

Vysoká škola báňská – Technická univerzita Ostrava

ZÁKLADY PROGRAMOVÁNÍ ŘÍDICÍCH SYSTÉMŮ

učební text a návody do cvičení

Jindřich Černohorský, Jakub Jirka

Ostrava 2012



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



OP Vzdělávání
pro konkurenceschopnost



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Recenze: Jaromír Konečný

Název: ZÁKLADY PROGRAMOVÁNÍ ŘÍDICÍCH SYSTÉMŮ
Autor: Jindřich Černohorský, Jakub Jirka
Vydání: první, 2013
Počet stran: 203

Studijní materiály pro studijní obor Fakulty elektrotechniky a Informatiky
Jazyková korektura: nebyla provedena.

Určeno pro projekt:

Operační program Vzdělávání pro konkurenceschopnost

Název: Inovace oboru Měřicí a řídicí technika na FEI, VŠB - TU Ostrava

Číslo: CZ.1.07/2.2.00/15.0113

Realizace: VŠB – Technická univerzita Ostrava

Projekt je spolufinancován z prostředků ESF a státního rozpočtu ČR

© Jindřich Černohorský, Jakub Jirka

© VŠB – Technická univerzita Ostrava

OBSAH

OBSAH.....	2
POKYNY KE STUDIU.....	4
1. ÚVODNÍ KAPITOLA DO PROGRAMOVÁNÍ	1
1.1. Jak funguje počítač.....	1
1.2. Java Virtual Machine – Virtuální stroj Javy	17
2. PROGRAMOVACÍ JAZYK C	21
2.1. První program v jazyku C	21
2.2. Příklad matematického programu.....	24
2.3. Cykly.....	27
2.4. Přepínač switch	29
2.5. Symbolické konstanty.....	31
2.6. Podmíněné výrazy.....	32
2.7. Pointery neboli ukazatele	34
2.8. Pole	37
2.9. Pole znaků.....	39
2.10. Periferní (I/O==input/output) operace.....	41
2.11. Funkce	44
2.12. Uživatelské a strukturované datové typy	48
2.13. Uživatelské typy enum a union.....	53
3. SOUČASNÉ VYUŽITÍ PROGRAMOVACÍHO JAZYKA C V OBLASTI MIKROČIPŮ.....	75
3.1. Programování mikroprocesorů AVR.....	75
4. POKROČILÉ TECHNIKY PROGRAMOVÁNÍ V JAZYCE C.....	99
4.1. Ukazatelé a jejich speciální případy	99
4.2. Dynamická alokace paměti.....	103
4.3. Vlastní speciální často používané datové typy, List, LinkedList	106
5. JAZYK C, PROGRAMOVÁNÍ V PROSTŘEDÍ UNIX – LIKE OPERAČNÍCH SYSTÉMŮ.....	109
5.1. Jazyk C, a Unix-like operační systémy	109
5.2. GNU GCC	112
5.3. Adresářová struktura OS Linux	113
5.4. Vytváření knihoven v OS Linux	117

5.5. Procesy, paralelismus a Real Time v OS Linux.....	123
6. STANDARD C A SPECIÁLNÍ VARIANTY PROGRAMOVACÍHO JAZYKA C (C99, C11, OBJECTIVE-C).....	130
6.1. Stručná historie jazyka C, standard C99	130
6.2. Standard C11 (2011)	133
6.3. Objective-C.....	138
7. VARIACE JAZYKA C PRO PROGRAMOVÁNÍ GRAFICKÝCH ČIPŮ CG (C FOR GRAPHICS)	144
7.1. Jazyk Cg.....	144
7.2. Cuda (Compute Unified Device Architecture)	150
7.3. Psaní aplikací pro CUDA Cg	156

POKYNY KE STUDIU

Základy Programování Řídicích Systémů

Pro předmět moderní informační technologie pro řízení, 5. semestru oboru měřicí a řídicí technika jste obdrželi studijní balík obsahující

- integrované skriptum pro distanční studium obsahující i pokyny ke studiu
- CD-ROM s doplňkovými animacemi vybraných částí kapitol
- harmonogram průběhu semestru a rozvrh prezenční části
- rozdělení studentů do skupin k jednotlivým tutorům a kontakty na tutorý
- kontakt na studijní oddělení

Prerekvizity

Pro studium tohoto předmětu se nepředpokládá žádná předchozí znalost z oboru programování, ačkoliv je výhodou.

Cílem předmětu

Je seznámení se základními dovednostmi a nástroji programování.

Pro koho je předmět určen

Modul je zařazen do bakalářského studia oborů 1. ročníku studijního programu měřicí a řídicí technika a biomedicínský technik, ale může jej studovat i zájemce z kteréhokoliv jiného oboru.

Skriptum se dělí na části, kapitoly, které odpovídají logickému dělení studované látky, ale nejsou stejně obsáhlé. Předpokládaná doba ke studiu kapitoly se může výrazně lišit, proto jsou velké kapitoly děleny dále na číslované podkapitoly a těm odpovídá níže popsaná struktura.

Při studiu každé kapitoly doporučujeme následující postup:



Čas ke studiu: xx hodin

Na úvod kapitoly je uveden **čas** potřebný k prostudování látky. Čas je orientační a může vám sloužit jako hrubé vodítko pro rozvržení studia celého předmětu či kapitoly. Někomu se čas může zdát příliš dlouhý, někomu naopak. Jsou studenti, kteří se s touto problematikou ještě nikdy nesetkali a naopak takoví, kteří již v tomto oboru mají bohaté zkušenosti.



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat ...
- definovat ...

- vyřešit ...

Ihned potom jsou uvedeny cíle, kterých máte dosáhnout po prostudování této kapitoly – konkrétní dovednosti, znalosti.



Výklad

Následuje vlastní výklad studované látky, zavedení nových pojmů, jejich vysvětlení, vše doprovázeno obrázky, tabulkami, řešenými příklady, odkazy na animace.



Shrnutí pojmů

Na závěr kapitoly jsou zopakovány hlavní pojmy, které si v ní máte osvojit. Pokud některému z nich ještě nerozumíte, vraťte se k nim ještě jednou.



Otázky

Pro ověření, že jste dobře a úplně látku kapitoly zvládli, máte k dispozici několik teoretických otázek.



Další zdroje

Seznam další literatury, www odkazů ap. pro zájemce o **dobrovolné** rozšíření znalostí popisované problematiky.



CD-ROM

Informace o doplňujících animacích, videosekvencích apod., které si může student vyvolat z CD-ROMu připojeného k tomuto materiálu.



Klíč k řešení

Výsledky zadaných příkladů i teoretických otázek výše jsou uvedeny v závěru učebnice v Klíči k řešení. Používejte je až po vlastním vyřešení úloh, jen tak si samokontrolou ověříte, že jste obsah kapitoly skutečně úplně zvládli.

Úspěšné a příjemné studium s touto učebnicí Vám přeje autorský kolektiv výukového materiálu:

JINDŘICH ČERNOHORSKÝ A JAKUB JIRKA.

1. ÚVODNÍ KAPITOLA DO PROGRAMOVÁNÍ

1.1. Jak funguje počítač



Čas ke studiu:

2 hodiny.



Cíl:

Po prostudování tohoto odstavce budete umět:

- popsat funkci osobního počítače,
- popsat funkci přerušovacího systému počítače,
- popsat základní stavební bloky programu,
- popsat principy objektového programování,
- popsat zásady tvorby moderního uživatelského rozhraní.



Výklad

□ Strojový jazyk

Tato kapitola je určena k seznámení s některými základními pojmy, s kterým je třeba se seznámit, chceme-li se věnovat programování úloh pro počítače.

Počítač je složitý systém tvořený řadou nejrůznějších komponent. Mezi nimi je nejdůležitější ta, která provádí všechny výpočty. Je to centrální jednotka (Central Processing Unit), kterou podle anglického názvu označujeme zkratkou CPU. V současných personálních počítačích (PC), je CPU tvořena jediným obvodem (single "chip") rozměrů řádově čtvereční palec (inch). 1 palec=2.54 cm. Úkolem CPU tedy je provádět programy.

Program je posloupnost jednoznačných instrukcí, které jsou mechanicky prováděny počítačem. Počítač je zkonstruován tak, že umí provádět množinu velmi jednoduchých instrukcí, které jsou zapisovány velmi jednoduchým způsobem, kterému říkáme strojový jazyk. Množina všech instrukcí, kterou počítač dovede provést, se pak nazývá strojový kód. Každý typ počítače má svůj vlastní strojový kód, a může provádět přímo pouze takový program, který je zapsán pomocí instrukcí v jeho strojovém kódu. Programy zapsané v jiném jazyce musejí být nejprve přeloženy do jazyka toho stroje, na kterém mají být provedeny.

Program, který má být proveden prostřednictvím CPU, je v počítači uložen do tzv. operační paměti, nazývané též RAM = Random Access Memory. Vedle programu

jsou, resp. mohou být, v operační paměti uložena též data, která jsou programem zpracovávána. Operační paměť je tvořena posloupností paměťových buněk. Buňky paměti jsou číslovány a pořadové číslo buňky se nazývá její adresa (address). Adresa poskytuje prostředek jak vybrat určitou konkrétní informaci z mnoha údajů uložených v paměti. Jakmile chce CPU získat nějakou instrukci programu, nebo nějaká data z konkrétní buňky, zašle adresu této buňky obvodům řídícím přístup do paměti a jako odpověď dostane data obsažená v konkrétním místě paměti. Podobným způsobem zajišťuje CPU uložení konkrétních informací do paměti.

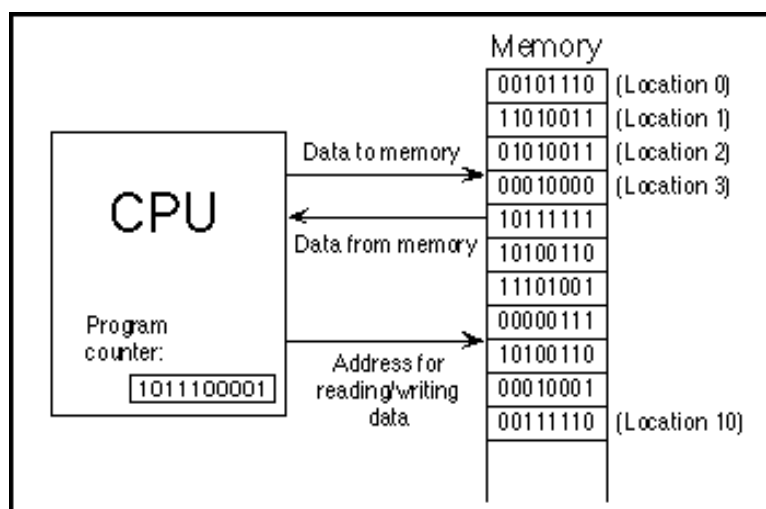
Fetch and Execute Cyklus. Na úrovni strojového jazyka jsou operace CPU vypadají jednoduše, avšak jejich detailní provedení je poměrně složité. CPU provádí program, který je uložen jako nějaká posloupnost strojových instrukcí v po sobě jdoucích buňkách operační paměti. Dělá to tak, že opakovaně, tj. postupně, čte z příslušné oblasti instrukci (tato fáze provedení instrukce se nazývá fetch), a provádí je. Fáze provedení instrukce se nazývá execute. Tento proces opakovaného čtení a provedení instrukcí se nazývá fetch-and-execute cyklus.

Podrobnosti fetch-and-execute cyklu nejsou v této chvíli nějak velmi důležité, avšak je několik základních věcí, které je třeba vědět. CPU obsahuje několik vnitřních registrů, což jsou vlastně specializované paměťové buňky umožňující uložit jedno číslo nebo jednu strojovou instrukci. Jeden z těchto registrů používá CPU, jako tzv. čítač instrukcí (program counter), pro zapamatování místa – adresy – instrukce programu, kterou právě provádí. Čítač instrukcí obsahuje před provedením instrukce vždy adresu další instrukce, kterou má CPU provést. Na začátku každého fetch-and-execute cyklu, se CPU „dívá“ do čítače instrukcí, kterou instrukci má přečíst a provést. Během fetch-and-execute cyklu se obsah v čítači instrukcí zvětší tak, aby ukazoval na následující místo v paměti. Tento obsah však může být změněn současně prováděnou instrukcí, pokud jde o tzv. skokovou instrukci, která požaduje provedení jiné instrukce programu, než té následující.

Počítač provádí strojový program mechanicky – tj. bez porozumění tomu co se provádí.

Strojové instrukce jsou vyjádřeny jako čísla ve dvojkové, binární (binary) soustavě, tj. jako binární čísla. Dvojkové číslo je zapsáno pouze pomocí dvou možných cifer, nul a jedniček. Strojová instrukce je také vyjádřena jako posloupnost nul a jedniček. Každá konkrétní instrukce je pak vyjádřena pomocí speciální kombinace nul a jedniček. Data, která počítač zpracovává, jsou také vyjádřena jako dvojková čísla. Počítač může pracovat s binárně kódovanou informací přímo, protože je principiálně konstruován na základě dvoustavových elementů, prvků, kde jeden stav je interpretován jako nula a druhý stav jako jedna.

Takže tomu jak pracuje počítač, můžete rozumět více méně takto: Operační paměť obsahuje program zakódovaný ve strojovém jazyce a data. To je všechno zakódováno jako binární čísla. Centrální jednotka (CPU) čte strojové instrukce programu z paměti jednu po druhé a vykonává je. Dělá to mechanicky, aniž by „rozuměla“ nebo „přemýšlela“ o tom co dělá; a proto program, který provádí musí být perfektní, úplný ve všech detailech a jednoznačný protože CPU nemůže dělat nic jiného než přesně to, co je napsáno v programu. Zde je schematický pohled na to co jsme si právě vysvětlili:



□ Asynchronní události: Dotazování (polling) a přerušení

CPU, stráví skoro všechn svůj čas čtením instrukcí z paměti a jejich prováděním. Avšak CPU a operační paměť jsou pouze dvěma komponentami z mnoha dalších komponent počítače. Celý počítač obsahuje další zařízení jako například:

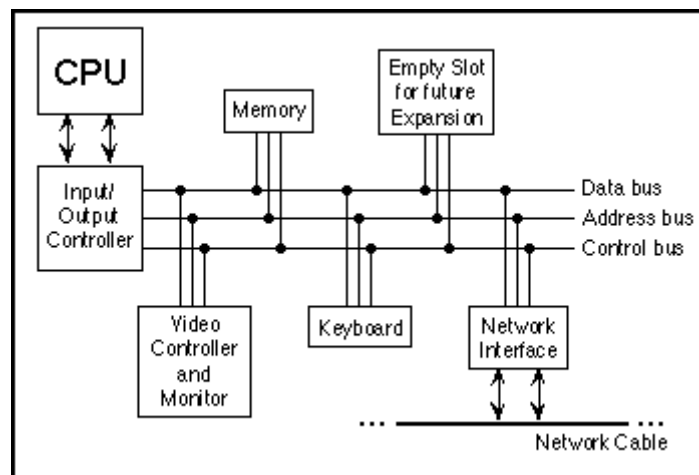
- Pevný disk (hard disk) pro ukládání a přechovávání programů a souborů dat. (Operační paměť obsahuje pouze relativně množství informace a uchovává ji jenom po dobu, po kterou je počítač zapnut. Hard disk je proto nezbytný pro permanentní uložení informace anebo pro uložení velkého objemu informací. Programy se proto před provedením musejí nahrát z disku do operační paměti, aby mohly být provedeny.
- Klávesnici (keyboard) a myš (mouse) jako uživatelské vstupy.
- Monitor (monitor) a tiskárnu (printer) které se používají pro zobrazení výstup (výsledků).
- Modem (modem), který umožňuje počítači komunikovat s ostatními počítači po telefonní lince.
- Síťové rozhraní (network interface) které umožňuje počítači komunikovat s ostatními počítači propojenými počítačovou sítí.
- Skener (scanner) který konvertuje obrázky do binárně kódovaných čísel, která mohou být uložena v některém z tzv. grafických formátů a zpracovávána počítačem.

Seznam zařízení je otevřený a počítačové systémy jsou zkonstruovány tak, že mohou být snadno rozšířeny o další zařízení. Tak či onak, CPU, musí nějak s těmito zařízeními komunikovat a řídit je. Může to dělat pouze pomocí vykonávání strojových instrukcí (to je jediné co umí). Způsob jakým to dělá, spočívá v tom, že pro každé zařízení je v paměti počítače uložen ovladač zařízení (device driver), který je tvořen softwarem, který CPU, provádí, když musí nějak pracovat s tímto zařízením. Instalace nějakého nového zařízení do systému spočívá obecně ve dvou krocích:

1. zasunutí, resp. připojení zařízení fyzicky do počítače resp. k počítači,
2. instalace software ovladače zařízení.

Bez ovladačů zařízení by odpovídající fyzické zařízení nebylo k ničemu, protože CPU by s ním neuměla komunikovat.

Výpočetní systém – počítač – k němuž je připojeno mnoho zařízení, je organizován tak, že jsou tato zařízení připojena na jednu nebo více sběrnic (bus). Sběrnice je systém vodičů, které přenášejí různé typy signálů mezi zařízeními spojenými těmito vodiči. Vodiče sběrnice přenášejí data, adresy a řídicí signály. Adresa směřuje data na konkrétní zařízení a též označuje speciální registr, nebo umístění v tomto zařízení. Řídicí signály mohou být použity například jedním zařízením to signalizovat jinému zařízení, že data pro něj jsou připravena na datové části. Výpočetní systém může být proto zcela jednoduše organizován takto:



Zařízení jako klávesnice, myš nebo síťové rozhraní (network interface) mohou nyní produkovat signály, jimiž oznamují, že chtějí být obslouženy centrální jednotkou. Jak se centrální jednotka dozví, že data jsou připravena? Jeden velmi jednoduchý, avšak nepřiliš uspokojivý způsob je, že se centrální jednotky neustále opakovaně dotazuje, zda jsou již data, nebo připojené zařízení připraveny. Jakmile zjistí, že tomu tak je, zpracuje data, nebo požadavek zařízení. Tato metoda se nazývá dotazování, polling, (the poll=průzkum). Toto je velmi jednoduché, ale velmi neefektivní, protože CPU přitom může ztratit velmi mnoho času čekáním na vstup.

Aby se zabránilo této neefektivnosti, je namísto dotazování použita technika přerušení (interrupt). Přerušení je signál zasílaný konkrétním zařízením centrální jednotce. CPU reaguje tak, že odloží vše, co v tomto okamžiku dělá a odpoví na signál přerušení tím, že provede kód pro obsluhu zařízení, které signál přerušení vyslalo. Jakmile je tento programový kód, tj. obsluha přerušení proveden, vrátí se CPU k tomu, co dělal před výskytem přerušení a pokračuje přesně v tom místě původního programu, kde byla před přerušením. Například když stisknete na klávesnici nějakou klávesu, je CPU zaslán signál přerušení od klávesnice. CPU odpoví na tento signál přerušením toho, co dělá, přečte kód klávesy, kterou jste stiskli, zpracuje případnou odpověď, pokud to stisk určité klávesy vyžaduje a pak se vrátí k úloze, kterou zpracovávala před přerušením.

Jde opět o čistě mechanický proces: Zařízení signalizuje přerušení vysláním signálu na tzv. **linku IRQ**. CPU mezi provedením dvou strojových instrukcí tuto linku testuje. Pokud je IRQ aktivní, uschová veškerou informaci, kterou potřebuje k tomu, aby po přechodu na program obsluhy přerušení a po jeho provedení bylo možno tuto

původní informaci obnovit a pokračovat v původní činnosti. Tato informace je tvořena obsahem důležitých vnitřních registrů CPU, jedním z nich je například čítač instrukcí. Pak CPU „odskočí“ na nějakou definovanou, známou adresu v paměti, kde se nalézá příslušný program pro obsluhu přerušení (device driver) a provede instrukce tam uložené. Tyto instrukce tedy tvoří program pro obsluhu přerušení (interrupt handler) jejichž provedením se zajistí obsluha přerušení (Interrupt handler je část software device driveru pro příslušené zařízení.) Na konci činnosti interrupt handleru je umístěna instrukce, která způsobí, že CPU se vrátí zpět na původní místo a obnoví stav (obnovením registrů), ve kterém byla před přerušením.

Přerušení dovolují CPU zpracovávat asynchronní události, to znamená události, které se vyskytnou v předem neznámých okamžicích nezávisle na vykonávaném programu, tj. asynchronně vzhledem k synchronní posloupnosti akcí prováděných během fetch-and-execute cyklu realizujícího momentálně prováděnou úlohu

Jiný příklad jak se využívá přerušení můžeme vidět z toho, jak postupuje CPU když chce přečíst data z harddisku. CPU může přistupovat k datům pouze tehdy, pokud jsou uložena v operační paměti. Data z disku musí být proto nejdříve zkopírována do operační paměti. Bohužel, ve srovnání s rychlostí jakou pracuje CPU je disková jednotka příliš pomalá. Když CPU potřebuje data z disku, zašle signál diskové jednotce a sdělí, kam a jaká data má do paměti načíst. (Tento signál je zaslán synchronně, tj. nějakým prováděným programem, který tato data potřebuje.) Pak namísto čekání na tato data po obecně dlouhou a nepředpověditelnou dobu, než to disková jednotka provede, CPU pokračuje ve zpracování nějaké jiné úlohy. Až disková jednotka data připraví, zašle signál přerušení CPU a interrupt handler může připravená data přečíst.

Jistě takový postup má smysl, pokud CPU má ke zpracování připraveno, resp. současně rozpracováno ne jednu, ale hned několik úloh. Pokud by tomu tak nebylo a nic lepšího neměla na práci, může jistě trávit svůj čas čekáním na to, až disková jednotka dokončí svoji úlohu. Proto všechny moderní systémy užívají tzv. multitasking (víceúlohové zpracování), aby mohly současně, přesněji souběžně, zpracovávat několik úloh. Některé počítače mohou být užívány více lidmi současně. Protože CPU je rychlá, může rychle přenášet pozornost z jedné úlohy na druhou a věnovat každé úloze zlomek sekundy. Takový režim, kdy se mezi úlohami přepíná pravidelně, případně na základě vykonávání periferní operace, se nazývá timesharing – sdílení času. Ale i v případě že u počítače sedí jeden uživatel, může být multitaskingu využito například pro současný tisk, přenos údajů po internetu a zpracování nějaké úlohy uživatelem.

Každá individuální úloha, kterou CPU zpracovává, se nazývá vlákno (thread), nebo proces. Mezi procesem a vláknem existují určité technické odlišnosti, avšak v principu pracují na podobných mechanismech. Avšak v každém časovém okamžiku může být centrální jednotkou fyzicky vykonáváno pouze jedno vlákno. CPU, bude pokračovat ve zpracování vlákna, pokud bude splněno:

- Vlákno může dobrovolně vzdát se zpracování procesorem řízení (yield control), aby dalo šanci ke zpracování jinému vlákně.
- Vlákno může čekat na nějakou asynchronní událost. Například na načtení dat z disku, nebo na to až uživatel stiskne nějakou klávesu na klávesnici. Když

čeká, je vlákno blokováno (blocked), a jiná vlákna mezitím mohou běžet. Když se událost vyskytne, přerušení vlákno probudí a to bude pokračovat ve zpracování.

- Vlákno může využít jemu přidělený čas pro zpracování a pak být odloženo (suspended) aby bylo umožněno zpracování jinému vláknu. Ne všechny systémy umožňují „z vnějšku“ vynuceně přerušit vlákno, které je zpracováváno. Tam kde to možné je, hovoříme o preemptivním multitaskingu. Při preemptivním multitaskingu, využívá počítač speciální časový čítač, který generuje pravidelně časová přerušení například 100-krát za sekundu. Když přerušení nastane, má CPU příležitost přepnout z jednoho vlákna na druhé.

Běžní uživatelé nebo programátoři se nemusejí starat o přerušení nebo interrupt handlers. Mohou se soustředit na různé úlohy nebo vlákna, které chtějí na počítači provádět; detaily jak to počítač zařídí, aby všechny úlohy byly vyřešeny, nejsou pro něho zase až tak podstatné. Ve skutečnosti většina uživatelů a mnoho programátorů se nemusí o vlákna a multitasking vůbec starat. Avšak vlákna se stala mnohem důležitější, jakmile se počítače staly mnohem výkonnější a začaly mnohem více využívat multitaskingu. Proto jsou vlákna zabudována do prostředků řady jazyků jako C# nebo jazyka Java jako základní programový koncept anebo jako nadstavba do programových knihoven které použijeme v případě, chceme-li vlákna použít.

Stejně tak důležitý je v Javě a v moderním programování obecně pojem asynchronní událost. Zatímco programátoři skutečně nezpracovávají přerušení přímo, občas musejí naprogramovat nějaký event handler, který, podobně jako interrupt handler, je volán asynchronně když se událost objeví. Takové na událostech založené programování ("event-driven programming") má velmi has a odlišný přístup od tradičního přímočarého synchronního programování.

Software, který obsluhuje všechna přerušení a komunikaci s uživatelem a hardwarovými vstupními a výstupními zařízeními se nazývá operační systém (operating system). Operační systém je základní, nezbytný software, bez něhož by žádný počítač nebyl schopen fungovat. Ostatní programy, jako textové procesory a Internetové prohlížeče, jsou závislé na operačním systému. Nejznámější operační systémy jsou například UNIX, Linux, DOS, Windows 98, Windows XP and Macintosh OS.

❑ Základní stavební bloky programů

V programování existují dva základní aspekty: data a příkazy (instrukce). Abyste mohli pracovat s daty, musíte porozumět, co jsou proměnné (variable) a co jsou typy (type). Pro práci s instrukcemi potřebujete vědět, co jsou řídicí příkazy (control structure) a co podprogramy (subroutine).

Proměnná je místo v paměti (anebo více míst v paměti, se kterými se manipuluje jako s celkem) která má jméno takže se na ni můžeme snadno odvolávat a používat ji v programu. Programátor se pouze stará o jméno, je věcí kompilátoru pamatovat si, které místo v paměti je toto jméno označuje. Programátor si však musí pamatovat, které jméno označuje nějaká data uložená v paměti.

V Javě a ve většině ostatních programovacích jazyků, má proměnná typ, je nějakého typu, což indikuje, jaký druh dat může obsahovat. Jeden typ proměnných může

obsahovat celá čísla (integer) --jako 3, -7, and 0 – zatímco jiný typ může uchovávat čísla v pohyblivé řádové čárce (floating point number). To jsou čísla s desetinnou tečkou jako 3.14, -2.7 nebo 17.0.

(Pozor, počítač rozlišuje mezi celým číslem (integer) 17 a číslem s pohyblivou řádovou čárkou 17.0; uvnitř počítače tato čísla skutečně vypadají odlišně!!) Existují také proměnné obsahující jednotlivé znaky ('A', ';', etc.), řetězce znaků ("Hello", "A string can include many characters", etc.), a méně obvyklé typy takových dat jako jsou barvy, zvuky, a libovolné další typy, se kterými program potřebuje pracovat a ukládat je.

Programovací jazyky musejí mít vždy příkazy, kterým je možno hodnoty dosadit do proměnných nebo naopak ke z těchto proměnných získat a příkazy pro provedení nejrůznějších výpočtů s daty. Například následující „přiřazovací příkaz“ (assignment statement), který se může objevit v Javovském programu, říká počítači, aby vzal číslo uložené v proměnné se jménem "principal", vynásobil ji číslem 0.07, a pak uložil výsledek do proměnné pojmenované "interest":

```
interest = principal * 0.07;
```

Existují také vstupní příkazy („input commands“) pro vstup dat od uživatele nebo ze souborů na disku počítače a výstupní příkazy („output commands“) pro zasílání dat obráceným směrem.

Tyto základní příkazy – pro přesun dat z místa na místo a pro provedení výpočtů – jsou stavební jednotky pro všechny programy. Tyto stavební bloky jsou kombinovány do složitějších programů s použitím řídicích příkazů a podprogramů.

Program je nějaká posloupnost příkazů. V běžném toku řízení („flow of control“) provádí počítač příkazy v posloupnosti, v jaké se objevují v zápise programu, tj., jeden po druhém. Takový způsob řízení výpočtu by byl však velmi omezený: počítač by se velmi brzo dostal na konec, protože množina takto zapsaných instrukcí by, pokud jde o délku, byla velmi omezená. Řídicí příkazy (Control structures) jsou speciální příkazy, které mohou změnit tok řízení programu z jednoduchého lineárního, postupného. Existují dva základní typy řídicích příkazů: cykly (loops), které umožňují opakovat nějakou sekvenci instrukcí, a příkazy výběru (branches), které umožňují se rozhodnout mezi dvěma nebo více alternativami akcí testováním podmínek které jsou vyhodnocovány za chodu programu

Například by to mohlo být tak, že je-li hodnota proměnné „principal“ větší než 10000, pak proměnná „interest“ by měly být vypočtena vynásobením proměnné principal hodnotou 0.05; jestliže uvedená podmínka neplatí, proměnná interest by měla být vypočítána násobením proměnné principal hodnotou 0.04. Program musí mít prostředek jak vyjádřit takový typ rozhodnutí. V Javě to může být vyjádřeno pomocí následujícího podmíněného příkazu („if statement“).

```
if (principal > 10000)
    interest = principal * 0.05;
else
    interest = principal * 0.04;
```

(Teď se nestarejte o detaily. Jen si pamatujte, že počítač může testovat podmínku a rozhodnout se co udělá dále na základě výsledku tohoto testu.)

Cykly jsou užívány tehdy, když určitá část výpočtu má být provedena opakovaně více než jedenkrát. Například chcete-li vytisknout adresní štítky pro každé jméno ze seznamu adres, můžete říci, "Přečti první jméno a adresu a vytiskni adresní štítek; přečti druhé jméno a adresu a vytiskni adresní štítek; přečti třetí jméno a adresu a vytiskni adresní štítek;..." Ale to se rychle stane absurdní -- a nemůže to vůbec fungovat, když nebudete dopředu vědět, kolik jmen je na seznamu. Mohli byste ale říci něco jako "Pokud (While) existují další jména na seznamu, přečti další jméno a adresu a vytiskni adresní štítek." Cyklus může být v programu užít pro vyjádření takového požadavku na opakování.

Velké programy jsou tak složité, že je skoro nemožné napsat je, pokud by neexistoval nějaký způsob rozdělit je na lépe ovladatelné části. Podprogramy jsou tím prostředkem, jak to udělat. Podprogram (subroutine) se skládá z příkazů seskupených v jakousi jednotku a řešících nějakou úlohu resp. zadání, pojmenované zvoleným jménem. Toto jméno může být pak použito jako reprezentant celé množiny do podprogramu seskupených příkazů. Předpokládejme například, že jedna z úloh, kterou je třeba v rámci vašeho programu řešit, je nakreslit na obrazovce dům. Můžete vzít příkazy nezbytné pro provedení takové úlohy, napsat je do podprogramu, a dát podprogramu vhodné jméno – řekněme - "drawHouse()". Pak kdekoliv ve vašem programu, kde potřebujete nakreslit dům, to můžete udělat jediným příkazem:

```
drawHouse ()
```

Bude to mít stejný účinek, jako kdybyste na tomto místě zapsali celou posloupnost příkazů, jimiž se nakreslí tento dům.

Výhodou je nejenom, že ušetříte psaní. Organizace vašeho programu do podprogramů Vám také pomáhá organizovat vaše přemýšlení a vaše úsilí při návrhu programu. Během psaní podprogramu pro kreslení domu se můžete koncentrovat na problém kreslení bez starosti o zbylou část programu. A jakmile je podprogram napsán, můžete zapomenout na detaily v něm obsažené, na detaily kreslení domu; tento problém je vyřešen, protože máte podprogram, který to za Vás udělá. Podprogram se stane jakoby vestavěnou částí jazyka, kterou můžete použít bez přemýšlení o detailech, co se děje v podprogramu.

Proměnná, typy, cykly, větvení a podprogramy jsou základy toho, co můžeme nazvat tradičním programováním. Avšak jak se programy stávaly větší, je třeba dodatečná struktura která by pomohla řešit tuto složitost. Jeden z nejefektivnějších nástrojů, který byl vytvořen, je objektově orientované programování, které je diskutováno v následující sekci.

❑ Objekty a Objektově orientované programování

Programy musí být navrhovány. Nikdo si nemůže jen tak sednout k počítači a z ničeho napsat program libovolné složitosti. Disciplína zvaná softwarové inženýrství (software engineering) se týká konstrukce správně pracujících, dobře napsaných programů. Softwarový inženýr se snaží o to, aby používal akceptovatelné a ověřené

metody pro analýzu problémů, které mají být řešeny a pro navrhování programu řešícího tento problém.

Během dvaceti let od roku 1970 bylo primární softwarově inženýrskou metodou tzv. strukturované programování (structured programming). Přístup strukturovaného programování návrhu programu byl založen na následující radě: Pro řešení velkých problémů je třeba rozložit problém na několik menších částí a pracovat na každé části odděleně; při řešení každé podčásti zacházet s touto jako s původním problémem a opět ji rozdělit na několik menších částí. Popřípadě, můžete řešit (programovat) určitou část již přímo, pokud je natolik srozumitelná a přehledná, že ji není nutné dále rozkládat. Tento postup se nazývá programování shora-dolů (top-down programming).

Na top-down programování není nic špatného. Je to cenný a často používaný přístup k řešení problémů. Avšak není úplný. Na jedné straně se zabývá skoro výhradně tvorbou příkazů potřebných pro řešení problému. Ale s postupem času se přišlo na to, že návrh datových struktur (data structures) je pro program nejméně tak důležitý, jako návrh podprogramů a řídicích struktur. Top-down programování nevěnuje dostatek pozornosti datům, se kterými program manipuluje.

Další problém se striktně top-down programováním je v tom, že činí obtížným znovupoužití práci již udělanou v jiných projektech. Začneme-li s určitým problémem a dělíme ho stále na menší části, top-down postup směřuje k vytvoření návrhu, který je jedinečný pro tento problém. Je nepravděpodobné, že bude možno převzít velký kus kódu z nějakého projektu a vložit ho do jiného projektu, aniž by v něm bylo třeba provést rozsáhlé úpravy. Produkce vysoce kvalitních programů je obtížná a drahá, takže programátoři i jejich zaměstnavatelé vždy tíhnou k využívání dřívějších prací.

Takže v praxi je návrh shora–dolů často kombinován se návrhem zdola-nahoru (bottom-up design). V postupu zdola-nahoru spočívá přístup v nastartování „dole“ s problémy které již dovedete řešit, resp. máte je naprogramovány (tj. máte po ruce k jejich řešení nějaké programové komponenty). Odtud pak můžete postupovat směrem nahoru, skládáním a doplňováním hotových částí a pokrýváním stále větší problémové oblasti.

Znovupoužitelné komponenty by měly být konstruovány tak modulárně jak jen je možné. Modul je komponenta většího systému, která interaguje se zbytkem systému jednoduchým, dobře definovaným a přímočarým způsobem. Hlavní myšlenkou je, že modul může být „zasunut“ („plugged into“) do existujícího systému. Detaily co se děje uvnitř modulu nejsou z hlediska systému jakožto celku podstatné potud, pokud modul plní jemu přidělenou roli korektně. To se nazývá ukrytí informace (information hiding) a je to jeden z nejdůležitějších principů softwarového inženýrství.

Jeden společný formát pro softwarové moduly předpokládá, že modul obsahuje nějaká data spolu s nějakými podprogramy pro manipulaci s daty. Například modul pro zpracování poštovního seznamu může obsahovat seznam jmen a adres spolu s podprogramem, který přidává do seznamu nové jméno, podprogram který tiskne adresní štítek, atd. V takových modulech, jsou samotná data skryta uvnitř modulu; Program, který tento modul užívá, může manipulovat s těmito daty jen nepřímo voláním podprogramů, poskytovaných modulem. To chrání data, protože ta mohou být zpracovávána jen známými a dobře definovanými způsoby. A pro ostatní

programy je snazší používat moduly, protože se nemusejí starat od detaily, jak jsou reprezentována data. Informace o reprezentaci dat je skryta.

Moduly, které by mohly garantovat tento druh ukrytí informačních detailů, se staly obecně používanými brzo po roce 1980 zejména díky programovacímu jazyku Modula. Od té doby, byla softwarovým inženýrstvím zavedena ještě jedna pokročilejší forma realizující tuto ideu. Tento poslední přístup se nazývá objektově orientované programování (object-oriented programming), který se často zkracuje jako OOP.

Ústředním pojmem OOP je objekt (object), který je určitým typem modulu obsahujícím data a podprogramy. Hlavním motivem v OOP je to, že nějaký objekt je určitým druhem soběstačné entity, která má nějaký vnitřní stav (state), (data, která obsahuje a jejich hodnoty) a že může odpovídat na zprávy (message), (zprávy jsou realizovány vyvoláním jejich podprogramů. Objekt pro poštovní seznam například má stav sestávající ze seznamu jmen a adres. Jestliže mu zašlete zprávu požadující přidání jména do seznamu, reaguje na to modifikací stavu, aby reflektoval tuto změnu. Jestliže mu zašlete zprávu požadující, aby se vytisknul, odpoví vytištěním všech jmen a adres seznamu.

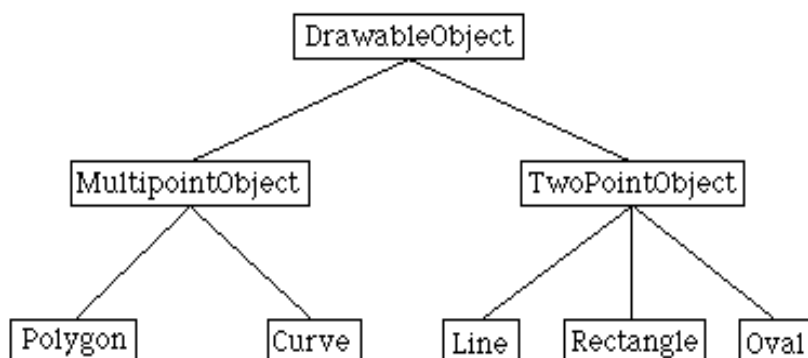
Přístup OOP k softwarovému inženýrství je započít s identifikací objektů vyskytujících se v daném problému a zpráv, kterými by tyto objekty mohly reagovat. Výsledkem je nakonec program, který obsahuje soubor objektů, z nichž každý má svoje data a vlastní množinu podprogramů, tzv. metod představujících „zodpovědnosti“ resp. chování objektu. Jsou vlastně funkce, které umí objekt provést. Objekty interagují, tím že si zasílají zprávy, tj. volají funkce objektů, po kterých chtějí, aby něco prostřednictvím příslušné funkce vykonaly. Zde není mnoho "top-down" postupů, a lidé zvyklí na tradiční programování mohou mít velké problémy, než se přeorientují na OOP. Avšak ti, kteří užívají OOP tvrdí, že objektově orientované programy jsou lepšími modely způsobů, jak funguje reálný svět, a tedy že je snadnější je napsat, snadnější porozumět, a s větší pravděpodobností že jsou správné.

Měli byste přemýšlet o objektech jakožto o "věducích" jak odpovídat na určité zprávy. Různé objekty by mohly odpovídat na stejnou zprávu různým způsobem. Například, zpráva "print" by měla vytvořit velmi odlišné výsledky, podle toho, kterému objektu byla zaslána. Tato vlastnost objektů – že různé objekty mohou odpovídat na tutéž zprávu různými způsoby – se nazývá polymorfismus (polymorphism).

Je společnou vlastností všech objektů, že jsou v určitém příbuzenském vztahu. Objekty, které obsahují tatáž data a odpovídají na tytéž zprávy stejným způsobem, patří do téže třídy (class). Ve skutečném programování je třída primární; neboli třída je vytvořena jako první a pak teprve je vytvořen jeden nebo více objektů podle třídy jakožto šablony. Avšak objekty mohou být podobné i tehdy aniž náležejí do stejné třídy.

Například uvažme kreslící program, který dovoluje uživateli kreslit přímky, obdélníky, ovály, polygony a křivky na obrazovce. V programu je každý na obrazovce viditelný objekt jako softwarový objekt v programu. Zde by mělo být pět tříd objektů jedna pro každý typ viditelného objektu, který má být nakreslen. Všechny přímky budou patřit do jedné třídy, všechny obdélníky do druhé atd. Tyto třídy jsou obvykle v nějakém

vztahu. Každá z nich reprezentuje objekt, který může být nakreslena. Ty by všechny mohly být například schopny odpovědět na zprávu "Nakresli_se". Jiná úroveň seskupování, založená na datech, která jsou třeba pro reprezentaci každého typu objektu, je méně obvyklá, ale byla by velmi užitečná v nějakém programu. Můžeme seskupit skupinu polygonů a křivek dohromady jako "vícebodové objekty," zatímco přímky, obdélníky, a ovály jsou "dvou-bodové objekty." Úsečka je určena svými koncovými body, obdélník svými dvěma rohy, a nějaký ovál dvěma rohy obdélníku, který ho obsahuje.) Můžeme zachytit tyto vztahy následovně:



DrawableObject, MultipointObject a TwoPointObject by byly třídy v programu. MultipointObject a TwoPointObject by byly podtřídy (subclasses) třídy DrawableObject. Třída Line by byla podtřídou třídy TwoPointObject a (nepřímo) i třídy DrawableObject. O podtřídě třídy se říká, že dědí (inherit) vlastnosti této třídy. Podtřída může ke svému dědictví přidávat a dokonce i přepsat, změnit ("override") část tohoto dědictví (definováním odlišné odpovědi na tutéž metodu, zprávu). Nicméně, přímky, obdélníky, a tak dále jsou nakreslitelné objekty, a třída DrawableObject vyjadřuje toto příbuzenství.

Dědičnost je mocný prostředek pro organizaci programu. Souvisí také s problémem znovu použití softwarových komponent. Každá třída je elementární znovupoužitelná komponenta. Nejenom, že může být znovupoužita přímo, jestliže přesně odpovídá tomu, co potřebuje nějaký program, který chcete napsat, ale pokud odpovídá skoro, stále ji ještě můžete použít definováním nějaké podtřídy a provedením pouze malých změn nezbytných pro její adaptaci Vaším potřebám.

Takže OOP je jak dokonalým nástrojem pro vývoj programů tak i částečným řešením problému znovupoužití software. Objekty, třídy, a objektově-orientované programování jsou důležitými tématy při výuce programovacího jazyka Java, nebo C# a také C++.

□ Moderní uživatelské rozhraní

Když byly uvedeny do provozu první počítače, běžní smrtelníci -- včetně většiny programátorů -- se nemohli dostat do jejich blízkosti. Počítače byly uzavřeny v místnostech s operátory oblečenými do bílých plášťů. Operátoři přebírali vaše programy a data, nahrávali je do počítače, a vracely výsledky z počítače o nějaký čas později. Když byly v 60-tých letech navrženy systémy se sdílením času (timesharing) -- kde počítač přepíná rychle svoji pozornost z jedné úlohy (osoby) na druhou, bylo možné, aby současně několik lidí komunikovalo s počítačem v témže čase. V systémech sdílení času sedí uživatelé u terminálů, kde píšou operátorské příkazy

pro počítač a počítač naopak píše svoje odpovědi. První personální počítače také používaly textově zapisované příkazy a odpovědi, kromě toho zde byla pouze jedna zainteresovaná osoba v daném čase. Tento typ rozhraní pro interakci mezi uživatelem a počítačem se nazývá textové rozhraní (command-line interface).

Dnes ovšem většina lidí komunikuje s počítači úplně jiným způsobem. Používají grafické uživatelské rozhraní (Graphical User Interface) neboli GUI. Počítač nakreslí na obrazovce komponenty tohoto rozhraní. Komponenty zahrnují věci jako okna (windows), lišty (scroll bars), menu, tlačítka a ikony pro manipulaci s komponentami se obvykle používá myš (mouse).

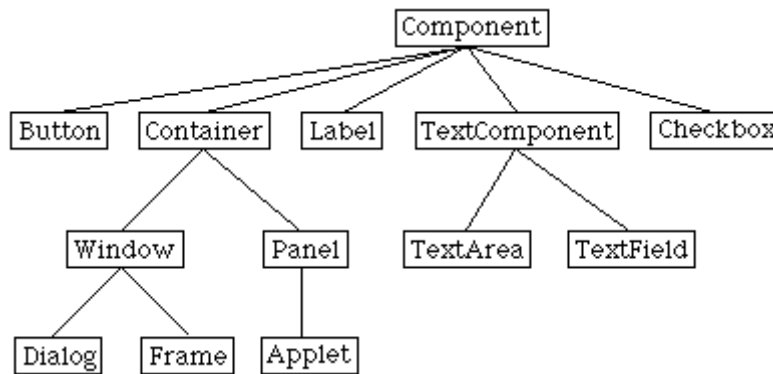
Mnoho komponent GUI rozhraní se stalo přímo standardy. To znamená, že mají podobný zevnějšek a chování na mnoha různých počítačových platformách včetně Macintosh, Windows, a různých UNIX okenních systémech. Java programy, o nichž se předpokládá, že poběží na mnoha různých platformách bez modifikace programu, mohou užít všechny standardní GUI komponenty. Mohou se platformu od platformy lišit trochu v podobě, ale jejich fungování by mělo být identické na libovolném počítači, na němž program běží.

Níže je uveden velmi jednoduchý Java program – ve skutečnosti nějaký applet ("applet,") protože běží právě zde ve středu stránky – ukazuje několik standardních GUI rozhraní. Jsou zde čtyři komponenty, jejichž prostřednictvím můžete komunikovat: tlačítkem (button), zaškrtačím políčkem (checkbox), textovým polem (text field), a vysunovacím menu (pop-up menu). Tyto komponenty jsou popsány. V appletu je několik dalších komponent. Samy popisy jsou komponentami (třebaže je nemůžete měnit). Pravá polovina appletu je textovou oblastí komponenty, kde je možno zobrazit více řádků textu. Ve skutečnosti v terminologii Javy, celý applet je sám považován za komponentu."

Java má nyní v podstatě dva kompletní systémy GUI komponent. Jeden z nich, AWT neboli Abstract Windowing Toolkit, byl dostupný již v počátečních verzích Javy. Druhý, známý jako Swing, je obsažen ve verzi 1.2 Javy a pozdějších. Zde je verze appletu který užívá Swing namísto AWT. Když vidíte applet nahoře jen jako prázdnou oblast, znamená to, že užíváte Webový prohlížeč (browser) užívající nějakou starší verzi Javy:

Když interagujete s GUI komponentami v těchto apletech, generujete "události". Například; kliknutí na push button generuje událost. Pokaždé když je generována událost, je zaslána appletu zpráva, která mu sdělí, že se vyskytla událost a applet odpovídá podle svého programu. Ve skutečnosti program je tvořen hlavně „handlersy událostí" ("event handlers") které říkají appletu, jak odpovídat na různé typy událostí. V tomto případě byly applety naprogramovány tak, aby odpovídaly na každou událost zobrazením zprávy v oblasti TEXTU.

Použití termínu "zpráva" je záměrné. Již jste se dověděli, že zprávy jsou zasílány objektům. Ve skutečnosti, jsou Javovské GUI komponenty implementovány jako objekty. Java zahrnuje mnoho předdefinovaných tříd, které reprezentují různé typy GUI komponent. Některé z těchto tříd jsou podtřídy jiných tříd. Zde je diagram ukazující některé GUI třídy v AWT a jejich vztahy:



Pro tuto chvíli se nestarejte o detaily, ale snažte se získat nějakou představu o tom, jak se zde používají objektové programování a dědičnost. Všimněte si, že všechny GUI třídy nebo podtřídy, jsou přímo nebo nepřímo, podtřídou třídy Component. Dvě z přímých podtříd třídy Component mají také svoje podtřídy. Třídy TextArea a TextField, které se do určité míry chovají podobně nebo stejně, jsou seskupeny dohromady jako podtřída třídy TextComponent (v ní je možno definovat to chování které je společné těmto dvěma podtřídám). Třída Container popisuje komponenty, které obsahují jiné komponenty. Třída Applet je nepřímo podtřídou třídy Container protože applety mohou obsahovat komponenty jako tlačítka (buttons) a textová pole (text fields).

Z tohoto krátkého rozboru je snad možno usoudit, jak učinit programování GUI efektivní použijeme-li objektově orientovaný přístup. Skutečně GUIs spolu s jejich „viditelnými objekty“, jsou pravděpodobně hlavním faktorem přispívajícím k oblíbenosti OOP.

Programování s GUI komponentami a událostmi je jedním z nejzajímavějších aspektů Javy.

□ Internet a World-Wide Web

Počítače mohou být dohromady spojeny sítí (network). Počítač v síti může komunikovat a ostatními počítači na téže síti prostřednictvím výměny dat a souborů a zasíláním zpráv. Počítače propojené v síti mohou dokonce spolupracovat na při řešení rozsáhlých výpočetně náročných úloh.

Dnes jsou miliony počítačů na celém světě propojeny k jediné gigantické síti počítačů zvané „Internet“. Nové počítače jsou připojovány k Internetu každý den. Počítač může být připojen k internetu i dočasně s použitím modemu, který zprostředkuje přístup do Internetové sítě prostřednictvím telefonní sítě.

Pro komunikaci po Internetu existují složité protokoly. Protokol je prostě podrobná specifikace jak se má postupovat při komunikaci. Aby mohly vůbec komunikovat dva počítače, musejí používat stejný protokol. Nejzákladnějšími protokoly na Internetu jsou Internet Protocol (IP), který specifikuje, jak jsou data přenášena z jednoho počítače na druhý fyzicky, a Transmission Control Protocol (TCP), který zajišťuje, že data odeslaná pomocí IP jsou dodána celá (všechna) a bezchybně. Tyto dva protokoly, které se dohromady nazývají TCP/IP, poskytují základ pro komunikaci. Další protokoly užívají TCP/IP pro zasílání specifických informací a dat jako jsou soubory a elektronická pošta.

Veškerá komunikace na Internetu se odehrává ve formě paketů (packet). Paket je tvořen nějakými daty posílanými z jednoho počítače na druhý a adresou, která indikuje, kam na Internetu se mají data přepravit. Představte si paket jako obálku s adresou na obálce a s daty, nebo zprávou uvnitř obálky. Paket také obsahuje zpáteční adresu, tj. adresu odesílatele. Paket může obsahovat pouze omezené množství dat; delší zprávy musejí být rozděleny mezi několik paketů, které jdou pak sítí odeslány individuálně a na místě doručení poskládány dohromady.

Každý počítač má na Internetu nějakou IP adresu (IP address), což je číslo které ho jednoznačně identifikuje mezi všemi počítači v síti. IP adresa se používá pro adresaci paketů. Počítač může poslat data jen jinému počítači na Internetu, zná-li jeho IP adresu. Protože lidé raději používají jména než čísla, je mnoho počítačů také identifikovatelných jmény, nazývanými doménami (domain names). Například „hlavní“ počítač VŠB má doménu `www.vsb.cz`. Domény jsou zvoleny pro pohodlí člověka, počítač potřebuje stále vědět předtím, než začne komunikovat IP adresou. Na Internetu jsou počítače, jejichž úkolem je překládat jména domén na IP adresy. Když použijete doménu, váš počítač zašle zprávu speciálnímu serveru (domain name server) aby zjistil odpovídající IP adresu. Pak váš počítač používá pro komunikaci s tímto počítačem získanou IP adresu místo domény.

Internet poskytuje řadu služeb počítačům na něj připojeným (a také samozřejmě uživatelům těchto počítačů). Tyto služby používají TCP/IP pro zasílání různých dat sítí. Mezi nejpopulárnější služby patří vzdálené připojení remote login, elektronická pošta, FTP a World-Wide Web.

Remote login umožňuje uživateli jednoho počítače připojit se (log) na jiný počítač. (Samozřejmě, že tento vzdálený uživatel musí znát uživatelské jméno (user name) a heslo (password) pro účet na druhém počítači.) Existuje několik různých protokolů pro vzdálené připojení, včetně tradičního telnetu (telnet) a bezpečnějšího ssh (secure shell). Telnet a ssh podporují pouze rozhraní příkazového řádku. V podstatě první počítač funguje jako terminal pro druhý počítač. Vzdálené připojení je užíváno často lidmi, kteří jsou mimo svoje pracoviště pro přístup k jejich počítači v zaměstnání – a mohou to udělat z libovolného počítače na Internetu odkudkoliv na světě.

Electronic mail, neboli email – elektronická pošta, poskytuje možnost komunikace mezi dvěma osobami přes Internet. Email je zaslán konkrétním uživatelem jednoho počítače druhému uživateli jiného počítače. Každá osoba je identifikována pomocí jedinečné emailové adresy, která se skládá ze jména domény počítače, kde dostávají svoje zprávy společně s jejich uživatelským jménem a osobním jménem. Emailová adresa má formu "username@domain.name". Například emailové adresy zaměstnanců VŠB jsou standardně ve tvaru *jmeno.prijmeni@vsb.cz*. Email je fyzicky přenášen z jednoho počítače na druhý s použitím protokolu SMTP (Simple Mail Transfer Protocol). Email je možná nejrozšířenější a nejdůležitější službou Internetu, ačkoliv s jeho popularitou úspěšně soupeří World-Wide Web.

FTP (File Transport Protocol), protokol přenosu souborů, je navržen pro kopírování souborů z jednoho počítače na druhý. Stejně jako při vzdáleném přihlášení uživatel FTP potřebuje uživatelské jméno a heslo pro přístup k počítači, z něhož chce kopírovat soubory. Avšak mnoho počítačů bylo vybaveno speciálními účty, přes které lze přistupovat prostřednictvím FTP s uživatelským jménem "anonymous" a

libovolným heslem. Tento tzv. anonymní FTP (anonymous FTP) může být použit pro vytváření souborů na jednom počítači veřejně dostupných komukoliv pře Internet.

World-Wide Web (WWW) je založen na stránkách (pages) které mohou obsahovat nejrůznějšího druhu a také odkazy (links) na jiné stránky. Tyto stránky jsou zobrazovány prostřednictvím programů --webových prohlížečů (Web browser) – jako jsou Netscape, Internet Explorer, Mozilla, Opera aj. Mnoho lidí si myslí, že World-Wide Web je Internet, ale to je doopravdy jen grafické uživatelské rozhraní na Internet. Stránky, které vidíte s pomocí Webového prohlížeče, jsou soubory, které jsou uloženy na počítačích připojených na Internet. Když požádáte, aby Váš prohlížeč nahrál stránku, kontaktuje počítač, na němž je stránka uložena a přenesení ji na Váš počítač s použitím protokolu známého jako HTTP (HyperText Transfer Protocol). Libovolný počítač na Internetu může publikovat svoje stránky na World-Wide Web. Když užíváte prohlížeč, máte přístup k rozsáhlému moři vzájemně propojených informací, mezi nimiž je možno navigovat bez zvláštní speciálních počítačových znalostí. Web je nejvíce vzrušující část Internetu a pohání Internet do vpravdě fenomenálního tempa růstu. Jestliže splní svoje sliby, může se Web stát universální a základní složkou každodenního života.

Je třeba poznamenat, že typický Webový prohlížeč může užívat i jiné protokoly vedle HTTP. Například, může také užít FTP pro přenos souborů. Tradiční uživatelský interface na FTP je rozhraní příkazového řádku, takže mezi mnoha věcmi, které dělá, poskytuje Web browser moderní grafické uživatelské rozhraní na FTP. To umožňuje lidem užít FTP, aniž by dokonce věděli, že taková věc existuje! (Tento fakt by vám mohl pomoci pochopit rozdíl mezi programem a protokolem. FTP není program. Je to protokol, tj. množina standardů pro jistý typ komunikace mezi počítači. Abyste mohli použít FTP, potřebujete program, který implementuje tyto standardy. Různé FTP programy vám to mohou prezentovat s velmi odlišným uživatelským rozhráním. Podobně různé Webové prohlížeče mohou předkládat informaci uživateli velmi odlišným rozhráním, ale všechny musejí užít HTTP, aby dostaly informaci z Webu.)

A co to má společného s Javou? Ve skutečnosti je Java velmi těsně spojena s Internetem a World-Wide Webem. Jak jsme viděli v předcházející sekci, speciální Javovské programy zvané applety jsou určeny k přenosu přes Internet a zobrazení na Webových stránkách. Webový server přenáší Javovský applet tak jako kdyby přenášel jakýkoliv jiný typ informace. Webový prohlížeč, který rozumí Javě -- to je, že zahrnuje interpreter pro JVM (Java virtual machine) – může pak spustit applet přímo na Webové stránce. Protože applety jsou programy, mohou dělat téměř všechno, včetně složité interakce s uživatelem. S Javou se Webová stránka stává více než jen pasivním zobrazovačem informace. Stává se něčím, co si programátoři mohou představit a implementovat.

Ve spojení s Webem není jediná výhoda Javy. Avšak mnoho dobrých programovacích jazyků bylo navrženo jen proto, aby byly vzápětí zapomenuty. Java měla velké štěstí, že se „svezla“ na vlně Web's a získala nesmírnou a vzrůstající popularitu.



Shrnutí pojmů

Osobní počítač (PC), Architektura, Centrální výpočetní jednotka (CPU), Java, Web, Instrukce



Otázky

1. Jednou z komponent počítače je *CPU*. Co je CPU a jakou úlohu hraje v počítači?
2. Vysvětlete, co je míněno pod pojmem "asynchronní událost." Uveďte nějaké příklady.
3. Jaký je rozdíl mezi "překladačem, kompilátorem" a "interpretrem"?
4. Vysvětlete rozdíl mezi vyšším programovacím jazykem a strojovým jazykem.
5. Máte-li zdrojový kód programu napsaný v Javě, a chcete tento program spustit, budete potřebovat, jak *překladač*, tak interpreter. Co bude dělat Java překladač a co Java interpreter?
6. Co je podprogram, subroutine?
7. Java je objektově – orientovaný programovací jazyk. Co je *object*?



Úlohy k řešení



Další zdroje

1.2. Java Virtual Machine – Virtuální stroj Javy



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- popsat virtuální stroj JVM (Java Virtual Machine).



Výklad

□ Java, Virtuální stroj Javy

Strojový jazyk je tvořen velmi jednoduchými instrukcemi, které mohou být provedeny přímo centrální jednotkou počítače. Skoro všechny programy jsou však zapsány v nějakém programovacím jazyce vyšší úrovně (high-level programming languages) jako jsou Java, Pascal, nebo C++. Program napsaný v nějakém programovacím jazyce nemůže být přímo proveden na počítači. Nejdříve musí být přeložen do strojového jazyka. Tento překlad se provádí programem zvaným překladač, nebo translátor nebo kompilátor (compiler). Překladač čte program ve vyšším programovacím jazyce (tzv. zdrojový program) a převádí ho do proveditelné formy strojového jazyka. Jakmile je překlad hotov, může přeložený program běžet mnohokrát avšak přirozeně jen na takovém typu počítače, do jehož strojového kódu byl přeložen. Obecně počítače různého typu mají odlišné strojové kódy. Pokud má program běžet na počítači odlišného typu, musí být opět přeložen pomocí vhodného kompilátoru.

Vedle kompilace existuje alternativa, jak vykonávat program ve vyšším programovacím jazyce. Namísto užití kompilátoru, který přeloží program najednou a pouze jednou, můžeme užít interpreter, který ho bude překládat postupně instrukci-po-instrukci, přičemž každou přeloženou instrukci bude okamžitě provádět=interpretovat. Interpreter je program, který funguje trochu podobně, jako CPU, v jakémsi vyšším režimu fetch-and-execute cyklu. Interpreter provádí program tak, že opakuje cyklus, v němž čte jeden příkaz (instrukci) programu, analyzuje ho a rozhodne, co je potřeba provést pro splnění tohoto příkazu, a pak provede tomu odpovídající posloupnost strojových instrukcí, aby se tak stalo.

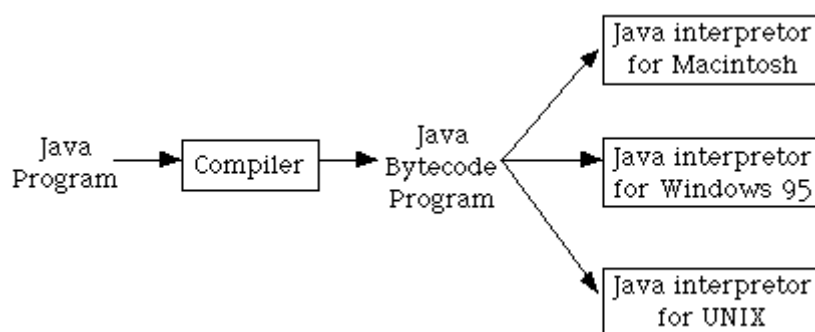
Interpretery se používají k vykonávání programů v určitých specifických vyšších programovacích jazycích. Například programovací jazyk Lisp je obvykle vykonáván pomocí interpreteru než pomocí kompilátoru. Původní verze jazyka Basic byla založena na interpreteru, velmi populární počítač osmdesátých let Sinclair obsahoval

interpreter Basicu jako firmware, tj. byl trvale zapsán v obvodech paměti počítače jako základní komunikační programovací prostředek systému. Interpretery mají ještě jedno zajímavé použití: umožňují chod nějakého programu ve strojovém jazyce určeného pro jeden typ počítače na zcela odlišném typu počítače. Například existuje program „Virtual PC“ který běží na počítačích Macintosh. Virtual PC je interpreter který vykonává strojový program napsaný pro počítače IBM-PC. Jestliže na vašem Macintosh je spuštěn interpreter Virtual P můžete pod jeho kontrolou provést libovolný PC program, včetně programů napsaných pod Windows. (Samozřejmě, že PC program zde poběží mnohem pomaleji než by běžel na skutečné hardwaru IBM PC. Kompilované programy vždy běží mnohem rychleji než interpretované.)

Tvůrci Javy zvolili „zlatou střední cestu“: kombinaci kompilovaného a interpretovaného režimu provedení. Programy zapsané v Javě jsou kompilovány do zvláštního strojového jazyka, který na žádném počítači ve skutečnosti neexistuje. Takový neexistující počítač je vlastně jakýmsi „virtuálním“ počítačem a hovoříme proto o virtuálním stroji Javy (Java virtual machine) - JVM. Strojový jazyk pro JVM se nazývá Java bytecode. Neexistuje důvod, proč by Java bytecode nemohl být použit jako strojový jazyk reálného počítače, než aby byl pouze jazykem nějakého virtuálního počítače. Dokonce firma Sun Microsystems, která se ujala vývoje Javy, vyvinula procesory, které používají Java bytecode jako svůj strojový jazyk.

Avšak Java může být skutečně použita na jakémkoliv počítači. Vše co k tomu počítač potřebuje je interpreter Java bytekódu. Takový interpreter simuluje virtuální stroj Javy stejným způsobem, jako Virtuální PC simuluje PC počítač.

Samozřejmě, že pro každý typ počítače je potřeba specifický interpreter Javovského bytekódu, ale jakmile je jednou na počítači nainstalován, může být na počítači proveden libovolný Java bytecode program. A tentýž Javovský Java bytecode program může běžet na libovolném počítači který má svůj interpreter. Toto jednou ze základních vlastností Javy: tentýž, do Java bytekódu zkompilovaný program, může běžet na mnoha různých typech počítačů.



Proč však vůbec užívat takový „mezikód“? Proč raději nedistribtovat originální Javovský program a ten přeložit na konkrétním počítači do jeho strojového kódu, podobně jako tomu je v jiných programovacích jazycích? Je zde více důvodů. Především kompilátor musí „rozumět“ Javě, což je relativně složitý programovací jazyk. Sám překladač je složitý. Interpreter Java bytekódu je na druhé straně poměrně malý, a jednoduchý program. Proto je snazší napsat interprete bytekódu pro úplně nový typ počítače. Jakmile se to podaří, může na počítači běžet libovolný Javovský program. Napsat kompilátor Javy pro tentýž počítač by bylo mnohem obtížnější.

A navíc mnoho Java programů je určeno pro nahrávání, stahování (download) prostřednictvím sítě – Internetu. To vzbuzuje obvykle obavy o bezpečnost. Interpreter bytekódu zde působí jako nárazník mezi vaším počítačem a programem který stahujete. Ve skutečnosti na počítači běží interpreter, který provádí po síti stažený program nepřímo. Interpreter Vás může ochránit před potenciálně nebezpečnými akcemi ze strany tohoto programu.

Měli bychom si říci, že není nezbytné propojení mezi Javou a Java bytekódem. Program napsaný v Javě může být určitě kompilován přímo do strojového kódu konkrétního počítače. A programy napsané v jiných programovacích jazycích by mohly být přeloženy do Java bytekódu. Avšak je to kombinace Javy a Java bytekódu která je nezávislá na platformě, bezpečná a síťově kompatibilní přičemž vám dovoluje programovat v moderním vyšším objektově orientovaném jazyce.

Řekněme si také, že skutečně obtížná část této platformové nezávislosti je ta, která podporuje grafické uživatelské rozhraní ("Graphical User Interface") – s okny, tlačítky, atd. – která bude pracovat na všech platformách, které podporují Javu.



Shrnutí pojmů

Interpreter, Java, Java virtuální stroj (JVM)



Otázky

1. Co je *proměnná*? (Je řada různých pojmů spojených s proměnnými v Javě. Zkuste si ve vaší odpovědi vzpomenout na všechny čtyři. Náповěda: Jeden z aspektů je jméno proměnné.)
2. Java je "na platformě-nezávislý jazyk." Co to znamená?
3. Co je "Internet"? Uvedte některé příklady, jak je využíván. (Jaké typy služeb poskytuje?)
4. Prostudujte text této kapitoly a vytvořte další otázky, které byste položili svým studentům, kdybyste byli v roli učitele



Úlohy k řešení



Další zdroje

[1] V tomto textu jsou důležité pojmy zobrazeny červenou barvou, za některými je pak ponechán jejich anglický ekvivalent. (Text je překladem první části elektronické učebnice Introduction to Programming Using Java, Version 4.1, June 2004, Author: David J. Eck, WWW: <http://math.hws.edu/eck/>)



CD-ROM

2. PROGRAMOVACÍ JAZYK C

Tento text je krátkým úvodem do programovacího jazyka C (dále používáme i název „Céčko“). Jeho cílem je umožnit studentovi co nejrychlejší start do programování v tomto jazyce a není zaměřena na veškeré detaily tohoto jazyka. V žádném případě zde není záměr nahradit nějakou z mnoha podrobných učebnic Céčka ani rozšířit jejich nepřebernou řadu. K nim může student sáhnout až v případě, kdy se bude hlouběji zabývat touto problematikou a zpracovávat nějaký složitější program a podle potřeby se seznamovat s dalšími nezbytnými detaily.

Praktickou znalost získá student ve cvičeních, která jsou zaměřena na zvládnutí základních dovedností a na všechny praktické otázky související s technickým vývojem programů.

2.1. První program v jazyku C



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- napsat jednoduchý program v jazyce C



Výklad

V úvodní kapitole jsme se dozvěděli, že počítač resp. jeho CPU, je schopen provádět pouze program zapsaný ve strojovém jazyce. Viděli jsme však také, že existují pohodlnější možnosti, kterými jsou programovací jazyky, protože programování ve strojovém kódu je velmi pracné a složité a pro naprogramování složitějších systémů téměř nemožné.

Programovací jazyky umožňují psát kód našeho programu ve formě, která je pro člověka jednodušší, srozumitelnější a přehlednější a to proto, že je tento zápis bližší přirozenému jazyku.

Jednou z těchto možností je jazyk C, který se používá pro programování úloh nejenom na PC počítačích, ale i na řadě jiných procesorů, které se používají s výhodou pro řízení a také pod různými operačními systémy.

Ukažme si snad nejjednodušší program v jazyce C.

```
#include <stdio.h>
/*      Salvete amici (lat.) můžeme přeložit Ahoj kamarádi      */
void main()
{
```

```
printf("\nSalvete amici\n");
}
```

Abychom mohli tento program spustit na počítači, musíme

1. Napsat jeho text a uložit do souboru `salvete.c` (neřešme zatím jak)
2. Přeložit (zkompilovat) tento program pomocí překladače z jazyka C pro daný počítač
3. Spustit zkompilovaný proveditelný (executable) výsledek překladače (bude uložen se jménem `salvete.exe`)

Výsledkem činnosti programu bude výpis textu `Salvete amici` s jedním předcházejícím volným řádkem na obrazovce resp. na tzv. konzole. Konzola je výstupní zařízení, které umožňuje výstup alfanumerické informace. V OS MS Windows je konzola představována samostatným oknem s černým pozadím. V některých vývojových systémech, např. v Eclipse, je výstup realizován v samostatné části, okně, která součástí okna vývojového systému.

Program v C obsahuje *funkce a proměnné*. Funkce specifikují jednotlivé akce, které budou provedeny programem. Funkce „main“ představuje hlavní funkci, v níž program začíná a která je obsažena v každém programu. Další funkce, které jsou v programu použity, se volají z funkce main resp. z funkcí, které byly z main zavolány. Obvykle je funkce main naprogramována co nejmenší a z ní volané funkce pak provádějí potřebné podúlohy.

Náš program `salvete.exe` volá funkci ***printf***, která je funkcí výstupu a provádí výpis textu na tzv. standardní výstupní zařízení, resp. konzolu. Tímto zařízením je zpravidla obrazovka resp. tzv. konzola. Které zařízení to konkrétně je, je definováno ve standardním souboru ***stdio.h***. Funkce ***printf*** tiskne zprávu „Salvete amici“ na logické zařízení „***stdout***“, „`\n`“ tiskne znak „nového řádku“, který posune kurzor na následující řádek. Posun řádku je proveden i tehdy, když znak „`\n`“ je jediným, který je v řetězci uveden. Následující program proto provede tentýž výstup jako předcházející program:

```
#include <stdio.h>
void main()
{
    printf("\n");
    printf("Salvete Amici");
    printf("\n");
}
```

První příkaz „`#include <stdio.h>`“ specifikuje, že má být do textu programu na toto místo vložen soubor `stdio.h`, což je tzv. hlavičkový soubor. Tento soubor specifikuje funkce a proměnné knihovny standardního vstupu a výstupu (I/O library). Všechny proměnné užívané v jazyku C musí být explicitně definovány před jejich prvním použitím. Soubory s koncovkou „.h“ jsou tzv. hlavičkové soubory („header files“), které obsahují definice proměnných a funkcí, které potřebujeme volat pro zajištění nezbytných funkcí programu, ať už to je v námi – tj. uživatelem – napsané části kódu anebo z jiných tzv. standardních knihoven C. Tzv. direktiva „`#include`“ tedy přikazuje preprocesoru, aby vložil obsah uvedeného souboru na příslušné místo uživatelského

kódu. Závorky <> uvnitř direktivy „<... >“ napovídají překladači, že má tento soubor hledat v oblasti tzv. standardních systémových knihoven.

Slovo void předcházející název „main“ znamená, že funkce main je typu „void“, tj. že nevrací nějaký výsledek po ukončení své činnosti.

Znak „;“ označuje konec příkazu. Bloky příkazů jsou ohraničeny závorkami {...}, podobně jako v definici funkce. Všechny příkazy C jazyka se píší v tzv. volném formátu tj., bez jakéhokoliv pevného rozvržení textu do sloupců. Tzv. bílé znaky (whitespace = tabulátory nebo mezery) nemají žádný význam. Význam mají pouze tehdy, jsou-li uvedeny mezi uvozovkami, kde tvoří součást textu, který je těmito uvozovkami omezen. Následující program bude produkovat stejný výstup jako program předcházející:

```
#include <stdio.h>
void main(){printf("\nSalvete amici\n");}
```

2.2. Příklad matematického programu



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

definovat co je to .NET Framework a jeho základní vlastnosti, definovat .NET Compact Framework

popsat základní vlastnosti .NET Frameworku a popsat rozdíly mezi jednotlivými verzemi, zařadit rodinu frameworků .NET mezi ostatní frameworky společnosti Microsoft



Výklad

Následující program, sinus.c, počítá tabulku funkčních hodnot funkce sinus pro úhly mezi 0 až 360 stupni.

```
#include <stdio.h>
#include <math.h>
void main()
{
    int    uhel_stupne;
    double uhel_radian, pi, value;

    /* Tisk hlavičky */
    printf ("\nVypocet tabulky funkce sinus\n\n");

    /* vypočítejme pi */
    /* nebo užijeme pi = M_PI, kde M_PI je definováno v math.h */
    pi = 4.0*atan(1.0);
    printf ( " Hodnota PI = %f \n\n", pi );
    printf ( " Uhel      Sinus  \n" );

    uhel_stupen=0;          /* počáteční hodnota úhlu ve stupních */
                           /* scan over angle */

    while ( uhel_stupen <= 360 ) /* cyklus opakující výpočet dokud
není proměnná uhel_stupen > 360 */
    {
        uhel_radian = pi * uhel_stupen/180.0 ;
        value = sin(uhel_radian);
        printf ( " %3d      %f \n ", uhel_stupen, value );

        uhel_stupen = uhel_stupen + 10; /* zvětší řídicí proměnnou cyklu */
    }
}
```

Kód programu začíná sérií komentářů, které mohou indikovat cokoli, co autor považuje za nutné uvést na začátku. Např. účel programu, jméno autora, datum vytvoření, verzi atp. Komentáře lze psát kdekoli v programu: jakýkoli znak mezi závorkami „/*“ a „*/“ je překladačem ignorován a tak lze pomocí vhodně umístěných komentářů učinit kód programu srozumitelnější. K srozumitelnosti programu lze také přispět vhodnou volbou identifikátorů (jmen) proměnných tak, aby odpovídala co nej přesněji věcnému významu proměnné v kontextu řešení úlohy, pro níž je program napsán.

Další příkaz `#include` obsahuje hlavičkový soubor pro standardní matematickou knihovnu `math.h`. Tuto knihovnu potřebujeme proto, abychom mohli použít volání trigonometrických funkcí `atan` a `sin` obsažených v knihovně `math.h`.

Jména proměnných můžeme volit poměrně libovolně, avšak musí být tvořena pouze znaky písmen, číslic a znakem podtržítka „_“, přičemž nesmí začínat číslicí. Takto utvořené jméno se nazývá identifikátor. Jazyk C používá následující tzv. standardní typy proměnných:

- `int` → integer
- `short` → short integer
- `long` → long integer
- `float` → single precision real (floating point)
- `double` → double precision real (floating point)
- `char` → character variable (single byte)

Překladač kontroluje konsistenci typu všech proměnných použitých v kódu. Záměrem je předejít chybám, které mohou vzniknout tím, že by s proměnnými určitého typu byly prováděny operace náležející typu jinému. Výpočty, které se provádějí ve funkcích matematické knihovny `math.h` jsou zpravidla prováděny v aritmetice s pohyblivé řádové čárce s dvojnásobnou přesností (`double`, 8 bytů). Skutečný počet bytů může záviset na konkrétní HW architektuře a s jejich vývojem se v průběhu let může měnit. Ve cvičeních budete mít možnost tuto otázku probrat podrobněji. Funkci `printf` můžeme použít nejenom k tisku textu, ale i k tisku hodnot proměnných různých typů, jako jsou `integer`, `float`, `double` atp. a to způsobem, který je definován tzv. výstupním formátem. Obecná syntaxe je

```
printf( "formát", "seznam proměnných" );
```

kde „formát“ specifikuje způsob konverze proměnných z binární formy, v níž jsou uloženy hodnoty proměnných v paměti počítače do tvaru, v němž budou viditelné po výpisu na výstupní zařízení a „seznam proměnných“ je soupis odpovídacích proměnných, které mají být vytištěny. Ukážeme si jna tomto místě pár užitečných konverzí, jako jsou

```
%nd    integer (volitelné n = počet sloupců; je-li hodnota je n pozic
vyplněno nulami)
%m.nf   float or double (volitelné m = počet sloupců,
                        n = počet desítkových míst)
%ns     string (volitelné n = počet sloupců)
%c      character
\n \t   pro výstup znaku nový řádek nebo znaku tabelátoru
\a      výstup zvuku "beep" (ring the bell)
```

Podrobnější informace o formátování jsou obsaženy v samostatné kapitole.

2.3. Cykly



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- použít cyklické příkazy jazyka C



Výklad

Většina programů obsahuje konstrukce, které umožňují provádět opakovaně, cyklicky, akce s nějakými daty. Existuje více způsobů, jak v jazyce C naprogramovat takové cykly. Dva z nejběžnějších příkazů cyklu jsou příkaz cyklu `while`

```
while (podmíněný výraz)           podmínka=podmíněný výraz
{
    Tělo cyklu...blok příkazů uvnitř cyklu while...
}
```

a příkaz cyklu `for`

```
for (výraz_1; výraz_2; výraz_3)
{
    Tělo cyklu...blok příkazů uvnitř cyklu for...
}
```

Příkaz `while` opakuje provádění těla cyklu tak dlouho, dokud je podmíněný výraz pravdivý (`true`). Jakmile se stane hodnota tohoto výrazu nepravdivá (`false`), opustí cyklus a pokračuje v provádění dalších příkazů následujících za tímto cyklem. Hodnota podmíněného výrazu se testuje na začátku cyklu. Cyklus se teď nemusí provést vůbec, je-li hned na začátku podmínka neplatná. Jako podmíněný výraz může být užít jakýkoliv logický výraz vyjadřující nějakou podmínku.

Existuje o varianta cyklu `while` (`do-while`) s testováním podmínky na konci cyklu, která má tvar

```
do
{
    Tělo cyklu...blok příkazů uvnitř cyklu do-while...
}
while (podmíněný výraz);           podmínka=podmíněný výraz
```

Příkaz `do - while` opakuje provádění těla cyklu tak dlouho, dokud je podmíněný výraz pravdivý (`true`). Jakmile se stane hodnota tohoto výrazu nepravdivá (`false`), opustí cyklus a pokračuje v provádění dalších příkazů následujících za tímto cyklem.

Hodnota podmíněného výrazu se testuje na začátku cyklu, cyklus se tedy provede minimálně jednou.

Cyklus for je speciálním případem cyklu a pomocí cyklu while je ho možno přepsat takto:

```
výraz_1;
while (výraz__2)
{
    ... blok příkazů uvnitř cyklu...
    výraz__3;
}
```

Například často se setkáme s tímto cyklem:

```
i = počáteční_hodnota_i;
while (i <= i_max)
{
    ... blok příkazů uvnitř cyklu...
    i = i + přírůstek_i;
}
```

Tato struktura může být přepsána pomocí konstrukce for v mnohem čitelnější podobě takto:

```
for (i = počáteční_hodnota_i; i <= i_max; i = i + přírůstek_i;)
{
    ...blok příkazů...
}
```

Můžeme vytvořit i „nekonečné cykly“, např. for(;;). Jazyk C obsahuje i příkaz break, který umožňuje „vyskočit z „nekonečného cyklu“ pokud to naprogramujete při splnění nějaké podmínky. Například:

```
uhel_ve_stupních = 0;
for (;;)
{
    ...blok příkazů...
    uhel_ve_stupních = uhel_ve_stupních + 10;
    if (uhel_ve_stupních == 360) break;
}
...první příkaz následující za koncem cyklu...
```

O opuštění cyklu rozhoduje podmínění příkaz if, který se dotazuje, zda uhel_ve_stupních je roven 360; pokud ano (true), cyklus je opuštěn a program pokračuje na první příkaz uvedené za koncem cyklu.

Kromě příkazu break existuje ještě jedna možnost, jak přerušit provádění těla cyklu. Je to příkaz continue, kterým však neopustíme cyklus úplně, ale pouze skočíme na konec těla cyklu a pokračujeme v provádění cyklu od začátku těla cyklu.

2.4. Přepínač switch



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- používat přepínač switch jazyka C



Výklad

Přepínač switch obsahuje několik návěstí (case) s celočíselnou hodnotou, nebo s intervalem zapsaným ve formě, menší... větší (např. 1... 5 nebo 'a'... 'e'). Mezi třemi tečkami a hodnotami menší a větší vždy vynechávejte mezery.

```
switch (vyraz)
{
    case hodnota:
        prikazy
    case hodnota:
        prikazy
    default:
        prikazy
}
```

Na základě argumentu za klíčovým slovem switch přeskočí program na návěstí se stejnou hodnotou jakou má výraz v kulatých závorkách za switch a pokračuje vykonáváním příkazů za ním.

Přepínač switch může obsahovat návěstí default, na které program skočí tehdy, když argument za klíčovým slovem switch neodpovídá hodnotě za žádným návěstím case. Default není povinné a může se použít kdekoliv (první před všemi case, mezi nimi i jako poslední).

Příkaz break je speciální příkaz, který způsobí „vyskočení“ z těla příkazu switch (a také cyklů, viz dále). Program pak pokračuje ve vykonávání příkazů za blokem switch. V další ukázce jsou uvedeny příklady dvou programů s příkazem switch, přičemž v jednom se pracuje s proměnnou typu int a v druhém s typem char, avšak funkce programu je navenek stejná.

PříkladCo jsem stiskl s **char**

```
#include <stdio.h>
int main(void)
{
    char c;
    printf("Stiskni klavesu: ");
    scanf("%c",&c);
    switch (c)
    {
        case '1': printf ("Stiskl jsi klavesu 1\n");
                  break
        case '2': printf ("Stiskl jsi klavesu 2\n");
                  break;
        case '3': printf ("Stiskl jsi klavesu 3\n");
                  break;
        default: printf ("Jina klavesa\n");
                  break;
    }
    system("PAUSE");
    return 0;
}
```

Co jsem stiskl s **int**

```
#include <stdio.h>
int main(void)
{
    int i;
    printf("Stiskni klavesu: ");
    scanf("%d",&i);
    switch (i)
    {
        case 1: printf ("Stiskl jsi klavesu 1\n");
                 break;
        case 2: printf ("Stiskl jsi klavesu 2\n");
                 break;
        case 3: printf ("Stiskl jsi klavesu 3\n");
                 break;
        default: printf ("Jina klavesa\n");
                 // break; není nutné
    }
    system("PAUSE");
    return 0;
}
```

2.5. Symbolické konstanty



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- popsat funkci symbolických konstant



Výklad

Jazyk C umožňuje definovat konstanty jakéhokoliv typu užitím direktivy „**#define**“. Její syntaxe je jednoduchá, například:

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

Zde se definují konstanty `ANGLE_MIN` a `ANGLE_MAX` a nastavují se jejich hodnoty na 0 a 360. Céčko rozlišuje mezi malými a velkými písmeny v názvech proměnných resp. konstant. Je vhodné používat pro názvy globálních konstant velkých písmen.

2.6. Podmíněné výrazy



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- popsat a použít podmíněné výrazy v jazyce C



Výklad

Podmínky neboli podmíněné výrazy se používají v příkazech `if` a `while`, například:

```
if (podmínka_1)
{
    ...blok příkazů které se provádějí, je-li podmínka_1 true...
}
else if (podmínka_2)
{
    ...blok příkazů které se provádějí, je-li podmínka_2 true...
}
else
{
    ... blok příkazů které se provádějí v ostatních případech...
}
```

Další varianty takového příkazu se odvodí z uvedeného schématu buď vypuštěním některých větví, nebo přidáním dalších podmínek. Tímto způsobem dosáhneme rozdělení určité části programu do různých větví, z nichž se ta či ona provede v závislosti na pravdivosti podmínky, která „hlídá“ vstup do příslušné větve programu

Podmínky resp. podmíněné výrazy jsou logické operace vznikající při porovnávání různých proměnných nebo hodnot (nejlépe téhož typu) užitím relačních operátorů:

<	menší než
<=	menší než nebo rovno
==	rovno
!=	nerovno
>=	větší nebo rovno
>	větší než

a logických (boolských) operátorů

&& a ... logický součin

|| nebo ... logický součet
! ne ... negace

Další způsob rozvětvení programu umožňuje příkaz switch (přepínač), který jsme si již ukázali:

```
switch (výraz)
{
    case konstantní_výraz_1:
    {
        ...blok příkazů...větev_1
        break;
    }
    case konstantní_výraz_2:
    {
        ...blok příkaz... větev_2
        break;
    }
    default:
    {
        ...blok příkaz... implicitní_větev
    }
}
```

Odpovídající větev tvořená příslušným blokem příkazů se vykoná tehdy, jestliže hodnota výrazu se rovná hodnotě konstantního výrazu „hlídající“ vstup do příslušné větve (začíná slovem case). Příkaz break zajišťuje, že při výběru určité větve se po jejím provedení nebude pokračovat v provádění následujících větví. Pokud by bylo třeba následující větve provést, vypustí se v těchto větvích příkazy break až po větev, kterou už nechceme provést.

2.7. Pointery neboli ukazatele



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- popsat a použít ukazatele v jazyce C



Výklad

Všechny proměnné programu jsou uloženy v paměti. Příkazy

```
float x;  
x = 6.5;
```

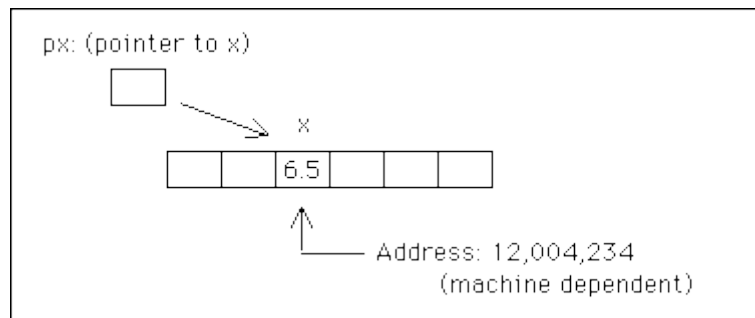
vyžadují, aby překladač zarezervoval 4 byty paměti pro proměnnou x, a zapsal do ní binárně hodnotu 6.5.

Céčko dovoluje programátorovi „nahlížet na a měnit“ přímo místa v paměti. To dává jazyku značnou flexibilitu a sílu, ale také ztěžuje život začátečníkům a vytváří prostředí pro provedení nebezpečných operací.

Adresu (místo v paměti) jakékoliv proměnné můžeme zjistit pomocí operátoru "&" jeho umístěním před jméno proměnné. Tak výraz „&x“ znamená adresu x. Céčko nám umožňuje navíc definovat nový typ proměnné, která se nazývá pointer (česky ukazatel). Ta může obsahovat adresu nějaké proměnné (tj. jakoby na ni „ukazovat“). Takovou proměnnou definujeme tím, že před její název v deklaraci umístíme znak hvězdičky „*“. Například příkazy:

```
float x;  
float* px;  
x = 6.5;  
px = &x;
```

definují proměnnou px jako pointer na proměnnou typu float, a nastavují její hodnotu rovnou adrese proměnné x, viz (Obr. 1).



Obr. 1 Použití ukazatele na proměnné

Obsah místa v paměti, na něž ukazuje pointer, dostaneme použitím operátoru hvězdička „*” (kterému se proto říká operátor dereference). Výraz `*px` tedy označuje hodnotu `x`.

Céčko umožňuje provádět aritmetické operace s pointery, například přičítat k nim 1. Musíme však dát pozor na to, že jednotkou v pointerové aritmetice je velikost (v bajtech) objektu, na který pointer ukazuje. Například, je-li `px` pointer na proměnnou typu `float`, pak výraz `px + 1` neodkazuje na další byte v paměti ale na místo dalšího objektu `float` tj. o 4 byty dále; pokud by `x` byl typu `double`, pak `px + 1` bude odkazovat na místo o 8 bytů dále (8 je velikost `double`), atd. Jen pokud je `x` typu `char` bude `px + 1` odkazovat skutečně na bezprostředně v paměti následující byte.

Takže v posloupnosti příkazů

```
char* pc;
float* px;
float x;
x = 6.5;
px = &x;
pc = (char*) px;
```

znamená v posledním řádku výraz `(char*)` „převedení“ typu (angl. tzv. „cast“), způsobující konverzi datového typu na jiný typ. Oba pointery `px` i `pc` odkazují na stejné místo v paměti -- adresu `x` - však `px + 1` a `pc + 1` odkazují k různým místům v paměti.

Podívejme se na následující program

```
void main()
{
    float x, y;           /* x a y jsou typu float */
    float *fp, *fp2;      /* fp a fp2 jsou ukazatele na proměnné float */
    x = 6.5;              /* x má nyní hodnotu 6.5 */
                           /* vytiskneme obsah a adresu of x */
    printf("Value of x is %f, address of x %ld\n", x, &x);
    fp = &x;              /* fp nyní ukazuje na adresu x */
    /* vytiskneme fp */
    printf("Hodnota v buňce paměti o adrese fp je %f\n", *fp);
    *fp = 9.2;            /* změněme obsah paměti */
    printf("New value of x is %f = %f \n", *fp, x);
    *fp = *fp + 1.5;       /* provede výpočet */
    printf("Final value of x is %f = %f \n", *fp, x);
    /* přeneseme proměnné */
    y = *fp;
    fp2 = fp;
    printf("Hodnota přenesená do y = %f a fp2 = %f \n", y, *fp2);
}
```

Po spuštění programu si všimněte, že zatímco hodnota pointeru (vytiskneme-li ji funkcí printf) je zpravidla hodně velké celé číslo, označující konkrétní místo v paměti, pointry nejsou proměnné typu integer, jsou to hodnoty úplně jiného typu.

2.8. Pole



Čas ke studiu:

1 hodina.



Cíl:

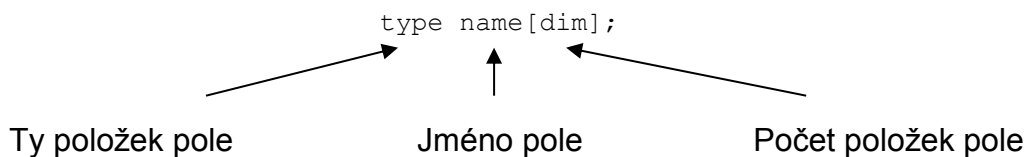
Po prostudování tohoto odstavce budete umět:

- pochopíte a budete umět pracovat s poli v jazyce C

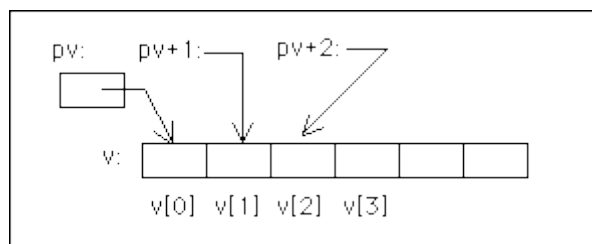


Výklad

Pole je posloupnost hodnot stejného typu uložená v po sobě jdoucích místech paměti. K jednotlivým položkám pole se přistupuje pomocí indexu, který se zapisuje do hranatých závorek. V Céčku je možno vytvořit pole jakéhokoliv typu. Syntaxe je velmi jednoduchá:



Prvním indexem pole je vždy 0. Céčko zachází se jménem pole (v naší definici name) tak, jako kdyby to byl pointer na první element pole, tj. na prvek s indexem 0. Toto je důležité si uvědomit při operování s přístupem ke složkám pole nepoužívajícím indexy ale pointery. Neboť je-li v nějaké pole, znamená `*v` totéž jako `v[0]`, `*(v+1)` je totéž jako `v[1]`, viz (Obr. 2).



Obr. 2 Užití pointeru pro pole

Podívejme se na následující kód:

```
#define SIZE 3
void main()
{
    float x[SIZE];
    float *fp;
    int i;

    /* inicializace pole x */
}
```

```
/* konverze ("cast") i          */
/* na ekvivalentní float      */
for (i = 0; i < SIZE; i++)
    x[i] = 0.5*(float)i;

/* print x                      */
for (i = 0; i < SIZE; i++)
    printf("  %d  %f \n", i, x[i]);

/* nastavení fp na adresu pole x */
fp = x;

/* print cestou přes pointer    */
/* prvky x jsou seřazeny za sebou */
/* souvisle v paměti          */
/* *(fp+i) odkazuje na obsah místa */
/* v paměti s adresou (fp+i) or x[i] */
for (i = 0; i < SIZE; i++)
    printf("  %d  %f \n", i, *(fp+i));
}
```

(Výraz "i++" znamená v Céčku zkratku pro zápis "i = i + 1", zatím jsme o tom nemluvili.) Protože x[i] označuje i-tý element pole x a fp = x ukazuje na začátek pole, pak *(fp+i) je obsah místa paměti vzdálený i pozic za fp, tj. x[i].

2.9. Pole znaků



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- pracovat s poli znaků (řetězci) v jazyce C



Výklad

Řetězcová konstanta (string constant), jako je například „I am a string“ je pole znaků. V Céčku je vnitřně reprezentována posloupností znaků v kódu ASCII. To znamená znaky: „I“, mezera, „a“, „m“,... jak jsou uvedeny v tomto řetězci, a zakončena speciálním ukončovacím znakem „\0“ tak, aby algoritmy pracující s řetězcem mohly určit, kde je konec řetězce.

Řetězcové konstanty se hodně využívají v příkaze `printf`, pro komentáře k vypisovaným hodnotám, aby se výstup výsledků učinil srozumitelnější;

```
printf("Salvete amici\n");
printf("Hodnota proměnné a je: %f\n", a);
```

Řetězcové konstanty mohou být označeny identifikátory. Céčko dovoluje vytvářet proměnné typu `char`, které obsahují jeden znak a jejichž délka je jeden byte. Řetězcová konstanta je uložena v poli znakového typu, jeden ASCII znak zabírá jednu složku pole. Je si třeba pamatovat že řetězec musí být ukončen koncovým znakem „\0“, takže v poli budeme potřebovat jeden znak navíc, než požadujeme pro viditelné znaky řetězce!

Céčko nemá žádný operátor, který by mohl operovat s celým řetězcem najednou. Řetězce jsou tak zpracovávány buď prostřednictvím ukazatelů, nebo pomocí podprogramů obsažených ve standardní knihovně `string` (`string.h`). Použití pointerů je poměrně snadné, protože jméno pole je ukazatelem na první prvek pole. Podívejme se na následující program:

```
void main()
{
    char text_1[100], text_2[100], text_3[100];
    char *ta, *tb;
    int i;
    /* message bude posloupnost znaků; inicializujeme ji */
    /* jako konstantní řetězec "... " */
    /* kompilátor rozhodne o jeho velikosti, použijeme-li [] */
    char message[] v message: %s\n", message);

    /* kopírujeme message do text_1 pomocí indexace */
```

```

i=0;
while ( (text_1[i] = message[i]) != '\0' )
    i++;
printf("Text_1: %s\n", text_1);
/* teď pomocí pointerů */
ta=message;
tb=text_2;
while ( ( *tb++ = *ta++ ) != '\0' )
    ;
printf("Text_2: %s\n", text_2);
}

```

Standardní knihovna "string" obsahuje řadu užitečných funkcí pro manipulaci s řetězci. Některé z nejpoužítejších funkcí jsou

```

char *strcpy(s,ct)      -> kopírování ct na s, včetně '\0'; vrací s
char *strncpy(s,ct,n)   -> kopíruje n znaků ct do s, vrací s
char *strncat(s,ct)     -> připojuje ct na konec s; vrací s
char *strncat(s,ct,n)   -> připojuje n znaků ct na konec s, ukončuje spojení
znakem with '\0'; vrací s
int strcmp(cs,ct)       -> porovnává cs s ct; vrací 0 když cs=ct,
                        hodnotu <0 když cs<ct, hodnotu>0 když cs>ct
char *strchr(cs,c)      -> vrací pointer na místo prvního výskytu c
                        v cs nebo NULL když se tam nevyskytuje
size_t strlen(cs)       -> vrací délku cs
(s a t jsou char*, cs a ct jsou const char*, c je char konvertovaný na typ
int a n je int.)

```

Podívejme se na následující kód, v němž je použito některých z těchto funkcí:

```

#include < string.h>
void main(){
    char line[100], *sub_text;
    strcpy(line,"Salve, jsem retezec"); /* inicializace řetězce */
    printf("Line: %s\n", line);
    strcat(line," kdo jsi?");          /* přidáme na konec řetězce */
    printf("Line: %s\n", line);
    /* nelezeme délku řetězce pomocí strlen */
    printf("Delka radku: %d\n", (int)strlen(line));
    /* najdeme výskyt subřetězců */
    if ( (sub_text = strchr ( line, 'W' ) )!= NULL )
        printf("Retezec zacinajici s \"W\" ->%s\n", sub_text);
    if ( ( sub_text = strchr ( line, 'w' ) )!= NULL )
        printf("Retezec zacinajici s \"w\" ->%s\n", sub_text);
    if ( ( sub_text = strchr ( sub_text, 'u' ) )!= NULL )
        printf("Retezec zacinajici s \"w\" ->%s\n", sub_text);
}

```

2.10. Periferní (I/O==input/output) operace



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- pracovat s vstupně výstupními zařízeními pomocí jazyka C



Výklad

□ I/O na úrovni jednotlivých znaků

Céčko poskytuje prostřednictvím svých knihoven různorodé I/O operace. Chceme-li provádět periferní operace „znak po znaku“, čte funkce `getchar()` jeden znak z tzv. standardního vstupu označovaného `stdin`, a funkce `putchar()` zase zapisuje jeden znak na standardní výstupní zařízení `stdout`. Například uvažme program:

```
#include <stdio.h>
void main()
{
    int i, nc;
    nc = 0;
    i = getchar();
    while (i != EOF) {
        nc = nc + 1;
        i = getchar();
    }
    printf("Počet znaků v souboru = %d\n", nc);
}
```

Tento program počítá počet znaků čtených ze vstupního řetězce. Program čte znaky ze standardního vstupu `stdin` (klávesnice), a jako výstupní zařízení používá standardní výstupní zařízení `stdout` pro výstup, a chybová hlášení zapisuje na standardní zařízení `stderr`. EOF je speciální návratová hodnota, definovaná v `stdio.h`, kterou `getchar()` vrací, když zjistí znak end-of-file při čtení znaků souboru. Tato hodnota je závislá na systému, ale kompilátor Céčka tento fakt před uživatelem skrývá definicí proměnné EOF. Takže program čte znaky ze zařízení `stdin` a přičítá jedničku k proměnné `nc`, dokud nenarazí na "end of file".

Předcházející příklad bychom mohli zapsat také takto:

```
#include <stdio.h>
void main()
{
    int c, nc = 0;
    while ( (c = getchar()) != EOF ) nc++;
}
```

```
printf("Number of characters in file = %d\n", nc);
}
```

Céčko umožňuje psát velmi kkrátké „zhuštěné“ zápisy kódu, avšak na úkor čitelnosti, zejména pro začátečníka.

Závorky () v příkazu (c = getchar()) říkají: proved' getchar() a přiřaď výsledek c předtím, než ho porovnáš s EOF; závorky jsou zde nezbytné. Připomeňme si že nc++ (a ostatně také ++nc) je jiný způsob zápisu nc = nc + 1. (Rozdíl mezi těmito zápisy s operátorem ++ je, že v zápise ++nc, je nc zvětšeno předtím než je použito, zatímco v zápise nc++, je nc použito předtím než je zvětšeno. V našem příkladu by to však nehrálo roli.) Tento zápis je kompaktnější (není to vždy výhoda), avšak často je efektivněji překládán překladačem, což je největší důvod ospravedlňující jeho existenci. Následující příklad počítá i řádky souboru, které jsou odděleny znakem '\n'.

```
#include <stdio.h>
void main()
{
    int c, nc = 0, nl = 0;
    while ( (c = getchar()) != EOF )
    {
        nc++;
        if (c == '\n') nl++;
    }
    printf("Number of characters = %d, number of lines = %d\n",
          nc, nl);
}
```

❑ Složitější vstupně-výstupní operace

Viděli jsme, jak printf zapisuje formátovaný výstup na stdout. Opačným směrem pro čtení ze stdin analogicky funguje scanf. Syntax příkazu je

```
scanf("format string", variables);
```

což připomíná printf. Formátovací řetězec může obsahovat mezery nebo tabelátory, normální ASCII znaky, odpovídající těm zapisovaným na stdin, a konverzní specifikace jako ve funkci printf.

Existují i analogické příkazy pro „čtení“ nebo „zápis“ informace obsažené v řetězci znaků. Jde vlastně o konverzi prováděnou v paměti:

```
sprintf(string, "format string", variables);
sscanf(string, "format string", variables);
```

Argument "string" je jméno (tj. pointer na) pole z, do nichž chcete zapsat nebo z nichž chcete číst konvertovanou informaci.

❑ Operace čtení a zápisu ze souborů a do souborů znaků

Podobné příkazy také existují pro čtení a zápis do a ze souborů. Tyto příkazy jsou ukázaný v následující ukázce.

```
#include <stdio.h>

FILE *fp;
```



```
fp = fopen(name, mode);
fscanf(fp, "format string", variable list);
fprintf(fp, "format string", variable list);
fclose(fp );
```

Vidíme, že ještě jeden parametr navíc, a to fp představující soubor.

Při práci se soubory proto musíme

1. definovat a lokální "pointer" typu FILE, který je definován v <stdio.h>(FILE *fp);
2. otevřít ("open") soubor a přiřadit mu tento pointer pomocí funkce fopen
3. provést I/O operace použitím fscanf a fprintf
4. uzavřít soubor pomocí příkazu fclose

Argument "mode"v příkazu fopen specifikuje režim, případně nastavení pozice v otvíraném souboru: "r" pro čtení (reading), "w" pro zápis (writing), a "a" pro nastavení pozice na konec (appending) souboru. Zkuste si tento příklad:

```
#include <stdio.h>

void main()
{
    FILE *fp;
    int i;
    fp = fopen("foo.dat", "w");          /* otevři foo.dat pro zápis */
    fprintf(fp, "\nUkazkový příklad\n\n"); /* napiš nějaký text */
    for (i = 1; i <= 10 ; i++)
        fprintf(fp, "i = %d\n", i);
    fclose(fp);                          /* zavři soubor */
}
```

Po provedení programu se podívejte, co bylo zapsáno do souboru foo.dat.

2.11. Funkce



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- deklarovat a použít funkce
- deklarovat prototypy funkcí



Výklad

Funkce umožňují rozdělit složité programy na menší části, které můžeme snadněji číst, napsat i udržovat. Již jsme se seznámili s použitím funkcí v případě vstupně výstupních (I/O) funkcí, matematických funkcí a nakonec i s „funkcí“ main. Podívejme se nyní na další knihovní funkce a také na to jak napsat a používat svoje vlastní.

▣ Volání Funkce

Volání funkce v Céčku znamená odkázat se na její jméno doplněné o vhodné argumenty. Překladač C kontroluje kompatibilitu mezi argumenty dodanými při vyvolání funkce a mezi argumenty specifikovanými při definici funkce.

Knihovní funkce (tzn. Funkce dodané ve standardních knihovnách s překladačem) nejsou obecně dostupné ve zdrojovém kódu. Kontrola typu argumentů se uskutečňuje s pomocí hlavičkových souborů (jako je např. `stdio.h`) které obsahují veškerou informaci potřebnou pro tuto kontrolu. Jak jsme již viděli, chceme-li použít standardní matematickou knihovnu, musíme ji zařadit do programu pomocí příkazu `#include` odvolávajícího se na `math.h` tj. pomocí příkazu `#include <math.h>`. Ten je uveden na začátku souboru obsahujícího náš kód. Nejčastěji používané hlavičkové soubory jsou

```
< stdio.h>  -> definuje I/O podprogramy
< ctype.h>  -> definuje podprogramy pro práci se znaky
< string.h> -> definuje podprogramy pro práci s řetězci znaků
< math.h>   -> definuje matematické podprogramy
< stdlib.h> -> definuje funkce pro kverzi čísel, alokaci paměti, atp.
< stdarg.h> -> definuje knihovny pro práci s funkcemi s nožností proměnného
počtu argumentů
< time.h>   -> definuje funkce pro operace s časem
```

Existují i další knihovny, například

```
< assert.h> -> definuje diagnostické podprogramy
```

```
< setjmp.h> -> definuje volání nelokálních funkcí
< signal.h> -> definuje handlersy pro zpracování signálů
```

```
< limits.h> -> definuje konstanty pro int typy
< float.h>  -> definuje konstanty pro float typy
```

❑ Jak psát svoje vlastní funkce

Funkce, resp. její deklarace, má následující strukturu:

```
návratový-typ jméno-funkce ( nepovinný_seznam_parametrů )
{
    ...lokální-deklarace...

    ...příkazy...

    return návratová-hodnota;
}
```

Není-li uveden návratový-typ, předpokládá Céčko implicitně int. Datová-hodnota musí být deklarovaného typu. Funkce může provést nějakou úlohu i bez toho, že by vracela nějakou hodnotu. V takovém případě bude mít následující strukturu:

```
void jméno-funkce (nepovinný_seznam_parametrů)
{
    ... lokální-deklarace...
    ... příkazy...
}
```

Jako příklad použití funkce se podívejme na následující kód:

```
/* vložte hlav. Soubory pro všechny funcce v programu */

#include < stdio.h>
#include < string.h>
/* uveďte prototypy všech funkcí definovaných později než jsou volány
   aby překladač mohl testovat správné typy aktuálních parametrů
*/
int n_char(char string[]);
void main()
{
    int  n;
    char string[50];
    strcpy(string,"Salvete amici"); /* strcpy(a,b)kopíruje řetězec b
do a*/
    n = n_char(string);             /* voláme naši vlastní funkci */
    printf("Length of string = %d\n", n);
}

/* definice lokální funkce n_char */
int n_char(char string[])
{
    int n;                          /* lokální proměnná v této funkci */
    n = strlen(string);             /* strlen(a) vrací délku řetězce a */
    if (n > 50)
        printf("Řetězec je delší než 50 znaků\n");
    return n; /* vrací hodnotu integer proměnné n */
}
```

V Céčku jsou vždy při volání funkcí argumenty předávány hodnotou (by value). To znamená, že „dovnitř“ funkce jsou předány kopie hodnot argumentů zadaných funkci

jako parametry. Jakákoliv změna argumentu provedená uvnitř funkce je tedy provedena na její vnitřní „lokální“ kopii a neprojeví se na odpovídající hodnotě argumentu vně funkce. Aby bylo možno změnit hodnotu argumentu vně funkce, musí být funkci předána při jejím volání adresa argumentu. Přitom musí být funkci známo (provede se při definici funkce), že jde o adresu a na místě, kde se s argumentem operuje, musí být příslušné operace prováděny s adresou argumentu.

Jako příklad uveďme výměnu hodnot dvou proměnných. Nejdříve se podívejme, co se stane, když budeme předávat argumenty hodnotou:

```
#include <stdio.h>
/* Prototypování funkce exchange */
void exchange(int a, int b);
void main()
{
    /* POZOR ŠPATNÝ KÓD */
    int a, b;
    a = 5;
    b = 7;
    printf("hodnoty v mainu: a = %d, b = %d\n", a, b);
    exchange(a, b);    // ← toto je špatně
    printf("Nyní jsme zpět v mainu po „výměně“: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    printf(" Vystupujeme z funkce: ");
    printf("a = %d, b = %d\n", a, b);
}
```

Spustíte tento kód a pozorujte proměnné „a“ a „b“, které **nejsou** zaměněny! Pouze kopie argumentů jsou zaměněny. Správný způsob, kterým můžeme docílit výměny (exchange) je s použitím ukazatelů:

```
#include <stdio.h>
/* Prototypování funkce exchange */
void exchange ( int *a, int *b );
void main()
{
    /* SPRÁVNÝ KÓD */
    int a, b;
    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);
    exchange(&a, &b); // ← toto je dobře
    printf(" Vystupujeme z funkce: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange ( int *a, int *b )
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    printf(" Vystupujeme z funkce: ");
    printf("a = %d, b = %d\n", *a, *b);
}
```

```
}
```

Pravidla použití jsou následující:

- Používejte jako argumenty funkce normální proměnné, pokud funkce nemění hodnoty těchto argumentů, ukazatele na proměnné, které mají být parametry funkce, musíte užít jako argumenty tehdy, pokud má funkce změnit hodnoty těchto argumentů=proměnných.

2.12. Uživatelské a strukturované datové typy



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- definovat, vytvořit a použít, uživatelské a strukturované datové typy
- použít a popsat strukturu (struct)
- vytvořit vlastní datový typ (typedef)



Výklad

V jazyce C předdefinované datové typy (např. int, float atp.), s výjimkou pole, jsou tvořeny vždy jedinou hodnotu. Často je však třeba použít jiných, složitějších a aplikačně vhodnějších typů. Ty můžeme v Céčku vytvářet pomocí, tzv. strukturovaných typů. Pojem strukturovaný typ představuje celou třídu různých typů vytvářených podle společného mechanismu. Protože si pak takové konkrétní typy vytvářejí jednotliví programátoři k určitému konkrétnímu použití, nazývají se takové typy také uživatelské typy, strukturované typy, nebo krátce struktury.

□ Datové struktury

Datová struktura (krátce struktura) umožňuje spojit několik různých datových typů do jednoho. Strukturu definujeme takto:

```
struct [jméno_struktury] {
    typ jméno_položky_1;
    typ jméno_položky_2;
    ...
} [seznam_proměnných];
```

Ukažme si příklad struktury OSOBA. S definicí struktury rovnou vytvoříme dvě proměnné typu OSOBA, které se budou jmenovat Martin a Pavla.

```
struct OSOBA {
    unsigned int rok_narozeni;
    int pohlavi;
    char jmeno[20];
    char prijmeni[20];
} Martin, Pavla;
```

Definice struktury vždy začíná klíčovým slovem struct. Poté může, ale také nemusí, být definováno jméno struktury. V našem případě jméno definováno je a je to

OSOBA. Pokud není definováno jméno struktury, není možné později vytvořit další proměnné této struktury, ani ji použít jako parametr funkce. Takovou konstrukci pak můžeme použít jen pro definici struktury proměnné nebo více proměnných (Pavla, Martin), kterou musíme uvést současně s definicí struktury.

V těle struktury (mezi závorkami {}) jsou deklarovány proměnné, nazývané též položky, které bude struktura obsahovat. Položkami mohou být i jiné struktury, nebo ukazatel na vlastní strukturu (jakoby „na sebe sama“). Struktura však nemůže obsahovat sebe samu jako proměnnou. Se strukturami jako s celkem lze provádět pouze operaci přiřazení =, případně testování na rovnost nebo nerovnost (operace =, !=). Tím se obsah jedné struktury zkopíruje do jiné struktury (téhož typu!!), případně porovnává vcelku celá datová struktura s jinou datovou strukturou stejného typu.

Dále lze přistupovat k jednotlivým položkám struktury. Pro přístup ke položkám struktury se používají operátory • (tečka) a -> (šipka). Druhý operátor se používá v případě, že pracujeme s ukazatelem na strukturu.

```
OSOBA p, *ukazatel;
ukazatel = &p;
p.rok_narozeni = 20;          /* přiřazení 20 do položky rok_narozeni */
(*ukazatel).rok_narozeni = 20; /* stejné přiřazení, jen přes ukazatel */
ukazatel->rok_narozeni = 20;  /* přehlednější přiřazení přes ukazatel */
```

Použité závorky jsou nutné, aby bylo jasné, že operátor dereference * (hvězdička) pracuje s proměnnou ukazatel, a ne s proměnnou rok_narozeni.

Pouhým vytvořením struktury však nevytváříme nový datový typ. Ten vznikne jedině tehdy, jestliže strukturu při definici přidělíme jméno, tj. pojmenujeme-li ji a zároveň použijeme k její definici klíčové slovo **typedef**, pomocí něhož definujeme nové typy a to i typy jiného druhu než ty, které vzniknou pomocí definice struktury. Bude uvedeno dále.

Struktury s výhodou používáme při práci se soubory, kdy pomocí struktury definujeme tzv. záznamy souborů, které zapisujeme do nebo čteme ze souboru jako celek jedním příkazem zápisu nebo čtení. Podrobněji si to ukážeme v příkladu se soubory. Před názvem struktury je vždy nutné používat klíčové slůvko **struct** jak při deklaraci parametrů, tak při definici proměnných struktur.

Pokud předáváme strukturu jako argument, kopíruje se celá struktura z proměnné, která označuje tuto strukturu. Pokud má struktura velkou délku, může to zpomalovat chod programu, pokud se takových operací provádí hodně. Proto je výhodnější „ponechat data, tj. celou strukturu na místě“ a použít jako argument ukazatel na strukturu.

□ Definování typu pomocí typedef

Pomocí typedef se vytvářejí uživatelské datové typy. Syntaxe příkazu tato:

```
typedef definice_typu identifikator;
```


Pro definici nového typu zastoupeného symbolem definice_typu můžeme využít například strukturu nebo nějaký základní typ jazyka C, identifikátor je pojmenování nového datového typu. Podívejme se na následující příklad:

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

```
typedef int integer;
```

```
typedef definice_typu identifikator;
```

```
typedef int integer;
int main(void) {
    int a, b;
    integer c, d;
    a = 5;
    b = a + 4;
    printf("a = %i, b = %i\n", a, b);
    d = (integer) a;
    c = d + 5;
    printf("d = %i, b = %i\n", (int) d, (int) c);
    return 0;
}
```



Všimněte si přetypování proměnné a při přiřazování její hodnoty do proměnné d na typ integer, a také přetypování proměnných d a c na typ int ve funkci printf(). Funkce printf() totiž nemůže očekávat typ integer, neboť jej norma jazyka C ani nezná. V našem příkladě je definice typu integer je sice jednoduchá, a v podstatě jen přejmenovává existující datový typ. Takovéto „přejmenovávání“ datových typů ale nepřináší žádné výhody a činí program nečitelnějším. Vraťme se nyní ke strukturám a podívejme se na vytváření nového datového typu pomocí struktury. Ukážeme si to na následujícím příkladu.


```
#include <stdio.h>
typedef struct pokus {
    int x,y;
    char text[20];
} Pokus;
int main(void)
{
    struct pokus a = { 6, 6, "sest" };
    Pokus b = { 4, 5, "Ahoj" };
    a.x = 0;
    b.x = 6;
    return 0;
}

#include <stdio.h>
#include <stdlib.h>
typedef struct clen
{
    char jmeno[25];
    int vek;
    float vaha;
    struct clen *dalsi;
} clen;
int getVek( clen S )
{
    return( S.vek);
}
void setVek( clen *S, int vek )
{
    (*S).vek = vek; }
void setDalsiClen ( clen *S, clen *DalsiClen )
{
    (*S).dalsi = DalsiClen; }
clen *getDalsiClen( clen *S )
{
    return ((*S).dalsi); }
NastavHodnotuVekuDalsiho ( clen *S, int vek )
{
    (*(S->dalsi)).vek = vek; }
int main(int argc, char *argv[])
{
    clen C, dalsiC;
    clen *dalsi = NULL;
    setVek( &C, 1); /* stejné jako C.vek = 1; */
    printf ("vek %d\n", getVek(C));
    setDalsiClen (&C,&dalsiC);
    NastavHodnotuVekuDalsiho( &C,5);
    dalsi = getDalsiClen( &C );
    printf ("vekDalsiho %d\n", (*dalsi).vek );
    system("PAUSE");
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {          /* použijeme */
```

Složitější příklad

Definujeme nový strukturovaný typ clen obsahující čtyři položky.

K tomuto typu vytvoříme tzv. přístupové funkce a to pro položku vek. Funkce getVek vrací aktuální hodnotu položky vek a setVek nastavuje její hodnotu podle zadaného parametru vek.

Podobně funkce setDalsiClen a getDalsiClen nastavují položku další obsahující adresu propojené další datové struktury -proměnné stejného typu.

A funkce NastavHodnotuVekuDalsiho dosazuje do položky vek vnitřní (odkazované) struktury člen zadanou hodnotu vek.

Ve funkci main je ukázáno použití těchto funkcí na strukturách clen, dalsiC a pomocné proměnné další.

Tento příkaz vytiskne hodnotu 5

Další příklad evidence nosičů DVD

Definujeme dva strukturované typy DATE a TIME pro uložení položek data a času

```
    int den; int mesic; int rok;
    } DATE ;
typedef struct {
    int hod; int min; int sec;
    } TIME ;
typedef struct {
    TIME t; DATE d;
    } TIMEDATE;
typedef enum {          /* použijeme */
    Volne, Rezervovane, Pujcene, Ztracene}
    STAV;
typedef struct {        /* použijeme */
    char name[30];
    DATE vyp_kdy;
    DATE do_kdy;
    int potermínu;
    STAV stav;
    long InvCis;
    } ZAZNAM;
DATE SetDate( int d,int m, int r ){
    DATE Datum;
    Datum.den=d;
    Datum.mesic=m;
    Datum.rok=r;
    return (Datum);
}

DATE d; TIME t; TIMEDATE td;
ZAZNAM z; char *str;

int main(int argc, char *argv[])
{
    printf ("size of DATE %d\n", sizeof td);
    // vyplníme záznam
    str = strcpy( &z.name[0], "Rude housle");
    z.vyp_kdy = SetDate(5,12,2007 );
    z.do_kdy = SetDate(12,12,2007);
    z.potermínu = 0;
    z.stav = Pujcene;
    z.InvCis = 1234567890;
}
```

2.13. Uživatelské typy enum a union



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- deklarovat a používat výčtový uživatelský typ enum a strukturální uživatelský typ union



Výklad

□ Výčtový typ enum

Tzv. výčtový typ **enum** umožňuje vytvářet proměnné, které obsahují hodnoty reprezentované jmény konstant vyjmenovanými („je proveden jejich výčet“) při deklaraci výčtového typu. Konkrétní výčtový typ se vytváří opět pomocí typedef. Lze vytvořit buďto jen výčtový typ enum (stejně jako jenom strukturu), nebo pomocí typedef definovat nový datový typ.

```
enum [jmeno] {
seznam konstant
...
} [promenne];
Příklad vytvoření výčtového typu:
enum kalendar {
leden = 1, unor, brezen, duben, kveten, červen, červenec, srpen, zari,
rijen, listopad, prosinec
} a, b;
...
a = unor;
...
if (b == duben) { atd...
```

V příkladu jsme vytvořili výčtový typ se jménem kalendar a definovali dvě proměnné „a“ a „b“. Konstanty výčtového typu (zde jsou zastoupeny jmény měsíců) jsou vždy celočíselné. Pokud konstantě nepřidáme hodnotu (jako leden=1), pak má hodnotu o jednotku vyšší, než konstanta předešlá (unor tedy odpovídá číslu 2, brezen 3 atd). Pokud nepřidáme hodnotu ani první konstantě, automaticky je jí přiřazena nula. Hodnoty lze přiřadit kterékoliv konstantě ve výčtovém typu. Opět platí, že následující konstanta, pokud nemá přiřazenou hodnotu, je o jednotku větší, než předcházející konstanta.

Hodnoty konstant uvedené ve výčtovém typu je i možné srovnávat s čísly (např. if(leden == 1)). V následujícím příkladu je vidět rafinované použití typu enum s konstrukcí typedef.

```
#include <stdio.h>
typedef enum {
    vlevo, vpravo, stred, center = stred
} zarovnani;
void tiskni_znak(char ch, zarovnani zr)
{
    switch (zr) {
        case vlevo:      printf("%c\n", ch);      break;
        case vpravo:     printf("%50c\n", ch);     break;
        case stred:      printf("%25c\n", ch);     break;
        default:         printf("%c%24c%25c\n", '?', '?', '?');
    }
}
int main(void)
{
    tiskni_znak('A', 50);
    tiskni_znak('X', vlevo);
    tiskni_znak('Y', center);
    tiskni_znak('Z', vpravo);
    return 0;
}
```

Výstup z programu:

?	?	?
X	Y	Z

V dalším příkladě se ukazuje použití výčtového typu k použití definice barev

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

enum barva{ cervena, modra, bila} barva_vlajky;

enum { cerv, modr, bil} barva_kalhot;

enum barva barva_triaka;

int getBarva( enum barva B )
{
    return( B);
}

int main(int argc, char *argv[])
{
    barva_kalhot=cerv;
    barva_vlajky=cervena;
    printf ("barva kalhot %d \n", barva_kalhot);
    barva_triaka= bila;
    printf ("barva triaka %d\n", barva_triaka);
    printf ("barva triaka a barva vlajky jsou ");
    if ( ((int)barva_kalhot)==((int)barva_vlajky))
        printf ("stejně\n"); else printf ("stejně\n");
    barva_vlajky=cervena;

    printf ("barva vlajky %d\n",
            (int) getBarva(barva_vlajky));
    system("PAUSE");
    return 0;
}
```

V první definici jsme zvolili jméno typu barva a vytvořili proměnnou barva_vlajky.

V druhé definici jsme jméno typu nezvolili (barva již použit nemůžeme) vytvořili jsme jedinečnou proměnnou barva_kalhot. Ve druhé definici nemůžeme také použít pro názvy barev stejná jména jako v prvním typu a zvolili jsme proto cerv, modr a bil.

Už také nemůžeme deklarovat další proměnnou jako u prvního typu kdy jsem ještě zvláštní definicí deklarovali proměnnou barva_triaka.

Všimněte si že ve funkci getBarva musíme před jméno parametru explicitně napsat

enum barva, protože barva není jméno typu (nedefinovali jsme ho pomocí typedef).

Při porovnávání barva_kalhot a barva_vlajky převádíme hodnot konstant na stejný typ,

Ale příkaz bude fungovat úplně stejně jako s podmínkou

```
if ( (barva_kalhot)==(barva_vlajky))
```

Takže překladač stejně „přehlídí“ že jde o typově různé výčty

□ Typ union

Syntaxe typu **union** je stejná jako syntaxe struct až na to, že místo klíčového slova struct se použije klíčové slovo union. Význam jednotlivých položek je stejný. Stejně je i jeho použití s konstrukcí typedef.

```
union [jmeno] {
    typ jmeno_polozky;
    typ jmeno_polozky;
    ...
} [promenne];
```

Rozdíl spočívá v tom, že struktura si vytvoří paměťové místo pro všechny položky, typ union zabírá v paměti jen tolik místa, kolik jeho největší položka. Z toho také vyplývá, že lze používat v jeden okamžik jen jednu položku. Kterou, to už závisí na programátorovi. Každá položka začíná na začátku paměti unionu. Položky unionu se v paměti překrývají.

Typ union může při programování ušetřit paměť, což by mohlo být zajímavé pouze pro procesory s malou operační pamětí. Navíc můžeme vytvořit funkci, která bude vracet různé datové typy jako typ union (viz příklad níže). Prvkem v union může být i struktura, nebo jiný typ union atp..

Příklad

```
#include <stdio.h>
#include <stdlib.h>
typedef union {
    int a;
    unsigned int b;
    float c;
    char s[10];
} U4;
void PrintUnion (U4 U){
    printf ("\n int=%d   uint=%u\n   float=%f   string=%s\n",U.a,U.b,U.c,U.s);
}
int main(int argc, char *argv[])
{
    U4 U;
    U.s[0] = 'A'; U.s[1] = 'h'; U.s[2] = 'o'; U.s[3] = 'j'; U.s[4] = ' ';
    U.s[5] = 'K'; U.s[6] = 'a'; U.s[7] = 'm'; U.s[8] = 'o'; U.s[9] = '\0';
    PrintUnion ( U );
    U.a = -1;
    PrintUnion ( U );
    U.a = 12L;
    PrintUnion ( U );
    system("PAUSE");
    return 0;
}

/*      výsledky
int=1785686081   uint=1785686081
float=723563990101199020000000000.000000   string=Ahoj Kamo

int=-1   uint=4294967295
float=-1.#QNAN0   string=           Kamo

int=12   uint=12
float=0.000000   string=+
Pokracujte stisknutím libovolné klávesy...
*/
```



2.14. Argumenty (parametry) programu v příkazovém řádku



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- deklarovat a používat výčtový uživatelský typ enum a strukturální uživatelský typ union



Výklad

Chceme-li zadat do programu nějaké parametry při jeho volání přímo z příkazového řádku, kterým spouštíme program, děláme to prostřednictvím tak zvaných parametrů programu. Ty jsou použity k tomu, aby změnily chování programu podobně, jako odlišné parametry funkce způsobují změnu v její činnosti. Parametry zadané v příkazovém řádku při spouštění programu je nutno přenést na nějaké proměnné v programu, které představují kopie takto zadaných hodnot a používají se pak při běhu programu na potřebných místech.

To, že program počítá s možností zadání vstupních parametrů při volání je třeba toto naznačit v příkazu main zadáním v stupních parametrů. Tyto parametry jsou dva: pole řetězců znaků (array of character strings), zpravidla nazývané `argv`, a celé číslo(integer), obvykle nazývané `argc`, které zadává počet řetězců v poli (tj. počet vstupních parametrů). Specifikace v mainu pak vypadá takto

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

(Syntaxe `char** argv` deklaruje že `argv` je pointer na pointer znaků, neboli pointer na zankové pole, jinak též na pole znakových řetězců. Tuto specifikaci bychom také mohli – asi čitelněji – zapsat jako `char* argv[]`. Jak s tím pracovat si objasníme dále.

Řekněme, že chceme spustit program `prog` s nějakými parametry. Pole `argv` bude obsahovat veškerou informaci zapsanou na příkazový řádek včetně názvu volaného programu. Parametry které jsou přečteny jako řetězce, jsou odděleny mezerami. Argument `argc` se rovná celkovému počtu řetězců zapsaných na řádku, tj. počtu argumentů plus jedna. Například když jsme napsali

```
prog -i 2 -g -x 3 4
```

program dostane

```
argc = 7
argv[0] = "prog"
argv[1] = "-i"
argv[2] = "2"
argv[3] = "-g"
```

```
argv[4] = "-x"
argv[5] = "3"
argv[6] = "4"
```

V tomto okamžiku jsou všechny argumenty, včetně numerických, ve formě řetězců (string). Proto je třeba je v programu dekodovat a použít ve formátu vhodném pro program.

Následující program vytiskne svoje jméno a zadané argumenty:

```
#include <stdio.h>
main(int argc, char** argv)
{
    int i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d] = \"%s\"\n", i, argv[i]);
}
```

Programátoři pod UNIXem mají určité konvence, jak interpretovat seznam parametrů.

Není to závazné, ale může to pomoci. Přepínače a klíčové pojmy jsou vždy uvozeny znakem "-". To je umožňuje snadno rozpoznat při procházení seznamem parametrů. Pak v závislosti na příznaku, může následující argument obsahovat informaci, která má být konvertována jako integer, float, nebo zůstat prostě řetězcem. S těmito konvencemi lze nejobvyklejším způsobem rozklíčovat seznam argumentů argv pomocí cyklu a příkazu přepínače takto:

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char** argv)

    /* Nastavení implicitních hodnot pro všechny parametry: */

    int a_value = 0;
    float b_value = 0.0;
    char* c_value = NULL;
    int d1_value = 0, d2_value = 0;
    int i;
    for (i = 1; i < argc; i++) {
        if (argv[i][0] == '-') { // přepínač musí začínat znakem "-"
            switch (argv[i][1]) { // co se bude dělat ??
                case 'a':        a_value = atoi(argv[++i]);
                                break;
                case 'b':        b_value = atof(argv[++i]);
                                break;
                case 'c':        c_value = argv[++i];
                                break;
                case 'd':        d1_value = atoi(argv[++i]);
                                d2_value = atoi(argv[++i]);
                                break;
            }
        }
    }
    printf("a = %d\n", a_value);
    printf("b = %f\n", b_value);
    if (c_value != NULL) printf("c = \"%s\"\n", c_value);
```



```
printf("d1 = %d, d2 = %d\n", d1_value, d2_value);
}
```

Všimněte si, že `argv[i][j]` znamená j-tý znak v i-tem řetězci znaků. Příkaz `if` testuje vedoucí znak na `-` (0. tý znak), a pak příkaz `switch` umožňuje několik možností zpracování v závislosti na dalším znaku v řetězci (1. znak {a, b, c, d}). Pozor na to, že zápis `argv[++i]` zvětšuje `i` před jeho použitím, umožňující tak přistoupit k následujícímu znaku v jediném příkazu společně s indexací. Konverzní funkce `atoi` a `atof` jsou definovány v `stdlib.h`. Konvertují řetězec znak na `int` resp. `double`.

Typická příkazová řádka může vypadat takto:

```
Prog -a 3 -b 5.6 -c "I am a string" -d 222 111
```

(Použití uvozovek `"` společně s příznakem `-c` zde způsobuje že celý řetězec mezi uvozovkami bude použit jako jeden objekt typu `string`, včetně mezer, které jsou v něm uvedeny a které by jinak působil jako oddělovač.)

Podobně mohou být zpracovány libovolně složité řádky. Zde je jednoduchý program ukazující jak rozvrhnout příkaz pro analýzu řádku v samostatné funkci, jejímž účelem je interpretovat příkazový řádek a provést konverzi příslušných argumentů:

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

void get_args(int argc, char** argv, int* a_value, float* b_value)
{
    int i;

    /* Zaczne pro i = 1 a přeskočíme jméno programu */
    for (i = 1; i < argc; i++) {

        /* Testujeme přepínač (začíná znakem "-"). */
        if (argv[i][0] == '-') {

            /* Další znak rozhoduje co se bude dělat */
            switch (argv[i][1]) {

                case 'a':      *a_value = atoi(argv[++i]);
                               break;
                case 'b':      *b_value = atof(argv[++i]);
                               break;
                default:        fprintf(stderr,
                                   "neznámý přepínač %s\n", argv[i]);
            }
        }
    }
}

main(int argc, char** argv)
{
    /* Nastavení implicitních hodnot pro všechny parametry */
    int a = 0;
    float b = 0.0;
    get_args(argc, argv, &a, &b);
    printf("a = %d\n", a);
}
```

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

2.15. Standardní vstup a výstup podrobněji



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- deklarovat a používat výčtový uživatelský typ enum a strukturální uživatelský typ union



Výklad

Každý program při svém spuštění dostane od operačního systému standardní vstup (obvykle klávesnice), standardní výstup (obvykle monitor) a standardní chybový výstup (obvykle monitor). Standardní vstup se označuje jako `stdin`, standardní výstup jako `stdout` a chybový výstup je `stderr`.

Funkce, které s těmito vstupy a výstupy pracují jsou definované v hlavičkovém souboru `<stdio.h>`. Například funkce `printf()` tiskne vše do standardního výstupu `stdout`. V OS máme možnost standardní vstupy a výstupy přesměrovat (například výstup do souboru místo na obrazovku).

Vstupní a výstupní zařízení lze rozdělit na znaková a bloková. Znaková jsou například klávesnice a monitor, bloková např. pevné disky, nebo dříve používané magnetické pásky.

Rozdíl mezi těmito typy zařízení spočívá ve způsobu fyzického čtení nebo zápisu na ně. Na znaková zařízení je možno zapisovat nebo je číst znak po znaku, na bloková zařízení zapisujeme informaci po blocích. Blok je tvořen posloupností znaků určité nebo zvolené délky. U diskových zařízení je to například délka sektoru u páskových zařízení je možno délku definovat.

Další rozdíl v přístupu k informaci uložené na nějakém zařízení je dán také tím, zda je informace zapisována tzv. sekvenčně nebo náhodným, přímým, přístupem. Souvisí to s organizací přístupu a ukládáním informace formou souborů a bude vysvětleno později v kapitole o souborech.

□ Funkce `printf()`

Funkce `printf()` se používá pro formátovaný standardní výstup (do `stdout`). Rozhraní funkce `printf()` vypadá takto:

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

Funkce `printf()` má návratovou hodnotu typu `int`. Tato hodnota je rovna počtu znaků zapsaných do výstupu, nebo speciální hodnotě EOF v případě chyby zápisu.

Hodnota EOF se používá jako označení konce souboru (End Of File), taktéž konce standardního vstupu a výstupu. EOF je definován v souboru `<stdio.h>`. Volat funkci `printf` však můžeme bez použití návratové hodnoty, jak to děláme ve většině případů. První argument definuje formát, v němž jsou argumenty obsažené v příkazu vypisovány na standardní výstup `stdout`. Formát je konstantní řetězec, který kromě textové informace, kterou vypisuje přesně ve tvaru v jakém je v tomto řetězci, může obsahovat speciální sekvence, na jejichž místo jsou dosaženy další argumenty a konvertovány dle ožadavku daném tzv. konverzní specifikací.

Pozice těchto argumentů je naznačena znakem `%`, za nímž následuje pak definice konverze, která se má s daným argumentem provést. Z toho vyplývá, že kolik je speciálních sekvencí v tomto řetězci, tolik dalších argumentů oddělených čárkou musíme funkci `printf()` předat. Argumenty by měly být očekávaného datového typu tak, aby byly v souladu s uvedenou konverzní specifikací.

Obecný formát konverzní specifikace je

```
%[flags][width][.prec][h|l|L]type
```

Vše, co je v deklaraci v hranatých závorkách `[]` je nepovinné, tj. v deklaraci může, ale nemusí být. Znak `|` lze číst jako „nebo“, tj. naznačuje výběr jedné z možností. Například `[h|l|L]` znamená, že v deklaraci konverzní specifikace může být `h` nebo `l` nebo `L` nebo ani jedna z těchto možností.

Působ konverze je dán znakem uvedeným za znakem `%`. Například sekvence `%i` má tisknout celé číslo se znaménkem. Pokud je třeba vytisknout přímo znak procento, pak jej stačí napsat dvakrát za sebou (`%%`). Další tabulka obsahuje seznam dalších možných konverzních typů.

Typ	Význam
d, i	Celé číslo se znaménkem.
U	Celé číslo bez znaménka.
O	Číslo v osmičkové soustavě.
x, X	Číslo v šestnáctkové soustavě. Písmena ABCDEF se budou tisknout jako malá při použití malého <code>x</code> , nebo velká při použití velkého <code>X</code> .
F	Racionální číslo (float, double) bez exponentu.
e, E	Racionální číslo s exponentem, implicitně jedna pozice před desetinnou tečkou a šest za ní. Exponent uvozuje malé nebo velké <code>E</code> .
g, G	Racionální číslo s exponentem nebo bez něj (podle absolutní hodnoty čísla). Neobsahuje desetinnou tečku, pokud nemá desetinnou část.
C	Jeden znak.
S	Řetězec.

Příklad:

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

```

int main(void)
{
    const char *COPYRIGHT = "(C) ";
    const int ROK = 2001;
    printf("%i %u %o %x %X %f %e %G\n", -5, -5, 200, 200, 200, 10.0, 10.0, 10.0);
    printf("%s %i\n", COPYRIGHT, ROK);
    return 0;
}

```

Výstup bude následující:

-5 4294967291 310 c8 C8 10.000000 1.000000e+01 10

(C) 2013

Pokud číslo -5 budeme tisknout konverzí %u, dostaneme číslo 4294967291, protože u označuje číslo bez znaménka, takže znaménkový bit se považuje na platnou binární číslici 1. Položky width a .prec určují délku výstupu. Délka width určuje minimální počet znaků na výstupu. Mohou být uvozeny mezerami nebo nulami. .prec určuje maximální počet znaků na výstupu pro řetězce. Pro celá čísla je to minimum zobrazených znaků, pro čísla typu float pak počet míst za desetinnou tečkou (má to tedy více významů v závislosti na typu).

Položka width může nabývat následujících hodnot:

Hodnoty width	
n	Vytiskne se nejméně n znaků doplněných mezerami
0n	Vytiskne se nejméně n znaků doplněných nulami
*	Vytiskne se nejméně n znaků, kde n je další argument funkce printf()

Položka .prec může nabývat následujících hodnot:

Hodnoty .prec	
.0	pro e , E , f nezobrazí desetinnou tečku
	pro d , i , o , u , x nastaví standardní hodnoty
.n	pro d , i , o , u , x minimální počet číslic
	pro e , E , f počet desetinných číslic
	pro g , G počet platných míst
	pro s maximální počet znaků
.*	jako přesnost bude použit následující parametr funkce printf()

Příklad:

```

#include <stdio.h>
int main(void)
{
    printf("%06i %06u %06x %6x %06f %06E %06G\n\n", -5, -5, 200, 200,
10.0, 10.0, 10.0);
    printf("%*s %1s %6s %06.2G\n", 10, "%i", "%06.2f", "%06.0E");
    printf("%*i %06.2f %06.0E %06.2G\n", 10, -5, 10.0 / 3, 10.0 / 3, 10.0 / 3);
    printf("\n%.8s %0*.*f\n", "Posledni vystup:", 10, 4, 10.0 / 3);
    return 0;
}

```

}

Výstup z programu:

```
-00005 4294967291 0000c8    c8 10.000000 1.000000E+01 000010
```

```
%*i %06.2f %06.0E %06.2G
```

```
-5 003.33 03E+00 0003.3
```

Poslední vystup: 00003.3333

Všimněte si použití %% při druhém volání funkce printf() a také rozdílu mezi %06.2f a %06.2G. Také si všimněte, že do délky čísla se započítává i pozice pro znaménko. Čísla, která jsou delší než požadovaná minimální délka se nezkracují, řetězce ano.

Příznak flags může nabývat hodnot z následující tabulky:

Hodnoty flags	
-	výsledek je zarovnán zleva
+	u čísla bude vždy zobrazeno znaménko
mezera	u kladných čísel bude místo znaménka "+" mezera
#	pro o, x, X výstup jako konstanty jazyka C
	pro e, E, f, g, G vždy zobrazí desetinnou tečku
	pro g, G ponechá nevýznamné nuly
	pro c, d, i, s, u nemá význam.

Znaky h l a L označují typ čísla. Znak h [typ short](#) (nemá smysl pro 16bitové překladače), l dlouhé celé číslo, L long double.

Příklad:

```
#include <stdio.h>
int main(void)
{
    long int x;
    long double y = 25.0L;
    x = -251; /* je mozno psat i jen x = -25, */
    printf("%10s < %+5i> < % 5ld> < %x>\n", "Cisla:", 25, x, -25);
    printf("%-10s < % -+5i> < % 5Lf> < %#x>\n", "Cisla:", 25, y, -25);
    return 0;
}
```

Výstup z programu:

```
Cisla: < +25> < -25> <ffffffe7>
```

```
Cisla:  <+25 > < 25.000000> <0xffffffe7>
```

□ Funkce scanf()

Funkce `scanf` se používá pro formátovaný standardní vstup (ze zařízení `stdin`), což bývá obvykle vstup z klávesnice. Deklarace funkce `scanf()` vypadá takto:

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

Návratová hodnota je rovna počtu bezchybně načtených a do paměti uložených položek, nebo hodnota EOF (End Of File) při pokusu číst položky z uzavřeného vstupu. První argument, podobně jako u funkce `printf()` řetězec, definující vstupní formát, v němž má být požadovaná informace přečtena.. Formát takové sekvence je obecně definován takto

```
%[*][width][h|l|L]type
```

Význam položek je následující:

Význam položek sekvence pro funkci `scanf()`

*	přeskočí popsany vstup (načte, ale nikam nezapíše)
width	maximální počet vstupních znaků
h l L	modifikace typu (jako u printf())
type	typ konverze.

Možné typy konverzí jsou v následující tabulce:

význam type	
d	celé číslo
u	celé číslo bez znaménka
o	osmičkové celé číslo
x	šestnáctkové celé číslo
i	celé číslo, zápis odpovídá zápisu konstanty v jazyce C, např. 0x uvozuje číslo v šestnáctkové soustavě
n	počet dosud přečtených znaků aktuálním voláním funkce <code>scanf()</code>
e, f, g	racionální čísla typu float, lze je modifikovat pomocí l a L
s	řetězec; úvodní oddělovače jsou přeskočeny, na konci je přidán znak '\0'
c	vstup znaku; je-li určena šířka, je čten řetězec bez přeskočení oddělovačů
[search_set]	jako s, ale se specifikací vstupní množiny znaků, je možný i interval, například %[0-9], i negace, například %[^a-c].

Oddělovače jsou tzv. bílé znaky (tabulátor, mezera, konec řádku (ENTER)). Ty se při čtení ze vstupu přeskakují (výjimkou může být typ c). Načítání tedy probíhá tak, že se nejdříve přeskočí oddělovače a poté se načte požadovaný typ. Pokud je požadovaný typ například číslo, ale místo něho je na vstupu písmeno, pak dojde k chybě. Pokud

se načte požadovaný typ, uloží se na adresu, která je uložena v dalším argumentu funkce `scanf()`. Volání funkce `scanf()` může vypadat např. takto:

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

Funkce `scanf()` přečte číslo a uložit jej do proměnné `x`, jejíž adresa je dalším argumentem funkce `scanf()`. Adresa proměnné se získává pomocí operátoru `&` a může být uložena v ukazateli.

Pokud se vstup provede úspěšně, vrátí funkce `scanf` hodnotu číslo 1 (načtena jedna správná položka). Pokud dojde při vstupu k chybě (např. místo čísla zadáme nějaké znaky, nebo vstup ukončíme), vrátí se EOF. Návratovou hodnotu se naučíme využívat až v kapitolách věnovaných řízení programu.

Příklad:

```
#include <stdio.h>
int main(void)
{
    int x = -1;
    printf("Zadej cislo jako konstantu jazyka C\n"
           "napr. 10 0x0a nebo 012: ");
    scanf("%i", &x);
    printf("Zadal jsi cislo %i\n", x);
    return 0;
}
```

Po spuštění programu, čeká funkce `scanf()` na vstup z klávesnice, dokud nějaký nedostane, nebo dokud nebude vstup uzavřen. Vstup z klávesnice se programu odešle až po stisku klávesy ENTER.

Možné výstupy:

Zadej cislo jako konstantu jazyka C

napr. 10 nebo 0x0a nebo 012: -50

Zadal jsi cislo -50

Zadej cislo jako konstantu jazyka C

napr. 10 nebo 0x0a nebo 012: 0xff

Zadal jsi cislo 255

Zadej cislo jako konstantu jazyka C

napr. 10 nebo 0x0a nebo 012: ff

Zadal jsi cislo -1

Při posledním spuštění programu bylo zadáno „ff“, což není číslo ale textový řetězec, proto funkce `scanf()` do proměnné `x` nic neuloží a tak v `x` zůstane hodnota -1.

Vstupní proud (`stdin`) můžete přerušit ve Windows a v DOSu klávesovou zkratkou CTRL+Z (stiskněte a držte CTRL a k tomu stiskněte písmeno "z"). V Linuxu pomocí klávesové zkratky CTRL+D.

❑ Escape sekvence

Escape sekvence jsou sekvence znaků, které nám umožňují vložit do řetězce některé zvláštní znaky. Přehled escape sekvencí jsou uvedeny v následující tabulce.

Escape sekvence		
Escape znak	význam	popis
\0	Null	Nula, ukončení řetězce (má být na konci každého řetězce)
\a	Alert (Bell)	pípnutí
\b	Backspace	návrat o jeden znak zpět
\f	Formfeed	nová stránka nebo obrazovka
\n	Newline	přesun na začátek nového řádku
\r	Carriage return	přesun na začátek aktuálního řádku
\t	Horizontal tab	přesun na následující tabulační pozici
\v	Vertical tab	stanovený přesun dolů
\\	Backslash	obrácené lomítko
\'	Single quote	apostrof
\"	Double quote	uvozovky
\?	Question mark	otazník
\000		ASCII znak zadaný jako osmičková hodnota
\xHHH		ASCII znak zadaný jako šestnáctková hodnota

Připomeňme, že pro vytištění znaku % pomocí funkce printf() je třeba zapsat znak % dvakrát za sebou. Nejedná se o escape znak, ale o vlastnost funkce printf() interpretovat % ve svém prvním argumentu zvláštním způsobem.

Příklad:

```
#include <stdio.h>
int main(void)
{
    printf("%s %%\n", "%");
    printf("%s\r%s\n", "AAAAAA", "BB");
    printf("\n\"a\"\\n");
    printf("\x6a\n");
    return 0;
}
```

Výstup z programu:

% %

BBAAAAA

""

j

Při třetím volání printf() se ozve pípnutí.

2.16. Soubory



Čas ke studiu:

1 hodina.



Cíl:

Po prostudování tohoto odstavce budete umět:

- deklarovat a používat výčtový uživatelský typ enum a strukturální uživatelský typ union



Výklad

Původním a hlavním cílem, pro něž byly soubory vynalezeny a vytvořeny mechanismy pro práci s nimi, je potřeba uchování informace na vnějším paměťovém mediu. To znamená nikoliv v operační paměti, kde by došlo k její ztrátě při vypnutí počítače. Dalším neméně důležitým důvodem je umožnit přehledné trvalé či přechodné uložení velkého rozsahu informací převyšujícím kapacitu operační paměti.

K ukládání této informace na vnější paměťová zařízení byla postupně použita a jsou používána různá media; magnetické pásky, pružné disky, pevné disky, flash-disky atp. Pro zápis a čtení informace z nich pak byly navrženy a zkonstruovány potřebné vstupně-výstupní jednotky. Způsob uložení informace na tato media a mechanismus přístupu k nim je zařízení od zařízení různý a liší se také od způsobu práce a ukládání informace v operační paměti. Proto bylo potřeba vytvořit programovou podporu, která by umožnila „smazat“ mezi nimi tyto rozdíly a dovolila uživateli (programátorovi) pracovat pouze s logickou strukturou souboru bez ohledu na zapeklitost jednotlivých vnějších paměťových zařízení. Tento mechanismus se skrývá v knihovnách, případně v části operačního systému zvané „Systém ovládání souborů“.

Koncept souboru vychází z představy, že každá aplikace (programátor) pracuje s jednou nebo několika důležitými abstrakcemi (entitami, datovými objekty) z nichž každá je charakterizována nějakou množinou dat a to jsou informace, které je nutné ukládat do souborů. Souhrn informací charakterizujících příslušný objekt se nazývá záznam. Jednotkou uložení informace do souboru, tak jak s ní pracuje aplikace, je tedy záznam. Uživatel pak vidí soubor jako posloupnost záznamů. Jednotlivé záznamy se člení na položky, každá položka obsahuje hodnotu jednoho datového údaje. Délka záznamu (v bytech) je dána součtem délek všech jeho datových položek a bude podle typu záznamu a podle typu aplikace odlišná pro každý soubor. Se všemi odlišnostmi se musí systém ovládání souborů vyrovnat.

Na soubor se ovšem můžeme dívat také tak, že je to posloupnost bytů, která je ukončena nějakým speciálním znakem EOF (end of file) označujícím konec souboru. Při kopírování souboru se pak nemusíme ohlížet na jeho logickou strukturu a čteme

z něho informaci byte po bytu, pokud nenarazíme na znak EOF. Avšak při práci s vnějším médiem nemůžeme číst nebo zapisovat informaci na medium byte po bytu, protože tam se ukládá informace po větších celcích, např. na mag. pásku po blocích, na disk po sektorech. Je tomu tak i tehdy, pracujeme li se záznamy, jejichž délky nebudou stejné, jako jsou délky sektorů nebo bloků. Ty budou pevné a typické pro vnější medium. I tímto problémem se musí vypořádat systém pro ovládání souborů.

Existence systému pro ovládání souborů nám tedy umožňuje nahlížet na každou operaci čtení nebo zápisu na vnější zařízení jako operaci se souborem. Přitom podstatnou zde je struktura zapisované informace, která vystupuje nebo vstupuje v jakémsi „proudu“ (stream) dat, a je jedno odkud nebo kam.

V Céčku je proto možné i nutné, kromě operací se standardními vstupními a výstupními zařízeními (stdin, stdout, stderr), vykonávat operace se soubory.

Máme dvě skupiny standardních funkcí pro dvě úrovně práce se soubory. Nižší úroveň tvoří funkce využívající přímo služby operačního systému. Vyšší úroveň pak funkce pro práci s tzv. proudy (stream) údajů. Táto úroveň je uživatelsky přijatelnější

Každá činnost spojená s komunikací se souborem se může resp. musí sestávat z těchto kroků:

otevření souboru

manipulace se souborem (čtení, zápis, nastavení)

zavření souboru

□ Práce s proudy (stream I/O) a se soubory

K otevření proudu resp. k otevření existujícího souboru nebo k vytvoření nového souboru slouží funkce `fopen()`

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

`jmeno` je označení souboru, který se má otevřít nebo vytvořit - řetězec nebo pole typu `char` obsahující jméno souboru, eventuálně včetně disku a cesty

parametr `modus` určuje mód, ve kterém se má s otevíraným souborem pracovat.

„r“ textový soubor pro čtení

„w“ textový soubor pro zápis nebo pro přepsání

„a“ textový soubor pro připojení na konec (append)

„rb“ binární soubor pro čtení

„wb“ binární soubor pro zápis nebo pro přepsání

„ab“ binární soubor pro připojení na konec

„r+“ textový soubor pro čtení a zápis

„w+“ textový soubor pro čtení, zápis nebo přepsání

„a+“ textový soubor pro čtení a zápis na konec

„rb+“ binární soubor pro čtení a zápis

„wb+“ binární soubor pro čtení, zápis nebo přepsání

„ab+“ binární soubor pro čtení a zápis na konec

Pro uzavření souboru je určena funkce `fclose()` s rozhraním

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

Pokud se soubor identifikovaný ukazatelem `file` nepodaří uzavřít, (např. nebyl-li otevřen), vrací funkce hodnotu symbolické konstanty `EOF`.

```
FILE *soubor;
...
soubor=fopen („DATA.TXT“, „r“);
...
fclose(soubor);
```

Operace otevření a uzavření souboru nemusí z určitých důvodů proběhnout, program by proto měl vždy testovat úspěšnost provedení těchto operací:

```
FILE *soubor;
...
soubor=fopen („DATA.TXT“, „r“);
if(soubor==NULL) {
    printf („\nChyba pri otevreni DATA.TXT“);
    return;
}
...
If (fclose(soubor)==EOF) {
    printf („\nChyba pri uzavreni DATA.TXT“);
    return;
}
```

Vstup znaků ze souboru a výstup znaků do souboru

Lze ji realizovat pomocí funkcí `getc` a `putc`. Všimněte si podpoby resp. odlišností od nesouborových funkcí `getchar` a `putchar`.

```
int getchar(void);          int getc(FILE *file);
int putchar(int c);         int putc(int c, FILE *file);
```

Příklad:

```
#include <stdio.h>
#include <ctype.h>
// Program urci pocet radek a neprazdnych radek textoveho souboru
// (prazdna radka - radka, na ktere jsou zapsany pouze „bile“ znaky)
// -----
main(){
    FILE *soubor; int c,zn=0,rd=0,rd_0=0;
    // zn - pocet znaku na radce (bile znaky nejsou zapocitany)
    // rd - pocet radek (radky obsahující pouze bile znaky nejsou
    //      zapocitany)
    // rd_0 - pocet radek (vcetne „prazdnych“ radek)
    // -----
    if((soubor=fopen („TEXT“, „r“))==NULL){          /* test - otevreni */
        printf („\nChyba pri otevreni TEXT“);
        return;
    }
    while((c=getc(soubor)) != EOF){                  /* test - konec
    souboru */
```

```

        if(c=='\n'){
- konec radky    */
            rd_0++;
            if(zn>0) rd++;
            zn=0;
        }
        If (isspace(c)==0) zn++;          /* podmínka neplatí pro „bílý“ znak
*/    }
        rd_0++;
        if(zn>0) rd++;
        printf(„\nPocet neprazdnych radek=%d\nPocet vseh
radek=%d“, rd, rd_0);
        if (fclose(soubor)==EOF) printf(„\nChyba pri uzavreni TEXT „);
        return 0;
    }

```

□ Formátovaný vstup a výstup do souboru a ze souboru

Používají se funkce fscanf(), fprintf() analogické funkcím scanf(), printf(). Porovnejte následující zápisy.

```

int scanf(const char *retezec, arg1, arg2, ..., argn);
int fscanf( FILE *file, const char *retezec, arg1,arg2,...,argn);
int printf(const char *retezec, arg1, arg2, ..., argn);
int fprintf( FILE *file, const char *retezec, arg1,arg2,...,argn);

```

□ Vstup řádek ze souboru a výstup řádek do souboru

Vstup

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

Přečte řetězec znaků až do znaku '\n' nejvýše však max znaků ze souboru identifikovaného ukazatelem file a uloží jej do řetězce str. Znak '\n' se neukládá a řetězec je automaticky ukončen znakem '\0'. Návrátová hodnota funkce je ukazatel na řetězec str. Pokud je řetězec prázdný, vrací funkce hodnotu symbolické konstanty NULL

Výstup

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

Vytiskne řetězec str do souboru identifikovaného ukazatelem file. Řetězec neukončuje znakem '\0' ale znakem '\n'. Funkce vrací nezáporné číslo, v případě, že operace nemůže z nějakého důvodu proběhnout, je návratová hodnota rovna EOF

Příklad

```

#include <stdio.h>                                     //kopírování souboru
main()
{
    char nazev[20]; int c; FILE *file_r,*file_w;
    printf(„\nKtery soubor se ma kopirovat?“
        „\n Zadej nazev existujiciho textoveho souboru!\n“);
    gets(nazev);                                       /* čtení názvu souboru, který
budeme kopírovat    */
    file_r=fopen(nazev,“r”);

```

```

if(file_r==NULL){
    printf(„\n Chyba pri otevreni souboru pro cteni!");
    return;
}
printf(„\n Do ktereho souboru kopirovat?“
    „\n Zadej novy odlisny nazev textoveho souboru!\n");
gets(nazev); /* cteni nazvu souboru, do
kterého uložíme kopii */
file_w=fopen(nazev,"w");
if (file_w==NULL){
    printf(„\n Chyba pri otevreni souboru pro zapis!");
    return;
}
//                                Kopírování souboru znak po znaku
/* Testovani konce souboru: */
while (c=getc(file_r))!=EOF) /* c=getc() musi byt v zavorce */
    putc(c,file_w);
if ( (fclose(file_r)==EOF) || ( fclose(file_w)==EOF ) )
    printf(„\nNektery ze souboru nelze uzavrit!");
return;
}

```

Použití fgets(), fputs(), pro kopírování po řádcích (záznamech)

Předpoklady: Jde o řádky == textové záznamy (řetězce), které neobsahují více než 100 znaků

```

...
char radka[101]; /* k popisu doplnime definici pole
'radka' */
...
/* Kopirovani souboru: */
while (fgets(radka,100,file_r) != NULL ) /* kopirovani po radkach */
    fputs(radka,file_w);
...

```

Práce s binárními soubory

čtení dat - funkce fread():

```
int fread(char *kam, int delka, int pocet, FILE *file);
```

fseek() a ftell().

Funkce fseek() se používá pro nové nastavení pozice v souboru:

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

kde

file - proměnná typu ukazatel na FILE identifikující binární soubor

posun - počet slabik (byte) od pozice v souboru dané parametrem 'odkud',

odkud - místo odkud se posun provede, parametr může mít jednu ze tří hodnot:

SEEK_SET - od začátku souboru

SEEK_CUR - od aktuální pozice v souboru

SEEK_END - od konce souboru

Návratová hodnota funkce fseek() je nula v případě úspěšného přesunu a nenulová hodnota v opačném případě.

Funkce ftell() se používá pro zjištění aktuální pozice v souboru:

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

Funkce vrací posunutí měřené ve slabikách (bytes) od začátku souboru

Příklad:

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
    FILE *f;
    char retezec[] = „Test funkci fseek() a fwrite().“;
    int i;
    clrscr();

    /* otevřeme soubor, ze kterého je možné i číst,
    pro zápis, */
    f=fopen(„S.TXT“, „wb+“);
    fwrite(retezec,strlen(retezec),1,f); /* zapiseme řetězec do souboru:
    */
    fseek(f, 0, SEEK_SET); /* nastavíme „ukazovátka“ na začátek souboru */
    do { /* čteme jednotlivé znaky ze
    souboru */
        i=fgetc(f);
        putchar(i); /* vytiskneme je:
    */
    } while (i != EOF);
    fclose(f); /* uzavře soubor S.TXT */
    return 0;
}
```

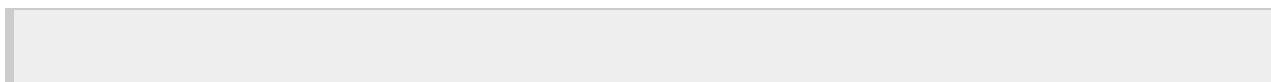
Funkce strlen() se používá pro zjištění délky řetězce (bez ukončujícího znaku),

Parametry funkce main()

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

Kdyby byl program např. VYPIS.EXE s hlavní funkcí uvedeného typu spuštěn z příkazové řádky instrukcí VYPIS TENTO TEXT

```
#include <stdlib.h> /* Obsahuje funkce pro práci s řetězcí */
#include <stdio.h>
main(int argc, char *argv[])
{
    long L1,L2;
    if(argc!=3){printf(„\nVyvolani programu napr.: SECTI 1 2“
    „\nProgram secte zadana dve cela cisla.“);
    return;
    }
    /* standardni funkce atol() konvertuje retezec na long : */
    L1=atol(argv[1]);
    L2=atol(argv[2]);
    printf(„\tSoucet = %ld“,L1+L2);
    return;
}
```



3. SOUČASNÉ VYUŽITÍ PROGRAMOVACÍHO JAZYKA C V OBLASTI MIKROČIPŮ

Tato kapitola přináší pohled do jedné z největších domén programovacího jazyka C v dnešní době přeplněné kvanty objektově orientovaných i sekvenčních skriptovacích jazyků a to sice domény mikroprocesorů – monolitických jednočipových procesorů.

3.1. Programování mikroprocesorů AVR



Čas ke studiu:

4 hodiny (s příklady 8 hodin).



Cíl:

Po prostudování tohoto odstavce budete umět

- Popsat architekturu mikroprocesoru AVR,
- vytvořit program pro mikroprocesor AVR,
- nahrát program do tohoto mikroprocesoru a odladit jej.



Výklad

Jednou z hlavních domén programovacího jazyka C, kde nalezne v dnešní době objektově orientovaných programovacích jazyků a dalších funkcionálních paradigmatických jazyků je programování mikroprocesorů ať už 8, 16 či 32-bitových.

Vzhledem k tomu, že se tyto skripta zabývají základy programování, probereme si základní 8-mi bitovou architekturu, která je pro pochopení této problematiky nejvhodnější.

□ Mikroprocesory AVR - architektura

Architektura tohoto 8-mi bitového mikroprocesoru je typu RISC (Reduced Instruction Set Computer).

S AVR-rodinou ukazuje firma Atmel, že RISC-architektura nemusí být používána jen u výkonových procesorů pro pracovní stanice nebo 32-bitových mikroprocesorů pro intenzivní početní úkony, ale má také smysl u 8-mi bitových mikroprocesorů. AVR-série nabízí běžné přednosti RISC-architektury, tj. jednocyklové instrukce, vyšší taktovací frekvence spojená s vyšším pracovním výkonem stejně jako efektivní optimalizace překladu.

Původně byla vyvinuta nová 8-bitová RISC-architektura v norském vývojové centru Nordic VLSI v Trondheimu. Před více jak třemi lety koupila firma Atmel tuto koncepci

na které je založena 8-bitová mikroprocesorová rodina, která přišla před dvěma lety na trh. Firma Atmel je známa svými produkty jako je programovatelná logika, paměti typu EPROM, EEPROM a FLASH a především také FLASH-mikroprocesory založené na Intelovské rodině mikroprocesorů 8051. AVR-rodina je konkurentem na trhu některých dobře zavedených mikroprocesorových rodin, jako jsou 8-bitové rodiny 6805 a 68HC11 od Motoroly stejně jako 8051 od Intelu, jejichž čipy vyrábějí různí vlastníci licencí včetně Atmelu. Hlavním konkurentem je firma Microchip s moderní 8-bitovou PIC-rodinou, která se stává v posledních letech značně populární. Microchip vsadil také na architekturu blízkou RISC-architektuře s mnoha jednocyklovými instrukcemi.

Jádro AVR-série se podobá jádru většiny RISC-procesorů, které jsou dostupné na trhu. AVR jádro se skládá ze 32 stejných 8-bitových registrů, které mohou obsahovat jak data tak adresy. Posledních 6 registrů můžeme ve dvojici použít jako ukazatele adresy pro nepřímé adresování paměti dat. Tyto registry označované písmeny X, Y a Z dovolují libovolné ukládací operace (Load/Store) viz.obr.č.1. Programátor má například na výběr, zda ukazatel adresy bude po zpracování určité instrukce inkrementovat nebo před zpracováním této instrukce dekrementovat. Užitečné je pro adresování využít možnosti 6-ti bitového posunu v ukazateli adresy v dvojitých registrech Y a Z.

AVR architektura má 5 adresovacích módů pro paměť dat:

- přímé adresování
- nepřímé adresování s posunutím (6-ti bitový posun)
- nepřímé adresování
- nepřímé adresování s dekrementací ukazatele adresy před zpracováním instrukce
- nepřímé adresování s inkrementací ukazatele adresy po zpracování instrukce

Instrukční sada mikroprocesoru Atmegax8

Instrukční sada představuje skupinu příkazů, které je schopno jádro mikroprocesoru zpracovat. Mezi tyto instrukce patří například:

- ADD(Rd, Rr) – Add without carry – operace součtu obsahu registrů Rd + Rr. Výsledek je uložen do registru Rd. V případě přetečení hodnoty 255 (2^8), není nastaven příznak C (Carry) v registru Status. Registry Rd+Rr jsou 8-mi bitové a tudíž se jedná o 8-mi bitový součet,
- další 8-mi bitové instrukce (ADDC – součet s příznakem, SUB rozdíl dvou registrů, SUBI – rozdíl registru a konstanty, INC - inkrementace, DEC - dekrementace),
- pro 16-ti bitový součet existuje operace ADIW (RdI, K16 – 16-ti bitová konstanta),
- jako registr Rd mohou sloužit pouze registry (R24, R26, R28, R30, XL, YL, ZL),
- pro další instrukce odkazuje čtenáře na [3].

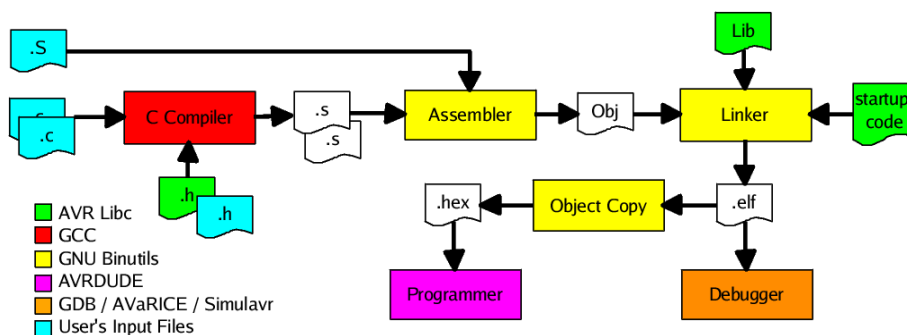
Ukázka kódu v jazyce symbolických adres (assembler), kde se používají výše zmíněné instrukce (ADD):

```
# Číslo 1
LDI R16 0x...
LDI R17 0x...
LDI R18 0x...
LDI R19 0x...
# Číslo 2
LDI R20 0x...
LDI R21 0x...
LDI R22 0x...
LDI R23 0x...
# Součet horní poloviny int – 16-ti bit
ADD R20,R16
# Součet dolní poloviny int – 16-ti bit
ADC R21,R17
ADC R22,R18
ADC R23,R19 # MSB
```

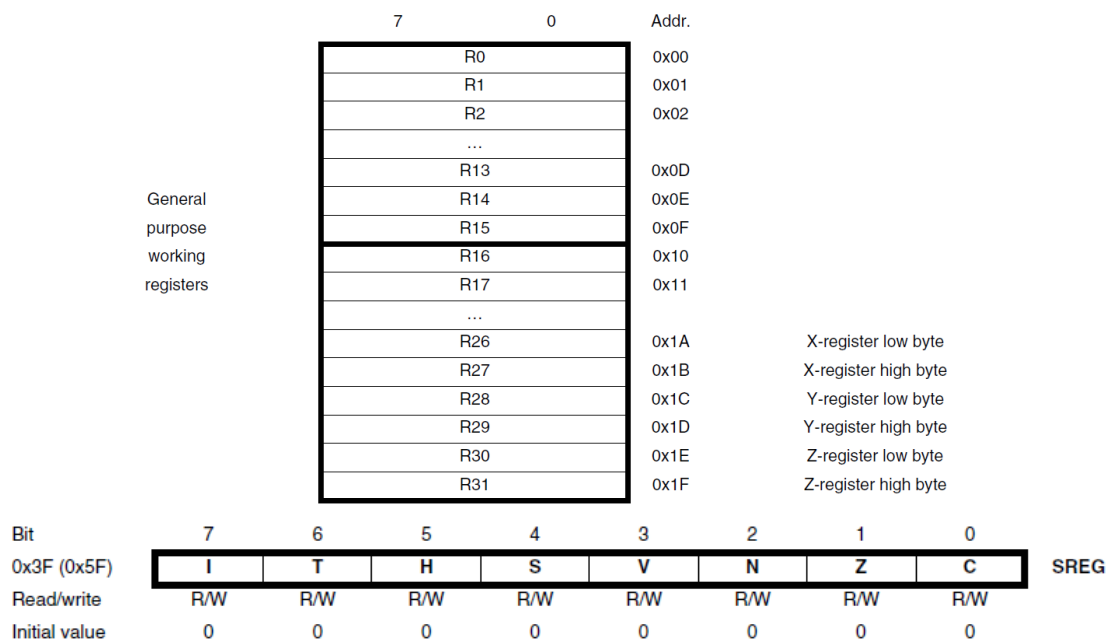
Naopak v jazyce C stačí pro takovýto součet pouze pár řádků:

```
#include <avr/io.h>
int main(void)
{
    int vysledek, a, b;
    a = 10;
    b = 56;
    vysledek = a+b;
}
```

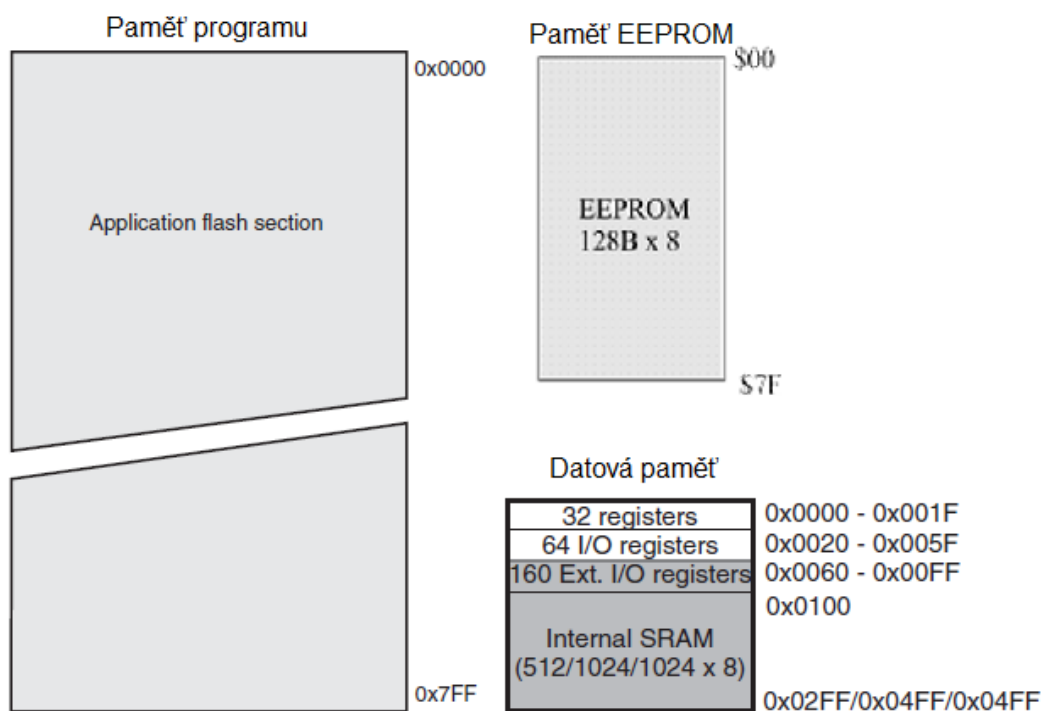
Kompilátor jazyka C pro mikroprocesor AVR sám zjistí jaké registry použít a jaké instrukce použít pro provedení operace součtu. Čili výstupem bude kód velice podobný assembler kódu, který bude dále přeložen pomocí assembleru a linkeru do spustitelného strojového kódu, který bude pomocí programátoru nahrán do mikroprocesoru.



Obr. 3 Mapa procesu překladač zdroj: <http://www.embedded.dk/>.



Obr. 4 CPU a status register mikroprocesoru AVR Atmega48 [2].



Obr. 5 Mapa paměti, mikroprocesoru ATmega48 [2].

Jak je u mnoha jednoduchých mikroprocesorů obvyklé, registry jsou zobrazovány přímo v adresovém prostoru dat. Prvních 32-bitů paměti, viz Obr. 5 Mapa paměti,

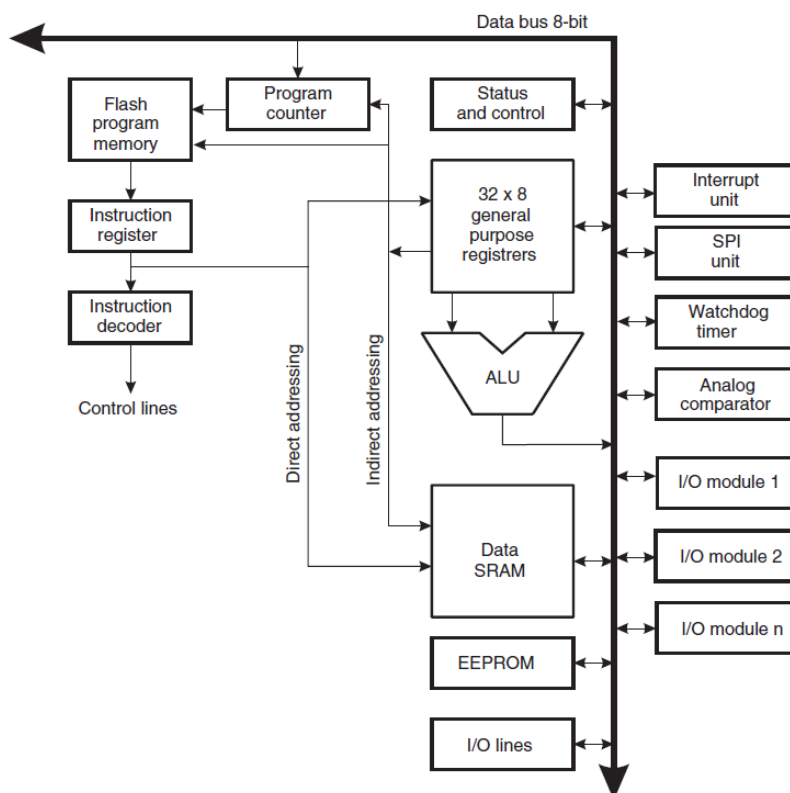
mikroprocesoru ATmega48 [2]. (0x00 až 0x1F) odpovídá registrům R0 až R31., ty jsou také přímo zamapovány do paměti dat. Proto je možno s každým registrem zacházet použitím standardních odkazů bez toho, aby programátor potřeboval znát řídicí instrukce registrů. AVR používá Harvardskou architekturu, tj. paměť programu a paměť dat jsou odděleny. Paměť programu je přístupná pomocí dvoustupňové pipeline. Když se určitá instrukce začíná vykonávat, další instrukce je připravována k zpracování. Tato konstrukce dovoluje zpracovávání instrukcí vždy v jednom hodinovém cyklu. Mikroprocesor, se kterým se seznámíte na cvičení je 8-mi bitový mikroprocesor z poslední generace AVR mikroprocesorů. Jedná se o mikroprocesor Atmega48. Tento mikroprocesor obsahuje 4kB paměti programu, 512B statické paměti pro data a 256kB paměti EEPROM.

Zpracování operandů probíhá následovně: během jednoho taktovacího cyklu se přivedou oba zdrojové operandy z pracovních registrů, uskuteční se potřebná operace a výsledek se uloží zpátky do registrů. Při úspěšném provedení je tedy potřebný čas jeden takt. Jedná se výhradně o operace typu registr - registr, na kterých je AVR-architektura založena a důsledně je dodržuje podle modelu LOAD/STORE.

Většina všech instrukcí je 16-ti bitových. Pouze 4 instrukce jsou 32 bitové, které jsou však omezeny dovoleným 16-ti bitovým adresováním. Ačkoli ukazatele na data mohou být až 16-ti bitové, je programový čítač (PC) pouze maximálně 12-bitový. Velikost programového čítače závisí na typu mikroprocesoru, například AT90S2323 má PC 10-ti bitový, kdežto AT90S8515 má PC 12-bitový.

Celá instrukční sada s maximálně 118 instrukcemi nutí také ke kompromisům. Většina instrukcí v instrukční sadě má přímý a jednocyklový přístup do všech registrů. Výjimku tvoří pouze aritmetické a logické instrukce jako SBCI, SUBI, CPI, ANDI a ORI mezi konstantou a registrem a instrukce LDI (load immediate constant data). Jedná se tedy hlavně o zpracování přímých ("immediate") datových typů (literals). Jen výše uvedené instrukce akceptují přímé hodnoty a mohou být používány pouze v horní polovině registrů (R16 až R31). Ještě je možné pro zpracování přímých datových typů použít instrukce ADIW a SBIW, které jsou přístupné pouze v posledních 8 registrech (R24 až R31).

Pro podmíněné větvení a skoky lze použít různé instrukce. Na každý z osmi příznaků ve Status registru připadají nejméně 2 instrukce podmíněného skoku (např. BREQ, BRTS atd.). Další instrukce pro skoky je RJMP (12-ti bitová), kterou lze posunout relativně kód o 2 kB. Pak je zde absolutní skoková instrukce IJMP, která provede skok na danou adresu. Zajímavé jsou také mnohé "Skip" instrukce (SBRC, SBRS, SBIC a SBIS), které umožňují podmíněný skok přes následující instrukci, je-li nastaven některý bit v daném registru či nikoli. Je-li například přeskočená instrukce RJMP, nechají se využít tyto "Skip" instrukce k efektivní implementaci podmíněného větvení s většími skoky. Právě tak také můžeme použít "Skip" instrukce k přeskočení jednotlivých aritmetických či logických operací při podmíněných operacích.



Obr. 6 Architektura mikroprocesoru Atmega48 [2].

Pro aritmetické a logické operace (jednotka ALU, viz Obr. 6) obsahuje instrukční soubor klasické instrukce sčítání ADD, ADC, odečítání SUB, SUBI, SBC, SBCI, logické instrukce jako AND, OR, EOR atd. Ačkoliv by měla být také definována v instrukčním souboru instrukce násobení, nebyla zatím realizována nebo plánována v žádném ze čtyř řad AVR-mikroprocesorů. V budoucnu by se toto mohlo změnit. Instrukce MUL má podle definice vynásobit spolu dva libovolné 8-mi bitové registry během dvou taktovacích cyklů a 16-bitový výsledek se má uložit do registrů R0 a R1.

Jako mnoho jiných mikroprocesorů, AVR-architektura disponuje také instrukcemi pro bitové operace. Se 16 definovanými instrukcemi je možno nastavit nebo vymazat každý příznak (Flag) ve Status registru, viz Obr. 4. Ačkoliv stejný efekt vznikne také pomocí logických operací, dosáhne se tímto způsobem při programování mnoha aplikací značného zjednodušení. Jsou zde k dispozici instrukce, se kterými se nechá nastavit nebo smazat každý bit víceúčelového nebo vstup/výstupního registru. Například instrukce SER a CLR nastaví, eventuálně smažou najednou celý registr, instrukce SBR a CBR mají vliv na vybrané bity v jednom registru.

❑ Mikroprocesor Atmega48

Na cvičení se seznámíte s mikroprocesorem Atmega48, viz Obr. 7, Obr. 8, pro který budete vytvářet aplikace. Předtím ale než přistoupíme k tvorbě aplikací, musíme se seznámit s jednotlivými perifériemi mikroprocesoru a práce s nimi.

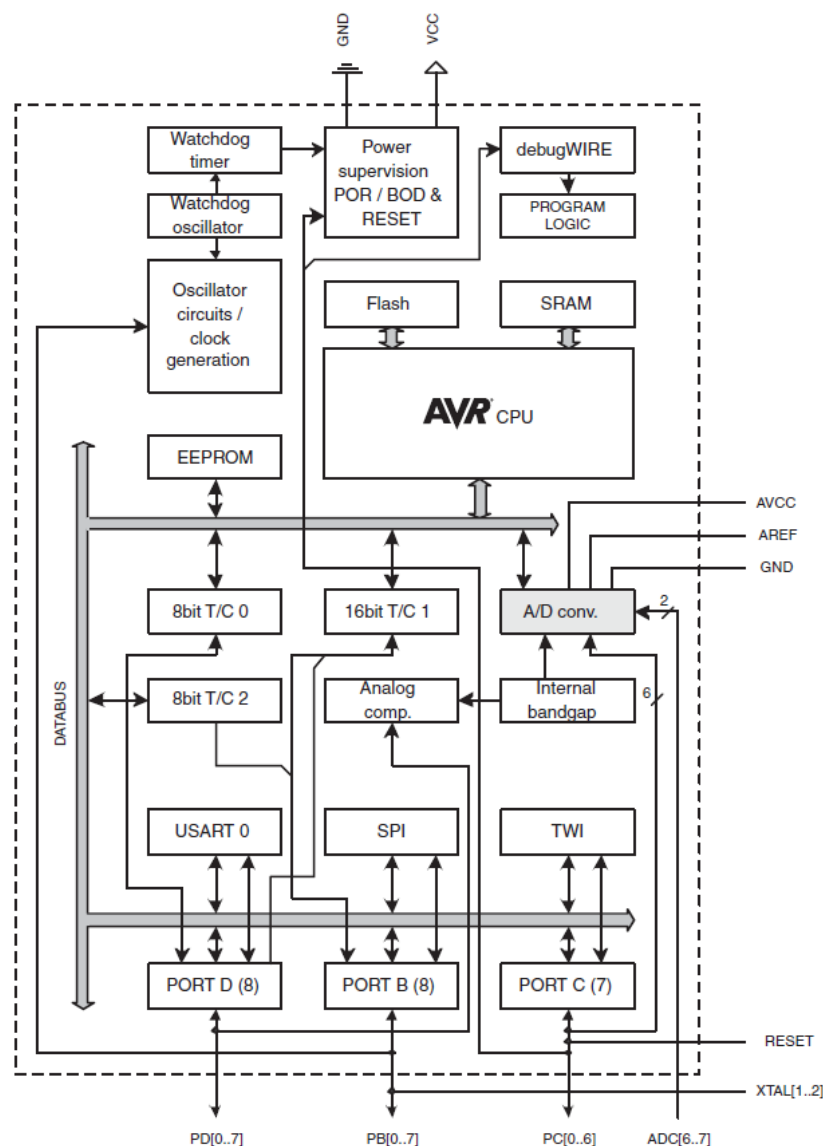


Obr. 7 Procesor Atmega48 v klasickém provedení.

□ **Atmega48 – Periferie**

Mikroprocesor Atmega48 je vyráběn ve dvou provedeních klasickém a SMD (Surface Mounted) a obsahuje několik vstupně-výstupních bran, které ale umí plnit alternativní funkce jako je například vstup pro AD převodník, PWM modul, apod. Mikroprocesor AVR obsahuje následující periferie:

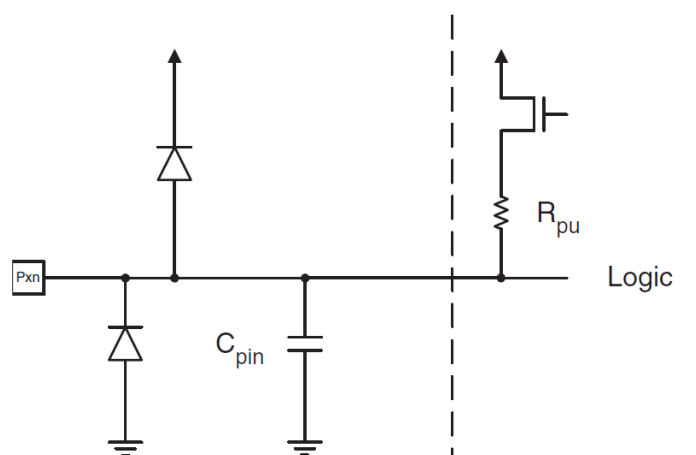
- 23 I/O bran,
- dva 8-mi bitové čítače/časovače s oddělenými předeličkami a komparačním režimem (pin ICP),
- jeden 16-ti bitový čítač/časovač s oddělenou předeličkou a capture režimem,
- čítač reálného času s odděleným oscilátorem,
- šest PWM kanálů,
- 8-mi (SMD pouzdro)/6-ti (klasické pouzdro) kanálový 10-ti bitový AD převodník,
- Programovatelné sériové rozhraní USART (Universal Synchronous Asynchronous serial Receiver and Transmitter)
- Master/slave SPI (Serial Peripheral Interface) komunikační rozhraní
- 2-wire komunikační rozhraní (kompatibilní s Philips I2C)
- Programovatelný watchdog s odděleným oscilátorem
- Analogový komparátor
- Přerušování při změně logické hodnoty vstupně/výstupní brány



Obr. 8 Blokové schéma mikroprocesoru Atmega48 [2].

□ Atmega48 – Vstupně/Výstupní brány

Ekvivalentní schéma vnitřní struktury vstupně/výstupního pinu mikroprocesoru lze vidět na obrázku Obr. 9. Každý vstupně/výstupní pin je vybaven ochrannými diodami, které brání proti přepětí na vstupu. C_{pin} představuje parazitní kapacitu vstupního vodiče. R_{pu} je programově nastavitelný pull-up rezistor. Tento rezistor lze pomocí vnitřního tranzistoru aktivovat, když je brána ve vstupním režimu pomocí zápisu hodnoty log. 1 do speciálního funkčního registru (SFR) $PINx$ (kde hodnota „x“ představuje číslo vstupně výstupní brány např. $PINB3$).



Obr. 9 Ekvivalentní schéma vstupně/výstupního pinu [2].

Většina I/O pinů mikroprocesoru Atmega48 plní určité alternativní funkce například pin 23, viz Obr. 10, slouží jako vstupně/výstupní digitální brána PC0, či jako vstup prvního kanálu vestavěného 10-ti bitového převodníku ADC0, či jako zdroj externího přerušení PCINT8.

(PCINT14/RESET) PC6	1	28	PC5 (ADC5/SCL/PCINT13)
(PCINT16/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)
(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)
(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)
(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)
(PCINT20/XCK/T0) PD4	6	23	PC0 (ADC0/PCINT8)
VCC	7	22	GND
GND	8	21	AREF
(PCINT6/XTAL1/TOSC1) PB6	9	20	AVCC
(PCINT7/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)
(PCINT21/OC0B/T1) PD5	11	18	PB4 (MISO/PCINT4)
(PCINT22/OC0A/AIN0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)
(PCINT23/AIN1) PD7	13	16	PB2 (\overline{SS} /OC1B/PCINT2)
(PCINT0/CLKO/ICP1) PB0	14	15	PB1 (OC1A/PCINT1)

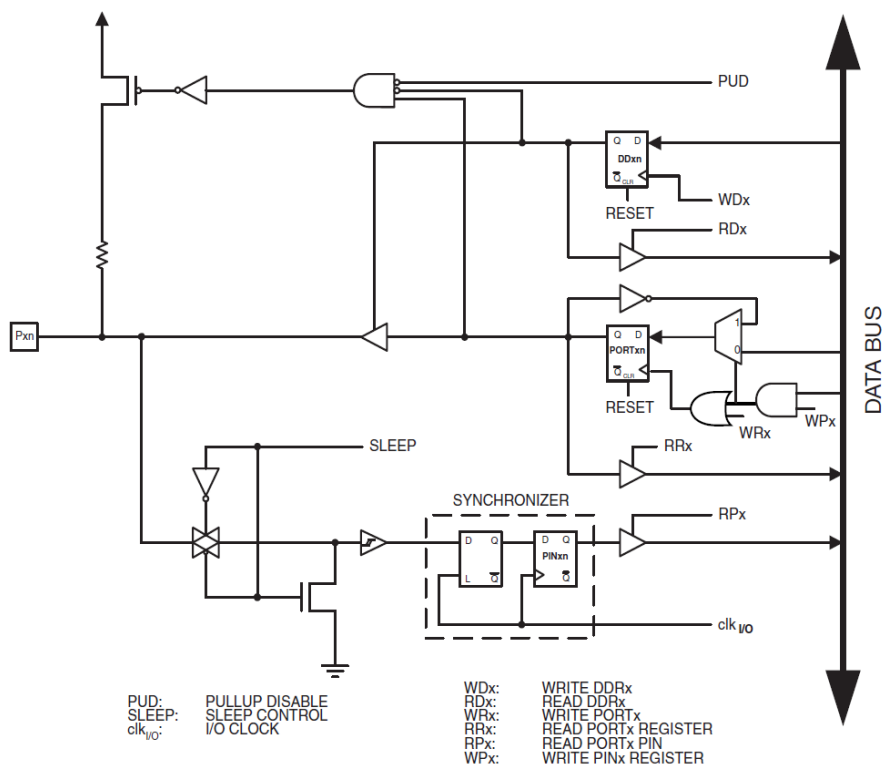
Obr. 10 Rozpis I/O pinů a jejich alternativních funkcí [2].

Ve své výchozí konfiguraci slouží vstupně/výstupní brána jako digitální rozhraní, které je schopno podle nastavení vysílat či přijímat data dle TTL logiky. Poté je vstupní logika procesoru zapojena dle logického schéma na obrázku Obr. 11.

Následuje ukázka zdrojového kódu v jazyce C pro práci s I/O branami.

```
DDRD |= (1 << DDD1);    //Nastavení pinu PD1 do výstupního režimu
PORTD ^= (1 << PD1);    //Změna hodnoty I/O pinu
```

Tyto dva jednoduché příkazy stačí ke konfiguraci pinu 3 - PD1 do výstupního režimu a na druhém řádku ke změně jeho výstupní hodnoty. Tyto příkazy a jejich logika budou detailněji vysvětlena v dalších kapitolách.

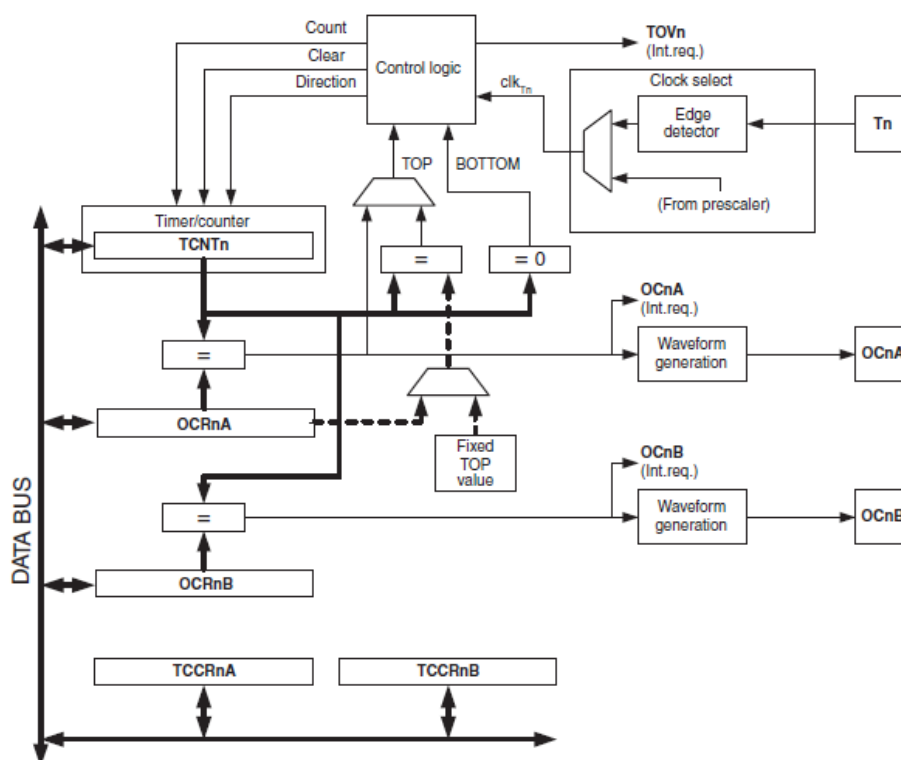


Obr. 11 Vstupně výstupní pin jako digitální port [2].

Na některé z alternativních funkcí vstupně/výstupních bran se v dalších podkapitolách podíváme podrobněji, protože jejich teoretické znalosti budou zapotřebí v řešení příkladů této kapitoly.

❑ Atmega48 – Čítač/Časovač, PWM modul

Mikroprocesor Atmega48 obsahuje celkem tři čítače, časovače. Dva 8-mi bitové a jeden 16-ti bitový.



Obr. 12 Blokové schéma čítače/časovače [2].

Jedná se o samostatné části procesoru, nezávislé na probíhajících instrukcích, které jsou schopny čítat čísla (zvětšovat svůj obsah o 1, případně svůj obsah zmenšovat). Čítač/časovač je schopen pracovat v následujících režimech:

Čítač - čítá impulzy z vnějšího zdroje Obr. 12. Pravidelným čtením a nulováním (například za 1 s) můžeme měřit frekvenci tohoto signálu. Další možnosti je odměření času načítáním vnějších pravidelných impulzů.

Časovač - načítá frekvenci, která je určena vnitřním zdrojem hodinových impulzů a případnou předděličkou zařazenou do cesty. Tento způsob použití se nejčastěji využívá pro časování pravidelných intervalů (blikání, odesílání zpráv a podobně).

Input Capture - slouží pro uložení stavu časovače / čítače. Jeho použití je ale u mikroprocesoru Atmega48 omezeno na časovač 1. Pouze tato periférie je vybavena vstupem ICP, který umožňuje uložení stavu časovače.

Output Capture - tento komparátor porovnává registr časovače se srovnávacím registrem. Při shodě změní výstup OC svůj stav podle daného nastavení.

Popis registrů čítače/časovače 0

TCNT0 - obsahuje hodnotu čítače / časovače 0

OCR0 - srovnávací registr

TCCR0 - registr pro řízení čítače / časovače 0

TIMSK - registr masek přerušení časovačů / čítačů

TIFR - registr příznaků časovačů / čítačů

SFIOR - zvláštní vstupně výstupní registr

Pro detailní popis jednotlivých registrů odkazují čtenáře na [2].

Registr TIFR

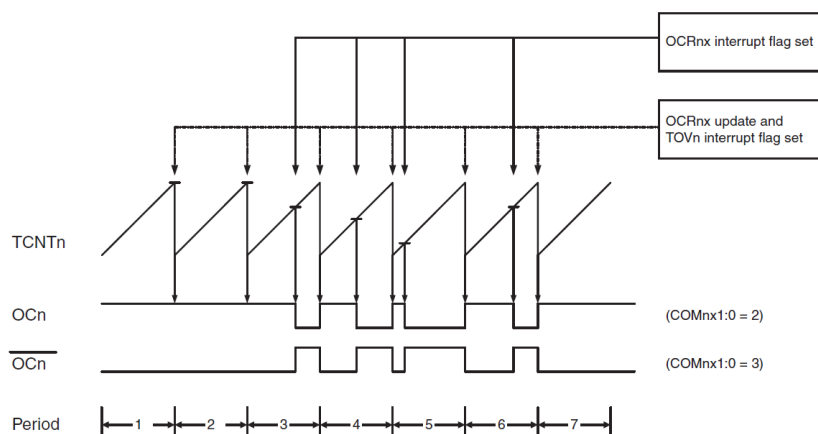
V tomto registru se automaticky nastaví příznak při vyvolání přerušení. Příznaky se nulují po obslužení daného přerušení. Pro zjednodušení představy si můžeme předem uvést, že přerušení je ten stav, kdy program je z normálního běhu přerušen nějakou událostí (vnější nebo událostí vyvolanou periferiemi či samotným procesorem). Pro vznik přerušení musí být povoleny odpovídající bity v masce. Po vzniku události, jež může být příčinou přerušení, je nastaven odpovídající příznak přerušení, program odskočí na vektor přerušení (zvlášť vyhrazená adresa v programu) a odtud je volán obslužný podprogram. Po vstupu do obslužné rutiny, jsou příznaky nulovány.

Režim rychlého PWM

Modul čítače/časovače také umožňuje práci v režimu pulzně šířkové modulace PWM (Pulse Width Modulation).

Rychlého PWM režimu můžeme dosáhnout nastavením čítače/časovače 1 do režimů 5, 6, 7, 14, 15. Pro mnoho z Vás bude jistě tento režim pochopitelnější, než funkce fázově korigovaného režimu, proto je zde probrán jako první.

Tento režim je specifický tím, že čítá ode dna (bottom) do maxima (up) (přičemž maximum lze měnit volbou režimu) a poté se vrátí zpět ke dnu. Toto nastavení a jeho vysvětlení je podpořeno obrázkem Obr. 13.



Obr. 13 Časový diagram rychlého PWM modulu [2].

V režimu 5 se jedná o osmibitovou PWM, vrcholem je tedy \$00FF. V tomto režim je možné využívat oba vývody (OC1A, OC1B). V režimu 6, mluvíme o 9-ti bitovém

rozlišení PWM pak režim 7 poskytuje PWM o rozlišení 10 bitů. Vrcholem je v tomto případě hodnota \$03FF.

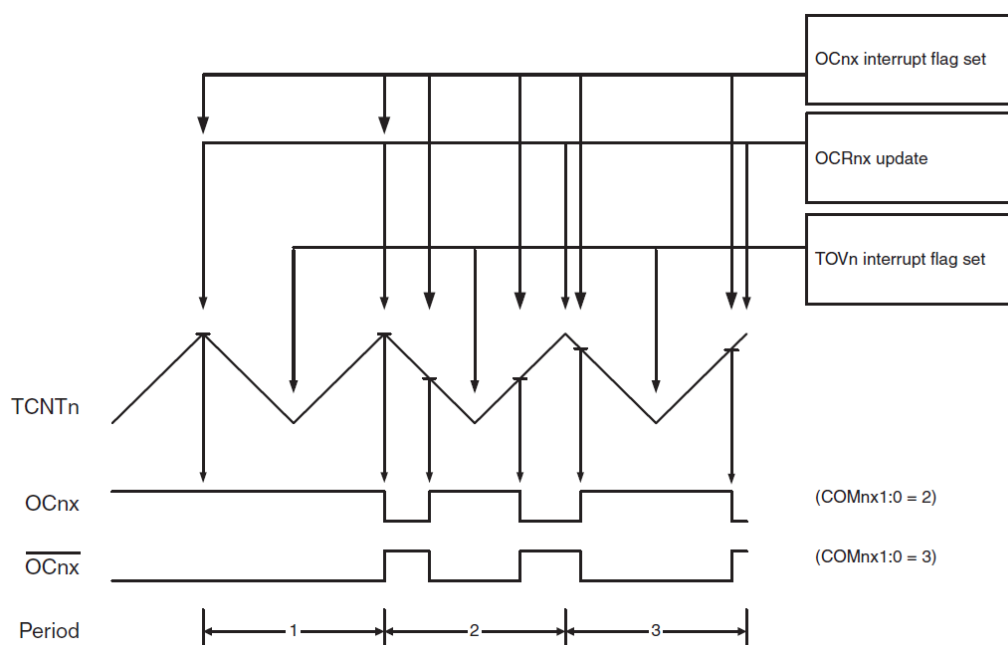
Zvláštními režimy jsou režimy 14, 15. Jedná se o PWM s vrcholem ICR1 a OCR1. V poslední jmenované situaci jsme tedy schopni použít pouze jeden vývod pro výstup PWM a to vývod OC1B. OC1A lze využít ke generování střídavy 1:1 nastavením na toggle. Jak už ale bylo napsáno dříve i přes některé nedostatky, umožňuje tento režim generování velmi rychlé PWM.

Fázově korigovaný PWM režim

Fázově korigovaný PWM režim poskytuje oproti PWM režimu 2x větší rozlišení. Je to dáno tím, že čítá nejenom nahoru, ale poté také i zpět ke dnu. Režim je možné zvolit nastavením WGM do stavů 1,2,3,10 nebo 11. Všechny tyto režimy, tak jako u rychlé PWM jsou něčím specifické.

TCNT1 s OCR1A (OCR1B) je výstup OC1A, případně OC1B vynulován. To ale platí pouze při čítání nahoru. Při čítání směrem dolů (druhá fáze) je vývod nastaven opět při rovnosti registrů. To má za následek snížení pracovní frekvence (prodloužení periody), ale umožňuje to zase přesnější nastavení střídavy.

Opět i u tohoto režimu můžeme volit mezi rozlišením na 8, 9, 10 bitů, případně využít registry ICR1 / OCR1A. Grafické znázornění funkce čítače / časovače v nastavení pro fázově korigovanou PWM si můžete prohlédnout na obrázku Obr. 14.



Obr. 14 Časový diagram fázově korigovaného PWM [2].

Příklady využití

Dalším příkladem může být ekvivalentní regulace svítivosti LED diody, či přes výkonové můstky regulace otáček motorku, apod.

Processor Atmega48 má obsahuje 10-ti bitový, 8-mi kanálový analogově digitální převodník (ADC – Analog to Digital Converter). Tento převodník je schopen měřit až osm analogových veličin současně s rozlišením 10-ti bitů, což znamená, že při referenčním napětí 5V je schopen tento převodník rozlišení

1 bit došlo na vstupu ke změně napětí o 4,88mV. Na obrázku Obr. 15 můžeme vidět ukázkové zapojení aplikace AD převodníku. Tato aplikace zapojuje elektretový mikrofon přes jednoduchý tranzistorový zesilovač do prvního vstupu AD převodníku ADC0. Na výstupu PD4 a PD5 jsou zapojeny LED diody, které jsou řízeny výstupními PWM moduly na kanálech T0 a T1. Jedná se o dva 8-mi bitové nastavitelné PWM moduly.

Tento vestavěný AD převodník má následující vlastnosti:

- 7 rozdílových vstupů (2 s možností nastavení zisku)
- nastavitelné rozlišení
- vstupní napětí a reference v celém rozsahu napájení
- nastavitelná vnitřní reference 2,56V
- možnost volby mezi ručním spouštěním a kontinuálním během
- možnost spuštění přerušování po dokončení konverze

Následuje ukázka zdrojového kódu práce s AD převodníkem, tato ukázka nepoužívá přerušování od AD převodníku, které je vždy vyvoláno po ukončení převodu, protože čtení hodnot s AD převodníku je jedinou činností, který tento program provádí a není tedy nutné provádět v hlavní smyčce programu žádné další operace.

```
#include <avr/io.h>

unsigned int read_adc(unsigned char kanal)
{
    ADMUX = 0; //Nultý kanál AD0
    ADCSRA |= 0x40; //Spuštění AD převodu
    while ((ADCSRA & 0x10)==0); //Čekání na dokončení AD převodu
    ADCSRA |= 0x10; //Zastavení AD převodu
    return ADCW; //Vrať přečtenou hodnotu
}

int main(void)
{
    unsigned int vysledek; // Proměnná pro uložení hodnoty AD převodníku
    PORTC = 0x00;
    DDRC = 0xFF; //Nastavení brány 0-tého kanálu AD převodníku do vstupního režimu
    ADMUX = ADC_VREF_TYPE; //AD převodník bere referenční napětí z referenčního vstupu
    ADCSRA = 0x87; //Zapnutí modulu AD převodníku a nastavení předěličky frekvence
    //převodu na 128
    while (1)
    {
        vysledek = read_adc(kanal); //Periodické ukládání výsledku převodu do proměnné
        //vysledek
        //Zde je možno cokoliv provádět s naměřeným výsledkem
    }
}
```

V případě, že nestačí rozlišení či rychlost převodu tohoto AD převodníku je možné zapojit externí AD převodník s vyšší vzorkovací frekvencí a s vyšším rozlišením. Příkladem takového externího převodníku může být převodník: ADS1244IDGST

MSOP10, který má rozlišení 24-bit což je při referenčním napětí 5V změna napětí o 298,02nV při změně 1 bitu.



Obr. 16 Ukázka externího AD převodníku.

Tento převodník komunikuje s hlavním procesorem pomocí jednoduchého dvou vodičového sériového rozhraní, čímž se dostáváme k poslední kapitole o mikroprocesoru Atmega48, kterou je komunikační rozhraní.

□ Atmega48 – Komunikační rozhraní (SPI, USART,...)

Mikroprocesor Atmega48 obsahuje několik komunikačních rozhraní. Tyto komunikační rozhraní jsou implementována hardwarovými moduly, které jsou vestavěny v bloku mikroprocesoru. Mikroprocesor Atmega48 obsahuje jedno rozhraní pro komunikaci RS232 – USART. Pokud ale vyžadujeme v naší aplikaci více sériových kanálů, poté není nezbytně nutné použít mikroprocesor s více USART moduly, ale lze si takovýto komunikační protokol naimplementovat softwarově.

Rozhraní USART

Jednou z často využívaných jednotek AVR mikrořadičů je jednotka sériové komunikace nazývaná USART (Universal Synchronous Asynchronous Receiver Transmitter). Tato jednotka díky své komplexnosti velmi usnadní řešení sériovou komunikací podle protokolu RS232. I přes to, že jde o velmi pomalou komunikaci z hlediska rychlosti procesoru, tak i přes to je pro nás výhodné tento typ komunikace využívat, protože nám stačí jednotku nastavit a pak už jen buď vložíme znak pro odeslání, nebo po úspěšném příjmu znaku si znak vyzvedneme - jinými slovy řečeno, nemusím se touto komunikací zabývat, většinu práce za nás udělat hardware,

Jednotka USART mcu AVR je poměrně flexibilní a dá se nastavit do mnoha různých způsobů komunikace. Při potřebě komunikace speciálním a neobvyklým způsobem, je možnost obsloužit USART vlastním způsobem. Zaměříme se zde jen na nejběžnější nastavení a jednoduchý příjem a vyslání znaku. Nyní následuje výčet možností nastavení sériové komunikace modulu USART dle protokolu RS232.

UDR - (USART Data Register)

Tab. 1 Registr UDR, obsahující přijatá data a data k odeslání pomocí modulu USART.

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Čtení)
	TXB[7:0]								UDR (Zápis)
Čtení/Zápis	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	
Poč. hodnota	0	0	0	0	0	0	0	0	

RXB: přijatý znak

TXB: odesílaný znak

UCSRA - USART Control and status Register A

Tab. 2 Registr UCSRA, pro práci s modulem USART.

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Čtení/Zápis	Č	Č/Z	Č	Č	Č	Č	Č/Z	Č/Z	
Poč. hodnota	0	0	1	0	0	0	0	0	

RXC: tento bit (příznak/flag) je nastaven, když v přijímacím bufferu je nepřečtený znak a vynuluje se po jeho přečtení. RXC se dá využít na generování přerušení, když dojde k příjmu znaku.

TXC: tento bit (příznak/flag) je nastaven, když se vysílaný znak přenesl z výstupního bufferu do odesílacího hardware. TXC se dá využít na generování přerušení, když dojde k odeslání znaku.

UCSRB - USART Control and status Register B

Tab. 3 Registr pro konfiguraci modulu USART, přerušení.

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Čtení/Zápis	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č	Č/Z	
Poč. hodnota	0	0	0	0	0	0	0	0	

RXCIE:(Receive Complete Interrupt Enable)

zapsáním 1 do tohoto bitu povolíme aby jádro AVR vyvolalo přerušení v závislosti na RXC bitu, tedy jinými slovy řečeno přerušení bude vyvoláno jen, když je tento bit = 1.

TXCIE:(Transmit Complete Interrupt Enable) zapsáním 1 do tohoto bitu povolíme aby jádro AVR vyvolalo přerušení v závislosti na TXC bitu, tedy jinými slovy řečeno přerušení bude vyvoláno jen, když je tento bit roven 1.

RXEN:(Receiver Enable)zapsáním 1 do tohoto bitu povolíme USART zahájit přijímání znaků. Normální nastavení odpovídajícího I/O pinu pro RXD je tímto přepsáno!

TXEN:(Transmitter Enable) zapsáním 1 do tohoto bitu povolíme USART zahájit odesílání znaků.

UCSZ2:(USART Character Size2)

viz praktický příklad v dalším textu. Bit nastavuje počet bitů odesílaného a přijímaného

ho znaku.

UCSRC - USART Control and status Register C / případně UBRRH

Tab. 4 Registr pro nastavení komunikačního modulu USART.

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Čtení/Zápis	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	
Poč. hodnota	1	0	0	0	0	1	1	0	

Tento registr je jednou z mnoha podivností AVR mcu. Na stejném místě v paměti jsou totiž registry dva (UCSRC a UBRRH). To, do kterého registru něco budeme zapisovat se řídí 7. bitem (URSEL). Pokud je URSEL=1, pak je registrem UCSRC, v opačném případě UBRRH. Jak metoda šetření místem je to prima, ale jinak je to hnus!

UMSEL: (USART Mode Select), když zapíšeme do tohoto bitu 0, pak je nastaven asynchronní způsob práce (defaultní způsob), v opačném případě se nastaví synchronní způsob práce.

USBS: (Stop Bit Select), zda nastavujeme počet stop bitů při vysílání. Přijímací registr na tohle z vysoka kašle! Pokud je zde 0, tak se vysílá 1 stop bit (default), pokud sem zapíšeme 1 potom se budou vysílat 2 stop bity.

UCSZ1, UCSZ0: (USART Character size), těmito bity a ještě bitem UCSZ2 z předcházejícího registru nastavujeme počet bitů v komunikaci.

Tab. 5 UBRRH, UBRL (USART Baud Rate Register H a L)

Bit	7	6	5	4	3	2	1	0	
	UMSEL	-	-	-	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
Čtení/Zápis	Č/Z	Č	Č	Č	Č/Z	Č/Z	Č/Z	Č/Z	
Čtení/Zápis	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	Č/Z	
Poč. hodnota	0	0	0	0	0	0	0	0	
Poč. hodnota	0	0	0	0	0	0	0	0	

UMSEL: jak bylo výše vysvětleno, tak pro UBRRH registr je v tomto bitu vždy 0!

UBRR: toto číslo rozložené do dvou registrů nám definuje jakou rychlostí probíhá komunikace.

Způsob výpočtu "správného" čísla UBRR

$$UBRR = \frac{F_{OSC}}{(16 * BaudRate) - 1}$$

Výše uvedený vzorec platí pro běžný způsob komunikace, tedy asynchronní normální mód! V případě potřeby komunikace jiným způsobem, si prosím ověřte výpočet správné hodnoty v příslušné technické dokumentaci - (datasheetu mikroprocesoru m48). Protože máme na UBRR číslo celkem jen 12 bitů (bit 0 až bit 11), tak může nabývat hodnot 0 až 4096 decimálně. Tím je také dána nejpomalejší rychlost, jakou můžeme komunikovat pro daný krystal. Takže pro 20 MHz krystal nikdy nebudeme moci nastavit rychlost nižší než 305Baud, což v praxi moc nevádí, protože se obvykle snažíme o dosažení vyšších rychlostí.

□ Atmega48 – přerušovací systém

V případě nutnosti řešení několika paralelních úloh tzn., že například v hlavní smyčce programu je prováděna komunikace s počítačem a mikroprocesor dále musí provádět měření z teplotního čidla pomocí vestavěného AD převodníku. V tomto případě je možno nastavit přerušování od AD převodníku tak aby byl přerušen chod hlavního programu v případě získání vzorku z AD převodníku. Nastavení přerušování se provádí v registrech AD převodníku:

ADCSRA(registr) – ADIEN(bit AD Interrupt ENable)

V případě nastavení tohoto bitu je po dokončení převodu na modulu AD převodníku vyvoláno přerušování, které pozastaví činnost hlavního programu a odskočí na příslušné místo v paměti mikroprocesoru, kde je vykonán kód obsluhy přerušování.

Ukázka kódu obsluhy přerušování AD převodníku je vidět na následujícím kódu:

```
ISR(ADC_vect)
{
    char conv_value;
    if(i > ADC_SAMPLES) { i=0; }
    adc_values[i] = ADCL; //Nižší bit 10-ti bitového výsledku je uložen do 16-ti bitové
    //proměnné typu int.
    adc_values[i] += (ADCH<<8); //Vyšší bit je zarotován a uložen
}
```

V jazyce C je příslušné místo v paměti vyznačeno pomocí makro instrukce *ISR(adresa_vektoru_preruseni)*

Místa v paměti, kde program odskočí, jsou vyznačena v tabulce Tab. 6.

Tab. 6 Přerušovací vektory mikroprocesoru

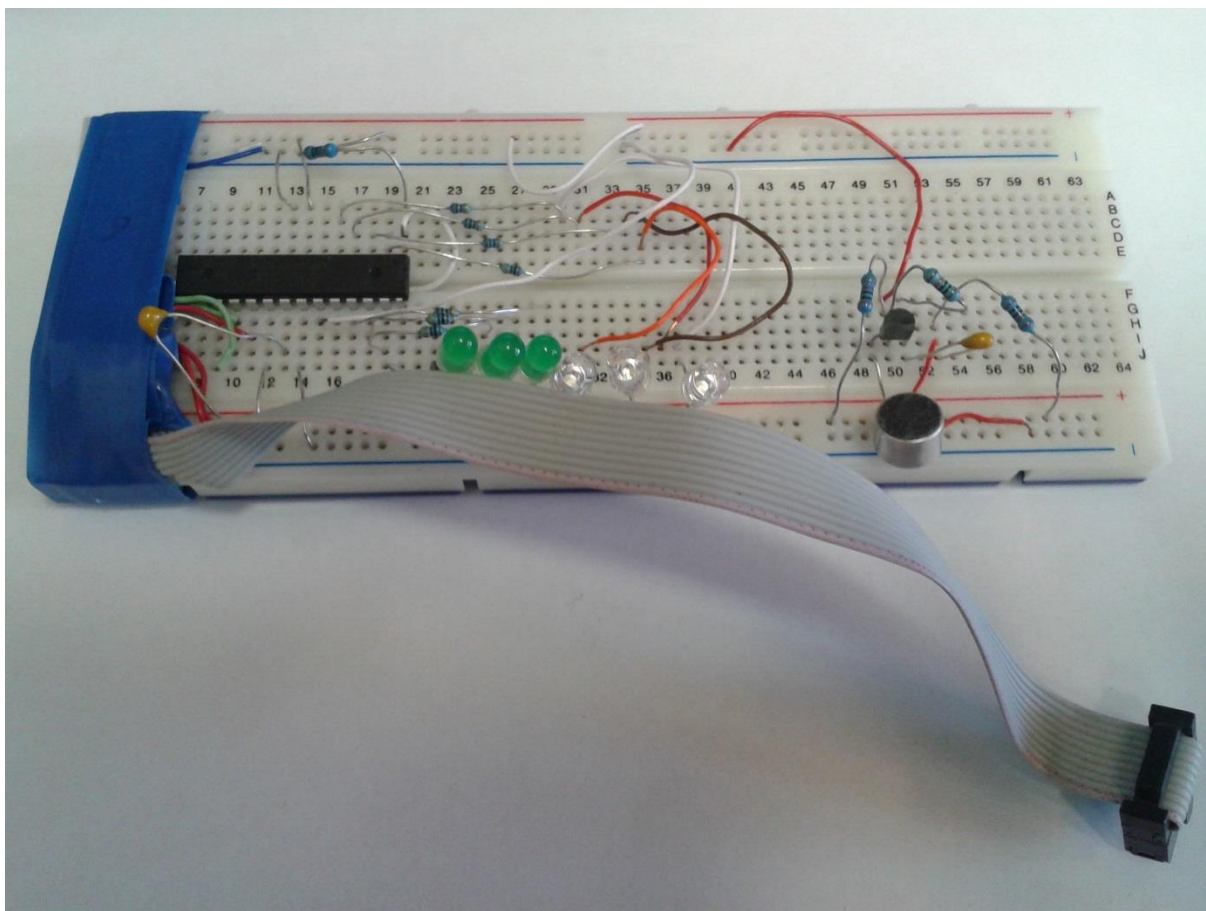
Vector no.	Program address	Source	Interrupt definition
1	0x000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x001	INT0	External interrupt request 0
3	0x002	INT1	External interrupt request 1
4	0x003	PCINT0	Pin change interrupt request 0
5	0x004	PCINT1	Pin change interrupt request 1
6	0x005	PCINT2	Pin change interrupt request 2
7	0x006	WDT	Watchdog time-out interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 compare match A
9	0x008	TIMER2 COMPB	Timer/Counter2 compare match B
10	0x009	TIMER2 OVF	Timer/Counter2 overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 capture event
12	0x00B	TIMER1 COMPA	Timer/Counter1 compare match A
13	0x00C	TIMER1 COMPB	Timer/Counter1 compare match B
14	0x00D	TIMER1 OVF	Timer/Counter1 overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 compare match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 compare match B
17	0x010	TIMER0 OVF	Timer/Counter0 overflow
18	0x011	SPI, STC	SPI serial transfer complete
19	0x012	USART, RX	USART Rx complete
20	0x013	USART, UDRE	USART, data register empty
21	0x014	USART, TX	USART, Tx complete
22	0x015	ADC	ADC conversion complete
23	0x016	EE READY	EEPROM ready
24	0x017	ANALOG COMP	Analog comparator
25	0x018	TWI	2-wire serial interface
26	0x019	SPM READY	Store program memory ready

□ Programování aplikací pro mikroprocesor Atmega48 a dalších

Co je zapotřebí k programování mikroprocesoru AVR?:

- Mikroprocesor – V našem případě se jedná o mikroprocesor Atmega48, který je zapojen na nepájivém poli, viz Obr. 17,
- programátor – Programátor, si lze vytvořit vlastní či zakoupit. Poměrně snadným a levným řešením je projekt USBASP (<http://www.fischl.de/usbasp/>), viz Obr. 18, způsob propojení s cílovým mikroprocesorem a způsob nahrání programu je popsán na stránkách: <http://www.elektronovinky.cz/konstrukce/usbasp-prakticke-zkusenosti-s-programatorem-avr>,
- vývojové prostředí – Firma Atmel nabízí zdarma dostupné vývojové prostředí Atmel Studio v současné verzi 6, stáhnutelné odsud (<http://www.atmel.com/tools/atmelstudio.aspx>). Pro možnost vývoje v programovacím jazyce C, je nutné si ještě stáhnout sadu knihoven, kompilátoru, linkeru, assembleru a dalších nástrojů pro tuto platformu. Tento balík se jmenuje WinAVR a je stažitelná odsud (<http://winavr.sourceforge.net/>). Obsahem tohoto balíku je cross-kompilátor,

linker a assembler. Pojem cross-kompilátoru vyznačuje, že program pro jinou platformu (AVR) je překládán na procesoru s jinou platformou (x86).



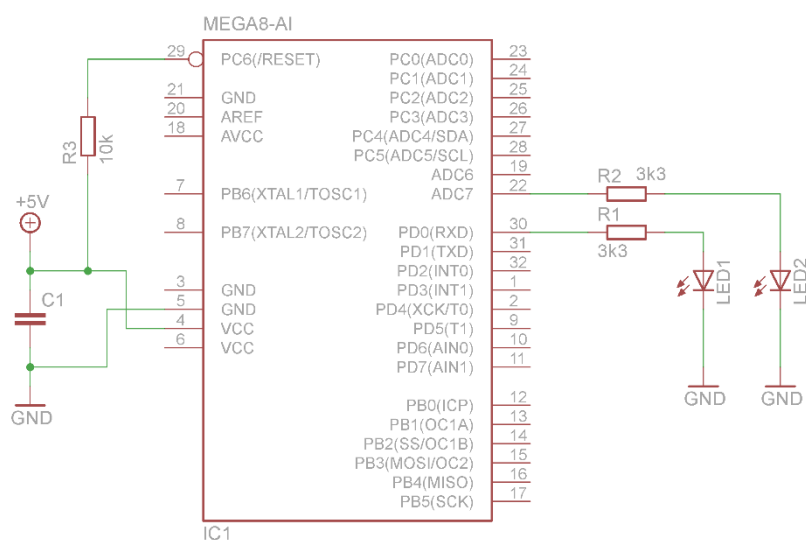
Obr. 17 Ukázka zapojení schéma dle Obr. 15 na nepájivém poli.



Obr. 18 Programátor USBasp.

Příklad

V tomto prvním příkladu si připojte k libovolnému procesoru AVR, LED diodu, přes ochranný proudový odpor 3k3 a vytvořte aplikaci, která touto LED diodou bude blikat. Pro algoritmus vytvářející zpoždění mezi zhasnutím a rozsvícením LED diody použijte metodu zpožďovacích smyček nebo zpoždění pomocí čítače/časovače. Obvod zapojte dle schéma na obrázku Obr. 19. Tento obvod můžete volitelně doplnit o tlačítko, kterým budete ovládat frekvenci blikání LED diod.



Obr. 19 Schéma zapojení obvodu pro příklad 1.



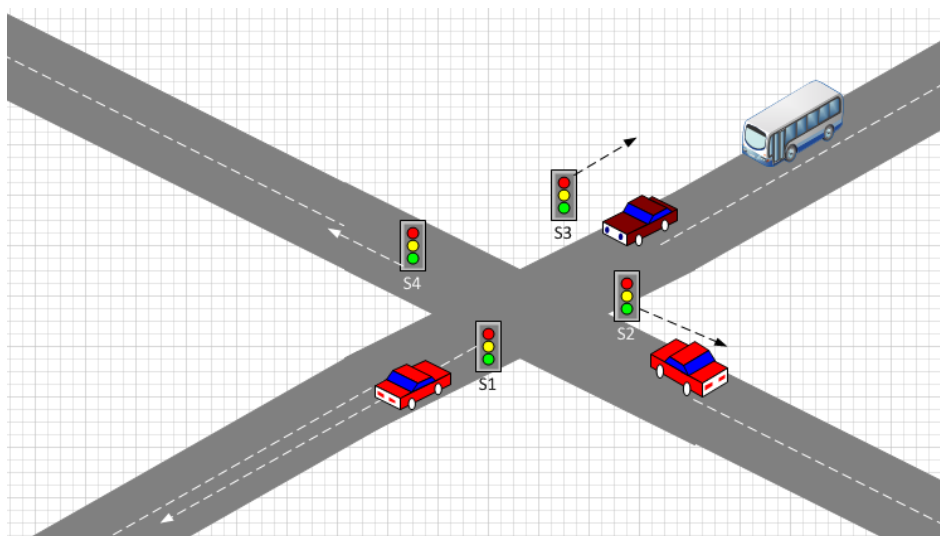
Příklad

V druhém příkladu si vyzkoušíme jednoduchou aplikaci s využitím AD převodníku. VU metr s využitím 8-mi diod, využití AD převodníku a zobrazení amplitudy zvuku měřeného elektretovým mikrofonom na LED diodách. Následuje schéma obvodu se zapojením elektretového mikrofonu včetně předzesilovače. Zapojení obvodu na nepájivém poli (Obr. 17) dle schéma na obrázku Obr. 15.



Příklad

V třetím a posledním příkladu vytvořte aplikaci, která bude simulovat blikání semaforů na křižovatce dle obrázku Obr. 20 a pravidel v tabulce Tab. 7. Časy pro zpoždění mezi semaforů tentokrát řešte pomocí čítače nebo zpožďovací smyčky.



Obr. 20 Náčrt simulované křižovatky.

Tab. 7 Pravidla, podle kterých se řídí provoz

Semafor	ČAS [s]	AKCE
S1 + S3	12	JEŘ (Zelená)
S1 + S3	2	BLIKÁNÍ (Zelená) - > STOP (Červená) && S2 + S4 BLIKÁNÍ (Červená) -> JEŘ (Zelená)
S4 + S2	10	JEŘ (Zelená)
S4 + S2	2	BLIKÁNÍ (Zelená) - > STOP (Červená) && S1 + S3 BLIKÁNÍ (Červená) -> JEŘ (Zelená)



Shrnutí pojmů

Mikroprocesor, Atmel, AVR, Atmega, Atmel studio, Kompilátor, Linker, Assembler, Časovač, Čítač, Analogově digitální převodník, Paměť, ROM, Flash.



Otázky

1. Co znamená zkratka SREG?
2. Jaké typy pamětí u mikroprocesoru Atmega48 znáte?

3. Jaké znáte režimy čítače/časovače mikroprocesoru Atmega48?
4. Jaké je rozlišení AD převodníku mikroprocesoru Atmega48?
5. K čemu slouží u mikroprocesoru přerušení?



Další zdroje

[2] Atmel Corporation. 8-bit Atmel Microcontroller with 4/8/16KBytes In-System Programmable Flash. May 2011.

[3] Váňa, Vladimír. Atmel AVR programování v jazyce C. BEN - technická literatura, 2003. ISBN: 80-7300-102-0

4. POKROČILÉ TECHNIKY PROGRAMOVÁNÍ V JAZYCE C

Tato kapitola popisuje a na příkladech vysvětluje pokročilé techniky programování v jazyce C. Kapitola je rozdělena do části popisující pokročilé použití ukazatelů, dynamickou alokaci paměti a nakonec tvorbu vlastních často používaných datových typů ze světa objektového programování, dynamických seznamů typu List a LinkedList.

4.1. Ukazatelé a jejich speciální případy



Čas ke studiu:

4 hodiny (s příklady 7 hodin).



Cíl:

Po prostudování tohoto odstavce pochopíte a budete umět

- speciální případy ukazatelů.



Výklad

Funkce ukazatelů v jazyce C je objasněna v předchozích kapitolách. Ukazatel v sobě obsahuje adresu v paměti, kde je uložena hodnota proměnné, na niž ukazuje a zároveň má v sobě uložen datový typ místa v paměti, na který ukazuje. Jestliže tedy máme příklad:

```
int main()
{
    int x = 5;
    int *p_x = &x;
}
```

Potom `*p_x` je ukazatel, který obsahuje adresu proměnné „x“ a datový typ „int“. Tak jako tento ukazatel ukazuje na hodnotu proměnné, může také ukazovat na pole či strukturu a další vlastní a složitější datové typy. Mimo jiné ale také může ukazovat na funkci. Funkce a její tělo je také uloženo v paměti a její začátek je uložen na určité adrese. Další výhodou tohoto přístupu je také to, že stejně jako můžeme funkci jako parametr předat ukazatel na proměnnou můžeme také předat funkci jako parametr ukazatel na funkci. Tento ukazatel může být například ukazatelem na matematickou funkci implementující určitý algoritmus, který je vyvolán ve vykonávající funkci. Tato zvolená matematická operace je pak předána jako parametr vykonávací funkci.

□ Ukazatel na funkci

Kromě klasických datových ukazatelů, s kterými jsme se doposud setkávali, nabízí jazyk C i ukazatele kódové. Ty ukazují na určitý úsek kódu (funkci) programu, který pak můžeme spustit nepřímo, právě pomocí těchto ukazatelů. Nejlépe si to ukážeme přímo na příkladu.

Uvažujme, že máme nadefinovanou tuto funkci pro sčítání čísel double:

```
double secti(double a, double b)
{
    return a+b;
}
```

Tuto funkci lze volat normálně, nebo lze vytvořit ukazatel na tuto funkci a volat ji pomocí tohoto ukazatele. Formální zápis definice takového ukazatele vypadá následovně:

typ (*identifikátor)(seznam_typů_parametrů)

Typem rozumíme datový typ, který vrací funkce, na kterou bude ukazovat náš pointer. Identifikátor je pak označením tohoto nového pointeru a v seznamu_typů_parametrů jsou, v odpovídajícím pořadí, uvedeny všechny typy parametrů odkazované funkce. Tento seznam sice lze ponechat prázdný, ale kompilátor by v tomto případě neměl informace o typech parametrů funkce. Pokud bychom pak došlo k volání funkce, nepřímo přes ukazatel, se špatnými typy parametrů, mohlo by to vést ke špatné funkci programu. Proto se velice doporučuje typy těchto parametrů uvádět.

Ukazatel na výše uvedenou funkci lze tedy definovat následovně:

double (*pf)(double, double);

Tento ukazatel nyní může ukazovat na jakoukoliv funkci, která vrací typ double a jako parametry přebírá dvě proměnné typu double. Pro čtenáře znalé jazyka C# a objektového programování lze ukazatel na funkci přímo porovnat s delegátem.

K přiřazení funkce ***secti***, která byla deklarována do ukazatele na funkci ****pf*** poté postačí jednoduchý zápis:

```
pf = secti;
```

❑ Volání funkce pomocí ukazatele

Samotné volání funkce pomocí ukazatele se posléze řídí podle jedné z následujících notací

(*identifikátor_ukazatele)(seznam_skutečných_parametrů);

nebo

identifikátor_ukazatele(seznam_skutečných_parametrů);

Volání funkce ***secti***, na kterou ukazuje ukazatel ***pf*** s parametry 15.5 a 16.8 pomocí ukazatele bude vypadat následovně:

```
pf(15.5, 16.8);
```

▣ Ukazatel na funkci jako parametr funkce

Stejně jako lze vytvářet pole proměnných a ukazatelů lze také vytvářet pole ukazatelů na funkci. V této kapitole si ukážeme jak vytvořit pole ukazatelů na funkci a toto pole posléze předat funkci, která vrátí pouze ten ukazatel na funkci, kde výsledek této funkce pro hodnotu 0.0 bude 0.0. Pro tyto výpočty využijeme goniometrické funkce z knihovny math (sin, cos, tan). Nejprve si ale na příkladu ukážeme jak předat funkci ukazatel na funkci jako parametr. Uvažujme funkci, které budeme předávat jako první parametr matematickou funkci, která bude provedena nad dalším parametrem funkce. Tímto způsobem lze za běhu programu měnit způsob vykonávání funkce, jejíž vykonávání se bude měnit podle toho, jaká funkce bude předána jako ukazatel. Definice takovéto funkce bude vypadat následovně.

```
double math_op(double (*fce)(double), double parameter);
```

Definici takovéto funkce čteme od leva doprava následovně. Funkce math_op je funkce se dvěma parametry, která vrátí typ double jako výsledek dané funkce definované ukazatelem na funkci. Prvním parametrem je ukazatel na funkci, která vrátí typ double a má parametr typu double, druhý parametr je číselné hodnoty nad kterou budou provedeny mat. operace v ukazateli na funkci.

Volání takovéto funkce může vypadat následovně.

```
#include <math.h>
math_op(sin, 10.5);
```

V této ukázce bude volána funkce sin ze systémové knihovny math, s parametrem 10.5.

Tento příklad pouze ilustruje způsob předání ukazatele funkce jako parametr funkce. Ve své samotné funkci nepřináší nějakou výhodu, ale uvažujme, že bychom chtěli provést více matematických operací nad stejnou hodnotou a vrátit výsledek součtu těchto parametrů. Toto lze vyřešit, jestliže jako parametr Funkci předáme pole s ukazateli na funkce, které budou vykonány nad touto hodnotou. Předpis takovéto funkce by mohl vypadat následovně:

```
double math_ops(double (*pole_fci[])(double), double parameter);
```

Jedná se zde o funkci se dvěma parametry, kde první parametr je pole ukazatelů na funkci vracející double a jedním parametrem typu double a druhým parametrem typu double. Funkce vrátí typ double.

Samotné volání funkce by vypadalo následovně:

```
#include <math.h>
double (*math_funkce[]) (double) = { sin, cos, tan };

math_ops(math_funkce, 10.5);
```

V závislosti na implementaci by tato funkce provedla operaci s parametrem 10.5 nad všemi funkcemi předanými v poli. Řešení takovéhoho problému je v řešených příkladech k této kapitole.

Poslední nejsložitější variantou ukazatelů na funkce je pokud by uvažujeme-li předchozí příklad, funkce vracela ne hodnotu výsledku dané funkce ale samotný ukazatel na funkci, jejíž výsledek je roven v bodě 1.0 roven 0. Tato podmínka jak víme, platí pro funkci ***sin*** a ***tan***. Předpis takovéto funkce by vypadal následovně:

```
double (*math_ops(double (*pole_fci[])(double), double parameter))(double);
```

Jedná se tedy o funkci, která má dva parametry. První parametr je pole ukazatelů na funkce s parametrem typu double vracející typ double. Druhý parametr je číslo typu double. Tato funkce vrací ukazatel na funkci s parametrem typu double, vracející hodnotu typu double.

Po pochopení i takto komplexního zápisu již lze pochopit jakkoliv složitý zápis deklarace funkcí v jazyce C.

Příklad

Vytvořte aplikaci, která pomocí pole ukazatelů na funkci provede součet výsledků goniometrických funkcí z pole ukazatelů na funkci nad zadaným parametrem. Pro začátek lze použít příklad, který částečně řeší zadání tohoto příkladu.



4.2. Dynamická alokace paměti



Čas ke studiu:

4 hodiny (s příklady 8 hodin).



Cíl:

Po prostudování tohoto odstavce pochopíte a budete umět

- popsat principy a použít dynamické alokace paměti.



Výklad

Dynamicky alokovat paměť znamená možnost "přidělit" si kousek paměti pro tvořený program ze systémové paměti. Tuto paměť dostane přidělenou z takzvané "haldy" (angl. heap). Ostatní statické proměnné (které byly doposud vytvářeny) se ukládaly na "zásobník" (angl. stack). O správu zásobníku se vlastní program nemusel starat. Nemá smysl alokovat malé množství paměťového prostoru, protože samotná režie na tuto alokaci a posléze systémových prostředků je časově náročná. Bude tedy alokováno více paměti a zároveň se bude také pracovat s poli a ukazateli.

Možnost přidělovat programu paměť je velice důležitá a bez ní lze napsat pouze velice jednoduché programy, u kterých stačí konečný počet proměnných. Představme si nejjednodušší situaci, kdy je zapotřebí napsat obyčejný textový editor. Jak by byla definována proměnná, do které by byl ukládán text? Pole? Ano je to možnost, ale s jakou velikostí? 1000 prvků?

```
char text[1000];
```

Program s takovým polem znaků bude fungovat pouze do té doby, dokud bude text mít maximální délku do 1000 znaků. Samozřejmě je možno tomuto poli dát velikost 1000000 ale není to zase moc? A velikost obsazené paměti by mohla zbytečně zpomalovat chod systému a také naší aplikace. Pole o velikosti 1000000 zabere sice asi 1MB v paměti systému což není v dnešní době mnoho, ale pravidlo všudypřítomné optimalizace by měl mít na mysli každý programátor. Zabírat staticky takovou oblastí paměti jenom pro jistotu toho aby aplikace fungovala, je neoptimální. V těchto případech lze výhodně využít dynamické alokace paměti, u které lze programu přidělit tolik místa v paměti kolik aktuálně potřebuje.

□ Dynamická alokace paměti

Prvním krokem je přidáním hlavičkového souboru malloc.h do psané aplikace.

Tento hlavičkový soubor obsahuje informace o funkcích, které jsou nutné pro dynamickou správu paměti. První z nich je funkce malloc(). Tato funkce vrací

ukazatel na začátek paměti, kterou alokovala. Jejím parametrem je číslo udávající počet bytů k alokaci. Vracený ukazatel je doporučeno přetypovat na takový typ dat, jakým bude paměť používána. Pro určení, kolik bajtů paměti se alokovalo lze použít funkci `sizeof()`.

```
#include <malloc.h>

int *pamet, i;
pamet = (int*)malloc(5*sizeof(int)); // alokce pameti

for(i=0; i<5; i++){
    pamet[i] = i;
}

printf("Do uvolnene pameti byla ulozena tato cisla: ");
for(i=0; i<5; i++){
    printf("%d ", pamet[i]);
}
```

V předcházejícím příkladu se nejprve vytvořil ukazatel s názvem `pamet`, kterému jsme následně funkcí `malloc()` přidělili adresu počátku paměti, kterou funkce `malloc()` alokovala. Uvnitř funkce je napsáno `5*sizeof(int)`. `sizeof()` vrátí počet bytů, které zabírá datový typ uvedený v závorkách. Program mohl také obsahovat `5*4`, protože víme, že v překladači na OS Windows, je `int` reprezentován 4-mi bajty. Pokud by ale tento program překládán překladačem, u kterého `int` zabírá 2 bajty, pak by se zbytečně alokovalo 20 bytů paměti a využil pak pouze 10. Opačný případ by byl horší, protože alokovaná paměť by ve skutečnosti nedostačovala pro uložení všech dat a přepisovala by se data v neznámé oblasti, což může skončit pádem programu. Přepisování neznámé paměti je tedy nutné se vyvarovat. Ukazatel vracený funkcí `malloc()` je přetypován na datový typ ukazatele na `int` pomocí `(int*)`. Takto alokovanou paměť lze posléze využívat normálně k práci jako pole 5-ti celočíselných hodnot typu `int`. První příkaz cyklu `for` naplní pole čísly od 0 po 4 a druhý `for` je vypíše na obrazovku.

□ Kontrola úspěšné alokace paměti

Pro kontrolu úspěšné alokace paměti (neúspěch může být například při nedostatku paměti) lze využít skutečnosti že funkce `malloc` při nespěšné alokaci paměti vrací ukazatel typu `NULL`. Samotná kontrola poté může vypadat následovně:

```
int *pamet;

if((pamet = (int*)malloc(5*sizeof(int))) == NULL){
    printf("Chyba pri alokaci pameti, program bude ukoncen");
    exit(0);
}
```

□ Uvolnění paměti

Při ukončení práce s obsazenou paměti je nutné obsazenou paměť uvolnit pro další použití jinými aplikacemi či operačním systémem. Jestliže nedojde k uvolnění alokované paměti, může dojít k situaci, že systém nebude mít dostupné žádné volné prostředky a v extrémních případech může dojít k „pádu“ celého systému. Pro

uvolnění alokované paměti se používá funkce **free()**. Uvolnit je možné pouze ukazatel, který byl inicializován pomocí funkce **malloc**. Jestliže tento ukazatel nebyl inicializován dojde k „pádu“ programu. Je proto výhodné do uvolněného ukazatele zapsat hodnotu NULL a na tuto hodnotu posléze před dalším uvolněním testovat.

```
free(pamet);  
pamet = NULL;
```

Příklad

Vytvořte aplikaci, která dynamicky alokuje paměť podle velikosti zadané uživatelem a naplní je hodnotami funkce **sin**. Hodnoty této funkce posléze i s adresami uložených hodnot v paměti vypište.



4.3. Vlastní speciální často používané datové typy, List, LinkedList



Čas ke studiu:

2 hodiny (s příklady 4 hodiny).



Cíl:

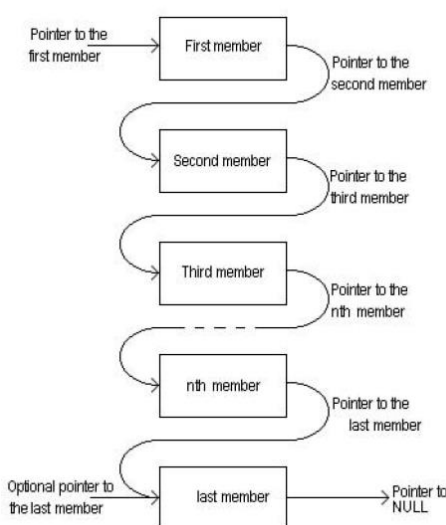
Po prostudování tohoto odstavce pochopíte a budete umět

- vytvořit vlastní speciální často používané datové typy, List, SortedList.



Výklad

Datové typy List a LinkedList jsou datové typy převážně známé programátorům z prostředí objektového programování. Jedná se o seznamy „pole“, které se dynamicky zvětšují podle počtu prvků, které jsou do nich vkládány. Datový typ LinkedList je takový datový typ kde každý prvek tohoto seznamu obsahuje adresu na další prvek v seznamu. Hlavní výhoda tohoto seznamu se projevuje při zaplnění mnoha prvků a vyhledávání či zisávání hodnot v pevném pořadí od začátku či konce. První prvek tohoto seznamu obsahuje adresu prvního prvku seznamu a poslední prvek obsahuje NULL čímž definuje konec tohoto seznamu. Lze kromě ukazatele na začátek tohoto seznamu uchovávat také ukazatele na konec tohoto seznamu a poté je možnost efektivně přidávat prvky nejen na začátek seznamu ale také na konec. Následující diagram, viz (Obr. 21) ilustruje jednoduchý seznam typu **LinkedList**.



Obr. 21 Funkční diagram LinkedListu [4]

Tento dynamicky rozšiřitelný seznam je poměrně jednoduché implementovat v jazyce C. V moderních jazycích jako je Java nebo C# najdeme tyto typy vestavěné v základních knihovnách, které tyto jazyky využívají. Implementace v jazyce C by mohla vypadat takto:

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

struct list_el {
    int val;
    struct list_el * next;
};
typedef struct list_el item;

void main() {
    item * curr, * head;
    int i;
    head = NULL;

    for(i=1;i<=10;i++) {
        curr = (item *)malloc(sizeof(item));
        curr->val = i;
        curr->next = head;
        head = curr;
    }

    curr = head;
    while(curr) {
        printf("%d\n", curr->val);
        curr = curr->next ;
    }
}
```

V této ukázce se jedná o LinkedList pro datový typ integer (int). Pro položku listu (item) je použita struktura list_el, která obsahuje samotnou hodnotu (val) a ukazatel na další položku v seznamu (*next). Posléze je v příkladu list naplněn hodnotami 1 – 10. Ke konci je položka curr nastavena na konec seznamu (head) a postupně je tento seznam procházen položku po položce pomocí ukazatele na další položku v seznamu uložené v ukazateli struktury (next).

Příklad

Implementujte ukázkovou aplikaci, která bude obsahovat seznam typu linked list, do kterého bude moci uživatel za běhu programu moc přidávat a zobrazovat n posledních hodnot podle zadání do konzole. Datový typ ukládaných hodnot vyberte dle vlastního uvážení



Shrnutí pojmů

Ukazatel, ukazatel na funkci, dynamická alokace paměti, halda (heap), malloc, free, spřažený seznam (linked list)



Otázky

1. K čemu lze využít ukazatel na funkci?
2. Jakým způsobem lze definovat pole ukazatelů na funkce, které vrací datový typ int a jako parametr bere dvě celočíselné hodnoty (int)?
3. Jaké jsou hlavní výhody dynamické alokace paměti?
4. Jaká je vazba heap na dynamickou alokaci paměti?
5. K čemu se používá funkce malloc?
6. K čemu se využívá a kde je efektivní využívat spřažený seznam (linked list)?



Další zdroje

[4] Peter D. Hispan. Advanced C. Sams Publishing, Indianapolis, USA, 1992. ISBN: 0-672-30168-7

5. JAZYK C, PROGRAMOVÁNÍ V PROSTŘEDÍ UNIX – LIKE OPERAČNÍCH SYSTÉMŮ

Tato kapitola popisuje, jednu z hlavních oblastí dnešního využití jazyka C. Jsou to Unix-like operační systémy u kterých a díky kterým jazyk C vznikl a také částečně dospěl do dnešní podoby a specifikace C11.

5.1. Jazyk C, a Unix-like operační systémy



Čas ke studiu:

4 hodiny (s příklady 6 hodin).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat základy Unix-like operačního systému,
- použít jazyk C v prostředí Unix-like operačního systému.



Výklad

Programovací jazyk C byl vytvořen v Bellových laboratořích a od počátku byl neodmyslitelně spjat s operačním systémem (OS) Unix, pro který byl také vytvořen. Jeho původní verze, která byla psána v jazyce Assembler byla jen velice těžce udržitelná při obměně platform na kterých tento OS měl jet. Pro účely přenositelnosti mezi platformami a univerzálnosti byl vytvořen právě jazyk C. S OS systémem Unix a jeho nástupci, kteří vychází z jeho ideologie a také systémové topologie jako například nejznámější OS Linux a iOS, je jazyk C úzce spjat dodnes. V této kapitole si ukážeme na ilustrovaných a komentovaných příkladech, kde se v dnešní době nachází místo tohoto poměrně starého programovacího jazyka, které si udrželo své místo po boku OS u kterého vznikl.

□ OS Unix

UNIX je v informatice ochranná známka operačního systému vytvořeného v Bellových laboratořích americké firmy AT&T v roce 1965. Většina současných operačních systémů je unixovými systémy různou měrou inspirována. Samotný UNIX byl inspirován (nedokončeným) systémem Multics.

Označení UNIX je ochranná známka, kterou v současné době vlastní konsorcium The Open Group a mohou ji používat pouze systémy, které jsou certifikovány podle *Single UNIX Specification*. Existují různé systémy, které jsou s UNIXem v různé míře kompatibilní, ale nemohou nebo nechtějí platit licenční poplatky, a proto často používají varianty názvů, které na název UNIX odkazují (například XENIX, MINIX,

Linux), ale mohou se jmenovat i jinak (například BSD varianty OpenBSD, NetBSD, ale též Mac OS X atd.). Souhrnně jsou tyto systémy označeny jako unixové systémy (anglicky unix-like). Pod pojmem „tradiční Unix“ se rozumí operační systém, který svojí charakteristikou odpovídá systémům Version 7 Unix nebo UNIX System V.

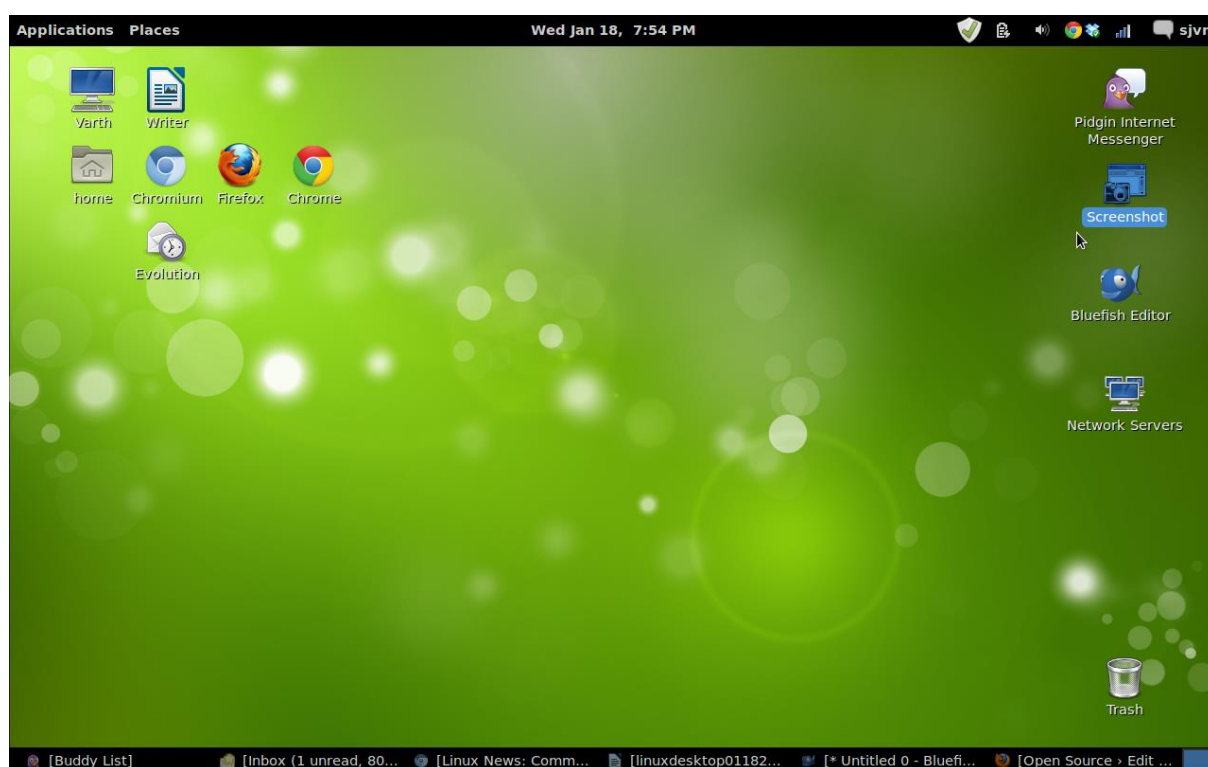
□ OS Linux, iOS

Operační systém Linux používá unixové jádro, které vychází z myšlenek Unixu a respektuje příslušné standardy POSIX a Single UNIX Specification. Název je odvozen z křestního jména jeho tvůrce Linuse Torvaldse a koncovka písmenem „X“ odkazuje právě na Unix (podobně jako XENIX, Ultrix, IRIX, AIX a další UN*Xy).

Jádro Linuxu je víceúlohové, takže může najednou běžet více oddělených procesů (spuštěných programů), které se v rychlém sledu střídají na procesoru, čímž vzniká dojem jejich současného běhu (tzv. multitasking). Zároveň se jedná o víceuživatelský operační systém, na kterém může pracovat více uživatelů zároveň. Proto jsou zavedeny uživatelské účty, ke kterým je přístup chráněn nějakým autentizačním mechanismem (např. jméno + heslo). K tomu jsou též zavedena přístupová oprávnění, která umožňují omezit přístup jednotlivých uživatelů (resp. jejich procesů) k souborovému systému (tj. souborům a adresářům).

□ Jazyk C v prostředí OS Linux

Vzhledem ke kořenům a historii jazyka C je jeho místo v operačním systému Linux (Obr. 22) jasně určené. V jazyce C je v operačním systému Linux vytvořeno spoustu aplikací a ovladačů pro různá zařízení pro tento systém. Jádro tohoto operačního systému je převážně psáno v jazyce C a posléze také assembler kódu v určitých časově kritických sekcích. V této kapitole bude předvedena aplikace tohoto jazyka v OS linux pro implementaci síťové komunikace na úrovni návrhového vzoru klient – server.



Obr. 22 snímek typického prostředí OS Linux (Linux Mint)

5.2. GNU GCC



Čas ke studiu:

½ hodiny.



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat sadu kompilačních nástrojů GCC (GNU Compiler Collection).



Výklad

GNU Compiler Collection (zkráceně GCC) je sada překladačů vytvořených v rámci projektu GNU. Původně se jednalo pouze o překladač programovacího jazyka C (a zkratka tehdy znamenala GNU C Compiler), později byly na stejném společném základě vytvořeny překladače jazyků C++, Fortran, Ada a dalších.

Původním autorem GCC je Richard Stallman, který ho roku 1987 vytvořil jako jednu ze základních částí svého projektu GNU; dnes projekt zastřešuje nadace FSF. GCC je šířen pod licencí GNU GPL a stal se již de facto standardním překladačem v open source operačních systémech unixového typu, ale používá se i v některých komerčních operačních systémech, např. na Mac OS X. Existují také jeho portace pro Microsoft Windows (např. MinGW).

5.3. Adresářová struktura OS Linux



Čas ke studiu:

1 hodina (s příklady 5 hodin).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat a pracovat s adresářovou strukturou operačního systému Linux (Unix-like operační systém).



Výklad

Pro pochopení základní práce s OS Linux je nutná orientace a znalost adresářové struktury. Například pro získání informací o hardware a připojených zařízeních či stavu jádra OS slouží adresářová struktura /proc. Do této struktury jsou namapovány informace s jádra OS (informace z ovladačů připojeného HW a periferních zařízení). Následuje výpis jednotlivých adresářů s popisem jejich funkčnosti:

- **/** - kořen souborového systému, začátek stromové struktury
- **/bin** - základní spustitelné soubory pro použití všemi uživateli
- **/boot** - zde je umístěno jádro (kernel) systému a jeho mapa, initrd, soubory zavaděče (boot loader) GRUB
- **/dev** - **soubory v tomto adresáři reprezentují jednotlivá fyzická zařízení nebo pseudozařízení systému. Také jsou zde další speciální soubory.**
- **/etc** - globální konfigurační soubory systému
- **/home** - domovské adresáře uživatelů
- **/lib** - základní sdílené knihovny systému, mapování klávesnice a konzolové fonty, moduly pro jádro systému (kernel)
- **/lost+found** - ztracené a opravené soubory po chybách FS (ext2,ext3)
- **/mnt** - do místních podadresářů se připojují další souborová zařízení, např. do "/mnt/floppy" disketa, do "/mnt/cdrom" CD ap.
- **/opt** - zde bývají SW aplikace, které nejsou standardní součástí distribuce
- **/proc** - **soubory nastavení a stavu systému a jednotlivých procesů - dalo by se říci, že je to mapa stavu paměti RAM**
- **/root** - domovský adresář superuživatele (root)
- **/sbin** - systémové privilegované spustitelné soubory, používané uživatelem root
- **/sys** - virtuální adresář (jádra 2.6.x)
- **/tmp** - adresář pro odkládací a pomocné soubory

- **/usr** - další stromová struktura, obsahuje velké množství informací, jako knihovny, zdrojové kódy, spustitelné soubory, konfigurační soubory a další. Nejdůležitější z nich jsou:
 - **/usr/bin** - jako v **/bin**, pro systém méně důležité aplikace
 - **/usr/X11R6** - kompletní soubory X Window systému (zkráceně X, nikdy ne X Windows!)
 - **/usr/include** - hlavičkové soubory jádra a knihoven (v jazyce C)
 - **/usr/lib** - další všeobecné a aplikační knihovny
 - **/usr/local** - bod pro instalace aplikací, obsahuje mj. podadresáře **/bin**, **/etc**, **/lib**, **/sbin**,
 - **/usr/sbin** - obdoba **/sbin**, méně důležité
 - **/usr/share** - soubory použitelné více aplikacemi/subsystemy (man a info stránky, dokumentace k prg. balíkům a jejich programům, pozadí a spořiče obrazovky, fonty, definice národních abeced a zvyklostí, měrných jednotek a časových zón, ikony, zvuky apod.
 - **/usr/src** - zdrojové soubory jádra
- **/var** - soubory, jejichž obsah se během chodu systému většinou mění; nejdůležitější:
 - **/var/named** - databáze DNS démona named - popisy spravovaných domén
 - **/var/lock** - "zámky" subsystémů a programů
 - **/var/log** - systémové a aplikační logy
 - **/var/spool** - různé systémové fronty
 - **/var/www** - data WWW serverů - adresáře vlastních HTML dokumentů, ikon, CGI scriptů, manuálových stránek.

Ze seznamu těchto adresářů Nás nejvíce zajímají pro další použití a příklady adresáře **/proc** a **/dev**.

□ Přístup k systémovým proměnným v prostředí OS Linux

V OS Linux existují stejně jako v OS Windows systémové proměnné, které v sobě uchovávají například typ procesoru, cestu k domovskému adresáři uživatele,... K těmto systémovým proměnným lze v jazyce C přistupovat pomocí funkce **getenv(„NAZEV_PROMENNE“)**;

Každý program běžící v Linuxu má přiřazeno jisté prostředí (enviroment). Prostředí je množina dvojic [systémová proměnná, hodnota]. Systémové proměnné jsou znakové řetězce a jejich jména se píší velkými písmeny. Je vysoce pravděpodobné, že jste se už setkali s proměnnými **USER**, **HOME**, **PATH**, **DISPLAY**. Proměnné můžete vytvářet v shellu zápisem **PROMENNA=hodnota**, poté je nutné je exportovat z shellu do prostředí příkazem **export PROMENNA**. Zjistit hodnotu proměnné můžete příkazem **echo \$PROMENNA**. A konečně, chcete-li vypsat prostředí shellu, zadejte příkaz **printenv**.

Funkce **getenv** má jediný argument, a to jméno proměnné, jejíž hodnotu chcete zjistit. Pokud není systémová proměnná definována, vrátí funkce **NULL**, jinak vrátí textový řetězec, který obsahuje hodnotu proměnné. Jestliže budete chtít nastavit nebo zrušit systémovou proměnnou, použijte funkci **setenv** resp. **unsetenv**. Následuje příklad použití systémových proměnných:


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

int main()
{
    char *conf_file = getenv("CONFIG_FILE");
    if(conf_file == NULL)
        /* Promenna CONFIG_FILE nebyla definovana,
        * pouzije se defaultni konfiguracni soubor
        */
        conf_file = "/etc/config";
    ...
    ...
    ...

    return(0);
}
```

Příklad

V prvním příkladu si vytvoříme jednoduchou aplikaci vypisující na konzoli přenosovou rychlost pevného disku instalovaného v počítači, na kterém je provozován OS Linux. Platforma, na které bude testován tento program, nemusí být nezbytně osobní počítač s procesory Intel či AMD ale může se jednat také o platformu chytrého mobilního telefonu s OS android, který je spuštěn na OS Linux. Pomocí tohoto příkladu se blíže seznámíte se základy programování v jazyce v OS Linux a také poznáte jaká je hierarchie souborového systému v OS Linux, která vychází ze souborového systému OS Unix a je diametrálně odlišná od souborového systému OS Windows.

Poznámky autora k řešení příkladu: Informace o přenosové rychlosti jakéhokoliv úložného zařízení (USB Flash disk, SD karta, HDD, SSD,...) lze získat pomocí příkazu `hdparm -tT /dev/sdx`, kde „sdx“ je systémová cesta k připojenému úložnému zařízení



Příklad

V dalším příkladu si vyzkoušíme zajímavější aplikaci, kde budeme data zapisovat přímo do zvukového čipu, který je mapován do adresářové cesty **/dev/snd**. Data do zvukové karty můžeme zapsat přímo následujícím příkazem, který zadáme do příkazové řádky:

```
cat /dev/urandom | padsp tee /dev/audio > /dev/null
```

Příkaz **cat**, slouží ke spojování souborů a k zápisu do standardního výstupu na konzoli, tento výstup, kterým jsou náhodná čísla (**/dev/urandom** – sesbírána z šumu a chybovosti ovladačů různých periferních zařízení) přesměrujeme pomocí příkazu **padsp** (přesměrování **/dev/dsp**, **/dev/audio** na nový zvukový server, které používají novější verze Linux s novějším jádrem OS Linux) a operátoru „>“ do zařízení **/dev/audio**, které představuje ovladač pro zvukovou kartu a posléze do **/dev/null** aby výstup generátoru náhodných čísel nebyl zobrazován do konzole. Po spuštění tohoto

příkazu ve sluchátkách či na reproduktorech uslyšíme šum. V Našem příkladu ale nebudeme generovat náhodný šum, ale vytvoříme software v jazyce C, který bude generovat harmonickou vlnu (\sin) a tyto data posléze přesměrujeme do zařízení zvukové karty (**/dev/audio**). Zápis příkazu do konzole z jazyka C, jste si už mohli vyzkoušet v prvním příkladu. Možné řešení zadání tohoto příkladu najdete v příloze řešených příkladů na konci skript.



5.4. Vytváření knihoven v OS Linux



Čas ke studiu:

1 hodina (s příklady 4 hodiny).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat způsob a vytvořit vlastní dynamické a statické knihovny psané v jazyce C v prostředí OS Linux,
- použít vytvořené knihovny ve vlastní aplikaci.



Výklad

Při vytváření větších projektů obsahující znovuvyužitelné části kódu je vhodné tyto části uzavřít do knihoven, které jsou následně našim či jiným programem využívány. Existují celkem dva druhy knihoven a to sice dynamické (OS Win: *.dll, OS Lin: *.so) a statické (OS Win: *.lib, OS Lin: *.a). Nyní si popíšeme si způsob vytvoření obou druhů těchto knihoven v prostředí OS Linux.

▣ Statické knihovny

Statické knihovny jsou kolekcí přemístitelného kódu uloženého v jediném souboru, a proto jsou občasné nazývány archívy. Když je linkeru předána statická knihovna, bude v ní linker vyhledávat kódy funkcí, jež jsou použity v programu, který tuto knihovnu využívá. Vyhledanou funkci extrahuje z knihovny a spojí ji kompilovaným programem stejně, jako by byl přemístitelný kód specifikován v linkeru odděleně.

Pro vytvoření statické knihovny lze použít program `ar`. Statické knihovny používají příponu `*.a`. Následující příklad vytváří ze souborů `test1.o` a `test2.o` knihovnu `libtry.a`.

```
# ar cr libtry.a test1.o test2.o
```

Přehled argumentů programu `ar`:

- `c` ... vytvoří nový archiv
- `r` ... vloží soubory do archivu
- `d` ... maže moduly podle jejich jména
- `t` ... vypíše tabulku modulů, ze kterých byl archiv vytvořen
- `p` ... vypíše specifikované položky archivu na standardní výstup

Při spojování programů s knihovnami (linkování) se používá program **`ld`** (linker). Jeho použití má však jednu záludnost. Program **`ld`**, vyhledá ve statické knihovně všechny odkazy, které našel v aktuálně zpracovávaném přemístitelném kódu a které nemá

definovány. Tyto kódy vyjme ze statické knihovny a spojí je s výsledným spustitelným kódem.

!!! V příkazovém řádku je nutné statické knihovny specifikovat vždy, jako poslední argumenty jinak se překlad nezdaří!!!

□ Dynamické/sdílené knihovny

Dynamické knihovny stejně jako statické knihovny obsahují kolekci přemístitelných kódů. Ale program používající dynamickou knihovnu neobsahuje její kód. Obsahuje jen odkaz na tento kód v dynamické knihovně. Dynamické knihovny může současně využívat několik běžících programů (termín sdílená knihovna).

Dynamická knihovna není pouhou kolekcí přemístitelných kódů, z nichž si spojovací program vybírá odkazy na nedefinované symboly. V dynamické knihovně jsou přemístitelné kódy spojeny do jediného celku a program tak má v době překladu k dispozici všechny kód obsažený v knihovně.

Pro vytvoření dynamické knihovny, je zapotřebí její moduly přeložit speciálním způsobem – s použitím volby -fPIC:

```
# gcc -c -fPIC test1.c
```

Zkratka PIC znamená Position-Independent Code (kód nezávislý na pozici). Tzn., že funkce ve sdílené knihovně mohou být zavedeny do paměti na různých pozicích. Proto kód těchto funkcí musí být nezávislá na pozici v paměti.

Po překladu lze přemístitelné kódy spojit do jediného souboru, tedy do dynamické knihovny:

```
# gcc -shared -fPIC -o libtry.so test1.o test2.o
```

Volba -shared programu **ld**, vytvoří sdílenou knihovnu a nikoliv spustitelný kód. Sdílené knihovny mají příponu .so (shared object).

Spojování kódu s dynamickými knihovnami je stejné jako spojování se statickými knihovnami:

```
# gcc -o egg -egg.o -L. -ltry
```

Za předpokladu že existuje dynamická knihovna libtry.so i statická knihovna libtry.a spojovací program **ld** bude prohledávat adresáře (nejdříve ten, který je uveden za volbou -L, pak systémové), dokud nenajde adresář obsahující libtry.a nebo libtry.so, pak prohledávání ukončí. Pokud se v daném adresáři nachází pouze první z uvedených knihoven, spojovací program ji vybere. Jinak dojde ke zvolení druhé knihovny, je-li přítomna. Pro vyhnutí se nejasnostem s vybíráním verze knihovny, lze při překladu zadat volbu -static, která definuje, že bude použita jen statická knihovna, i když je v adresáři přítomna dynamická.

```
# gcc -static -o egg egg.o -L. -ltry
```

Pomocí příkazu ldd si můžete vypsat seznam všech sdílených knihoven, které jsou spojeny s daným spustitelným programem. V seznamu dynamických knihoven se

vždy zobrazí knihovna ld-linux.so, která je standardní součástí spojovacího mechanismu v systému GNU/Linux.

Zavádění dynamických knihoven

Občas je v aplikacích zapotřebí zavést jistý kód do paměti až za chodu programu. Příkladem může být třeba webový prohlížeč nebo aplikace zpracovávající zvukový signál. Do většiny takovýchto aplikací je možné zavádět moduly (pluginy), kterými je rozšířena funkcionality programu případně doplněny metody analýzy aplikace pro zpracování zvuku.

Pro zavádění dynamických knihoven do paměti je možné použít funkci `dlopen()`. Předpis této funkce je

```
void *dlopen (const char *filename, int flag);
```

Druhý parametr je přepínač, který nastavuje způsob vazby symbolů ve sdílené knihovně. Nejčastěji se používá volba `RTLD_LAZY` (další volby viz `# man 3 dlopen [5]`). Návrátový kód se používá jako odkaz na sdílenou knihovnu. Hodnotu návratového kódu lze předat funkci `dlsym()`, která vrací adresu funkce, jež byla zavedena spolu s dynamickou knihovnou. Funkční prototyp této funkce je

```
void *dlsym(void *handle, char *symbol);
```

- **Handle** je zmíněný odkaz na knihovnu,
- **Symbol** je název funkce, jejíž adresa je předána jako návratový kód. Funkce `dlsym` může být použita také k získání ukazatele na statickou proměnnou ve sdílené knihovně.

Obě funkce (`dlopen` a `dlsym`) vracejí hodnotu `NULL`, pokud se ukončí s chybou. V takovém případě lze volat funkci `dLError()` (bez parametrů), a získat tak textový popis příčiny chyby při otevírání knihovny.

```
char *dLError(void);
```

Při ukončení aplikace nebo práce s knihovnou se volá funkce `dlclose()`, která uvolní knihovnu z paměti.

```
int dlclos (void *handle);
```

Pro využití uvedených funkcí, musí být do zdrojového souboru vložen odkaz na hlavičkový soubor `dlfcn.h` a přeložit program s volbou `-ldl`. Níže je uveden příklad [5] pro aplikaci využívající dynamického načítání knihoven za běhu programu. Program vypočítá kosinus dvou funkcí `cos()` z dynamicky zavedené knihovny `libm.so`.

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so", RTLD_LAZY);
```

```

if (!handle) {
    fputs (dlerror(), stderr);
    exit(1);
}

cosine = dlsym(handle, "cos");
if ((error = dlerror()) != NULL) {
    fprintf (stderr, "%s\n", error);
    exit(1);
}

printf ("%f\n", (*cosine)(2.0));
dlclose(handle);
}

```

Příklad

V posledním příkladu si vytvoříme ukázkovou statickou knihovnu pro práci s textem. Tato knihovna bude poskytovat služby pro podbarvení textu v konzoli, změnu fontu v konzoli a změnu tloušťky fontu (bold). Dále v rámci tohoto příkladu vytvoříme testovací aplikaci, která použije tuto knihovnu a otestuje její funkčnost. Níže je uveden zdrojový kód, který lze použít pro implementaci funkcí této knihovny. Pro změny atributů fontů v konzoli je použit standardní příkaz **tput** [5].

Zdrojový soubor knihovny

```

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <string.h>
#include "shellLib.h"
#define SHELL_CMD_SIZE 200
void changeShellColor(SHELL_COLORS pColor)
{
    char outputCmd[SHELL_CMD_SIZE];
    switch(pColor)
    {
        case SHELL_COL_RED:
            strcpy(outputCmd, "tput setaf 1");
            break;
        case SHELL_COL_BLUE:
            strcpy(outputCmd, "tput setaf 4");
            break;
        case SHELL_COL_GREEN:
            strcpy(outputCmd, "tput setaf 2");
            break;
        case SHELL_COL_ORANGE:
            strcpy(outputCmd, "tput setaf 3");
            break;
        case SHELL_COL_VIOLET:
            strcpy(outputCmd, "tput setaf 5");
            break;
        case SHELL_COL_LGTLBLUE:
            strcpy(outputCmd, "tput setaf 6");
            break;
        case SHELL_COL_WHITE:
            strcpy(outputCmd, "tput setaf 7");

```

```

        break;
    case SHELL_COL_DEFAULT:
        strcpy(outputCmd, "tput init");
        break;
    default:
        strcpy(outputCmd, "tput init");
        break;
}
system(outputCmd);
}

void changeShellFont(char *pFontName)
{
    char outputCmd[SHELL_CMD_SIZE];
    strcpy(outputCmd, "sudo setfont /usr/share/consolefonts/");
    strcat(outputCmd, pFontName);
    system(outputCmd);
}

void changeShellFontWeight(bool pBold)
{
    if(pBold == true)
        system("tput bold");
    if(pBold == false)
        system("tput offbold");
}

```

Hlavičkový soubor knihovny

```

#ifndef SHELLLIB_H_
#define SHELLLIB_H_

#include <inttypes.h>

typedef enum { SHELL_COL_RED, SHELL_COL_BLUE,
               SHELL_COL_GREEN, SHELL_COL_ORANGE, SHELL_COL_VIOLET,
               SHELL_COL_LGTBLUE, SHELL_COL_WHITE, SHELL_COL_DEFAULT }
SHELL_COLORS;

typedef uint8_t bool;
#define true 1
#define false 0

void changeShellColor(SHELL_COLORS pColor);
void changeShellFont(char *pFontName);
void changeShellFontWeight(bool pBold);

#endif /* SHELLLIB_H_ */

```

Zdrojový kód testovacího příkladu s ukázkou výstupu:

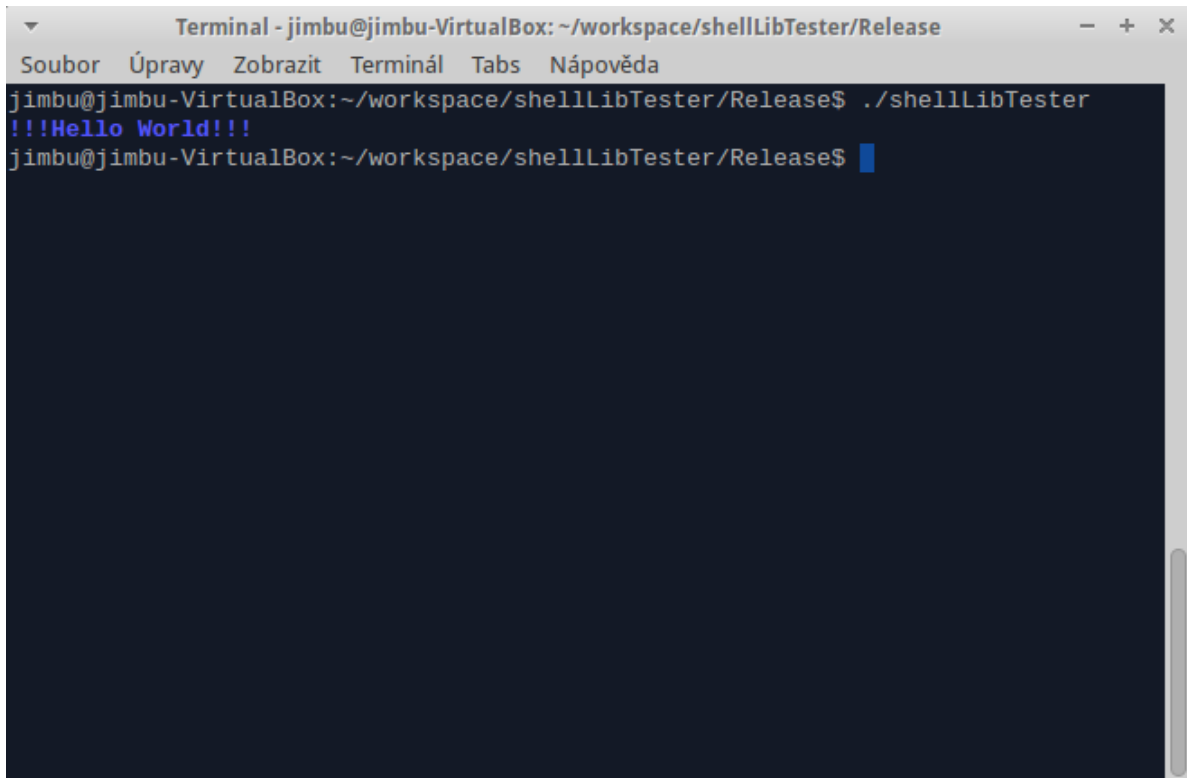
Při kompilaci této testovací aplikace (**Obr. 23**) nesmíme zapomenout zadat spojovacímu programu cestu a název ke knihovně, která byla vytvořena v předchozím příkladu následujícím způsobem

```
gcc -L"/home/xxx/cesta/ke_knihovne/" -o "shellLibTester"
./src/shellLibTester.o -lshellLib
```

Zdrojový kód testovací aplikace

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

int main(void) {
    changeShellColor(SHELL_COL_BLUE);
    changeShellFontWeight(true);
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
    changeShellColor(SHELL_COL_DEFAULT);
    return EXIT_SUCCESS;
}
```

A screenshot of a terminal window titled "Terminal - jimbu@jimbu-VirtualBox: ~/workspace/shellLibTester/Release". The window has a menu bar with "Soubor", "Úpravy", "Zobrazit", "Terminál", "Tabs", and "Nápověda". The terminal shows the command `./shellLibTester` being executed, which outputs `!!!Hello World!!!` in blue text. The prompt `jimbu@jimbu-VirtualBox:~/workspace/shellLibTester/Release$` is visible on the line below.

```
Terminal - jimbu@jimbu-VirtualBox: ~/workspace/shellLibTester/Release
Soubor  Úpravy  Zobrazit  Terminál  Tabs  Nápověda
jimbu@jimbu-VirtualBox:~/workspace/shellLibTester/Release$ ./shellLibTester
!!!Hello World!!!
jimbu@jimbu-VirtualBox:~/workspace/shellLibTester/Release$
```

Obr. 23 ukázka aplikace využívající statickou knihovnu



5.5. Procesy, paralelismus a Real Time v OS Linux



Čas ke studiu:

1 hodina (s příklady 4 hodiny).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat a pracovat s procesy a jejich správou v OS Linux.



Výklad

Každému procesu v OS Linux je přiděleno jedno vlákno, tudíž jeho zpracování probíhá pseudo-paralelně případně paralelně v případě víceprocesorového počítače (samotné plánování procesů je řízeno na úrovni OS Linux). Každému z těchto procesů může být také přidělena priorita až na úroveň `REAL_TIME`, kdy je proces prováděn v reálném čase. Konkrétní detaily plánování a rychlosti, přesnosti tohoto „reálného času“ jsou dány konkrétní konfigurací jádra operačního systému Linux. V této kapitole bude předvedeno, jakým způsobem lze běžící proces rozdělit do více subprocesů a tyto nezávisle na sobě provozovat.

□ Vytáření podřízeného procesu

Pro snadné vytvoření procesu (přesněji spuštění programu) z programu je možné použít funkci `system()`. Použití této funkce se ale pro vytváření nových procesů nedoporučuje, protože nejdříve vytvoří subproces, ve kterém běží `bash (/bin/sh)`, který pak zadaný příkaz spustí. To s sebou nese bezpečnostní rizika – spuštěný program podléhá všem vlastnostem, omezením a bezpečnostním opatřením platným pro systémový příkazový procesor.

□ Fork

Pomocí funkce `fork()` se vytvoří duplicitní kopie aktuálního procesu, která se nazývá podřízený proces. Nadřízený proces pokračuje ve své existenci a v realizaci svého kódu následujícího po volání funkce `fork()`. Podřízený proces realizuje stejný program od stejného místa. Funkce `fork()` vrací jiný návratový kód nadřízenému procesu a jiný podřízenému procesu. Návratovou hodnotou v nadřízeném procesu je identifikační číslo podřízeného procesu, návratovou hodnotou v podřízeném procesu je nula. Díky tomuto faktu je možné odlišit nadřízený proces od podřízeného při psaní kódu.

Program v příkladu vytvoří duplikát procesu. První blok v příkazu `if` se provede pouze v nadřízeném procesu, zatímco blok po klíčové slově `else` se provede pouze v podřízeném procesu.

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

```
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t child_pid;

    printf("ID procesu je %d\n", (int) getpid());

    child_pid = fork();
    if (child_pid != 0) {
        printf("Toto je rodicovsky proces, s ID %d\n", (int) getpid());
        printf("ID podrizeného procesu je %d\n", (int) child_pid);
    }
    else
        printf("Toto je podrizený proces s ID %d\n", (int) getpid());

    return(0);
}
```

□ Spouštění programů pomocí funkcí skupiny exec

Funkce exec nahradí program běžící v podržitém procesu jiným programem. Při volání funkce exec proces ukončí provádění aktuálního programu a zahájí realizaci specifikovaného programu od začátku (pokud nenastala chyba).

Ve skupině funkcí exec je několik funkcí, které se mírně liší způsobem volání a výsledkem své činnosti:

- Pokud funkce ve svém označení obsahuje písmeno p (execvp, execlp), přijímají jako argument jméno programu a hledají uvedený program v aktuálním seznamu adresářů uvedeném v proměnné PATH. Pokud jméno funkce neobsahuje písmeno p, musí se uvést jméno spouštěného programu včetně úplné cesty.
- Funkce s písmenem v (execv, execvp a execve) přijímají seznam argumentů (nulou ukončené pole ukazatelů na znakové řetězce), které mají předat spouštěnému programu.
- Funkce obsahující ve svém označení písmeno e (execve a execl) akceptují další argument, pole systémových proměnných. Tímto argumentem je nulou ukončené pole ukazatelů na znakové řetězce. Každý řetězec musí být ve tvaru „PROMENNA=hodnota“.

Seznam argumentů předávaný novému programu pracuje na stejném principu jako seznam argumentů předávaný funkci main(). Nesmíte zapomenout, že první argument musí obsahovat jméno programu (stejně jako argv[0]).

□ Spuštění programu pomocí funkcí fork() a exec

Při spouštění podprogramu uvnitř jiného programu nejdříve volejte funkci fork() a teprve pak funkce ze skupiny exec. Tímto docílíte toho, že volající kód v nadřitém procesu může pokračovat ve své realizaci, zatímco v podržitém procesu je volající program nahrazen novým programem.

Program v příkladu zobrazuje seznam souborů v hlavním adresáři pomocí příkazu `ls -l /`:

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

/* Funkce vytvori podrizeny proces a spusti v nem novy
program. PROGRAM je jmeno programu, který se ma spustit.
ARG_LIST je seznam argumentu programu. Vraci ID
vytvoreneho procesu. */

int spawn(char *program, char **arg_list)
{
    pid_t child_pid;

    /* Vytvoreni noveho procesu */
    child_pid = fork();
    if (child_pid != 0)
        /* Toto je rodicovsky proces */
        return(child_pid);
    else {
        /* Nyni se spusti PROGRAM */
        execvp(program, arg_list);
        /* Toto funkce vraci pouze pokud nastala chyba */
        fprintf(stderr, "Chyba pri volani funkce execvp()\n");
        abort();
    }
}

int main()
{
    /* Pole argumentu predavanych pri volani prikazu ls */
    char *arg_list[] = {
        "ls",
        "-l",
        "/",
        NULL
    };

    /* Volani funkce spawn */
    printf("PID spousteneho procesu je %d\n", (int) spawn("ls", arg_list));

    return(0);
}
```

□ Signály pro řízení procesů

Přehled nejpoužívanějších signálů

Jména zde uvedených signálů jsou definována pomocí maker, pokud je budete chtít používat, vkládejte do svých zdrojových kódů hlavičkový soubor `signal.h`. Skutečné definice jsou uvedeny v souboru `/usr/include/sys/sig-num.h`, který je vložen kauzulí `#include` do souboru `signal.h`. Kompletní seznam signálů naleznete v manuálových stránkách (`man 7 signal`).

- SIGHUP – Systém odešle tento signál, pokud byl odpojen terminál. Spousta programátorů pokouží signál k jiným účelům (většinou k signalizaci znovunačtení konfiguračních souborů)
- SIGINT – Tento signál bude odeslán procesu po stisknutí kombinace kláves Ctrl+C
- SIGILL – Tento signál obdrží proces, který se pokusil provést neplatnou instrukci. Také to může znamenat, že je poškozen zásobník programu.
- SIGABRT – Funkce abort() tento signál odešle volajícímu procesu.
- SIGFPE – Proces realizoval neplatnou matematickou operaci v pohyblivé řádové čárce.
- SIGKILL – Okamžité ukončení procesu. Signál nelze ošetřit obslužnou funkcí.
- SIGUSR1 – Rezervováno pro využití v aplikacích.
- SIGUSR2 – -//-
- SIGSEGV – Program se pokusil o neplatné zpřístupnění paměti. Přístup může být uplatňován na neplanou adresu virtuálního adresového prostoru, nebo může být zakázán přístupovými právy. Signál může být také způsoben dereferencí „divokého ukazatele“.
- SIGPIPE – Program se pokusil zpřístupnit nefunkční proud dat, jako je např. síťové spojení, které bylo uzavřeno.
- SIGTERM – Tento signál ukončuje proces. Jedná se o implicitní signál vysílaný příkazem kill.
- SIGCHLD – Systém tento signál odesílá při ukončení podřízeného procesu. Jeho pomocí lze asynchronně odstraňovat podřízené procesy (bude probráno v dalším dílu).
- SIGXCPU – Proces signál obdrží, pokud překročil jemu vymezený časový limit.

❑ Ukončování procesů – kill

Standardním způsobem se proces ukončuje buď zavoláním funkce exit(), anebo opustěním main(). Pokud chcete proces ukončit „zvenčí“, můžete použít příkaz kill (funkce, která toto obsluží z vašeho programu, má stejné jméno a proberu ji o pár řádků níž). Parametrem příkazu je signál, který se má procesu předat (implicitně SIGTERM), musí mu předcházet pomlčka („-“) a PID procesu, kterému chcete signál poslat. Tento příkaz se dá použít k odeslání jakéhokoliv signálu, stačí jen zadat jeho číslo nebo jméno. Příklad shoení procesu:

```
% kill -KILL pid      #nebo kill -9 pid
```

V programu můžete použít systémové volání kill(). Pokud ho budete používat, musíte includovat do vašeho zdrojového kódu hlavičkové soubory sys/types.h a signal.h. Prvním argumentem systémového volání je PID procesu, kterému chcete poslat signál, druhým pak samotný signál. Systémové volání vrátí 0, pokud vše proběhlo v pořádku. Pokud ne, vrátí -1 a nastaví proměnnou errno (více viz man 2 kill).

```
int kill(pid_t pid, int sig);
```

❑ Ukázkový příklad dvou souběžně běžících procesů

```

/* Includes */
#include <unistd.h>      /* Symbolic Constants */
#include <sys/types.h>   /* Primitive System Data Types */
#include <errno.h>       /* Errors */
#include <stdio.h>       /* Input/Output */
#include <sys/wait.h>    /* Wait for Process Termination */
#include <stdlib.h>      /* General Utilities */

int main()
{
    pid_t childpid; /* promenna pro ulozeni PID potomka */
    int retval;     /* proces potomka: návratový kód */
    int status;     /* proces rodice: exit status potomka */

    /* pouze 1 int promenna je zapotrebi protože každý proces ma vlastní
     * instanci promenne zde, 2 int promenne jsou pouzity tedy jen pro
     * nazornost aplikace
     */

    /* vytvoreni nového procesu - potomka */
    childpid = fork();

    if (childpid >= 0) /* fork byl uspesny */
    {
        if (childpid == 0) /* fork() vraci 0 procesu potomka */
        {
            printf("CHILD: I am the child process!\n");
            printf("CHILD: Here's my PID: %d\n", getpid());
            printf("CHILD: My parent's PID is: %d\n", getppid());
            printf("CHILD: The value of my copy of childpid is: %d\n",
childpid);
            printf("CHILD: Sleeping for 1 second...\n");
            sleep(1); /* sleep for 1 second */
            printf("CHILD: Enter an exit value (0 to 255): ");
            scanf(" %d", &retval);
            printf("CHILD: Goodbye!\n");
            exit(retval); /* potomek konci s uzivatelsky zadany kodem */
        }
        else /* fork() vraci novy PID rodičovskému procesu */
        {
            printf("PARENT: I am the parent process!\n");
            printf("PARENT: Here's my PID: %d\n", getpid());
            printf("PARENT: The value of my copy of childpid is %d\n",
childpid);
            printf("PARENT: I will now wait for my child to exit.\n");
            wait(&status); /* počkej na ukončení procesu potomka */
            printf("PARENT: Child's exit code is: %d\n",
WEXITSTATUS(status));
            printf("PARENT: Goodbye!\n");
            exit(0); /* rodičovský proces ukončuje svou činnost */
        }
    }
    else /* pokud fork() selže vrací -1 */
    {
        perror("fork"); /* zobrazení chybové hlášky */
        exit(0);
    }
}

```

```
}  
}
```

Příklad

V tomto příkladu si vyzkoušíme síťovou komunikaci mezi klientem a serverem pomocí vlastního programu v programovacím jazyce C. Tato aplikace bude schopna zasílat textové zprávy mezi klientem a serverem. Na tomto příkladu se naučíte používat knihovny pro práci se sítěmi a TCP/IP pakety. Na tomto příkladu lze také vidět, že Linux jako operační systém opravdu podporuje nativní jazyk z Unixového prostředí a to sice jazyk C tím, že pro tento jazyk a jazyk C++ poskytuje velké množství nativních knihoven.

Poznámka autora: Problematika socketů a jejich seskupování, což je strategie, kterou je výhodné použít pro příklad více klientů připojených na server, kterým jsou zasílána data v kopii, je velice rozsáhlá a není v těchto skriptech probírána. Doporučuji se tedy tuto problematiku prostudovat z [6]. Ukázka funkční aplikace chatu, jejíž zdrojový kód je v kapitole cvičení, s řešenými příklady.

```
Terminal - jimbu@jimbu-VirtualBox: ~/workspace/chat/Release
Soubor Úpravy Zobrazit Terminál Tabs nápověda
jimbu@jimbu-VirtualBox:~$ cd workspace/
chat/
jimbu@jimbu-VirtualBox:~/workspace/chat/Release$ ./chat
*** Spustění chat serveru (zadejte "quit" pro ukončení):
Client 0 připojen
Client 1 připojen
4Zpráva od klienta A
5Zpráva od klienta B

Terminal - jimbu@jimbu-VirtualBox: ~/workspace/chat/Release
Soubor Úpravy Zobrazit Terminál Tabs nápověda
jimbu@jimbu-VirtualBox:~$ cd workspace/chat/Release/
jimbu@jimbu-VirtualBox:~/workspace/chat/Release$ ./chat localhost
*** Spustím klienta... (zadejte "quit" pro ukončení):
Zpráva od klienta A
4 5Zpráva od klienta B

Terminal - jimbu@jimbu-VirtualBox: ~/workspace/chat/Release
Soubor Úpravy Zobrazit Terminál Tabs nápověda
jimbu@jimbu-VirtualBox:~$ cd workspace/chat/Release/
jimbu@jimbu-VirtualBox:~/workspace/chat/Release$ ./chat localhost
*** Spustím klienta... (zadejte "quit" pro ukončení):
5 4Zpráva od klienta A
Zpráva od klienta B
```

Obr. 24 ukázková aplikace chatu



Příklad

Vytvořte aplikaci, která vytvoří dva procesy pomocí funkce fork0. Druhý proces tedy potomek spustí stejný příkaz jako první příklad této kapitoly, čili příkaz pro zjištění přenosové rychlosti disku.

```
hdparm -tT /dev/sdx
```



Shrnutí pojmů

Unix, Linux, GNU, GCC



Otázky

1. Jaká je cesta k souboru obsahující informace o teplotě CPU?
2. Jaké hlavičkové soubory jsou použity pro síťovou komunikaci v OS Linux?
3. Jakým způsobem je možné zabezpečit (šifrovat) síťové spojení v příkladu č.2?



1.1.1 Další zdroje

[5] Michael Kerrisk, The Linux man-pages v 3.51, 2004. Dostupné online z <
<https://www.kernel.org/pub/linux/docs/man-pages/>>.

[6] Radim Dostál. Sokety a C/C++: Množina soketů. 2003. Dostupné online z <
<http://www.root.cz/clanky/sokety-a-c-mnozina-soketu/>>.

6. STANDARD C A SPECIÁLNÍ VARIANTY PROGRAMOVACÍHO JAZYKA C (C99, C11, Objective-C)

Tato kapitola popisuje historii jazyka C a jeho vývoj, včetně ukázek rozdílů mezi jednotlivými standardy. Na konci této kapitoly je popsána speciální varianta jazyka C a to sice Objective-C.

6.1. Stručná historie jazyka C, standard C99



Čas ke studiu:

2 hodiny.



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat, mít přehled a pracovat s aktuálními specifikacemi jazyka C se všemi jeho doplňky a novými prvky.



Výklad

Vývoj jazyka C začal v Bellových laboratořích AT&T mezi léty 1969 a 1973. Pojmenování „C“ bylo autory (Brian W. Kernighan, Dennis M. Ritchie), protože mnoho vlastností bylo přebráno ze staršího jazyka zvaného „B“, jehož název byl zase odvozen od jazyka BCPL.

□ K&R C (1978)

V roce 1978, Ritchie a Brian Kernighan vydali první vydání knihy The C Programming Language. Tato kniha, mezi programátory C známá jako „K&R“, sloužila po mnoho let jako neformální specifikace jazyka. Tato verze jazyka C bývá označována jako „K&R C“. Byl specifikován (datový typ struct, datový typ long int, datový typ unsigned int, Operátor += byl změněn na +=, K&R C je považován za základní normu, kterou musejí obsahovat všechny překladače jazyka C. Tato základní specifikace dále definovala: datový typ void * a funkce vracející void, funkce vracející typ struct nebo union, položky ve struct se ukládají do odděleného jmenného prostoru pro každý struct, modifikátor const, standardní knihovnu zahrnující většinu funkcí implementovaných různými dodavateli, výčtový typ enumeration, datový typ float

□ ANSI C a ISO C (1989 resp. 1990)

V roce 1983 American National Standards Institute (ANSI) sestavila komisi X3J11, která vytvořila standardní specifikaci C. Po dlouhém a pracném procesu byl standard dokončen v roce 1989 a schválen jako **ANSI X3.159-1989 „Programming**

Language C“. Tato verze jazyka je často stále označována jako „**ANSI C**“. V roce 1990 byl standard ANSI C (s drobnými změnami) adoptován institucí International Organization for Standardization (ISO) jako „**ISO 9899|ISO/IEC 9899:1990**“ známý také jako „**ISO C**“. Jedním z cílů standardizačního procesu ANSI C byla vytvořit nadmnožinu K&R C zahrnující mnoho „neoficiálních vlastností“. Navíc standardizační komise přidala několik vlastností jako funkční prototypy, které byly převzaty z C++ a dále byl také vylepšen preprocesor jazyka C.

□ C99 (2000)

Po standardizaci jazyka v roce 1989 se většina vývoje soustředila na jazyk C++. Ke konci 90. let však došlo k vydání dokumentu ISO 9899:1999 známý také jako „C99“, který byl následně v březnu 2000 přijat také jako ANSI standard. C99 doplnil předchozí ISO C o nové vlastnosti a rozšíření, které byly již většinou implementovány v překladačích jako doplňky. Tato rozšíření a vylepšení jsou:

- Inline funkce
- Proměnné mohou být deklarovány kdekoli (jako v C++), v C89 mohly být deklarovány pouze na začátku bloku
- Několik nových datových typů, včetně long long int (tj. 64 bitový integer), bool (logický ano/ne typ) nebo typ complex určený na reprezentaci komplexních čísel.
- Pole s nekonstantní velikostí
- Podpora pro zakomentování jednoho řádku //, tak jako v jazycích C++ nebo BCPL
- Nové knihovní funkce, hlavně ve formě náhrady za funkce náchylné na přetečení na zásobníku, např. snprintf()
- Nové hlavičkové soubory, např. stdint.h
- Variadická makra (makra C preprocesoru s proměnným počtem argumentů)
- Klíčové slovo restrict, které v deklaraci ukazatele specifikuje, že na paměť odkazovanou tímto ukazatel nepřistupuje žádný jiný ukazatel (např. int* restrict foo;), díky čemuž překladač může produkovat optimalizovanější kód; zajištění, aby tomu skutečně tak bylo, je na programátorovi (při nedodržení je chování programu nedefinované)

C99 a hlavičkový soubor stdint.h

Většina výše popsanych vlastností jazyka C dle specifikace C99 je srozumitelných až na hlavičkový soubor stdint.h. Tento hlavičkový soubor obsahuje a definuje standardní celočíselné typy, které nebyly mezi platformami jednotné. Jak lze vidět na obrázku (**Obr. 25**), který zobrazuje velikosti (v bajtech) datových typů na různých platformách s OS Linux. Jak lze vidět existují rozdíly u datových typů long a ptr. Na dalších platformách, které zde nejsou zobrazeny, existují také rozdíly u datového typu long-long. Hlavičkový soubor tedy zavádí a definuje nové datové typy, které jsou uvedené v tabulkách (**Tab. 8**, **Tab. 9**, **Tab. 10**).

Fixed width integer	signed	unsigned
8 bit	int8_t	uint8_t

16 bit	int16_t	uint16_t
32 bit	int32_t	uint32_t
64 bit	int64_t	uint64_t

Tab. 8 Celočíselné datové typy s pevnou velikostí.

Small&fixed integer types	signed	unsigned
8 bit	int_least8_t	uint_least8_t
16 bit	int_least16_t	uint_least16_t
32 bit	int_least32_t	uint_least32_t
64 bit	int_least64_t	uint_least64_t

Tab. 9 Malé celočíselné datové typy s pevnou velikostí.

Fast & fixed integer types	signed	unsigned
8 bit	int_fast8_t	uint_fast8_t
16 bit	int_fast16_t	uint_fast16_t
32 bit	int_fast32_t	uint_fast32_t
64 bit	int_fast64_t	uint_fast64_t

Tab. 10 Rychlé celočíselné datové typy s pevnou velikostí.

arch	Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386		1	2	4	4	4	8	1	2	4	8
alpha		1	2	4	8	8	8	1	2	4	8
armv4l		1	2	4	4	4	8	1	2	4	8
ia64		1	2	4	8	8	8	1	2	4	8
m68k		1	2	4	4	4	8	1	2	4	8
mips		1	2	4	4	4	8	1	2	4	8
ppc		1	2	4	4	4	8	1	2	4	8
sparc		1	2	4	4	4	8	1	2	4	8
sparc64		1	2	4	4	4	8	1	2	4	8
x86_64		1	2	4	8	8	8	1	2	4	8

Obr. 25 velikost datových typů v bajtech pro OS Linux [8].

6.2. Standard C11 (2011)



Čas ke studiu:

2 hodiny (s příklady 3 hodiny).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat nejnovější specifikaci standardu jazyka C (C11),
- použít prvky nového standardu jazyka C.



Výklad

Tento standard je poslední známý standard jazyk C, který byl schválen v roce 2011. Označení „**C11**“ je neformální označení pro specifikaci **ISO/IEC 9899:2011**. Tato specifikace nahrazuje a doplňuje předchozí specifikaci „C99“ o nové prvky a vlastnosti, které byly již většinou přítomny v aktuálních verzích kompilátorů pro jazyk C. Tato nová úprava zahrnuje hlavně detailní paměťový model pro zlepšení podpory více-vláknových aplikací. Specifikace C11 je zahrnuta v systému kompilátoru GCC od verze 4.6 a CLang verze 3.1. Oproti specifikaci C99, C11 obsahuje:

- Specifikaci zarovnání paměti (`_Alignas` specifikátor, `alignof` operátor, `aligned_alloc` funkci, `<stdalign.h>` hlavičkový soubor). Tato specifikace se svými podúrnými knihovnamy a funkcemi slouží k zarovnání dat v paměti do n – násobku velikosti datového bloku.
- `_Noreturn` funkční specifikátor, který je obsažen ve funkci, která se nevrací a nepředává řízení funkci, která jí volala. Tento parametr může v kompilátoru optimalizovat při překladu takovouto funkci.
- Type-generic výrazy s použitím klíčového slova `_Generic`. Například následující makro `cbrt(x)` je přeloženo na `cbrtl(x)`, `cbrt(x)` nebo `cbrtf(x)` v závislosti na datovém typu proměnné `x`:
 - `#define cbrt(X) _Generic((X), long double: cbrtl, \ default: cbrt, \ float: cbrtf)(X)`
- Více-vláknová podpora (`_Thread_local` storage-class specifikátor, `<threads.h>` hlavičkový soubor který poskytuje funkce k vytvoření/správy vláken, mutexy, podmínkové proměnné a další vláknovou funkcionalitu. Dále také poskytuje typ `_Atomic` a hlavičkový soubor `<stdatomic.h>` pro nepřerušitelný přístup k objektu).
- Vylepšená Unicode podpora, která vychází z *C Unicode Technical Report ISO/IEC TR 19769:2004* (typy `char16_t` a `char32_t` pro ukládání UTF-16/UTF-32 dat (řetězců), dále funkce pro konverzi do a z Unicode, jejichž prototypy

jsou obsaženy v hlavičkovém souboru <uchar.h> a odpovídající u a U string prefixy, stejně jako u8 prefix pro UTF-8 enkódované literáty).[9]

- Odstranění funkce **gets()**, která byla označená jako zastaralá (deprecated) již v předchozí specifikaci jazyka C99 (ISO/IEC 9899:1999/Cor.3:2007(E)). Tato funkce byla nahrazena novou bezpečnější alternativou **gets_s**.
- Kontroly vazeb rozhraní [9]
- Anonymní struktury a uniony, výhodné k použití v případech kdy jsou tyto prvky mezi sebou provázány, např.:

```
struct T { int tag; union { float x; int n; }; };
```

- Exklusivní režim Create-Open ("...x") pro fopen. Tento režim se chová jako O_CREAT|O_EXCL v POSIXu, což je normálně použito pro lock soubory.
- Funkce **quick_exit** jako třetí způsob ukončení programu, která je určená k minimální deinicializaci pokud ukončení pomocí funkce **exit** selže. [10]
- Makra pro deklaraci komplexních proměnných. [11]

Na co se C11 tedy převážně zaměřuje? V první řadě je to bezpečnost, se kterou jazyk C na určitých místech vždy pokulhával. Jedná se například o funkce, které manipulovali s řetězci, u kterých nedocházelo ke kontrole adres v paměti a tak často se tyto aplikace dostávali do nestabilních či kritických stavů. Dále zde byly funkce pro práci se soubory, u kterých nedocházelo ke kontrole argumentů. Tyto části aplikací byly často zdrojem chyb nebo útoků.

C11 přišlo se sadou nových bezpečnějších funkcí, které nahrazují staré méně bezpečné funkce, ale jsou stále přítomny jako zastaralé (Deprecated) i ve standardu C11.

C11 také přináší podporu pro Unicode, pro komplexní čísla, anonymní struktury a uniony a podporu generických funkcí což jsou prvky známé především z objektových jazyků jako je C++, Java, C#, Python (skriptovací multiparadigmatický programovací jazyk). Všechny tyto nové vlastnosti budou předvedeny na příkladech. Specifikace C11 ale hlavně také přináší první oficiální podporu pro více-vláknové aplikace.

□ C11 a generické funkce

Jak již bylo řečeno C11 přináší novou podporu v oblasti generických funkcí. Nyní si předvedeme, co to znamená a jak se implementují generické funkce.

Mluvíme-li o generickém kódu, jedná se o kód, který využívá parametrizované typy. Parametrizované typy jsou nahrazeny konkrétními typy v době použití kódu.

Jazyk C++ využívá pro implementaci genericky tzv. šablony (angl. templates). Jazyk C tyto šablony nemá ale využívá pro definici generických funkcí makro **_Generic**. Uvedeme si nyní příklad, který počítá třetí odmocninu z čísla. Vytvoříme pomocí generiky prototypy funkcí, které budou definovány pro vstupní parametry různých datových typů. Toto je možné samozřejmě provést i bez použití generiky ale s novým generickým datovým typem je to o mnoho snazší.

Funkce pro výpočet třetí odmocniny je „Cubic Root“ - **cbirt(X)**. Tato funkce se pomocí generiky rozšíří na **cbirtl(long double)**, **cbirtf(float)** a výchozí **cbirt(double)**, v závislosti na použitém datovém typu parametru **X**.

```
//C11
#define cbrt(X) _Generic((X), long double: cbrt1,
                        default: cbrt,
                        float: cbrtf)(X)
```

Parametr X se přeloží do specifického typu argumentu funkce. Kompilátor posléze vybere správnou variantu funkce **cbrt()**.

Dále si uvedeme příklad definující a implementující generickou funkci pro výpočet druhé mocniny čísla - **pow()**.

```
#define pow(x, y) _Generic((x), \
    long double complex: cpowl, \
\
    double complex: _Generic((y), \
    long double complex: cpowl, \
    default: cpow), \
\
    float complex: _Generic((y), \
    long double complex: cpowl, \
    double complex: cpow, \
    default: cpowf), \
\
    long double: _Generic((y), \
    long double complex: cpowl, \
    double complex: cpow, \
    float complex: cpowf, \
    default: powl), \
\
    default: _Generic((y), \
    long double complex: cpowl, \
    double complex: cpow, \
    float complex: cpowf, \
    long double: powl, \
    default: pow), \
\
    float: _Generic((y), \
    long double complex: cpowl, \
    double complex: cpow, \
    float complex: cpowf, \
    long double: powl, \
    float: powf, \
    default: pow) \
)(x, y)
```

Následuje ukázka příkladu jak lze generické datové typy využít například k tisku hodnot druhé mocniny různých datových typů

```
#define printf_dec_format(x) _Generic((x), \
    char: "%c", \
    signed char: "%hhd", \
    unsigned char: "%hhu", \
    signed short: "%hd", \
    unsigned short: "%hu", \
    signed int: "%d", \
    unsigned int: "%u", \
    long int: "%ld", \
```

```
unsigned long int: "%lu", \
long long int: "%lld", \
unsigned long long int: "%llu", \
float: "%f", \
double: "%f", \
long double: "%Lf", \
char *: "%s", \
void *: "%p")

#define print(x) printf(printf_dec_format(x), x)
#define println(x) printf(printf_dec_format(x), x), printf("\n");
We can then print values like so:
    println('a');      // prints "97" (on an ASCII system)
    println((char)'a'); // prints "a"
    println(123);      // prints "123"
    println(1.234);    // prints "1.234000"
```

□ C11 a anonymní struktury a uniony

Anonymní struktura nebo union je taková, která nemá pojmenování ani typedef pojmenování. Tyto anonymní typy jsou vhodné především pro vnořené provázání unionu do struktury, viz příklad níže. Jestliže tedy vytvoříme strukturu, která obsahuje union, můžeme do prvků unionu přistupovat přímo přes adresu struktury. Toto nebylo v dřívějších variantách jazyka C možné.

```
struct T //C++, C11
{
    int m;
    union //anonymous
    {
        char * index;
        int key;
    };
};
struct T t;
t.key=1300; //přistupuje ke členu Unionu přímo
```

□ C11 a další novinky

Podpora UTF (8,16,32). Do verze C11 byly unicode řetězce ukládány do proměnných typu char (UTF8), unsigned short (UTF16), unsigned long (UTF32). Vzhledem k tomu, že short a long jsou datové typy určené pro ukládání celočíselných hodnot byla čitelnost takovýchto zdrojových kódů často špatná a matoucí. Specifikace C11 proto představila nové datové typy a to sice **char (UTF8)**, **char16_t (UTF16)**, **char32_t (UTF32)**. C11 také poskytuje prefixy „u“ a „U“ pro unikodové řetězce a „u8“ pro UTF8. Také jsou zde konverzní funkce pro unicode řetězce, jejichž předpisy lze najít v hlavičkovém souboru <uchar.h>

Noreturn je funkce která deklaruje funkci, která se nikdy nevrací. Tento nový specifikátor má dvě hlavní funkce: potlačit varování kompilátoru na funkci, která se nevrací a zapnutí určitých optimalizací, které jsou možné pouze u funkcí, které se nikdy nevrací.

```
_Noreturn void func (); //C11, beznávratová funkce
```

□ C11 a více-vláknová podpora

Více-vláknová podpora je v jazyce C přítomna už po několik desítek let, ale nikdy tato podpora nebyla oficiální a jednalo se o doplňkové knihovny což nezajišťovalo přenositelnost takového kódu na všechny platformy. Nyní byla tato podpora implementována do samotného standardu jazyka C (C11) a tím je a bude zajištěna přenositelnost takového kódu na všechny platformy podporující C11. C11 obsahuje nový hlavičkový soubor **<threads.h>**, který obsahuje prototypy funkcí pro práci s vlákny, mutexy, podmíněnými proměnnými a `_Atomic` typ. Další nový hlavičkový soubor `<stdatomic.h>`, obsahuje nástroje pro nepřerušitelný (atomický) přístup k objektům (proměnné). Nakonec C11 také definuje nový specifikátor typu uložení proměnné v paměti ***Thread local***, který definuje vlastní paměťový prostor pro proměnnou, která je sdílena mezi vlákny.

Vzhledem k tomu, že většina dnešních nejpoužívanějších kompilátorů standard C11 plně nepodporuje, bude v zadání příkladu uveden pouze příklad s minimální počtem nových vlastností standardu C11. (<http://gcc.gnu.org/onlinedocs/gcc/Standards.html>, <http://gcc.gnu.org/wiki/C11Status>) Nejvíce prvků tohoto nového standardu dle oficiálního prohlášení podporuje kompilátor ***llvm***.

Příklad

Vytvořte aplikaci, která bude obsahovat nový uživatelský datový typ telefonního seznamu dle následující specifikace.

Název proměnné	Typ
Jméno	<code>char16_t</code>
Příjmení	<code>char16_t</code>
Telefoní číslo	<code>uint32_t</code>
union { uint32_t index, uint16_t key }	Anonymní union

Vytvořte aplikaci, který bude moci ukládat položky telefonního seznamu do souboru.

Poznámka autora: Pro ukládání do souboru lze doporučit ukládání do formátu `.csv`, který je možné posléze otevřít a zpracovávat také v libovolném tabulkovém procesoru.



6.3. Objective-C



Čas ke studiu:

2 hodiny (s příklady 4 hodiny).



Cíl:

Po prostudování tohoto odstavce budete umět

- popsat, mít přehled a pracovat s objektovou variantou programovacího jazyka C (Objective-C).



Výklad

Tento jazyk je prezentován, jako rozšíření či jednoduchá nadstavba nad jazyk C. Jedná se vsuktu o jednoduchou avšak velice silnou nadstavbu, která jistě stojí za nahlédnutí a vyzkoušení. Přináší do procedurálního jazyku, kterým jazyk C je paradigma objektového programování. Není tedy nutností se učit nový programovací jazyk při změně programovacího stylu z procedurálního na objektové.

Objective-C byl vyvinut Brad Coxem ze společnosti Stepstone počátkem osmdesátých let 20. století. Objective-C byl původně vyvinut jako hlavní programovací jazyk pro počítače NeXT s operačním systémem NeXTSTEP. Počítače NeXT už se nevyrábějí, ale myšlenka softwarového prostředí přetrvala ve standardu OpenStep. Objective C bylo původně preprocesor pro jazyk C, ale narozdíl od direktiv klasického preprocesoru které začínaly znakem mřížka ('#'), direktivy Objective C začínaly znakem zavináč ('@'). Objective C byl licencován společností NextStep Steva Jobse v době, kdy ještě nebyl ve firmě Apple. Poté, co byl NextStep odkoupen firmou Apple se začal Objective C používat i v produktech firmy Apple a stal se jedním z jejich předních vývojových nástrojů na operačních systémech MacOS, iOS a používá se k vývoji aplikací na oblíbených platformách iPhone a MacBook.

Jazyk Objective-C svou syntaxí tedy do velké míry vychází z jazyka Smalltalk a představuje oproti jazyku C následující nové datové typy:

- BOOL
- Class (v podstatě totéž co id a je s ním zaměnitelný; umožňuje lepší typovou kontrolu při překladu)
- id
- IMP
- SEL

V jazyku Objective-C lze použít všechny znalosti z jazyka C (ANSI C a C99). Jazyk Objective-C jazyk C totiž jen rozšiřuje a tak lze použít všechny známé paradigmaty jazyka C dle specifikací (ANSI C a C99).

Pro hlubší studium lze zájemcům doporučit výborný interaktivní výukový materiál dostupný ze stránek <http://tryobjectivec.codeschool.com/>.

□ Objekt

Objekt (object) je množina **proměnných** (int, float, char, ukazatel na pole, ukazatel na jiné objekty atd.), které se nazývají v objektovém programování **vlastnostmi (atributy)**, (tak jako v reálném světě i v tom programovém má každý objekt svou vlastnost). Objekt má dále definovanou sadu **funkcí** (v OOP názvosloví **metod**), jenž s těmito vlastnostmi dokáže pracovat.

Každý objekt je vytvořen na základě předpisu, který se nazývá třída (class). Třída definuje, jaká data objekt obsahuje a co s těmito daty může objekt provádět (tj. definuje metody, které s těmito daty dokáží pracovat). Podle tohoto předpisu (třídy) lze vytvořit v paměti libovolné množství objektů. Každému takto vytvořenému objektu se říká instance třídy (class instance) → **objekt = instance třídy**.

Později v této kapitole si ukážeme praktickou ukázkou této teorie na objektu osobního automobilu, kde třída představuje předpis, podle kterého se auto bude vyrábět (takže jakési výrobní technické plány a dokumentace, která bude obsahovat seznam atributů, které auto má, jako například karoserii, barvu, velikost, volant, kola, barvu kol, startérem plynový pedál... a metod, které budou definovat, co se s tímto objektem dá provést čili nastartovat, přidat plyn,...). Vytvoření objektu podle této instance bude představovat reálný automobil, který máte doma. Další automobil, který máte doma, představuje druhý objekt vytvořen podle stejné třídy (pokud se jedná o identické značky a typy vozidel).

□ Práce s objekty

Existuje několik základních pojmů, které s objektově orientovaným objektovým programováním neoddělitelně souvisí. Všechny ostatní pojmy a vlastnosti většinou souvisí spíše s konkrétním programovacím jazykem případně s konkrétní implementací daného programovacího jazyka.

Dědičnost (Inheritance): Třidu, lze takzvaně podědit, tj. lze vytvořit jinou třídu, která má tytéž vlastnosti, jako předchozí třída ze které dědí. Do této nově vzniklé třídy, která je identická s rodičovskou (rodič – potomek) třídou, můžete doplnit nové vlastnosti, které jsou specifické pouze pro potomka. Takto lze dědit opakovaně a každá nová třída má vlastnosti všech předchozích tříd, tj. stejná data a stejné metody.

Mnohotvárnost (Polymorphism): Zděděná třída může měnit činnost rodičovských metod (může je přetížit) - metoda pak má stejný předpis (vypadá pro okolní objekty stejně), ale její implementace může být jiná. U přetěžovaných metod lze také měnit počet a typy parametrů, které jsou jí předávány (metoda nadále jmenuje stejně).

Zapouzdření (Encapsulation): Termín vyjadřuje možnost skrýt některé atributy před všemi ostatními objekty. Za normálních okolností totiž může jakýkoliv objekt přistupovat k atributům jiného objektu, dokonce k atributům objektu jiné třídy. Lze ale

specifikovat, že atributy jsou z vnějšku skryté, a tudíž se nimi může pracovat jen v rámci daného objektu. Jiný objekt pak může s atributy jiného objektu pracovat pouze skrze jeho metody (ne přímo).

Delegování (Delegation): Jeden objekt může část své funkcionality delegovat na jiný objekt. To se může dít dvojím způsobem: Jeden objekt spoléhá na to, že jiný objekt implementuje určité metody, které tento objekt využívá nebo první objekt deklaruje předpisy metod, ale ne jejich implementaci a spoléhá na to, že ji definuje jeho potomek.

Toto je nejnutnější a jednoduchý základ pro stručné pochopení objektově orientovaného programování zájemce o detailnější studium této problematiky odkazují na [12].

Cocoa

Cocoa je knihovna pro jazyk Objective-C, kterou vytvořila firma Apple. Není tedy přímou součástí jazyka Objective-C a není ani v pravém slova smyslu standardní. Vzhledem k tomu že jazyk Objective-C se převážně používá pro programování výrobků firmy Apple je knihovna firmy Apple v jistém smyslu standardem.

V tomto smyslu je pak Cocoa chápána jako období standard C library pro jazyk C (libstdc).

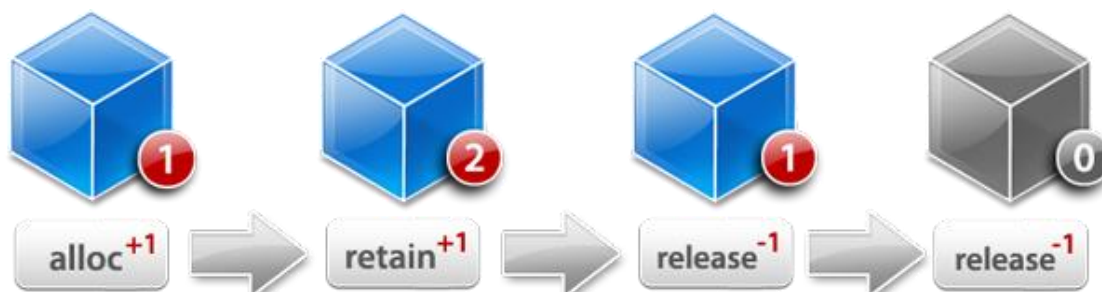
❑ Základní vlastnosti jazyka Objective-C

Zdrojové kódy jsou ukládány do souborů celkem tří typů:

- .h - hlavičkové soubory (headers) v nichž jsou deklarace a to deklarace tříd, konstant a funkcí.
- .m - zdrojové soubory (sources) v nichž jsou definice metod a funkcí
- .mm - toto jsou také zdrojové soubory, ale soubory s touto koncovkou mohou obsahovat i C++ kód.

Komentáře jsou identické s jazykem C.

Správa paměti je automatická a k uvolňování objektů dochází automaticky. Jediné na co musí být sledováno, jsou reference na objekty, které musí být uvolňovány. Životní cyklus objektu v jazyce Objective-C lze vidět na obrázku (Obr. 26).



Obr. 26 Proces životního cyklu objektu (od alokace k uvolnění) [13].

❑ Třída

Třída je rozepsána do dvou souborů. V hlavičkovém souboru (.h), stejně jako v jazyce C jsou obsaženy deklarace a ve zdrojovém souboru (.m) je uložena implementace těchto deklarovaných atributů a metod. Syntaxe a struktura předpisu a implementace třídy je uvozena klíčovým slovem **@interface** a končí klíčovým slovem **@end** respektive **@implementation** a **@end** jak lze vidět na následujících příkladech.

```
@interface JmenoTridy : JmenoPrechudce <vycet protokolu>
{
    vyctet promennych
}

vyctet metod

@end
```

```
@implementation JmenoTridy

vyctet definic metod tridy

@end
```

❑ Základní datový typ (třída) NSObject

Tento základní datový typ lze přirovnat k typu object z rodiny Microsoft .NET programovacích jazyků. Tento typ je sice součástí knihovny Cocoa a ne samotného standardu Objective-C ale je ve své podstatě základním objektem, ze kterého většina aplikací psaná v Objective-C vychází. Tento základní datový typ (třída) totiž poskytuje nejběžnější způsob jak založit, zrušit i zpracovat objekty. Navíc samotný Objective-C žádný výchozí datový typ neobsahuje.

❑ Zprávy @ funkce / metody

Syntaxe posílání zpráv (volání metod) objektům je odvozena ze syntaxe jazyka Smalltalk na rozdíl od C++, které vychází z jazyka Simula 67.

```
[příjemce zpráva]
```

Tímto způsobem je možné volat metodu na instanci, statickou metodu na třídě nebo metodu na přímém předkovi pomocí klíčového slova **super**. Každá instance disponuje proměnnou **self**, což je ukazatel na sebe sama ekvivalentní **this** z jazyka C++.

❑ Rozhraní

Rozhraní třídy je v jazyce Objective-C definováno v hlavičkovém souboru. Je dobrým programátorským zvykem (best programmer's practise) pojmenovat soubor rozhraní stejně jako je název třídy.

Ukázka rozhraní:

```
@interface jméno_třidy : předek
```

```
{
    proměnné instance
}
+ metoda třídy
+ metoda třídy
...
- metoda instance
- metoda instance
...
@end
```

□ Protokol

Protokol představuje alternativu k rozhraní interface z programovacího jazyku C# či Java. Svou implementací předepisuje (vnucuje) třídám, které jej implementují metody a atributy, které musí implementovat.

```
@protocol Protokol_1
- metoda;
@end

@interface Třída : Rodič <Protokol_1, Protokol_2, ...>
...
@end
```

Příklad

Vytvořte jednoduchou aplikaci v jazyce Objective-C, která vytvoří objekt zlomku s atributy jmenovatele a čitatele a bude mít metody pro výpočet a zobrazení reálného výsledku tohoto zlomku.



Standard jazyka C (K&R C, ANSI C, ISO C, C99, C11), Objective – C



Otázky

1. Který standard jazyka C je nejnovější?
2. Jaký je hlavní rozdíl mezi jazykem C a jazykem Objective-C?



Další zdroje

[7] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language, Second Edition. Prentice Hall, Inc., 1988. ISBN: 0-13-110362-8.

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers - Third Edition. O'Reilly 2005.

[9] Berin Babcock-McConnell. "API02-C. Functions that read or write to or from an array should take an argument to specify the source or target size"

[10] P.J. Plauger, Lawrence Crowl. WG14 N1327 Abandoning a Process. October 2007. Dostupné online z <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm>

[11] Fred J. Tydeman. September 2010. Creation of complex value. Dostupné online z <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1464.htm>

[12] Rostislav Fojtík, Vývoj objektových aplikací 1. 2002, Ostrava. Dostupné online z <http://www1.osu.cz/~fojtik/doc/VOA1.pdf>

[13] Scott Stevenson. Learn Objective C. 2008. Dostupné online z http://cocoadevcentral.com/d/learn_objectivec/.

7. VARIACE JAZYKA C PRO PROGRAMOVÁNÍ GRAFICKÝCH ČIPŮ Cg (C for graphics)

Tato kapitola popisuje jedno z nejmodernějších využití varianty jazyka C. Aplikace v masově paralelních úlohách, které jsou zpracovávány na grafických čipech, které obsahují velké množství stream multiprocessorů a jsou tak vhodné pro jednoduché ale vhodně navržené paralelně prováděné úlohy.

7.1. Jazyk Cg



Čas ke studiu:

4 hodiny.



Cíl:

Po prostudování tohoto odstavce pochopíte a budete umět

- popsat variaci jazyka C (gC – graphical C) pro výpočetní aplikace s využitím grafického jádra (GPU - Graphical Processing Unit).



Výklad

Cg (z anglického C for graphics) je vyšší programovací jazyk pro psaní shaderů vyvinutý společností NVIDIA (od roku 2002). Syntakticky je velmi podobný jazyku High Level Shader Language (HLSL) od společnosti Microsoft. V této kapitole budeme probírat možnosti využití vysokého výpočetního výkonu, který velké množství paralelních jader grafického čipu přináší. Příklady budou testovány na grafickém procesoru firmy NVidia s podporou Cuda (Compute Unified Device Architecture). A proč se tedy vůbec zabýváme touto variantou programovacího jazyka? Tato varianta totiž přináší možnost nahlédnout do budoucnosti, kde se již masivní paralelismus pomalu objevuje. Problém spočívá ve skutečnosti, že možnosti zmenšování a taktování dnešních moderních procesorových čipů dosahuje svých hranic a tak jedním z možných řešení je mimo (kvantové stroje) také cesta masivní paralelizace výpočetních algoritmů. Dnešní moderní masově dostupné procesory obsahují maximálně 8 jader, což oproti grafickým procesorům nedovoluje až tak masivní paralelismus. Oproti tomu grafické jádra jsou konstrukčně řešena jinak a nabízejí až několik stovek výpočetních jader (streaming multiprocessorů), jejichž instrukční sada není sice až tak bohatá ale pro většinu základních matematických operací je dostačující. Rozdíl v architekturách CPU a GPU lze vidět na obrázku Obr. 27.



Obr. 27 Rozdíl architektúr CPU a GPU. [15]

Historie grafických čipů a poskytovaného výpočetního výkonu

Níže uvedená tabulka zobrazuje historii a výpočetní výkony grafických čipů

Generace	Rok výroby	Název	Process výroby	Počet tranzistorů	Antialiasing Fill Rate	Polygon Rate
First	Late 1998	RIVA TNT	0.25 m	7 M	50 M	6 M
First	Early 1999	RIVA TNT2	0.22 m	9 M	75 M	9 M
Second	Late 1999	GeForce 256	0.22 m	23 M	120 M	15 M
Second	Early 2000	GeForce2	0.18 m	25 M	200 M	25 M
Third	Early 2001	GeForce3	0.15 m	57 M	800 M	30 M
Third	Early 2002	GeForce4 Ti	0.15 m	63 M	1200 M	60 M
Fourth	Early 2003	GeForce FX	0.13 m	125 M	2000 M	200 M

Tab. 11 Přehled generací a výpočetních výkonů

Syntaxe jazyka Cg je odvozena od jazyka C. Mezi důležitá rozšíření jazyka patří vektorové operace, operátor swizzle, další datové typy (half, fixed, vektorové a maticové typy, sampler* typy pro textury).

□ Ukázka zdrojového kódu Cg

```
// vstupní vrchol
struct VertIn {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
};

// výstupní vrchol
struct VertOut {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
```

```
};

// vstup vertex shaderu
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos      = mul(modelViewProj, IN.pos); // spočítej výstupní
souřadnice
    OUT.color     = IN.color; // zkopíruj vstupní barvu do výstupní
    OUT.color.z   = 1.0f; // modrá složka barvy = 1.0f
    return OUT;
}
```

Jazyk gC vznikl díky velkému technickému pokroku ve vývoji grafických čipů. Programování v oblasti 3D grafiky se stávalo čím dál složitější uměrně s technickým pokrokem. Pro zjednodušení programování byly do grafických karet implementovány programovatelné vykreslovací pipelines, které používaly vertexové a pixelové shadery. Shadery jsou počítačové programy sloužící k řízení jednotlivých částí programovatelného grafického řetězce. Logicky tedy vyplývá, že vertex shadery jsou programy prováděné nad Vertexy (vertex = bod v prostoru). Zpočátku byly tyto shadery programovány v nízkourovňovém jazyce assembleru, který sice dovolil programátorovi kompletní kontrolu nad kódem a flexibilitu ale stejně jako u osobních počítačů došel i tento assembler mezím čitelnosti, jedoduchosti použití a přenositelnosti. Z těchto důvodů byl vytvořen vyšší programovací jazyk gC.

Mnohé z algoritmů pro výpočty různých matematických operací je daleko rychlejších než na procesoru osobního počítače například algoritmus pro výpočet rychlé Fourierovy transformace (FFT), který je implementován v knihovně cuFFT^[14] provádí optimalizovaný výpočet rychlé furierovy transformace, který je až 10x rychlejší než na nejnovějších procesorech. Toto rychlostní navýšení není ale díky závratným rychlostem grafického jádra, které je často pomalejší než procesorové jádro osobního počítače ale díky mnohonásobnému paralelismu. Je zřejmé, že pro takovýto čip je ona vyšší rychlost a efektivita výpočtu vykoupena komplikovanějším návrhem algoritmů a aplikací pro takovéto výpočty.

□ Datové typy Cg

Cg má šest základních datových typů, některé z nich jsou již známé z programovacího jazyka C, jiné jsou doplněny výhradně pro použití s grafickými čipy:

- float - 32bit floating point,
- half - 16bit floating point,
- int - 32bit integer,
- fixed - 12bit fixed point number,
- bool – boolean,
- sampler* - objekt s textúrou,
- vector, matrix – typy pro vektory a matice, které jsou založeny na základních datových typech float3 a float4x4,
- struktury a pole – tyto datové typy fungují obdobně jako v jazyce C.

□ Operátory v Cg

Cg podporuje širokou škálu matematických operátorů, včetně aritmetických operátorů známých z jazyka C. Dále podporuje operátory pro práci s maticemi a běžně používané logické operátory.

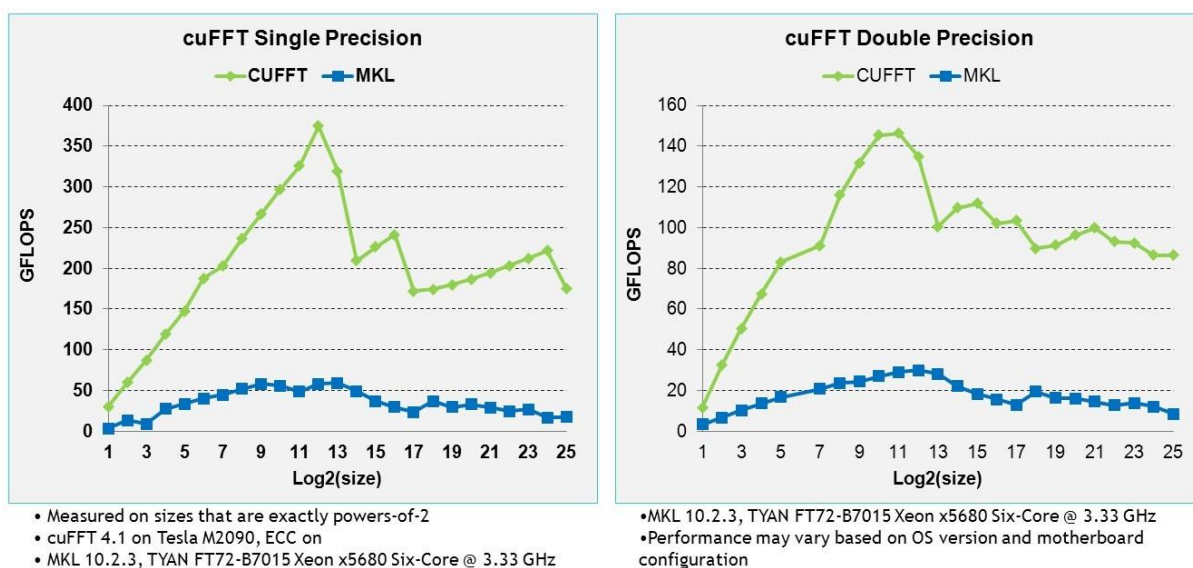
□ Funkce a řídicí struktury

Cg sdílí řídicí struktury s jazykem C (if/else, while, for) a má podobnou syntaxi definice funkcí.

□ Standardní Cg knihovna

Tak jako v jazyce C, Cg standardní knihovna poskytuje sadu funkcí typických pro aplikaci s výpočty na grafických jádrech. Některé tyto funkce jsou ekvivalentní k funkcím v jazyce C jen jejich implementace je jiná (sin, cos, abs). Ostatní funkce jsou specifické pouze pro grafické čipy. Výrobci grafických čipů většinou také nabízí vlastní knihovny, které jsou optimalizovány pro jejich produkty. Například společnost NVidia nabízí knihovnu cuFFT^[14], která provádí optimalizovaný výpočet rychlé furierovy transformace, který je až 10x rychlejší než na nejnovějších CPU procesorech (Obr. 28). Toto rychlostní navýšení není ale díky závratným rychlostem grafického jádra, které je často pomalejší než procesorové jádro osobního počítače ale díky mnohonásobnému paralelismu. Je zřejmé, že pro takovýto čip je ona vyšší rychlost a efektivita výpočtu vykoupěna komplikovanějším návrhem algoritmů a aplikací pro takovéto výpočty. Další často používané knihovny, které jsou poskytovány:

- NVIDIA CUDA Math Library – obsahuje standardní matematické algoritmy (trigonometrické, exponenciální funkce, besselovy funkce, podporu pro všechny matematické funkce definované standardem C99, statistické algoritmy,...)
- GPU AI – Path Finding – obsahuje algoritmy pro vyhledávání objektů a optimálních tras
- cuBLAS (CUDA Basic Linear Algebra Subroutines) – obsahuje základní algoritmy a funkce pro lineární algebru, které jsou 6-17x rychlejší než tato knihovna pro klasické procesory.



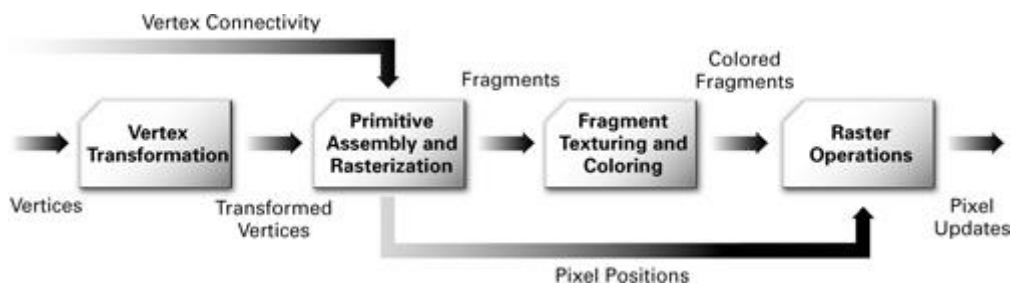
Obr. 28 Grafy srovnávající rychlost výpočtu rychlé fourioerovy transformace mezi CPU a GPU. [14]

□ Cg runtime

Cg programy jsou z velké části vertexové a pixelové shadery doplněné o podporující procedury nunté pro výpočet transformací. Cg může být použito s dvěma nejčastějšími grafickými API a to sice: DirectX a OpenGL. Každá z těchto API má vlastní sadu Cg funkcí, které s těmito API komunikují.

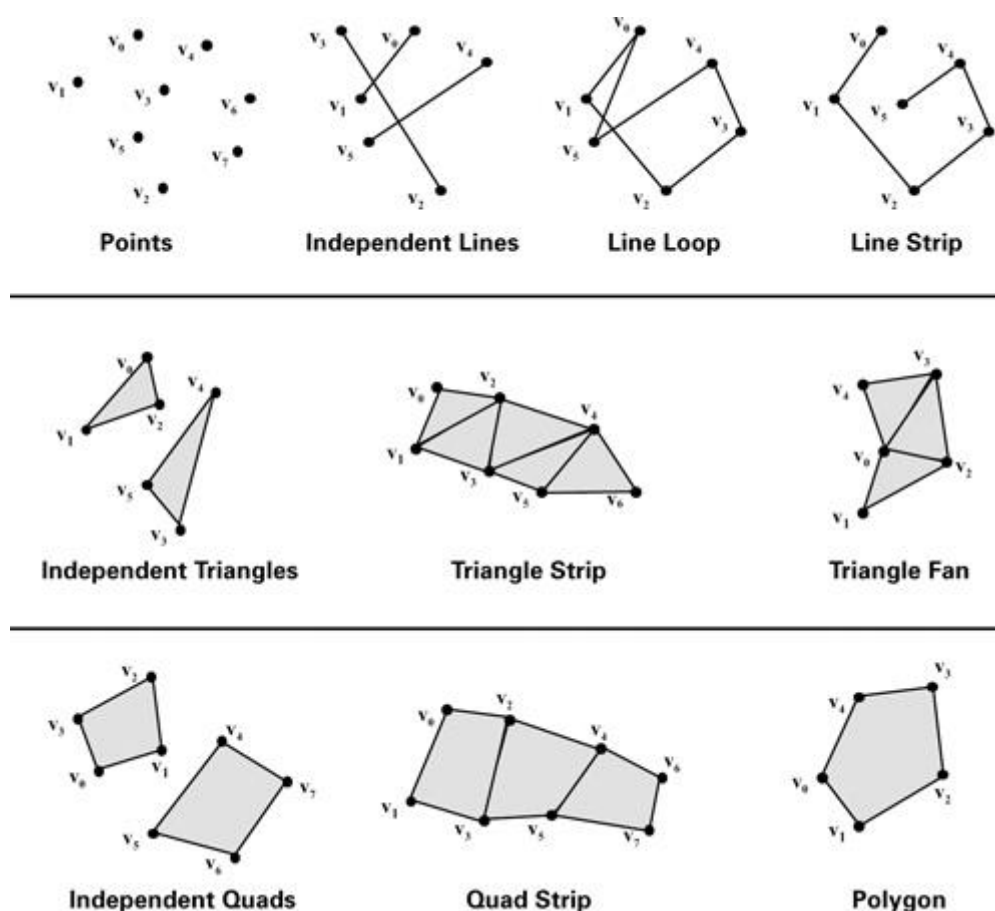
□ Hardware pipeline grafického čipu a její programovatelné části

Hardwarová pipeline grafického čipu představuje sekvenci výpočetních úrovní, která je prováděna ve fixním a paralelním uspořádání. Každá z těchto úrovní je zobrazena na obrázku (Obr. 29).



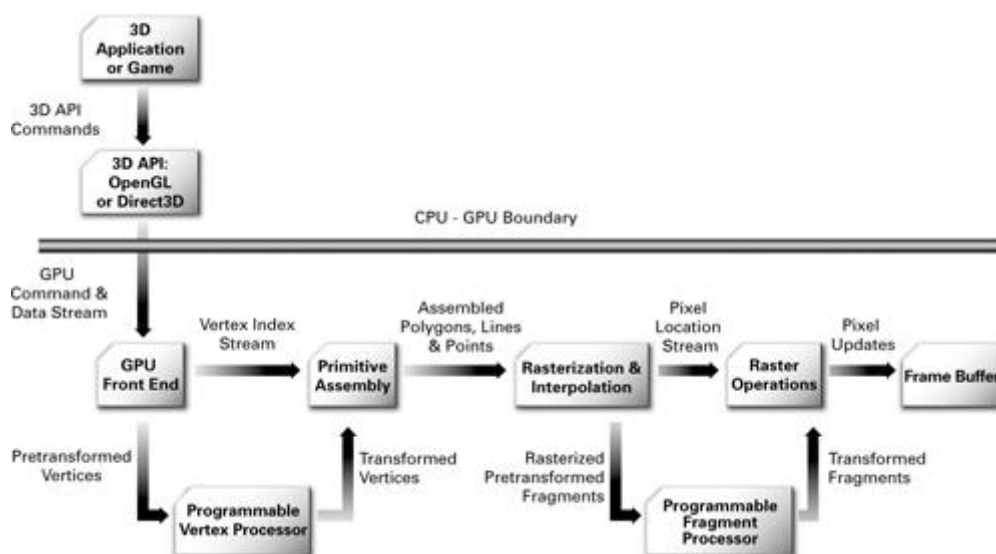
Obr. 29 Hardwarová pipeline grafického čipu. [15]

Program psaný v Cg pošle této hardwarové pipeline sekvenci vertexů spojených do geometrických primitiv (Obr. 30) - (polygony, čáry a body), která nad nimi provede algoritmus shaderů definovaný v jazyce Cg.



Obr. 30 Grafické primitiva. [15]

Programovatelnými jednotkami, na kterých lze spouštět program psaný v Cg patří programovatelný Vertex procesor a programovatelný Fragment procesor, viz (Obr. 31).



Obr. 31 Programovatelná grafická pipeline. [15]

Tyto dva procesory představují GPU část kódu z Cg aplikace.

7.2. Cuda (Compute Unified Device Architecture)



Čas ke studiu:

3 hodiny.



Cíl:

Po prostudování tohoto odstavce pochopíte a budete umět

- popsat technologii CUDA a obecnou architekturu GPU (Graphical Processing Unit) procesorů.



Výklad

Cuda je hardwarová a softwarová architektura, která umožňuje na GPU spouštět programy napsané v jazycích C/C++, FORTRAN nebo programy postavené na technologiích OpenCL, DirectCompute a jiných. Tato architektura je dostupná pouze na grafických akcelerátorech společnosti NVIDIA, která ji vyvinula. Konkurenční technologie společnosti AMD se nazývá ATI Stream.

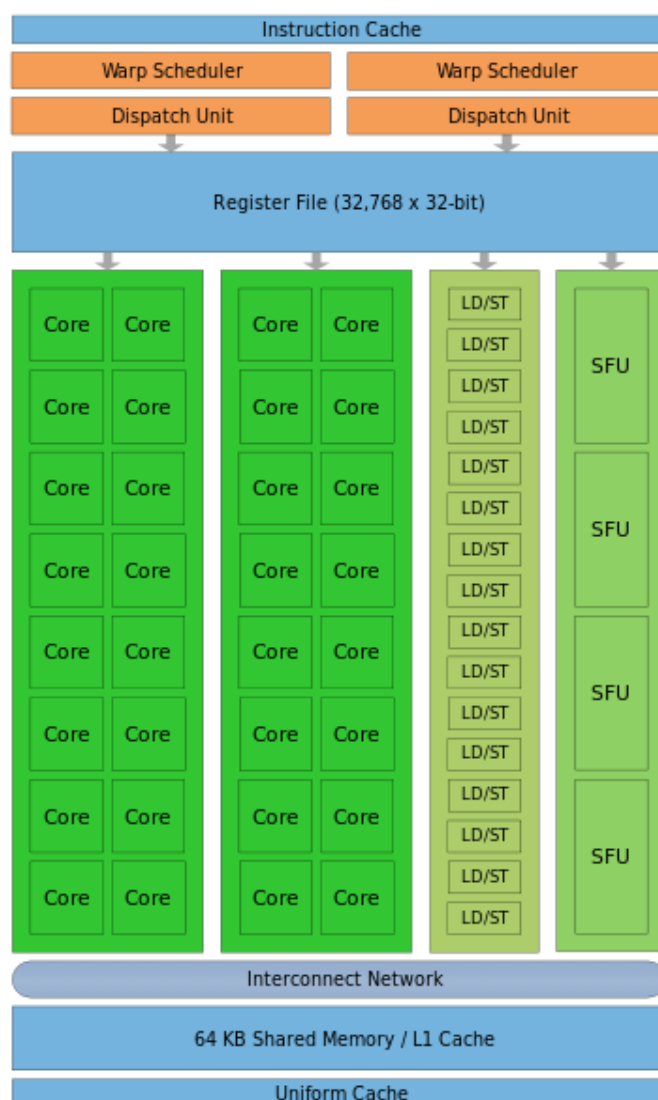
Technologii představila společnost NVIDIA v roce 2006. V roce 2007 byla uvolněno SDK ve verzi 1.0 pro karty NVIDIA Tesla založené na architektuře G80. Ještě v prosinci téhož roku vyšla verze CUDA SDK 1.1, která přidala podporu pro karty série GeForce verze 8

□ Architektura GPU

Drtivou většinu plochy čipu grafického akcelérátoru zabírá velké množství relativně jednoduchých skalárních procesorů (na rozdíl od architektury konkurenční firmy AMD, jejíž multiprocesory jsou tvořeny VLIW jednotkami), které jsou organizovány do větších celků zvaných streaming multiprocesory. Vzhledem k tomu, že se jedná o SIMT architekturu, řízení jednotek a plánování instrukcí je jednoduché a spolu s velmi malou vyrovnávací pamětí zabírá malé procento plochy GPU čipu. To má bohužel za následek omezené predikce skoků a časté zdržení výpadky cache (některé typy pamětí dokonce nejsou opatřeny cache). Poslední významnou částí, která je, rozměrově velice podobná CPU je RAM řadič, viz obrázek Obr. 27.

□ Struktura multiprocesoru

Obecně se multiprocesor skládá z několika (dnes až z 32) stream procesorů, pole registrů, sdílené paměti, několika load/store jednotek a Special Function Unit - jednotky pro výpočet složitějších funkcí jako sin, cos, ln.



Obr. 32 Schéma architektury streaming multiprocesoru typu Fermi. [15]

□ Programovací model CUDA a Cg aplikací

Tak jako v jazyce C je používán procedurální programovací model či v jazyce Objective-C je používán objektový programovací model má i jazyk Cg svůj vlastní programovací model (způsob programování a uvažování), který je odlišný.

CUDA aplikace je složena z částí, které běží buď na hostu (CPU) nebo na CUDA zařízení (GPU). Části aplikace běžící na zařízení jsou spouštěny hostem zavoláním kernelu, což je funkce, která je prováděna každým spuštěným vláknem (thread).

Blok (thread block)

Vlákna jsou organizována do 1D, 2D nebo 3D bloků, kde vlákna ve stejném bloku mohou sdílet data a lze synchronizovat jejich běh. Počet vláken na jeden blok je závislý na výpočetních možnostech zařízení. Každé vlákno je v rámci bloku identifikováno unikátním indexem přístupným ve spuštěném kernelu přes zabudovanou proměnnou ***threadIdx***, viz Obr. 33.

Mřížka (grid)

Bloky jsou organizovány do 1D, 2D nebo 3D mřížky. Blok lze v rámci mřížky identifikovat unikátním indexem přístupným ve spuštěném kernelu přes zabudovanou proměnou **blockIdx**. Každý blok vláken musí být schopen pracovat nezávisle na ostatních, aby byla umožněna škálovatelnost systému (na GPU s více jádry půjde spustit více bloků paralelně, oproti GPU s méně jádry kde bloky běží v sérii), viz Obr. 33.

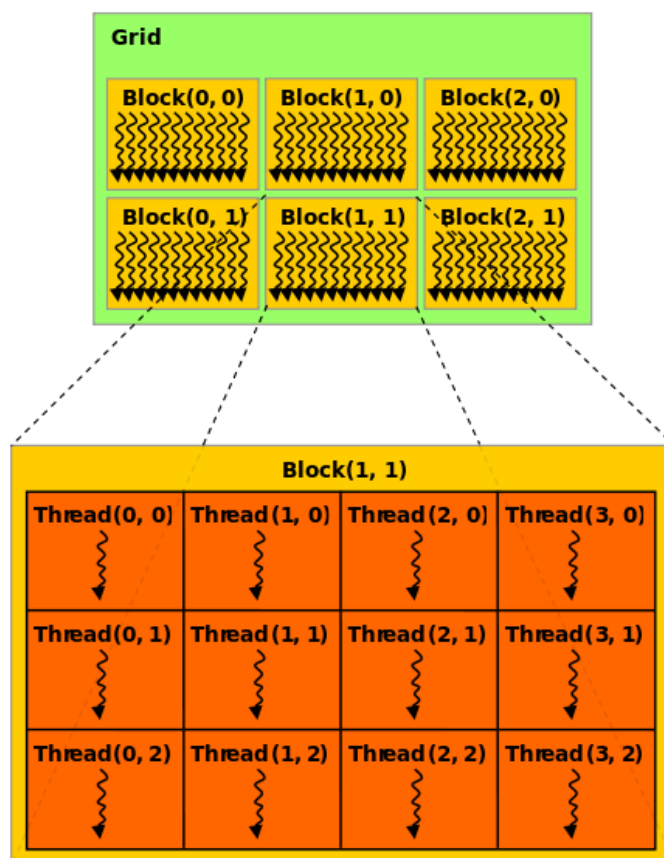
Warp

Balík vláken zpracovávaných v jednom okamžiku se nazývá warp. Jeho velikost je závislá na počtu výpočetních jednotek.

Počet a organizace spuštěných vláken v jednom bloku a počet a organizace bloků v mřížce se určuje při volání kernelu.

Typický průběh výpočtu v grafickém procesoru je:

1. Vyhrazení paměti na GPU
2. Přesun dat z hlavní paměti RAM do paměti grafického akcelérátoru
3. Spuštění výpočtu na grafické kartě
4. Přesun výsledků z paměti grafické karty do hlavní RAM paměti



Obr. 33 Programový model CUDA Cg aplikace – uspořádání vláken. [15]

Ukázka kódu v CUDA C^[15]

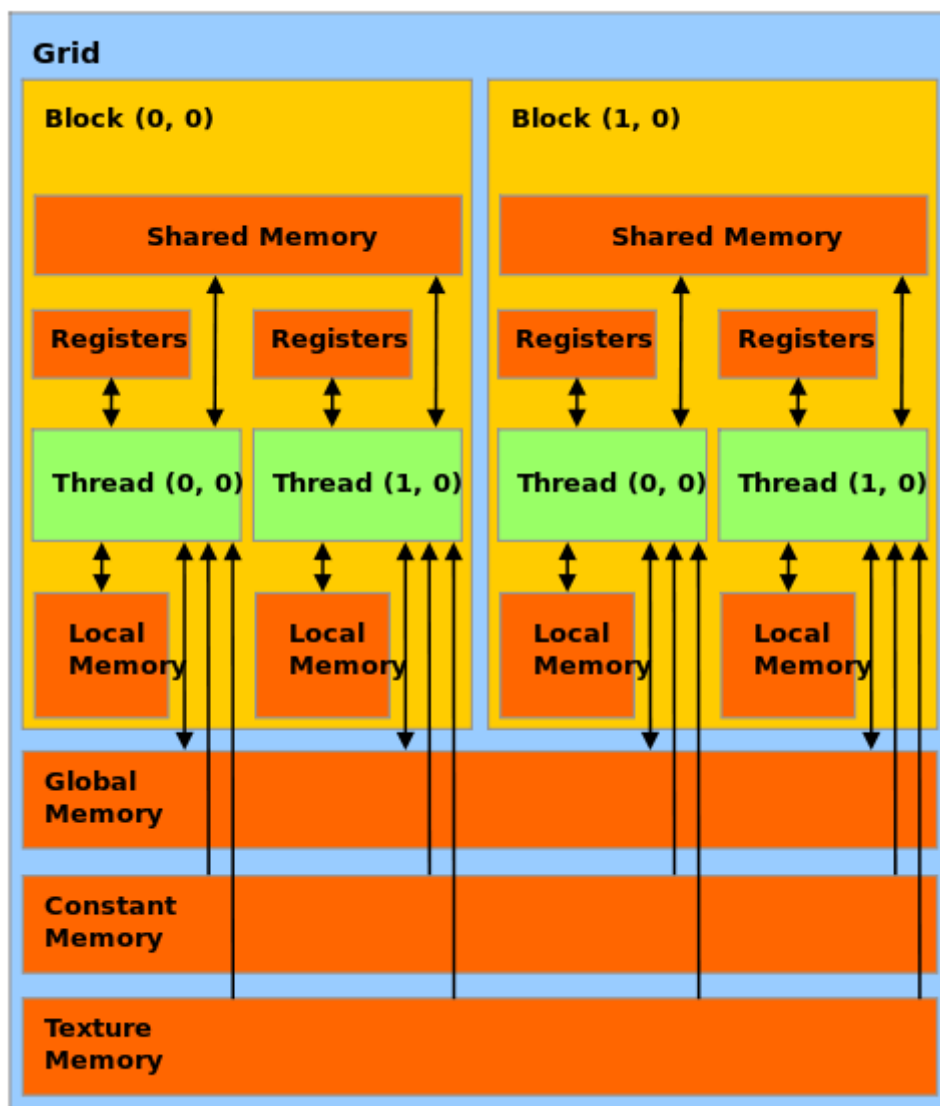
```
// Kód pro GPU
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
// Kód pro CPU
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Alokace vstupních vektorů h_A and h_B v hlavní paměti
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Inicializace vstupních vektorů
    ...
    // Alokace paměti na zařízení
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Přesun vektorů z hlavní paměti do paměti zařízení
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Volání kernelu
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C, N);
    // Přesun výsledků z paměti zařízení do hlavní paměti
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Uvolnění paměti na zařízení
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Uvolnění hlavní paměti
    ...
}
```

□ Paměťový model

Grafická karta obsahuje celkem 6 typů pamětí, ke kterým je při vývoji aplikace přístup, viz (Obr. 34).

- Pole registrů - je umístěno na jednotlivých stream multiprocesech a jejich rozdělení mezi jednotlivé stream procesory plánuje překladač. Každé vlákno může přistupovat pouze ke svým registrům.
- Lokální paměť - je užívána v případě, že dojde k vyčerpání registrů. Tato paměť je také přístupná pouze jednomu vláknu, je ale fyzicky umístěna v globální paměti akcelérátoru a tak je přístup k ní paradoxně pomalejší než např. ke sdílené paměti.

- Sdílená paměť - je jedinou pamětí kromě registrů, která je umístěna přímo na čipu streaming multiprocesoru. Mohou k ní přistupovat všechna vlákna v daném bloku. Ke sdílené paměti se přistupuje přes brány zvané banky. Každý bank může zpřístupnit pouze jednu adresu v jednom taktu a v případě, že více streaming procesorů požaduje přístup přes stejný bank, dochází k paměťovým konfliktům, které se řeší prostým čekáním. Speciálním případem je situace, kdy všechna vlákna čtou ze stejné adresy a bank hodnotu zpřístupní v jednom taktu všem streaming procesorům pomocí broadcastu.
- Globální paměť - je sdílená mezi všemi streaming multiprocesory a není ukládána do cache paměti.
- Paměť konstant - je paměť pouze pro čtení, stejně jako globální paměť je sdílená s tím rozdílem, že je pro ni na čipu multiprocesoru vyhrazena L1 cache. Podobně jako sdílená paměť umožňuje rozesílání výsledku broadcastem.
- Paměť textur - je také sdílená mezi SMP, určena pro čtení a disponuje cache pamětí. Je optimalizována pro 2D prostorovou lokalitu, takže vlákna ve stejném warpu, které čtou z blízkých texturovacích souřadnic, dosahují nejlepšího výkonu.



Obr. 34 Paměťový model grafických karet s podporou CUDA. [15]

7.3. Psaní aplikací pro CUDA Cg



Čas ke studiu:

3 hodiny (s příklady 10 hodin).



Cíl:

Po prostudování tohoto odstavce pochopíte a budete umět

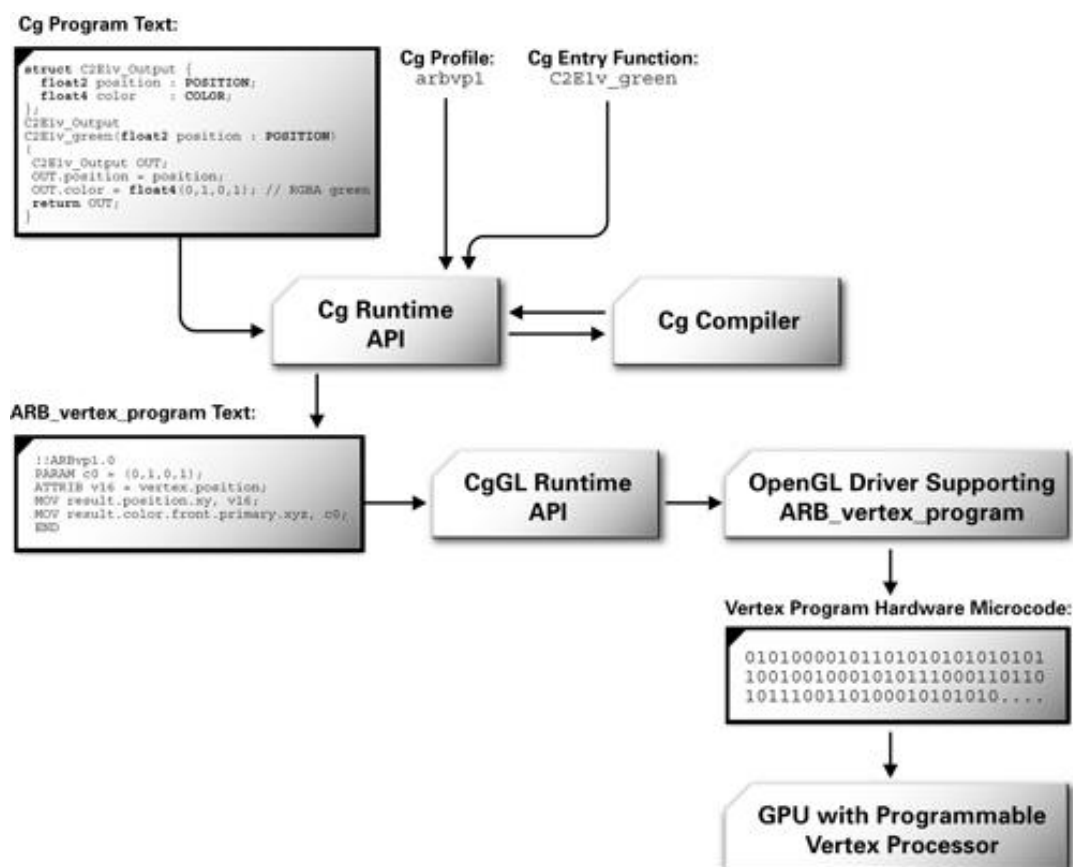
- napsat jednoduchou aplikaci v jazyce Cg pro technologii CUDA



Výklad

Zdrojový kód psaný v jazyce Cg je velice podobný jazyku C a navíc je obohacen o prvky specifické pro hardware grafického čipu. Při psaní aplikace je však třeba mít na paměti několik zásadních věcí. Kód, který je psán musí sledovat paradigma, které bylo probráno v kapitole (7.2). GPU čipy jsou specializované procesory s poměrně malou instrukční sadou a proto ne všechny kód, který je napsán v jazyce Cg lze spustit na daném grafickém čipu. Kompilátor překládá a spouští daný kód na CPU či GPU procesoru podle profilů. Tyto profily jsou jednoznačně dány kombinací grafické karty a DirectX či OpenGL API. Psaný program tedy musí být jednat sémanticky, syntakticky a algoritmicky správně napsán ale musí také splňovat omezení, které specifikuje daný hardware či API (profil). Dané paradigma také vnucuje jiný způsob myšlení při návrhu výpočetních algoritmů a jejich poměrně komplikovanou maximálně možnou paralelizaci.

Proces kompilace a nahrání Cg programu do GPU je ilustrován na obrázku (Obr. 35).



Obr. 35 Kompilace a nahrání Cg programu do GPU. [15]

□ Vertexový program – program pro vertexový mikroprocesor

Níže je uveden ukázkový program pro vertexový mikroprocesor. Tento program přidělí vertexovému mikroprocesoru výstupní pozici a zelenou barvu vertexu.

```

struct C2Elv_Output {

    float4 position : POSITION;

    float4 color    : COLOR;

};

C2Elv_Output C2Elv_green(float2 position : POSITION)
{

    C2Elv_Output OUT;

    OUT.position = float4(position, 0, 1);

    OUT.color    = float4(0, 1, 0, 1); // RGBA zelená

    return OUT;

}
    
```

- C2E1v_green — vstupní bod a název funkce
- position — parametr funkce
- OUT — místní proměnná
- float4 — vektorový datový typ
- color and position — členové struktury
- POSITION and COLOR — sémantika

Práce s vektory

```
float4 data = { 0.5, -2, 3, 3.14159 }; // Inicializace,  
                                         // jako v C  
  
int index = 3;  
float scalar;  
scalar = data[3];      // Efektivní  
scalar = data[index];  // Neefektivní nebo nepodporováno
```

Datový typ float4 není to stejné co float[4]. Jedná se o tzv. packed array (zabalené pole). Vektorové operace nad tímto typem jsou provedeny o mnoho rychleji. Jestliže GPU obdrží hodnoty tohoto typu potom vektorový součet, součin jsou provedeny v jednom instrukčním cyklu.

▣ Práce s maticemi

Navíc k vektorovým typům Cg nativně podporuje maticové typy. Níže jsou uvedeny příklady deklarací matic v Cg

```
float4x4 matrix1; // Matice 4x4 s 16 prvky  
  
half3x2 matrix2;  // Matice 3x2 s 6 prvky  
  
fixed2x4 matrix3; // Matice 2x4 s 8 prvky
```

Prvky matice lze inicializovat stejným způsobem jako prvky pole v jazycích C nebo C++.

```
float2x3 matrix4 = { 1.0, 2.0,  
                     3.0, 4.0,  
                     5.0, 6.0 };
```

▣ Sémantika

Sémantika jazyka Cg propojuje členy programu (proměnné a prvky struktury) s hardware grafické karty. Například prvky POSITION a COLOR z prvního příkladu propojují dané prvky struktury s konkrétním HW prvkem pipeline struktury, která je uvedena na obrázku Obr. 29.

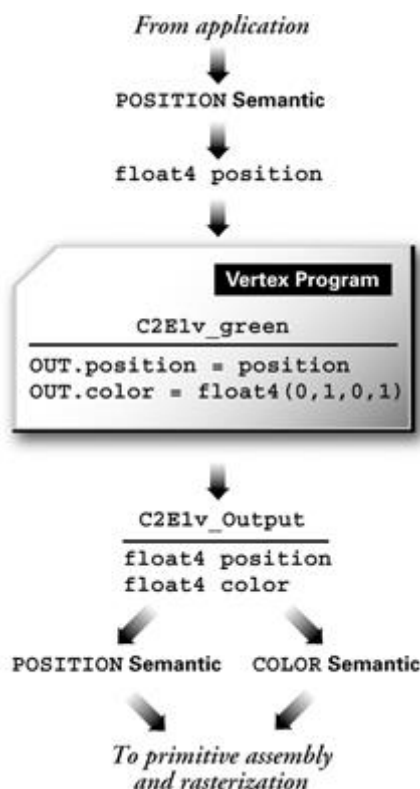
▣ Funkce v Cg

Funkce jsou Cg deklarovány stejně jako v C a C++. Funkce v Cg mohou být buď vstupní funkce, nebo interní funkce.

- **Vstupní funkce** – deklaruje vertexový nebo fragmentový program, viz (Obr. 31). Vstupní funkce je to stejné jako funkce main v jazyce C. Ve výše uvedeném příkladu je vstupní funkce **C2E1v_Output C2E1v_green(float2 position : POSITION)**. Tato funkce vrácí **C2E1v_Output**, takže vrácí jak pozici, tak barvu vertexu.
- **Interní funkce** – interní funkce jsou pomocné funkce a mohou být volány ze vstupních funkcí či jiných interních funkcí. Tyto funkce neobsahují sémantiku, a pokud ano pak je tato sémantika ignorována. Sémantiku mohou obsahovat pouze vstupní funkce.

□ Vstupní a výstupní sémantika

Vstupní a výstupní sémantika tak, jak je uvedena v ukázkové aplikaci je názorně rozdělena na obrázku Obr. 36.



Obr. 36 Vstupní a výstupní sémantika. [15]

□ Kompilace programu Cg pro Nvidia CUDA

Nyní máme všechny potřebné znalosti ke zkompileování prvního nejjednoduššího programu. Pro psaní programů v jazyce Cg pro Nvidia Cuda je zapotřebí mít nainstalováno Visual C++ 2010 Express (zdarma) a Cuda Toolkit (<https://developer.nvidia.com/cuda-toolkit>).

Ukázková aplikace, kterou budeme kompilovat má za úkol spouštět rekurzivně (opakovaně) 2 bloky obsahující 2 vlákna dokud nedosáhne limit maximálního počtu spuštěných bloků. První dva bloky budou spuštěny na CPU. Každé vlákno, které bude, ale už vykonáváno na GPU vytvoří další dva bloky s dvěma vlákny, ale tentokrát už budou spuštěny přímo na GPU.

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <helper_cuda.h>
#include <helper_string.h>

////////////////////////////////////
/////
// Promena na GPU pouzita k ulozeni jednoznačného ID bloku.
////////////////////////////////////
/////
__device__ int g_uids = 0;

////////////////////////////////////
/////
// Vytisknutí jednoduché zprávy pro probuzení bloku.
////////////////////////////////////
/////

__device__ void print_info(int depth, int thread, int uid, int parent_uid)
{
    if (threadIdx.x == 0)
    {
        if (depth == 0)
            printf("BLOCK %d launched by the host\n", uid);
        else
        {
            char buffer[32];

            for (int i = 0 ; i < depth ; ++i)
            {
                buffer[3*i+0] = '|';
                buffer[3*i+1] = ' ';
                buffer[3*i+2] = ' ';
            }

            buffer[3*depth] = '\\0';
            printf("%sBLOCK %d launched by thread %d of block %d\n",
buffer, uid, thread, parent_uid);
        }
    }

    __syncthreads();
}

////////////////////////////////////
/////
// Jádro používá CUDA dynamický paralelismus
//
// Je vytvořen unikátní identifikátor pro každý blok. Dále je vytištěna
// informace o tomto bloku. Nakonec pokud nebylo dosaženo max_depth,
// blok spustí nový blok přímo na GPU.
////////////////////////////////////
/////

__global__ void cdp_kernel(int max_depth, int depth, int thread, int
parent_uid)
{
    // Vytvoříme unikátní ID pro každý blok. Vytvoření provede vlákno 0 a
    poté toto ID sdílí s ostatními vlákny.
```

```

__shared__ int s_uid;

if (threadIdx.x == 0)
{
    s_uid = atomicAdd(&g_uids, 1);
}

__syncthreads();

// Vytiskneme ID bloku a informace o jeho rodiči.
print_info(depth, thread, s_uid, parent_uid);

// Spustíme nový blok jestliže jsme nedosáhli maximálního limitu.
if (++depth >= max_depth)
{
    return;
}

cdp_kernel<<<gridDim.x, blockDim.x>>>(max_depth, depth, threadIdx.x,
s_uid);
}

////////////////////////////////////
/////
// Hlavní funkce main.
////////////////////////////////////

int main(int argc, char **argv)
{
    printf("starting Simple Print (CUDA Dynamic Parallelism)\n");

    // Parse a few command-line arguments.
    int max_depth = 2;

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "h"))
    {
        printf("Usage: %s depth=<max_depth>\t(where max_depth is a value
between 1 and 8).\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "depth"))
    {
        max_depth = getCmdLineArgumentInt(argc, (const char **)argv,
"depth");

        if (max_depth < 1 || max_depth > 8)
        {
            printf("depth parameter has to be between 1 and 8\n");
            exit(EXIT_SUCCESS);
        }
    }

    // Najdi/nastav zařízení.
    int device_count = 0, device = -1;
    checkCudaErrors(cudaGetDeviceCount(&device_count));

```

```

for (int i = 0 ; i < device_count ; ++i)
{
    cudaDeviceProp properties;
    checkCudaErrors(cudaGetDeviceProperties(&properties, i));

    if (properties.major > 3 || (properties.major == 3 &&
properties.minor >= 5))
    {
        device = i;
        std::cout << "Running on GPU " << i << " (" << properties.name
<< ")" << std::endl;
        break;
    }

    std::cout << "GPU " << i << " (" << properties.name << ") does not
support CUDA Dynamic Parallelism" << std::endl;
}

if (device == -1)
{
    std::cerr << "cdpSimplePrint requires GPU devices with compute SM
3.5 or higher. Exiting..." << std::endl;
    exit(EXIT_SUCCESS);
}

cudaSetDevice(device);

// Informace o aplikaci.

printf("*****\n");
printf("The CPU launches 2 blocks of 2 threads each. On the device each
thread will\n");
printf("launch 2 blocks of 2 threads each. The GPU we will do that
recursively\n");
printf("until it reaches max_depth=%d\n\n", max_depth);
printf("In total 2");
int num_blocks = 2, sum = 2;

for (int i = 1 ; i < max_depth ; ++i)
{
    num_blocks *= 4;
    printf("+%d", num_blocks);
    sum += num_blocks;
}

printf("=%d blocks are launched!!! (%d from the GPU)\n", sum, sum-2);

printf("*****\n\n");

// Nastavení limitu zařízení na maximální hloubku.
cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, max_depth);

// Spuštění jádra z CPU.
printf("Launching cdp_kernel() with CUDA Dynamic Parallelism:\n\n");
cdp_kernel<<<2, 2>>>(max_depth, 0, -1);
checkCudaErrors(cudaGetLastError());

```



```
// Ukončení aplikace.  
checkCudaErrors(cudaDeviceSynchronize());  
checkCudaErrors(cudaDeviceReset());  
  
exit(EXIT_SUCCESS);  
}
```

Příklad

Prostudujte si příklad pro výpočet obrazové 2D konvouce pomocí grafického jádra, který je popsán v kapitole řešených příkladů (CVIČENÍ č. 5. – Řešené příklady (kapitola 7)).



Shrnutí pojmů

Cg (C for graphics), CUDA (Compute Unified Device Architecture), Shader, Vertex, Fragment, Pipeline



Otázky

3. Z čeho se skládá pipeline grafické karty?
4. Které části pipeline grafické karty jsou programovatelné?
5. Co to znamená a jaký je význam pojmů „vertex shader“ a „fragment shader“?



Další zdroje

[14] NVIDIA Corporation. CUFFT LIBRARY verze 5. October 2012. Dostupné online z <http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf>.

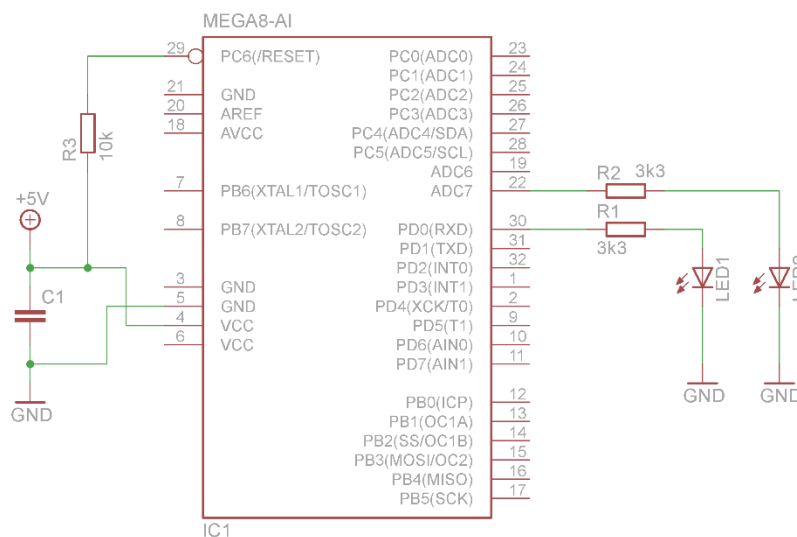
[15] NVIDIA CUDA Programming Guide Version 1.1 [online]. NVIDIA, 2007-11-29, [cit. 2011-12-01]. Dostupné online z <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf>.

CVIČENÍ č. 1. – Řešené příklady (kapitola 3).



Řešený příklad

V tomto prvním příkladu si připojte k libovolnému procesoru AVR, LED diodu, přes ochranný proudový odpor 3k Ω a vytvořte aplikaci, která touto LED diodou bude blikat. Pro algoritmus vytvářející zpoždění mezi zhasnutím a rozsvícením LED diody použijte metodu zpožďovacích smyček nebo zpoždění pomocí čítače/časovače. Obvod zapojte dle schématu na obrázku Obr. 37. Tento obvod můžete volitelně doplnit o tlačítko, kterým budete ovládat frekvenci blikání LED diod.



Obr. 37 Schéma zapojení obvodu pro příklad 1.

Řešení:

Zapojte schéma dle obrázku Obr. 37 propojte mikroprocesor s programátorem a nahrajte do ně aplikaci dle následujícího zdrojového kódu (chapter3_ledBlink.zip):

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#define DEBOUNCE_TIME 50 //Debounce time v ms - cas po, který program
nereaguje na další stisk tlačítka - filtr proti zakmitum

void setup_timer();
void setup_io();

int main(void)
{
    setup_io();
    setup_timer();
    sei(); //Povolit globalne preruseni

    while(1)
    {
        if(bit_is_clear(PINC, PC1))//Tlacitko je zmacknuto
        {
```

```

        _delay_ms(DEBOUNCE_TIME);
        if(bit_is_clear(PINC, PC1)) {
            PORTD ^= (1 << PD0); //Zmen stav LED
        }
    }
    else
    {
        PORTD &= ~(1 << PD0);
    }
}

void setup_io()
{
    DDRD |= (1 << DDD1); //Nastav pin PD1 do vystupniho rezimu
    DDRD |= (1 << DDD0);
    DDRC &= ~(1 << DDC1);
    PORTC |= (1 << PC1); //Aktivuj pull up
}

void setup_timer()
{
    TCCR1B = (1<<WGM12); //Nastavit do režimu CTC
    OCR1A = 62500; //Přerušeni každé 2s
    TIMSK1 |= (1<<OCIE1A); //Povolí přerušeni při dočítání na hodnotu
compare registru A
    TCNT1 = 0; //Vynuluj registr čítače

    TCCR1B|=(1<<CS12); //Prescaler nastaveny na 256 a spust casovac
}

ISR(TIMER1_OVF_vect)
{
    PORTD ^= (1 << PD0); //Zmen stav LED
}

ISR(TIMER1_COMPA_vect)
{
    PORTD ^= (1 << PD1); //Zmen stav LED
    TCNT1 = 0;
}

```



CD-ROM

1. Zdrojový kód aplikace blikání LED diody – chapter3_ledBlink.zip.


```

{
    uint8_t adc_ch = channel;
    ADMUX=(ADMUX&0xF0) | adc_ch; //Výběr kanálu AD převodníku
    sei (); //Globální povolení přerušení
    ADCSRA |= (1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
// Spuštění AD převodníku, povolení přerušení, nastavení předeličky na
128=62,5 kHz
}

//Obsluha přerušení od AD převodníku
ISR (ADC_vect)
{
    uint16_t adc_value;
    adc_value = ADCL;
    adc_value += (ADCH<<8); //Uložení hodnoty AD převodu

    //Jedno z možných, záměrně ne úplně nejefektivnějších řešení jak
ovládat LED diody VU metru
    if(adc_value > 125)
    {
        PORTD |= (1<<PD0);
    }
    if(adc_value > 250)
    {
        PORTD |= (1<<PD1);
    }
    if(adc_value > 375)
    {
        PORTD |= (1<<PD2); // rozsvít LED
    }
    if(adc_value > 500)
    {
        PORTD |= (1<<PD3);
    }
    if(adc_value > 625)
    {
        PORTC |= (1<<PC1);
    }
    if(adc_value > 750)
    {
        PORTC |= (1<<PC2);
    }
    if(adc_value > 875)
    {
        PORTC |= (1<<PC3);
    }
    if(adc_value > 930)
    {
        PORTC |= (1<<PC4);
        _delay_ms (100);//
    }
    if(adc_value < 925)
    {
        PORTC &= ~(1<<PC4);
    }
    if(adc_value < 875)
    {
        PORTC &= ~(1<<PC3);
    }
    if(adc_value < 750)
    {
        PORTC &= ~(1<<PC2); // rozsvít LED
    }
}

```

```

    }
    if(adc_value < 625)
    {
        PORTC &= ~(1<<PC1);
    }
    if(adc_value < 500)
    {
        PORTD &= ~(1<<PD3);
    }
    if(adc_value < 375)
    {
        PORTD &= ~(1<<PD2);
    }
    if(adc_value < 250)
    {
        PORTD &= ~(1<<PD1);
    }
    if(adc_value < 125)
    {
        PORTD &= ~(1<<PD0);
    }
    ADCSRA |= (1<<ADSC);
}

void setup_io ()
{
    DDRD |= (1 << DDD0); // Nastavení všech použitých bran
    DDRD |= (1 << DDD1); // pro LED na výstupní režim
    DDRD |= (1 << DDD2);
    DDRD |= (1 << DDD3);
    DDRC |= (1 << DDC4);
    DDRC |= (1 << DDC3);
    DDRC |= (1 << DDC2);
    DDRC |= (1 << DDC1);
    PORTD &= ~(1<<PD0);
    PORTD &= ~(1<<PD1);
    PORTD &= ~(1<<PD2);
    PORTD &= ~(1<<PD3);
    PORTC &= ~(1<<PC4);
    PORTC &= ~(1<<PC3);
    PORTC &= ~(1<<PC2);
    PORTC &= ~(1<<PC1);
}

```



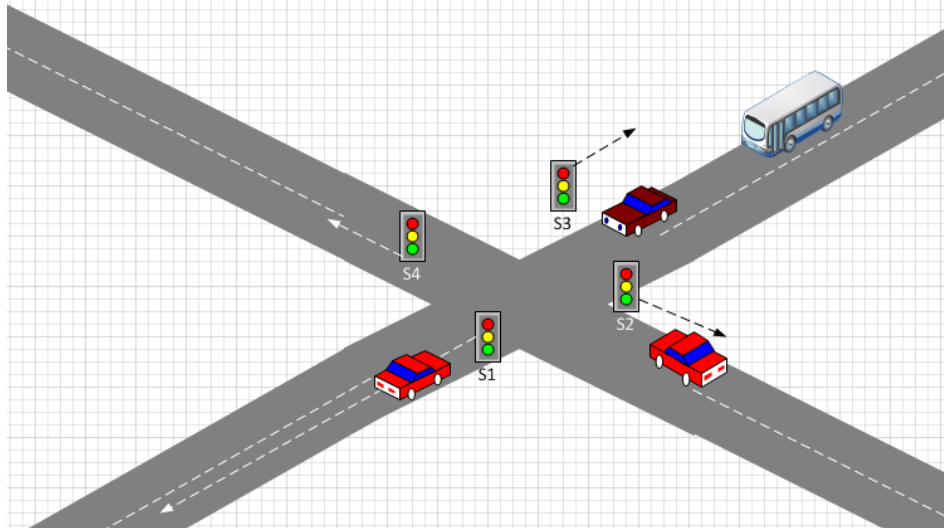
CD-ROM

1. Zdrojový kód aplikace AD převodníku - chapter3_ADC_app_megax8.zip.



Řešený příklad

V třetím a posledním příkladu této kapitoly vytvořte aplikaci, která bude simulovat blikání semaforů na křižovatce dle obrázku Obr. 20 a pravidel v tabulce Tab. 7. Časy pro zpoždění mezi semaforey řešte pomocí čítače nebo zpožďovací smyčky.



Obr. 38 Náčrt simulované křižovatky.

Tab. 12 Pravidla, podle kterých se řídí provoz

Semafor	ČAS [s]	AKCE
S1 + S3	12	JEDĚ (Zelená)
S1 + S3	2	BLIKÁNÍ (Zelená) - > STOP (Červená) && S2 + S4 BLIKÁNÍ (Červená) -> JEDĚ (Zelená)
S4 + S2	10	JEDĚ (Zelená)
S4 + S2	2	BLIKÁNÍ (Zelená) - > STOP (Červená) && S1 + S3 BLIKÁNÍ (Červená) -> JEDĚ (Zelená)

Řešení:

Nahrajte do mikroprocesoru aplikaci dle následujícího zdrojového kódu ([semaphor.zip](#)):

```
#include <avr/io.h>
#include <util/delay.h>
int main(void) {
    //nastav port PD0 na hodnotu 1 (smer)
    DDRD |= (1 << DDD0);
    DDRD |= (1 << DDD1);
    DDRD |= (1 << DDD2);
    DDRD |= (1 << DDD3);
    DDRC |= (1 << DDC5);
    DDRC |= (1 << DDC4);
    DDRC |= (1 << DDC3);
    DDRC |= (1 << DDC2);
    //nastav hodnotu brany na log 0 (hodnota), ~ znamena negace
    PORTD &= ~(1 << PD0);
    PORTD &= ~(1 << PD1);
    PORTD &= ~(1 << PD2);
    PORTD &= ~(1 << PD3);
    PORTC &= ~(1 << PC5);
    PORTC &= ~(1 << PC4);
```

```
PORTC &= ~(1 << PC3);
PORTC &= ~(1 << PC2);

for(;;) {
    // zhasni vsechny led
    PORTD |= (1 << PD0);
    PORTD |= (1 << PD1);
    PORTD |= (1 << PD2);
    PORTD |= (1 << PD3);
    PORTC |= (1 << PC2);
    PORTC |= (1 << PC3);
    PORTC |= (1 << PC4);
    PORTC |= (1 << PC5);
    _delay_ms(300);
    // semafor 1,3 jed; semafor 2,4 stuj
    PORTD &= ~(1 << PD3);
    PORTD &= ~(1 << PD0);
    PORTC &= ~(1 << PC4);
    PORTC &= ~(1 << PC3);
    _delay_ms(12000);
    // zmena barev na semaforech
    PORTD |= (1 << PD3);
    PORTD |= (1 << PD0);
    PORTC |= (1 << PC4);
    PORTC |= (1 << PC3);
    _delay_ms(300);
    PORTD &= ~(1 << PD3);
    PORTD &= ~(1 << PD0);
    PORTC &= ~(1 << PC4);
    PORTC &= ~(1 << PC3);
    _delay_ms(300);
    PORTD |= (1 << PD3);
    PORTD |= (1 << PD0);
    PORTC |= (1 << PC4);
    PORTC |= (1 << PC3);
    _delay_ms(300);
    PORTD &= ~(1 << PD3);
    PORTD &= ~(1 << PD0);
    PORTC &= ~(1 << PC4);
    PORTC &= ~(1 << PC3);
    _delay_ms(300);
    PORTD |= (1 << PD3);
    PORTD |= (1 << PD0);
    PORTC |= (1 << PC4);
    PORTC |= (1 << PC3);
    _delay_ms(300);
    PORTD &= ~(1 << PD3);
    PORTD &= ~(1 << PD0);
    PORTC &= ~(1 << PC4);
    PORTC &= ~(1 << PC3);
    _delay_ms(300);
    PORTD |= (1 << PD3);
    PORTD |= (1 << PD0);
    PORTC |= (1 << PC4);
    PORTC |= (1 << PC3);
    _delay_ms(300);
    // semafor 1,3 stuj; semafor 2,4 jed
    PORTD &= ~(1 << PD1);
    PORTD &= ~(1 << PD2);
    PORTC &= ~(1 << PC2);
    PORTC &= ~(1 << PC5);
    _delay_ms(10000);
```



```

// zmena barev na semaforech
PORTD |= (1 << PD1);
PORTD |= (1 << PD2);
PORTC |= (1 << PC2);
PORTC |= (1 << PC5);
_delay_ms(300);
PORTD &= ~(1 << PD1);
PORTD &= ~(1 << PD2);
PORTC &= ~(1 << PC2);
PORTC &= ~(1 << PC5);
_delay_ms(300);
PORTD |= (1 << PD1);
PORTD |= (1 << PD2);
PORTC |= (1 << PC2);
PORTC |= (1 << PC5);
_delay_ms(300);
PORTD &= ~(1 << PD1);
PORTD &= ~(1 << PD2);
PORTC &= ~(1 << PC2);
PORTC &= ~(1 << PC5);
_delay_ms(300);
PORTD |= (1 << PD1);
PORTD |= (1 << PD2);
PORTC |= (1 << PC2);
PORTC |= (1 << PC5);
_delay_ms(300);
PORTD &= ~(1 << PD1);
PORTD &= ~(1 << PD2);
PORTC &= ~(1 << PC2);
PORTC &= ~(1 << PC5);
_delay_ms(300);
PORTD |= (1 << PD1);
PORTD |= (1 << PD2);
PORTC |= (1 << PC2);
PORTC |= (1 << PC5);
_delay_ms(300);
}
}

```



CD-ROM

Zdrojový kód aplikace křižovatka - chapter3_semaphore.zip.

CVIČENÍ č. 2. – Řešené příklady (kapitola 4).



Řešený příklad

Vytvořte aplikaci, která pomocí pole ukazatelů na funkci provede součet výsledků goniometrických funkcí z pole ukazatelů na funkci nad zadaným parametrem. Pro začátek lze použít příklad, který částečně řeší zadání tohoto příkladu.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.



Řešený příklad

Vytvořte aplikaci, která dynamicky alokuje paměť podle velikosti zadané uživatelem a naplní je hodnotami funkce sin. Hodnoty této funkce posléze i s adresami uložených hodnot v paměti vypíše.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.



Řešený příklad

Implementujte ukázkovou aplikaci, která bude obsahovat seznam typu linked list, do kterého bude moci uživatel za běhu programu moc přidávat a zobrazovat n posledních hodnot podle zadání do konzole. Datový typ ukládaných hodnot vyberte dle vlastního uvážení

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

CVIČENÍ č. 3. – Řešené příklady (kapitola 5).



Řešený příklad

V prvním příkladu si vytvoříme jednoduchou aplikaci vypisující na konzoli přenosovou rychlost pevného disku instalovaného v počítači, na kterém je provozován OS Linux. Platforma, na které bude testován tento program, nemusí být nezbytně osobní počítač s procesory Intel či AMD ale může se jednat také o platformu chytrého mobilního telefonu s OS android, který je spuštěn na OS Linux. Pomocí tohoto příkladu se blíže seznámíte se základy programování v jazyce v OS Linux a také poznáte jaká je hierarchie souborového systému v OS Linux, která vychází ze souborového systému OS Unix a je diametrálně odlišná od souborového systému OS Windows.

Poznámky autora k řešení příkladu: Informace o přenosové rychlosti jakéhokoliv úložného zařízení (USB Flash disk, SD karta, HDD, SSD,...) lze získat pomocí příkazu `hdparm -tT /dev/sdx`, kde „sdx“ je systémová cesta k připojenému úložnému zařízení

Řešení:

Samotné řešení je poměrně jednoduché s využitím příkazu `hdparm` s parametry `-tT`. V jazyce C lze pro spuštění procesu a získání jeho výstupu z příkazové řádky s úspěchem použít funkcí `popen` a `pclose`. Řešení zadaného příkladu je uvedeno níže.

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

int main( int argc, char *argv[] )
{
    FILE *fp;
    int status;
    char path[1035];

    /* Otevreme prikaz pro cteni */
    fp = popen("/usr/bin/sudo /sbin/hdparm -tT /dev/sda", "r");
    if (fp == NULL) {
        printf("Chyba pri spusteni/vykonovavani prikazu\n" );
        exit;
    }

    /* Preceteme vystup prikazu a posleze zobrazime na displeji */
    while (fgets(path, sizeof(path)-1, fp) != NULL) {
        printf("%s", path);
    }

    /* Zavreme stream k procesu */
    pclose(fp);

    return 0;
}
```



CD-ROM

1. Zdrojový kód řešené aplikace k získání parametrů úložného zařízení – `chapter5_hdparm.zip`.



Řešený příklad

Příkaz **cat**, slouží ke spojování souborů a k zápisu do standardního výstupu na konzoli, tento výstup, kterým jsou náhodná čísla (**/dev/urandom** – sesbírána z šumu a chybovosti ovladačů různých periferních zařízení) přesměrujeme pomocí příkazu **padsp** (přesměrování **/dev/dsp**, **/dev/audio** na nový zvukový server, které používají novější verze Linux s novějším jádrem OS Linux) a operátoru „>” do zařízení **/dev/audio**, které představuje ovladač pro zvukovou kartu a posléze do **/dev/null** aby výstup generátoru náhodných čísel nebyl zobrazován do konzole. Po spuštění tohoto příkazu ve sluchátkách či na reproduktorech uslyšíme šum. V Našem příkladu ale nebudeme generovat náhodný šum, ale vytvoříme software v jazyce C, který bude generovat harmonickou vlnu (sin) a tyto data posléze přesměrujeme do zařízení zvukové karty (**/dev/audio**). Zápis příkazu do konzole z jazyka C, jste si už mohli vyzkoušet v prvním příkladu. Možné řešení zadání tohoto příkladu najdete v příloze řešených příkladů na konci skript.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

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

#define CHAR_SAMPLE_BUFFER_SIZE      4
#define CMD_BUFFER_SIZE              20000

typedef char* string;

string sound(double freq, double amp, double time);

int main()
{
    char output_cmd[CMD_BUFFER_SIZE];
    strcpy(output_cmd, "cat"); // Skládání příkazu
    strcat(output_cmd, sound(100.0, 100.0, 1.0)); // f = 100Hz, Amp =
100, t = 1s
    strcat(output_cmd, " > /dev/audio"); // pro konzoli
    printf("\n%s\n", output_cmd); //Zobrazení vykonávaného příkazu na
obrazovku
    system(output_cmd); // Zaslání příkazu na konzoli
    return 0;
}

/** @param freq: Frekvence v Hertz (s^-1)
 * @param amp: Amplituda v rozsahu 0-256.
 * @param time: Délka v sekundách.
 * @return Řetězec reprezentující data k přehrání (zápisu
 * /dev/audio).
 */
string sound(double freq, double amp, double time) {
    int i = 0;
```

```
double r = 6000; // Pocet znaku (vzorku), ktere cip prehraje /s
int n = (int)time*r;
char retVal[n*CHAR_SAMPLE_BUFFER_SIZE];
char tempVal[CHAR_SAMPLE_BUFFER_SIZE];
printf("Samples count: %d \n", n);
strcpy(retVal, "128"); // Na první pozici zapíšeme amplitudu 0
for (i = 0; i < n; i++) {
    sprintf(tempVal, "%d", (int) (sin((double)i*freq)*amp+128.0));
    strcat(retVal, tempVal);
}
return retVal;
}
```



CD-ROM

1. Zdrojový kód aplikace chatu – chapter5_soundgen.zip.



Řešený příklad

V dalším příkladu si vyzkoušíme síťovou komunikaci mezi klientem a serverem pomocí vlastního programu v programovacím jazyce C. Tato aplikace bude schopna zasílat textové zprávy mezi klientem a serverem. Na tomto příkladu se naučíte používat knihovny pro práci se sítěmi a TCP/IP pakety. Na tomto příkladu lze také vidět, že Linux jako operační systém opravdu podporuje nativní jazyk z Unixového prostředí a to sice jazyk C tím, že pro tento jazyk a jazyk C++ poskytuje velké množství nativních knihoven.

Poznámka autora: Problematika socketů a jejich seskupování, což je strategie, kterou je výhodné použít pro příklad více klientů připojených na server, kterým jsou zasílána data v kopii, je velice rozsáhlá a není v těchto skriptech probírána. Doporučuji se tedy tuto problematiku prostudovat z [6]. Ukázka funkční aplikace chatu, jejíž zdrojový kód je v kapitole cvičení, s řešenými příklady.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define MSG_SIZE 80
#define MAX_CLIENTS 95
#define DEFAULT_PORT 7400

// Prototypy funkcí
void exitClient(int fd, fd_set *readfds, char fd_array[], int
*num_clients);

int main(int argc, char *argv[]) {
    int i=0;

    int port;
    int num_clients = 0;
    int server_sockfd, client_sockfd;
    struct sockaddr_in server_address;
    int addresslen = sizeof(struct sockaddr_in);
    int fd;
    char fd_array[MAX_CLIENTS];
    fd_set readfds, testfds, clientfds;
    char msg[MSG_SIZE + 1];
    char kb_msg[MSG_SIZE + 10];

    /*Promenne pro klienta=====*/
    int sockfd;
    int result;
```

```
char hostname[MSG_SIZE];
struct hostent *hostinfo;
struct sockaddr_in address;
char alias[MSG_SIZE];
int clientid;

/*Client code=====*/
if(argc==2 || argc==4){
    if(!strcmp("-p",argv[1])){
        if(argc==2){
            printf("Neplatne parametry.\nPouziti: chat [-p PORT]
HOSTNAME\n");
            exit(0);
        }else{
            sscanf(argv[2],"%i",&port);
            strcpy(hostname,argv[3]);
        }
    }else{
        port=DEFAULT_PORT;
        strcpy(hostname,argv[1]);
    }
    printf("\n*** Spoustim klienta... (zadejte \"quit\" pro
ukonceni): \n");
    fflush(stdout);

    /* Vytvoreni socketu pro klienta */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    /* Pojmenuj socket po potvrzeni serverem */
    hostinfo = gethostbyname(hostname); /* Vyhledej hostname */
    address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
    address.sin_family = AF_INET;
    address.sin_port = htons(port);

    /* Spojení socketu na server socket */
    if(connect(sockfd, (struct sockaddr *)&address,
sizeof(address)) < 0){
        perror("spojuji...");
        exit(1);
    }

    fflush(stdout);

    FD_ZERO(&clientfds);
    FD_SET(sockfd,&clientfds);
    FD_SET(0,&clientfds); //stdin

    /* Čekání na zprávu ze serveru */
    while (1) {
        testfds=clientfds;
        select(FD_SETSIZE,&testfds,NULL,NULL,NULL);

        for(fd=0;fd<FD_SETSIZE;fd++){
            if(FD_ISSET(fd,&testfds)){
                if(fd==sockfd){ /*Přijetí dat z otevřeného socketu */
                    //přečtení dat z otevřeného socketu
                    result = read(sockfd, msg, MSG_SIZE);
                    msg[result] = '\0'; /* Ukončení stringu - null */
                    printf("%s", msg +1);
```



```

        if (msg[0] == 'X') {
            close(sockfd);
            exit(0);
        }
    }
    else if (fd == 0) { /*proces keyboard aktiviy*/
        fgets(kb_msg, MSG_SIZE+1, stdin);
        if (strcmp(kb_msg, "quit\n")==0) {
            sprintf(msg, "XClient se vypíná.\n");
            write(sockfd, msg, strlen(msg));
            close(sockfd); //uzavření klienta
            exit(0); //ukončení programu
        }
        else {
            sprintf(msg, "M%s", kb_msg);
            write(sockfd, msg, strlen(msg));
        }
    }
}
}
}
} // end Client code

/*Server=====*/
if(argc==1 || argc == 3){
    if(argc==3){
        if(!strcmp("-p",argv[1])){
            sscanf(argv[2],"%i",&port);
        }else{
            printf("Neplatný parametr.\nPoužití: chat [-p PORT]
HOSTNAME\n");
            exit(0);
        }
    }else port=DEFAULT_PORT;

    printf("\n*** Spouštění chat serveru (zadejte \"quit\" pro
ukončení): \n");
    fflush(stdout);

    /* Vytvoření a pojmenování socketu pro server */
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(port);
    bind(server_sockfd, (struct sockaddr *)&server_address,
addresslen);

    /* Vytvoření fronty čekající na spojení od klienta */
    listen(server_sockfd, 1);
    FD_ZERO(&readfds);
    FD_SET(server_sockfd, &readfds);
    FD_SET(0, &readfds); /* Přidání klávesnice do popisovače */

    /* Čekání na klienty a jejich požadavky */
    while (1) {
        testfds = readfds;
        select(FD_SETSIZE, &testfds, NULL, NULL, NULL);

        for (fd = 0; fd < FD_SETSIZE; fd++) {
            if (FD_ISSET(fd, &testfds)) {

```

```

        if (fd == server_sockfd) { /* Přijetí požadavku na
spojení od klienta */
            client_sockfd = accept(server_sockfd, NULL, NULL);

            if (num_clients < MAX_CLIENTS) {
                FD_SET(client_sockfd, &readfds);
                fd_array[num_clients]=client_sockfd;
                /*Client ID*/
                printf("Client %d připojen\n",num_clients++);
                fflush(stdout);

                sprintf(msg,"M%2d",client_sockfd);
                /*Zapsání 2 bajtového clientID */
                send(client_sockfd,msg,strlen(msg),0);
            }
            else {
                sprintf(msg, "XSorry, prilis mnoho klientů
Zkuste to pozdeji.\n");
                write(client_sockfd, msg, strlen(msg));
                close(client_sockfd);
            }
        }
    }
    else if (fd == 0) { /* Process keyboard activity */
        fgets(kb_msg, MSG_SIZE + 1, stdin);
        if (strcmp(kb_msg, "quit\n")==0) {
            sprintf(msg, "XServer se vypíná....\n");
            for (i = 0; i < num_clients ; i++) {
                write(fd_array[i], msg, strlen(msg));
                close(fd_array[i]);
            }
            close(server_sockfd);
            exit(0);
        }
        else {
            sprintf(msg, "M%s", kb_msg);
            for (i = 0; i < num_clients ; i++)
                write(fd_array[i], msg, strlen(msg));
        }
    }
    else if(fd) { /*zpracování zprávy od klienta*/
        //přečtení dat z otevřeného socketu
        result = read(fd, msg, MSG_SIZE);

        if(result==-1) perror("read()");
        else if(result>0){
            /*Přečtení ClientID*/
            sprintf(kb_msg,"M%2d",fd);
            msg[result]='\0';

            /*Spojení ClientID se */
            strcat(kb_msg,msg+1);

            /*Přeposlání zprávy pro ostatní klienty*/
            for(i=0;i<num_clients;i++){
                if (fd_array[i] != fd) /*Nezapisuj zprávu
stejnému klientovi*/
                    write(fd_array[i],kb_msg,strlen(kb_msg));
            }
            /*Tisk na serveru*/

```

```

        printf("%s", kb_msg+1);

        /*Ukončení klienta*/
        if(msg[0] == 'X'){
            exitClient(fd,&readfds,
fd_array,&num_clients);
        }
    }
else { /* Odhlašování klienta */
    exitClient(fd,&readfds, fd_array,&num_clients);
} //if
} //for
} //while
} //end Server code

} //main

/**
 * Funkce pro vyčištění a uvolnění proměnných po odhlášení
 klientovi
 */
void exitClient(int fd, fd_set *readfds, char fd_array[], int
*num_clients){
    int i;

    close(fd);
    FD_CLR(fd, readfds); //vymazání klienta ze seznamu
    for (i = 0; i < (*num_clients) - 1; i++)
        if (fd_array[i] == fd)
            break;
    for (; i < (*num_clients) - 1; i++)
        (fd_array[i]) = (fd_array[i + 1]);
    (*num_clients)--;
}

```



2. Zdrojový kód aplikace chatu - chapter5_chat.zip.



Řešený příklad

V posledním příkladu si vytvoříme ukázkovou statickou knihovnu pro práci s textem. Tato knihovna bude poskytovat služby pro podbarvení textu v konzoli, změnu fontu v konzoli a změnu tloušťky fontu (bold). Dále v rámci tohoto příkladu vytvoříme testovací aplikaci, která použije tuto knihovnu a otestuje její funkcionality. Níže je uveden zdrojový kód, který lze použít pro implementaci funkcí této knihovny. Pro změny atributů fontů v konzoli je použit standardní příkaz **tput** [5].

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

Zdrojový soubor knihovny

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <string.h>
#include "shellLib.h"
#define SHELL_CMD_SIZE 200
void changeShellColor(SHELL_COLORS pColor)
{
    char outputCmd[SHELL_CMD_SIZE];
    switch(pColor)
    {
        case SHELL_COL_RED:
            strcpy(outputCmd, "tput setaf 1");
            break;
        case SHELL_COL_BLUE:
            strcpy(outputCmd, "tput setaf 4");
            break;
        case SHELL_COL_GREEN:
            strcpy(outputCmd, "tput setaf 2");
            break;
        case SHELL_COL_ORANGE:
            strcpy(outputCmd, "tput setaf 3");
            break;
        case SHELL_COL_VIOLET:
            strcpy(outputCmd, "tput setaf 5");
            break;
        case SHELL_COL_LGTBLUE:
            strcpy(outputCmd, "tput setaf 6");
            break;
        case SHELL_COL_WHITE:
            strcpy(outputCmd, "tput setaf 7");
            break;
        case SHELL_COL_DEFAULT:
            strcpy(outputCmd, "tput init");
            break;
        default:
            strcpy(outputCmd, "tput init");
            break;
    }
    system(outputCmd);
}
```

```

void changeShellFont(char *pFontName)
{
    char outputCmd[SHELL_CMD_SIZE];
    strcpy(outputCmd, "sudo setfont /usr/share/consolefonts/");
    strcat(outputCmd, pFontName);
    system(outputCmd);
}

void changeShellFontWeight(bool pBold)
{
    if(pBold == true)
        system("tput bold");
    if(pBold == false)
        system("tput offbold");
}

```

Hlavičkový soubor knihovny

```

#ifndef SHELLLIB_H_
#define SHELLLIB_H_

#include <inttypes.h>

typedef enum { SHELL_COL_RED, SHELL_COL_BLUE,
               SHELL_COL_GREEN, SHELL_COL_ORANGE, SHELL_COL_VIOLET,
               SHELL_COL_LGTBLUE, SHELL_COL_WHITE, SHELL_COL_DEFAULT }
SHELL_COLORS;

typedef uint8_t bool;
#define true 1
#define false 0

void changeShellColor(SHELL_COLORS pColor);
void changeShellFont(char *pFontName);
void changeShellFontWeight(bool pBold);

#endif /* SHELLLIB_H_ */

```

Zdrojový kód testovacího příkladu s ukázkou výstupu:

Při kompilaci této testovací aplikace (**Obr. 23**) nesmíme zapomenout zadat spojovacímu programu cestu a název ke knihovně, která byla vytvořena v předchozím příkladu následujícím způsobem

```

gcc -L"/home/xxx/cesta/ke_knihovne/" -o "shellLibTester"
./src/shellLibTester.o -lsHELLLib

```

Zdrojový kód testovací aplikace

```

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

int main(void) {
    changeShellColor(SHELL_COL_BLUE);
    changeShellFontWeight(true);
}

```

```
puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */  
changeShellColor(SHELL_COL_DEFAULT);  
return EXIT_SUCCESS;  
}
```



CD-ROM

1. Zdrojový kód statické knihovny pro práci s konzolí a aplikaci využívající tuto knihovnu - chapter5_library.zip.



Řešený příklad

Vytvořte aplikaci, která vytvoří dva procesy pomocí funkce `fork0`. Druhý proces tedy potomek spustí stejný příkaz jako první příklad této kapitoly, čili příkaz pro zjištění přenosové rychlosti disku.

```
hdparm -tT /dev/sdx
```

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

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

/* Spawn a child process running a new program.  PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list.  Returns the process id of
   the spawned process.  */

int spawn (char* program, char** arg_list)
{
    pid_t child_pid;

    /* Duplicate this process.  */
    child_pid = fork ();
    if (child_pid != 0)
        /* This is the parent process.  */
        return child_pid;
    else {
        /* Now execute PROGRAM, searching for it in the path.  */
        execvp (program, arg_list);
        /* The execvp function returns only if an error occurs.  */
        fprintf (stderr, "an error occurred in execvp\n");
        abort ();
    }
}

int main ()
{
    /* The argument list to pass to the "ls" command.  */
    char* arg_list[] = {
        "ls",          /* argv[0], the name of the program.  */
        "-l",
        "/",
        NULL           /* The argument list must end with a NULL.  */
    };

    /* Spawn a child process running the "ls" command.  Ignore the
       returned child process id.  */
    spawn ("ls", arg_list);

    printf ("done with main program\n");
}
```

```
return 0;  
}
```



CD-ROM

1. Zdrojový kód aplikace pracující s dělením procesů - chapter5_processes.zip.

CVIČENÍ č. 4. – Řešené příklady (kapitola 6).



Řešený příklad

Vytvořte aplikaci, která bude obsahovat nový uživatelský datový typ telefonního seznamu dle následující specifikace.

Název proměnné	Typ
Jméno	char16_t
Příjmení	char16_t
Telefoní číslo	uint32_t
union { uint32_t index, uint16_t key }	Anonymní union

Vytvořte aplikaci, který bude moci ukládat položky telefonního seznamu do souboru.

Poznámka autora: Pro ukládání do souboru lze doporučit ukládání do formátu .csv, který je možné posléze otevřít a zpracovávat také v libovolném tabulkovém procesoru.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

```
#include <stdio.h>
#include <stdlib.h>
#include <uchar.h>
#include <inttypes.h>
#include "libCSV.h"

#define MAX_PHONE_BOOK_ITEMS 4096 // Maximální počet položek v tel. seznamu
#define PB_ITEM_MAX_LEN 100 // Maximální délka atributu položky v
tel. seznamu
#define MENU_NONE 'N' // Menu položka pro pokračování
#define MENU_WRITE_TO_FILE 'W' // Menu položka pro zápis do souboru
#define MENU_QUIT 'Q' // Menu položka pro ukončení aplikace

/**
 * Struktura položky telefonního seznamu
 */
typedef struct {
    char name[PB_ITEM_MAX_LEN];
    char surname[PB_ITEM_MAX_LEN];
    char phone_num[PB_ITEM_MAX_LEN];
    union {
        int index;
        int key;
    };
} PhoneBookItem;

// Funkční prototypy
void consoleReadLine(string text_buf);
int phoneBookItemToList(PhoneBookItem item, string *list_buf);

// Globální proměnné
```

```

PhoneBookItem PhoneBook[MAX_PHONE_BOOK_ITEMS]; // Telefoni seznam
char MenuSelection = MENU_NONE; // Stavova promenna menu

int main(void) {
    uint16_t i = 0, j = 0, list_count = 0;
    char itemsVar[100][100];

    while(MenuSelection == MENU_NONE)
    {
        // Načtení informací do telefoního seznamu od uživatele
        fflush(stdin);
        puts("Zadejte jmeno: ");
        consoleReadLine(PhoneBook[i].name);
        puts("\nZadejte primeni: ");
        consoleReadLine(PhoneBook[i].surname);
        puts("\nZadejte tel. cislo: ");
        consoleReadLine(PhoneBook[i].phone_num);
        PhoneBook[i].index = i;
        PhoneBook[i].key = i;

        // Tisk základní menu nabídky programu
        puts("\n=====\\n");
        puts("Vyberte si dalsi operaci: ");
        puts("\n\tN = Zadani dalsi polozky telefoního seznamu");
        puts("\n\tW = Zapsani telefoního seznamu do souboru");
        puts("\n\tQ = Ukonceni aplikace\\n\\n");
        MenuSelection = getchar();
        i++;
    }

    switch(MenuSelection) // Výběr akce dle menu
    {
        case MENU_WRITE_TO_FILE: // Zápis do souboru
            printf("\\n\\nUkladam do souboru...");
            for(j = 0; j < i; j++)
            {
                list_count = phoneBookItemToList(PhoneBook[j], (string
*)itemsVar);
                writeCsvFileRow("TelefoniSeznam.csv", (string *)itemsVar,
list_count);
            }
            puts("Uloženo!!!");
            puts("\\nDekujeme za pouziti aplikace Telefoni seznam v 1.0");
            break;
        case MENU_QUIT: // Konec programu
            puts("\\nDekujeme za pouziti aplikace Telefoni seznam v 1.0");
            break;
    }
    closeCsvFile();
    return EXIT_SUCCESS;
}

/**
 * Funkce pro převod struktury PhoneBookItem na pole řetězců
 * @param item Položka telefoního seznamu
 * @param list_buf Ukazatel na seznam do kterého budou položky uloženy
 * @return Počet položek v seznamu
 */
int phoneBookItemToList(PhoneBookItem item, string *list_buf)
{
    list_buf[0] = item.name;

```

```

        list_buf[1] = item.surname;
        list_buf[2] = item.phone_num;

        return 3;
    }

/**
 * Funkce pro přečtení řádku z konzole, ze stdin
 * @param text_buf Ukazatel na proměnnou do které bude text uložen
 */
void consoleReadLine(string text_buf)
{
    gets(text_buf);
}

```

Zdrojový kód knihovny pro ukládání dat do souboru .csv.

Hlavičkový soubor CSV knihovny (libCSV)

```

#include <inttypes.h>

#define true 1
#define false 0

#ifndef LIBCSV_H_
#define LIBCSV_H_

typedef char * string;
typedef uint8_t bool;

/**
 * Přečte CSV soubor
 * @param fp cesta k .csv souboru
 * @param buffer buffer do kterého bude uložen načtený obsah
 */
void readCsvFile(string fp, string *buffer);

/**
 * Funkce pro zápis dat (jednoho řádku) do .csv souboru. Data budou dle
 standardního .csv formátu oddělena středníkem
 * @param fp cesta k .csv souboru do kterého budou data zapsána
 * @param src_data pole řetězců k zapsání. Jeden prvek pole představuje
 soupec, celé pole představuje řádek
 * @param src_len počet prvků (sloupců) k zapsání
 * @return true jestliže se zápis vydařil | false jestliže se zápis
 nezdařil
 */
bool writeCsvFileRow(string fp, string *src_data, uint16_t src_len);

/**
 * Otevře CSV souboru
 * @param fp cesta k .csv souboru
 */
void openCsvFile(string fp);

/**
 * Funkce pro zavření .csv souboru
 * @return true jestliže se zavření vydařilo | false jestliže se zavření
 nezdařilo
 */

```

```

*/
bool closeCsvFile();

#endif /* LIBCSV_H_ */

```

Soubor zdrojového kódu CSV knihovny (libCSV)

```

#include <stdio.h>
#include <stdint.h>
#include "libCSV.h"

FILE *_csvFp;

void readCsvFile(string fp, string *buffer)
{
    int i;
    _csvFp = fopen(fp, "r");
    if (_csvFp == NULL)
        exit(0);
}

bool writeCsvFileRow(string fp, string *src_data, uint16_t src_len)
{
    int i = 0;
    if (_csvFp == NULL)
        _csvFp = fopen(fp, "a");
    if (_csvFp == NULL)
        exit(0);

    /* Zápis dat do souboru - zapíše se jeden řádek */
    for (i=0; i<=src_len; i++)
        fprintf(_csvFp, "%s;", src_data[i]);
    fprintf(_csvFp, "\n");
    return true;
}

void openCsvFile(string fp)
{
    _csvFp = fopen(fp, "a+");
    if (_csvFp == NULL)
        exit(0);
}

bool closeCsvFile()
{
    if (_csvFp != NULL) {
        fclose(_csvFp); /* close the file */
        _csvFp = NULL;
    }
    return true;
}

```



CD-ROM

1. Zdrojový kód aplikace telefonního seznamu – chapter6_pb.zip



Řešený příklad

Vytvořte jednoduchou aplikaci v jazyce Objective-C, která vytvoří objekt zlomku s atributy jmenovatele a čitatele a bude mít metody pro výpočet a zobrazení reálného výsledku tohoto zlomku.

Řešení:

Jedno z možných řešení zadaného příkladu je pomocí následujícího zdrojového kódu.

Hlavičkový soubor obsahující rozhraní třídy Fraction

```
#import <Foundation/NSObject.h>

@interface Fraction: NSObject {
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end
```

Zdrojový soubor obsahující implementaci třídy Fraction

```
#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(void) print {
    printf( "%i/%i", numerator, denominator );
}

-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
@end
```

Zdrojový soubor obsahující hlavní funkci main

```
#import <stdio.h>
#import "Fraction.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] init];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    // free memory
    [frac release];

    return 0;
}
```

Tento zdrojový soubor stačí posléze zkompilovat v příkazové řádce pomocí následujícího příkazu

```
gcc objcTest .m -Wall -lobjc
```

!!! V systému musí být nainstalován kompilátor, který podporuje jazyk Objective-C, čili například gcc.!!!

**CD-ROM**

1. Zdrojový kód aplikace v jazyku Objective-C – chapter6_objc.zip.

CVIČENÍ č. 5. – Řešené příklady (kapitola 7).



Řešený příklad

Prostudujte si příklad pro výpočet obrazové 2D konvouce pomocí grafického jádra, který je popsán v kapitole řešených příkladů.

Řešení:

Následují zdrojové kódy aplikace pro výpočet 2D konvoluce implementované v GPU jádře.

Convolution.cu – soubor s kódem pro GPU

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <helper_cuda.h>
#include "convolutionFFT2D_common.h"
#include "convolutionFFT2D.cuh"

////////////////////////////////////
/////
/// Position convolution kernel center at (0, 0) in the image
////////////////////////////////////
/////
extern "C" void padKernel(
    float *d_Dst,
    float *d_Src,
    int fftH,
    int fftW,
    int kernelH,
    int kernelW,
    int kernelY,
    int kernelX
)
{
    assert(d_Src != d_Dst);
    dim3 threads(32, 8);
    dim3 grid(iDivUp(kernelW, threads.x), iDivUp(kernelH, threads.y));

    SET_FLOAT_BASE;
    padKernel_kernel<<<grid, threads>>>(
        d_Dst,
        d_Src,
        fftH,
        fftW,
        kernelH,
        kernelW,
        kernelY,
        kernelX
    );
    getLastCudaError("padKernel_kernel<<<>> execution failed\n");
}
```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
// Prepare data for "pad to border" addressing mode
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
extern "C" void padDataClampToBorder(
    float *d_Dst,
    float *d_Src,
    int fftH,
    int fftW,
    int dataH,
    int dataW,
    int kernelW,
    int kernelH,
    int kernelY,
    int kernelX
)
{
    assert(d_Src != d_Dst);
    dim3 threads(32, 8);
    dim3 grid(iDivUp(fftW, threads.x), iDivUp(fftH, threads.y));

    SET_FLOAT_BASE;
    padDataClampToBorder_kernel<<<grid, threads>>>(
        d_Dst,
        d_Src,
        fftH,
        fftW,
        dataH,
        dataW,
        kernelH,
        kernelW,
        kernelY,
        kernelX
    );
    getLastCudaError("padDataClampToBorder_kernel<<<>>> execution
failed\n");
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
// Modulate Fourier image of padded data by Fourier image of padded kernel
// and normalize by FFT size
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
extern "C" void modulateAndNormalize(
    fComplex *d_Dst,
    fComplex *d_Src,
    int fftH,
    int fftW,
    int padding
)
{
    assert(fftW % 2 == 0);
    const int dataSize = fftH * (fftW / 2 + padding);

    modulateAndNormalize_kernel<<<iDivUp(dataSize, 256), 256>>>(
        d_Dst,
        d_Src,

```



```

        dataSize,
        1.0f / (float)(fftW *fftH)
    );
    getLastCudaError("modulateAndNormalize() execution failed\n");
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
// 2D R2C / C2R post/preprocessing kernels
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
static const double PI = 3.1415926535897932384626433832795;
static const uint BLOCKDIM = 256;

extern "C" void spPostprocess2D(
    void *d_Dst,
    void *d_Src,
    uint DY,
    uint DX,
    uint padding,
    int dir
)
{
    assert(d_Src != d_Dst);
    assert(DX % 2 == 0);

#ifdef POWER_OF_TWO
    uint log2DX, log2DY;
    uint factorizationRemX = factorRadix2(log2DX, DX);
    uint factorizationRemY = factorRadix2(log2DY, DY);
    assert(factorizationRemX == 1 && factorizationRemY == 1);
#endif

    const uint threadCount = DY * (DX / 2);
    const double phaseBase = dir * PI / (double)DX;

    SET_FCOMPLEX_BASE;
    spPostprocess2D_kernel<<<iDivUp(threadCount, BLOCKDIM), BLOCKDIM>>>(
        (fComplex *)d_Dst,
        (fComplex *)d_Src,
        DY, DX, threadCount, padding,
        (float)phaseBase
    );
    getLastCudaError("spPostprocess2D_kernel<<<>>> execution failed\n");
}

extern "C" void spPreprocess2D(
    void *d_Dst,
    void *d_Src,
    uint DY,
    uint DX,
    uint padding,
    int dir
)
{
    assert(d_Src != d_Dst);
    assert(DX % 2 == 0);

#ifdef POWER_OF_TWO

```

```

uint log2DX, log2DY;
uint factorizationRemX = factorRadix2(log2DX, DX);
uint factorizationRemY = factorRadix2(log2DY, DY);
assert(factorizationRemX == 1 && factorizationRemY == 1);
#endif

const uint threadCount = DY * (DX / 2);
const double phaseBase = -dir * PI / (double)DX;

SET_FCOMPLEX_BASE;
spPreprocess2D_kernel<<<iDivUp(threadCount, BLOCKDIM), BLOCKDIM>>>(
    (fComplex *)d_Dst,
    (fComplex *)d_Src,
    DY, DX, threadCount, padding,
    (float)phaseBase
);
getLastCudaError("spPreprocess2D_kernel<<<>>> execution failed\n");
}

////////////////////////////////////
/////
// Combined spPostprocess2D + modulateAndNormalize + spPreprocess2D
////////////////////////////////////
/////
extern "C" void spProcess2D(
    void *d_Dst,
    void *d_SrcA,
    void *d_SrcB,
    uint DY,
    uint DX,
    int dir
)
{
    assert(DY % 2 == 0);

#ifdef POWER_OF_TWO
    uint log2DX, log2DY;
    uint factorizationRemX = factorRadix2(log2DX, DX);
    uint factorizationRemY = factorRadix2(log2DY, DY);
    assert(factorizationRemX == 1 && factorizationRemY == 1);
#endif

    const uint threadCount = (DY / 2) * DX;
    const double phaseBase = dir * PI / (double)DX;

    SET_FCOMPLEX_BASE_A;
    SET_FCOMPLEX_BASE_B;
    spProcess2D_kernel<<<iDivUp(threadCount, BLOCKDIM), BLOCKDIM>>>(
        (fComplex *)d_Dst,
        (fComplex *)d_SrcA,
        (fComplex *)d_SrcB,
        DY, DX, threadCount,
        (float)phaseBase,
        0.5f / (float)(DY * DX)
    );
    getLastCudaError("spProcess2D_kernel<<<>>> execution failed\n");
}

```

Convolution.cpp – zdrojový kód psaný v jazyce C++ pro spuštění na CPU

```

////////////////////////////////////
////
// Reference straightfroward CPU convolution
////////////////////////////////////
////
extern "C" void convolutionClampToBorderCPU(
    float *h_Result,
    float *h_Data,
    float *h_Kernel,
    int dataH,
    int dataW,
    int kernelH,
    int kernelW,
    int kernelY,
    int kernelX
)
{
    for (int y = 0; y < dataH; y++)
        for (int x = 0; x < dataW; x++)
        {
            double sum = 0;

            for (int ky = -(kernelH - kernelY - 1); ky <= kernelY; ky++)
                for (int kx = -(kernelW - kernelX - 1); kx <= kernelX;
kx++)
                    {
                        int dy = y + ky;
                        int dx = x + kx;

                        if (dy < 0) dy = 0;

                        if (dx < 0) dx = 0;

                        if (dy >= dataH) dy = dataH - 1;

                        if (dx >= dataW) dx = dataW - 1;

                        sum += h_Data[dy * dataW + dx] * h_Kernel[(kernelY -
ky) * kernelW + (kernelX - kx)];
                    }

            h_Result[y * dataW + x] = (float)sum;
        }
}

```

**CD-ROM**

1. Zdrojový kód aplikace pro výpočet konvoluce na GPU – chapter7_gpuConv.zip.