsou konvence pro používání více zdrojových souborů v C
- jak používat kód, který napsal někdo jiný, a nasdílel ho v podobě knihovny

Linker

V této sekci si vysvětlíme detailněji, jak probíhá překlad *C* programů, jehož základní fungování již bylo stručně popsáno v sekci o překladu. Díky tomu pak budeme schopni vytvářet programy skládající se z více než jednoho zdrojového souboru.

Prozatím jsme naše programy (skládající se z jediného zdrojového souboru) překládali pomocí příkazu podobnému tomuto:

```
$ gcc soubor.c -o program
```

Tímto příkazem jsme ve skutečnosti prováděli dvě věci najednou: **překlad** (*translation*) a **linkování** (*linking*). Níže si vysvětlíme obě dvě tyto části detailněji.

Překlad a linkování se dohromady nazývá **kompilace** programu.

Překlad programu

Programy v *C* se skládají z jedné či více tzv. **jednotek překladu** (*translation unit*). Jedná se o nezávislé komponenty, ze kterých je nakonec vytvořen cílový program. Každá jednotka je obvykle tvořena jedním zdrojovým souborem (obvykle s příponou `.c`). Při překladu **překladač** převede jednotku ze zdrojového kódu v *C* do instrukcí procesoru, tzv. **objektového kódu** (*object code*).

Pokud chceme překladačem `gcc` (pouze) přeložit zdrojový soubor do objektového kódu (resp. objektového souboru), použijeme přepínač `-c`:

```
$ gcc -c soubor.c
```

Pokud nezadáme název výstupu pomocí přepínače `-o`, tak `gcc` implicitně vytvoří objektový soubor `<nazev-vstupu>.o`

(tj. zde `soubor.o`).

Jednotky překladu jsou na sobě nezávislé, můžeme tedy každou jednotku (zdrojový soubor) přeložit zvlášť:

```
$ gcc -c a.c
$ gcc -c b.c
...
```

Jak ale nyní jednotlivé soubory propojíme? Aby vůbec mělo rozdělení do více jednotek (souborů) smysl, tak musíme mít možnost v jednom souboru používat kód (např. funkce nebo globální proměnné), který je nadefinovaný v jiném souboru. V *C* toto propojení jednotek neprobíhá při překladu, ale až v následné fázi nazývané linkování.

Linkování programu

Jakmile přeložíme všechny naše zdrojové soubory postupně do objektových souborů, potřebujeme z nich vytvořit finální spustitelný soubor, což je práce programu nazývaného **linker**. Linker obdrží seznam všech (již přeložených) objektových souborů, ze kterých se má program skládat, propojí je dohromady a vytvoří z nich spustitelný soubor.

Jak propojení jednotlivých souborů probíhá? Představme si například, že v souboru `a.c` voláme funkci `foo`, která v tomto souboru neexistuje. Při překladu tohoto souboru překladač vytvoří objektový soubor `a.o`, ve kterém bude uložena informace, že voláme funkci `foo`. Dejme tomu, že tato funkce existuje v souboru `b.c`, který je přeložen do objektového souboru `b.o`. Při linkování linker obdrží seznam všech objektových souborů, tedy `a.o` i `b.o`. Když narazí na informaci, že z `a.o` chceme volat funkci `foo`, pokusí se tuto funkci naleznout v některém z předaných objektových souborů:

- Pokud jej nenalezne, tak vypíše chybu a program se nepřeloží[1].

  [1]V takovém případě byste se setkali s chybou `undefined reference to 'foo'`.
- Pokud jej nalezne (v tomto případě v `b.o`), tak volání funkce "propojí" tak, aby se volala správná funkce původně vytvořená v `b.c`.

Manuální použití linkeru [2] je relativně složité, proto i linker budeme používat přes `gcc`. Tomu můžeme předat sadu objektových souborů a on se postará o správné zavolání linkeru, který je spojí a vytvoří finální spustitelný soubor:

[2]Na Linuxu lze naleznout například linker `ld`.

```
$ gcc a.o b.o -o program
```

Při finálním linkování programu také dochází ke kontrole toho, jestli je v některém z objektových souborů obsažena funkce `main`, aby program věděl, kde má začít své vykonávání.

Proč takto složitě?

Možná vás napadlo, proč kompilace *C* programů probíhá takto komplikovaně a nestačí prostě překladači dát všechny zdrojové soubory našeho programu tak, jak jsme to dělali doposud:

```
$ gcc soubor1.c soubor2.c soubor3.c ...
```

Ve skutečnosti i to lze provést (tento postup se nazývá tzv. **unity build**). Nicméně má dvě vážné nevýhody:

- Pokud bychom neuměli používat zvlášť překladač a linker, nemohli bychom používat knihovny , u kterých obvykle nemáme přístup k samotnému zdrojovému kódu, ale pouze k již přeloženému objektovému kódu[3].

  [3]Například proto, aby autor knihovny zatajil původní zdrojový kód, který je jeho duševním vlastnictvím.

- Pokud bychom překládali celý náš program najednou, při sebemenší změně kódu bychom museli přeložit všechny soubory znovu. Pokud bychom tak měli obrovský program s tisícem zdrojových souborů a změnili jeden znak v jednom souboru, muselo by se všech tisíc souborů přeložit znovu, což může být dost pomalé[4]. Pokud překládáme každý soubor zvlášť, tak po změně v jednom souboru stačí přeložit daný soubor a znovu slinkovat všechny objektové soubory (ty původní můžeme znovuvyužít, protože se nezměnily). To je u velkých programů mnohem rychlejší než překládat vše od nuly.

  [4]Velké programy v *C* může trvat přeložit klidně i několik hodin nebo dokonce dnů!

Používání kódu z jiných souborů

Nyní už víme, jak přeložit program skládající se z více jednotek překladu (zdrojových souborů) a následně tyto jednotky spojit dohromady pomocí linkeru. V této sekci si ukážeme, jak můžeme použít kód, který existuje v jiném zdrojovém souboru.

Pokud chceme zavolat funkci, kterou jsme napsali v jiném souboru, můžeme ji prostě zavolat a linker se postará o zbytek:

```
// soubor1.c
int main() {
    moje_funkce();
    return 0;
}

// soubor2.c
void moje_funkce() {}
```

Pokud tyto dva soubory přeložíme a poté slinkujeme, tak se zavolá správná funkce:

```
$ gcc -c soubor1.c
$ gcc -c soubor2.c
$ gcc soubor1.o soubor2.o -o program
```

Nicméně, pokud bychom používali kód z jiných souborů takto "naslepo", narazili bychom na několik problémů. Tím, že překladač v souboru `soubor1.c` nemá přístup k popisu funkce `moje_funkce`, tak nemůže ověřit, jestli jsme jí předali správné počet argumentů se správnými datovými typy, a ani neví, jaká je návratová hodnota této funkce. Tento přístup navíc nebude vůbec fungovat pro použití globálních proměnných (při pokusu o použití neexistující proměnné by překladač ohlásil chybu).

Deklarace vs definice

Ideálně bychom potřebovali překladači říct, jak bude kód, který chceme použít, vypadat -- jaký bude datový typ a název globální proměnné, popř. jaké budou parametry, návratový typ a název funkce. Toho můžeme dosáhnout pomocí tzv. **deklarace** (*declaration*).

Deklarace "slibuje", že bude v programu existovat nějaká proměnná či funkce s konkrétním názvem a typem, ale

neříká, kde bude tato proměnná či funkce vytvořena (může to být například v jiném zdrojovém souboru). Samotné vytvoření funkce či proměnné se nazývá **definice** (*definition*). Zatím jsme tedy prováděli vždy definice funkcí i proměnných, nyní si ukážeme, jak vytvořit deklaraci.

Deklaraci funkce provedeme tak, že zadáme její signaturu, ale ne její tělo:

```
int funkce(int a, int b);           // deklarace funkce
int funkce(int a, int b) { ... }    // definice funkce
```

Deklaraci globální proměnné lze provést tak, že před ní dáme klíčové slovo `extern`[1]:

[1]Toto klíčové slovo můžeme použít i před deklarací funkce, nicméně není to potřeba, `extern` je na tomto místě předpokládáno implicitně.

```
extern int promenna;    // deklarace proměnné
int promenna;           // definice proměnné
```

Při sdílení kódu napříč soubory má smysl se bavit pouze o globálních proměnných. Lokální proměnné lze totiž používat vždy pouze v jejich funkci.

Díky deklaracím tak můžeme v jednom zdrojovém souboru určit, jak mají vypadat funkce/proměnné, které chceme používat, aby překladač mohl provádět kontrolu datových typů. Linker pak během linkování použije správné proměnné/funkce z odpovídajících zdrojových souborů. Více o tom, kde a jak deklarace vytvářet, se dozvíme v příští sekci o hlavičkových souborech.

Jednoprůchodový překlad

Z historických důvodů překladače *C* fungují v tzv. jednoprůchodovém režimu (*one-pass compilation*). Znamená to, že překladač "čte" náš zdrojový kód shora dolů, a v momentě, kdy chceme například použít nějakou funkci nebo proměnnou, tak již překladač dříve musel vidět (alespoň) její deklaraci, popř. rovnou i definici.

Například v následujícím programu:

```
void funkce1() {
    funkce2();
}
void funkce2() {}
```

si překladač bude stěžovat na to, že na řádku 2 nezná funkci `funkce2`, protože tato funkce je v souboru nadefinovaná až po funkci `funkce1`, která ji používá:

```
test.c: In function ‘funkce’:
test.c:2:5: warning: implicit declaration of function ‘funkce2’;
    2 |     funkce2();
```

Pokud tedy potřebujeme nadefinovat funkci na pozdějším místě, než je její první použití, můžeme nejprve vytvořit její deklaraci a až později (popř. v úplně jiném souboru) vytvořit její definici:

```
void funkce2();     // deklarace

void funkce1() {
    funkce2();
}
```

```
void funkce2() {}   // definice
```

Takovýto program už se přeloží bez varování. Koncept deklarování funkcí či proměnných v jednoprůchodových překladačích se nazývá **dopředná deklarace** (*forward declaration*).

Pravidlo jedné definice

V *C* platí tzv. **pravidlo jedné definice** (*one definition rule*). Každá proměnná i funkce musí být v programu *nadefinována* právě jednou (deklarována může být vícekrát). To platí jak v rámci jednoho souboru, tak v rámci celého programu (tj. napříč všemi zdrojovými soubory).

- Pokud bychom proměnnou či funkci pouze nadeklarovali a/nebo použili bez definice:

  ```
  // soubor.c
  void funkce();

  int main() {
      funkce();
      return 0;
  }
  ```

  Tak by kompilace selhala v době linkování, protože by nenašel žádnou funkci/proměnnou, kterou by mohl použít:

  ```
  $ gcc -c soubor.c
  $ gcc soubor.o
  /usr/bin/ld: test.o: in function `main':
  test.c:(.text+0xe): undefined reference to `funkce'
  collect2: error: ld returned 1 exit status
  ```

- Pokud bychom naopak nadefinovali proměnnou či funkci více než jednou:

  ```
  // soubor1.c
  void funkce() {}

  int main() {
      funkce();
      return 0;
  }
  // soubor2.c
  void funkce() {}
  ```

  Tak by linkování opět selhalo, protože by linker nevěděl, kterou definici použít:

  ```
  $ gcc -c soubor1.c
  $ gcc -c soubor2.c
  $ gcc soubor1.o soubor2.o
  /usr/bin/ld: soubor2.o: in function `funkce':
  soubor2.c:(.text+0x0): multiple definition of `funkce'; test.o:test.c:(.text+0x0): first
  defined here
  collect2: error: ld returned 1 exit status
  ```

Viditelnost funkcí a proměnných

Z jiných souborů lze používat pouze funkce a proměnné, které jsou *veřejné*. Implicitně jsou všechny funkce i všechny globální proměnné veřejné. Pokud byste chtěli zamezit tomu, aby mohly ostatní soubory používat nějakou funkci nebo globální proměnnou, můžete ji označit klíčovým slovem `static`, abyste z nich udělali *soukromé* funkce či proměnné:

```
static void soukroma_funkce() {}
static int soukroma_promenna;
```

Takovéto funkce a proměnné půjde používat pouze v souboru, ve kterém byly nadefinovány. Doporučujeme `static` používat pro označení proměnných a funkcí, které nechcete sdílet se zbytkem programu. Půjde tak na první pohled poznat, které funkce jsou určeny k použití z jiných souborů a které ne[2].

[2]Použití `static` také může v určitých případech vést k vygenerování efektivnějšího kódu a menší velikosti výsledného spustitelného souboru.

Klíčové slovo `static` lze také použít u lokálních proměnných, zde má ovšem úplně jiný význam než u globálních proměnných! Použití `static` u lokální proměnné z ní udělá globální proměnnou, kterou ovšem půjde použít pouze ve funkci, ve které je vytvořena. Takováto proměnná se nainicializuje, když se program poprvé dostane k řádku s její definicí. Proměnná bude existovat po celou dobu běhu programu a udrží si svou hodnotu i po skončení volání funkce:

```
#include <stdio.h>

void test() {
  static int x = 0;
  x += 1;
  printf("%d\n", x);
}

int main() {
  test();
  test();
  return 0;
}
```

Hlavičkové soubory

Nyní už víme, že pro použití kódu z jiných souborů bychom nejprve měli dané funkce a proměnné deklarovat. Pokud bychom však museli v každém souboru, ve kterém chceme použít kód z jiného souboru, museli vytvářet deklarace pro každou funkci či proměnnou, kterou chceme použít, bylo by to docela zdlouhavé. Pokud by navíc došlo ke změně datového typu či názvu takovéto sdílené funkce či proměnné, museli bychom deklarace upravit ve všech programech, kde funkci či proměnnou používáme.

Pro vyřešení tohoto problému se v *C* často využívá koncept tzv. **hlavičkových souborů** (*header files*). Pro každý zdrojový soubor, jehož kód chceme sdílet, vytvoříme hlavičkový soubor, který bude obsahovat deklarace všech sdílených veřejných funkcí a globálních proměnných z daného zdrojového souboru. Ve zdrojovém souboru pak budou jejich definice. Dle jmenné konvence se hlavičkový soubor pojmenovává jako `<název zdrojového souboru>.h`:

```
// soubor.h
int moje_funkce();
extern int moje_promenna;

// soubor.c
int moje_funkce() {}
int moje_promenna;
```

Hlavičkový soubor tak udává tzv. rozhraní odpovídajícího zdrojového souboru -- obsahuje seznam funkcí a proměnných, které jsou sdílené a zbytek programu je může používat.

Ostatní soubory, které chtějí funkce z nějakého zdrojového souboru použít, pak vloží jeho hlavičkový soubor pomocí preprocesoru, aby mohly používat sdílené funkce a globální proměnné s korektní kontrolou datových typů:

```
// main.c
#include "soubor.h"

int main() {
    moje_funkce();
    int x = moje_promenna;

    return 0;
}
```

Pokud dojde ke změně signatury funkce či typu/názvu proměnné, tak stačí změnu udělat v hlavičkovém (a odpovídajícím zdrojovém) souboru. Všechny ostatní soubory, které danou funkci nebo proměnnou používají, pak budou okamžitě využívat upravenou deklaraci díky použití `#include`.

S hlavičkovými soubory jsme již setkali při použití standardní knihovny . V souborech jako je `stdio.h` se nazývají deklarace funkcí jako je například `printf`, jejichž definice je poté obsažena v objektových souborech standardní knihovny.

Obsah hlavičkového souboru

Jelikož hlavičkové soubory jsou určeny k tomu, aby byly využívány (vkládány) v různých zdrojových souborech, tak se jejich obsah přirozeně může vyskytnout ve více jednotkách překladu. Aby tak nebylo porušeno pravidlo jedné definice, je důležité do hlavičkových souborů dávat **pouze deklarace, a ne definice** funkcí a proměnných!

Pokud byste do hlavičkového souboru dali například definici funkce, a tento soubor by se vyskytnul ve více jednotkách překladu, tak by linkování selhalo kvůli vícenásobné definici. Pokud byste přecejenom opravdu chtěli definici nějaké funkce "propašovat" do hlavičkového souboru, můžete před ní použít klíčové slovo `inline`:

```
// soubor.h
inline void moje_funkce() { ... }
```

Tímto klíčovým slovem slibujete linkeru, že všechny definice funkce s tímto názvem jsou stejné. Pokud tak linker narazí na definici této funkce vícekrát (což nastane, když tento hlavičkový soubor bude vložen ve více jednotkách překladu), tak nebude hlásit chybu, ale prostě si jednu z těchto definicí vybere. Pokud by definice stejné nebyly, může to vést k dost zvláštnímu chování. Pokuste se tak `inline` raději nevyužívat.

> U proměnných nemá valný důvod `inline` používat.

Kromě deklarací funkcí a proměnných se do hlavičkových souborů také běžně vkládají struktury, které jsou součástí

typů sdílených proměnných či parametrů nebo návratových hodnot sdílených funkcí.

Aby mohly zdrojové soubory používat sdílené struktury i sdílené funkce v libovolném pořadí, tak obvykle zdrojové soubory vkládají svůj vlastní hlavičkový soubor:

```
// soubor.h
typedef struct {
    int vek;
} Osoba;

int zpracuj_osobu(Osoba osoba);

// soubor.c
#include "soubor.h"
int zpracuj_osobu(Osoba osoba) { ... }
```

Pro použití struktur nebo např. `typedefů` je také běžné, že hlavičkové soubory vkládají jiné hlavičkové soubory.

Ochrana vkládání

U hlavičkových souborů je nutné řešit ještě jednu další věc. Jelikož se běžně používají v kombinaci s `#include`, může se stát, že i v rámci jedné jednotky překladu se jeden hlavičkový soubor vloží do výsledného zdrojového souboru více než jednou. To může způsobovat různé typy problémů:

- Pokud se budou hlavičkové soubory vkládat navzájem, mohlo by dojít k cyklické závislosti. Například zde by překlad selhal, protože by se hlavičkové soubory snažili vložit se navzájem donekonečna:

  ```
  // a.h
  #include "b.h"


  // b.h
  #include "a.h"
  ```

- Hlavičkový soubor se zbytečně vícekrát načítá překladačem, což prodlužuje dobu překladu.
- Pokud by hlavičkový soubor obsahoval nějakou definici, tak i kdyby byl použit pouze v jedné jednotce překladu, došlo by k chybě při linkování, protože by definice byla zduplikovaná.

Abychom těmto situacím zamezili, tak u hlavičkových souborů budeme používat tzv. **ochranu vkládání** (*include guard*). Pomocí ochrany vkládání zajistíme, že jeden hlavičkový soubor se v rámci jedné jednotky překladu vloží maximálně jednou.

Zamezení vícenásobného vložení můžeme dosáhnout pomocí podmíněného překladu:

```
// soubor.h
#ifndef SOUBOR_H
#define SOUBOR_H

void moje_funkce();

#endif
```

Tohle je nicméně trochu zdlouhavé. Moderní překladače obsahují mnohem jednodušší způsob. Na začátek hlavičkového souboru stačí vždy vložit řádek `#pragma once` a dál nemusíte nic řešit:

```
// soubor.h
#pragma once

void moje_funkce();
```

Knihovny

Nyní už známe vše potřebné na to, abychom si rozdělili náš vlastní program do libovolného množství zdrojových souborů. Často také ale budeme chtít používat kód, který už před námi napsal někdo jiný. Pokud bychom si totiž museli vše psát od nuly, tak bychom se daleko nedostali[1], respektive trvalo by nám to dlouho.

[1] I když napsat si nějaký systém "od nuly" je dobrý způsob, jak se zlepšit v programování.

Aby programátoři mohli sdílet svůj kód s ostatními programátory, tak využívají tzv. **knihovny** (*libraries*). Knihovna je kód, který řeší nějakou ucelenou funkcionalitu (např. vykreslování grafiky , sazbu fontů nebo kompresi dat) a obsahuje návod (dokumentaci), jak tento kód používat. Klíčové vlastnosti knihoven jsou znovupoužitelnost (můžeme je použít v různých programech) a abstrakce (nemusíme rozumět, jak knihovna funguje, pouze ji využijeme k vyřešení konkrétního problému).

> Knihovna není program – neobsahuje žádnou funkci `main` a nelze ji ani přímo spustit. V kontextu jazyka *C* je knihovna typicky sada funkcí, struktur a globálních proměnných.

Například pokud bychom programovali hru, můžeme využít knihovny na vykreslení grafiky, na přehrávání zvuku, na snímání vstupu z klávesnice/myši atd. Náš kód se pak může zabývat zejména logikou hry a nemusí tolik řešit problémy, které již vyřešila spousta programátorů před námi.

Na internetu můžete naleznout tisice různých knihoven, které řeší rozlišné problémy.

Sdílení knihoven

Teoreticky bychom mohli knihovny používat prostě tak, že si nějakou najdeme na internetu, stáhneme její hlavičkové a zdrojové soubory k našeho programu a začneme je využívat. I když i tak to lze někdy udělat, není to obvyklé, protože tento přístup má spoustu nevýhod:

- Jelikož obvykle nebudeme autory knihovny, kterou chceme použít, tak nemusíme ani být schopní danou knihovnu přeložit. Potřebuje daná knihovna konkrétní překladač nebo jeho specifické nastavení? Má závislosti na dalších knihovnách? Přeložit "cizí" knihovnu ze zdrojových souborů nemusí být zdaleka přímočaré.
- Pokud dojde k vydání nové verze knihovny, která může přinášet opravy chyb a novou funkcionalitu, museli bychom (kromě potenciální úpravy našeho kódu) také překopírovat nebo správně upravit nové a změněné soubory knihovny, což by bylo náročné a náchylné na chyby.
- Zdrojový kód knihoven není vždy zveřejněn, například aby si jejich autoři uchránili duševní vlastnictví. Často se tak setkáme se situací, že máme k dispozici pouze objektový kód (např. `.dll` nebo `.so`) a nemůžeme zkopírovat k našemu programu zdrojové soubory.

Z tohoto důvodu jsou knihovny obvykle sdíleny ve formě objektových souborů (ty obsahují implementaci funkcí) a odpovídajících hlavičkových souborů (ty obsahují deklarace , aby šlo knihovnu jednoduše používat).

Statické vs dynamické knihovny

Předávat překladači desítky či stovky objektových souborů by bylo docela nepraktické, proto se tyto soubory při distribuci knihovny balí do jednoho či více archivů, které mají standardizovaný formát a překladače s nimi umí přímo pracovat. Knihovna může být distribuována v jednom ze dvou typů archivů, které určují to, jak bude daná knihovna "přilinkována" (připojena) k našemu programu:

- **Dynamická knihovna** (*dynamic library*) - objektové soubory takovéto knihovny nebudou součástí našeho programu (tj. nebudou obsaženy ve spustitelném souboru, který bude vytvořen překladačem). K jejich načtení dojde až "dynamicky" při spuštění programu[2].

  [2]O toto načítání se stará tzv. dynamický linker .

  Výhody tohoto přístupu jsou, že bude mít náš spustitelný soubor menší velikost, a to jak na disku, tak v operační paměti. Operační systém totiž dokáže jednu dynamickou knihovnu sdílet více programům zároveň. Dynamickou knihovnu také půjde aktualizovat bez nutnosti překládat znovu náš program a můžeme také při spuštění programu knihovnu nahradit jinou implementací.

  Nevýhodou je, že při spuštění našeho programu musíme zajistit, že knihovna bude na daném systému k dispozici (pokud by nebyla nalezena, tak program nepůjde spustit). To může být způsobovat problémy zejména při distribuci našeho programu na ostatní počítače. Kvůli tomu, že se knihovna načítá dynamicky, také může v určitých případech být její použití méně efektivní než v případě statické knihovny.

  Archivy s objektovými soubory dynamických knihoven mají příponu `.so`.

- **Statická knihovna** (*static library*) - objektové soubory takovéto knihovny budou přímo přibaleny k našemu programu (jako bychom je přímo jeden po druhém předali překladači).

  Výhody tohoto přístupu jsou, že náš program bude "samostatný" – knihovnu bude obsahovat uvnitř svého spustitelného souboru, takže nebude nutné ji mít dostupnou na cílovém systému (narozdíl od dynamické knihovny).

  Nevýhodou je, že výsledný spustitelný soubor bude větší a knihovnu nepůjde aktualizovat bez opětovného překladu celého programu.

  Archivy s objektovými soubory statických knihoven mají příponu `.a`.

Názvy přípon statických a dynamických knihoven závisí na operačním systému. Například na Windows se můžete setkat s příponami `.lib` pro statické knihovny a `.dll` pro dynamické knihovny.

Použití knihoven s `gcc`

Nyní si ukážeme, jak říct překladači `gcc`, aby připojil nějakou knihovnu k našemu programu. Pro to musíme mít k dispozici archiv s objektovými soubory knihovny (s příponou `.a` nebo `.so`, v závislosti na typu knihovny) a obvykle také i adresář s hlavičkovými soubory knihovny.

Nejprve si ukážeme, jak překladači předat cestu k hlavičkovým souborům knihovny. Ty obvykle nebudou součástí našich zdrojových kódů, ale budou nainstalovány v nějakém systémovém adresáři (jako tomu je např. u `stdio.h`). Budeme je tedy chtít vkládat pomocí syntaxe `#include <>`. Překladači můžeme předat dodatečné adresáře, ve kterých

má hledat (hlavičkové) soubory pro vkládání, pomocí přepínače `-I`. Pokud bychom tak měli hlavičkové soubory knihovny např. v adresáři `/usr/foo/include`, tak překladači při překladu předáme přepínač `-I/usr/foo/include`.

Dále je třeba překladači říct, kde nalezne archivy s objektovými soubory knihovny. K tomu slouží dva přepínače. `-L` udává adresář, ve kterém se budou vyhledávat knihovny a `-l` poté specifikuje konkrétní knihovnu, která má být přilinkována k našemu programu. Pokud bychom tak měli například archiv knihovny v souboru `/usr/foo/lib/libknihovna.so`, tak překladači předáme parametry `-L/usr/foo/lib` a `-lknihovna`. Při použití přepínače `-l` je třeba si dávat pozor na dvě věci:

- Všimněte si, že se použila zkrácená konvence pro pojmenování knihovny. Obecně se knihovny pojmenovávají `lib<název>.so` (nebo `lib<název>.a`) a překladači se poté předává pouze jejich název, tj. `-l<název>`.
- Přepínač `-l` se aplikuje na zdrojové/objektové soubory, které byly v příkazové řádce zadány před ním. Používejte jej tedy až po předání vašich zdrojových souborů:

```
# správně
$ gcc main.c -lknihovna

# špatně
$ gcc -lknihovna main.c
```

Celý příkaz pro připojení knihovny k vašemu programu by tak mohl vypadat např. takto:

```
$ gcc -o program main.c -L/usr/foo/lib/ -lfoo -I/usr/foo/include
```

Předání cesty k dynamické knihovně

Pokud přeložíte program s dynamickou knihovnou, může se stát, že při jeho spuštění nebude schopen danou knihovnu najít. V takovém případě při spuštění programu můžete pomocí **proměnné prostředí**[3] (*environment variable*) `LD_LIBRARY_PATH` předat cestu k adresáři, ve které se daná knihovna nachází:

[3]Proměnné prostředí jsou způsobem, jak parametrizovat chování programů (podobně jako například parametry příkazového řádku. V programu si můžete přečíst hodnotu konkrétní proměnné prostředí pomocí funkce `getenv`.

```
$ LD_LIBRARY_PATH=/usr/foo/lib ./program
```

Zobrazení vyžadovaných dynamických knihoven

Pokud si přeložíte nějaký program a použijete na něj program `ldd`, dozvíte se, které dynamické knihovny vyžaduje ke svému běhu. Měli byste mezi nimi naleznout mj. i standardní knihovnu *C* (`libc`) a dozvědět se tak její umístění na disku:

```
$ ldd ./program
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0d3a328000)
```

Standardní knihovna jazyka *C* je používána téměř každým programem a mj. z tohoto důvodu je obvykle linkována dynamicky, aby její paměť šla sdílet mezi programy.

Vytvoření knihovny

Pokud byste si chtěli vytvořit vlastní knihovnu, můžete toho jednoduše dosáhnout pomocí `gcc`. Dejme tomu, že máte soubory `a.c` a `b.c`, které chcete zabalit do knihovny. Nejprve každý zdrojový soubor přeložíme do objektového souboru[4]:

[4] Parametr `-fPIC` je nutný při překladu zdrojových souborů, které poté chceme umístit do knihovny. Více se můžete dozvědět např. zde .

```
$ gcc -c -fPIC a.c
$ gcc -c -fPIC b.c
```

Další postup závisí na tom, jaký typ knihovny chceme vytvořit:

- **Vytvoření statické knihovny** - použijeme program `ar` (archiver):

  ```
  $ ar rcs libknihovna.a a.o b.o
  ```

- **Vytvoření dynamické knihovny** - použijeme program `gcc` s přepínačem `-shared`:

  ```
  $ gcc -shared a.o b.o -o libknihovna.so
  ```

Automatizace překladu

Možná vás napadlo, že v případě rozdělení programu do více zdrojových souborů a při použití knihoven začne být docela namáhavé náš program vůbec přeložit. Musíme přeložit zvlášť každou jednotku překladu, nakonec je všechny slinkovat dohromady a případně ještě předat potřebné cesty k použitým knihovnám. A toto je třeba po jakékoliv změně v kódu našeho programu, pokud ji budeme chtít otestovat.

Tento problém se dá řešit různými způsoby, od vytvoření shell skriptu , pomocí kterého můžeme všechny tyto úkony provést pomocí jediného příkazu v terminálu, až po pokročilé **sestavovací systémy** (*build systems*), které umí automaticky vyhledávat cesty ke knihovnám a překládat pouze změněné soubory pro urychlení opakovaných překladů programu.

Bohužel neexistuje jednotný standardní sestavovací systém pro programy napsané v *C*. Různé projekty či knihovny tak používají různé sestavovací systémy, což může někdy představovat problém při jejich integraci do našich programů. Sestavovací systémy se navíc obvykle nastavují pomocí konfiguračních souborů, které jsou psány v proprietárních jazycích, které se musíte naučit a pochopit, abyste daný sestavovací systém mohli používat. Situaci nepomáhá ani to, že se od sebe jednotlivé systémy značně liší a bývají velmi složité.

`make`

Asi stále nejpoužívanějším sestavovacím systémem je `make`, který existuje už od roku 1976. Pro jeho použití musíte vytvořit soubor `Makefile`, ve kterém popíšete, jak se má váš program přeložit, a poté spustíte program `make`, který jej dle konfiguračního souboru přeloží.

Návod pro vytvoření konfiguračního souboru `Makefile` a použití `make` naleznete například zde .

CMake

Poněkud modernější alternativou je `CMake`. Jedná se o meta systém, ve skutečnosti totiž neřídí překlad vašeho programu, ale pouze generuje potřebné soubory pro nějaký jiný sestavovací systém, který váš program teprve přeloží. Výhodou pak je, že z jednoho `CMake` konfiguračního souboru tak můžete vygenerovat např. `Makefile` pro přeložení na Linuxu anebo jiné konfigurační soubory pro přeložení stejného programu pod Windows.

Další výhodou `CMake` je, že některá vývojová prostředí (např. Visual Studio Code nebo CLion) mu rozumí a dokáží díky němu usnadnit analýzu a ladění vašeho programu.

Instalace

`CMake` můžete na Ubuntu nainstalovat následujícím příkazem v terminálu:

```
$ sudo apt install cmake
```

Použití

Pro použití `CMake` musíte vytvořit konfigurační soubor `CMakeLists.txt`, ve kterém popíšete jednotlivé zdrojové soubory vašeho programu, a také zadáte knihovny, které chcete k vašemu programu připojit. Vzorový soubor `CMakeLists.txt` může vypadat např. takto:

```
cmake_minimum_required(VERSION 3.4)

# Název projektu
project(hra)

# Přidání přepínačů překladače
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fsanitize=address")

# Vyhledání knihovny SDL
find_package(SDL2)

# Vytvoření programu s názvem `hra`
# Program se bude skládat ze dvou zadaných zdrojových souborů (jednotek překladu)
add_executable(hra hra.c grafika.c)

# Přidání adresářů s hlavičkovými soubory k programu (obdoba -I)
target_include_directories(hra PRIVATE ${SDL2_INCLUDE_DIRS})

# Přilinkování knihoven k programu (obdoba -l)
target_link_libraries(hra ${SDL2_LIBRARIES} m)
```

Jakmile tento soubor vytvoříte, musíte pomocí příkazu `cmake` vytvořit `Makefile`:

```
$ mkdir build
$ cd build
$ cmake ..
```

a poté pomocí `make` program finálně přeložit:

```
$ make
```

Dobrá zpráva je, že pokud používáte kompatibilní vývojové prostředí, tak tyto úkony typicky provádí za vás a vám tak

stačí správně nastavit soubor `CMakeLists.txt`.

Návod k použití `CMake` naleznete například zde .

Použití ve Visual Studio Code

Pokud chcete spustit či ladit `CMake` projekt ve VSCode, tak proveďte tyto kroky:

1. Nainstalujte si toto rozšíření do VSCode
2. Otevřete ve VSCode složku, která bude obsahovat soubor `CMakeLists.txt`
3. Spusťte program pomocí `Ctrl + F5`

Úlohy

V této sekci si ukážeme několik jednoduchých aplikovaných přístupů a knihoven, které můžete použít například na:

- Práci s obrázky pomocí formátu TGA.
- Práci s animacemi pomocí formátu GIF.
- Tvorbě interaktivních aplikací a her pomocí knihovny SDL.
- Simulaci fyzikálních procesů pomocí knihovny Chipmunk .

TGA

TGA je formát pro ukládání rastrových obrázků na disk. Slouží tedy ke stejnému účelu jako známější formáty `JPEG` nebo `PNG`, ale oproti nim je mnohem jednodušší. Díky tomu můžeme načíst i zapsat `TGA` soubor pomocí několika řádků kódu, zatímco např. u `JPEG` nebo `PNG` bychom potřebovali buď použít již existující knihovnu anebo naimplementovat jejich relativně komplikované standardy, které čítají stovky stránek.

Hlavička `TGA`

Soubory ve formátu `TGA` obsahují na svém začátku tzv. **hlavičku** (*header*), která obsahuje informace popisující daný obrázek. Tyto informace jsou reprezentovány byty, které jsou umístěny na pevně daných pozicích. Zde je seznam jednotlivých částí hlavičky TGA:

Název	Pozice prvního bytu	Počet bytů
ID	0	1
Typ barevné mapy	1	1
Typ obrázku	2	1
Barevná mapa	3	5
Počátek X	8	2
Počátek Y	10	2
Šířka	12	2
Výška	14	2
Barevná hloubka	16	1
Popisovač	17	1

Tato tabulka udává, jak máme interpretovat jednotlivé byty na začátku `TGA` souboru. Pokud bychom tedy například otevřeli `TGA` soubor a přečteli si jeho 12. a 13. byte, tak se dozvíme šířku tohoto obrázku. Nás budou zajímat zejména tučně vyznačené části:

- **Typ**: Hodnota `2` udává nekomprimovaný RGB obrázek, hodnota `3` udává nekomprimovaný černobílý obrázek. Ostatní platné hodnoty typu obrázku můžete nalézt např. na Wikipedii.
- **Počátek**: Tato část hlavičky určuje, kde bude počátek souřadnicového systému obrázku. Jinak řečeno, pokud do obrázku zapíšete pixel na pozici `(0, 0)`, tak se objeví na pozici počátku. Pokud použijete počátek `(0, 0)`, tak tato pozice bude v levém dolním rohu obrázku. Počátek je rozdělený do souřadnic `x` a `y`, obě dvě zabírají dva byty.
- **Rozměry**: Tato část hlavičky určuje rozměry obrázku. Stejně jako u počátku šířka i výška zabírá dva byty (aby formát podporoval i obrázky s rozměry většími než 255 pixelů).
- **Barevná hloubka**: Udává, kolik bitů bude zabírat každý pixel obrázku. Pokud použijeme typ obrázku RGB, měli bychom použít hloubku 24 bitů (8 bitů na každou barevnou složku), pokud použijeme černobílý typ obrázku, tak použijeme hloubku 8 bitů.

Při načítání binárních dat ze souborů musíme dávat pozor na to, jestli jsou hodnoty uloženy v **little-endian** nebo **big-endian** formátu. U `TGA` je určeno, že musí být v little-endian, což je zároveň s velkou pravděpodobností i formát, který používá vás počítač, nemusíme tedy provádět žádnou konverzi. Více o tzv. **endianness** můžete nalézt např. zde .

Načtení hlavičky ze souboru

Jednotlivé části z hlavičky bychom mohli načítat byte po bytu, nicméně to by bylo dosti nepraktické. V případě, že formát, který chceme načíst, má pevně dané rozložení bytů, je mnohem jednodušší nadefinovat si strukturu , která bude danému rozložení odpovídat, a poté celou strukturu načíst ze souboru najednou.

Jednotlivé hodnoty v hlavičce jsou reprezentovány byty bez znaménka. Jelikož tento datový typ v *C* má trochu zdlouhavý název, vytvořme si pro něj nejprve nové jméno `byte`:

```
typedef unsigned char byte;
```

Nyní si vytvořme strukturu, která bude reprezentovat `TGA` hlavičku. Jednotlivé atributy struktury musí přesně odpovídat hodnotám v hlavičce a musí být také uvedeny ve stejném pořadí:

```
typedef struct {
    byte id_length;
    byte color_map_type;
    byte image_type;
    byte color_map[5];
    byte x_origin[2];
    byte y_origin[2];
    byte width[2];
    byte height[2];
    byte depth;
    byte descriptor;
} TGAHeader;
```

Možná vám přijde zvláštní, proč např. šířku definujeme jako pole dvou bytů namísto použití "dvou-bajtového celého čísla", tedy datového typu `short`. Děláme to, aby do této struktury překladač nevložil žádné mezery .

Pokud by je tam vložil, tak by naše struktura v paměti už neodpovídala hlavičce `TGA` v souboru a četli bychom tak neplatné hodnoty. Když použijeme pro všechny atributy datový typ s velikostí 1 byte, tak překladač žádné mezery vkládat nebude.

Nyní už stačí pouze otevřít nějaký `TGA` soubor (např. tento), načíst z něj počet bytů odpovídající naší struktuře a poté si z ní můžeme přečíst informace o daném obrázku:

```
FILE* file = fopen("carmack.tga", "rb");
assert(file);

TGAHeader header = {};
assert(fread(header, sizeof(TGAHeader), 1, file) == 1);

printf("Image type: %d, pixel depth: %d\n", header.image_type, header.depth);
```

Pokud budeme chtít pracovat s hodnotami rozměrů či počátku, musíme je nejprve převést z pole bytů na celé číslo. Toho můžeme dosáhnout pomocí funkce `memcpy`:

```
int width = 0;
int height = 0;

memcpy(&width, header->width, 2);
memcpy(&height, header->height, 2);
```

Načtení pixelů ze souboru

Jakmile jsme načetli hlavičku, můžeme načíst ze souboru i samotné pixely. Ty jsou umístěny v souboru hned za hlavičkou. Každý pixel má odpovídající počet bytů podle typu obrázku (u RGB 3 byty[1], u černobílých obrázků 1 byte) a počet pixelů je dán rozměry obrázku (šířka * výška).

[1] V `TGA` jsou jednotlivé barevné složky uložené v pořadí `blue`, `green`, `red`. Jedná se tedy vlastně o formát BGR.

Můžeme si tak vytvořit pole pro pixely a načíst je z obrázku. Pro RGB obrázky by načtení pixelů mohlo vypadat např. takto:

```
typedef struct {
    byte blue;
    byte green;
    byte red;
} Pixel;

Pixel* load_pixels(TGAHeader header, FILE* file) {
    int width = 0;
    int height = 0;

    memcpy(&width, header.width, 2);
    memcpy(&height, header.height, 2);

    Pixel* pixels = (Pixel*) malloc(sizeof(Pixel) * width * height);
    assert(fread(pixels, sizeof(Pixel) * width * height, 1, file) == 1);
    return pixels;
}
```

Zapsání `TGA` do souboru

Pokud byste chtěli `TGA` obrázek naopak do souboru zapsat, tak musíte vytvořit hlavičku a nastavit do ní odpovídající hodnoty. Hlavičku poté musíte zapsat binárně do souboru a hned za ní zapsat všechny pixely obrázku, řádek po řádku.

GIF

GIF je velmi populární formát pro sdílení animací. `GIF` animace se skládá z jednoho nebo více tzv. **snímků** (*frames*), které mají určenou délku, po kterou se mají zobrazit. Při přehrání animace se pak jednotlivé snímky zobrazují postupně jeden za druhým, což vytváří dojem animace.

Pořád se jedná o relativně jednoduchý formát, nicméně je už trošku složitější než např. TGA, protože používá kompresi a pixely nejsou uloženy v souboru přímo, místo toho je každý pixel reprezentován indexem do tabulky (palety) předpřipravených barev.

Pro vytvoření `GIF` animace tak použijeme kód, který už pro nás připravil někdo jiný. Konkrétně se bude jednat o knihovnu `gifenc`[1]. Stáhněte si soubory `gifenc.c` a `gifenc.h` a použijte je při překladu svého programu.

[1] I když jsme se předtím bavili o tom, že sdílet knihovny ve formě zdrojových kódů není úplně běžné, tato knihovna je velmi malá a jednoduchá a zároveň je open-source, takže zkopírovat její zdrojové kódy do našeho programu je asi nejjednodušší způsob, jak ji použít.

Vytvoření `GIF` animace

Pro práci s `GIF` souborem si nejprve musíme nadefinovat tzv. **paletu** (*palette*). Paleta není nic jiného než pole barev, které můžeme v naší animaci používat. Jednotlivým pixelům každého snímku pak pouze řekneme, jaký index z této palety se má použít pro jejich vykreslení. Například tato paleta definuje čtyři barvy:

```
typedef unsigned char byte;

byte palette[] = {
    0x00, 0x00, 0x00, /* 0 -> černá */
    0xFF, 0x00, 0x00, /* 1 -> červená */
    0x00, 0xFF, 0x00, /* 2 -> zelená */
    0x00, 0x00, 0xFF, /* 3 -> modrá */
};
```

Pokud použijeme pro pixel index `1`, bude vykreslen červenou barvou, protože v této paletě se na pozici `1` nachází červená barva.

Jakmile máme nadefinovanou paletu, můžeme použít funkci `ge_new_gif`, která umožňuje vytvořit nový `GIF` soubor. Funkci musíme předat cestu k výstupnímu souboru, jeho rozměry, informace o paletě a o tom, kolikrát se má animace přehrát[2]:

[2] Pro použití hlavičkového souboru knihovny nezapomeňte na začátku svého programu vložit hlavičkový soubor `gifenc.h`.

```
int width = 300;
int height = 300;
```

```
ge_GIF* gif = ge_new_gif(
    "output.gif",
    width,
    height,
    palette,
    2,  /* hloubka palety */
    0   /* opakovat neustále dokola */
);
```

Parametr hloubky palety by měl být nastaven na dvojkový logaritmus počtu baret v paletě. V naší paletě jsou 4 barvy, takže jsme zde předali hodnotu parametru `2`. Poslední parametr udává, kolikrát se má animace přehrát. Hodnota `0` udává, že se má animace opakovat neustále dokola.

Zápis snímků

Když nyní máme vytvořenou animaci, můžeme do ní postupně zapisovat snímky. Zápis probíhá následovně:

1. Do pole uloženého v atributu `gif->frame` zapíšeme hodnoty všech pixelů jednoho snímku. Každá hodnota by měla být indexem odpovídající barvy z námi zvolené palety. Pro adresování použijeme klasický převod z 2D na 1D index.
2. Zavoláme funkci `ge_add_frame`, které řekneme, na jak dlouhou dobu se má tento snímek zobrazit. Tato doba je v setinách vteřiny.

Jakmile zapíšeme jeden snímek, můžeme celý proces opakovat pro zápis více snímků.

Uhodnete, jakou animaci vygeneruje následující kód [3]?

[3] Pro ověření tipu si program přeložte a podívejte se na výslednou animaci. Zakomentujte řádek s `memset` a zkuste odhadnout, jak a proč to změní výslednou animaci.

```
for (int i = 0; i < 100; i++) {
    memset(gif->frame, 0, sizeof(uint8_t) * width * height);

    for (int row = 0; row < height; row++) {
        gif->frame[row * height + i] = ((i * 10) / 30) % 3 + 1;
    }
    for (int col = 0; col < width; col++) {
        gif->frame[i * height + col] = ((i * 10) / 30) % 3 + 1;
    }

    ge_add_frame(gif, 8);
}
```

Výsledek animace

Dokončení práce s animací

Jakmile zapíšeme všechny snímky, které chceme v animaci mít, nesmíme zapomenout animaci uložit do souboru a uvolnit její paměť pomocí funkce `ge_close_gif`:

```
ge_close_gif(gif);
```

Načtení GIF animace

Pokud byste naopak chtěli nějakou GIF animaci načíst ze souboru a něco s ní dále provést, můžete použít knihovnu gifdec od stejného autora, která slouží k načítání GIF souborů.

SDL

SDL je knihovna pro tvorbu interaktivních grafických aplikací a her. Umožňuje vám vytvářet okna, vykreslovat do nich jednotlivé pixely, obrázky či text, snímat vstup z myši a klávesnice či třeba přehrávat zvuk. Jedná se tak v podstatě o tzv. **herní engine**, i když ve srovnání např. s enginy Unity nebo Unreal je tento engine velmi jednoduchý.

Instalace SDL

Narozdíl od knihovny, kterou jsme si ukazovali pro vytváření GIF animací , SDL obsahuje spoustu zdrojových i hlavičkových souborů, a nebylo by tak ideální ji kopírovat k našemu programu. Připojíme ji tedy k našemu programu jako klasickou knihovnu . Abychom knihovnu mohli použít, nejprve si ji musíme stáhnout. To můžeme udělat dvěmi způsoby:

- **Instalace pomocí správce balíčků** (*doporučeno*): Jelikož je SDL velmi známá a používaná knihovna, ve většině distribucí Linuxu není problém ji nainstalovat přímo z balíčkového manažeru. V Ubuntu to můžete provést pomocí následujícího příkazu v terminálu, který nainstaluje kromě balíčku se základní funkcionalitou také dva další balíčky nutné pro vykreslování obrázků a textu[1]:

  ```
  $ sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev
  ```

  Výhodou tohoto způsobu je, že knihovna bude nainstalována v systémových cestách, gcc ji tak bude mět naleznout i bez toho, abychom mu museli zadat explicitní cestu. Nevýhodou je, že verze knihoven v systémových balíčcích typicky bývají zastaralé.

  [1]Pokud by vás zajímalo, které všechny soubory a kam se nainstalovaly, můžete po instalaci balíčků použít příkaz

  ```
  $ dpkg -L libsdl2-dev
  ```

- **Manuální stažení knihovny**: Knihovnu si můžete také stáhnout manuálně, např. z webu SDL . Některé knihovny můžete naleznout na internetu už přeložené, nicméně SDL oficiálně pro Linux přeložené knihovní soubory (.so) nenabízí. V tomto případě tak musíte knihovnu nejenom stáhnout, ale také ručně přeložit, než ji budete moct použít ve svém programu.

Přilinkování knihovny SDL

Pokud jste nainstalovali SDL pomocí systémových balíčků, stačí při překladu programu přilinkovat knihovnu SDL2:

```
$ gcc main.c -lSDL2
```

Pokud jste knihovnu překládali manuálně, musíte ještě použít parametry -I pro předání cesty k hlavičkovým

souborům a `-L` pro předání cesty k adresáři s přeloženou knihovnou, jak jsme si vysvětlovali zde .

Pro práci s obrázky bude dále nutné přilinkovat knihovnu `SDL2_image` a pro práci s textem knihovnu `SDL2_ttf`.

Dokumentace

Abyste mohli používat nějakou složitější knihovnu, je nutné se zorientovat v její dokumentaci. V té naleznete jednak deklarace a popis fungování jednotlivých funkcí, které knihovna nabízí, ale také různé návody pro to, jak s knihovnou pracovat.

Dokumentaci funkcí `SDL` naleznete zde , návody pro jeho použití například tady .

> `SDL` je relativně rozsáhlá knihovna a není v silách tohoto textu, abychom ji plně popsali. Proto níže naleznete pouze velmi stručný "Hello world" a seznam věcí, které vám SDL umožňuje. Zbytek najdete v dokumentaci a návodech na internetu.

`SDL` hello world

Abychom něco vykreslili, tak jako první věc musíme nainicializovat SDL a vytvořit okno[2]:

[2]Pro zpřehlednění kódu bude v ukázkách níže vynechána kontrola chyb. Celý program i s kontrolou chyb naleznete na konci této sekce.

```
#include <SDL2/SDL.h>

int main()
{
    SDL_Init(SDL_INIT_VIDEO);   // Inicializace SDL

    // Vytvoření okna
    SDL_Window* window = SDL_CreateWindow(
        "SDL experiments",  // Název
        100,                // Souřadnice x
        100,                // Souřadnice y
        800,                // Šířka
        600,                // Výška
        SDL_WINDOW_SHOWN
    );
```

Jakmile máme otevřené okno, můžeme do něj něco začít vykreslovat. K tomu musíme nejprve vytvořit `SDL_Renderer`, neboli kreslítko:

```
    // Vytvoření kreslítka
    SDL_Renderer* renderer = SDL_CreateRenderer(
        window,
        -1,
        SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC
    );
```

S kreslítkem už můžeme něco nakreslit na obrazovku. Musíme vytvořit tzv. **herní smyčku** (*game loop*), která se bude provádět neustále dokola. Ve smyčce nejprve získáme události, které nastaly (např. došlo ke stisknutí klávesy nebo

pohybu myši), poté je zpracujeme, vykreslíme nový obsah okna a odešleme jej k vykreslení (za použití tzv. **double bufferingu**).

Konkrétně budeme vykreslovat jednoduchou posouvající se čáru, dokud uživatel nezavře otevřené okno:

```
    SDL_Event e;
    bool quit = false;
    int pos = 100;

    while (!quit)
    {
        while (SDL_PollEvent(&e))
        {
            if (e.type == SDL_QUIT)
            {
                quit = true;
            }
        }
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255); // Nastavení barvy na černou
        SDL_RenderClear(renderer);                      // Vykreslení pozadí

        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255); // Nastavení barvy na červenou
        SDL_RenderDrawLine(renderer, pos, pos, pos + 10, pos + 10); // Vykreslení čáry

        pos++;

        SDL_RenderPresent(renderer);  // Prezentace kreslítka
    }
```

A na konci už akorát vše uvolníme:

```
    // Uvolnění prostředků
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Pokud spustíte program využívající `SDL` s Address sanitizerem, může se stát, že vám sanitizer zobrazí nějakou neuvolněnou paměť. Pokud zdroj alokace nepochází z vašeho kódu, můžete tyto chyby ignorovat.

Celý kód i s ošetřením chyb

Co lze dělat pomocí `SDL`?

Knihovna `SDL` nabízí spoustu funkcionality k tvorbě interaktivních aplikací a her. Můžete s ní například:

- Vykreslovat body, čáry či obdélníky.
- Reprezentovat obdélníky a počítat jejich průniky (např. pro detekci kolizí herních objektů).
- Reagovat na vstup uživatele, ať už z klávesnice nebo z myši.
- Načítat a vykreslovat obrázky .
- Načítat a vykreslovat text.
- Přehrávat zvuk.

Chipmunk

Při tvorbě interaktivních grafických aplikací nebo her můžeme chtít simulovat pohyb objektů tak, aby dodržoval fyzikální zákony (působení gravitace, tření a kolize objektů, pohyb lana atd.). Pro to můžeme použít nějakou knihovnu na simulaci fyziky. `Chipmunk` je knihovna pro simulování jednoduchých fyzikálních procesů ve 2D prostoru. Zde se můžete podívat, co všechno se s takovou knihovnou dá udělat.

Pokud znáte hry jako Angry Birds nebo Fruit Ninja, tak tyto hry by se bez knihovny pro simulaci fyziky neobešly.

Instalace

Knihovna Chipmunk nenabízí distribuce již přeložených objektových souborů, musíme tedy její zdrojové soubory přidat k našemu projektu a přeložit je ručně.

Stáhněte si poslední verzi zdrojových kódů knihovny z webu Chipmunku, rozbalte je a výslednou složku přejmenujte z `Chipmunk-X.Y.Z` na `Chipmunk`.

Dále můžete knihovnu přidat ke svému `CMake` projektu pomocí následující `CMakeLists.txt` souboru:

Ukázkový CMakeLists.txt soubor pro Chipmunk

`Chipmunk` hello world

Stejně jako u `SDL` není v silách tohoto textu poskytnout kompletního průvodce touto knihovnou. Pro to můžete použít manuál nebo podrobnou dokumentaci funkcí.

Zde je okomentovaná ukázka "hello-world" příkladu, který simuluje pád sady kostek a vykresluje je pomocí SDL:

Okomentovaný program využívající knihovny Chipmunk a SDL

Ukázka fungování programu:

Tento program spolu s `CMakeLists.txt` souborem a knihovnou Chipmunk si můžete stáhnout zde. Přeložit a spustit ho můžete pomocí následujících příkazů:

```
$ mkdir build
$ cd build
$ cmake ..
$ make -j
$ cd ..
$ ./build/physics
```

Co dál?

C je relativně malý jazyk, pokud jste si tedy přečetli předchozí část tohoto textu, tak znáte většinu důležitých konstrukcí, která jsou v *C* dostupné. Nicméně neukázali jsme si úplně všechny – zde je seznam několika vybraných věcí, které byly buď moc pokročilé pro UPR anebo jsme je jednoduše nepotřebovali použít:

- **Variadiacké funkce**, které umožňují přijímat libovolný počet parametrů (takto funguje například i nám

známá funkce `printf`).
- **Ukazatele na funkce** (*function pointers*), které umožňují ukládat adresy funkcí do ukazatelů.
- **Enumerace** (*enumerations*), které umožňují seskupit pojmenované konstanty.
- **Sjednocené struktury** (*unions*), které umožňují interpretovat strukturu jako více různých datových typů.
- **Bitová pole** (*bit fields*), která umožňují rozdělit paměť struktury na úrovni jednotlivých bitů.
- **Široké znaky** (*wide chars*) a s nimi související funkce standardní knihovny umožňují používat složitější kódování než ASCII.
- **Komplexní čísla** (*complex numbers*) vám umožní pracovat s datovými typy reprezentujícími komplexní čísla.

Pokud si chcete ověřit, jak jste na tom se znalostí jazyka *C*, projděte si tyto slidy. Pokud budete umět odpovídat jako blonďatý kluk, tak znáte základy jazyka *C*. Pokud budete umět odpovídat jako dívka s růžovými vlasy, tak už vás v jazyce *C* téměř nic nepřekvapí.

Co se dále naučit

Se znalostí samotného jazyka *C* souvisí i spousta dalších konceptů, se kterými se postupně musíte seznámit, pokud chcete opravdu dopodrobna pochopit, co přesně se v počítači děje, když spustíte vámi napsaný program. Poté můžete těchto znalostí využít k tvorbě robustnějších a rychlejších programů. Na následujících odkazech se můžete dozvědět například:

- Jak fungují operační systémy.
  - Nebo dokonce jak si nějaký napsat od nuly .
- Jak komunikovat s jinými programi po síti.
- Jak psát programy pomocí instrukcí procesoru
- Jak urychlit provádění programů:
  - Pomocí vláken , které umí využít potenciál vícejádrových procesorů.
  - Pomocí vektorových instrukcí , které umí pracovat s více než jednou hodnotou najednou.
  - Pomocí pochopení architektury procesoru , která silně ovlivňuje výkon programů.
- Jak si napsat vlastní překladač či programovací jazyk .
- Jak si napsat vlastní databázi .
- Jak funguje počítačová grafika.
- Jak si napsat vlastní 3D herní engine pomocí OpenGL.
- Jak si napsat program pro nějaké vestavěné (*embedded*) zařízení, například Arduino.

Různé

Tato sekce obsahuje různá témata a návody, které nezapadají do zbytku textu, ale je dobré o nich vědět.

Rozklad problému

Často se setkáte s tím, že dostanete k naprogramování úlohu, se kterou si nevíte rady a netušíte ani jak začít. Například:

```
Načti obrázek z disku, změň jeho velikost, ulož ho do jiného souboru a vykresli jej na obrazovku.
```

Tato úloha vypadá velmi jednoduše, když je zadaná větou (v češtině), ale obzvláště pro začínající programátory je obtížné převést takovouto úlohu do programovacího jazyka. Obecným pravidlem k usnadnění řešení složitých úloh je rozdělovat je na menší a jednodušší podúlohy tak dlouho, dokud se nedostaneme k podúloze, kterou již umíme vyřešit. Poté z těchto malých kousků, které máme vyřešené, zpětně poskládáme celý program, který vyřeší původní úlohu.

Například zmíněnou úlohu můžeme rozdělit na následující podúlohy:

- Načti obrázek z disku
  - Otevři soubor se vstupním obrázkem
  - Načti hlavičku obrázku
  - Vytvoř paměť pro pixely obrázku
    - Naalokuj dostatek paměti dle hlavičky (šířka x výška)
- Změň velikost obrázku
  - Vytvoř obrázek s novým rozměrem
  - Překopíruj původní obrázek do nového obrázku
    - Projdi všechny pixely nového obrázku
      - Projdi každý řádek
      - Pro každý řádek projdi každý sloupec
    - Pro každý pixel spočítej původní pozici pixelu
      - Pro výpočet použij poměr šířky/výšky nového/starého obrázku
    - Překopíruj pixel ze starého obrázku do nového
  - Vrať nový obrázek
- Zapiš upravený obrázek
  - Otevři soubor k zápisu
  - Zapiš hlavičku obrázku do souboru
  - Zapiš pixely obrázku do souboru
- Vykresli upravený obrázek
  - Vytvoř okno pro vykreslení obrázku
  - Překopíruj pixely obrázku do otevřeného okna
  - Zobraz okno s obrázkem

Pomocí tohoto univerzálního postupu se dříve či později dostanete k (pod)úloze, kterou byste již měli umět vyřešit (např. otevření souboru). Jakmile danou podúlohu vyřešíte, tak budete o krok blíže k řešení původní složité úlohy.

Tímto způsobem můžeme programy rovnou od začátku začít psát. Například při řešení výše zmíněné úlohy můžeme začít nadefinováním hlavní logiky programu pomocí volání funkcí, kde každá funkce bude reprezentovat jednu podúlohu. I když funkce zatím nebudou naprogramované a později se třeba trochu změní, tak nám toto rozdělení může pomoct přemýšlet nad problémem abstraktněji, zorientovat se v něm a také získat naději, že se úlohu podaří vyřešit. Stejný princip opět můžeme použít při implementaci jednotlivých funkcí. Program (či funkci) pak lze přečíst jako větu a je tak jednodušší pochopit, co má vlastně dělat.

```
int main() {
    // Načti obrázek
    FILE* vstupni_soubor = otevri_soubor(...);
    Img obrazek = nacti_obrazek(vstupni_soubor);

    // Uprav jeho velikost
    Img upraveny_obrazek = uprav_velikost_obrazku(&obrazek);

    // Zapiš obrázek
    FILE* vystupni_soubor = otevri_soubor(...);
    zapis_obrazek(vystupni_soubor, &upraveny_obrazek);
```

```
    // Vykresli obrázek
    vykresli_obrazek(&upraveny_obrazek);

    return 0;
}
```

Generování náhodných čísel

Počítače jsou **deterministické** stroje, což znamená, že stejný program vždy na stejný vstup vrátí stejný výstup. Často ovšem chceme, aby naše programy obsahovaly prvky "náhody", když chceme například:

- Hodit si kostkou v deskové hře
- Udělit náhodný počet zranění v rozsahu zbraně
- Oživit hráče na náhodné pozici na mapě

Počítače samy o sobě opravdovou náhodu vytvořit nemohou, nicméně můžou ji simulovat pomocí tzv. **pseudo-náhodných generátorů čísel** (*pseudo-random number generation*).

Vygenerovat (pseudo-)náhodnou sekvenci čísel pomocí deterministických operací můžeme například následujícím algoritmem:

1. Začneme s číslem `S`, které se nazývá **počáteční náhodná hodnota** (*random seed*).
2. Aplikujeme nějakou matematickou operaci na `S` a vyjde nám nové číslo `N`.
3. `N` použijeme jako vygenerované "náhodné číslo".
4. Nastavíme `S = N`.
5. Opakujeme postup od bodu 2).

Ukázka kódu, který takovýto algoritmus implementuje:

```
int S = 5;
int vygeneruj_cislo() {
    int N = S;
    N = (5 * N + 3) % 6323;
    N = (4 * N + 2) % 8127;
    S = N;
    return N;
}
int main() {
    int r1 = vygeneruj_cislo(); // 114
    int r2 = vygeneruj_cislo(); // 2294
    int r3 = vygeneruj_cislo(); // 4348
    int r4 = vygeneruj_cislo(); // 2971
    int r5 = vygeneruj_cislo(); // 723
    return 0;
}
```

Takovýto algoritmus bude generovat (nekonečnou) sekvenci čísel, která bude lidem připadat "náhodná" (bude těžké uhodnout, jaké číslo algoritmus vrátí příště).

Volba počáteční hodnoty `S`

Určite jste si všimli, že výše zmíněný algoritmus bude pokaždé generovat stejnou sekvenci čísel pro stejné počáteční `S`. To se může hodit, chceme-li například mít možnost zpětně přehrát sekvenci pseudo-náhodných čísel, například pro

odladění chyby v programu. Nicméně pokud by sekvence byla pokaždé stejná, tak o (pseudo-)náhodě nemůže být řeč.

Proto se obvykle hodnota *seedu* volí tak, aby při každém spuštění programu byla jiná. Přirozenou volbou pro počáteční hodnotu `S` je tak například čas[1] při spuštění programu. Lze ale také použít například pohyby myši nebo stisky kláves, které nedávno na počítači proběhly.

[1]Ve formě UNIX časového razítka, tedy počtu vteřin uběhlých od 1. 1. 1970.

Pseudo-náhodný generátor ve standardní knihovně *C*

Při praktickém použití si obvykle nebudete psát generátor pseudo-náhodných sami, ale použijete již hotové řešení. To nabízí například standardní knihovna *C* ve formě funkcí `srand` (nastav hodnotu *seed* u) a `rand` (vygeneruj pseudo-náhodné číslo):

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

int main() {
    int cas = (int) time(NULL); // získej současný čas
    srand(cas); // nastav S na současný čas

    int cislo1 = rand(); // pseudo-náhodné číslo z intervalu [0, RAND_MAX]
    int cislo2 = rand() % 100; // z intervalu [0, 99]
    int cislo3 = rand() % 100 + 5; // z intervalu [5, 104]
    float cislo4 = rand() / (float) RAND_MAX; // z intervalu [0.0, 1.0]

    return 0;
}
```

Funkce `main`

Funkce `main` je speciální funkce, která se začne vykonávat při spuštění programu. Může vypadat například takto:

```
int main() {
    return 0;
}
```

Proč tato funkce vrací číslo (`int`) a proč jsme v dosavadních programech z ní vždy vraceli hodnotu `0`? Operační systémy mají zavedenou konvenci, že každý spuštěný program by měl po svém vykonání vrátit číselnou hodnotu, která systému napoví, jestli program proběhl úspěšně nebo ne. Díky tomu pak lze relativně jednoduše detekovat, jestli v programu nastala chyba, a případně na ni nějak zareagovat (na Windows možná znáte dialog "Program neproběhl správně...").

Číslo, které vrátíte z funkce `main`, se použije právě jako návratová hodnota programu pro operační systém. Význam navrácených čísel není nijak standardizován, jediné, co platí obecně, je, že hodnota `0` značí úspěch a jakákoliv jiná hodnota značí neúspěch. Proto tedy za normálních okolností z `main`u vracíme `0`, abychom dali systému najevo, že program proběhl úspěšně.

Vstupní parametry funkce `main`

Funkce `main` je speciální ve více ohledech. Kromě formy bez parametrů, kterou jste viděli výše, můžete `main` použít také takto, s dvěma parametry:

```
int main(int argc, char** argv) {
    return 0;
}
```

První parametr je typu `int` a druhý parametr typu ukazatel na řetězec. Do těchto parametrů se uloží hodnoty zadané při spuštění programu v terminálu, tzv. **argumenty příkazového řádku** (*command line arguments*). Parametr `argc` (*argument count*) bude obsahovat počet předaných argumentů a parametr `argv` obsahuje ukazatel na první prvek pole *C* řetězců, kde každý řetězec bude obsahovat jeden argument. Prvním argumentem je dle konvence vždy cesta k spustitelnému souboru programu, který je právě spouštěn, další argumenty se nastaví podle zadaného textu v terminálu (argumenty jsou oddělené mezerou).

Například, pokud program spustíte takto: `./program hello world`, tak parametry funkce `main` budou mít následující hodnoty:

- `argc` bude obsahovat celé číslo `3`
- `argv[0]` bude obsahovat řetězec `"./program"`
- `argv[1]` bude obsahovat řetězec `"hello"`
- `argv[2]` bude obsahovat řetězec `"world"`

Parametry překladače

Překladač `gcc` obsahuje sadu několika stovek parametrů, pomocí kterých můžeme ovlivnit, jak překlad programu proběhne. Můžeme například určit, pro jaký procesor se mají vygenerovat instrukce, jakou variantu jazyka *C* má překladač očekávat nebo jestli má náš program zoptimalizovat, aby běžel rychleji.

> Kromě `gcc` existuje řada dalších překladačů *C*, například `clang`. Nejčastější parametry (jako je např. `-O`) obvykle fungují ve všech překladačích obdobně, každý překladač ale obsahuje sadu specifických parametrů, které můžete naleznout v jeho dokumentaci.

Seznam všech parametrů můžete naleznout v dokumentaci `gcc`, zde je uveden seznam nejužitečnějších parametrů:

- **Optimalizace**: Existuje spousta parametrů, pomocí kterých můžete ovlivnit, jak překladač převede váš zdrojový kód na strojové instrukce a jak je zoptimalizuje. Nejzákladnějším parametrem je `-O`:
  - `-O0` - nebudou použity téměř žádné optimalizace. Toto je implicitní nastavení, pokud ho nezměníte. Program v tomto stavu lze dobře krokovat, ale může být dost pomalý.
  - `-O1` - aplikuje základní optimalizace.
  - `-O2` - aplikuje nejužitečnější optimalizace. Pokud chcete získat rozumně rychlý program, doporučujeme použít tento mód. Díky němu může být program třeba až 1000x rychlejší než s `-O0`.[1]

    [1] Anebo nemusí být rychlejší vůbec, záleží na programu.
  - `-O3` - aplikuje ještě více optimalizací. Program tak může být ještě rychlejší než s `-O2`. Obecně při použití optimalizací však platí, že čím vyšší optimalizační stupeň, tím více hrozí, že se váš program přestane

chovat správně, pokud program obsahuje jakékoliv nedefinované chování . Je tak třeba dávat pozor na to, aby k tomu nedošlo.

Kromě parametru `-O` lze použít spousty dalších parametrů, které ovlivňují například použití vektorových instrukcí. O těch se dozvíte více například v předmětu Programování v C++ .

- **Ladění programu**: Jak už jste jistě poznali, při použití jazyka *C* je velmi jednoduché způsobit nějaké nedefinované chování, například nějakou paměťovou chybou. Aby šlo tyto chyby detekovat, obsahují překladače tzv. *sanitizery*. Při použití sanitizeru se do vašeho programu přidají dodatečné instrukce, které poté při běhu programu kontrolují, jestli nedochází k nějakému problému. Cenou za tuto kontrolu je pomalejší běh programu (cca 2-5x). Sanitizery tak raději používejte pouze při vývoji programu.

  Existuje více typů sanitizerů, my si ukážeme dva:

  - `-fsanitize=address` - použije tzv. *Address Sanitizer*, který hlídá paměťové chyby, například přístup k nevalidní paměti nebo neuvolnění dynamické paměti. Tento sanitizer je nesmírně užitečný a doporučujeme ho používat při vývoji automaticky.
  - `-fsanitize=undefined` - použije tzv. *Undefined behaviour sanitizer*, který hlídá dodatečné situace, při kterých může dojít k nedefinovanému chování (kromě paměťových chyb).

  Obecně při ladění programu je taky vhodné vždy použít přepínač `-g`. Ten způsobí, že překladač přidá do výsledného spustitelného souboru informace o zdrojovém kódu (ty jinak ve spustitelném souboru chybí). Díky tomu budou sanitizery schopny zobrazit konkrétní řádek, na kterém vznikl nějaký problém a také půjde program ladit a krokovat.

- **Analýza kódu**: Kromě sanitizerů, které kontrolují váš program za běhu, lze také spoustu chyb odhalit již při překladu programu. Bohužel překladač `gcc` v implicitním módu není moc striktní a některé vyloženě chybné situace vám promine a program přeloží, i když je již dopředu jasné, že při běhu pak dojde např. k pádu programu. Abychom tomu předešli, můžeme zapnout při překladu dodatečná **varování** (*warnings*), která nás mohou na potenciálně problematické situace upozornit:

  - `-Wall` - zapne sadu několika desítek základních varování.
  - `-Wextra` - zapne dodatečnou sadu varování.
  - `-pedantic` - zapne striktní kontrolu toho, že dodržujete předepsaný standard *C*. V kombinaci s tímto přepínačem byste také měli explicitně říct, který standard chcete použít. V UPR používáme standard *C99*, který lze zadat pomocí `-std=c99`.
  - `-Werror` - tento přepínač způsobí, že libovolné varování bude vnímáno jako chyba. Pokud tak v programu `gcc` nalezne jakékoliv varování, program se nepřeloží.

  Pokud chcete mít při překladu co největší zpětnou vazbu od překladače a zajistit co největší "bezpečnost" vašeho programu, doporučujeme používat tuto kombinaci přepínačů:

```
$ gcc -g -fsanitize=address -Wall -Wextra -pedantic -std=c99
```

Úlohy

V této sekci naleznete různé úlohy, které si můžete zkusit naimplementovat, abyste se zlepšili v programování.

Další úlohy můžete najít také například na těchto odkazech:

- Advent of Code
- Project Euler
- Online Judge
- W3 C programming exercises

Základy

Obvod a obsah obdélníku

Program vypočítá a vypíše obvod a obsah obdélníku ze dvou celočíselných velikosti stran *a*, *b* podle známých vzorců.

a b

$$o = 2 \quad (a + b)$$
$$S = a \quad b$$

Výstup

Obsah vyšrafované plochy

Ze zadané délky strany čtverce *a* a průměru kružnice *d* vypočítáme obsah vyšrafované plochy. Výpočet budeme provádět pomocí datového typu `float` s využitím konstanty `M_PI` z knihovny `<math.h>`. Druhou mocninu vypočítáme násobením, ale také pomocí funkce `pow` . Při použití matematických funkcí je nutné program linkovat s knihovnou `math`[10]. Výsledek zapíšeme na výstup na 4 desetinná místa.

[10]Pro překlad s touto knihovnou použijte `-lm`:

```
$ gcc obsah.c -o obsah -lm
```

a a r

$$S_{kruh} = \frac{\pi \quad d^2}{4}$$

Výstup

Prohození dvou čísel

Pomocí dočasné proměnné provedeme prohození čísel ve dvou proměnných.

Výstup

Maximum ze tří čísel

Ze tří čísel nalezneme maximum.

Výstup

Výpis sudých čísel

Vypište sudá čísla od 0 do 100 (včetně).

FizzBuzz

Naimplementujte FizzBuzz [1]. Vypište čísla 1 až 100 tak, že:

[1]Tento program často bývá obsahem interview programátorů ve firmách.

- pokud je číslo násobkem 3, tak vypište místo čísla `Fizz`
- pokud je číslo násobkem 5, tak vypište místo čísla `Buzz`
- pokud je číslo násobkem 3 i násobkem 5, tak vypíše místo čísla `FizzBuzz`

Výstup programu

Složitější varianta: Naimplementujte tento program bez použití podmínek. Nesimulujte ani podmínku žádným cyklem. Použijte jediný cyklus `for` pro průchod čísly 1 až 100 a uvnitř tohoto cyklu nepoužijte žádnou podmínku.

Fibonacciho číslo

Napište funkci, která vypočte `n`-té Fibonacciho číslo (`n` bude parametrem funkce).

Výstup funkce

Faktoriál

Napište funkci, která vypočte faktoriál předaného parametru.

Výstup funkce

Textové kreslení obrazců

Vykreslete následující obrazce. Napište program tak, aby počet řádků, na který se obrazec vykreslí, byl konfigurovatelný, tj. pro změnu počtu řádků by mělo stačit změnit jediný řádek (jedinou proměnnou).

Vyplněný čtverec
Nevyplněný čtverec
Čtverec vyplněný rostoucími čísly
Diagonála
Trojúhelník

Písmeno Z

Načítání PINu

Načtěte od uživatele PIN (4 číslice). Poté opakovaně vyzývejte uživatele k zadání PINu. Pokud uživatel zadá 3x nesprávný PIN, vypište chybovou hlášku a ukončete program.

Ukazatele

Nastavení maxima

Vytvořte funkci `set_max`, která přijme adresu celého čísla (`int`) pomocí ukazatele a dvě další čísla a nastaví paměť na dané adrese na větší ze dvou zadaných čísel.

```
int res;
set_max(&res, 5, 6);
// res == 6
```

Prohození hodnoty

Vytvořte funkci `swap`, která přijme dva ukazatele a prohodí hodnoty proměnných, na které ukazují.

```
int a = 5, b = 6;
swap(&a, &b);
// a == 6, b == 5
```

Výpočet kořenů kvadratické rovnice

Vytvořte funkci `quadratic_roots`, která vrátí počet kořenů kvadratické rovnice $ax^2 + bx + c = 0$ pomocí `return` a vypočítané kořeny vrátí pomocí předaných ukazatelů v argumentech funkce.

```
int quadratic_roots(float a, float b, float c, float *x1, float *x2);
```

Počet kořenů lze zjistit vypočítáním diskriminantu $D = b^2 - 4ac$. Pokud vyjde diskriminant záporný, tak funkce vrátí nulu, protože žádné řešení v R neexistuje. Pro nulový diskriminant funkce vrátí `1` a uloží dvojnásobný kořen na adresu ukazatelů `x1`, `x2`. Pro kladný diskriminant funkce vrátí `2` a vypočítá kořeny pomocí:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

Pole

Naplnění pole

Vytvořte funkci `fill_array`, která naplní pole `array` čísly zvětšujícími se po `increment` a začínajícími od `start`.

```
void fill_array(int* array, int len, int start, int increment);
```

Počítání výskytů čísla

Vytvořte funkci `num_count`, která spočítá a vrátí počet výskytů čísla `num` v poli `array`.

```
int num_count(int* array, int len, int num);
```

Počítání čísel v intervalu

Vytvořte funkci `in_interval`, která spočítá počet čísel z uzavřeného intervalu `[from, to]` v poli `array`.

```
int in_interval(int* array, int len, int from, int to);
```

Průměrná hodnota

Vytvořte funkci `average`, která spočítá průměr čísel v poli `array`.

```
double average(int* array, int len);
```

Při dělení nezapomeňte přetypovat alespoň jeden operand na typ `double`, aby nedošlo k celočíselnému dělení.

Minimální hodnota v poli

Vytvořte funkci, která v poli `array` nalezne minimální hodnotu.

```
int array_min(int *array, int len);
```

Následně funkci upravte, aby funkce vrátila pomocí ukazatele první index s minimální hodnotou.

```
int array_min(int *array, int len, int *min_index);
```

Minimální a maximální hodnota

Předchozí funkci upravte, aby hledala minimum a maximum zároveň. Nalezené extrémy vraťte pomocí ukazatelů `min` a `max`.

```
void min_max(int* array, int len, int *min, int *max);
```

Ve funkci si nejprve nastavte index minimální a maximální hodnoty na nultý prvek. Parametr `min` je ukazatel, a je tedy nutné přistupovat k jeho hodnotě pomoci dereference - `*min`, protože výraz `min` obsahuje pouze adresu, kde je minimální index uložen. Následně projděte pole a pokud bude hodnota aktuálního prvku menší než hodnota prvku na dosud nalezeném indexu, nastavte hodnotu minimálního indexu na aktuální index. Stejný postup aplikujte i pro

nalezení maximálního prvku (stačí udělat jeden průchod polem).

Obrácení pole

Vytvořte funkci `array_reverse`, která obrátí prvky v poli.

```
void array_reverse(int* array, int len);
```

Pole projděte pomoci cyklu do jeho půlky a vždy prohazujte prvky z obou konců.

Přehození dvou prvků nemůžete udělat najednou. Uložte si například prvek z levého konce do proměnné a následně do tohoto prvku zapište hodnotu z pravého konce. Poté hodnotu z proměnné uložte do pravého konce. Alternativně také můžete využít dříve naimplementovanou funkci `void swap(int* a, int* b)`.

Skalární součin

Vytvořte funkci `dot`, která spočítá skalární součin.

```
int dot(int* a, int* b, int len);
```

Načtení dynamického počtu hodnot

Načtěte od uživatele číslo `n`. Poté naalokujte paměť o velikosti `n` `int`ů a načtěte ze vstupu `n` čísel, které postupně uložte do vytvořeného pole. Vypište součet načteného pole.

Counting sort

Vygenerujte pole 10 000 000 náhodných čísel z intervalu $\langle 1000, 2000 \rangle$. Pomocí algoritmu counting sort seřaďte čísla v poli od nejmenšího po největší.

1. vytvořte pole počítadel pro všechny možné hodnoty v poli
2. vynulujte počitadla na 0
3. sekvenčně projděte pole čísel a inkrementujte odpovídající počítadlo
4. projděte pole počítadel a tiskněte hodnotu tolikrát, kolik je hodnota počítadla

Třízení

Naimplementujte funkci, která setřídí pole. Můžete použít například algoritmus bubble sort.

Střelba na terč

Vytvořte program, který načte souřadnice terčů a střel, a vykreslí je do obrázku ve formátu vektorové grafiky SVG. Po najetí myší na terč by se mělo ukázat skóre vybraného terče.

Ze vstupu přečtěte počet terčů a následně si dynamicky alokujte 3 pole typu `float` pro `x` souřadnice terčů, `y` souřadnice terčů a poloměry terčů.

Poté pro každý terč přečtěte jeho `x` souřadnici, `y` souřadnici, poloměr a uložte je do odpovídajících polí. Například následující vstup nám popisuje 2 terče. První terč má střed na souřadnici [50, 70] a poloměr 40 a druhý terč leží na středu [160, 90] s poloměrem 60.

```
2
50 70 40
160 90 60
```

Tento vstup nezadávejte pořad dokola z klávesnice, ale přesměrujte si jej do programu ze souboru:

```
$ ./main < terce.txt
```

Terče si pomocí `printf` vykreslete do vektorového obrázku ve formátu svg, ve kterém lze pomocí tagů definovat útvary. Útvary v obrázku obalte tagem `svg`:

```
<svg xmlns='http://www.w3.org/2000/svg'>
  <!-- kresleni kruhu -->
</svg>
```

Terč se středem [50, 70] a poloměrem 40 lze vykreslit pomocí:

```
<circle cx='50' cy='70' r='40' stroke='black' fill='red' />
```

Vytvořený SVG obrázek si ze standardního výstupu přesměrujte do souboru a otevřete si jej například v prohlížeči firefox.

```
$ ./main < terce.txt > obrazek.svg
$ firefox obrazek.svg
```

Následně si ze vstupu přečtěte počet střel a alokujte pro ně dvě pole - jedno bude reprezentovat `x` souřadnice a druhé `y` souřadnice jednotlivých střel. Souřadnice si následně přečtěte do těchto polí. Pole si projděte a vykreslete do obrázku jako kruhy např. s poloměrem 4.

Střela zasáhla terč, pokud leží na kruhu. Jinými slovy - střela zasáhla terč, pokud je vzdálenost od středu terče menší než poloměr terče. Vzdálenost vypočítáme jednoduše pomocí Pythagorovy věty, kde `x` odvěsna je rozdíl mezi `x` souřadnici středu terče a `x` souřadnici střely. Odvěsna `y` lze vypočítat obdobně a poté můžeme vypočítat přeponu, která reprezentuje vzdálenost střely od středu terče.

dist

Protože máme více terčů a více střel, tak musíme aplikovat výpočet vzdálenosti mezi každou střelou a každým terčem pomocí dvou vnořených `for` cyklů. Vnější cyklus bude procházet střely a vnitřní cyklus bude procházet terče. Ve vnitřním cyklu vypočítáme vzdálenost mezi střelou a terčem a pokud je menší než poloměr, tak tento konkrétní terč

byl zasažen střelou z vnějšího cyklu. V případě, že se více kruhů překrývá, tak střela zasáhla terč s menším poloměrem. Budeme tedy hledat zasáhnutý terč s nejmenším poloměrem.

Skóre při zasažení středu s poloměrem 20 je 10 bodů a body postupně klesají. Zdrojový kód SVG ukázek si můžete zobrazit.

Dva terče
Překrývající se terče
Překrývající se terče se stejným středem

Dvourozměrné pole

Vytisknutí matice

Vytvořte funkci `print_matrix`, která vypíše obrázek reprezentovaný dvourozměrným (2D) polem.

```
void print_matrix(int* matrix, int rows, int cols);
```

Projděte matici po řádcích a sloupcích a vypište jednotlivé prvky.

Vykreslení hvězdice

Vytvořte funkci `draw_star`, která do 2D matice vykreslí hvězdici.

```
void draw_star(int* matrix, int rows, int cols);
```

```
X    X    X
 X   X   X
  X  X  X
   X X X
    XXX
XXXXXXXXXXX
    XXX
   X X X
  X  X  X
 X   X   X
X    X    X
```

Hvězdici můžete vykreslit do pole pomocí jediného cyklu. Zkuste vytvořit funkce na vykreslení dalších tvarů (čára, čtverec, kružnice, trojúhelník, ...).

Násobení matice skalárem

Vytvořte funkci `matrix_mul_scalar`, která vynásobí každý prvek matice číslem `k`.

```
void matrix_mul_scalar(int* matrix, int rows, int cols, int k);
```

Násobení matice vektorem

Vytvořte funkci `matrix_mul_vector`, která vynásobí matici vektorem.

```
int* matrix_mul_vec(int* matrix, int rows, int cols, int *vec, int len);
```

Násobení matice maticí

Vytvořte funkci pro násobení matice A o rozměrech $rows_1 \times cols_1$ s druhou maticí B o rozměrech $rows_2 \times cols_2$. Funkce vrátí `NULL` v případě, že matice nepůjdou vynásobit např. v případě, že počet řádků první matice není shodný s počtem sloupců druhé matice. Výslednou matici o rozměrech $rows_1 \times cols_1$ alokujte dynamicky.

Řetězce

Převod na velké znaky

Vytvořte funkci, která převede textový řetězec na velké znaky.

```
char str[] = { "hello" };
uppercase(str);
// str by se zde měl rovnat "HELLO"
```

Nahrazení znaku

Vytvořte funkci, která v řetězci nahradí všechny výskyty daného znaku za znak `'X'`.

```
char str[] = { "hello" };
replace(str, 'l');
// str by se zde měl rovnat "heXXo"
```

Šifrování řetězce

Vytvořte funkci, která "zašifruje" řetězec tím, že ke každému znaku přičte číslo (klíč). K ní vytvořte funkci, která řetězec opět odšifruje (odečtením klíče).

```
char str[] = { "abc" };
encrypt(str, 1);
// str by se zde měl rovnat "bcd"
decrypt(str, 1);
// str by se zde měl opět rovnat "abc"
```

Délka řetězce

Vytvořte funkci `my_strlen`, která vypočte délku řetězce (obdoba funkce `strlen` ze standardní knihovny *C*).

```
my_strlen("");          // 0
my_strlen("abc");       // 3
my_strlen("abc 0 asd"); // 9
```

Porovnávání řetězců

Vytvořte funkci, která vrátí `true`, pokud jsou dva předané řetězce stejné. Vytvořte i variantu funkce, která porovnává řetězce bez ohledu na velikosti znaků.

```
strequal("ahoj", "ahoj");               // 1
strequal("ahoj", "aho");                // 0
strequal_ignorecase("ahoj", "AhOj");    // 1
```

Palindrom

Vytvořte funkci, která vrátí `true`, pokud je předaný řetězec palindrom (slovo, které se čte stejně zepředu i pozpátku).

Histogram

Vytvořte funkci, která vypočte histogram znaků v řetězci. Histogram je pole, ve kterém prvek na pozici `x` udává, kolikrát se znak `x` vyskytoval v daném řetězci.

```
int histogram[255] = {};
calc_histogram("aabacc", histogram);
// histogram['a'] == 3
// histogram['b'] == 1
// histogram['c'] == 2
// histogram['d'] == 0
```

Převod textu na číslo

Vytvořte funkci, která převede řetězec na číslo v desítkové soustavě. Pokud číslo nelze převést, vraťte hodnotu `0`.

```
convert("5");   // vrátí int s hodnotou 5
convert("123"); // vrátí int s hodnotou 123
```

Zkuste přidat i podporu pro záporná čísla.

Struktury

Vytvořte strukturu `Student`, která bude obsahovat atributy pro jeho věk, jméno, počet bodů a nejlepšího přítele (to bude také student). Dále naimplementujte tyto funkce:

```
/**
 * Nainicializujte studenta se zadaným věkem a jménem.
 * Počet bodů i nejlepší přítel by měli být nastaveni na nulu.
 */
void student_init(Student* student, int age, const char* name) {}

/**
 * Spočítejte, kolik studentů v předaném poli má maximálně zadaný věk.
 * Příklad:
 *   Student students[3];
```

```
 *   students[0].age = 18;
 *   students[1].age = 19;
 *   students[2].age = 16;
 *
 *   count_young_students(students, 3, 18); // 2
 */
int count_young_students(Student* students, int count, int maximum_age) {}

/**
 * Přiřaďte studentům body na základě výsledků testů.
 * V poli `points` jsou body pro jednotlivé studenty v poli `students`.
 * Parameter `count` obsahuje počet studentů a testů.
 */
void assign_points(Student* students, const int* points, int count) {}

/**
 * Vraťe v parametru `good_students` pole studentů, kteří mají alespoň 51 bodů a v
 * parametru `good_student_count` jejich počet.
 * Budete muset dynamicky naalokovat nové pole s odpovídající velikostí.
 */
void filter_good_students(
    const Student* students,
    int count,
    Student** good_students,
    int* good_student_count
);

/**
 * Otestujte, jestli je student šťastný.
 * Student je šťastný, pokud:
 * 1) Má alespoň 51 bodů, a zároveň
 * 2) Jeho nejlepší přítel je šťastný
 *
 * Pokud student nemá nejlepšího přítele, pokládejte podmínku 2) za splněnou.
 */
int student_is_happy(Student* student) {}
```

K otestování vaší implementace můžete použít následující testovací program[1]:

[1]Implementace svých funkcí v tomto programu umístěte nad `main` a program spusťte s `Address sanitizerem`. Pokud program nic nevypíše, máte implementaci pravděpodobně správně.

Testovací program

Kreslení obrazovky Apple Watch

Pro tuto úlohu využijte stuktury a funkce pro zápis obrázku formátu TGA do souboru.

Vše má svůj příběh a tak tedy započněme naši cestu např. ve firmě Apple...

Představte si, že jste vývojářem/kou ve firmě Apple a Steve Jobs Vás pověří programátorským úkolem.

Firma aktuálně pracuje na super tajném projektu nových smart hodinek , které chce uvést na trh. Vašim úkolem je pod přímým vedením Steva Jobse (původního zakladatele firmy) naprogramovat digitální ciferník nových hodinek.

Technické specifikace displeje

Displej hodinek má rozlišení `368x448` px (pixelů).

Schéma pro zobrazení znaků

Na obrázku níže je rozklesleno, jak by se měl zobrazovat čas na hodinkách.

První řádek slouží pro zobrazeni hodin, druhý řádek pro zobrazení minut. Tloušťky jednotlivých segmentů a rozestupy jsou také zakresleny. Modře je znázorněna oblast, kde se nevykreslují číslice, ale je možno kreslit pozadí ciferníku. Jsou znázorněna jen čtyři čísla, zbytek si již odvodíte sami.

Funkce a struktury na implementaci

Postupně naimplementujte následující funkce a struktury.

Funkce pro vykreslení času

```
void watch_draw_time(TGAImage* self, const int hours, const int minutes);
```

Funkce nakreslí do obrázku `self` čas zadaný pomocí času v hodinách (`hours`) a minutách (`minutes`). Barvu čísel si zvolte libovolně, stejně jako barvu pozadí.

Struktura pro reprezentaci barvy pixelu (`RGBA`)

Barva se do každého pixelu zapisuje jako čtveřice bajtů `BGRA` (`B` - Blue, `G` - Green, `R` - Red, `A` - Alpha). Nadefinujte si strukturu `RGBA`, která bude tyto bajty reprezentovat pomocí čtyř proměnných: `r`, `g`, `b`, `a` patřičného datového typu.

Funkce pro vykreslení času s určením barev

```
void watch_draw_time_color(
    TGAImage* self,
    const int hours,
    const int minutes,
    const RGBA* fg_color,
    const RGBA* bg_color
);
```

Funkce nakreslí do obrázku `self` čas zadaný pomocí času v hodinách (`hours`) a minutách (`minutes`). Barva čísel je předána parametrem `fg_color`, barva pozadí pak parametrem `bg_color`.

Soubory

Spočítání řádků

Naimplementujte funkci, která načte soubor na zadané cestě a vrátí počet řádků, které se v něm vyskytují.

```
int count_lines(const char* path) {}
```

Kopírování souboru

Naimplementujte funkci, která přijme cestu ke vstupnímu a výstupnímu souboru a zkopíruje obsah vstupního souboru do výstupního souboru.

```
void copy_file(const char* src, const char* destination) {}
```

Šifrování souboru

Naimplementujte funkci, která přičte číslo `key` ke všem znakům v souboru na zadané cestě.

```
void encrypt_file(const char* path, int key) {}
```

Dále udělejte druhou funkci, která od znaků v souboru na zadané cestě naopak číslo `key` odečte. Otestujte, že soubor po zašifrování a odšifrování obsahuje stejný obsah. Pro testování používejte soubory s ASCII textem.

```
void decrypt_file(const char* path, int key) {}
```

Různé

Hádací hra (*guessing game*)

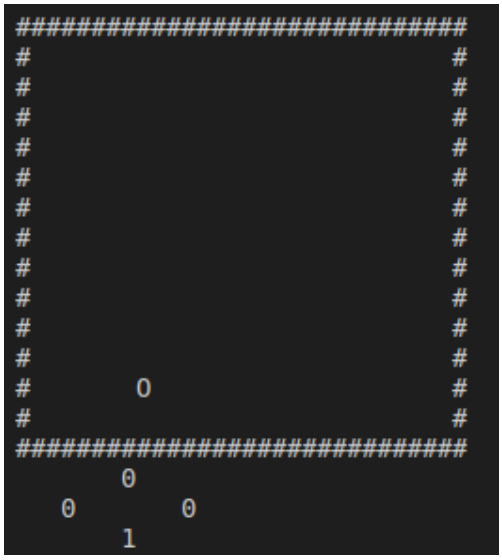
Vygenerujte náhodné číslo. Poté nechte uživatele hádat, jaké číslo program vygeneroval. Po každém tipu uživateli dejte vědět, jestli uhádl správně nebo jestli jeho tip byl vyšší či nižší než číslo, které hádá.

Odrážející se kulička v terminálu

Vykreslujte do terminálu obdélník spolu s pohybující se kuličkou. Jakmile kulička narazí do stěny čtverce, zvyšte počítadlo nárazů pro danou zeď. Dodržujte princip zákonu odrazu .

Přibližný postup řešení

Výsledek by měl vypadat zhruba takto:



Kalkulačka

Načtěte ze vstupu programu nebo z parametrů příkazového řádku matematický výraz, který bude obsahovat celá čísla a operátory `+`, `-`, `/`, `*` a vypište výsledek tohoto výrazu.

- *Varianta 1*: Použijte klasický zápis v infixové notaci. Nemusíte řešit prioritu operátorů.
- *Varianta 2*: Přidejte podporu pro prioritu operátorů a závorky `(`, `)`. Použijte algoritmus Shunting yard.
- *Varianta 3*: Použijte postfixovou notaci. Zde bude fungovat priorita operátorů a "závorkování" bez nutnosti složitého načítání vstupu z varianty 2.

Tvorba animace

Pomocí knihovny pro práci s GIF animacemi vytvořte nějakou zajímavou animaci. Například se zkuste přiblížit této animaci z Matrixu:

Časté chyby

V této sekci najdete často se vyskytující chyby, na které můžete narazit, spolu s návodem, jak je vyřešit.

Záměna `=` a `==`

- Operátor `=` přiřazuje hodnotu do svého levého operandu a vyhodnotí se s hodnotou pravého operandu.
- Operátor `==` porovnává dvě hodnoty a vyhodnotí se jako pravdivostní hodnota `bool`.

Je důležité tyto operátory nezaměňovat! Oba dva operátory jsou výrazy, takže se v něco vyhodnotí a i když je použijete špatně, tak často nedostanete chybovou hlášku, což jejich záměnu dělá ještě nebezpečnější.

```
int a = 0;
a = 5; // nastaví hodnotu `5` do proměnné `a`
a == 5; // porovná `a` s hodnotou `5`, vrátí hodnotu `true`, ale nic se neprovede

// podmínka se provede, pokud se `a` rovná `5`
if (a == 5) {}

// podmínka se provede vždy, výraz `a = 5` se vyhodnotí na `5` (`true`)
// zároveň při provedení podmínky se přepíše hodnota proměnné `a` na `5`
if (a = 5) {}
```

Záměna `&` s `&&` nebo `|` s `||`

- Operátor `&` provádí bitový součin, očekává jako operandy celá čísla (např. `int`) a vrací celé číslo.
- Operátor `&&` provádí logický součin, očekává jako operandy pravdivostní hodnoty (`bool`) a vrací pravdivostní hodnotu.

Je důležité tyto operátory nezaměňovat. Jelikož `bool` lze implicitně převést na celé číslo a naopak, záměna těchto

operátorů opět typicky nepovede k chybě při překladu, nicméně program nejspíše při jejich záměně nebude fungovat tak, jak má. Operátor `&` má zároveň větší přednost než `&&`, takže se výraz s tímto operátorem může vyhodnotit jinak, než očekáváte. Obdobná situace platí i u dvojice operátorů `|` (bitový součet) a `||` (logický součet).

```
int a = 3;
a & 4; // `0`
a && 4; // `true`

// stejné jako a > (5 & a) < 6
if (a > 5 & a < 6) {}
```

Středník za `for`, `while` nebo `if`

Příkazy `for`, `while` nebo `if` za svou uzavírací závorkou `)` očekávají jeden příkaz:

```
if (a > b) printf("%d", a);
```

nebo blok s příkazy:

```
if (a > b) {
    printf("%d", a);
    ...
}
```

Pokud však za závorku dáte rovnou středník (`;`), tak překladač to pochopí jako prázdný příkaz, který nic nedělá.

V následující ukázce se provede 10× prázdné tělo cyklu `for` a následně se jednou vypíše řetězec `"Hello\n"`.

```
#include <stdio.h>

int main() {
    for(int i = 0; i < 10; i++); {
        printf("Hello\n");
    }
    return 0;
}
```

Zde opět středník za `if` reprezentuje prázdný příkaz, takže blok kódu s příkazem `printf` se provede vždy, i když je tato podmínka nesplnitelná.

```
#include <stdio.h>

int main() {
    if(0); {
        printf("Hello\n");
    }

    return 0;
}
```

Je to ekvivalentní, jako byste napsali

```
#include <stdio.h>

int main() {
    if (0) { /* zde není co provést */ }

    // tento blok se provede vždy
```

```
    {
      printf("Hello\n");
    }

    return 0;
}
```

Špatné volání funkce

Abychom zavolali funkci (tj. řekli počítači, aby začal vykonávat kód, který v ní je), napíšeme název funkce, závorky a do nich případně seznam argumentů. Při volání funkce už nezadáváme její návratový typ, ten se udává pouze u definice funkce.

```
int secti(int a, int b) {
    return a + b;
}
int main() {
    secti(1, 2);        // správně
    int secti(1, 2);    // špatně

    return 0;
}
```

Záměna ' s "

- Apostrof (') slouží k zapsání (jednoho) znaku . Neukládejte do něj více znaků či celý text.
- Uvozovky (") slouží k zapsání řetězce, tj. pole znaků ukončeného hodnotou `0`.

```
char a = 'asd'; // špatně, více znaků v ''
char a = "asd"; // špatně, ukládáme řetězec do typu `char` (mělo by být `const char*`)

char a = 'x';               // správně
const char* str = "hello";  // správně
```

Špatná práce s ukazatelem

Ukazatele jsou čísla, která interpretujeme jako adresy v paměti. Můžete s nimi sice provádět některé aritmetické operace (například sčítání či odčítání), nicméně v takovém případě provádíte výpočet s adresou, ne s hodnotou, která je na dané adrese uložena.

Například v této funkci, která by měla přičíst hodnotu `x` k paměti na adrese `ptr`, musíte nejprve přistoupit k hodnotě na dané adrese (`*ptr`), a až k této hodnotě pak přičíst `x`:

```
void pricti_hodnotu(int* ptr, int x) {
    ptr += x;   // špatně, přičteme `x` k adrese `ptr`
    *ptr += x;  // správně, přičteme `x` k hodnotě na adrese `ptr`
}
```

Vytváření spousty proměnných místo použití pole

Pokud potřebujete jednotně pracovat s větším počtem hodnot v paměti, použijte pole . Signálem, že jste měli použít pole, může být to, že máte ve funkci spoustu proměnných a pro rozlišení každé proměnné musíte přidat nový řádek

kódu:

```
for (a0 = 0, a1 = 0, a2 = 0, a3 = 0, a4 = 0, a5 = 0; i < pocet; i++)
{
    if (hodnota == 1)
    {
        a0++;
    }
    else if (hodnota == 2)
    {
        a1++;
    }
    else if (hodnota == 3)
    {
        a2++;
    }
    ...
}
```

`undefined reference to 'NAZEV'`

Snažíte se zavolat funkci `NAZEV`, která nebyla nalezena v žádném objektovém souboru, který jste předali pro překlad. Ověřte si, že máte název volané funkce správně.

Paměťové chyby

V *C* lze s pamětí programu pracovat manuálně, což velmi často vede k různým paměťovým chybám, které můžou způsobit špatné chování či pád programu. Jsou také nejčastějším zdrojem různých zranitelností , které umožňují útočníkům převzít kontrolu nad programem nebo celým počítačem.

Pro částečnou prevenci paměťových chyb silně doporučujeme při vývoji *C* programů používat nástroj Address sanitizer.

Stack overflow

Pokud bychom vytvořili v zásobníkovém rámci moc proměnných, proměnné, které jsou moc velké , anebo bychom měli v jednu chvíli aktivních moc zásobníkových rámců (například při moc hluboké rekurzi), tak může dojít paměť určená pro zásobník. Tato situce se nazývá **přetečení zásobníku** (*stack overflow*):

```
int funkce(int x) {
    return funkce(x + 1);
}
int main() {
    funkce(0);
    return 0;
}
```

Segmentation fault

Tato chyba je způsobena pokusem o zapsání nebo čtení neplatné adresy v paměti. K této chybě často dochází zejména

při těchto situacích:

- Zapísujeme nebo čteme z paměti pole mimo jeho rozsah (tj. "před" nebo "za" pamětí pole). Tato situace se nazývá *buffer overflow*. Tato chyba už způsobila nespočet bezpečnostních chyb v různých softwarech.

```
#include <stdlib.h>

int main() {
    int* p = (int*) malloc(sizeof(int));
    p[1] = 5;
    return 0;
}
```

- Pokoušíme se přečíst hodnotu na adrese 0 (`NULL`), která je používána pro inicializaci ukazatelů. Tato situace se nazývá *null pointer dereference*.

```
int main() {
    int* p = (void*) 0;
    int a = *p;

    return 0;
}
```

- Snažíme se přistoupit k paměti, která již byla uvolněna. Tato situace se nazývá *use-after-free*.

```
#include <stdlib.h>

int main() {
    int* p = (int*) malloc(sizeof(int));
    free(p);

    *p = 1;
    return 0;
}
```

Přístup k již uvolněné paměti může nastat i bez použití dynamické paměti. Například tento kód není správně:

```
#include <stdlib.h>

int* vrat_ukazatel(int x) {
    int y = x + 1;
    return &y;
}

int main() {
    int* p = vrat_ukazatel(1);
    *p = 1;
    return 0;
}
```

Jakmile totiž vykonávání funkce `vrat_ukazatel` skončí, tak se uvolní paměť jejich lokálních proměnných.

Adresa uložená v `p` tak obsahuje nevalidní paměť a je chybou k ní přistupovat (ať už číst, tak zapisovat).

- Snažíme se uvolnit pamět, která již byla uvolněna. Tato situace se nazývá *double free*.

```
#include <stdlib.h>

int main() {
    int* p = (int*) malloc(sizeof(int));
    free(p);
    free(p);
    return 0;
}
```

Memory leak

Pokud (opakovaně) alokujeme dynamickou paměť a neuvolňujeme ji, tak dochází k tzv. *memory leaku* (úniku paměti). Pokud paměť programu stále roste a není nijak uvolňována, tak postupem času počítači nutně dojde paměť a program tak bude násilně ukončen.

```
void leak() {
    // adresa alokované paměti je zahozena, nelze ji tedy uvolnit
    malloc(sizeof(int));
}
```

Tato chyba je celkem zákeřná, protože pokud paměť roste pomalu, tak může trvat dost dlouho, než se projeví. K nalezení chyby doporučujeme použít Address sanitizer, který na konci programu zkontroluje, jestli všechny dynamicky naalokované bloky byly korektně uvolněny.

> Nemusíte se však bát, že by neuvolněná paměť ve vašem programu nějak narušovala chod operačního systému. I když paměť manuálně neuvolníte, tak moderní operační systémy veškerou paměť vašeho spuštěného programu uvolní, jakmile program skončí. Dokud však program běží, tak bude neuvolněná paměť zabírat místo, což může způsobovat problémy.