evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA

FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Ing. Ivo Špička, Ph.D.

Ostrava 2013

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

Název: PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Autor: Ing. Ivo Špička, Ph.D.

Vydání: první, 2013

Počet stran: 155

Náklad: 5

Jazyková korektura: nebyla provedena.

Tyto studijní materiály vznikly za finanční podpory Evropského sociálního fondu a rozpočtu České republiky v rámci řešení projektu Operačního programu Vzdělávání pro konkurenceschopnost.

Název: Modernizace výukových materiálů a didaktických metod

Číslo: CZ.1.07/2.2.00/15.0463

Realizace: Vysoká škola báňská – Technická univerzita Ostrava

© Ing. Ivo Špička, Ph.D.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

EVROPSKÝ SOCIÁLNÍ FOND V ČR · EVROPSKÁ UNIE · MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY · OP Vzdělávání pro konkurenceschopnost

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Úvod

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

1 ÚVOD ... 4
1.1 Pokyny ke studiu ... 5
1.1.1 Název předmětu ... 5
1.1.2 Prerekvizity ... 5
1.1.3 Cílem předmětu a výstupy z učení ... 5
1.1.4 Pro koho je předmět určen ... 5
1.1.5 Při studiu každé kapitoly doporučujeme následující postup ... 5
1.1.6 Způsob komunikace s vyučujícími ... 5
1.2 Seznámení s jazykem C- Základy jazyka C ... 5
1.3 Historie ... 6
1.4 Standardizace ... 6
1.5 Obecné vlastnosti programu v jazyce C ... 6
1.6 První program v C ... 6
1.6.1 Řešené úlohy ... 7
1.7 Vývojové prostředí Microsoft C++ express ... 7
1.8 Překlad programu, sestavení programu, spustitelný program ... 16
1.9 Chyby v překladu ... 16
1.10 Kostra programu ... 20
1.11 Datové typy ... 20
1.12 Celočíselné konstanty ... 20
1.13 Reálné konstanty ... 21
1.14 Proměnné a konstanty ... 21
1.15 Klíčová slova ... 21
1.16 Přetypování ... 22
1.17 Výpis pomocí printf() ... 23
1.18 Načtení hodnoty ... 23
1.19 Operátory ... 23
1.19.1 Matematické operátory ... 24
1.19.2 Bitové operátory ... 24
1.19.3 Operátory porovnávání ... 25
1.19.4 Logické operátory ... 25
1.19.5 Spojení přiřazovacího příkazu a operátoru ... 25

1.19.6 Unární operátory ... 26

1.20 Podmínky a cykly ... 26

1.20.1 Podmínky ... 26

1.20.2 Vícenásobné větvení ... 28

1.20.3 Ternární výraz ... 29

1.20.4 Cykly ... 29

1.20.5 Break a Continue ... 30

1.20.6 Řetězce v jazyce C ... 30

1.20.7 Řetězec jako literál ... 30

1.20.8 Escape sekvence ... 30

1.21 Inicializace řetězce ... 31

1.22 Práce s řetězci ... 31

1.22.1 Výčet funkcí ... 31

1.22.2 Krátký program ... 33

1.22.3 Porovnávání řetězců ... 34

1.22.4 Procházení řetězců ... 34

1.23 Funkce ... 34

1.23.1 Deklarace a definice funkce ... 34

1.23.2 Deklarace funkce ... 35

1.23.3 Předávání parametrů funkcím. ... 35

1.23.4 Formální a skutečný parametr ... 35

1.23.5 Způsoby předávání parametru ... 36

1.23.6 Rekurze ... 46

1.24 Funkce main () ... 47

1.25 Pole ... 48

1.25.1 Úvod do polí ... 48

1.25.2 Deklarace pole ... 48

1.25.3 Inicializace pole ... 49

1.25.4 Práce s polem ... 49

1.25.5 Indexace ... 49

1.25.6 Porovnávání polí ... 49

1.25.7 Pole jako parametr funkce ... 50

1.25.8 Vícerozměrná pole ... 50

1 ÚVOD

OBSAH KAPITOLY:

Jaké datové typy znáte

Jak jsou definovány konstanty

Jak jsou definovány proměnné

Co znamená přetypovaní

Jaké jsou instrukce pro načtení a výpis

CÍL:

Po prostudování tohoto odstavce budete umět:

- Seznámí s historii jazyka, standardizacemi,
- Vytvoří první program v C
- Seznámí s vývojovým prostředím Microsoft C++
- Používat datové typy, vytvářet proměnné a konstanty, pole, funkce, pracovat s řetězci, cykly a podmínkami, načtení a výpis hodnot.

1.1 POKYNY KE STUDIU

1.1.1 Název předmětu

Pro předmět Programování řídicích systémů 5 semestru bakalářského studijního oboru Řízení průmyslových systémů jste obdrželi studijní balík obsahující integrované skriptum pro kombinované studium obsahující i pokyny ke studiu.

1.1.2 Prerekvizity

Pro studium tohoto předmětu se předpokládá absolvování předmětu ....

1.1.3 Cílem předmětu a výstupy z učení

Předmět seznamuje posluchače s teoretickými i praktickými otázkami programování řídicích systémů s počítači a to především v oblasti reálného času. Doplňuje teorii programování řídicích systémů o základní znalosti operačních systému. Pro prezentaci a cvičení je používán jazyk Visual C++ a prostředí operačního systému Windows.
Student bude umět analyzovat úlohy počítačového řízení.
Student porozumí základním principům programování v jazyce C.
Student bude schopen

- analyzovat základní principy chování OS;
- vytvářet základní programy v prostředí operačního systému Windows.

1.1.4 Pro koho je předmět určen

Předmět je zařazen do magisterského studia oborů Ekonomika a řízení průmyslových systémů studijního programu Řízení průmyslových systémů, ale může jej studovat i zájemce z kteréhokoliv jiného oboru, pokud splňuje požadované prerekvizity.
Studijní opora 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.

1.1.5 Při studiu každé kapitoly doporučujeme následující postup

Na začátku každého bloku je uveden stručně obsah znalostí a dovedností, které po prostudování získáte. Abyste se ujistili, zda jste látce porozuměli, jsou na konci každého bloku uvedeny otázky, shrnují podstatné znalosti. Odpovědi na tyto otázky naleznete přímo v textu nebo v souhrnu nejdůležitějších pojmů na konci každého bloku.

1.1.6 Způsob komunikace s vyučujícími

Důležitou součástí je návštěva seminářů a cvičení, pro denní formu studia se doporučuje pravidelná účast na přednáškách.
Problémy a nejasnosti je možno řešit přímo s vyučujícími na seminářích a po domluvě i osobními konzultacemi. Jednodušší problémy je možno řešit emailovou korespondencí na kontakty uvedené ve školním seznamu.
Na začátku semestru budou posluchači seznámeni s projekty, které budou potřebné ke zdárnému ukončení studia tohoto předmětu.

1.2 SEZNÁMENÍ S JAZYKEM C- ZÁKLADY JAZYKA C

Programovací jazyk C je rozšířený univerzální programovací jazyk. Jde jazyk nižší úrovně, používá standardní datové typy, jako jsou znaky, celá a reální čísla, má blízko ke strojové úrovni a zároveň je i jazykem vyšší úrovně, používá uživatelem definované datové typy. Má

velmi úsporné vyjadřování, je strukturovaný, má velký soubor operátorů a používá moderní datové struktury. Samotný jazyk C je kompaktní, efektivní a výkonný. Veškeré funkce jsou obsaženy v knihovnách. Tento návrh odděluje vlastnosti jazyka od implementace spojenou s konkrétním procesorem či architekturou a tím umožňuje snadněji psát přenositelné programy.

1.3 HISTORIE

Jazyk C vytvořil v Bellových laboratořích AT&T Denis Ritchie. Záměrem bylo napsat jazyk pro snadnou a přenositelnou implementaci Unixu. Na vývoji jazyka se dále podíleli Brian Kernighan a Ken Thompson. Přímým předchůdcem programovacího jazyka C byl jazyk B, který byl vyvinut Kenem Thompsonem. Byl kandidátem pro přepis zdrojového kódu jádra Unixu napsaného v assembleru, ale ukázal se pro tento účel nevhodný.
Roku 1972 díky Dennisu Ritchiemu světlo světa spatřil nový programovací jazyk C. Byl tedy vytvořen jazyk s bohatými datovými typy, přičemž zůstala zachována jednoduchost a přímý přístup k hardware.
V roce 1983 vyvinul Bjarne Stroustrup z Bellových laboratoří jazyk C++ jako objektové rozšíření jazyka C.

1.4 STANDARDIZACE

Pojem ISO norma jazyka C definuje moderní vyšší programovací jazyk všeobecného použití. ISO je zkratkou slov International Standards Organisation. ISO norma jazyka C je bezpečnější, než byl původní jazyk K&R C. Současně je důležité, že většina zdrojových textů, napsaných před vydáním normy, je novými překladači přijímána.
http://homel.vsb.cz/~s1a10/educ/C_CPP/kurs_C/ch01.html
http://www.jazykc.ic.cz/vyuka/historie.html
http://www.jazykc.ic.cz/vyuka/historie.html
Překladače C nabízí řada nezávislých producentů programového vybavení. Dostupné jsou i verze volně (bezplatně) šiřitelné. Podstatné však je, že C je k dispozici prakticky pro libovolné technické vybavení (hardware). Budeme-li ISO normu jazyka C dodržovat, máme velkou šanci, že budeme schopni přenést zdrojový text na jiný stroj (s jiným procesorem i OS), program přeložit, spojit s funkcemi z knihoven a konečně i spustit. Rozhodně by neměly vyvstat neřešitelné problémy.C99
Norma C99 vznikla v roce 1999. Následně byla přijata i jako ANSI standard. Přináší několik nových vlastností jako inline funkce a deklarace proměnných kdekoli.

1.5 OBECNÉ VLASTNOSTI PROGRAMU V JAZYCE C

1.6 PRVNÍ PROGRAM V C

V kapitole je shrnut význam hlavičkových a zdrojových souborů a stručně popsána funkce main () a printf(). Jsou popsány možnosti editace programu a uvedeny doporučené postupy pro pojmenování zdrojových souborů. Výše uvedené pojmy jsou použity v programu „Muj první program v jazyce C”.
Zdrojový kód
První program v jakémkoliv programovacím jazyce bývá program „Hello world“. To protože vypsat něco na výstup patří k základním programovým operacím a mimo jiné jde i o primární ladící nástroj (tedy způsob jak nalézt v programu chyby). Zdrojový kód v následujícím textu bude napsán čistém ANSI C.
Zdrojový kód všech běžných programovacích jazyků je běžný textový soubor, pro psaní programu můžeme použít libovolný textový editor (nejsou vhodné tzv. textové procesory

OpenOffice, Write nebo Microsoft Word). Vhodné je použít přímo vývojové prostředí, použijeme prostředí Visual studio C++ Express.
V editačním okně zdrojového souboru napíšeme následující kód.

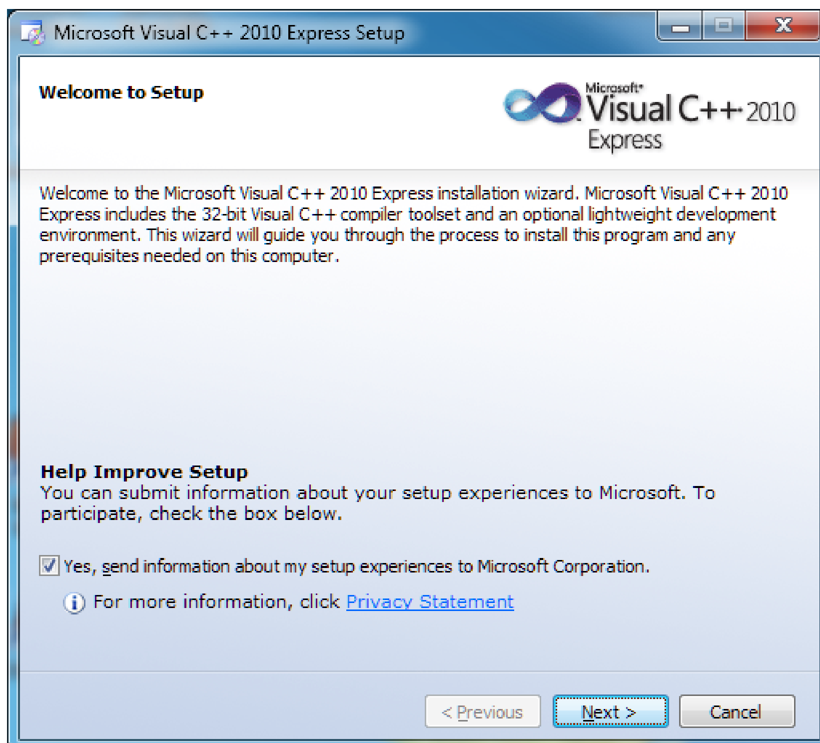
1.6.1 Řešené úlohy

```
1
2  #include <stdio.h>
3
4  /* můj první program */
5
6  int main(int argc, char **argv)
7  {
8  printf("Muj prvy program!\n"); // vypis textu
9  return 0;
10 }
11
```

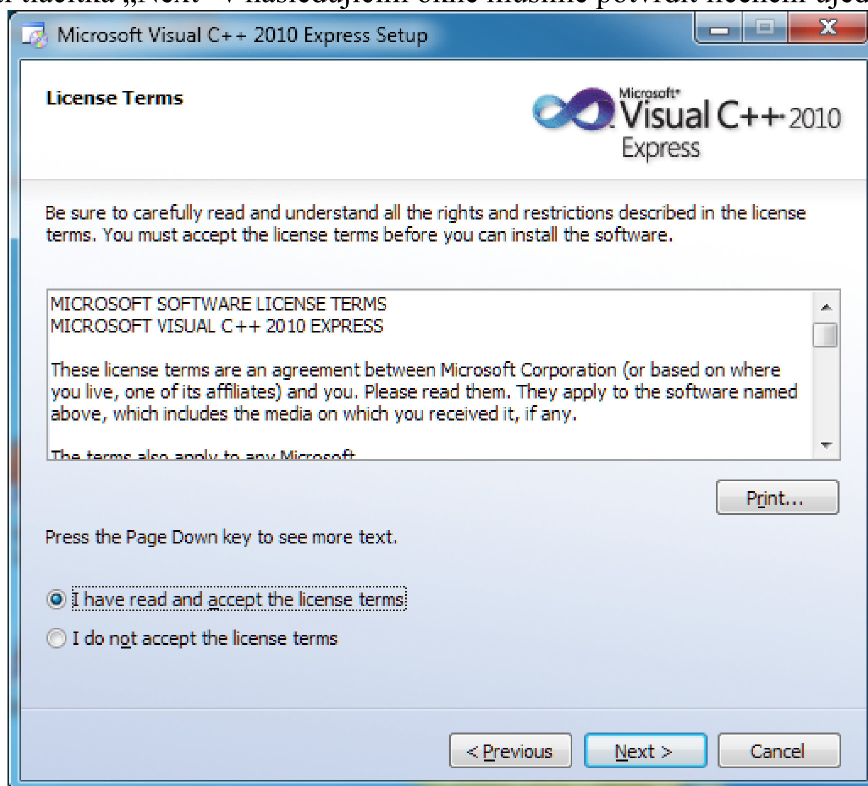
Nyní si rozebereme každý řádek zvlášť. Druhý řádek uvozený znakem # obsahuje direktivu. Direktiva #include říká překladači, že na toto místo má vložit soubor stdio.h. Soubory s příponou .h jsou tzv. hlavičkové soubory. Obsahují deklarace funkcí a datových typů a tak se na ně lze se na ně dívat jako na rozšíření jazyka o další prvky.
Jazyk C je v základu velmi minimalistický, jeho možnosti lze značně rozšířit vložením vhodného hlavičkového souboru. Soubor stdio.h obsahuje funkce pro práci se vstupem a výstupem (input-output, IO).
Na čtvrté řádce máme ukázku komentáře. V jazyce C existuje pouze jediný typ komentáře, a to víceřádkový komentář /* */(ve standardu C99 je navíc řádkový komentář //, v našem příkladu na řádce osm). Vše, co je uvozeno mezi hvězdičkami, nebo co leží za dvojitým lomítkem až do konce řádku, je při zpracování kódu ignorováno. Komentáře jsou velmi mocný nástroj a nemělo by se jimi šetřit, vysvětlují použití kódu (v budoucnu si pak programátor ušetří spoustu času, který by jinak strávil porozumění cizího nebo i svého kódu).
Jádro programu tvoří funkce main (). Tato funkce je základem každého programu v jazyce C. V každém programu musí být právě jedna funkce main, neboť se jedná o vstupní bod programu, program vždy začíná voláním funkce main. Bližší vysvětlení týkající se tvaru funkce a jejích parametrů bude podáno v sekci o funkcích. V těle funkce main () se nachází jediná funkce printf(). Deklaraci této funkce obsahuje hlavičkový soubor stdio.h. Funkce printf() nám vypíše na monitor text „Muj prvy program!“ a přejde na nový řádek.
Znak \n patří mezi řídící znaky, což jsou Meta znaky pro často používané netisknutelné znaky. \n symbolizuje znak nového řádku (z New Line). Funkci main () ukončuje příkaz return 0. V tomto místě se vykonávání funkce main ukončí a návratová hodnota funkce je nula (tím říkáme, že program doběhl bez chyby).

1.7 VÝVOJOVÉ PROSTŘEDÍ MICROSOFT C++ EXPRESS

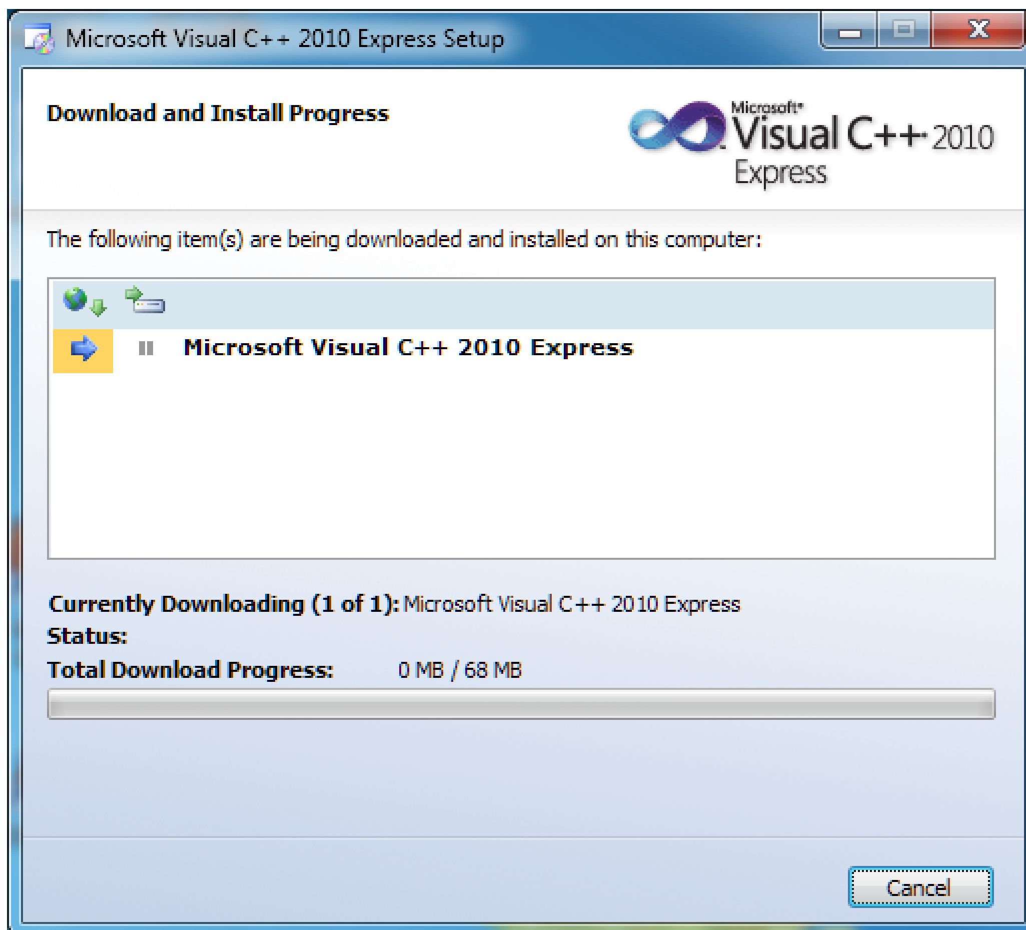
Pokud na počítači nemáme instalováno prostředí Microsoft Visual C++ Express, pak si jej nejprve musíme stáhnout z webu Microsoftu, například pomocí odkazu
http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express
Po spuštění nás povede dále průvodce instalací.



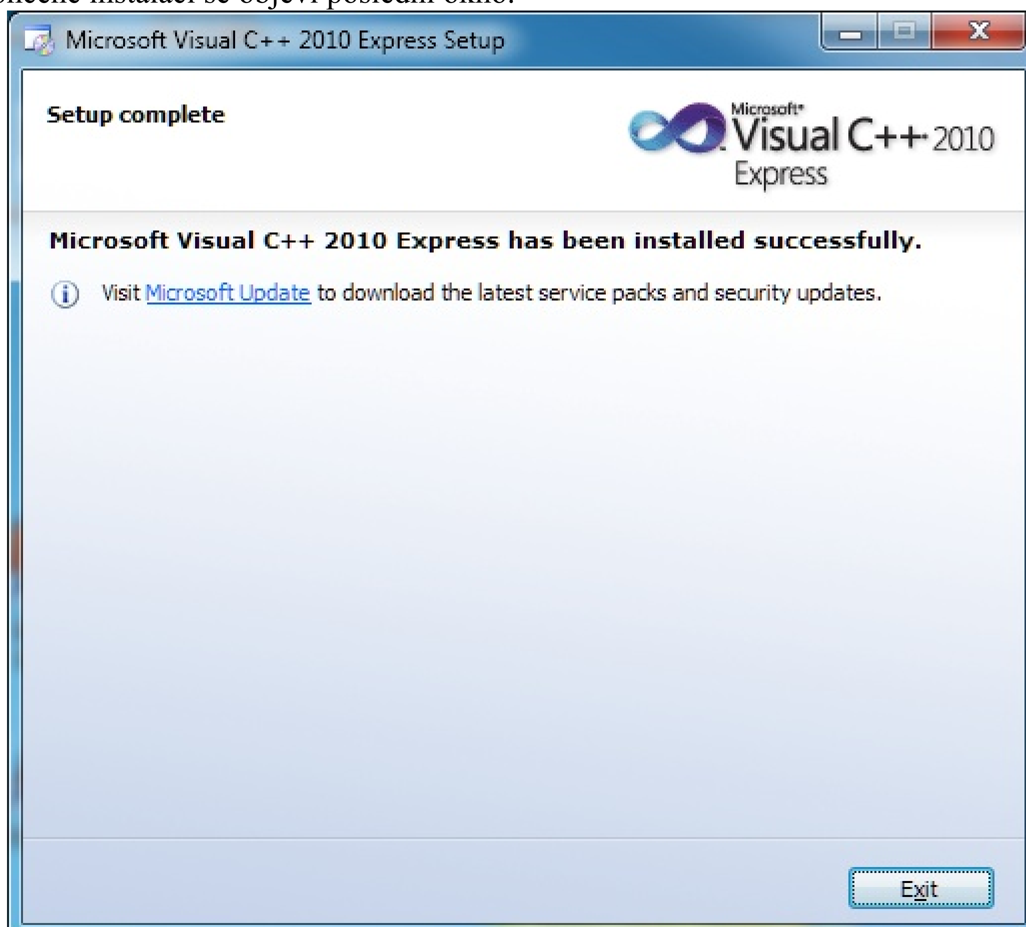
Po zmáčknutí tlačítka „Next“ v následujícím okně musíme potvrdit licenční ujednání



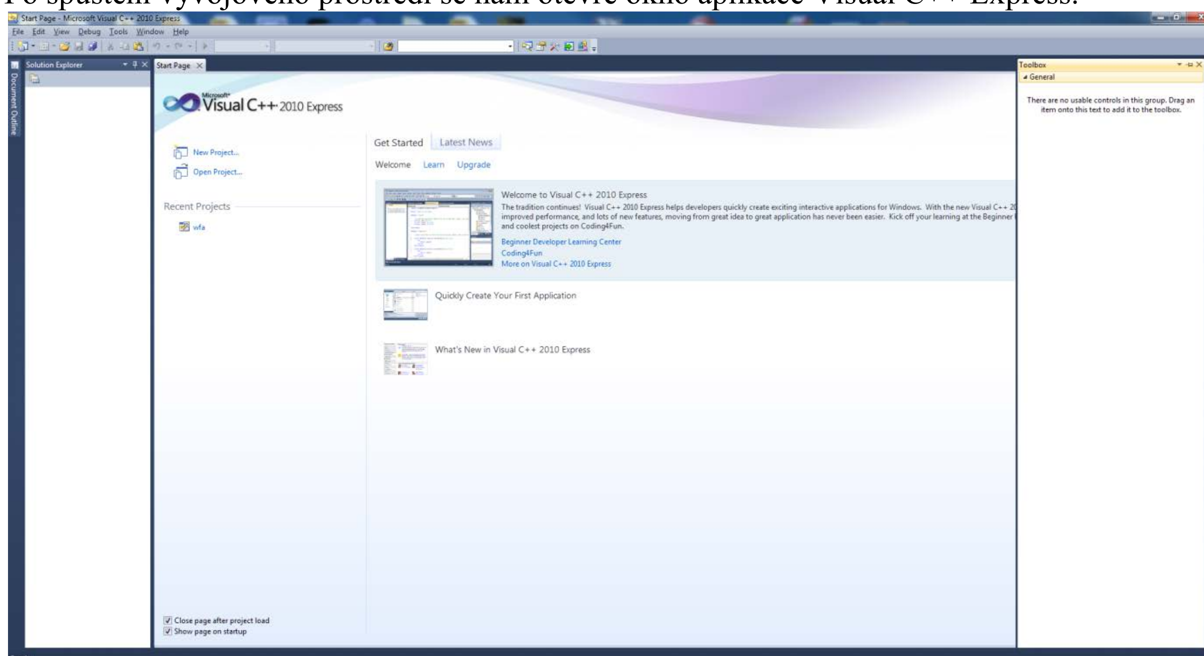
Dvakrát opět stiskneme tlačítko Next (cílový adresář nelze měnit) a spustí se instalace.



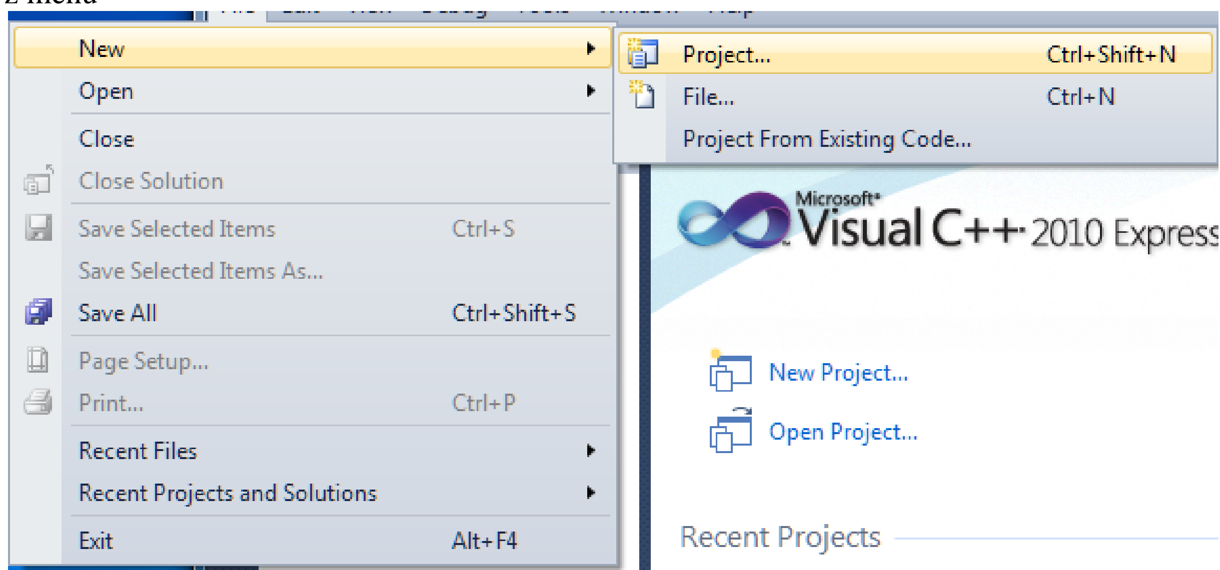
Po ukončené instalací se objeví poslední okno.



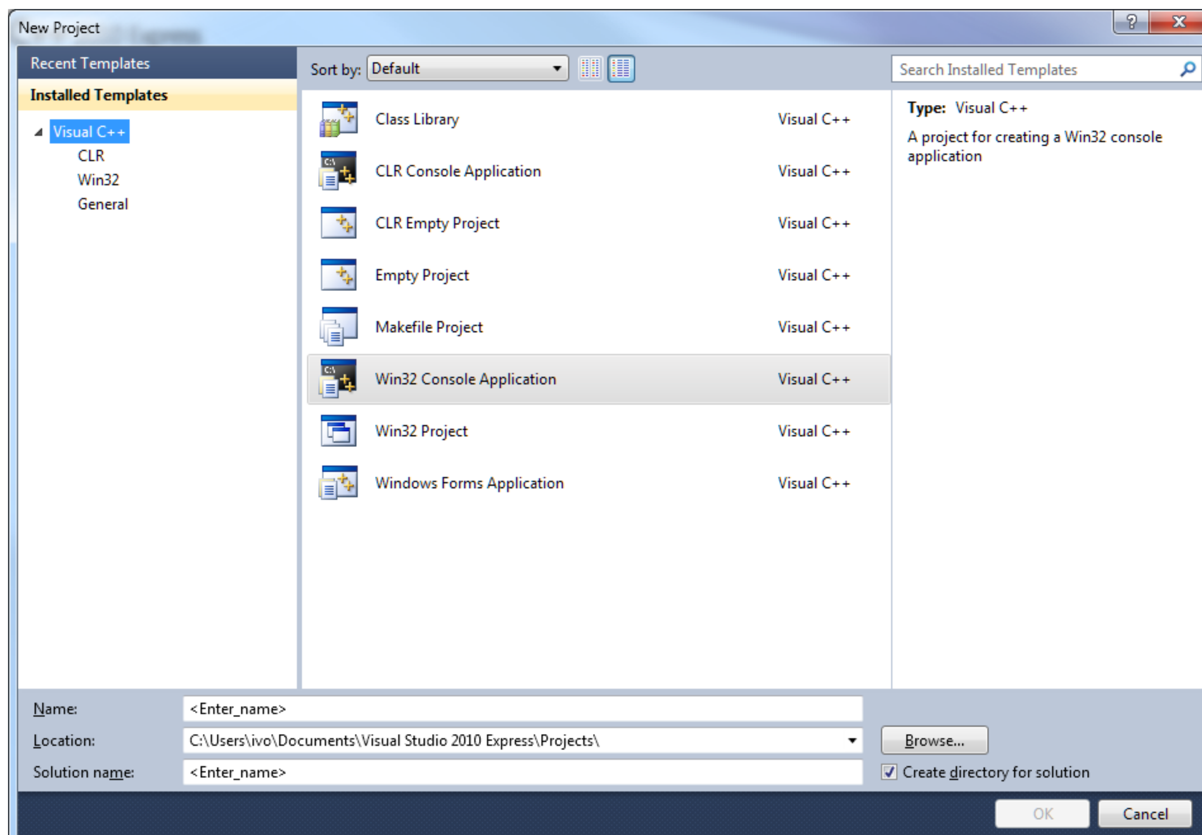
Spuštění vývojového prostředí.
Po spuštění vývojového prostředí se nám otevře okno aplikace Visual C++ Express.



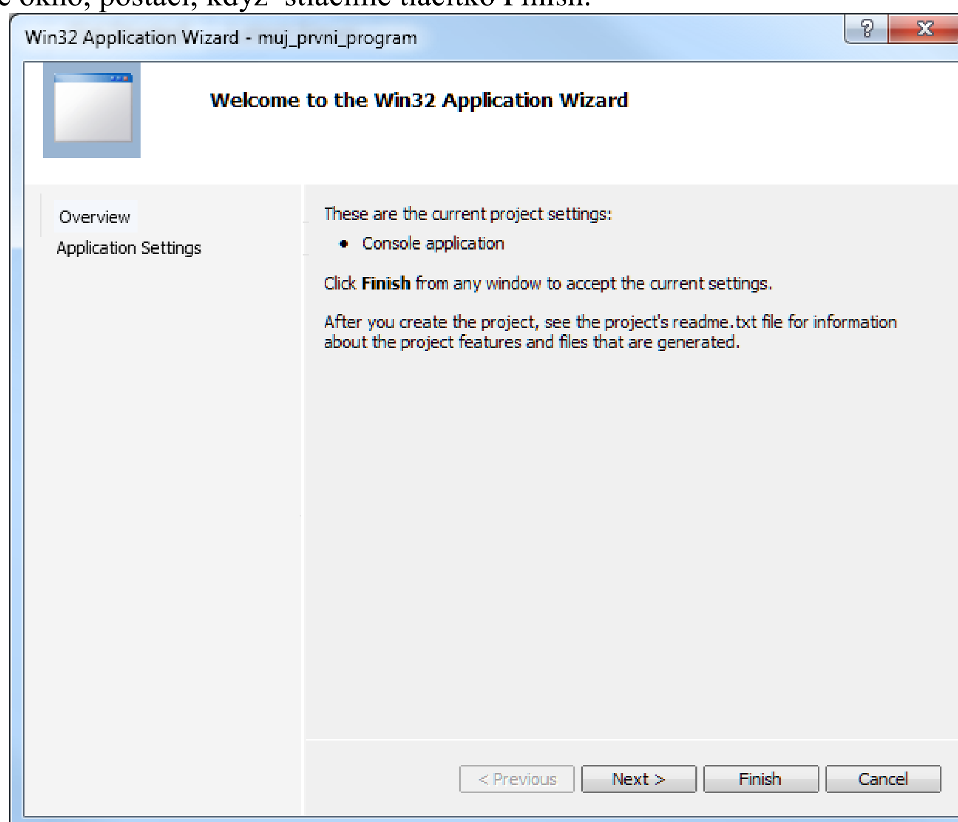
Nejprve se musí vytvořit nový projekt. To lze učinit několika způsoby. Pomocí příkazu NEW z menu



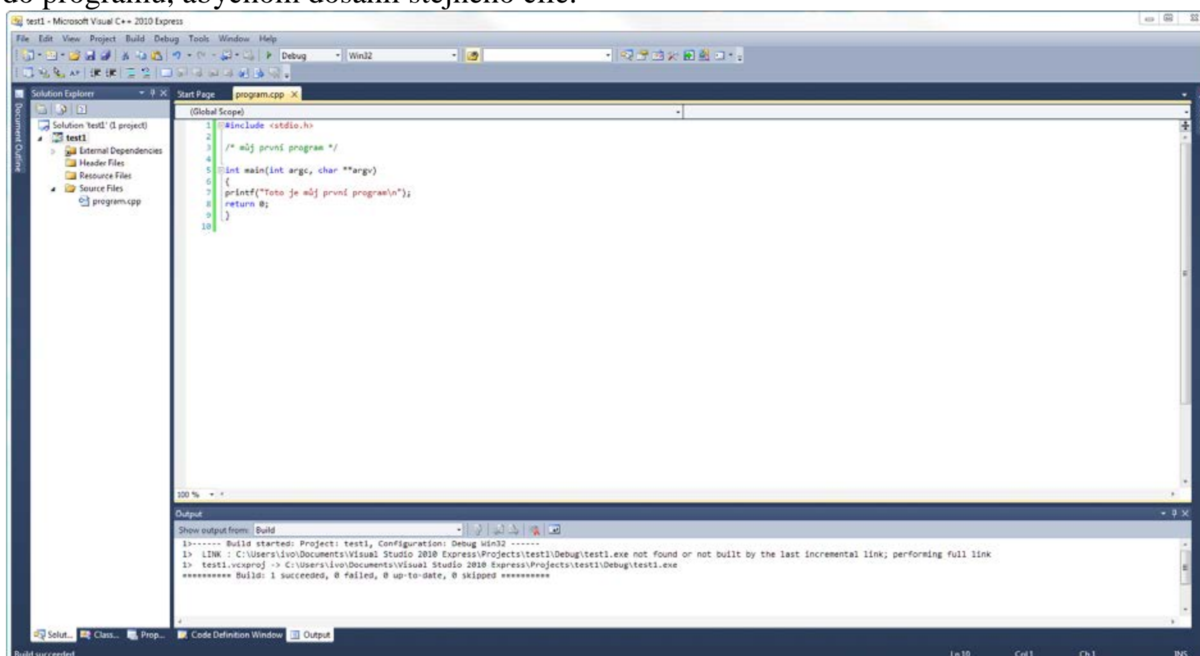
Otevře se nové dialogové okno, ve kterém musíme vybrat šablonu našeho projektu a jeho jméno.



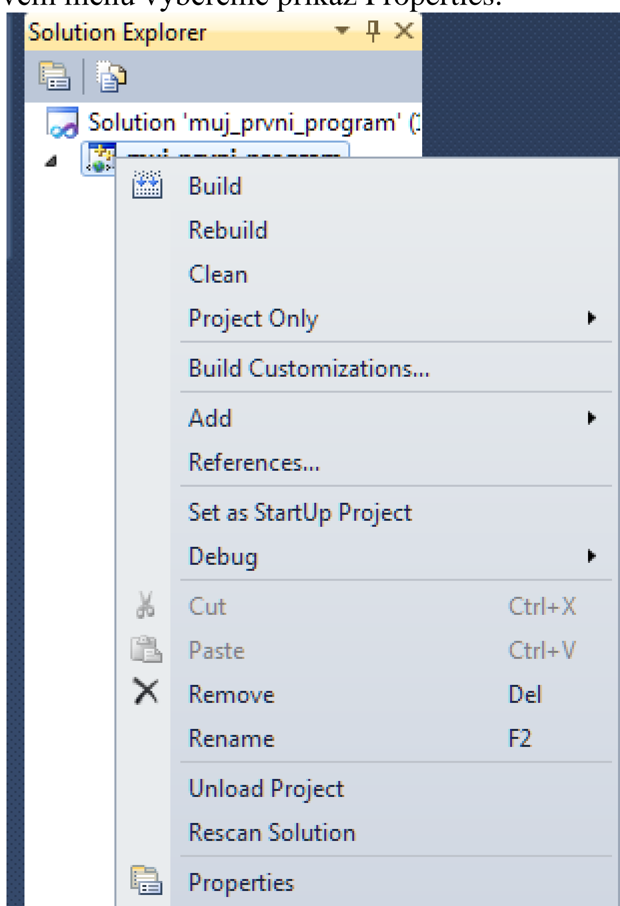
V dalším budeme v tomto kurzu používat konzolovou aplikaci. Do políčka Name napíšeme jméno našeho projektu, například muj_prvni_program. Pokud nechceme měnit standardní cestu ke složce s projekty, necháme nastavenou cestu, jak nám ji nabízí program. Nová cesta by se zadávala do pole Location. Po ukončeném zadání stlačíme tlačítko OK. Otevře se nové dialogové okno, postačí, když stlačíme tlačítko Finish.



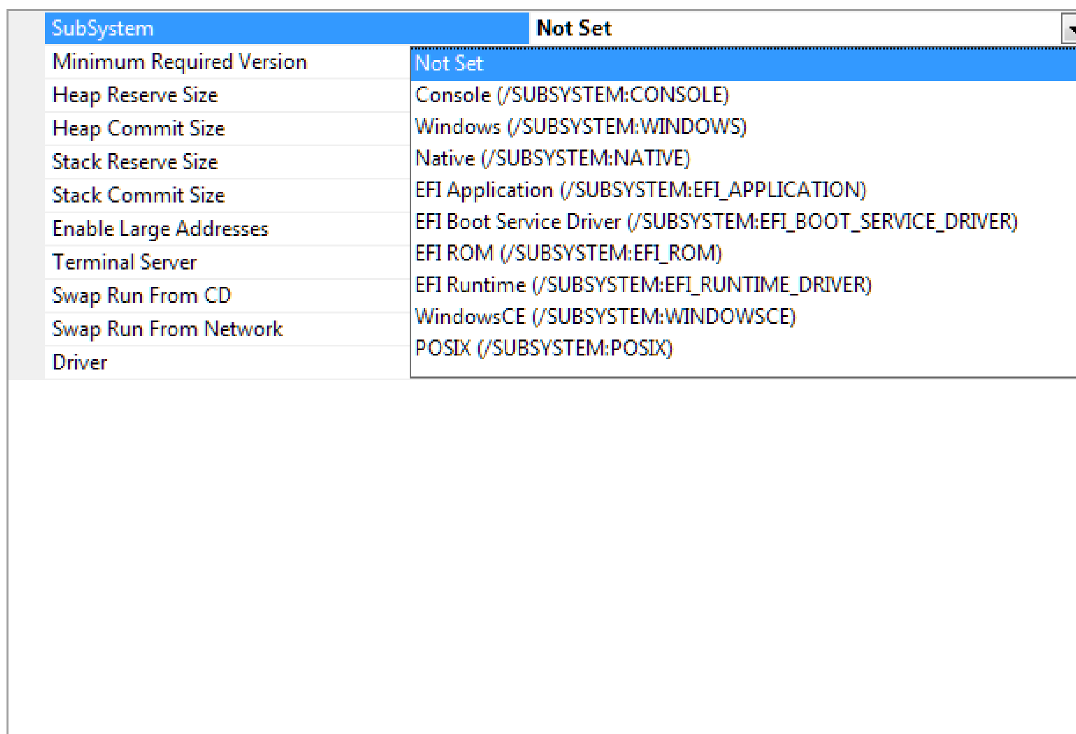
Abychom viděli, co vlastně náš program udělal, potřebujeme ještě nastavit některé vlastnosti našeho projektu. Jinak by nám problikne okno našeho programu, a ještě než bychom stačili cokoli přečíst, zase by se zavřelo. Existují samozřejmě možnosti dopsat vhodné příkazy přímo do programu, abychom dosáhli stejného cíle.



Pravým tlačítkem myši klikneme v okně Solution Explorer na ikonku muj_prvni_program. V otevřeném kontextovém menu vybereme příkaz Properties.

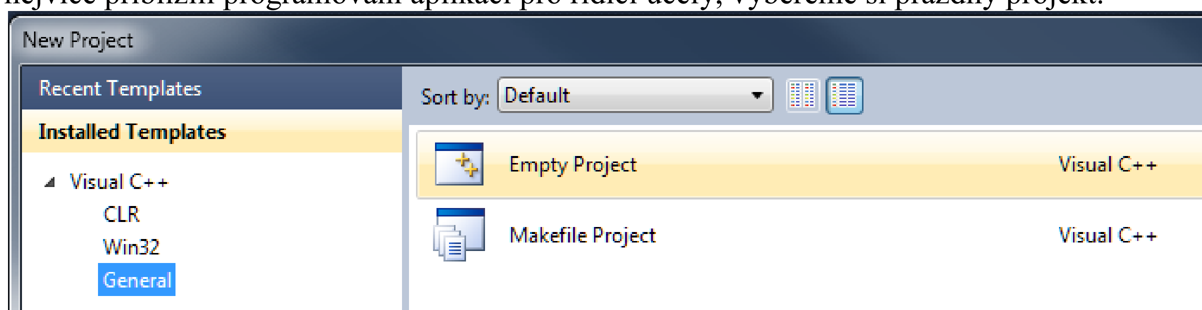


V nově otevřeném okně v prvé části rozbalíme položku Configuration Properties a pak položku Linker. Klikneme na řádek Systém a v prvém řádku v pravé části okna klikneme do řádku vpravo od názvu SubSystem.

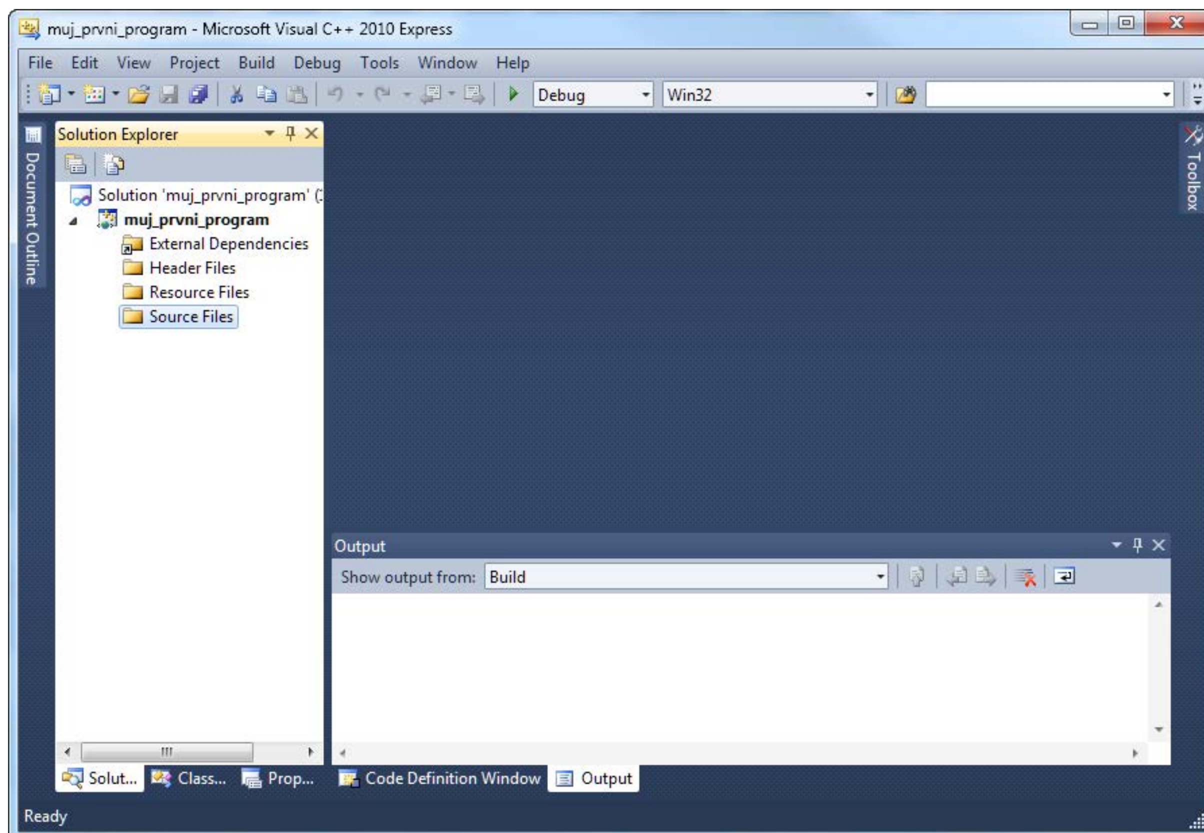


Z rozbalovacího menu vybereme hned druhou položku Console (/SUBSYSTEM:CONSOLE). Volbu musíme potvrdit tlačítkem OK.

Naše programy můžeme tvořit pomocí šablony Win32 Console Application, abychom se co nejvíce přiblížili programování aplikací pro řídicí účely, vybereme si prázdný projekt.

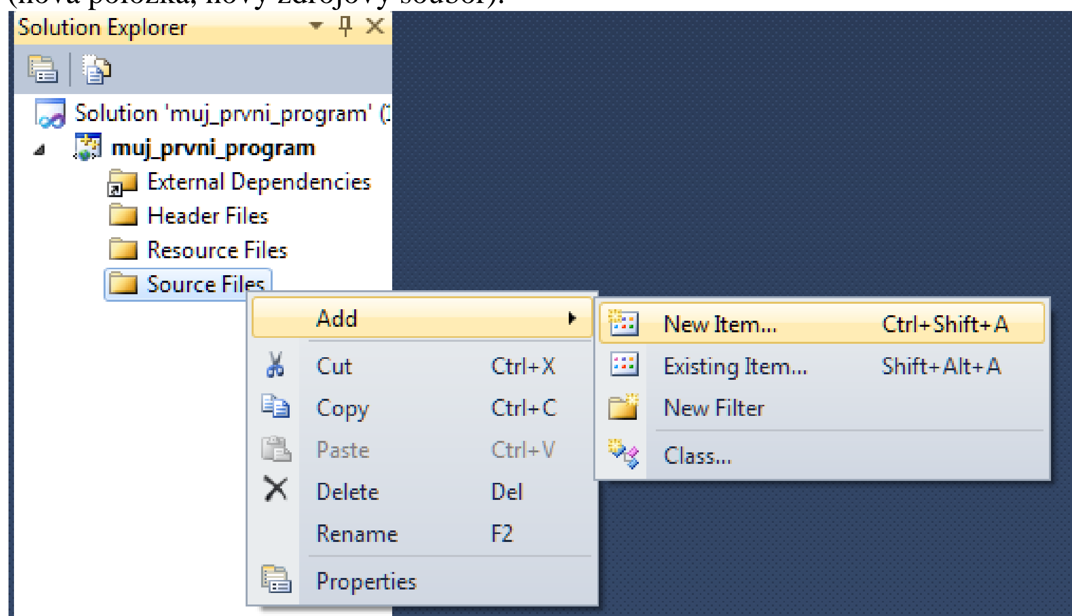


Strukturu prázdného projektu, kterou obdržíme po zadání jména projektu, ukazuje obrázek.

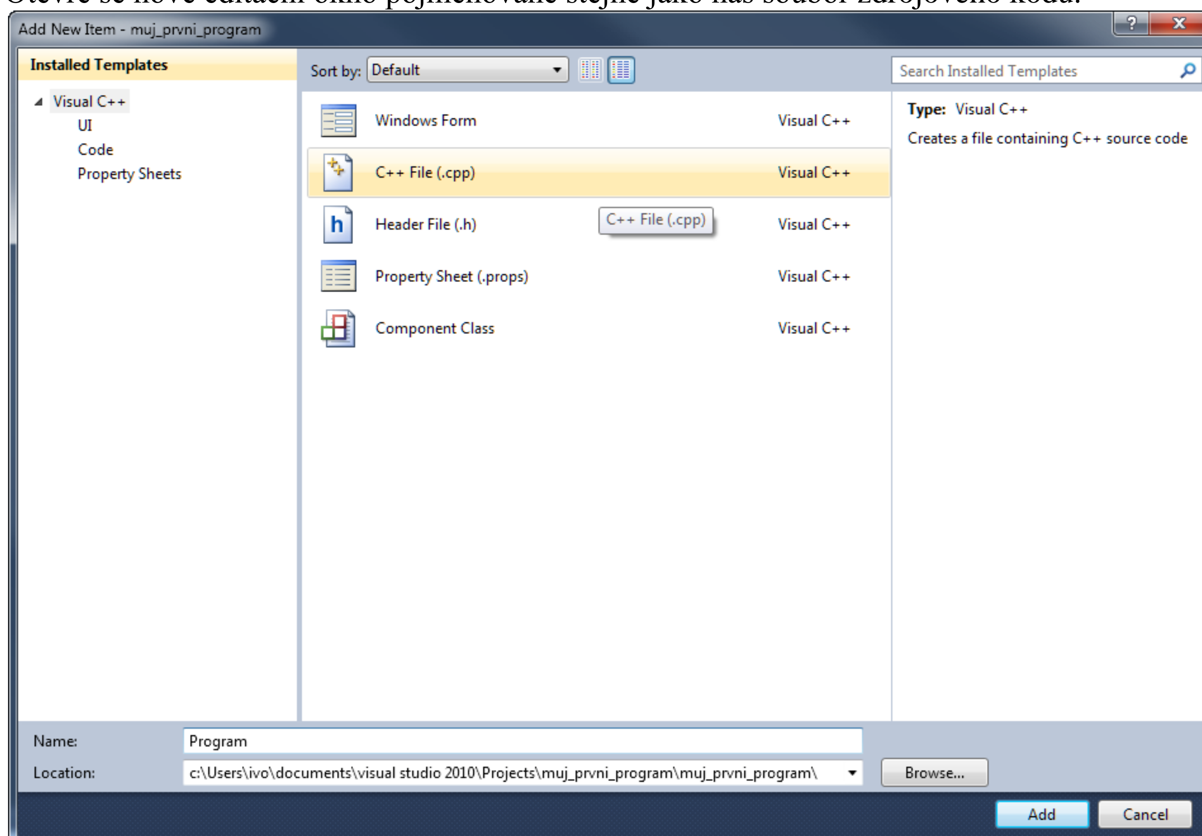


V tomto okamžiku máme zvolenou záložku Solution Explorer, okno výstupu Output. Projekt je tvořen zatím zcela prázdnými složkami External Dependencies, Header Files, Resourced Files a Source Files.

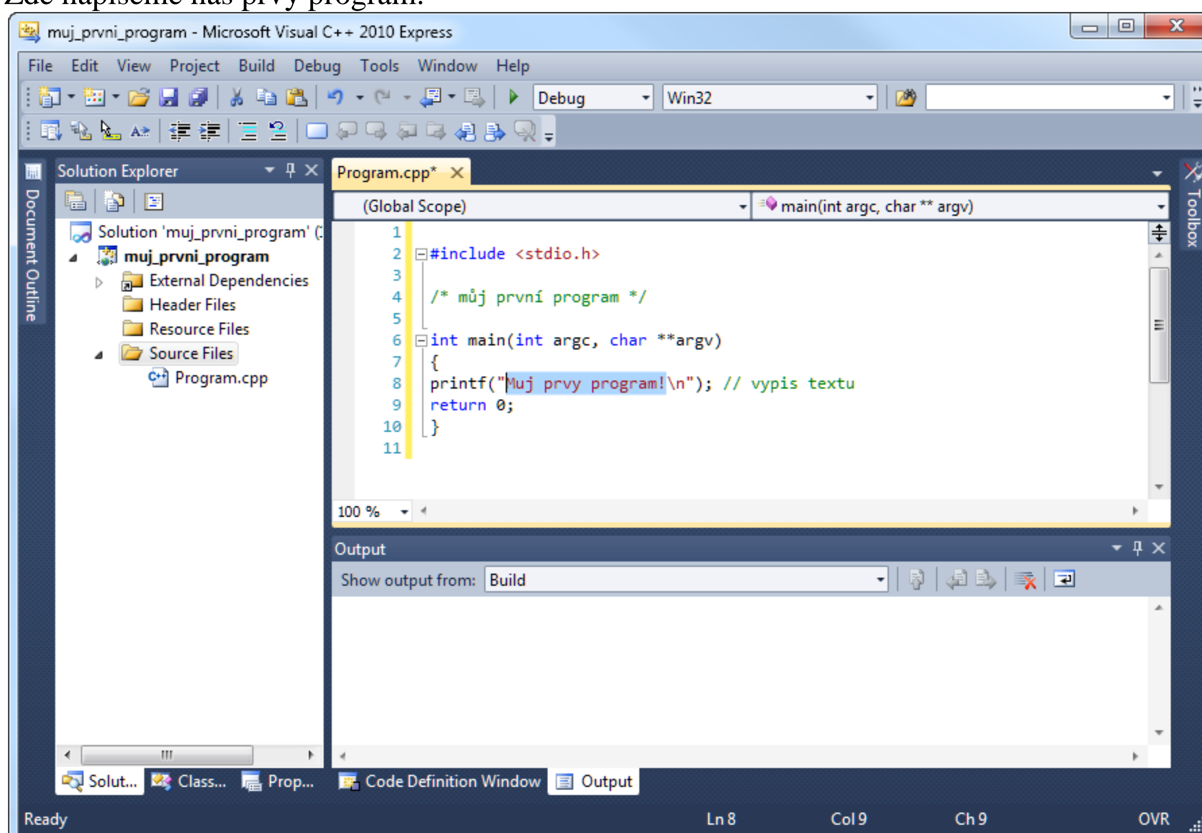
Jak si řekneme dále, každý program v jazyce C má alespoň jeden zdrojový soubor. Přesto, že my budeme psát programy v jazyce C, vývojové prostředí je určeno pro psaní programů v objektovém rozšíření jazyka C, tedy pro jazyk C++, tak přípona našich souborů zdrojových textů bude .CPP místo C. Nejprve tedy musíme vytvořit prázdný zdrojový soubor pro náš první program. To lze například takto: klikneme pravým tlačítkem na ikonku Source Files v okně Solution Explorer, tím se nám otevře kontextové menu a z něj vybereme Add -> New Item (nová položka, nový zdrojový soubor).



Otevře se nám nové dialogové okno, vybereme položku C++ File, v poli Name dopíšeme název zdrojového souboru, zde např. Program. Prázdný zdrojový soubor se nám vytvoří po stlačení tlačítka Ad. Současně se stane součástí našeho projektu.
Otevře se nové editační okno pojmenované stejně jako náš soubor zdrojového kódu.



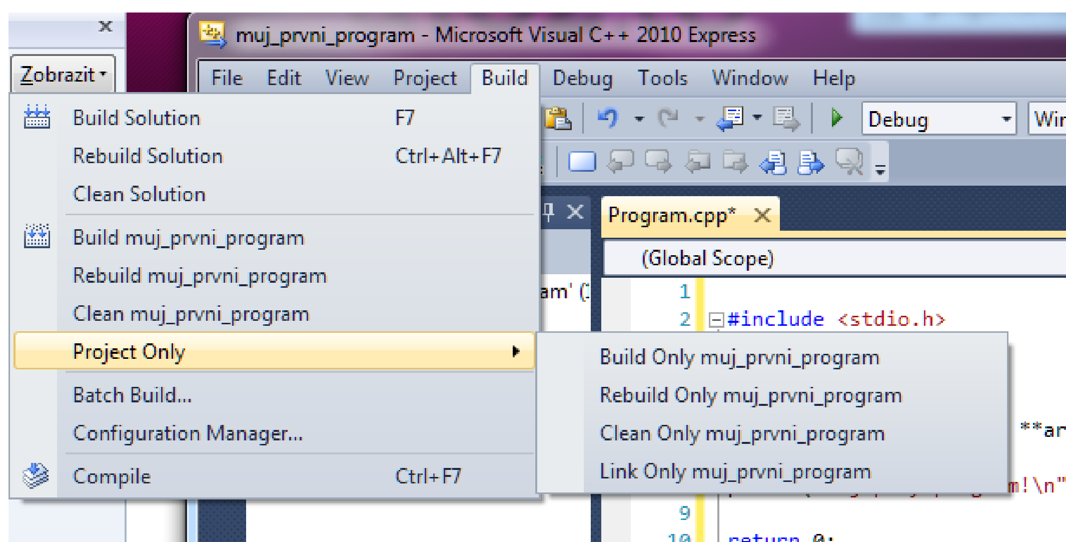
Zde napíšeme náš prvý program.



MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD
CZ.1.07/2.2.00/15.0463

Rozlišuje se velikost písmen (jazyk C je case-sensitive), přičemž většina slov tvořících program (klíčová slova jazyka, datové typy, názvy standardních funkcí a maker) jsou psány malými písmeny.
Ignorují se tzv. bílé znaky (odřádkování, tabulátor, mezery). Doporučuje se využívat těchto znaků pro zvýšení přehlednosti zdrojového kódu programu.
Program se skládá z příkazů - výrazů ukončených středníkem. Pro větší přehlednost programu bývá zvykem psát jednotlivé příkazy na samostatné řádky.

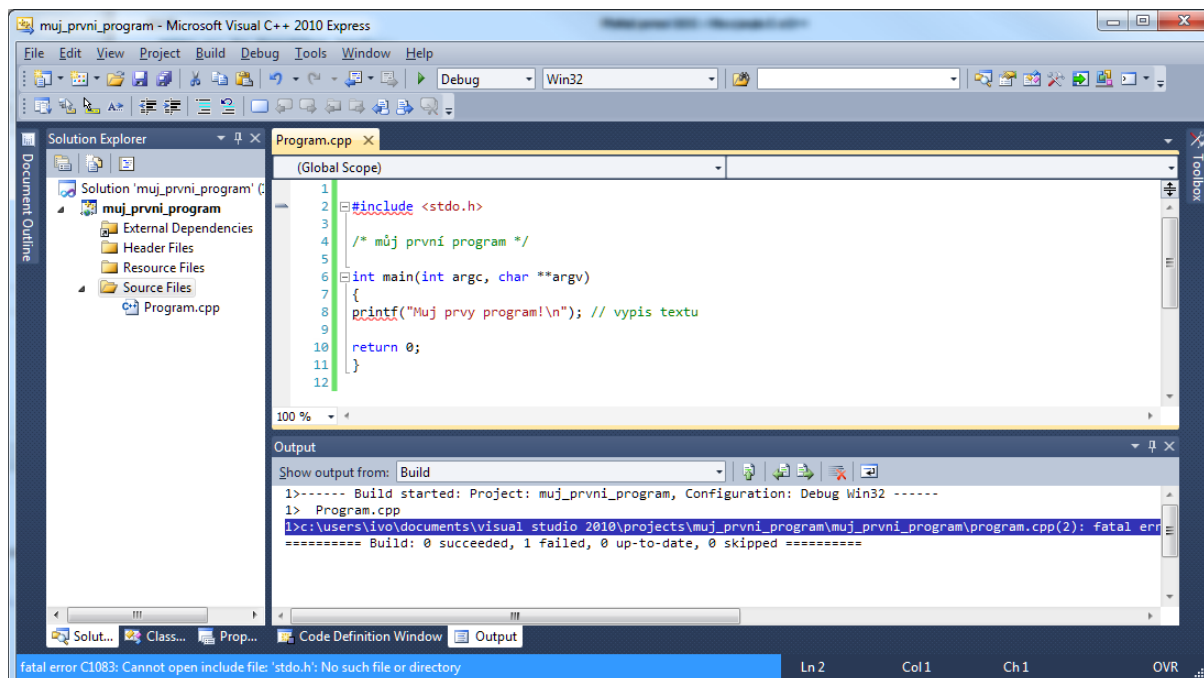
1.8 PŘEKLAD PROGRAMU, SESTAVENÍ PROGRAMU, SPUSTITELNÝ PROGRAM



Příkazem Build Solution (nebo klávesovou zkratkou F7) přeložíme a sestavíme spustitelný program. Příkazem Compile pouze přeložíme aktuálně editovaný soubor zdrojového kódu (.obj). . Pokud je překlad bez chyby, pak jej musíme ještě sestavit (linkovat) s externími funkcemi v knihovnách (.lib) příkazem Link Only. V některých případech je vhodné znovu přeložit a sestavit celý program příkazem Rebuild případně Rebuild Solution.
Před vlastním překladem se spustí preprocesor jazyka C, který řeší rozvoj maker, direktivy (příkazy) preprocesoru. Typickou direktivou je příkaz #include, který do daného místa vloží text souboru, jehož jméno je uvedeno za touto direktivou.

1.9 CHYBY V PŘEKLADU

V případě, že je vše v pořádku a program se přeloží, nevypíše překladač žádnou zprávu. V případě chyby nám překladač sdělí, na které řádce zdrojového kódu je něco špatně a jaká je příčina chyby. Poznat z výpisu příčinu chyby je mnohdy náročné a správné pochopení vyžaduje dobrou znalost jazyka C. Platí pravidlo: vždy opravte nejprve prvou chybu, ostatní chyby mohou být zavlečené, překladač je prvou chybou zmaten a program po opravě prvé chyby může být bezchybně přeložen. Zkontrolujte též jednu řádku nad označeným místem, může obsahovat chybu, kterou ale překladač nerozezná, protože syntaxe příkazu je v pořádku a tato chyba se projeví až na dalším řádku.
Chyby mohou nastat ve všech třech částech překladu programu.
Chyby preprocesoru jsou typicky omezeny na nenalezení připojovaného souboru:



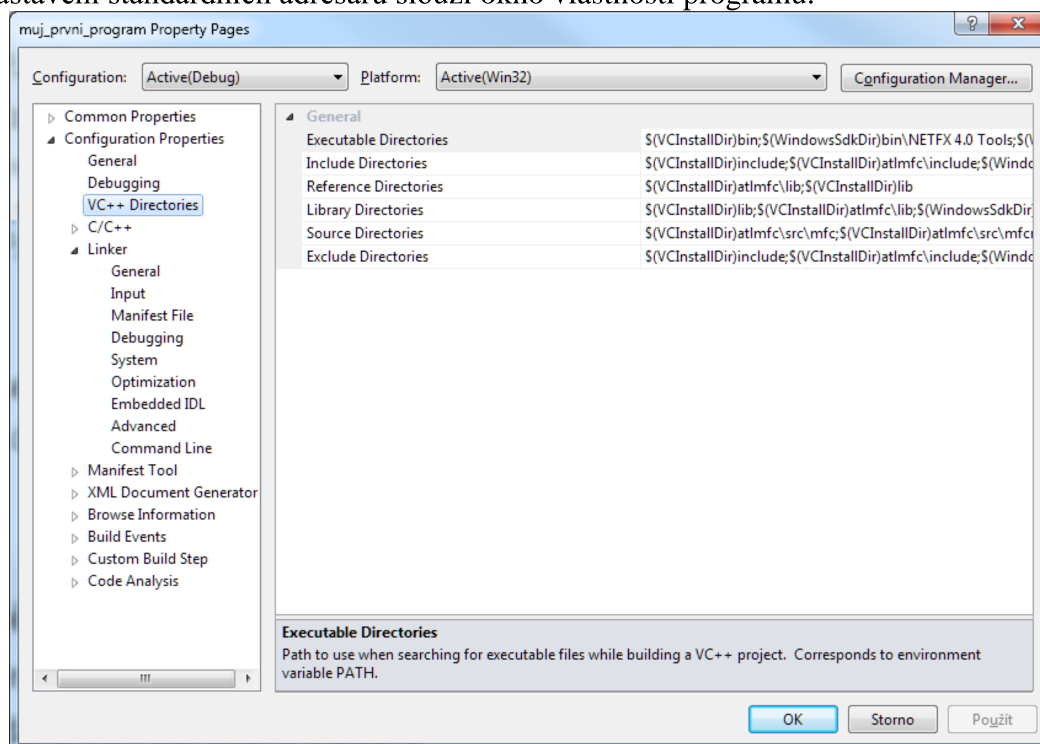
Poklepeme na řádku výpisu prvé chyby v okně output, v editačním okně se nám modrou značkou označí řádek s touto chybou. Zde se jedná o řádek č. 2. Výpis chyby v tomto případě zní:

„1>c:\users\ivo\documents\visualstudio2010\projects\muj_prvni_program\muj_prvni_program\program.cpp(2): fatal error C1083: Cannot open include file: 'stdo.h': No such file or directory“

Program se nesestaví, viz poslední řádek výstupu:

„========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========“

V tomto případě je nutné zkontrolovat jméno připojovaného souboru a opravit překlep. Pokud připojujeme vlastní zdrojové soubory, měli bychom se ujistit, že je překladač dokáže najít. Pro nastavení standardních adresářů slouží okno vlastností programu.



Toto okno vyvoláme buď kliknutím pravého tlačítka myši na ikonu programu v okně Solution Explorer

nebo příkazem Properties z menu Project nebo použitím klávesové zkratky Alt+F7.

Nejčastější jsou asi chyby vzniklé při překladu. Každá taková chyba znamená, že náš program je syntakticky nesprávně napsaný, a že překladač nerozumí kódu, který mu předkládáme. Chybových výpisů existuje nepřeberné množství. V následujícím kódu je syntaktická chyba na řádce šest, ale překladač označí až řádku sedm.

```
#include <stdio.h>
int main (int argc, char *argv[])
{
int a = 3;
int b = 5
printf("a je %d, b je %d\n", a, b);
return 0;
}
```

Na řádce šest totiž chybí středník, na což překladač přijde, až když se pokusí najít pokračování příkazu řádky šest na řádce sedm. Proto označí za místo chyby až řádek sedm.

```
1
2  #include <stdio.h>
3  int main (int argc, char *argv[])
4  {
5  int a = 3;
6  int b = 5
7  printf("a je %d, b je %d\n", a, b);
8  return 0;
9  }
10
```

```
1>c:\users\ivo\documents\visualstudio2010\projects\muj_prvni_program\muj_prvni_program\program.cpp(7): error C2146: syntax error : missing ';' before identifier 'printf'
```

Všimněte si, že editor potrhl červenou vlnovkou klíčové slovo 'printf', tím nám ulehčuje hledání chyb.

Posledním typem chyb jsou chyby při linkování (spojování). Poznat z takové chyby příčinu je mnohdy problém, protože překladač nám neoznačí žádné speciální místo v kódu. V nejvíce případech chyba nastane, když používáme některou z knihoven a zapomeneme ji připojit k programu. Kupříkladu následující kód.

```
#include <stdio.h>
double sqrt(double);
int main(int argc, char **argv)
{
int a = 3.0;
double D = sqrt(a);
printf ("%f\n", D);
return 0;
}
```

se přeloží, ale neslinkuje. Linker nám jen stroze oznámí, že nemůže najít odkaz na sqrt:

```
1>------ Build started: Project: muj_prvni_program, Configuration: Debug Win32 ------
1>Program.obj : error LNK2019: unresolved external symbol "double __cdecl sqrt(double)" (?sqrt@@YANN@Z) referenced in function _main
1>c:\users\ivo\documents\visual studio 2010\Projects\muj_prvni_program\Debug\muj_prvni_program.exe : fatal error LNK1120: 1 unresolved externals
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

1.10 KOSTRA PROGRAMU

Jednoduché programy, které budete v několika prvních cvičeních psát, se budou vždy skládat pouze z tzv. hlavní funkce programu. Veškeré příkazy budete psát do této funkce.

```
#include <stdio.h>
int main()
{
/* Tady bude vlastni posloupnost prikazu. */
return 0;
}
```

1.11 DATOVÉ TYPY

Jazyk C je typový jazyk a definuje tak základní číselné typy různých vlastností. V tomto článku bude vysvětlen pojem definice proměnné a konstanty a dále čtení ze vstupního zařízení do proměnné a výpis proměnné na výstupní zařízení.
Jazyk C je typový jazyk stejně jako drtivá většina ostatních staticky kompilovaných jazyků. Proto jazyk C definuje několik základních číselných typů. Tyto typy se liší rozsahy i typem uchovávaných dat.
Základní datové typy jazyka C můžeme rozdělit na celočíselné datové typy a reálné datové typy. U celočíselných datových typů pak navíc rozlišujeme, zda se jedná o typ znaménkový (pro kladná i záporná čísla) nebo neznaménkový (pouze pro nezáporná čísla). Jazyk C nemá datový typ odpovídající logické hodnotě (pravda nebo nepravda), místo něj je možné použít některý z celočíselných datových typů, přičemž hodnota 0 je brána jako nepravda a jakákoli jiná hodnota jako pravda.

Datový typ	**Popis**
signed/unsigned char	Jde o nejmenší z datových typů. Běžně se používá pro uchovávání znaků. Podle standardu má minimálně 8 bitů.
signed/unsigned short int	Číselný typ o velikosti minimálně 16 bitů.
signed/unsigned int	Číselný typ o velikosti minimálně 16 bitů. Musí být alespoň tak velký jako typ short int.
signed/unsigned long int	Číselný typ o velikosti minimálně 32 bitů. Musí být alespoň tak velký jako typ int.
signed/unsigned long long int	Číselný typ o velikosti minimálně 64 bitů. Tento typ se nachází až ve standardu C99. Musí být alespoň tak velký jako typ long.
float	Číselný typ o velikosti 32 bitů. Uchovává čísla s plovoucí desetinnou tečkou s přesností 5-7 desetinných míst.
double	Číselný typ o velikosti 64 bitů. Uchovává čísla s plovoucí desetinnou tečkou s přesností 15-16 desetinných míst.

Typy short int, long int a long long int mají své kratší ekvivalenty short, long a long long. Je vidět, že standard neurčuje pevně velikost typů a toto rozhodnutí je záleží na cílové platformě. Všechny výše uvedené typy mohou být specifikovány jako signed nebo unsigned (např. signed short int, unsigned char), přičemž signed je implicitní a tudíž se většinou vynechává.

1.12 CELOČÍSELNÉ KONSTANTY

V jazyku C jsou celočíselné konstanty vnitřně reprezentovány implicitně typem int, uvedením znaku "L" (resp. "l") za konstantu lze tento typ změnit na long int.
Uvedením znaku "U" (resp. "u") za konstantu lze změnit vnitřní reprezentaci na unsigned.

Pro zápis konstant v jazyku C je základní číselnou soustavou soustava desítková. Dále je možné využít osmičkový zápis (uvedením znaku "0" na začátku konstanty) a šestnáctkový zápis (uvedením dvojice znaků "0x" nebo "0X" na začátku konstanty). V šestnáctkové soustavě pak kromě číslic "0" až "9" používáme číslice "A" až "F" (resp. "a" až "f").
Znakové konstanty jsou tvořeny libovolným znakem uzavřeným do apostrofů.
Příklady:

desítkový zápis	**10, 1234589, 15u, 1366L, -56, 42LU**
osmičkový zápis	07, 0124, 073
šestnáctkový zápis	0xA1B, 0x0, 0X1d3, 0xac
znakové konstanty	'a', '*', '3', '\'' (pro znak apostrof)

1.13 REÁLNÉ KONSTANTY

V jazyku C jsou reálné konstanty vnitřně reprezentovány implicitně typem double, uvedením znaku "L" (resp. "l") za konstantu lze tento typ změnit na long double a uvedením znaku "F" (resp. "f") za konstantu na typ float. Reálné konstanty je možné psát také v semilogaritmickém tvaru, kde mantisa a exponent jsou odděleny znakem "E" (resp. "e").
Příklady:

standardní zápis	**123.56, 15. , .86, -13.2f, 1.23F, 23.128L**
semilogaritmický tvar	2.1425e-3, 2.1e+4L, 2E-10

1.14 PROMĚNNÉ A KONSTANTY

Definice proměnné v jazyku C povinně obsahuje specifikaci datového typu a identifikátoru proměnné, který je tvořen libovolnou posloupností písmen, číslic a znaku podtržítka. Volitelně je možné uvést také inicializační hodnotu proměnné. Pro větší přehlednost zdrojového kódu obvykle definujeme každou proměnnou novým příkazem, mnohdy je lépe a syntakticky správné zapsat definici více proměnných stejného typu do jediného příkazu.
Proměnné nemohou být shodné s klíčovými slovy.

1.15 KLÍČOVÁ SLOVA

ANSI norma jazyka C určuje následující slova jako klíčová. Tyto slova mají v jazyce C speciální význam a nelze je používat jako uživatelem definované identifikátory (např. jména funkcí, proměnných, konstant atd.). Jejich význam si postupně probereme v kapitole za kapitolou. Zatím se jejich významem nezabývejte.

auto	double	int	struct	break	else	long
switch	case	enum	register	typedef	char	extern
return	union	const	float	short	unsigned	continue
for	signed	void	default	goto	sizeof	volatile
do	if	static	while			

Proměnná v jazyce C definuje zápisem typu před jméno proměnné. Chceme-li tedy vytvořit proměnnou x typu int, provedeme to zápisem.

```
int x;
```

Proměnnou můžeme v jednom kroku také inicializovat na libovolnou hodnotu. To provedeme zápisem

```
int x = 3;
```

což je ekvivalentní zápisu

```
int x;
x = 3; // přiřazení do proměnné
```

Zde jsme proměnnou x nastavili na hodnotu 3. Proměnné v jazyce C se implicitně nenulují! V proměnné bude po jejím vytvoření libovolná hodnota, která je v danou chvíli na daném místě v paměti (tedy v podstatě je náhodná). Proto je vždy před prvním použitím proměnné ve výrazu nutné nastavit počáteční hodnotu proměnné, většinou nulu.
Proměnné v jazyce C mohou být dvou typů, a to lokální a globální. Ukážeme si na jejich rozdíl.

```
const int x = 1;

int main (int argc, char **argv)
{
int x = 3;
x = 4;
//...
}
```

Na řádce jedna vytváříme tzv. konstantu. Konstantu definujeme tak, že před datový typ proměnné napíšeme navíc klíčové slovo const. Konstantu musíme při vytvoření inicializovat, nelze to udělat později. Definovali jsme konstantu x typu int s hodnotou 1. Protože definice této konstanty je mimo tělo jakékoliv funkce (mimo jakékoliv složené závorky v rámci jednoho souboru), bude hodnota této konstanty přístupná všude v celé délce souboru (pokud nedojde k jejímu tzv. zastínění). Na řádce pět definujeme tzv. lokální proměnnou, neboť je umístěna v těle funkce main. Tato definice zastíní globální definici konstanty x. Jinými slovy v těle funkce main se bude pracovat s lokální proměnnou x. Mimo jiné to znamená, že přiřazení hodnoty čtyři proměnné x na řádce šest bude v pořádku. Kdybychom řádek pět s definicí lokální proměnné x vymazali, program se nepřeloží, neboť se budeme pokoušet přiřadit hodnotu konstantě. Další příklady deklarací a definic proměnných.

```
int moje_cislo;
unsigned short int male_kladne_cislo;
char muj_znak = '*'; /* definice s inicializací */
int c1, c2 = 3, c3; /* definice více proměnných */
```

1.16 PŘETYPOVÁNÍ

V jazyce C existují dva typy přetypování, a to explicitní a implicitní. Implicitní přetypování probíhá na pozadí a provádí ho překladač.

- **Implicitní přetypování**

```
double a = 3;
```

Zde jsme do proměnné a typu double přiřadili celočíselnou hodnotu. Překladač celočíselnou hodnotu 3 automaticky převede na hodnotu double 3.0. Při strátě přesnosti při implicitním přetypování překladač vypíše varování.

- **Explicitní přetypování.**

Explicitní přetypování předepisujeme překladači sami. Provedeme to zápisem cílového datového typu v kulatých závorkách před přetypovávanou proměnnou.

```
double a = 3.0;
double b = 2.0;
int c = (int) a % (int) b;
```

V příkladu je použit operátor modulo, který je možné má ba operandy celočíselné. Protože jsme z nějakého důvodu měli čísla hodnoty uložené v proměnných a , b typu double, musíme explicitně přikázat, že je chceme použít jako proměnné typu int. Pokud potřebujeme přetypovat výsledek složitějšího výrazu, musí jej celý uzavřít do kulatých závorek a před levou závorku umístit v závorkách cílový typ.

1.17 VÝPIS POMOCÍ PRINTF()

Proměnnou libovolného typu lze vypsat pomocí již v úvodu zmíněné funkce printf().

```
int x = 3;
printf("promenna x ma hodnotu %d\n", x);
```

Funkce printf má dva parametry. První parametr funkce printf() je řetězec s tzv. maskou. Speciální sekvence uvozená % říká, že na toto místo se ve vypisovaném řetězci se bude vkládat výpis obsahu proměnné typu int (viz níže). Za maskou je pak nutné uvést jako další parametr samotnou proměnnou, která se bude vypisovat. Veškeré ostatní znaky v masce se vypíší beze zmeny. Samozřejmě je možné v jednom volání vypsat libovolné množství hodnot, jen je nutné mít ke každému místu v masce přiřazenou právě jednu proměnnou správného typu.

Příkaz printf z příkladu vypíše na výstup řetězec znaků „promenna x ma hodnotu 3" přejde na nový řádek. Každému typu proměnné odpovídá jiná sekvence. Několik nejpoužívanějších uvedeme.

%c	**Zastupuje jeden znak.**
%s	Zastupuje řetězec znaků.
%d	Zastupuje znaménkový číselný typ (signed int).
%u	Zastupuje neznaménkový číselný typ (unsigned int).
%f	Zastupuje typ s plovoucí desetinnou tečkou (float, double).
%e	Zastupuje typ s plovoucí desetinnou tečkou ve vědeckém zápisu s exponentem (float, double).

1.18 NAČTENÍ HODNOTY

K načtení hodnoty ze vstupu slouží funkce scanf(). Tato funkce opět používá masku jako první parametr.

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

Maska má v této funkci úlohu jakéhosi importního filtru. Hodnota je načtena jen a pouze tehdy, pokud vstup vyhoví masce a na určeném místě je znakově uvedena hodnota správného typu, v tomto případě typu int. Pokud bychom chtěli např. načítat hodnotu vektoru zapsaného jako [3,2]. Pak lze zavolat funkci scanf takto:

```
int a, b;
scanf("[%d, %d]", &a, &b);
```

Všechny bílé znaky před každým zástupným symbolem jsou ignorovány, tedy i vstup [3, 2] by vyhověl masce. Pokud nevyhoví, budou do proměnných načteny nesprávné hodnoty nebo nic. Znak & je u volání této funkce nutný. Důvod bude vysvětlen v kapitole o ukazatelích.

Další příklady vstupu a výstupu:

```
printf("Součet je %d", suma);
printf("Součet je %d", x + y);
printf("Součet je %d\t Součin je %d\n", x + y, x * y);
printf("Plán jsme splnili na 100%%.");
printf("Dekadicky %d je oktalově %o a hexadecimálně %x.\n", cislo, cislo, cislo);
scanf("%d", &cislo);
scanf("%d %o %x", &cislo1, &cislo2, &cislo3);
```

1.19 OPERÁTORY

V tomto článku najdete krátký přehled různých operátorů, se kterými se setkáme v jazyce C.

1.19.1 Matematické operátory

V jazyce C nalezneme celkem pět základních matematických operátorů, které přesně kopírují pět základních matematických operací. Jde o sčítání, odčítání, násobení, dělení a modulo (zbytek po dělení).
Operátory představují symboly:

+	**Operátor sčítání**
++	Operátor inkremetace, který zvýší hodnotu o jedna.
-	Operátor odčítání. Též unární mínus.
--	Operátor dekrementce , který sníží hodnotu o jedna.
*	Operátor násobení.
/	Operátor dělení. Tento operátor zastává funkci jak operátoru celočíselného dělení, tak i funkci dělení desetinných čísel.
%	Operátor modulo celým číslem.

Operátor dělení si probereme podrobněji. Výsledek výrazu obsahující operátor dělení je totiž dán vstupními typy parametrů. Je-li alespoň jeden ze vstupních parametrů neceločíselný, výsledech bude také neceločíselný. V případě, že oba parametry jsou celočíselného typu, bude se výsledek celočíselný, desetinná část se zahodí. V případě, že máme oba typy celočíselné a přesto chceme dělit v oboru reálných čísel, musíme některý z parametrů operátoru přetypovat.

```
int a = 2;
int b = 3;
int c = a/b; // výsledek bude 0!!
int d = (double)a/b; // výsledek bude 1.66666, přetypování
//a na typ double
```

Operátory splňují standardní matematické priority operátorů. Tedy chceme-li např. dělit složitějším výrazem, je nutné výraz uzavřít do kulatých závorek, které mají nejvyšší prioritu (vyčíslují se prvé).

```
float a = 3.0;
float b = 2.0;
int c = a/2*b; // výsledek bude 3, (3/2)*2!!
int d = a/(2*b); // výsledek bude 0.75, 3/(2*2),
// promenna d bude tedy obsahovat 0
```

1.19.2 Bitové operátory

V jazyce C je nepřímo možné pracovat s jednotlivými bity. Slouží k tomu bitové operátory.

&	AND – bitové násobení.
|	OR – bitové sčítání.
^	XOR – bitová non-ekvivalence.
~	bitová negace.
<<	bitový posun vlevo. Posune bity o zadaný počet vlevo a zprava doplní nuly
>>	bitový posun vpravo. Posune bity o zadaný počet vpravo. Zleva doplní nuly, pokud posouváme neznaménkový typ, jinak se kopíruje nejvyšší bit (znaménko).

Bitové posuny jsou velmi výhodné, pokud potřebujeme počítat s mocninami dvou. V každé poziční soustavě je posun vlevo vždy roven násobení základem soustavy, v našem případě dvěma. Posun vpravo pak celočíselnému dělení základem soustavy, zde opět dvěma.

```
int a = 1 << 2; // v a bude binárně 100, tedy a = 49.
int b = 3 << 3; // b binárně 11 se posune doleva na 11000, tedy na b = 24
```

Logický operátor & je vhodný pro určení sudosti nebo lichosti čísla. Ukažme si dvě možnosti:

```
int a = 13;
```

```
if ((a % 2) == 0)
printf("cislo je sude");
else
printf("cislo je liche");
if ((a & 1) == 0)
printf("cislo je sude");
else
printf("cislo je liche");
```

Oba testy se liší tím, jak je počítána podmínka příkazu if. V prvém případě se provede a modulo dvěma a testuje se, zdali je výsledek nula. Ve druhém případě se provede a & jedna, bitové operace probíhají vždy na stejnolehlých bitech, tedy testujeme, zda je nejnižší bit čísla a je jedna, nebo nula, a nakonec zdali je výsledek nula. Druhý způsob se může zdát zbytečně složitý, ale zato je výpočetně mnohem rychlejší než první způsob.

1.19.3 Operátory porovnávání

Jedno z nejčastějších operací v podmíněných výrazech je porovnávání. K porovnávání slouží řada relačních operátorů.

>	**Operátor ostře větší než.**
>=	Operátor větší než nebo rovno.
<	Operátor ostře menší než.
<=	Operátor menší než nebo rovno.
==	Operátor rovnosti. Neplést s operátorem přiřazení, který se píše jen s jedním rovnítkem!
!=	Operátor nerovnosti.

1.19.4 Logické operátory

&&	logické AND. Vrátí pravda pouze, pokud jsou oba operandy pravdivé.
||	logické OR. Vrátí pravda, pokud je alespoň jeden operand pravdivý.
!	logické NOT. Otočí pravdivostní hodnotu vstupního operandu.

U operátoru rovnosti je nutné se mít na pozoru a nezaměnit dvě rovnítka za jedno, kód sice (naneštěstí) bude syntakticky správný, ale výsledek může být zcela odlišný od původně zamýšleného. Obdobně nelze zaměňovat bitové a logické operátory.

1.19.5 Spojení přiřazovacího příkazu a operátoru

Návrh jazyka C je minimalistický a snaží se ušetřit všude, kde jen to jde. Podívejme se na následující kód:

```
int a;
// ...
a = a + 2;
```

proměnné přičteme nějaký výraz a výsledek přiřadíme to té samé proměnné.
Ekvivalentně lze poslední příkaz zapsat takto:

```
int a;
// ...
a += 2;
```

Nejde o nic jiného než o zkrácenou formu výše popsaného příkazu. Tento zápis lze použít pro všechny binární operátory v jazyce C.
Operátory lze rozdělit několika způsoby. Mimo jiné na unární, binární, ternární, neboli na operátory s jedním, dvěma nebo třemi operandy. Operandy jsou data (většinou čísla), se kterými operátory (např. plus) pracují. Operátory se také rozdělují na aritmetické, bitové, relační a logické. Čtěte dále.

1.19.6 Unární operátory

Operátor	Význam
+,-	unární plus a mínus,
&	reference (získání adresy objektu)
*	dereference (získání objektu dle adresy)
!	logická negace
~	bitová negace
++, --	inkrementace a dekrementace hodnoty
(typ)	přetypování
sizeof	operátor pro získání délky objektu nebo typu!

Unární plus a mínus určuje znaménko čísla. Například ve výrazu 6 + (-4) je plus binární operátor (má dva operandy) a mínus je unární operátor (vztahuje se jen ke čtyřce).
Operátory reference & a * dereference vysvětleme v části týkající se práce s ukazateli.
V jazyce C neexistuje datový typ boolean. Pokud potřebujeme někde získat nebo uchovat hodnotu pravda/nepravda (TRUE/FALSE), můžeme k tomu využít např. typ int. V jazyce C je totiž vše nenulové považováno za TRUE a ostatní (včetně např. prázdného řetězce, tj. řetězce, jehož první znak je nulový znak) za FALSE. Nula je tedy FALSE a každé jiné číslo (nejčastěji se používá jednička) TRUE.
Výsledkem logické negace ! je TRUE nebo FALSE, což jazyk C vyhodnocuje jako 1 nebo 0. Například !5 je 0.
Bitová negace ~ představuje něco úplně jiného. Bitová negace neguje jednotlivé bity ve výrazu. Takže například ~0x05 je 0xFA (~00000101 je 11111010).

1.20 PODMÍNKY A CYKLY

Podmínky jsou nezbytné pro ovlivňování chodu programu. Cykly nám usnadní zpracovávání opakujících se událostí bez zbytečné duplikace kódu. V tomto článku se krátce zaměříme na obě možnosti řízení běhu programu.

1.20.1 Podmínky

Obecná plná podmínka má následující strukturu

```
if (<podmínkový výraz>)
<příkaz>
else
<příkaz>
```

Všimněme si kulatých závorek kolem podmínkového výrazu. Ty jsou nutné a nelze je vynechat, ukončují totiž výraz. Zpracování probíhá tak, že program nejdříve vyhodnotí podmínkový výraz a je-li výsledek pravda, provede první příkaz. Pokud podmínka splněná není, provede se druhý příkaz za klíčovým slovem ***else***. Větev s ***else*** lze vynechat, pak mluvíme o neúplné podmínce. Pokud pak není podmínkový výraz splněn, nic se neprovede a program bude pokračovat prvním příkazem za podmínkou. V těle podmínky samozřejmě můžeme mít víc příkazů, pak je musíme uzavřít do bloku pomocí složených závorek {}. Před klíčové slovo ***else*** středník nepíšeme, neboť samotný středník představuje prázdný příkaz. Jinak se středníkem součástí každého příkazu.

```
if (a > 0)
a *= -1; // zde musí být středník
else
a += 2;
```

Ale zde středník nepíšeme.

```
if (a > 0)
{
a *= -1;   // zde musí být středník
}// zde nesmí být středník
else
{
a += 2;
}
```

Výše je uvedena ukázka jednoduché úplné podmínky. Je-li a ostře větší než nula, vynásobí se proměnná a mínus jednou, čímž dojde k otočení znaménka. Jinak se do proměnné a přičtou dvě.

```
if (a < 0)
printf("číslo je záporné\n");
```

V tomto případě se text vypíše jen a pouze tehdy, pokud je hodnota proměnné a ostře menší než nula. V podmínkovém výrazu můžeme využít řadu porovnávacích a logických operátorů. Je třeba dát pozor a neplést si logické a bitové operátory, logické operátory se zapisují zdvojeně!

Podmínky lze vnořovat. Struktura programu pak bude následující.

```
if (podminka1)
telo bloku
else if (podminka2)
telo bloku
else if (podminka3)
telo bloku
else
telo bloku
```

Jazyk C je velmi tvárný, pokud jde o vyhodnocování podmínkových výrazů. Podívejme se na následující ukázky:

```
int a = 1;
if (a) //...
if (a == 1) //..
if (a = 1) //..
```

V sekci proměnných jsme se zmiňovali o implicitním přetypování. To se použije i v případě logických výrazů. Jakákoliv nenulová hodnota je vyhodnocena jako pravda. Jakákoliv forma nuly je vyhodnocena jako nepravda. Jak jsme již uvedli v části týkající se operátorů, jazyk C nemá žádný speciální pravdivostní typ bool (ten zavádí až norma C99 a C++), vystačit si tak musíme s typy char a int, které se pro uchování logických hodnot nejčastěji používají. První podmínka if (a) tak vyhodnotí jako pravda a první část podmíněného příkazu se provede. Ve druhém případě se ptáme, zdali je a rovno jedné. To je splněno, a tudíž se první část podmíněného příkazu provede. Co ale znamená poslední podmínka if (a = 1)? Zde totiž ve výrazu přiřazujeme do proměnné a hodnotu jedna. Výsledek výrazu přiřazení je přiřazovaná hodnota, a tak je podmínka vyhodnocena jako jedna, tedy vždy jako pravda!! Zde je vidět, jak je nutno dbát na to, zda při porovnání opravdu operátor zapíšeme jako dvě rovnítka.

Jazyk C provádí tzv. zkrácené vyhodnocování výrazů. V praxi to znamená, že podmínka je vyhodnocována zleva doprava a jakmile je jasné, jaké hodnoty podmínka nabude, vyhodnocování je ukončeno.

```
if ((a == 1) && vypocetFunkce())
//...
```

V kódu výše je v podmínce uveden složený výraz. Pokud je hodnota proměnné a jiná než jedna, k volání funkce ***vypocetFunkce ()*** vůbec nedojde.

1.20.2 Vícenásobné větvení

Programovací jazyk C umožňuje přehledně zapsat podmínku, která má více možných stavů v závislosti na hodnotě výrazu. Jedná se o příkaz ***switch*** s následující syntaxí:

```
switch (<podmínkový výraz>)
{
case <hodnota>:<příkaz>
...
<[default:<příkaz>]>
}
```

Podmínkový výraz se musí vyhodnotit jako celočíselná hodnota (tedy i znak). Porovnávání jiných typů není možné. Příkaz ***switch*** funguje tak, že se nejprve vyhodnotí podmínkový výraz a jeho výsledek se pak hledá odshora v hodnotách jednotlivých ***case*** větvích. <hodnota> musí výt konstantní výraz. Jakmile se najde první shoda, vykoná se příkaz za dvojtečkou. Pak pokračuje ve vykonávání následujících větví, interně se totiž při shodě hodnot skočí na odpovídající místo v kódu a vše od tohoto místa se vykonává dál. Pokud chceme vykonat právě jednu větev (což je obvyklé), je nutné příkaz na konci větve ukončit příkazem ***break***. Nenajde-li se shoda s žádnou hodnotou, provede se nepovinná větev default. Pokud není větev default v příkazu ***switch*** přítomna, neprovede se nic.

```
#include <stdio.h>
int main(void)
{
char znak;
printf("Opravdu chcete smazat vsechna data na disku? [a/n/k]> ");
znak = (char) getchar();
switch (znak) {
default:
printf("Mel jsi zmacknout \'a\',\'n\' nebo \'k\' ne \'%c\'\n",
znak);
case 'k':
return 0;
case 'N':
printf("Stejne smazu co muzu!\n");
break;
case 'n':
printf("Nechcete? Smula!\n");
case 'a':
case 'A':
printf("Data byla smazana !!!\n");
break;
}
printf("Ne, nebojte se, to byl jenom zertik.\n");
return 0;
}
```

V tomto příkazu ***switch*** se rozhodujeme na základě hodnoty proměnné znak. Je-li hodnota proměnné ***'k'***, program končí. Je-li ***'N'*** vypíše text ***"Stejne smazu co muzu!\n"***, je-li ***'n'*** vypíše ***"Nechcete? Smula!\n"***, je-li ***'a'*** nebo ***'A'*** vypíše ***"Data byla smazana !!!\n"***. Pokud byl zadán jiný znak, provede se příkaz v sekci ***default***. Protože ten nekončí příkazem ***break***, provede se i následující příkaz jako v případě, když byla hodnota proměnné znak ***'k'*** tedy příkaz ***return*** a program končí. Jinak program bude pokračovat za příkazem ***switch*** a vypíše text ***"Ne, nebojte se, to byl jenom zertik.\n"***.

1.20.3 Ternární výraz

V jazyce C existuje právě jeden ternární výraz. Hodí se pro zkrácené zapsání jednodušší podmínky.

```
<podmínka>?<příkaz>:<příkaz>
```

Při zpracování příkazu se nejprve vyhodnotí podmínka a je-li výsledek pravda, vykoná se první příkaz před dvojtečkou. Je-li výsledek nepravda, vykoná se příkaz za dvojtečkou. Klasickou ukázkou je implementace funkce ***abs().***

```
(x >= 0) ? x : -x;
```

1.20.4 Cykly

V jazyce C máme celkem tři typy cyklů. Jsou to cykly ***for, while a do-while***.

Cykly ***while a do-while***

Jedná se o tzv. indukční typy cyklů. Zde je vyhodnocena podmínka provádění cyklu a to buď na začátku, nebo na konci. Připomeňme si pravidlo, že podmínka musí být v případě cyklu while nastavena ještě před zahájením cyklu. Znamená to, že proměnné ve výrazu tvořící podmínku (řídicí proměnné cyklu) musí mít přiřazeny smysluplné hodnoty. V průběhu cyklu se musí hodnota řídicích proměnných měnit, jinak bude příkaz vykonáván neustále a program se tzv. zacyklí.

Cykly while a do-while jsou si velmi podobné a liší se jen prvním průběhem. U while se podmínkový výraz vyhodnotí před prvním provedením těla cyklu (jedná se o cyklus podmínku na začátku), zatímco u do-while se tělo provede vždy alespoň jednou (cyklus podmínkou na konci).

```
while (<podmínkový výraz>)
<příkaz>
do
<příkaz>
while (<podmínkový výraz>)
```

Stejně jako u podmínky i zde můžeme do cyklu vložit více příkazů, které i zde musíme umístit do bloku. U obou variant se cyklus provádí do té doby, dokud je výraz vyhodnocen jako pravda (pomůcka: ***while*** znamená česky dokud).

Cyklus For

Cyklus for je z celé skupiny. Příkaz má následující strukturu:

```
for (<[inicializace]>;<[podmínkový výraz]>;<[iterace]>)
<příkaz>
```

Všechny části uvnitř kulatých závorek jsou nepovinné. První část ***<[inicializace]>*** se provede před prvním provedením cyklu a zde můžeme nastavit hodnotu proměnných. Druhá část je ***<[podmínkový výraz]>***, který jako u ostatních typů cyklů řídí ukončení cyklu. Pokud ho vynecháme, poběží cyklus donekonečna. Poslední část se provádí na závěr každého cyklu. Část ***<[iterace]>*** lze použít pro zvýšení hodnoty řídcí proměnné nebo jakékoliv nastavení, které je nutné provést na konci každého průchodu cyklem. Všechny tři čísti mohou obsahovat seznam příkazů oddělených čárkami.

```
int i, j, k;
for (i = 0, k= 4; i < 10, j > 4; i++, k++)
{
//...
}
```

Klasická forma cyklu for vypadá takto.

```
for (i = 0; i < 10; i++)
{
//...
}
```

Tento cyklus provede celkem deset iterací. Řídící proměnná musí existovat před voláním cyklu, její definici v těle cyklu lze provést až v C99 nebo v C++. Cyklem for lze ale také simulovat chování cyklu while pouhým vynecháním první a poslední části:

```
int i;
// ...
for (;i != 10;)
{
//...
}
```

kde cyklus poběží, dokud bude proměnná i různá od deseti. Zde ale musíme hodnotu proměnné i v těle cyklu měnit sami.

1.20.5 Break a Continue

V jazyce C existují dva příkazy pro ovlivnění vykonávání cyklů. Je to break a continue. Příkaz break ukončí okamžitě vykonávání cyklu a program následuje prvním příkazem za cyklem. continue okamžitě ukončí aktuální iteraci a započne novou. Oba příkazy lze volat pouze uvnitř těla cyklu (příkaz break lze umístit i do příkazu switch).

1.20.6 Řetězce v jazyce C

S řetězci se v jazyce C pracuje jako s poli. Máme mnoho funkcí pro práci s řetězci. Tento článek přináší
krátký úvod do práce s řetězci.

1.20.7 Řetězec jako literál

LS literály jsme se setkali v kapitole týkající se číslic a znaků. Jedná se tedy o nepojmenované konstanty v programu. Řetězec jako literál zapíšeme ve tvaru

```
"jakykoli retezec znaku".
```

Tento řetězec obsahuje kódy jednotlivých znaků a navíc je ukončen znakem ´\0´ tedy byte s hodnotou nula. Jak s takovým řetězcem pracovat si ukážeme dále. Pokud s konstantním řetězcem chceme dále pracovat, můžeme si jeho adresu uložit do proměnné typu ukazatel, podrobněji v kapitole o ukazatelích.

```
char * ret = "jakykoli retezec znaku";
```

1.20.8 Escape sekvence

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

Escape sekvence		
Escapeznak	**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 tabelační 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

1.21 INICIALIZACE ŘETĚZCE

Řetězce jsou v jazyce C reprezentovány jako pole znaků. Jazyk C nemá žádný speciální typ pro uchování řetězce na rozdíl od Pascalu, Javy, C++ a dalších. Přesto nám jazyk C umožňuje s řetězcem normálně pracovat. Proměnnou „uchovávající" řetězec se vytvoří takto:

```
char jmeno[<delka>];
char jmeno[<[delka]>] = "<řetězec>";
char jmeno[<delka>] = {'a', 'h', 'o', 'j', '\0'};
```

Jak dále uvedeme v kapitole o polích, jedná se proměnné pole typu char. První řádka vytvoří pole jmeno a vyhradí prostor pro udaný počet znaků. Ostatní uvedené postupy vyplní místo předaným řetězcem. Délka řetězce je nepovinný údaj, pokud ale pole definujeme, tedy přiřazujeme jeho prvkům hodnoty znaků. Nelze tedy napsat

```
char jmeno[];
```

Pokud délku pole vynecháme, překladač spočítá znaky v uvozovkách a vytvoří řetězec o jedna větší. Důvod, proč je řetězec větší, je to, že v jazyce C se všechny řetězce ukončují hodnotou nula (binární nula je znak '\0', neplést se znakem '0'!!). Proto se také občas řetězce v jazyce C označují jako nulou ukončené řetězce.

Je jasné, že když chceme zjistit délku řetězce, musíme projít celý řetězec a počítat znaky, dokud nenarazíme na bajt s hodnotou nula (samozřejmě existuje funkce, která toto udělá za nás, viz níže). To je vcelku pomalé, zato tento způsob uchování umožňuje vytvářet libovolně dlouhé řetězce. Při vytváření řetězce nikdy nesmíme zapomínat na to, že řetězec musí být efektivně větší o jeden znak právě kvůli zarážce.

```
char retezec1[5] = "ahoj";
char retezec2[] = "ahoj";
char retezec3[5] = {'a', 'h', 'o', 'j', '\0'};
char retezec4[3] = "ahoj";
```

V příkladu výše budou první tři řetězce stejné, neboť jsme je vytvořili ekvivalentními metodami. Jinak je tomu u řetězce retezec4? Zde pole znaků retezec4 o délce tři znaky inicializujeme řetězcem o délce pět. V tomto případě se inicializační hodnota ořízne na řetězec "aho" pole znaků nebude obsahovat zakončující znak *'\0'* a řetězec nebude správně ukončen!. Pokud se takový řetězec pokusíme vypsat nebo s ním jinak pracovat, budeme zasahovat i do paměti, která se nachází za vlastním obsahem, a program se může chovat nedefinovaně.

1.22 PRÁCE S ŘETĚZCI

Když už umíme vytvořit řetězec, chtěli bychom ho také umět používat. V následujícím shrnutí uvádíme nejpoužívanější funkce pro práci s řetězci. Všechny se nacházejí v hlavičkovém souboru ***string.h.***

1.22.1 Výčet funkcí

Vzhledem ke zjednodušení popisu se nebudeme vyjadřovat zcela exaktně a například místo „řetězec, na který ukazuje ptr" budu psát „řetězec ptr". Typ size_t je ekvivalentní typu unsigned int.

Důležité upozornění

Pokud funkce někam kopírují / přidávají data, nekontrolují, zda na to mají dostatek místa. To musíte zajistit vy, jako programátoři. Jinak se může snadno stát, že program skončí kvůli „neoprávněnému přístupu do paměti".

- Funkce kopírování:

Funkce	Popis
memcpy	Kopíruje blok paměti (funkce)
memmove	Přesunuje bloku paměti (funkce)
strcpy	Kopíruje řetězec (funkce) char ***strcpy**(char *dest, const char *src); Zkopíruje řetězec src do řetězce dest. Vrací ukazatel na dest.
strncpy	Kopíruje znaky z řetězce (funkce) char ***strncpy**(char *dest, const char *src, size_t n); Jako strcpy(), ale zkopíruje maximálně **n** znaků. (Je-li jich právně **n**, nepřidá zarážku)
strdup	Funkce vrátí kopii zadaného řetězce. char* strdup(const char *s); Pozor, takto vytvořenou kopii je později nutné uvolnit voláním funkce free() (více informací v sekci o dynamické alokaci).

- Funkce zřetězení:

Funkce	Popis
strcat	**Spojí řetězce (funkce)** **char *strcat(char *dest, const char *src);** **Funkce připojí řetězec src k řetězci dest. Funkce vrací ukazatel na řetězec dest.**
strncat	Připojí znaky z řetězce (funkce) char ***strncat**(char *dest, const char *src, size_t n); Jako funkce strcat, ale přidá jen **n** znaků z src. Tato funkce je bezpečnější z hlediska možného "přetečení" (pokus zapisovat více dat, než jakou má velikost dest).

- Funkce srovnání:

Funkce	Popis
memcmp	Porovnání dvou bloků paměti (funkce)
strcmp	Porovnání dvou řetězců (funkce) nt **strcmp**(const char *s1, const char *s2); Porovnává řetězce s1 a s2. Pokud je s1 < s2 pak vrací hodnotu menší než 0, pokud jsou si rovny, pak vrací 0, pokud je s1 > s2 pak vrací hodnotu větší jak 0.
strcoll	Porovnání dvou řetězců pomocí locale (funkce)
strncmp	Porovnání znaků dvou řetězců (funkce) nt **strncmp**(const char *s1, const char *s2, size_t n); Jako strcmp(), porovnává však jenom n znaků.
strxfrm	Transformace řetězec pomocí locale (funkce)

- Funkce vyhledávání:

Funkce	Popis
memchr	Vyhledá znak v bloku paměti (funkce)
strchr	Vyhledá první výskyt znaku v řetězci (funkce) char ***strchr**(const char *s, int c); Vrací ukazatel na první pozici, kde se vyskytuje znak **c**, nebo NULL v případě, že jej nenajde. (Nenechte se zmást, že znak c je typu int a ne char, má to tak být).
strcspn	Vrátí počet shodných znaků v řetězci (funkce)
strpbrk	Vyhledá znaky v řetězci (funkce)
strrchr	Vyhledá poslední výskyt znaku v řetězci (funkce)

	char ***strrchr**(const char *s, int c); To samé jako strchr(), ale hledá první výskyt zprava.
strspn	Získá počet znaků ve kterých jsou řetězce shodné (funkce) size_t **strspn**(const char *s, const char *accept); Vrací počet znaků, ve kterých se řetězec s schoduje s řetězcem accept.
strstr	Vyhledá řetezec (funkce) char ***strstr**(const char *haystack, const char *needle); Vrací ukazatel na první výskyt řetězce needle v řetězci haystack. Pokud jej nenajde, vrací NULL.
strtok	Rozdělit řetězec na tokeny (funkce)

- Ostatní funkce:

Funkce	**Popis**
memset	Vyplní blok paměti (funkce)
strerror	Získá ukazatel na řetězec chybových hlášení (funkce)
strlen	Získá délku řetězce (funkce) size_t **strlen**(const char *s); Vrací délku řetězce s.

- Makra

Makro	**Popis**
NULL	Null pointer (makro)

- Typy

Typ	**Popis**
size_t	ekvivalent unsigned int

Následující funkce se nacházejí v hlavičkovém souboru stdio.h.

getchar	**int getchar()** **Vrátí nejvýše jeden znak ze standartního vstupu jako hodnotu typu na int. V případě, že nastane chyba nebo že už není co načíst, vrátí funkce konstantu EOF.**
putchar	void (int znak) Zapíše na standartní výstup právě jeden znak z proměnné typu int.
gets	char* gets(char *ret) Načte jednu řádku ze standartnímu vstupu do řetězce ret. Skončí, pokud narazí na konec řádky nebo na konec vstupu. V obou případech na konec přidá ukončovací nulu. Není prováděna kontrola přetečení a je na volajícím, aby vyhradil dostatečný prostor načítaný řetězec.
puts	char* puts(const char *ret) Zapíše do standartního výstupu předaný řetězec bez ukončovací nuly.
sprintf	int sprintf(char *ret, const char *maska, …) Stejně jako u funkce printf()popsané výše provede funkce formátovací operace a výsledek namísto standardního výstupu uloží do řetězce. Je na volajícím, aby vytvořil dostatek místa pro uložení.

1.22.2 Krátký program

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

int main(void)
{
char text[80];
printf("Zadejte libovolny text: ");
```

```
/* puts() by nam na konec vypsalo znak nového radku, a to nechceme */
(void) gets(text);
puts(text);
printf("Zadejte libovolny text: ");
scanf("%s", text);
puts(text);
return 0;
}
```

1.22.3 Porovnávání řetězců

Řetězcové proměnné nelze porovnávat jako ostatní proměnné pomocí operátoru ==. Důvodem je, že řetězce jsou pouze převlečená pole. Pokud chceme porovnat dva řetězce, provedeme to pomocí výše uvedené funkce strcmp.

```
char ret[] = "ahoj";
//...
if (ret == "ahoj") // chyba!, nelze porovnávat tímto způsobem
if (strcmp(ret, "ahoj") == 0)
// nesmíme zapomínat, že jsou-li řetězce stejné, vrací funkce nulu
```

1.22.4 Procházení řetězců

Řetězce se dají procházet znak po znaku jako každé jiné pole. Pokud chceme přepsat n-tý znak v řetězci,
provedeme to takto:

```
char ret[] = "ahoj";
ret[1] = 'c'; // v ret bude ted ahoj
```

V jazyce C pole indexujeme od nuly, proto znak a má index nula a ne jedna.

1.23 FUNKCE

Funkce jsou stavebním kamenem jazyka C. V následujícím textu probereme základní vlastnosti funkcí.

1.23.1 Deklarace a definice funkce

V definici funkce musíme určit, jak se bude funkce chovat navenek. Musíme určit jméno funkce, vstupní parametry a návratovou hodnotu a definovat vlastní tělo funkce.
Obecně tedy definice funkce má tvar
Definice funkce je následující:

```
navratovy_typ jmeno ([parametry])
{
telo funkce
}
Konkrétně například takto:
int secti (int a, int b)
{
return a + b;
printf("%d", a + b); // neprovede se
}
// ...
int c = secti(2, 3); // volání funkce, v proměnné c bude 5
```

Ve výpisu kódu jsme definovali funkci secti. V jazyce C není vyhrazené klíčové slovo pro funkci. Hlavička funkce se skládá pouze z návratového typu, jména a seznamu parametrů

oddělených čárkami v kulatých závorkách. Zde tedy máme funkci secti (jména smějí obsahovat pouze alfanumerické znaky, podtržítka a první znak nesmí být číslice), která vrací celočíselnou hodnotu a přebírá celkem dva parametry typu int, které jsme pojmenovali a a b. Jelikož jsme vytvořili funkci, která má vracet hodnotu, musíme zajistit předání její hodnoty. Příkazem return okamžitě ukončíme provádění funkce a vrátíme hodnotu výrazu zapsaný za ním. Funkce printf()se tedy nikdy neprovede.
Speciální označení pro funkci, která nic nevrací – procedura - se v jazyce C nepoužívá, pouze označíme, že je návratový typ prázdný. K tomu je určen speciální typ void.
Tento typ značí prázdnou hodnotu, hodnotu bez typu. Procedura v jazyce C tedy vypadá následovně:

```
void vypis_hello (void)
{
printf("hello world\n");
return; // není nutné
}
// ...
vypis_hello();
```

Vytvořili jsme funkci, která nic nevrací, neboli proceduru. Pokud někdy potřebujeme ukončit vykonávání procedury předčasně, zavoláme return bez výrazu. Pokud funkce, která nevrací hodnotu neobsahuje příkaz return, vykonávání funkce ukončí pravá složená závorka ukončující blok těla funkce. Za povšimnutí stojí také to, že i když funkce nepřejímá žádné parametry, musíme napsat kulaté závorky. Uvnitř závorek by mělo být podle standardu buď void, nebo nic.
Standard C99 povoluje jen první možnost. Pokud voláme funkci bez parametrů, musíme též napsat za název kulaté závorky. Pokud je vynecháme, funkce se nezavolá, a překladač nás neupozorní na chybu. Napsali jsme výraz, který odpovídá ukazateli na funkci.

1.23.2 Deklarace funkce

Při deklaraci funkce se uvádí pouze datový typ návratové hodnoty a typy argumentů. To je vše, co potřebuje překladač k tomu, aby její ***volání*** mohl do programu zapsat. Mohou se uvést i názvy argumentů, ale to není nutné. Než je funkce v programu volána, ***musí být deklarována.*** Definována může být až později. Pokud funkci předem nedeklarujete, ale rovnou definujete, je definice zároveň i deklarací. Pokud se deklarace s pozdější definicí neshodují, oznámí překladač chybu.
V následujícím programu si funkci nejdříve deklarujeme, použijeme jí v jiné funkci a pak jí teprve definujeme

1.23.3 Předávání parametrů funkcím.

Funkce se ve většině případů mají své parametry. Při deklaraci funkce hovoříme o formálních parametrech, při volání funkce jsou to pak parametry skutečné. Bez znalosti, jak se parametry funkcím předávají, nebudeme schopni napsat funkční kód.

1.23.4 Formální a skutečný parametr

Tento pojem ve skutečnosti zahrnuje dvě odlišné věci:
Formální parametr je parametr použitý při psaní funkce, její vnitřní proměnná. Tato proměnná je pro funkci lokální. Při volání funkce je vždy nahrazována hodnotou skutečného parametru.
Skutečný parametr je proměnná nebo výraz dosazený při volání funkce. Při volání funkce je jeho hodnota přiřazena formálnímu parametru.
Parametry jsou v jazyku C předávány pouze hodnotou. Skutečným parametrem tedy může být vždy výraz. Je-li formálním parametrem pole, předává se (hodnotou) adresa prvního prvku. To znamená, že pokud jsou formální i skutečný parametr pole stejného typu se stejným

rozsahem indexů, dojde vlastně k předání odkazem. Skutečným parametrem ovšem může být i libovolný výraz s hodnotou typu ukazatel na typ složek pole. Můžeme také předat pole s jiným rozsahem indexů (i s jiným počtem indexů). Pak je ovšem zcela na programátorovi, aby zabezpečil, že při práci s tímto parametrem nepřepáše důležitá data v paměti.

1.23.5 Způsoby předávání parametru

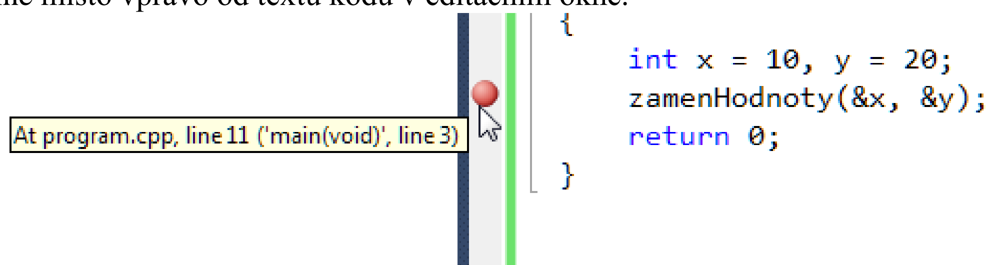
Navázání skutečného parametru na formální lze dosáhnout několika odlišnými způsoby. Ve většině dnešních programovacích jazyků se používají hlavně předávání parametrů hodnotou a odkazem:

Při předávání hodnotou (call-by-value) se těsně před zpracováním těla funkce skutečný parametr vyčíslí a výsledek se zkopíruje do lokální proměnné uvnitř volané funkce. Jakékoli změny parametru uvnitř volané funkce nemají vliv na volající funkci, neboť se pracuje s lokální kopií, předávání hodnotou tedy lze používat pouze pro vstupní parametry. Tento způsob je typický např. při vytváření aritmetických funkcí.

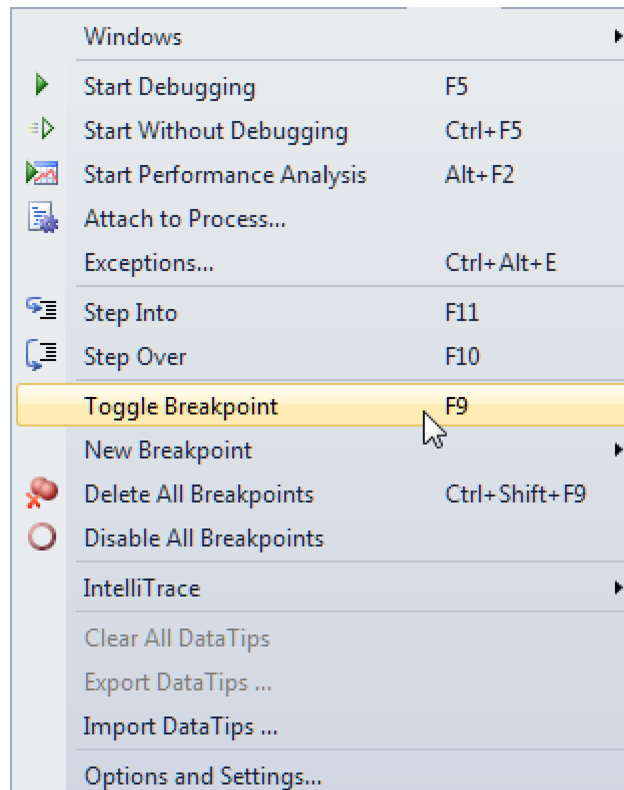
Funkce v jazyce C mohou mít několik parametrů a nejvýše jednu návratovou hodnotu, [1], [5]. Vstupní parametry funkcí je možné v jazyce C předávat pouze hodnotou. Na obr. 12 je zobrazena situace při volání funkce hodnotou.

```
void fce(float a) {
a = 10;
}
//...
void main(void) {
float x = 5;
fce(x);
}
```

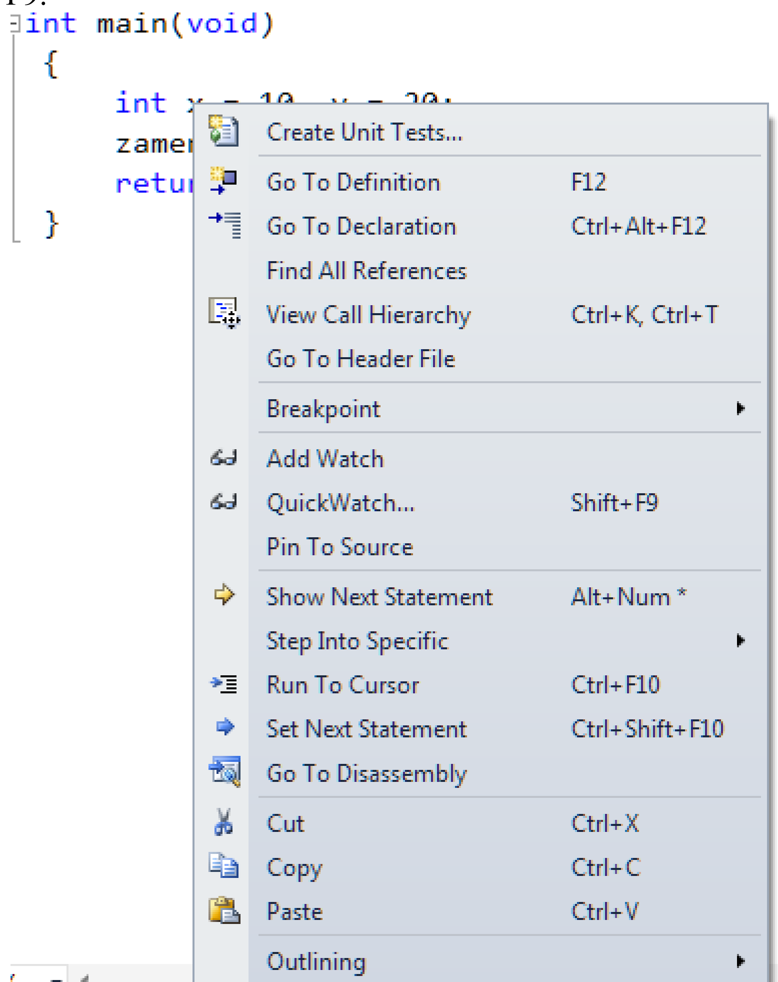
Abychom si názorně ukázali, jak se mění hodnoty proměnných, použijeme mocný ladící nástroj, a to debugger. V programu si nastavíme místo přerušení – breakpoint, na kterém se v režimu ladění zastaví vykovávání programu. To provedeme nejjednodušeji poklepáním na volné místo vpravo od textu kódu v editačním okně.



Hodnotu proměnných můžeme sledovat v okně Watch, které slouží pro výpis aktuálních stavů proměnných. Nové proměnné vložíme do sledování tak, že si v programu nastavíme kurzor na aktuální proměnnou a pomocí pravého tlačítka myši vyvoláme kontextovou nabídku. Zvolíme příkaz Add Watch.



Totéž lze učinit tak, že v textu nastavíme kurzor na řádek kódu, před který chceme vložit bod přerušení. Pak použijeme příkaz z menu Debug -> New Breakpoint->Toggle Breakpoint nebo funkční klávesu F9.



Ve funkci main je definována proměnná x, která je předána jako parametr funkce. Na obrázku ještě nemá přiřazenou hodnotu.

Watch 1

Name	Value	Type
x	-1.0737418e+008	float
a	CXX0017: Error: symbol "a" not foun	

Autos Locals Threads Modules Watch 1

Před voláním funkce fce je proměnné x přiřazena hodnota 10. Žlutá šipka ukazuje místo aktuálně prováděného příkazu.

```
void fce(float a) {
  a = 10;
}
//...
void main(void) {
  float x = 5;
  fce(x);
}
```

Watch 1

Name	Value	Type
x	5.0000000	float
a	CXX0017: Error: symbol "a" not foun	

Autos Locals Threads Modules Watch 1

Při volání funkce fce jsou všechny parametry zkopírovány do jiné části paměti a jsou nazvány jménem příslušného formálního parametru. Tyto proměnné jsou viditelné v celé funkci a zastiňují globální proměnné se stejným názvem. Kromě proměnné x funkce main je uvnitř funkce k dispozici nová proměnná a, proměnná vytvořená pro skutečný parametr funkce v okamžiku volání funkce. Do této proměnné se kopíruje hodnota proměnné x.

```
void fce(float a) {
  a = 10;
}
//...
void main(void) {
  float x = 5;
  fce(x);
}
```

Watch 1

Name	Value	Type
x	5.0000000	float
a	5.0000000	float

Autos Locals Threads Modules Watch 1

Po vykonání funkce fce a ještě před jejím ukončením je situace v paměti znázorněna na dalším obrázku. Proměnná a je naplněna hodnotou 10.

Watch 1

Name	Value	Type
x	5.0000000	float
a	10.000000	float

Autos Locals Threads Modules Watch 1

```
void fce(float a) {
    a = 10;
}
//...
void main(void) {
    float x = 5;
    fce(x);
}
```

Po ukončení funkce je proměnná a z paměti odstraněna a její obsah zapomenut. Funkce ve svém výsledku neudělá nic. Pokud mají změny provedené uvnitř funkce ovlivnit proměnnou i mimo tuto funkci, musí se předávat pomocí ukazatelů. Na obrázku vidíme situaci po návratu do funkce main. Proměnná a ve v okně Watch je šedá. Což znamená, že její hodnota není aktuální.

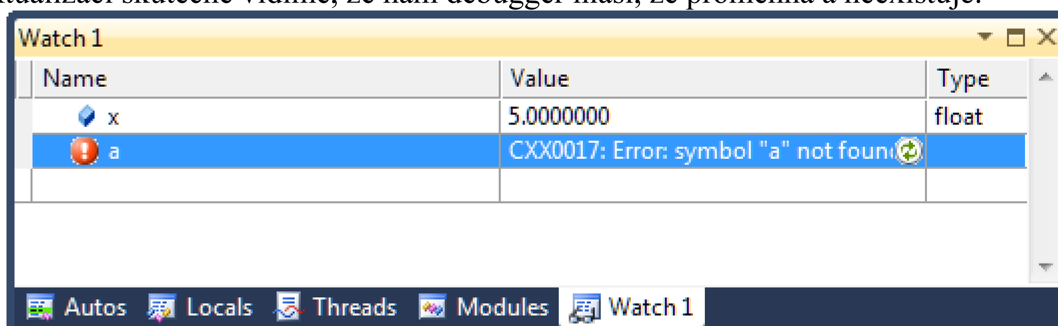
Watch 1

Name	Value	Type
x	5.0000000	float
a	10.000000	float

Autos Locals Threads Modules Watch 1

```
void fce(float a) {
    a = 10;
}
//...
void main(void) {
    float x = 5;
    fce(x);
}
```

Po aktualizaci skutečně vidíme, že nám debugger hlásí, že proměnná a neexistuje.



Při předávání odkazem (call-by-reference) se formální parametr uvnitř volané funkce bere jen jako jiné označení (alias) pro proměnnou předanou jako skutečný parametr, tzn. ve volané funkci se pracuje přímo s předávanou proměnnou, nevytváří se tedy kopie (což zvláště u strukturovaných proměnných znamená zpravidla úsporu času i paměti). Volaná funkce změnou parametru ovlivňuje i volající funkci (takže předávání odkazem lze používat pro výstupní či vstupně-výstupní parametry), nevýhodou však je, že parametrem může být jen proměnná a nikoli výsledek obecného výrazu. Předávání odkazem se obvykle implementuje pomocí ukazatele na předávanou proměnnou.

```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
int pom;
pom = a;
a = b;
b = pom;
}
```

Tuto metodu budeme volat v programu následně:

```
int main(void)
{
int x = 10, y = 20;
zamenHodnoty(x, y);
return 0;
}
```

Obdobně, jako v příkladu výše si ukážeme, jaké hodnoty nabývají proměnné x, y a parametry funkce a a b při vykovávání programu. Stav proměnných před volání funkce zamenHodnoty.

```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
    int pom;
    pom = a;
    a = b;
    b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(x, y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	CXX0017: Error: symbol "pom" not fc	
a	CXX0017: Error: symbol "a" not foun	
b	CXX0017: Error: symbol "b" not foun	

Autos Locals Threads Modules Watch 1

Stav proměnných na začátku vykonávání funkce zamenHodnoty.

```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
    int pom;
    pom = a;
    a = b;
    b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(x, y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	-858993460	int
a	10	int
b	20	int

Autos Locals Threads Modules Watch 1

Hodnoty proměnných před opuštěním funkce zamenHodnoty.

```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
    int pom;
    pom = a;
    a = b;
    b = pom;
}
int main(void)
{
    int x = 10, y = 20;
    zamenHodnoty(x, y);
    return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	10	int
a	20	int
b	10	int

Autos | Locals | Threads | Modules | Watch 1

Hodnoty proměnných po návratu do funkce main.

```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
    int pom;
    pom = a;
    a = b;
    b = pom;
}
int main(void)
{
    int x = 10, y = 20;
    zamenHodnoty(x, y);
    return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	CXX0017: Error: symbol "pom" not fo	
a	CXX0017: Error: symbol "a" not foun	
b	CXX0017: Error: symbol "b" not foun	

Autos | Locals | Threads | Modules | Watch 1

Předávání ukazatelem

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
int pom;
pom = *a;
*a = *b;
*b = pom;
}
int main(void)
 {
 int x = 10, y = 20;
 zamenHodnoty(&x, &y);
 return 0;
```

Tuto metodu budeme volat v programu následně:

```
int main(void)
 {
 int x = 10, y = 20;
 zamenHodnoty(&x, &y);
 return 0;
```

Opět si ukážeme trasováním programu, jak se mění hodnoty jednotlivých proměnných. Před voláním funkce zamenHodnoty mají proměnné x hodnotu deset a y hodnotu dvacet. Funkce zamenHodnoty je nyní deklarovaná jako funkce se dvěma ukazateli na int *a a *b. Proto musíme funkce předat nikoli obsah proměnných x a y ale jejich adresy. Ty získáme pomocí operátoru získání adresy &.

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
    int pom;
    pom = *a;
    *a = *b;
    *b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(&x, &y);
     return 0;
}
```

Name	Value	Type
x	10	int
y	20	int
pom	CXX0017: Error: symbol "pom" not fo	
a	CXX0017: Error: symbol "a" not foun	
b	CXX0017: Error: symbol "b" not foun	

Stav v na začátku vykonávání funkce zamenHodnoty.

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
    int pom;
    pom = *a;
    *a = *b;
    *b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(&x, &y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	-858993460	int
a	0x003dfb68	int *
	10	int
b	0x003dfb5c	int *
	20	int

Autos | Locals | Threads | Modules | Watch 1

Vidíme, že hodnota parametrů funkce zamenHodnoty a a b tentokrát není 10 a 20, ale a obsahuje hodnotu 0x003dfb68 a b pak hodnotu 0x003dfb5c, což jsou adresy proměnných a a b ve funkci main.

Druhý řádek funkce do proměnné pom neuloží obsah parametru a, ale pomocí operátoru dereference * se získá hodnota paměťového místa s adresou, kterou obsahuje ukazatel a, tudíž se získá hodnota proměnné x, která se uloží do pomocné proměnné pom.

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
    int pom;
    pom = *a;
    *a = *b;
    *b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(&x, &y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	10	int
a	0x003dfb68	int *
	10	int
b	0x003dfb5c	int *
	20	int

Autos Locals Threads Modules Watch 1

Podívejme se, jak bude vypadat situace po provedení příkazu *a = *b;

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
    int pom;
    pom = *a;
    *a = *b;
    *b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(&x, &y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	10	int
a	0x003dfb68	int *
	20	int
b	0x003dfb5c	int *
	20	int

Autos Locals Threads Modules Watch 1

Hodnota na adrese 0x003dfb68 se změnila na hodnotu dvacet, kterou obsahuje proměnná místo s adresou uloženou v ukazateli b. Uloženou získáme nepřímo dereferencí ukazatele *b. Uvědomme si, že se jedná o hodnotu proměnné y.
Před návratem do funkce main bude stav následující:

Watch 1

Name	Value	Type
x	10	int
y	20	int
pom	10	int
a	0x003dfb68	int *
	20	int
b	0x003dfb5c	int *
	10	int

Autos Locals Threads Modules Watch 1

Obsah paměťového místa s adresou 0x003dfb5c se změnil na hodnotu deset. K tomuto místu jsme opět přistoupili nepřímo dereferencí ukazatele *b. Opět si uvědomme, že adresa 0x003dfb5c je adresou proměnné y. Po návratu do funkce main se z paměti uvolní dočasné proměnné odpovídající parametrům funkce zamen

```
void zamenHodnoty(int *a, int *b) //zameni hodnoty v promennych
{
    int pom;
    pom = *a;
    *a = *b;
    *b = pom;
}
int main(void)
{
     int x = 10, y = 20;
     zamenHodnoty(&x, &y);
     return 0;
}
```

Watch 1

Name	Value	Type
x	20	int
y	10	int
pom	10	int
a	0x003dfb68	int *
	20	int
b	0x003dfb5c	int *
	10	int

Autos Locals Threads Modules Watch 1

1.23.6 Rekurze

V jazyce C lze bez problému využít rekurzi. Stačí pouze zavolat funkci samu v jejím těle. Je samozřejmě na nás, abychom zajistili konečnost rekurze. To znamená, že volání funkce uvnitř jejího těla musí být v nějaké podmínce anebo před jejím voláním musí být v podmínce příkaz return.

```
void vypisSetupneRadu (int cislo)
{
```

```
if (cislo < 0) return;
printf("%d\n", cislo);
vypisSetupneRadu(cislo - 1);
}
// ...
vypisSetupneRadu(10);
```

Funkce vypisSestupneRadu vypíše čísla od hodnoty svého parametru postupně až po hodnotu nula. Zde funkce jde volána z nějakého místa v programu s hodnotou skutečného parametru deset, budou vypsána postupně čísla od desíti po nulu. Tato rekurze má ukončovací podmínku, kterou je test zápornosti předaného parametru. Pokud je parametr funkce záporný, rekurze se ukončí. Jinak na řádku pět voláme v těle funkce tu samou funkci znovu se změněným parametrem. V okamžiku volání funkce se přeruší vykonávaní původní funkce, vykonává se vnořená funkce a až po návratu z této funkce se pokračuje ve vykonávání vnější funkce. V našem případě již je funkce ukončena koncem bloku svého těla, tedy pravou složenou závorkou. Tento kód stejně jako většina ostatních případů rekurze, lze mnohem efektivněji přepsat za pomoci cyklu.

1.24 FUNKCE MAIN ()

Už v prvním programu jsme psali funkci main (). Tato funkce je vstupním bodem programu, proto musí být v každém programu, a to právě jednou! Její hlavička je poměrně komplikovaná, její účel je ale zřejmý.

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

Dříve jsme se setkali jen s druhým typem hlavičky. Prvá hlavička je tzv. plná definice funkce main (). Konzolové aplikace mohou měnit svoje chování v závislosti na předaných parametrech příkazové řádky. Tyto parametry pak program obdrží právě v oněch dvou parametrech int argc, char **argv funkce main ().

První parametr udává, s kolika parametry je program volán, druhý parametr je seznam řetězců, ve kterém jsou postupně uložené všechny parametry příkazové řádky. V případě, že nemáme o tyto parametry zájem, můžeme využít hlavičku funkce main bez parametrů.

Návratová hodnota funkce main () představuje možnost, jak oznámit systému, ze kterého je program spouštěn, že program skončil v pořádku, či s chybou. Standardně je hodnota IT_SUCCESS(definovaná v souboru stdlib.h většinou jako nula) považována za signalizaci správného ukončení programu. V případě předání EXIT_FAILURE došlo při běhu programu k chybě.

```
#include <stdio.h>
int main (int argc, char **argv)
{
int i;
for (i = 0; i < argc; i++)
{
puts(argv[i]);
}
return 0;
}
```

program vypíše všechny předané parametry. Máme-li program přeložen pod jménem parametry a zavoláme z příkazové řádky v následující podobě

```
./parametry parametr1 --parametr2 par3
obdržíme výpis
./parametry
```

```
parametr1
--parametr2
par3
```

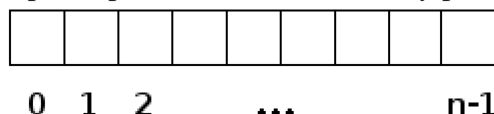
Všimněme si, že první parametr není parametr1, ale. /parametry, tedy vlastní jméno spouštěného programu včetně cesty.
První skutečný parametr má tedy index jedna a ne nula, jak bychom očekávali. Parametry se nijak neupravují a předávají se s pomlčkami i jiným formátováním. Při volání funkce jsou pouze rozsekány podle mezer mezi nimi. Je na programátorovi, aby si parametry sám zpracoval.

1.25 POLE

Pole jsou jednou ze složitějších datových struktur. Tento článek stručně popisuje deklaraci pole, jeho inicializaci a práci s ním. Jsou též zmíněna vícerozměrná pole.

1.25.1 Úvod do polí

Pole je homogenní datová struktura, ve které jsou data shodného datového typu uchována za sebou v paměti. Přístup do pole je časově konstantní, zatímco manipulace s polem má lineární složitost. Pole je v paměti zarovnáno tak, jak je to zobrazeno na obrázku níže. V jazyce C se indexuje od nuly. Tedy první prvek pole má index nula a n-tý prvek má index n mínus jedna.



struktura pole, jeho podoba v paměti RAM

1.25.2 Deklarace pole

Deklarace pole má následující strukturu:

```
<typ> <jméno>[<[velikost]>];
```

Deklarace se tedy příliš neliší od deklarace proměnné. Typ je základní typ buňky pole (pole je homogenní, není možné měnit typ u jednotlivých buněk), jméno je identifikátor a velikost určuje, kolik prvků bude pole mít. V klasickém jazyce C nebylo možné použít jako velikost nic jiného než konstantu či výraz, který dokázal překladač vyhodnotit již době překladu. Standard C99 umožňuje používat i nekonstantní výrazy (pro lokální nekonstantní pole). V C99 může být mez pole proměnná, pokud je pole deklarováno lokálně ve funkci:

```
int pocet;
scanf("%d",&pocet);
double x[pocet],y[pocet]
```

Pro globální pole platí totéž, co pro rozměr pole platí v klasickém ANSI C.
Pokud potřebujeme měnit velikost pole, pak je vhodné pro určení rozměru pole použít makro.

```
#define VELIKOST 81
// ...
char nadpis[VELIKOST];
```

Velikost je nepovinný údaj, jak jsme se zmínili v kapitole o řetězcích, stačí inicializovat pole a překladač si velikost doplní sám. Pokud velikost uvedeme, musíme vždy použít celé nezáporné číslo. V konkrétním případě tedy pole o deseti prvcích se jménem pole a typem int bude vypadat

```
int pole[10];
```

1.25.3 Inicializace pole

Pole můžeme inicializovat jako každou jinou proměnnou při jejím vzniku. U pole je to ale o něco složitější. Musíme totiž vyjmenovat obsahy jednotlivých buněk.

```
int pole1[5] = {1, 2, 3, 4, 5};
int pole2[] = {1, 2, 3, 4, 5};
int pole3[5] = {1, 2};
int pole4[5] = {0};
```

Na první řádce pole s pěti buňkami kompletně inicializujeme hodnotami ve složených závorkách. Na druhé řádce vytvoříme ekvivalentní pole s polem pole1, viz výše. Jak bude inicializováno pole pole3? Zde jsou nastaveny pouze první dva prvky pole, zbytek bude vynulován. Stejně tak pole4 bude kompletně vyplněné nulami.

1.25.4 Práce s polem

Pro práci s poli je dodržovat určitá pravidla. Ta se týkají přístupu k prvkům pole, kopírování či porovnávaní polí.

1.25.5 Indexace

Máme-li vytvořenou proměnnou typu pole, můžeme k jednotlivým buňkám přistupovat pomocí operátoru indexace, neboli indexem v hranatých závorkách.
Nesmíme ovšem zapomínat na to, že první buňka má index nula

```
int pole[5] = {1, 2, 3, 4, 5};
int a = pole[0]; // první buňka pole, v a bude 1
int b = pole[1]; // druhá buňka pole, v b bude 2
int c = pole[5]; // chyba, čteme za konecem pole!!
```

Na poslední řádce se pokoušíme číst buňku s indexem pět. To je samozřejmě chyba, protože pole má poslední buňku s indexem čtyři. Čteme tudíž hodnotu z paměti, ležící bezprostředně za posledním prvkem pole. Jazyk C nemá žádnou zabudovanou kontrolu mezí polí. To může způsobit nepředvídatelné chování nebo pád programu.

1.25.6 Porovnávání polí

Pole nelze jednoduše porovnávat, jako to jde s jednoduchými číselnými typy. Od následujícího kódu nelze očekávat korektní chování, pokud nám jde o srovnání
obsahu polí:

```
if (pole1 == pole2) // ...
```

Jak se zmíníme později, identifikátor pole bez operátoru indexace – hranatých závorek - je totiž chápán jako ukazatel na počátek pole, tedy na buňku pole s indexem 0.
Chceme-li srovnat obsahy polí, nezbude nám nic jiného, než napsat cyklus a porovnat buňku po buňce jedno pole s druhým. Máme-li stejně dlouhá pole, provedeme srovnání následovně:

```
int shoda = 1;
for (i = 0; i < 10; i++)
{
if (pole1[i] != pole2[i]) // pole se liší
{
shoda = 0;
break; // přerušíme cyklus a pokračujeme za ním
}
}
if (shoda)
{
printf("pole se rovnaji\n");
```

```
}
```

Proměnnou shoda vlastně ani nepotřebujeme. Pokud cyklus projde všechny prvky pole a nebude přerušen příkazem break, pak jsou pole shodná.

```
int i;
for (i = 0; i < 10; i++)
{
if (pole1[i] != pole2[i]) // pole se liší
{
break; // přerušíme cyklus a pokračujeme za ním
}
}
if (i == 10)  // cyklus prošel všechny prvky pole, jsou tudíž shodné
{
printf("pole se rovnaji\n");
}
```

Pokud budeme uvažovat jak dále program zkrátit, pak to lze učinit úpravou podmínky cyklu.

```
int i;
for (i = 0; (i < 10) && (pole1[i] == pole2[i]); i++);

if (i == 10)  // cyklus prošel všechny prvky pole, jsou tudíž shodné
{
printf("pole se rovnaji\n");
}
```

Povšimněte si, že tělo cyklu je prázdné, tvoří ho prázdný příkaz, který ve výpise představuje středník bezprostředně za hlavičkou cyklu.
Další možností je použít funkci

```
int memcmp(const void *pole1, const void *pole2, size_t velikost)
```

z hlavičkového souboru string.h, která porovná dva úseky paměti zadané délky a vrátí podobně jako funkce strcmp()hodnotu menší než nula, nula, nebo větší než nula, pokud je první pole po bajtech menší, rovno, nebo větší než druhé pole.

1.25.7 Pole jako parametr funkce

Pole lze předávat do funkce jako každou jinou proměnnou. Stačí pouze označit, že se jedná o pole párem hranatých závorek:

```
void funkce (int pole[], int velikost)
```

Jako první parametr funkce funkce se očekává pole typu int. Ve druhém parametru předáváme délku pole. Délka pole se nedá jednoduše zjistit, a proto ji musíme předávat samostatně. S polem se v těle funkce pracuje jako s obyčejným lokálně vytvořeným polem.

1.25.8 Vícerozměrná pole

Nejsme omezeni ale jen jednorozměrným polem. Jazyk C nás ale v dimenzích pole neomezuje. Dvourozměrné pole vytvoříme zadáním velikosti nadvakrát

```
int pole2d [3][3];
```

Zde jsme vytvořili dvourozměrnou matici 3x3. Vícerozměrná pole můžeme také inicializovat. Jelikož jsou pole v jazyce C ukládaná po řádkách, musíme
inicializaci psát řádkově.

```
int pole2d1 [3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int pole2d2 [3][3] = {{1, 2, 3}};
int pole2d3 [3][3] = {{0}};
```

Jako v případě jednorozměrných polí, i zde můžeme část inicializace vynechat. Překladač zbytek doplní nulami. Pole pole2d3 bude tedy reprezentovat nulovou matici 3x3.

Na libovolný prvek pole se dá dostat vícenásobnou indexací.

```
int a = pole2d[1][1];
```

Stejně jako u jednorozměrných polí i zde se indexuje od nuly. Levý horní prvek matice má tedy index [0][0].

Příklad deklarace vícerozměrného pole a způsob procházení jeho prvků je ukázán v následujícím příkladu.

```
#define RADKY 10
#define SLOUPCE 8
//...
int table[RADKY][SLOUPCE]; /* rozměry musí být konstantní */
//...
for(i=0; i<RADKY; i++)
for(j=0; j<SLOUPCE; j++)
{ ...
... table[i][j] ...  /* POZOR: nikoli table[i,j] !!! */
/* zde pracujeme s prvkem o indexech i,j */
}
```

evropský sociální fond v ČR · EVROPSKÁ UNIE · MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY · OP Vzdělávání pro konkurenceschopnost

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Pokročilejší programovací techniky

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

2 POKROČILEJŠÍ PROGRAMOVACÍ TECHNIKY 4
2.1 Ukazatele 5
2.1.1 Motivace 5
2.1.2 Deklarace ukazatele 5
2.1.3 Ukazatel bez doménového typu a neplatná adresa 5
2.1.4 Prázdný ukazatel 6
2.1.5 Konstantní ukazatel a ukazatel na konstantu 7
2.1.6 Vztah pole a ukazatele 7
2.2 Adresová aritmetika a operátor sizeof() 9
2.3 Ukazatele na funkce 9
2.4 Vícenásobné ukazatele aneb ukazatele na ukazatele 10
2.4.1 Pole ukazatelů. 10
2.5 Dynamická alokace paměti 10
2.5.1 Motivace 10
2.5.2 Vyhrazení a uvolnění paměti 11
2.6 Alokace vícerozměrného pole 12
2.7 Struktury a výčtový typ 13
2.7.1 Výčtový typ enum 13
2.7.2 Deklarace enum 14
2.7.3 Deklarace s použitím typedef 14
2.7.4 Přetypování na int 14
2.7.5 Porovnávání výčtových proměnných 15
2.8 Struktura 15
2.8.1 Deklarace struktury 15
2.8.2 Inicializace struktury 15
2.8.3 Přístup ke členům struktury 15
2.8.4 Kopírování a porovnávání 16
2.9 Unie 16
2.10 Soubory 16
2.10.1 Otevření souboru 16
2.10.2 Ostatní funkce pro práci se souborem 17
2.11 Standardní vstup a výstup jako soubory 19

2.12 Preprocesor 19

2.12.1 První fáze překladu 19

2.12.2 Připojení souboru 19

2.12.3 Podmíněný překlad 20

2 POKROČILEJŠÍ PROGRAMOVACÍ TECHNIKY

OBSAH KAPITOLY:

Co je a k čemu slouží ukazatel

Jaký je rozdíl mezi staticky a dynamicky alokovanou pamětí

MOTIVACE:

Po prostudování této přednášky budete schopni popsat smysl použití databází a databázových systémů, identifikovat základní problémy, které databázový přístup řeší a definovat základní databázové pojmy.

CÍL:

Po prostudování tohoto odstavce budete umět:

- Pracovat s ukazateli
- Používat adresovou aritmetiku a operátor sizeof()
- Ukazatel na funkce
- Dynamickou alokaci paměti
- Struktury a výčtový typ

2.1 UKAZATELE

Mimo základní datové typy v jazyce c existují typy odvozené. Jedním z těchto typů jsou ukazatele. Ukazatel je v podstatě abstraktnější ztvárnění adresy. Jde o techniku, jak nepřímo přistupovat k proměnným a datům v paměti. V tomto článku se seznámíme s ukazateli, přetypováním ukazatelů, ukazatelům na funkce a ukazatelům na ukazatele. Dále bude vysvětlena adresová aritmetika a některé další detaily týkající se ukazatelů.

2.1.1 Motivace

Proč vůbec zavádět ukazatele, když doteď jsme se bez nich obešli. Ukazatele jsou potřebné ze dvou důvodů, O prvním důvodu jsme se zmínili, když jsme rozebírali způsob předávání parametrů funkcím. Druhým důvodem je použití dynamicky (tedy proměnně) vytvářených proměnných.

2.1.2 Deklarace ukazatele

Obecná deklarace ukazatele (pointeru) se od deklarace proměnné liší pouze použitím hvězdičky před jménem proměnné.

```
<domenovy_typ> *<jméno>;
```

Konkrétně pak například

```
int a = 3;
int *pa = &a;
```

Zde jsme definovali proměnnou a s hodnotou tři a poté jsme definovali ukazatel na typ int, do kterého jsme uložili adresu proměnné a. Všechna data leží někde v paměti, kterou si můžeme představit jako obrovské pole. Každá buňka tohoto pole (bajt) má svoji adresu, což je pořadové číslo buňky v paměti. Ukazatel tak není nic jiného než proměnná uchovávající toto číslo.

K získání adresy dat slouží operátor získání adresy (reference na proměnnou) &.

Pokud chceme přistupovat k datům, na která ukazuje ukazatel, musíme použít operátor dereference *

Výhoda tohoto přístupu spočívá v tom, že se můžeme odkazovat na velké množství dat jen s pomocí čísla uloženého na čtyřech byte, což je velikost proměnné typu ukazatel, a nemusíme data kopírovat, například když je potřebuje zpřístupniv jako parametr funkce.

Parametr nelze nikdy inicializovat přímo číselnou hodnotou.

```
pa = 2; // chyba
*pa = 2;
```

V první řádce našeho příkladu je sémantická chyba, kde místo hodnoty proměnné a přepisujeme adresu uchovanou v ukazateli.

2.1.3 Ukazatel bez doménového typu a neplatná adresa

Ukazatele mohou ukazovat na libovolná data. Existuje však speciální forma ukazatele, tzv. ukazatel bez doménového typu.

```
void *ukazatel;
```

Tento ukazatel může ukazovat na libovolná data a jakýkoliv ukazatel lze převést na ukazatel bez doménového typu. Takovýto ukazatel ale logicky nelze dereferencovat, neboť nevíme, na jaká data ukazuje, a tudíž nevíme, jak máme odkazovaná data reprezentovat. Před přístupem k datům tak ukazatel musíme explicitně přetypovat na ukazatel s konkrétním doménovým typem. Ukazatel bez doménového typu se hojně využívá v místech, která mají být dostatečně obecná a musí umět pracovat s různým typem dat.

2.1.4 Prázdný ukazatel

Ukazatel může také ukazovat "nikam". Jeho hodnota je potom dána standardním makrem NULL (obvykle je to nula). Ukazatel NULL nelze dereferencovat. Makro je standardně definováno takto:

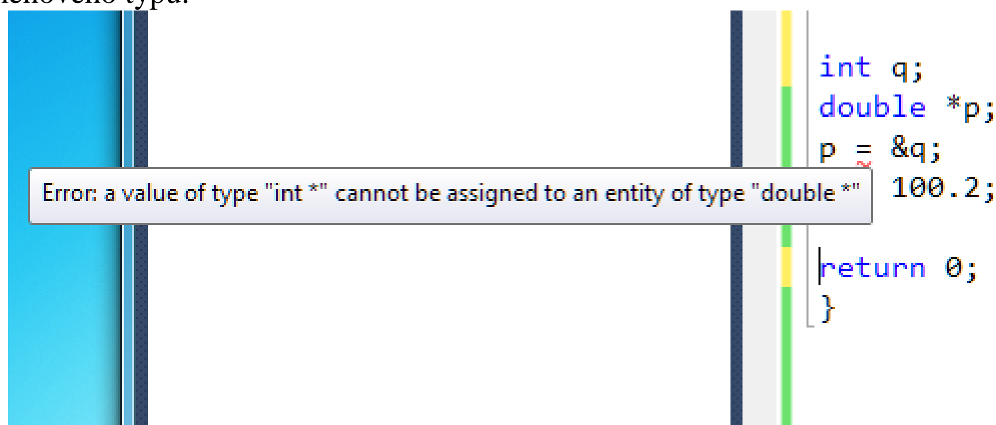
```
#define NULL ((void*)0)
```

Jinými slovy neplatný ukazatel je na adrese nula. Proto můžeme testovat ukazatele na platnost v podmínkách jako každé jiné číslo. Ukazatel NULL se vyhodnotí jako nepravda. Snažte se dodržovat pravidlo, že pokud ukazatel nikam neukazuje, má mít hodnotu NULL.

Nyní si ukážeme, jak lze jednoduše udělat chybu.

```
int q;
double *p;
p = &q;
p = 100.2;
```

Ukazateli p je přiřazena adresa čísla int. Tato adresa je pak použita na levé straně přiřazovacího příkazu pro přiřazení hodnoty s pohyblivou řádovou čárkou. Jelikož je však číslo typu int obvykle kratší než double, způsobil by přiřazovací příkaz přepsání paměti sousedící s q. Proto lze přiřazovat ukazatele sobě navzájem, jen pokud jsou shodného doménového typu.



Další možný chybný zápis vzniká tehdy, pokud se snažíme použít ukazatel dříve než je do něj přiřazena adresa proměnné. V takovém případě program pravděpodobně zhavaruje. Je důležité si uvědomit, že deklarace ukazatele pouze vytvoří proměnnou schopnou uchovávat adresu paměti. Nedá ji však žádnou smysluplnou hodnotu.

```
int *p;
*p = 10; //chyba - ukazatel p na nic neukazuje!
```

Další chyba může vzniknout následujícím způsobem.

```
int q,r;
int *p;
p = &q;
r = *p;
```

Jaká bude hodnota proměnné r? V příkladu jsme nejprve deklarovali dvě proměnné, q a r. Deklarovali jsme ukazatel na int p. Inicializovali jsme ukazatel p adresou proměnné q. Zatím je vše v pořádku. Chybný je poslední řádek kódu. Dereferencovali jsme ukazatel p. To znamená, že pomocí ukazatele p nepřímo čteme obsah proměnné q. Obsah této proměnné, protože jsme proměnnou zatím nikde nepřiřadili hodnotu, neinicializovali ji, je zcela náhodný. Tato chyba vede k neočekávanému chování programu či jeho zhroucení.

Místa, na která se ukazatel odkazuje (jejichž adresu obsahuje) se mohou lišit. Takže můžeme například psát cykly, v nichž pomocí jediného ukazatele postupně přistupujeme ke všem prvkům pole. Další možnosti použití ukazatelů se otevírají při dynamickém přidělování paměti.

2.1.5 Konstantní ukazatel a ukazatel na konstantu

V deklaraci ukazatele lze přidat klíčové slovo const. Podle toho, kam ho vložíme, vytvoříme konstantní ukazatel, nebo ukazatel na konstantu.

```
const int *p; // ukazatel na konstantu
int *const p; // konstantní ukazate
```

První řádek deklaruje ukazatel na konstantu. Takový ukazatel můžeme měnit, odkazovaná data jsou ale konstantní a tedy neměnná. Při zápisu do dereferencovaného ukazatele obdržíme chybové hlášení o přístupu k datům pouze pro čtení.
Druhá řádka deklaruje konstantní ukazatel. Takový ukazatel je pevně spojen s obsahem a jeho hodnotu nelze měnit. Lze ale měnit odkazovaná data.

2.1.6 Vztah pole a ukazatele

Pole a ukazatele v jazyce C spolu souvisí. Pole, jejich identifikátor bez operátoru indexování, představují totiž konstantní ukazatele na první prvek pole (s indexem nula). Toto chování se liší pouze při použití operátoru sizeof(), který vrátí velikost pole a nikoliv velikost ukazatele, a při získávání adresy pole, kde vrácený ukazatel představuje jiný typ než ukazatel na první prvek (číselná hodnota je ale stejná).
Pokud se pokusíme přiřadit ukazateli na typ p adresu pole, získanou operátorem získání adresy a identifikátorem pole, obdržíme chybové hlášení, že ukazateli na int nelze přiřadit ukazatel typu int(*)[10].

```
p = pole;
p = &pole;
    &pole[0];
velikostPole = sizeof(pole);
```

Error: a value of type "int (*)[10]" cannot be assigned to an entity of type "int *"

```
int main (int argc, char **argv)
{
int pole[10];
int velikostPole;
int pocetPorvkuPole;
int *p, *q;
p = pole;
// p = &pole; nelze
q = &pole[0];
velikostPole = sizeof(pole);
pocetPorvkuPole = sizeof(pole)/sizeof(int);
}
```

Hodnoty ukazatelů pole, p a q budou shodné.

Watch 1

Name	Value	Type
⊟ p	0x0014f7cc	int *
	-858993460	int
⊟ q	0x0014f7cc	int *
	-858993460	int
⊟ pole	0x0014f7cc	int [10]
[0]	-858993460	int
[1]	-858993460	int
[2]	-858993460	int
[3]	-858993460	int
[4]	-858993460	int
[5]	-858993460	int
[6]	-858993460	int
[7]	-858993460	int
[8]	-858993460	int
[9]	-858993460	int
velikostPole	40	int
pocetPorvkuPole	10	int

Proměnná velikostPole bude obsahovat hodnotu 40, což je výsledek výrazu ***sizeof***(pole) a jelikož velikost datového typu int je čtyři byte, pak počet prvků pole je 10, což odpovídá hodnotě proměnné pocetPrvkuPole.

```
void funkce (int pole[], int velikost)
{
int pocet;
pocet = sizeof(pole)/sizeof(int);
}
```

Ukažme si princip na příkladu.Pozor ale v okamžiku, kdy pole budeme předávat jako parametr funkce, pak je toto pole předáno hodnotou adresy prvého prvku pole, která je uložena do proměnné odpovídající parametru funkce, v následujícím příkladu parametru int pole[]. Uvnitř funkce již není dostupná informace o velikosti původního pole a proto se proměnná počet nastaví na hodnotu jedna, ať je původní pole jakkoli velké.

```
int main (int argc, char **argv)
{
int pole[10];
int velikostPole, velikostPole1, velikostPole2;
int *p, *q;
p = pole;
// p = &pole; nelze
q = &pole[0];
velikostPole = sizeof(pole);
velikostPole1 = sizeof(p);
= sizeof(q);
}
```

Proměnné p a q jsou ukazatele na int a je jim přiřazena adresa ukazatele pole a adresa ukazatele na první prvek pole. Do proměnné se uloží zjištěná velikost pole. Do proměnné velikostPole1 se uloží velikost proměnné p a do proměnné velikostPole2 pak velikost proměnné q. Mohli bychom mylně předpokládat, že pokud ukazatele p a q ukazují na pole o deseti prvcích typu int, pak operátorem sizeof získáme onoho velikost pole. Není tomu tak. Obě zjištěné velikosti budou jen čtyři, což odpovídá počtu byte, na kterém je implementován typ ukazatel, jak vidíme z hodnot proměnných velikostPole1 a velikostPole2.

Watch 1

Name	Value	Type
velikostPole	40	int
velikostPole1	4	int
velikostPole2	4	int

Proto nebude fungovat funkce, jelikož skutečný parametr předaného pole se vyčíslí jako hodnota ukazatele pole, stejně jako je tomu v případě ukazatele p v našem příkladu.

2.2 ADRESOVÁ ARITMETIKA A OPERÁTOR SIZEOF()

Jelikož je ukazatel číslo, můžeme na něj aplikovat určité matematické operátory. Tyto operace byly navrženy pro efektivní implementaci indexace v poli. K ukazateli lze přičíst konstantu a lze spočítat rozdíl dvou ukazatelů.

```
int pole[3] = {1, 2, 3};
int a = pole[0]; // v a bude 1
a = *(pole + 1); // v a bude 2
```

Pozor na výraz pole + 1, mohli bychom očekávat, že výraz bude o jedničku větší než hodnota pole. Tak tomu ale není. Hodnota ukazatele se zde změní o velikost jeho typu, zde hodnotu čtyři. Ukazatel doménového typu nemá žádný typ, tudíž je podle standardu nelze s ním počítat pomocí ukazatelové aritmetiky. Nelze určit, o kolik by se měla hodnota ukazatele změnit, pokud bychom k němu přičetli jedničku. Abychom jej mohli použít v adresové aritmetice, je nutné ho nejprve přetypovat na nějaký typový ukazatel.

Na třetí řádce výpisu tak kód pole + 1 způsobí posunutí ukazatele na druhý prvek pole. Následná dereference tedy vrátí hodnotu dvě. Je tudíž ekvivalentní zápis pole[1] a * (pole + 1);

Přesně takto se interně převádí indexace polí v jazyce C – přičtení konstanty a následná dereference. Proto je také první prvek pole má index nula. Závorky kolem vlastního výrazu pole + 1 jsou nutné, neboť priorita operátoru dereferencování je vyšší než priorita operátoru přičtení. Proto následující kód do proměnné a zapíše hodnotu čtyři.

```
int pole[3] = {3, 2, 1};
a = *pole + 1; // v a bude 4
```

Ukazatele lze také odčítat. Musí jít o ukazatele stejného typu a měly by ukazovat na stejné pole. Výsledkem této operace je počet prvků pole mezi těmito ukazateli. U ukazatelů bez doménového typu platí to samé, co bylo uvedeno výše u přičítání konstanty. I zde je nutné ukazatel nejprve přetypovat.

K získání velikosti typu lze použít operátor sizeof(). Tento operátor přejímá název typu a vrací jeho velikost v bajtech. Vyhodnocení tohoto operátoru se provádí při překladu, protože překladač ví, jak jsou jednotlivé typy veliké.

```
int velikost = sizeof(int);
int velikost2 = sizeof(int*);
```

Hodnoty velikost a velikost2s e budou lišit podle toho, kde kód přeložíme. Na 64 bitovém systému budou hodnoty nejpravděpodobněji čtyři a osm. Není proto vhodné převádět ukazatel na číslo typu int, protože se na určitých architekturách nemusí hodnota ukazatele do rozsahu čísla int vejít.

2.3 UKAZATELE NA FUNKCE

V jazyce C se lze odkazovat na funkce. Deklarace ukazatele na funkci vypadá následovně:

```
int (*funkce)(int a, int b); // je deklarace ukazatele
int *funkce(int a, int b); // je deklarace funkce
```

Zde jsme vytvořili ukazatel na funkci, která vrací hodnotu typu int a přejímá dva celočíselné parametry. Od deklarace funkce se liší pouze přidáním hvězdičky a závorek. Závorky jsou povinné, neboť jinak bychom deklarovali funkci, která vrací ukazatel na int. Jak ale získat adresu funkce? Na funkci nelze použít operátor adresace &. Adresu funkce získáme přímo zápisem jméno funkce bez závorek.

```
int funkce(void);
// ...
funkce (); // je volání funkce
funkce; // ukazatel na funkci
```

Proto je při volání funkce bez parametrů důležité psát prázdné kulaté závorky. Poslední řádek výpisu ve skutečnosti nic nedělá. Takový výraz se při překladu odstraní, protože je zbytečný. Ukazatel na funkci lze použít pro předání určité konkrétní implementace funkce do obecného algoritmu. Jako příklad poslouží funkce qsort()ze souboru stdlib.h.

```
void qsort(void *base, size_t n, size_t size,
int(*compar)(const void *, const void *));
```

Tato funkce vyžaduje tzv. komparátor, tedy funkci, která implementuje srovnání dvou prvků ve struktuře, na níž ukazuje ukazatel base.

2.4 VÍCENÁSOBNÉ UKAZATELE ANEB UKAZATELE NA UKAZATELE

Jelikož ukazatel je proměnná jako každá jiná, lze získat adresu ukazatele. Mluvíme pak o ukazateli na ukazatel. Jak takový vícenásobný ukazatel vypadá?

```
int a = 2;
int *uka = &a;
int **ukuka = &uka;
int b = *(*ukuka);
```

Ukazatel na ukazatel int je ukazatel typu int*. Proto se nám v deklaraci objeví dvě hvězdičky. S podobnou konstrukcí jsme se setkali u funkce main (), jejíž druhý parametr je typu char**. Je to tedy ukazatel na ukazatel na char, zde se jedná o pole ukazatelů na řetězce znaků. Vícenásobné ukazatele se používají nejčastěji u vícerozměrných datových struktur, jako jsou matice nebo obecně pole polí. Z pohledu práce se nijak neliší od běžných ukazatelů, jen musíme provést vícenásobnou dereferenci, abychom se dostali k uloženým datům.

2.4.1 Pole ukazatelů.

Potřebujeme-li pole konstantních řetězců, je možno deklarovat pole ukazatelů na char a inicializovat ho:

```
const char *tab[]={"nula","jedna","dva","tri","ctyri"};
```

tab[0] nyní představuje řetězec "nula" apod. Výhodou proti dvourozměrnému poli char je to, že jednotlivé řetězce mohou mít různé délky.

2.5 DYNAMICKÁ ALOKACE PAMĚTI

V jazyce C se rozlišují dva typy dat. Jsou to staticky alokovaná data a dynamicky alokovaná data. V tomto článku si vysvětlíme rozdíl mezi statickou a dynamickou alokací a též si uvedeme funkce pro práci s pamětí.

2.5.1 Motivace

Až doteď jsme byli schopni pracovat s daty, o kterých jsme dopředu věděli, jak jsou rozsáhlá. Pokud jsme chtěli zpracovávat text po řádkách, museli jsme mít a priori znalost maximální délky řádku. Důvodem bylo, že jsme museli řádku uchovat v paměti a vytvořené pole mělo

velikost danou přímo v programu (byla pevně daná už před překladem). Tento problém částečně řeší lokální nekonstantní pole zavedené ve standardu C99. Nicméně takto vytvořené pole nelze měnit v době běhu programu, jeho délka je neměnná.
Proměnná či pole vytvořená v době překladu se nazývá staticky alokovaná proměnná. Její velikost zná překladač předem. Výhoda takového přístupu je prvně jeho rychlost a též bez obslužnost – při definování proměnné se vyhradí data a při opuštění příslušného bloku kódu se paměť automaticky uvolní. Tím, že se místo vyhrazuje pouze posunem ukazatele na zásobníku programu, máme zaručenu konstantní složitost takové alokace.
Dále v textu se budeme zabývat tzv. dynamicky alokovanými daty, tedy daty, jejichž místo v paměti je vyhrazeno až při běhu programu. Výhodou tohoto přístupu je velká flexibilita – máme kompletní kontrolu nad délkou alokace a můžeme ji dodatečně ovlivnit. Nevýhodou je relativní náročnost tohoto typu alokace, kde se musí projít halda (kde se dynamicky alokované proměnné ukládají) a vyhledat v ní příslušně dlouhý úsek spojité paměti. Správa paměti je přenechaná plně na programátorovi, který musí zajistit správné zacházení se zdroji v paměti, jinak může dojít k pádům programu nebo k vyčerpání volné paměti.

2.5.2 Vyhrazení a uvolnění paměti

Pro alokaci úseku paměti máme k dispozici několik funkcí z hlavičkového souboru stdlib.h:

void* malloc(size_t velikost)	**Vyhradí velikost bajtů paměti a vrátí ukazatel na první prvek. Ukazatel je bez doménového typu, je tedy třeba ho přetypovat na typ, který požadujeme.**
void* calloc(size_t nclenu, size_t velikost)	Vyhradí pole o nclenu členech, kde každý má velikost velikost bajtů. Alokované pole je vyplněno binárními nulami. Tato funkce je pomalejší než malloc(). Funkce vrátí ukazatel na první prvek a stejně jako u malloc() je vhodné přetypování.
void* realloc(void *pole, size_t velikost)	Realokuje pole pole. Funkce vytvoří nové pole o velikosti velikost bajtů a překopíruje do něj obsah původního pole, které muselo být dříve vytvořeno pomocí jedné z funkcí malloc(), calloc()nebo realloc(). Funkce vrátí ukazatel na první prvek nového pole. Nové pole může ležet na jiném místě v paměti než původní pole, původní ukazatel by se tedy již po zavolání neměl používat (zdrojové pole bude automaticky uvolněno). Je-li velikost menší, než je délka původního pole, bude v novém poli pouze fragment původního obsahu. Bude-li větší, nově vyhrazené místo nebude inicializováno. Nastavíme-li velikost na nulu, bude zdrojové pole uvolněno.

Velikost je udávaná v bajtech. Pokud chceme vytvořit pole typu int o deseti položkách, můžeme použít operátor sizeof()vysvětlený v sekci o ukazatelích. Výsledná velikost bude 10*sizeof(int).
Každá z výše uvedených funkcí vrátí ukazatel na první prvek alokovaného pole. Pokud ukazatel ppřepíšeme nebo pokud pozbude platnosti, zůstane místo vyhrazené, ale k vyhrazené paměti se již nedostaneme. Mluvíme pak o úniku paměti (memory leak), kdy se neuvolní alokované místo a zabírá donekonečna zdroje. Takové chování může u programů, které běží dlouho, způsobit vyčerpání volných zdrojů nebo nadměrné využívání odkládacího prostoru a

tedy zpomalení reakcí počítače. Řádné uvolnění paměti se provede voláním následující funkce:

Funkce	Popis
void free(void *pole)	Bezpečně uvolní zabrané zdroje. Pole pole muselo být dříve vytvořeno voláním jedné z výše uvedených funkcí. Předáme-li parametr NULL, funkce nic neprovede.

Pokusíme-li se uvolnit paměť dvakrát, obdržíme chybu segmentace a program bude typicky ukončen.

Ke každé alokaci by správně mělo příslušet i volání free(). Toto bývá ve složitějších programech často velmi náročné, a proto existují nástroje, které odhalí neuvolněnou paměť.

Pro vyplnění vyhrazené paměti určitou hodnotou slouží následující funkce:

Funkce	Popis
void *memset(void *zdroj, int vzor, size_t pocet)	Vyplní počet prvků pole zdroj vzorem uloženým v parametru vzor. Vrátí ukazatel na vyplněné pole zdroj.

2.6 ALOKACE VÍCEROZMĚRNÉHO POLE

Dynamicky alokované vícerozměrné pole lze vytvořit dvěma různými způsoby. Prvním a jednodušším postupem je vytvoření serializované verze vícerozměrného pole. Alokaci si vysvětlíme na příkladu vytvoření matice 3x3:

```
int i, j;
// jednorozměrné pole velikosti
int *matice3x3 = (int*)malloc(3*3*sizeof(int));
for (i = 0; i < 3; i++)
{
for (j = 0; j < 3; j++)
{
// vyplnění matice daty
// indexuje se jedním indexem
matice3x3[i*3+j] = i + j;
}
}
// ...
free(matice3x3);
```

Tento kód je jednoduchý, jeho jedinou nevýhodou je nutnost uchovávat velikost řádky pole, abychom mohli prvním indexem skákat přes celé řádky. Navíc tento přístup umožní operačnímu systému před načítat data, která jsou uchována v řadě za sebou, tudíž je přístup do paměti rychlejší. Příkaz

```
matice3x3[i*3+j] = i + j
```

nahrazuje indexový přístup do pole, číselně by odpovídal výrazu matice3x3[i][j] pokud by se jednalo o pole deklarován jako

```
int matice3x3[3][3];
```

Díky tomu, že ale nelze zaměňovat ukazatele int* int** nelze tento zápis použít.

Druhý přístup se velmi často vyskytuje v učebnicích o programování. Přesto se jedná o horší z obou přístupů. To proti, že konstrukce a uvolnění pole je komplikovanější, a navíc vlivem nesouvislosti dat je přístup do pole pomalejší. Výhodou je, že můžeme místo pointerové aritmetiky použít indexování.

```
int i, j;
// vytvoření přístupového vektoru
int **matice3x3 = (int**)malloc(3*sizeof(int*));
for (i = 0; i < 3; i++)
{
// vytváření jednotlivých řádek
matice3x3[i] = (int*)malloc(3*sizeof(int));
for (j = 0; j < 3; j++)
{
// vyplnění matice daty
// indexuje se klasicky dvěma indexy
matice3x3[i][j] = i + j;
}
}
// ...
for (i = 0; i < 3: i++)
{
//uvolnění řádek matice
free(matice3x3[i]);
// uvolnění přístupového vektoru
free(matice3x3);
```

Základem tohoto postupu je vytvoření tzv. přístupového vektoru, tedy pole ukazatelů na jednotlivé řádky. V cyklu pak naalokujeme jednotlivé řádky, které jsou již klasická jednorozměrná celočíselná pole. Indexace pak probíhá tak, jak jsme zvyklí u staticky alokovaných polí přes dva indexy (nemusíme si pamatovat velikost řádky). Uvolňování pole probíhá v opačném pořadí – nejprve uvolníme řádky a poté přístupový vektor.

2.7 STRUKTURY A VÝČTOVÝ TYP

Pro uchování množiny různorodých dat slouží struktury. Pro zacházení s množinou identifikátorů slouží enumerace. Nízko úrovňový kód zase využije unie. V tomto článku se zmíníme o všech těchto konstrukcích.

2.7.1 Výčtový typ enum

V programování se často stává, že máme proměnnou, která může nabývat jen přesně daného množství hodnot. Příkladem může být vykreslování, kde proměnná barva bude nabývat hodnot modra, cervena nebo zelena. Jednou z možností je používat číslo typu int a v dokumentaci vyjmenovat povolené hodnoty proměnné barva. Tento způsob není příliš praktický, neboť když pak narazíme na řádek kódu barva = 2, nemáme ponětí, která barva odpovídá kódu dva, dokud neprostudujeme dokumentaci. Můžeme místo toho vytvořit množinu konstant celočíselných konstant:

```
const int modra = 0;
const int cervena = 1;
const int zelena = 2;
const int blika = 3;
const int blika_rychle = 4;
// ...
barva = modra;
rychlost = rychle;
```

Tento zápis je už srozumitelnější, nicméně stále můžeme napsat:

```
barva = rychle;
```

Abychom se těmto problémům vyhnuli, lze využít výčtový typ enum.

2.7.2 Deklarace enum

Deklarace má následující syntaxi:

```
enum <[jméno typu]> {<výčet hodnot>}<[deklarace]>;
```

Začínáme klíčovým slovem enum, za kterým následuje nepovinné jméno nového výčtového typu. Ve složených závorkách je výčet možných hodnot oddělených čárkami. Jako poslední položka před středníkem je nepovinný seznam deklarací oddělený čárkami. V našem ovladači bychom tedy měli následující dva výčtové typy:

```
enum barva { modra, cervena, zelena} barvy;
enum blik
{blika = 3, blika_rychle} blikani;
```

Vytvořili jsme dvě proměnné: barvy typu enum barva a blikani typu enum blik.
Ve druhé deklaraci je zajímavé použití přiřazení hodnoty tři. Jelikož jsou výčty číselné množiny, každý člen je interně reprezentován číslem z nejmenšího rozsahu, který pokryje všechny prvky výčtu. Pokud žádné číslo nezadáme, jako je tomu v případě výčtu barva, začne překladač přiřazovat jednotlivým konstantám výčtového typu přiřazovat hodnoty od nuly a každá další položka dostane o jedničku vyšší hodnotu. V případě výčtu blik se začnou přiřazovat hodnoty tři a čtyři (překladač inkrementuje poslední zadanou hodnotu). Číselné hodnoty se ve výčtu mohou opakovat.
Pokud budeme chtít deklarovat proměnnou výčtového typu později, provedeme to následovně:

```
enum barva moje_baryvy;
```

Pokud se chceme opisování enum vyhnout, můžeme využít definici vlastního datového typu pomocí klíčového slova typedef.

2.7.3 Deklarace s použitím typedef

```
typedef <starý typ> <nový typ>;
```

Klíčové slovo typedef umožňuje pojmenovat nový typ, který se skládá z již existujících typů. Například je možné vytvořit typ ukazatel_na_int, který bude ve skutečnosti ukazatelem na hodnotu typu int.

```
typedef int* ukazatel_na_int;
//...
int a = 3;
ukazatel_na_int pa = &a;
Pojmenovaný výčet můžeme vytvořit tímto zápisem:
typedef enum {modra, cervena, zelena } barvy;
barvy moje_barva;
```

2.7.4 Přetypování na int

Jelikož jsou prvky výčtu ve skutečnosti čísla, můžeme je implicitně přetypovat na číselnou hodnotu. V jazyce C je možné i opačné přetypování a proměnné tak lze přiřadit i hodnotu mimo rozsah výčtu.

```
typedef enum {modra, cervena, zelena } barvy;
barvy moje_barva;
int a = modra; //
moje_barva = 4; // ok v C, chybně v C++
moje_barva = ( barvy)2; // explicitní přetypování
```

2.7.5 Porovnávání výčtových proměnných

Výčtové proměnné se při porovnávání chovají jako číselné hodnoty. V jazyce C jde bez problému porovnávat hodnoty z jiných výčtů.

2.8 STRUKTURA

Struktury jsou kontejnery pro heterogenní data. Srozumitelněji řečeno, jedná se společné pojmenování jednotlivých položek různých datových typů. Tak jako pole obsahuje prvky stejného typu, struktury se mohou skládat z položek různých datových typů. Typickým příkladem použití struktury je zápis adresy konkrétní osoby, jako je jméno, město, ulice, číslo, PSC.

2.8.1 Deklarace struktury

Deklarace struktury má následující syntaxi:

```
struct <[jméno typu]> {<výčet členů>}<[deklarace]>;
```

Od deklarace výčtu se liší pouze použitým klíčovým slovem a také výčtem členů. Výše uvedený záznam
adresáře by se uchoval ve struktuře:

```
truct adresa
{
char jmeno[30];
char mesto[20];
char ulice[20];
int cislo;
int PSC;
} Pepa;
```

Pro deklaraci s klíčovým slovem typedef a bez něj platí stejná pravidla jako u výčtového typu. Uvnitř
struktury může být jakýkoliv typ včetně ukazatelů na strukturu samotnou nebo jiné vnořené struktury.

2.8.2 Inicializace struktury

Struktura se inicializuje podobně jako pole, tedy výčtem hodnot oddělených čárkami ve složených závorkách. V případě vnořených struktur inicializujeme vnitřní strukturu dalším vnořeným párem složených závorek. Pokud vynecháme některé hodnoty, překladač zbytek doplní nulami stejně jako u polí.

```
struct adresa Pepa = {"Pepik","Nova Paka","Zahumenni", 28, 30100};
struct adresa Tonda = {"Tonda ","Pha " };
struct adresa Nikdo = {};
```

2.8.3 Přístup ke členům struktury

Pro přístup ke členům struktury slouží operátor tečka. Máme-li našeho zaměstnance Pepu, změníme mu věk následovně:

```
Pepa.cislo = 25;
```

Pokud máme pouze ukazatel na strukturu, máme dvě možnosti, jak se dostat ke členským proměnným. Buď ukazatel nejprve dereferencujeme a poté použijeme operátor tečka, nebo použijeme přímo operátor nepřímého přístupu, který provede prvně zmíněné kroky interně.

```
struct adresa *pk = &Pepa;
(*pk).cislo = 20;  // použití dereference
pk->PSC = 3100; //  pohodlnější,
//ekvivalentní dereferenci a použití operátoru přímého přístupu
```

```
// ke složce struktury tečka
```

2.8.4 Kopírování a porovnávání

Proměnné struktury stejného typu lze jednoduše kopírovat (na rozdíl od polí). Stačí pouze přiřadit jednu proměnnou do druhé a automaticky dojde ke zkopírování všech členů struktury.

```
// Franta a Tonda se stanou jednou osobou se stejnou adresou
Pepa = Tonda;
```

Problém nastává, když máme ve struktuře dynamicky alokovaná data. Při kopírování se zkopírují pouze ukazatele, ale data zůstanou sdílená (jde o tzv. mělkou kopii). V takovém případě se o zkopírování dat musíme postarat sami.

Pokud chceme proměnné struktur porovnávat, musíme tak učinit člen po členu, nelze jednoduše napsat:

```
if (Pepa == Franta) //...
```

2.9 UNIE

Unie se používají ve velmi specifických případech v nízko úrovňovém kódu a běžně se s nimi nesetkáme.

Syntaxí se velmi podobají strukturám, jen místo klíčového slova struct se použije union. Jediným rozdílem od struktur je to, že v daný okamžik existuje v paměti pouze jeden člen unie. Unie má tak velikost odpovídající prostorově největšímu typu v unii, mění se pouze interpretace dat.

```
union ZnakCislo
{
int cislo;
char znak;
} unie;
// unie bude mít velikost rovnou sizeof(int)
// ...
// nyní bude v unii číslo
unie.cislo = 65;

/*
Dotazujeme se na data v unii jako na znak. Hodnota ve znaku závisí na rozložení dat
v paměti (little nebo big endian). Dostaneme buď nejvýznamnější, nebo nejméně významný
bajt čísla. Vyhodnocení podmínky tak závisí na architektuře stroje.
*/
if (unie.znak == 'A') //...
```

2.10 SOUBORY

Soubory patří k základním datovým prvkům v počítači. Převážná většina programovacích jazyků má podporu určité formy práce se soubory. V jazyce C k tomuto účelu slouží funkce fopen(), fclose(), fscanf(), fprintf(),getc(), putc()a další.

2.10.1 Otevření souboru

Se soubory se v každém systému pracuje trochu jinak. Například v systému Windows jsou cesty oddělovány zpětnými lomítky a disk se značí velkým písmenem. V systémech odvozených od UNIXu se adresáře oddělují obyčejnými lomítky a pojmenování diskových oddílů písmeny neexistuje. Každý systém má také jinou filozofii oprávnění. Standard jazyka

C poskytuje jistou formu abstrakce v podobě funkce fopen()z hlavičkového souboru stdio.h, která otevře soubor pro manipulaci.
Funkce má následující syntaxi:

```
FILE *fopen(const char *cesta, const char *způsob);
```

Funkce přejímá dva parametry, oba jsou řetězce. První řetězec je cesta k souboru, Funkce přejímá dva parametry, oba jsou řetězce. První řetězec je cesta k souboru, ke kterému si přejeme přistoupit. Formát cesty je závislý na cílové platformě. Způsob je kód operace, kterou si přejeme se souborem provést. Dostupné způsoby práce jsou:

Způsob	Popis
"r"	Otevře soubor pro čtení. Zápis do souboru skončí chybou. Soubor musí existovat.
"w"	Vytvoří prázdný soubor pro zápis. Pokud soubor existuje, bude jeho obsah nejprve vymazán.
"a"	Otevře soubor pro zápis na konec. Soubor je vytvořen, pokud neexistuje. Veškeré zápisy probíhají na konci existujícího obsahu
"r+"	Otevře soubor pro čtení i zápis. Soubor musí existovat.
"w+"	Vytvoří prázdný soubor pro čtení i zápis. Pokud soubor existuje, bude jeho obsah nejprve vymazán.
"a+"	Otevře soubor pro čtení kdekoliv a zápis na konec. Soubor je vytvořen, pokud neexistuje. Veškeré zápisy probíhají na konci existujícího obsahu, i když předtím čteme na jiném místě souboru.

Kromě výše uvedených způsobů manipulace existuje ještě varianta s písmenem b, které lze umístit na konec řetězce nebo před znak +. Takový soubor bude chápán jako binární soubor, kdežto výše uvedené způsoby otevírají soubor jako textový. Na systémech odvozených od UNIXu není ve způsobu zacházení s textovými a binárními soubory žádný rozdíl, přesto se písmeno b doporučuje přidat kvůli přenositelnosti.
Funkce vrací ukazatel na strukturu FILE. Pokud dojde k chybě, vrátí funkce NULL, jinak obdržíme ukazatel na strukturu, v níž jsou uchované některé důležité údaje nutné pro práci se souborem. Tyto informace jsou ovšem před programátorem skryty, neboť se mohou systém od systému lišit.

```
FILE *f, *g;
// otevření konfiguračního souboru pro čtení UNIXu
f = fopen("/etc/X11/xorg.conf", "r");
/* otevření konfiguračního souboru pro čtení v systému Windows je nutné psát dvě zpětná lomítka, jinak bude následující znak chápán jako řídící sekvence
*/
g = fopen("C:\\boot.ini", "r");
```

2.10.2 Ostatní funkce pro práci se souborem

Abychom mohli pracovat se souborem, je nejprve nutné ho otevřít pomocí výše uvedené funkce fopen() a získat ukazatel na strukturu FILE. Poté se se souborem pracuje podobně jako se standardním vstupem a výstupem (což není náhoda). Následuje seznam několika důležitých funkcí pro manipulaci se souborem:

Funkce	Popis
int fclose(FILE *f)	Bezpečně uzavře předem otevřený soubor. Každý otevřený soubor by měl být řádně ukončen. Funkce vrací EOF(End Of File), nastane-li chyba.
int fscanf(FILE *f, const char *maska, ...)	Tato funkce funguje naprosto stejně jako funkce scanf()s tím rozdílem, že zdrojem dat je otevřený soubor f.

int fprintf(FILE *f, const char *maska, ...)	Tato funkce funguje naprosto stejně jako funkce printf()s tím rozdílem, že cílem je otevřený soubor f.
int fgetc(FILE *f)	Funkce přečte ze souboru právě jeden znak a přesune se na další. Pokud nastane chyba, vrátí funkce EOF.
int fputc(int c, FILE *f)	Funkce vypíše právě jeden znak do souboru. Pokud nastane chyba, vrátí funkce EOF.
long ftell(FILE *f)	Vrátí aktuální pozici v souboru v bajtech.
int fseek(FILE *f, long posun, int start)	Přesune se na pozici v souboru. Cíl je určen relativním posunem o posun bajtů vůči startovní pozici start. Startovní pozice může být SEEK_SET (začátek souboru), SEEK_CUR (aktuální pozice), SEEK_END (konec souboru). Funkce vrací mínus jedna, pokud nastane chyba.
int fflush(FILE *f)	Vyprázdní vyrovnávací paměť a zapíše veškerý dosud nezapsaný obsah do souboru. Tato funkce by se měla volat mezi po sobě jdoucím čtením a zápisem nebo zápisem a čtením do souboru. Pokud nastane chyba, vrátí funkce EOF.
FILE *tmpfile(void)	Vytvoří soubor s unikátním jménem tak, aby nedošlo ke jmenné kolizi. Soubor bude automaticky smazán po uzavření.

Jednoduchý program, který vypíše obsah předaného souboru, bude tedy vypadat takto:

```
#include <stdio.h>
int main (int argc, char **argv)
{
// byl předán název souboru?
if (argc < 2) return -1;
// otevřeme soubor, relativní cesta se vyhodnocuje k pozici
// spuštěného programu
FILE *f = fopen(argv[1], "r");
// povedlo se otevřít?
if (f == NULL) return -1;
int c;
// přečti obsah znak po znaku a vypiš
while ((c = fgetc(f)) != EOF)
{
putc(c);
}
// uzavřeme soubor
fclose(f);
return 0;
}
```

Všimněme si nyní, že se pro manipulaci se znakem používá typ int a nikoliv char, jak bychom očekávali. Je to z toho důvodu, že funkce fgetc()signalizuje chybu pomocí zvláštní návratové hodnoty EOF(End Of File). Kdyby funkce vracela znaky, nezbyl by pro tuto konstantu žádný kód. Proto je nutné rozšířit návratový typ na int, kde už EOF existovat může.

2.11 STANDARDNÍ VSTUP A VÝSTUP JAKO SOUBORY

Ač se to může zdát zvláštní, standardní vstup a standardní výstup nejsou nic jiného než soubory a jazyk C s nimi ani jinak nepracuje. Standard pouze poskytuje funkce, které práci s nimi usnadňují, můžeme s nimi ale pracovat pomocí souborových funkcí. Před spuštěním našeho kódu dojde k automatickému otevření tří souborů – stdin, stdout a stderr. Tyto soubory jsou na konci programu automaticky uzavřeny.
Soubor stdin je otevřen pro čtení a obsahuje vstup programu, soubor stdout je otevřen pro zápis a obsahuje výstup programu. Konečně soubor stderr je soubor otevřený pro zápis a obsahuje chybová hlášení programu. Výstup a výpis chyb tak lze oddělit. Pokud chceme vypsat hlášení na standardní chybový výstup, můžeme použít funkci fprintf()s prvním parametrem stderr.

2.12 PREPROCESOR

Preprocesor umožňuje ovlivnit kód před jeho vlastním překladem. Nachází uplatnění v multiplatformním kódu a při spojování větších projektů.

2.12.1 První fáze překladu

Direktivy preprocesoru jsou vždy uvozeny dvojkřížkem (#). Vše od tohoto znaku až do konce řádky je chápáno jako příkaz preprocesoru. Pokud potřebujeme uvést delší příkaz, který by zabral několik řádek, můžeme příkaz zalomit použitím zpětného lomítka na konci řádky. Slovo preprocessing znamená předzpracování. Tak tomu ve skutečnosti opravdu je. Překladač totiž nejprve zpracuje veškeré příkazy preprocesoru a až poté začne s fází překladu. Preprocesorem se dá ovlivnit kód několika způsoby: vyjmutím části kódu, nahrazení části kódu a vložení kódu z externího souboru.

2.12.2 Připojení souboru

Už v prvním programu jsme využili preprocesor. Příkazem

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

jsme preprocesoru řekli, že na toto místo v kódu si přejeme vložit obsah souboru stdio.h. Obecně můžeme pomocí příkazu include vložit na jakékoliv místo v programu obsah jiného souboru. Nicméně musíme být opatrní, abychom některý soubor nevložili víckrát, protože preprocesor toto nekontroluje a provede jen vložení. Při následném překladu překladač vyhodnotí, že kód obsahuje duplikátní deklarace.
V jazyce C se vkládají pouze hlavičkové soubory. Soubory se zdrojovým kódem je nesmyslné vkládat, neboť překladač potřebuje znát pouze deklarace použitých funkcí, nikoliv jejich těla. Příkaz include má dvě varianty.
Ta první používá špičaté závorky okolo jména souboru, ta druhá používá uvozovky. Význam se liší podle toho, v jakých adresářích se hledají příslušné soubory. Špičaté závorky jsou vyhrazeny pro systémové hlavičky a hledají se běžně v systémových adresářích (nebo v adresářích uvedených v parametrech překladače).
Naproti tomu soubory uvedené v uvozovkách se hledají relativně k umístění aktuálně překládaného souboru.

```
#include <stdio.h>
// vloží obsah hlavičky projektu
#include "mojehlavicka.h"
// ...
/*
opětovné vložení by mohlo způsobit problémy
v tomto případě se ale nic nestane z důvodu
vysvětleného níže
*/
#include <stdio.h>
```

Máme-li rozsáhlejší projekt, není mnohdy snadné uhlídat pořadí vkládání jednotlivých hlavičkových souborů. Velmi snadno se nám může stát, že nám vkládání zacyklí či že vložíme jednu hlavičku dvakrát. Abychom se tomuto vyhnuli, používá se standardní konstrukce, která zabrání vícenásobnému vložení.

2.12.3 Podmíněný překlad

Podmíněný překlad je mocná technika, která umožňuje adaptovat kód v závislosti na prostředí, v němž je kód překládán. Nejde o nic jiného než o podmínky preprocesoru, které umí vypouštět bloky kódu.
Struktura příkazu se podobá příkazu if

```
#ifdef <makro> nebo #ifndef <makro> nebo #if <podmínka>
// ..
#else nebo #elif <podmínka>
// ...
#endif
```

Příkaz funguje následovně. Nejprve se vyhodnotí podmínka: v prvních dvou případech se ptáme, zdali byla nebo nebyla dříve definovaná proměnná preprocesoru (makro), ve třetím případě se vyhodnocuje podmínka.
Podmínka musí být vyhodnotitelná v době překladu a smí obsahovat pouze makra, nikoliv konstanty nebo proměnné z kódu. Je-li podmínka vyhodnocena jako pravda, vloží se do kódu blok programu až do else, elif nebo endif, pokud není uvedeno větvení. Je-li podmínka vyhodnocena jako nepravda, vloží se druhý blok kódu (je-li uveden).
Makro můžeme definovat několika způsoby. Tím nejběžnějším je použít příkaz define, který zruší předem definované makro.

```
/*
makra se běžně pojmenovávají velkými písmeny
tento příkaz vytvoří nové makro PROMENNA bez přiřazené
hodnoty
*/
#define PROMENNA
// nové makro PI s hodnotou 3.1415, středník se nepíše!!
#define PI 3.1415
// definice funkčního makra
#define abs(x) (((x)>=0)?(x):-(x))
Opakem příkazu define je příkaz
#undef <existujici-makro>,
```

Makra se používají jako substituční členy v programu a jako řídící proměnné. Poslední uvedená definice definuje funkční makro, které se chová jako funkce (za jménem makra jsou kulaté závorky). Zde definujeme absolutní hodnotu. Důvod, proč používáme tolik závorek v těle funkčního makra, je ten, že ve fázi předzpracování se neřeší priorita operátorů, což znamená, že bez závorek se může výraz vyhodnotit chybně.

V příkladu výše jsme vytvořili makro PI. Kdekoliv v kódu ho tak můžeme použít jako obyčejnou konstantu.

```
float y = sin(PI*x);
```

Před vlastním překladem se totiž všechna makra rozvinou, což znamená, že se všechny výskyty maker nahradí jejich hodnotami. Substituce probíhá na úrovni textu kódu, nedochází k žádné kontrole typů.
Proto je možné napsat následující chybný kód.

```
#define MAKRO "abc"
int a = MAKRO;
```

Zde si bude překladač stěžovat na nekompatibilitu typů, která se schovává v použití makra. Proto se doporučuje pro konstantní hodnoty využívat konstant namísto maker. Obecně chyby způsobené špatným rozvinutím makra jsou někdy obtížně dohledatelné, protože překladač si bude stěžovat až u substituovaného kódu.
Zabránění vícenásobnému vložení hlavičky
Jedním z možných řešení výše popsaného problému je použití následující konstrukce.

```
#ifndef SPECIFICKE_JMENO_PRO_HLAVICKU
#define SPECIFICKE_JMENO_PRO_HLAVICKU
// kód hlavičky
#endif
```

Při prvním vložení ještě neexistuje uvedené makro, a tak se hlavička vloží. Při dalším pokusu už makro existuje, a tudíž se celý kód hlavičky přeskočí.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Počítačové řízení

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

3 POČÍTAČOVÉ ŘÍZENÍ 3

3.1 Počítačové řízení 4

3.2 Prvky počítačového řídicího systému 4

3 POČÍTAČOVÉ ŘÍZENÍ

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat prvky počítačového řídicího systému

3.1 POČÍTAČOVÉ ŘÍZENÍ

Nejstarší návrh použití počítače pracujícím v „reálném čase“ jako část řídicího systému pochází z roku 1950 od Browna a Campbella. Jejich návrh je znázorněn na následujícím obrázku. Brown a Campbell předpokládali spíše požití analogových , než číslicových prvků.
První číslicový počítač vyvinutý speciálně pro řízení v reálném čase byl použit pro řízení leteckých operací. Tímto počítačem byl číslicový počítač Digitrac v roce 1954.
První průmyslová aplikace v roce 1958 v Lousianě, kde byl užit počítač Daystrom pro monitorování procesů výroby el. energie v elektrárně. To však byl pouze systém monitorování, nikoli řídicí. Prvním řídicím systémem, realizujícím řízení v uzavřené smyčce, byl v roce 1959 systém RW-300 (Ramo-Wogldridge Company) v rafinérii společnosti Texaco Company v Texasu.
Prvním systémem přímého číslicového řízení (DDC – Direct Digital Control) byl systém Ferranti-Arguo 200 instalovaný v továrně ICI na výrobu amonia-soda ve Fleetwodu v Lancashire. To již byl velký systém se 120 regulovanými okruhy a 256 měřenými veličinami.
Počítače užívané pro řízení počátkem šedesátých let, užívaly kombinace magnetické feritové paměti a bubnové paměti, případně s možností přepisu bubnu na diskovou paměť. Nejznámější systémy té doby jsou General Eletric 4000, IBM 1800, CDC 1700, Foxboro Fox 1 A a 1A, SDS a Xerox SIGMA, Ferranti Arguo a Elliot Automation 900.
Požadavky na spolehlivost těchto systémů vedly ke zvýšení jejich ceny, takže v tehdejších aplikacích byl použit většinou jen jeden počítač vykonávající jak funkce DDC tak monitorování. To vedlo k problémům při vývoji programového vybavení, které bylo zpočátku prováděno v assembleru (kvůli krátkosti a rychlosti kódu). Rozsah kódu neustále narůstal. T9m se vývoj programového vybavení stával čím dále tím hůře kontrolovatelný a řiditelný. Nemožnost umístit celý kód do paměti si vyžádal vývoj technik výměny obsahu operační a záložní paměti (swapping) a realizaci kompilovaných technik segmentace programu.
To vše vytvořilo velký tlak na vývoj obecně konstruovaných operačních systémů a programových jazyků podporující vývoj a implementaci aplikací řízení v reálném čase.
Byly to především různé verze kompilátorů FORTRANu vyvinuté pro účely řízení koncem šedesátých let společně s operačními systémy reálného času.
Tehdy ještě relativně vysoká cena počítačů a zcela specifické požadavky na jejich programové vybavení diktované růzností jednotlivých technologických aplikací pak vedly k vývoji menších, dle přání zákazníka konfigurovatelných systémů, tzv. minipočítačů. Mezi nimi vynikaly tehdy, i dnes známé, DEC PDP-8, PDP-11, Data General Nova, Honywell 316, HP2116. Modulární struktura spolu s rozšířením spektra aplikací vedly ke snížení ceny, což již umožňovalo ekonomicky více počítačů v jedné aplikaci, ať již byly úlohy distribuovány na jednotlivé systémy nebo prostě jeden z počítačů byl neustále připraven jako záložní počítač pro případ havárie. Zde je tedy zárodek distribuovaných systémů, které přicházejí na scénu po příchodu mikroprocesoru v roce 1974.

3.2 PRVKY POČÍTAČOVÉHO ŘÍDICÍHO SYSTÉMU

Jako příklad si ukážeme jednoduchý příklad ohřívač větru. Větrák fouká vzduch přes ohřívací element do potrubí. Na výstupu z potrubí je umístěn odporový teploměr a vytváří jedno rameno mostu.
Zesílený výstup z tohoto místa je dostupný v bodě B a poskytuje napětí v rozmezí 0-10 V úměrné teplotě. Proud dodávaný do ohřívacího elementu může být měněn změnou napětí v rozsahu 0-10V v bodě A.
Pozice klapy regulující množství vzduchu vstupující do ohřívače je měněna pomocí reverzibilního motoru. Motor pracuje konstantní rychlostí a je vypínán a zapínán logickým signálem připojeným na jeho řízení. Druhý logický signál určuje směr rotace. Ke vstupní

klapě je připojen potenciometr, takže napětí na něm je úměrné pozici klapy. K detekci úplného otevření a uzavření jsou použity mikrospínače.
Operátor má k dispozici panel, na němž může být nastaveno řízení ruční nebo automatické. V ručním režimu může být výstup tepla a pozice klapy řízena potenciometrem.
Přepínače jsou nainstalovány proto, aby řídily operace foukání a ohřívání. Světla na povrchu indikují následující stavy“

- fouká se
- topí se
- větrák otevřen
- větrák uzavřen
- auto/manuál status

Řízení operací tohoto jednoduchého zařízení počítačem vyžaduje

- monitorování
- řídicí výpočty
- ovládání

Monitorování zahrnuje získávání informací o aktuálním stavu zařízení. V našem případě jsou tyto informace získávány ve formě analogových veličin pro teplotu vzduchu a pozici vstupní klapy větráku. Ve formě digitálního vstupu pak pro extremní polohy vstupní klapy větráku (plně otevřeno, pně uzavřeno) a jako další stavy signálů:

- auto/manual
- motor běží
- topení topí

Řídicí výpočty představují číslicový ekvivalent zpětnovazebního řízení pro řízení teploty (DDC). Dále je zde zpětnovazební řízení velikosti vstupního otvoru větráku a posloupnosti logických řídicích operací:

- ohřívač by neměl topit neběží-li větrák
- dojde-li ke změně z ručního na automatické řízení, musí se nastavit příznaky pro zapojení řady logických operací jako např. paralelní logické operace, časově následné řízení a časování operací.

Ovládání vyžaduje vytvoření napětí úměrné požadovanému výstupu tepla pro řízení ohřívače a nastavení logických signálů ovládající vypínání a směr natáčení vstupní klapy větráku a logických signálů pro operátorský display.
Monitorování a ovládání tak zahrnuje řadu zařízení realizujících převod údajů na různých rozhraních, jako jsou analogově číslicové a číslicově analogové převodníky, digitální vstupy a výstupy a generátory impulsů.
Vstupy a program jsou cyklicky ošetřovány, a tím je vytvářena průběžně informace o změně stavu zařízení v čase.
Výstupní stav řídicího systému určují hodnoty jeho výstupů, které jsou určeny řídicími výpočty cyklicky prováděného programu.
Úlohy, které jsou součástí počítačového řízení tak mohou být rozděleny do tří hlavních skupin:

- úlohy sběru vstupních dat
- úlohy zpracování vstupních dat a řízení (řídicí algoritmus)
- úlohy zobrazování a zápisu výstupních dat

Komunikace s operátorem je přitom považována za součást vstupních a výstupních úloh. Řada aplikací však nevystačí s jednoduchými vstupy a výstupy. Zejména v rozsáhlejších aplikacích je třeba sbírat a vysílat informace na veliké vzdálenosti a na různá zařízení; často dochází k výměně informací mezi počítači. Také tyto komunikační úlohy musí být řešeny v rámci řídicího systému.

evropský sociální fond v ČR | EVROPSKÁ UNIE | MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY | OP Vzdělávání pro konkurenceschopnost

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Klasifikace RT systémů

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

4 KLASIFIKACE RT SYSTÉMŮ 3

4.1 Klasifikace RT systémů 4

4.2 Systémy řízené časem 4

4.3 Systémy řízené událostmi 4

4.4 Interaktivní systémy 5

4.5 Systémy reálného času . definice 5

4 KLASIFIKACE RT SYSTÉMŮ

OBSAH KAPITOLY:

Popište systémy řízené časem

Popište systémy řízené událostmi

Popište interaktivní systémy

Popište systémy reálného času

MOTIVACE:

Systémy řízené časem

Řídicí počítač musí být pak schopen v reálném čase, tj. během jediné vzorkovací periody, provést měření, jeho zpracování a eventuálně generovat akční zásah.

Systémy řízené událostmi

Velmi často není na počítači požadovaná odezva závislá na čase, ale na nějaké časově nepředvídatelné události, která se udá ve vnějším prostředí.

Systémy reálného času . definice

O systémech reálného času hovoříme tehdy, jestliže:

1. Pořadí výpočtů je dáno tokem času nebo je určují vnější události.
2. Výsledky jednotlivých výpočtů mohou záviset na hodnotě proměnné čas v době provádění výpočtu nebo když jeden z parametrů výpočtu je hodnota okamžitého.

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat systémy řízené časem
- systémy řízené událostmi
- interaktivní systémy
- systémy reálného času

4.1 KLASIFIKACE RT SYSTÉMŮ

Při použití počítačů pro řízení reálných procesů je počítač s okolním prostředím spojen fyzikálními přístroji – čidly, resp. akčními členy. Vnější procesy pracují ve svém vlastním časovém režimu a my říkáme, že počítač pracuje v reálném čase, jestliže akce prováděné počítačem odpovídají tomuto časovému měřítku.
Vztah vůči vnějšímu prostředí může být popsán časem (fyzikální veličinou) – čas, datum, časové intervaly atp., nebo může být definován pomocí událostí (event). Událost může představovat například stlačení tlačítka, sepnutí spínače atp.
Ve druhém případě jde o vztah interaktivní, v němž je vztah mezi akcemi počítače a událostmi vnějšího prostředí nejtěsnější. V tomto případě je o požadavek, aby byla odezva počítače na externí událost provedena v předem stanoveném čase, resp. časovém režimu. Do této kategorie spadá např. většina komunikačních úloh.
Úlohy řízení, ačkoliv nejsou přímo propojeny s externím prostředím, také musí operovat v reálném čase, protože musí vypočítávat potřebné parametry pro realizaci řídicích úloh a realizovat řídicí zásahy.
Posuzujeme-li RT systémy podle toho, jaké externí vlivy určují převažující režim jejich činnosti můžeme je rozdělit do těchto kategorií:

- systémy řízené reálným časem
- systémy řízené událostmi
- interaktivní systémy

V praxi ovšem většinou jen trochu složitější řídicí systém naplňuje v různé míře všechny výše uvedené charakteristiky.

4.2 SYSTÉMY ŘÍZENÉ ČASEM

Různé reálné (tj, fyzikální, technické) systémy pracují v různě náročných časových režimech a reagují na vnější poruchy různou rychlostí, s různou časovou odezvou, které odpovídají časovým konstantám procesu. Hodnota tšcht konstatn mohou být hodiny u jiných procesů se může jednat o milisekundy (řízení letadla). Pro zpětnovazební řízení proto bude vzorkovací frekvence snímání dat procesu odpovídat časové konstantě procesu. Čím menší časová konstanta tím je pro přesné zmapování stavu systému potřebná vyšší vzorkovací frekvence. Řídicí počítač musí být pak schopen v reálném čase, tj. během jediné vzorkovací periody, provést měření, jeho zpracování a eventuálně generovat akční zásah.
Vykonání potřebných operací počítačem během specifikovaného času je závislé jak na množství potřebných operací, tak na rychlosti počítače. Základním technickým prostředkem umožňující synchronizaci akcí počítače s časovými událostmi jsou vestavěné hodiny, nazývané hodiny reálného času (real – time clock), jejichž takt generuje hardwarové přerušení, na jehož základě je možno provést měření času.

4.3 SYSTÉMY ŘÍZENÉ UDÁLOSTMI

Velmi často není na počítači požadovaná odezva závislá na čase, ale na nějaké časově nepředvídatelné události, která se udá ve vnějším prostředí. Může jít například o uzavření ventilu v okamžiku, kdy je naplněna nějaká nádrž, nebo vypnutí motoru, když zařízení dosáhlo určené pozice.
Identifikace stavu, na nějž je třeba reagovat, je pováděna a signalizována určitým senzorem - čidlem a systémy, které se na vstupech využívají čidla, jsou klasifikovány jako systémy založené na senzorech (sensor – based systems). K určení specifikované události obyčejně užívají senzory mechanismu přerušení (interrupt). V některých případech se užívá technika dotazování (pooling) spočívající v cyklickém dotazování na stav senzoru. Součástí

specifikace úlohy je většinou také požadavek , v jakém časovém intervalu musí počítač reagovat na signál senzoru a zajistit odpovídající odezvu.

4.4 INTERAKTIVNÍ SYSTÉMY

Interaktivní systémy pravděpodobně reprezentují nejrozsáhlejší třídu RT systémů. Patří k nim například bankomaty, rezervační systémy, informační systémy nejrůznějšího druhu atp. Požadavek a zpracování v reálném čase je většinou specifikován jako požadavek na průměrnou dobu odezvy. U bankomatu to může být například 20 sekund. I když má tento systém mnoho podobného se systémy reagujícími na sensory, nemusí, na rozdíl od nich, reagovat vždy v čase menším než jisté maximum a garantuje pouze průměrnou dobu odezvy. Skutečná doby odezvy pak závisí na vnitřním stavu počítače, na jeho zatížení zpracováním jiných úloh, což může být dáno např. souběhem většího počtu požadavků v daném okamžiku.
Ve všech třech předchozích případech jsme viděli, že řídicí počítač musí být schopen pracovat v časovém souladu se zařízením, technologií nebo provozem, pro jejichž monitorování nebo řízení je nasazen. Takové řídicí systémy nazýváme systémy reálného času.

4.5 SYSTÉMY REÁLNÉHO ČASU . DEFINICE

O systémech reálného času hovoříme tehdy, jestliže:

1. Pořadí výpočtů je dáno tokem času nebo je určují vnější události.
2. Výsledky jednotlivých výpočtů mohou záviset na hodnotě proměnné čas v době provádění výpočtu nebo když jeden z parametrů výpočtu je hodnota okamžitého.

Systémy reálného času pak musejí reagovat na vnější podněty jedním z uvedených způsobů:

A. Systém musí mít střední dobu odezvy na podnět a jeho zpracování měřenou na definovaném časovém intervalu menší, než je specifikované maximum.
B. Odezva (reakce a zásah) musí být v každé situaci provedena v době kratší než je specifikované maximum

Diskuse.
Druhý typ je obvykle daleko závislejší na výkonnosti výpočetního systému. Je typickou charakteristikou takzvaných vestavných systémů (embleded systems). Jedná se o systémy, kdy je počítač přímo součástí nějakého zařízení. Tyto systémy kladou specifické nároky na návrh hardware i software těchto počítačů.
Typickým příkladem systému s chováním typu A je bankovní automat. Je to systém reálného času, protože je řízen externími událostmi (umístění karty ve stroji inicializuje transakci) a výpočet závisí na časové proměnné (existují denní, resp. týdenní omezení na velikost vybírané částky).
Nyní je třeba upřesnit, co máme na mysli, říkáme-li, že systém je řízený externími událostmi, je typu senzor-based. Ta událost vnikne ve vnějším prostředí, je registrována nějakým čidlem a poté ji zaregistruje počítač, proběhne její zpracování v počítači v podstatě předem známým postupem a tempem, které již není závislé na okolním prostředí. Proč o tom hovoříme?
V jistém smyslu jsou totiž i všechny interaktivní programy řízeny externími událostmi (například stisknutí tlačítka klávesnice). Poté se však zpracování odvíjí tempem, které je dáno tempem následné komunikace s vnějším prostředím. Teprve během této komunikace může dojít k postupnému vytváření zadání.
Například automatizovaný bankovní systém nikdy předem neví, který zákazník přijde, a který bankomat bude aktivován. Jakmile však vloží zákazník kartu do automatu, začne se vůči němu systém chovat jako interaktivní.
Typickým příkladem systému s chováním typu B je řídicí smyčka pro řízení foukání horkého vzduchu, jak bylo popsáno výše. V pojmech teorie řízení je teplotní© smyčka vzorkovací datový systém (sampled data systém), který lze znázornit obrázkem č. ….

Návrh algoritmu řízení vyžaduje výběr vzorkovacího intervalu Tv. Specifikace tohoto intervalu musí být součástí návrhu software realizující řídicí algoritmus. Specifikace by měla říci, že vzorkovací frekvence bude např. 100 Hz (Tv=10 msec). To bude vyžadovat, aby každých 10 ms byla přečtena vstupní hodnota a jejím základě vypočten regulační zásah a zaslán na výstup; tento proces by neměl být přerušen jinými aktivitami.
V praktických případech není většinou žádný řídicí systém čistě prvního nebo druhého typu. V případě ohřívače vzduchu asi občasný výpadek jednoho vzorku nebude vážně ohrožovat provoz zařízení, stejně tak jako mírné variace ve vzorkovací frekvenci, kde např. bude jistě přípustná tolerance 9.950 <=Tv<=10.050 ms, pro který vychází střední doba vzorkovacího intervalu na 10 ms.
Na druhé straně bankomat asi nebude pracovat uspokojivě, jestliže by v jednotlivých případech byla doba obsloužení překračovala neúnosnou míru (řekněne 10 minut), i když by střední doba obsluhy by byla přijatelná. V takovém případě je třeba stanovit požadavky tak, aby například v provozním intervalu 24 hodin byla střední doba odezvy 15 sec s tím, že 95% požadavků bylo uspokojeno během 30 sekund a žádný nepřevýšil dobu obsluhy 60 sekund. V případě systémů druhého typu by však, v případě výstražných signálů (alarmu), podobná formulace časových podmínek byla nevyhovující. Zde musí dojít k obsloužení kritického požadavku dříve, než je dáno určitou maximálně přípustnou hranicí. Je-li například ohřívač vzduchu použit k sušení nějakých součástek, které by mohly být zničeny při teplotě vzduchu větší než 50 °C, bude nezbytné reagovat při dosažení této teploty nejpozději za 10 sekund.
Naštěstí většina systémů řízení pracující v reálném čase, které nemusí pracovat jako real time systémy, tj. v tom smyslu, že nekladou požadavky na rychlost obsluhy, časovou odezvu atp.) což zjednodušuje návrh a implementaci těchto systémů.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Klasifikace řídicích systémů

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

5 KLASIFIKACE ŘÍDICÍCH SYSTÉMŮ 3
5.1 Klasifikace řídicích systémů 4
5.2 Sekvenční typ 4
5.3 Typ multitasking 4
5.4 Programy typu real-time 4

5 KLASIFIKACE ŘÍDICÍCH SYSTÉMŮ

OBSAH KAPITOLY:

Charakterizujte tři základní typy operací (funkcí) potřebných v řídicích systémech, Uveďte příklady.

Jaké vlivy vnějšího (řízeného) okolí určují chrakter činnosti RT systémů a jak můžeme dle toho charakterizovat RT systémy? Uveďte příklady.

Uveďte definici systému reálného času.

Klasifikujte systémy reálného času z hlediska typu odezvy. Uveďte příklady.

Do jakých kategorií můžeme rozdělit řídicí systémy z hledisky časové organizace (návaznosti) jejich funkcí_ Uveďte příklady.

MOTIVACE:

Sekvenční typ

V běžném sekvenčním programu jsou jednotlivé funcke, akce programu uspořádány v jednoznačné časové posloupnosti, takže chování programu závisí výhradně na chování jednotlivých funckí a jejich seřazení.

Typ multitasking

Od klasického sekvenčního typu se liší především tím, že jednotlivé funkce, akce, nemusejí být prováděny v čase celistvě, nepřerušovaně, ale že se jejich provedení časově prolíná. To může být nezbytné pro takové funkce, které mají být prováděny v čase paralelně.

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat sekvenční typ, typ multitasking,
- programy typu real- time

5.1 KLASIFIKACE ŘÍDICÍCH SYSTÉMŮ

Specifikace a oddělení nejrůznějších aktivit (funkcí) vykonávaných řídicím počítačem na ty, které jsou nebo nesou typu úlohou reálného času je důležitá proto, že je podstatně obtížnější konstrukce programů s operacemi typu real-time a operacemi zajišťujícími styk s periferními zařízeními (zejména nestandardními)., než konstrukce standrdních programů např. pro dávkové zpracování dat.
Z hlediska časové organizace akcí můžeme rozdělit řídicí programy do skupin“

a) sekvenční
b) typu multitásking
c) typu real-time

5.2 SEKVENČNÍ TYP

V běžném sekvenčním programu jsou jednotlivé funcke, akce programu uspořádány v jednoznačné časové posloupnosti, takže chování programu závisí výhradně na chování jednotlivých funckí a jejich seřazení.

5.3 TYP MULTITASKING

Od klasického sekvenčního typu se liší především tím, že jednotlivé funkce, akce, nemusejí být prováděny v čase celistvě, nepřerušovaně, ale že se jejich provedení časově prolíná. To může být nezbytné pro takové funkce, které mají být prováděny v čase paralelně. Sekvenční vztah mezi úlohami tím není, resp. nemusí být nijak oslaben, nebo eliminován. Takový program se skládá z částí, tzv. procesů, které samy o sobě, jejich vnitřní struktura, jsou sekvenční. Je zřejmé, že součinnost takových procesů musí být nějak koordinována, synchronizována. Jednou z nejdůležitějších úloh při vytváření programového vybavení pro řídicí systémy je synchronizace programových procesů.

5.4 PROGRAMY TYPU REAL-TIME

Vzhledem k předchozímu přibývá další druh funkcí; těch, jejichž časová posloupnost není určena programátorem, ale událostmi z vnějšího prostředí a jejichž výskyt nemá žádný vztah k právě probíhajícím akcím v počítači.
Takové události není možno podřídit žádným pravidlům meziprocesorové komunikace. Procesy, které mají být aktivovány na základě vnějších událostí, nemohou být odloženy a čekat na základě vnějších událostí, nemohou být odloženy a čekat na nějaké synchronizační signály. Zde je vlastně synchronizačním signálem sám vnější podnět. Proto je validací těchto programů podstatná (na rozdíl od předchozích dvou typů) odezva řídicího systému, t. j. kolik času skutečně trvá provedení jednotlivých funkcí (akcí, procesů).
V důsledku toho bylo třeba vyvinout speciální prostředky pro ošetřování stavů a zajišťování funkcí vyžadovaných programy s vlastnostmi reálného času a zabudovat je do operačních systémů. Klasické programovací jazyky pak vybavit funkcemi umožňujícími využití těchto speciálních funkcí, tzv. RT-funkcí operačních systémů.
Dále byly vyvinuty speciální jazyky, RT- jazyky, které obsahují funkce, prostředky, potřebné pro vývoj RT- programů. Podrobněji o nich bude pojednáno v kapitole o RT-programovacích jazycích.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Pojmy počítačového řízení

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

6 POJMY POČÍTAČOVÉHO ŘÍZENÍ 3
6.1 Charakteristika průmyslových procesů 4
6.2 sekvenční řízení 4
6.3 Jednoduchá řídicí smyčka. 5
6.4 Výpočet žádané hodnoty 5
6.5 Více řídicích smyček 6
6.6 Časově kritické operace. 6
6.7 Složitost řídicích systémů 6
6.8 Programové adaptivní řízení. 7
6.9 Autoidentifikace 7
6.10 Adaptivní řízení na základě modelu 7
6.11 Kombinace zpětné a dopředné vazby. 7
6.12 DCC – přímé číslicové řízení 7
6.13 Monitorování 8
6.14 Komunikace člověk- stroj 9
6.15 Centralizované řízení 9
6.16 Hierarchické 9
6.17 Distribuované 10

6 POJMY POČÍTAČOVÉHO ŘÍZENÍ

OBSAH KAPITOLY:

Popište sekvenční řízení, výpočet žádané hodnoty, časově kritické operace, programové adaptivní řízení, autoidentifikaci, adaptivní řízení na základě modelu, kombinaci zpětné a dopředné vazby, příme číslicové řízení, monitorování, centralizované řízení.

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat sekvenční řízení, výpočet žádané hodnoty, časově kritické operace, programové adaptivní řízení, autoidentifikaci, adaptivní řízení na základě modelu, kombinaci zpětné a dopředné vazby, příme číslicové řízení, monitorování, centralizované řízení

6.1 CHARAKTERISTIKA PRŮMYSLOVÝCH PROCESŮ

Průmyslové a laboratorní procesy, které používají počítače pro řízení nebo monitorování svých operací, monou být klasifikovány jako systémy náležející k jednomu nebo více následujících typů

- dávkové
- spojité
- laboratorní nebo testovací

Dávkovými rozumíme procesy (systémy), v nichž určitá posloupnost operací vyprodukuje určitou kvntitu nebo objem nějakého produktu – tedy dávku – a tato posloupnost je dále opakována a vytváří další dávky. Mezi započetím výroby další dávky je možno specifikovat parametry dále produkované dávky.
Typickým příkladem je válcování plechů. Ingot je protahován válcovací stolicí tak dlouho, až dosáhne požadovaných rozměrů. Další ingot může být jiných parametrů (rozměry, složení), anebo požadujeme odlišné parametry výsledného plechu (rozměry, kvalitu atp.)-
Charakteristickou veličinou dávkového systému je doba nastavení (set – up time, down time, changeover time), což je doba potřebná k přípravě zařízení pro spuštění další dávky. Je to čas, kdy není vytvářen produkt. Důležitým údajem je proto poměr mezi operačním časem (časem během něhož je vytvářen produkt) a časem nastavení pro určení optimální velikost dávky.
Termín spojitý, nepřetržitý, je užíván pro systémy, kde je produkce vyráběna po relativně dlouhou dobu bez přerušení, které by vyvolalo změny ve výrobním procesu. Typickým příkladem takových systémů jsou systémy řízení chemických provozů. Zde, i když dochází ke změně parametrů, dochází k nim buď v průběhu výroby, za chodu anebo je možno tyto změny uskutečnit velmi rychle. Ve smíšených systémech, kde jsou jednotlivé dávky produkovány kontinuálně, pak vzniká specifický problém transportu materiálu, rozpoznávání, kde končí jedna a začíná druhá dávka.
Laboratorní systémy jsou často iniciovány (uvedeny do chodu) operátorem, aby mohl systém použit pro řízení určitého experimentálního testu, nebo experimentálního zařízení sloužící pro rutinní testy produkce atp,
Ve všech uvedených typech systémů je možno nalézt několik specifických počítačových aktivit, prvků, které jsou v oblasti řízení počítači typické.
Typické aktivity (rysy) řídicích systémů

6.2 SEKVENČNÍ ŘÍZENÍ

řízení v uzavřené smyčce (a DDC)

- monitorování
- sběr dat
- analýza dat
- komunikace s lidskou obsluhou (MMI)

Sekvenční řízení
Sekvenční řízení je vždy nějakou součástí každého řídicího systému, převládá však v dávkových systémech.
Dávkové systémy jsou časté při výrobě potravin, chemikálií. V nich se používají takové operace jako míchání surovin, provedení nějakého technologického procesu a následné dávkování.
Řekněne, že chemikálie je vyráběna reakcí dvou jiných chemikálií při určité teplotě. Chemikálie jsou smíchány dohromady v reaktoru, přičemž je řízena teplota operace přidáváním teplé nebo studené vody do prostoru mezi reaktorem a dalším pláštěm reaktoru, který obklopuje vnitřní nádobu reaktoru. Tok vody je řízen dvěma ventily C a D. Přidávání a ubírání surovin do reaktoru je řízeno ventily A a B s odběr produktu ventilem E. Unitř

reaktoru je monitorována teplota a tlak. Postup fungování celého systému může být popsán asi takto:

1. Otevření ventilu A pro dávkování suroviny 1
2. Monitorování tlaku v nádobě, tím se kontroluje, zda již bylo dodáno potřebné množství suroviny 1. Až tomu tak bude, uzavře se ventil A.
3. Nastartování míchče, který míchá chemikálie dohromady.
4. Opakování bodů 1 a 2 s ventilem B, aby se dosáhlo požadovaného množství látky 2.
5. Zapnutí regulátoru a nastavení žádané hodnoty směsi tak, aby směs byla zahřáta na požadovanou teplotu.
6. Monitorování teploty rekce; jakmile dosáhne teplota žádané hodnoty, nastartuje se časovač, aby bylo možno měřit dobu reakce.
7. Jakmile časovač indikuje dosažení požadovaného času, vypne se regulátor a otevře ventil C, aby se ochladil obsah reaktoru. Vypne se míchač.
8. Monitoruje se teplota obsahu; jakmile se ochladí na požadovanou teplotu, otevře se ventil E a vypustí obsah nádoby.

Při řízení počítačem budou všechny uvedené akce, včetně časování, realizovány programově. Pro větší chemický provoz mohou být takové řídící algoritmy velmi dlouhé a komplikované a pro dosažení efektivnosti musejí být prováděny na souběžně pracujících technologiích paralelně.

Proto může být účelné ponechat inicializační sekvenci operací operátorovi. Komplexní automatizace procesu je na jedné straně žádoucí, protože eliminuje možnost zasahování člověka do procesu tam, kde má osobní zájem na jeho ovlivňování (např. nespouštět dávku před koncem směny, zadávat příznivější údaje atp.). V řadě případů však jsou schopnosti člověka posuzovat situaci a řídit proces nanahraditelné a je třeba vytvořit určitý kompromis mezi plně automatizovaným řízením vedeným bez zásahu lidské obsluhy a řízením samotným, kde má operátor možnost do průběhu procesu zasahovat. Jeho zásahy pak musejí být ovšem pečlivě dokumentovány.

V mnoha dávkových systémech je však také, kromě sekvenčního řízení, také nějaké spojité zpětnovazební řízení teploty, tlaku, množství atp. To je někdy realizováno i jako přímě číslicové řízení (DDC).

6.3 JEDNODUCHÁ ŘÍDICÍ SMYČKA.

Uvažujeme následující experiment. V nádobě je třeba udržovat nějakou kapalinu na konstantní teplotě. V kapalině je ponořen ohřívací element, jímž protéká proud ovládaný napětím v rozsahu 0-10 V. Řídicí akce, zvětšení nebo zmenšení tohoto řídicího napětí se uskutečňuje na základě měření skutečné teploty kapaliny a jejího porovnání se žádanou teplotou.

Teplota musí být měřena periodicky s frekvencí danou časovou konstantou procesu. Čím větší je kapacita tanku, tím větší je jeho časová konstanta a tím pomaleji bude reagovat teplota na řídicí zásahy. Perioda vzorkování je tedy z hlediska přesnosti řízení velmi důležitá.

6.4 VÝPOČET ŽÁDANÉ HODNOTY.

V chemické výrobě se může v závislosti na fázích, v nichž se nachází výroba produktu, měnit požadavek na žádanou hodnotu reagující směsi. Tato žádaná hodnota může být určena na základě výpočtu nebo tabulkou jako funkce času nebo jako funkce nějakých měřených parametrů.

Při řízení robota musí být například počítán a řízen jeho pohyb po nějaké žádané trajektorii na základě cesty, jejíž tvar je vložen do počítače v tabelární formě nebo rovnicí parametrické křivky. Trajektorie musí být počítána on-line a opakovaně, což může vyvolat velký objem

početních operací. Systém řízení a ovládání robota pak musí velmi rychle reagovat na změny v trajektorii a na řídicí zásahy. Takové systémy se nazývají servomechanismy.

6.5 VÍCE ŘÍDICÍCH SMYČEK

V trochu složitějších systémech řízení bývá více než jedna řídicí smyčka. Například v systémech řízení otápění v budovách může být teplota řízena v každé místnosti, protože skutečná teplota závisí na řadě faktorů, které jsou v každé místnosti obecně různé. Takové systémy mohou být řízeny lokálními nezávislými regulátory, avšak mohou být též záležitostí jednoho centrálního řídicího počítače, Počítač musí pak být dostatečně dimenzován na to, aby stačil zpracovat všechna data v reálném čase.

6.6 ČASOVĚ KRITICKÉ OPERACE.

Některé systémy vyžadují extrémně rychlé reakce na změny. Například řízení rychlosti otáček motorů na válcovací trati musí být synchronizováno velmi přesně, jinak by se mohl pás plechu přetrhnout, nebo krčit. Uvážíme-li velmi vysokou rychlost pohybu materiálu (10-100 m/s), musí být korekce rychlosti individuálního motoru provedena během několika milisekund.

6.7 SLOŽITOST ŘÍDICÍCH SYSTÉMŮ

Řídicí systémy se neužívají pouze k regulaci a sekvencování řídicích zásahů. Další úlohy v něm obsažené jsou nejrůznější výpočty, sledování kritických hodnot a situací a vydávání tomu odpovídajících alarmů, tj. chybových a varovných hlášení, měření a zobrazování řady veličin, které bezprostředně s řízením nesouvisí atd.
Velký počet čidel a regulačních smyček může sám o sobě učinit úlohu složitou, avšak další komplikace může způsobit složitost samotného procesu, způsobená:

- jeho nelineárností
- měnícími se okolními podmínkami
- měnícími se parametry vlastního procesu
- velkými časovými prodlevami mezi získáním vstupních dat a vykonáním odpovídajících řídicích operací (může být způsobeno nutností měřit v nevhodném místě, nebo vlastnostmi měřicích přístrojů). Proto existují regulátory kompenzující tzv, mrtvý čas – dead time a používající místo měřené hodnoty hodnotu upravenou o vypočtenou kompenzaci
- vnitřní spřažení dílčích procesů, které ovlivňují celkové nepředvídané chování systému

Podle složitosti a charakteru řízeného procesu se pak můžeme setkat s třemi základními modely řízení.
Řízení v uzavřené smyčce na základě zpětné vazby (feedback control)
Řízení lze charakterizovat následujícím schématem.
Řízení v uzavřené smyčce využívající dopředné vazby (feedforward control)
Při řízení se užívá informace o změně žádané hodnoty regulovaných parametrů. Tato změna může být odvozena z nějakého měření nebo výpočtu. Na jeho základě pak stanovujeme žádanou hodnotu.
Nereagujeme tedy v první řadě na výstupy, ale zahrnujeme do řídicího algoritmu vstupní veličiny. Např. při válcování oceli je teplota svitku známa v okamžiku, kdy se blíží k prvnímu pořadí, takže nastavení válcovací mezery může být vypočteno přesně a stejně tak úběry v následujících stolicích.

Výsledkem je zrychlení procesu odezvy na poruchy (za poruchu se zde považuje měnící se teplota svitku), ovšem jen na ty, které můžeme měřit.
Je-li založena regulace i na hodnotách, které nemohou být měřeny, ale vypočítány z ostatních měřených hodnot nebo jiných veličin, hovoříme též o interferenčním řízení (to infer = odvodit).
Využití dopřední vazby je vlastně druhem adaptivního řízení. Adaptivní řízení může mít několik forem, z nichž nejobecnější jsou tři následující:

6.8 PROGRAMOVÉ ADAPTIVNÍ ŘÍZENÍ.

Adaptivní mechanizmus provádí nastavení předem vybraných změn na základě měření parametrů procesu nebo vnějšího okolí.

6.9 AUTOIDENTIFIKACE

Adaptivní řízení s autoidentifikací (selftuning) užívá technik identifikace k nepřetržitému určování parametrů regulovaného procesu; změna parametrů identifikujících proces má pak za následek změnu parametrů řízení (regulátoru).

6.10 ADAPTIVNÍ ŘÍZENÍ NA ZÁKLADĚ MODELU

Řízení na základě reference k modelu vychází ze schopnosti vytvořit adekvátní model a během procesu řízení měřit odchylky mezi chováním modelu a skutečným chováním procesu.
Řízení v uzavřené smyčce využívající dopředné vazby realizované na základě změn žádané hodnoty i měření poruch v systému.
Je zřejmé, že známe-li poruchy a můžeme-li je měřit, může být někdy i možné je korigovat pře tím. než se jejich vliv projeví na výstupních veličinách. To může radikálně zkvalitnit regulaci.
Jednoduchým příkladem může být systém regulace otopu budovy, který bude reagovat nejen na teplotu v jednotlivých místnostech (žádané hodnoty), ale i na venkovní teplotu, která působí jako porucha.
Tato situace je znázorněna na obrázku…

6.11 KOMBINACE ZPĚTNÉ A DOPŘEDNÉ VAZBY.

Realizovaný model řízení by měl obsahovat oba druhy vazeb. Úkolem dopředné vazby je reagovat rychle na známé hodnoty vstupních veličin (žádaná hodnota, poruchy) a to jak měřené tak i vypočtené.
Úkolem zpětné vazby je korekce řízení v pomalejším časovém měřítku prováděná na základě měření výstupních hodnot soustavy. Tak působí zpětná vazba jako korektor nepřesností obsažených v měření veličin procesu, nepřesností v popisu procesu (jeho modelu, identifikace atp.).

6.12 DCC – PŘÍMÉ ČÍSLICOVÉ ŘÍZENÍ

V přímém číslicovém řízení je počítač zařazen do zpětnovazební smyčky. To má za následek, že se počítač stane kritickým prvkem systému a je třeba provést takové zajištění celého systému, aby tento pracoval správně i v případě, pokud počítač přestane plnit své funkce. Obvykle se toho dosáhne tak, že je stanoveno omezení na inkrementální změny prováděné na akčních členech, přičemž omezení se týká především změny nastavení akčního členu.

Rozvoj mikroprocesorové techniky umožnil vývoj regulátorů obsahujících jeden nebo dva mikročipy, které realizují DDC algoritmy, přičemž řada z nich může přímo nahrazovat starší analogové regulátory. Proto mnoho řídicích aplikací nevyužívá zlepšených možností číslicového systému, ale přejímá prověřenou metodiku PID regulátorů. Ten má obecně tvar

m = Kp (e + 1/Ti+ ∫ e dt + Ti de/ dt)

kde

e = r-y je odchylka žádané hodnoty r od měřené y

Kp je celkový zisk regulátoru

Ti je integrační časová konstanta

Td derivační časová konstanta regulátoru

V číslicových systémech je implementace PID regulátoru provedena pomocí difernčních rovnic. Je vzorkovací interval T sekund, je možno použít jednoduchou aproximaci

de/dt = (ek – ek-1) /T

e. dt = ekT k = 0, 1, 2…

Rovnice pak má tvar

mk= Kp (ek + T/Ti Sk + Td/ T (ek – ek-+))

přičemž Sk = Sk-1 +ek jsou součty odchylek.

Na první pohled se to zdá jednoduché, ale je třeba dát pozor na několik věcí. První z nich je omezení akčního signálu, který musí ležet uvnitř nějakého intervalu <Umin, Umax>, což je dáno fyzikálními vlastnostmi akčního členu. Stejně tak je třeba dbát na hodnotu integrálu Sk, která by neměla nabývat velkých hodnot. Jisté problémy představují délka slova počítače a vzorkovací frekvence.

DDC řízení může být aplikováno v regulačním okruhu implementovaném na malém mikropočítači nebo na velkém systému, který pak může zpracovávat až stovky okruhů.

Avšak číslicové řízení není pouze otázkou PID regulátorů; existují i jiné přístupy. Jejich použití je však znevýhodňováno faktem, že použití nastavování PID regulátorů je velmi dobře zvládnuto, a že jsou vhodné pro 90% aplikací. Použití DDC není omezeno na jednoduché zpětnovazební smyčky, ale je možno ho požít k realizaci všech řídicích modelů o nichž byla řeč v předchozím textu.

6.13 MONITOROVÁNÍ

Použití počítačů pro řízení procesů zvětšilo rozsah činností, které mohou být provedeny nejenom z účelem přímého řízení procesů, ale i pro získávání informací, které mohou provozním a vedoucím pracovníkům poskytnout koncentrovaný obraz o stavu všech technologických operací v provozu nebo zařízení.

To je úkolem monitorování nejrůznějších technologických a výrobních parametrů a jejich zobrazení v co nelépe vypovídající formě.

Dřívější zapisovače, digitální displeje a přepínače byly nahrazeny, resp. doplněny obrazovkami a klávesnicemi – to byla z vnějšího pohledu největší změna; vlastní základ technik zpětnovazebního řízení se příliš nezměnil. Mnoho z prvních aplikací počítačů v oblasti řízení užívalo počítače především pro účely monitorování, nikoli pro přímé číslicové řízení a to proto, že počítače nebyly příliš spolehlivé a bylo třeba, aby se výroba (proces) nezastavila při výpadku počítače.

Většina aplikací řízení, kde je počítač použít ve funkci supervizora, je založena na znalosti charakteristik procesu v ustáleném stavu.

V některých řídicích systémech je možno používat s velkým ziskem rafinovaných optimalizačních výpočtů založených na hledání lokálních optim, lineární programování nebo simulací, nelineárních ekonomických dynamických modelů. V takových aplikacích pak musí být tyto algoritmy součástí řídicího software a provozovány v reálném čase paralelně s řízením operací podniku nebo provozu.

6.14 KOMUNIKACE ČLOVĚK- STROJ

Klíč k úspěšné implementaci řídicího systému je často závislý na vhodnosti prostředků, které poskytneme technologické obsluze ke každodenní práci s řídicím systémem.
Všechny informace podstatné pro vytvoření si představy o aktuálním stavu provozu by měly být snadno k dispoici stejně tak jako prostředky pro umožnění interakce s řízeným procesem; zde jde například o možnosti nastavení žádaných hodnot, přizpůsobení akčních členů, potvrzování alarmových hlášení atd. Značná část úsilí při vývoji programového vybavení je proto směřována do programovacích prostředků umožňujících snadnou komunikaci člověka a počítače.
Standardní programové balíky poskytují obvykle prostředky pro zobrazování standardních informací nebo situací, se kterými se setkáváme v oblasti řízení, jako jsou například:

- přehled alarmů
- zobrazení stavu regulačních okruhů
- grafické zobrazení technologických schémat s aktuálními hodnotami
- zobrazení trendů měřených a regulovaných veličin v numerické i grafické podobě atp.

6.15 CENTRALIZOVANÉ ŘÍZENÍ

V 60-tých letech řada aplikací řízení využívala jeden centrální počítač pro řízení celého provozu (zařízení) a to zejména proto, že výpočetní technika byla drahá. V důsledku centralizovaného přístupu nebyla příliš akceptována myšlenka využití jediného počítače pro DDC řízení a to zejména ze dvou důvodů:

1. Vývoj a testování programového vybavení byly poměrně náročné a ještě ztížené nutností provozovat veškeré programové vybavení ne jednom centrálním počítači.
2. Počítače byly nespolehlivé, takže bylo nebezpečí, že při výpadku počítače se ztratí řízení celého provozu.

V polovině 60-tých let však začali výrobci vyrábět digitální regulátory s analogovým zálohováním. Jejich základem byl běžný analogový regulátor, avšak s možností vyslat číslicový signál z počítače do akčního členu. Analogový regulátor pak sledoval, zda počítač vysílá řídicí signály, a když tomu tak nebylo, přešel automaticky na vlastní analogové řízení. To sice zaručovalo bezpečnost řízení při použití DDC, ale zvyšovalo to cenu regulace následkem zdvojení řídicího systému.
V 70-tých letech však cena výpočetní techniky poklesla natolik, že bylo možno přejít k použití zálohování výpočetní techniky a využívání tzv. duálních systémů, kdy v případě výpadku jednoho počítače, přebírá druhý řízení ručně nebo automaticky. Tento přístup měl své slabiny. Kabeláž a interface obvykle nebyly zdvojeny, stejně jako programové vybavení ve smyslu nezávisle navrženého a vytvořeného programového vybavení. Úzkým a kritickým místem se stalo řešení situací výpadku a přepínání řízení, které podstatně zkomplikovalo záležitosti technicky a zejména pak z hlediska programového vybavení. Snižování ceny techniky nakonec přivedlo celou problematiku do rozumného stavu. Použití systémů s více procesory, resp. počítači. Systémy tohoto typu jsou v podstatě dvojího druhu:

6.16 HIERARCHICKÉ

Zde jsou úlohy rozděleny do několika řídicích úrovní v závislosti na svoji vzdálenosti do přímého číslicového řízení, resp. příslušnosti k organizační hierarchické struktuře provozu, výroby atp.
Nejnižší úroveň řízení se pak zabývá vlastním fyzickým řízením, tj. DDC algoritmy atp. Vyšší systémy pak zpracováním údajů, které dostávají od systémů nižší úrovně, a to postupně až na úroveň podniku.

6.17 DISTRIBUOVANÉ

V těchto systémech provádí více počítačů v podstatě podobné úlohy paralelně a předávají si navzájem data, která potřebují druhé počítače pro řízení vlastního svěřeného úseku.
Příklad hierarchicky organizovaného řídicího systému ukazuje následující schéma
Systém je tvořen třemi hierarchickými úrovněmi. Na nejnižší hierarchické úrovni, označené řízení strojů se odehrává řízení jednotlivých strojů resp. zařízení. Odpovídající řídicí systémy mají obvykle následující vlastnosti:

- Přímé spojení vstupů i výstupů s technologickým procesem
- ROM paměť s rozsahem řádově desítek kilobytů pro uchovávání programů
- Poměrně omezená RAM paměť (jednotky až desítky kilobytů)
- Konstrukčně odolné provedení vyhovující tvrdým podmínkám s ohledem na teplotu, prašnost a otřesy prostředí
- Jednoduché komunikační prostředky pro obsluhu (panely, přepínače, tabla, signalizační prvky atp.)
- Některé systémy mají i automatický restart po výpadku napájení. Tato vlastnost však není vždy žádoucí, zejména u systémů, které nejsou monitorovací, ale skutečně provádějí akční zásahy.

Kromě řízení vlastního stroje může být systém řízení realizován i tak, že umožňuje, v omezené míře, náhradní, záložní řízení sousedních strojů na téže úrovni. Takové záložní řízení se provádí omezeným způsobem a jeho smyslem je zajistit životně důležité funkce systému, například zamezení havárie. Často bývá nutné ¨, aby v takovém záložním režimu byla postižená část technologie provozována v určitém minimálním provozním režimu.
Úkolem nadřazeného systému na další hierarchické úrovni, označované jako řízení provozu, je v normálních provozních podmínkách výběr, koordinace a řízení činnosti řídicích systémů nižší úrovně tak, aby celý chod této části byl optimální. Řídicí systémy této úrovně pak mají většinou následující vlastnosti:

- Obsahují systém sběru a zpracování dat z celého provozu tvořený jak daty předávané podřízenými řídicími systémy, tak vlastními vstupními zařízeními a prostředky
- Jsou vybaveny pevnými disky
- Umožňují přímé záhy do řízení podřízených strojů. Tato možnost je odvozena od filozofie konkrétného řídicího systému a její rozsah může být systém od systémů velmi variabilní
- Obsluha má možnost komunikace prostřednictvím obrazovkových terminálů, kde získá informace jak o celkovém stavu zařízení, tak i detailní informace o provozu jednotlivých strojů
- Možnost připojení záložních pamětí (pružné disky, magnetopáskové jednotky atp.)
- Bohatší programové vybavení umožňující kvalitativně vyšší úroveň zpracování a zobrazení monitorovaných dat a výsledků

Úkolem třetí, nejvyšší hierarchické úrovně řízení je zejména plánování a rozvrhování výroby. Na této úrovni bývají používány prostředky výpočetní techniky běžné ve výpočetních střediscích, například počítače IBM, VAX. Tyto systémy mají kromě toho ještě další úkoly související s agendami závodu, nebo podniku, jako například evidenci skladového hospodářství, zakázek, fakturace, vytváření výrobních přehledů atp.
Jednotlivé úrovně pracují s různými časovými úseky, horizonty.
První, nejnižší úroveň řízení operuje s časy potřebnými k provedení jedné, nebo několika pracovních operací. V závislosti na typu technologie to mohou být časy v rozmezí desetin sekund až několika minut.
Střední úroveň řízení uvažuje s časovými horizonty řádově minut, desítek minut, hodin, směn a dní.

Nejvyšší úroveň řízení uvažuje s časovými horizonty v trvání dní, měsíců a let.
Uvedené rozdělení je orientační a časové rozdělení jednotlivých úrovní může překrývat. Záleží na tom, jak sou stanoveny cíle celého systému i jeho jednotlivých úrovní, jaký druh technologického a výrobního procesu je předmětem řízení, a zejména pak záleží na časových konstantách jeho jednotlivých složek. Ty totiž určují, jak rychle reaguje řízený systém na akční zásahy, na poruchy a na měnící se vlivy okolního prostředí a tím je třeba v jednotlivých úrovních řízení počítat.
Systémy pracující uvedeným způsobem se také nazývají automatizované systémy řízení (ASŘ) a podle toho, jakou hierarchickou úrovní se zabývají se označují jako

- ASŘTP – ASŘ technologického procesu
- ASŘVP – ASŘ výrobního procesu
- ASŘZ - ASŘ závodu
- ASŘP – ASŘ podniku

Souhrn více ASŘ uvedených typů pokrývající ucelený okruh problémů řízení je pak někdy nazýván Integrovaným řídicím systémem.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Požadavky na hardware pro RT- aplikace

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

7 POŽADAVKY NA HARDWARE PRO RT- APLIKACE 3

7.1 Požadavky na hardware pro RT- aplikace 4

7.2 Centrální jednotka počítače 4

7.3 Paměť 4

7.4 Vstup a výstup. 4

7.5 Struktura sběrnic 4

7 POŽADAVKY NA HARDWARE PRO RT- APLIKACE

OBSAH KAPITOLY:

popište centrální jednotku počítače, paměť, vstup výstup, strukturu sběrnic

MOTIVACE:

Aritmeticko- logická jednotka (ALU) společně s řídicími obvody a obecnými registry tvoří centrální jednotku (CPU). Dnes se využívají dva základní typy CPU: s úplnou sadou instrukcí (CISC) a s redukovanou sadou instrukcí (RISC). Výhodou prvé je široká škála instrukcí, výhoda druhé je jednoduchost, rychlost a mnohdy stejně dlouhé vykonávaní všech instrukcí

Dnes je možné používat paměti RAM o kapacitě desítek Mbytů a omezení tkví v postatě jen v adresovacích možnostech CPU. Kromě pamětí RAM je dnes běžné vybavení paměťmi typu ROM, PROM, nebo EPROM pro uložení kritických programů nebo předdefinovaných funkcí

Sběrnice je (z fyzikálního hlediska) soubor vodičů, vedoucích elektrické signály. Mohou být na tištěných spojích, ale mohou to být i spoje kabelové. Toto provedení charakterizuje mechanické vlastnosti sběrnic. Pro návrh rozhraní je třeba také znát elektrické chrakteristiky, jimiž jsou:

- úroveň signálů
- typ výstupních bran (otevřený kolektor, třístavový atp.)

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat centrální jednotku počítače, paměť, vstup výstup, strukturu sběrnic

7.1 POŽADAVKY NA HARDWARE PRO RT- APLIKACE

I když dnes v podstatě každý počítač je schopen provádět operace v režimu reálného času, optimální výkon poskytují specializované počítače. Jejich důležitou charakteristikou je modularita; umožňují připojit dodatečné jednotky, zejména specializované vstupní a výstupní zařízení, k základní konfiguraci.
Další důležitou charakteristiku jsou vstupní a výstupní kanály, umožňující připojení specializovaných vstupů a výstupů a také displeje a vstupní zařízení pro operátora procesu. Důležitá je též schopnost komunikovat s jinými počítači.

7.2 CENTRÁLNÍ JEDNOTKA POČÍTAČE

Aritmeticko- logická jednotka (ALU) společně s řídicími obvody a obecnými registry tvoří centrální jednotku (CPU). Dnes se využívají dva základní typy CPU: s úplnou sadou instrukcí (CISC) a s redukovanou sadou instrukcí (RISC). Výhodou prvé je široká škála instrukcí, výhoda druhé je jednoduchost, rychlost a mnohdy stejně dlouhé vykonávaní všech instrukcí.
Tyto vlastnosti zlepšují celkovou rychlost počítače. Čím složitější instrukční kód, tím je však komplikovanější programování v assembleru. O to více je žádoucí existence kvalitních překladačů, které by měly využít specifických vlastností strojového kódu.
Pro řídicí systémy jsou též klíčovou otázkou vlastnosti a rychlost přenosu informace mezi CPU, periferiemi a pamětí. Důležitá je rychlost, schopnost přenosu paralelně s činností CPU (nezávislou na přenosu) a umění komunikovat se širokým spektrem periferních zařízení.
Podstatná je i flexibilní a efektivní víceúrovňová struktura přerušovacího systému.

7.3 PAMĚŤ

Dnes je možné používat paměti RAM o kapacitě desítek Mbytů a omezení tkví v postatě jen v adresovacích možnostech CPU. Kromě pamětí RAM je dnes běžné vybavení paměťmi typu ROM, PROM, nebo EPROM pro uložení kritických programů nebo předdefinovaných funkcí.Užití paměti ROM usnadňuje řešení situací výpadku systému a jeho opětovného nastartování, případně kontrolu nepovolených operací zápisu do chráněných oblastí paměti.
Externí vnější paměti jsou paměti s nízkou vybavovací rychlostí, mají však schopnost asynchronní činností s CPU a zapisují informace po blocích. Proto u nich musí být věnována pozornost otázkám přenosu dat. Obvyklým řešením je použití DMA (Direct Memory Access), kde je k dispozici speciální řadič, vykonávající kontrolu přenosu bloků dat mezi periferním zařízením a operační pamětí asynchronně s CPU.

7.4 VSTUP A VÝSTUP.

Rozhraní (interface) pro vstupy a výstupy (IO) je jednou s nejsložitějších částí výpočetního systému. Je to tím, že musí umožnit připojení a obsloužení širokého spektra periferních zařízení s velmi různorodými nároky na rychlost přenosu. Například tiskárna může pracovat rychlostí 300 baudů a disk rychlostí 500 Kbaudů. Přitom může jít o přenos paralelní nebo sériový, přenos jehož součástí jsou konverze analogových, případně číslicových hodnot atp.

7.5 STRUKTURA SBĚRNIC

Sběrnice je (z fyzikálního hlediska) soubor vodičů, vedoucích elektrické signály. Mohou být na tištěných spojích, ale mohou to být i spoje kabelové. Toto provedení charakterizuje

mechanické vlastnosti sběrnic. Pro návrh rozhraní je třeba také znát elektrické chrakteristiky, jimiž jsou:

- úroveň signálů
- typ výstupních bran (otevřený kolektor, třístavový atp.)

Je třeba též znát funkční charakteristiku sběrnic, tj. co reprezentují signály přenášené sběrnicí. Zde mohou být tři typy informací:

- adresové linky – říkají kam
- datové linky – říkají co
- řídicí a stavové linky – říkají kdy

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Průmyslová rozhraní

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

8 PRŮMYSLOVÁ ROZHRANÍ 3

8.1 Průmyslová rozhraní 4

8.2 Digitální hodnoty 4

8.3 Analogové hodnoty 4

8.4 Pulsní hodnoty 4

8.5 Telemetrie 4

8 PRŮMYSLOVÁ ROZHRANÍ

OBSAH KAPITOLY:

Definujte digitální hodnoty, analogové hodnoty, pulzní hodnoty, telemetrie

CÍL:

Po prostudování tohoto odstavce budete umět:

- Definovat digitální hodnoty, analogové hodnoty, pulzní hodnoty, telemetrie

8.1 PRŮMYSLOVÁ ROZHRANÍ

Čidla a akční členy připojené k procesu nebo provozu mohou být různých typů a forem podle toho, k jakému měření slouží. Jejich společnou charakteristikou je, že je třeba převést fyzikální (měřenou) veličinu (tlak, teplota, rychlost) na nějakou elektrickou veličinu, která je dále digitalizována.
Z toho pohledu pak potřebujeme následující typy rozhraní:

8.2 DIGITÁLNÍ HODNOTY

Jsou buď binární (stav otevřeno / zavřeno, vypnuto /zapnuto) nebo obecné digitální hodnoty (výstup z digitálního voltmetru atp.)

8.3 ANALOGOVÉ HODNOTY

Termočlánky, tenzometry, atd. dávají výstup v mV. Ten může být zesílen až do rozsahu +-10V; další čidla mohou poskytovat proudové výstupy, obvykle v rozsahu 0-20 mA. Ty jsou odolnější proti rušení. Tyto hodnoty musí být konvertovány na číslicové hodnoty.

8.4 PULSNÍ HODNOTY

Řada měřících přístrojů (např. měřiče průtoku) poskytují výstup ve formě impulsů. Použití krokových motorů jako akčních členů vyžaduje použití pulsních výstupů. Mnoho klasických regulátorů také pulsních výstupů; např. ventily pro řízení průtoku jsou často ovládány vypínáním a zapínáním DC nebo AC motoru, přičemž délka pulsu udává míru změny otevření ventilu.

8.5 TELEMETRIE

Vzrůstající použití vzdálených měřících stanic (nezbytných např. pro řízení tlaku v plynových sítích) zvyšuje požadavky na použití dálkového přenosu dat. Ten se může uskutečnit zemním vedením, rádiem nebo veřejnou telefonní sítí. Obyčejně jsou data přenášena sériově a zakódována do kódu ASCII.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Techniky přenosu dat

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

9 TECHNIKY PŘENOSU DAT 3

9.1 Techniky přenosu dat 4

9.2 Přerušovací systém a přenos dat. 4

9.3 Srovnání technik přenosu dat. 4

9.4 DMA – technika přímého přístupu do paměti 5

9 TECHNIKY PŘENOSU DAT

OBSAH KAPITOLY:

Popsat přerušovací systém a přenos dat, srovnání technik přenosu dat, techniku DMA – technika přímého přístupu do paměti

CÍL:

Po prostudování tohoto odstavce budete umět:

- popsat přerušovací systém a přenos dat, srovnání technik přenosu dat, techniku DMA – technika přímého přístupu do paměti

9.1 TECHNIKY PŘENOSU DAT

Charakteristickým rysem většiny hardwarových rozhraní periferních zařízení je, že pracují souběžně s CPU, přičemž jsou značně pomalejší. Je-li zařízení řízeno počítačem, bude jeho malá rychlost nepříznivě zpomalovat i činnost počítače. Tento přístup přímého řízení periferie se nazývá programovaný přenos. Alternativním přístupem je použití DMA – přímého přístupu do paměti.
Inteligentní možnost nabízení specializované nikroobvody, resp. mikroprocesory, určené pro detailní řízení periferních operací, které probíhají zcela v jejich režii. Údaje jsou nejdříve přenášeny v časové režii IO- mikroobvodu do jeho paměti, tzv. vyrovnávací paměti (bufer) po větších úsecích. V další fázi je pak aktivován přenos dat z vyrovnávací paměti (bufer) do paměti počítače, který již probíhá velmi rychle, nejsa zdržován pomalou rychlostí periferie. Této technice se říká buferovaný přenos. Největším problémem při přenosu je časování, nebť je třeba kontrolovat a synchronizovat činnost CPU a periferního zařízení.

9.2 PŘERUŠOVACÍ SYSTÉM A PŘENOS DAT.

Přerušení je mechanismus, jímž může být dočasně pozastaven chod aktuálně prováděného programu, aby byla umožněna dočasná činnost speciálního programového kódu (modulu, programu, podprogramu). Když je činnost tohoto modulu skončena, je chod programu, který byl dočasně přerušen a odložen, obnoven.
Přerušovací systém umožňuje, aby byl běh programu přerušován událostmi nezávislými na stavu provádění programu, neboli asynchronními událostmi. To znamená, že procesor přeruší běžící program, začne vykonávat jiný program a po jeho skončení se vrátí zpět k původnímu programu do místa, kde byl přerušen.
Tyto asynchronní události způsobují hardwarová přerušení, která generují různá periferní zařízení počítače jako např. myš, klávesnice, časovač apod. Programy, které tyto události obsluhují, se nazývají obslužné programy přerušení (interrupt handler), stručněji též rutinami přerušení.
Tímto způsobem lze tedy dosáhnout asynchronního řízení periferních operací a jejich souběžného provádění s kódem programu, který v dném okmžiku nemusí mít s prováděnými periferními operacemi nic společného. Je jasné, že tato nezávislost je podstatná zejména pro systémy pracujícím v reálném čase. Přerušení je proto věnována samostatná kapitola.

9.3 SROVNÁNÍ TECHNIK PŘENOSU DAT.

Metoda dotazování (pooling) je jednoduchá na programovou implementaci. Použijeme-li techniku přerušení, můžeme dostat podprogram ¨, který méně komplikovaný, než program který používal metodu dotazování. Prakticky nelze vytvořit program, který by používal techniku dotazování a byl výkonem srovnatelný s programem využívající přerušení. Přerušení může vzniknout v okamžiku, kdy CPU provádí jakoukoli instrukci programu a technika dotazování, která by se měla vyrovnat s touto v podstatě nespočetnou množinou požadavků na ošetření IO, by byla nepředstavitelně komplikovaná, ne-li nemožná.
Avšak systémy založené na použití technik přerušení se daleko hůře odlaďují, protože mohou obsahovat časově závislých chyb a je tudíž velmi náročné napsat takové testovací rutiny, které by prověřily všechny myslitelné situace.
V případě vysokých přenosových rychlostí je použití technik přerušení neefektivní v důsledku relativně značné časové režie v rutině přerušení. Ta je zapříčiněna ukládáním a obnovováním registrů, které by asi bylo velmi časté a značně snižovalo maximální dosažitelnou přenosovou rychlost. Protože použití technik dotazování též není ideálním řešením, používá se technika přímého přístupu do paměti (DMA -direct Memory Access).

9.4 DMA – TECHNIKA PŘÍMÉHO PŘÍSTUPU DO PAMĚTI

Technika DMA používá harwarového řešení. Užívají se tři varianty.

- Burst móde

Řadič DMA uzamyká CPU po určitou krátkou periodu potřebnou pro přenos např. 256 bytů z operační paměti do odpůrné vyrovnávací paměti řadiče. V některých aplikacích to však může podstatným způsobem snížit rychlost odezvy počítače a proto rychlé RT aplikace se tato varianta nedoporučuje.

- Distribuovaný mode

Zde řadič občas vezme CPU strojový cyklus a využije ho k přenosu informace mezi operační a vyrovnávací pamětí. To je také pro RT aplikace nepřijatelné, protože stejné úseky programu by nemusely být prováděny stejně dlouho. Proto se používá

- Metoda kradení cyklů

Zde se uskutečňuje přenos dat pouze v těch cyklech, kdy CPU nepoužívá datovou sběrnici. Tak může program pokračovat plnou rychlostí, aniž by byl ovlivněn činností DMA. Je to však pochopitelně ta nejpomalejší metoda přenosu dat z těchto tří variant.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Přerušení

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

10 PŘERUŠENÍ 3
10.1 Přerušení 4
10.2 Mechanismy odezvy na přerušení 4
10.3 Vektor odezvy na přerušení 4
10.4 Víceúrovňová přerušení 5
10.5 Volání a návraty z rutin přerušení 5
10.6 RT jazyky 5
10.7 Základní vlastnosti RT jazyků 5
10.7.1 Zabezpečenost 5
10.7.2 Flexibilita 6
10.7.3 Jednoduchost 6
10.7.4 Protabilita 6
10.7.5 Efektivita 7
10.8 Požadavky na RT – jazyky 7
10.9 Chyby a poruchy v RT-systémech 7
10.10 Poruchy a chyby 7
10.10.1 Definice 7
10.11 Implementační úroveň 8
10.11.1 Chyby 1.skupiny 8
10.11.2 Důsledky chyb 9
10.11.3 Monitorování. 9
10.11.4 Chyby 2. skupiny 10
10.11.5 Chyby 3, skupiny . aplikační úroveň 10
10.11.6 Chyby 4. skupiny 11
10.12 Ošetření chyb v RT- systémech 11
10.13 Klasické prostředky pro ošetření chyb 12
10.13.1 Obsluhy výjimek 12

10 PŘERUŠENÍ

Obsah kapitoly:

Definujte požadavky na RT jazyky, chyby a poruchy v RT systémech, implementační úrovně, ošetření chyb v RT systémech

Popište popsat mechanismy odezvy na přerušení, vektor odezvy na přerušení, víceúrovňová

Přerušení, volání a návraty z rutin přerušení, RT jazyky ,základní vlastnosti RT jazyků

CÍL:

Po prostudování tohoto odstavce budete umět:

- definovat požadavky na RT jazyky, chyby a poruchy v RT systémech, implementační úrovně, ošetření chyb v RT systémech
- popsat mechanismy odezvy na přerušení, vektor odezvy na přerušení, víceúrovňová přerušení, volání a návraty z rutin přerušení, RT jazyky
- základní vlastnosti RT jazyků

10.1 PŘERUŠENÍ

Přerušení je mechanismus, jímž může být dočasně pozastaven chod aktuálně prováděného programu , aby byla umožněna dočasná činnost speciálního programového kódu (modulu, programu, podprogramu). Když je činnost tohoto modulu skončena, je chod programu, který byl dočasně přerušen a odložen, obnoven.
Přerušení jsou podstatným mechanismem pro korektní činnost většiny RT-systémů. Jsou důležité zejména pro:

1. Realizaci hodin reálného času – speciální obvod, generuje pravidelné časové signály, které generují přerušení. Podprogram pro obsluhu těchto přerušení počítá tyto signály a na jejich základě udržuje a aktualizuje hodiny reálného času.
2. Přerušovací (alarmové) vstupy – různá čidla mohou na základě změny logické úrovně nějakého stavu generovat signál přerušení signalizující nějakou událost ve vnějším okolí. Protože tyto změny mohou být velmi řídké, představuje přerušovací mechanismus velmi efektivní způsob jejich ošetření.
3. Signalizace poruchy hardware . může být řešena pomocí přerušení.
4. Výpadek napájení je ošetřován pomocí speciálních obvodů, které velmi rychle vyhodnotí pokles napětí a vytvoří signál přerušení na jehož základě je vyvolána obslužná rutina, která zajistí dokončení rozpracovaných instrukcí a úklid registrů před zastavením počítače. Eventuálně může automaticky zajistit přepnutí na náhradní napětí.
5. Prostřednictvím přerušení jsou též ošetřovány tzv. výjimečné situace jako přetečení a podtečení aritmetických operací.

10.2 MECHANISMY ODEZVY NA PŘERUŠENÍ

Nejznámější metody zpracování a odezvy na přerušení jsou tyto.:

1. Přenesení zpracování na specifickou adresu – nejčastěji instrukce CALL.
2. Naplnění čítače instrukcí novou hodnotou ze specifikovaného registru nebo paměťového místa.
3. Provedení instrukce CALL na adresu dodanou z externího systému.
4. Užití výstupního signálu – Interrupt Acknovledge (potvrzení přerušení) – nahráním instrukce z externího zařízení.

Metody 1 a 2 jsou programové (software biased), protože vyžadují minimum technických prostředků a určení zdroje přerušení a jeho kompletní obsloužení přenechávají software.
Metody 3 a 4 jsou technické (hardware-biased), protože problém identifikace a vyvolání obsluhy přerušení řeší technickými prostředky.
Metody 2, 3, 4 jsou metodami, které v určité formě využívají (každá) tzv. vektor přerušení.
Vektor přerušení je místo v paměti. tj. slovo (dvojsovo), obsahující adresu rutiny pro obsluhu příslušného přerušení, případně jinou informaci, která umožňuje vyvolat odpovídající obsluhu přerušení.

10.3 VEKTOR ODEZVY NA PŘERUŠENÍ

Vektor může mít různé formy. Může obsahovat instrukce, adresy obslužných rutin jednotlivých přerušení, adresy ukazatelů na tyto rutiny, případně další možnosti. Konkrétní případ řešení přerušovacího systému je uveden dále na příkladu PC počítačů.

10.4 VÍCEÚROVŇOVÁ PŘERUŠENÍ

Ve většině RT systémů je nepřijatelná jedna úroveň přerušení, neboť úkolem přerušení je poskytnout co nejrychlejší odezvu na externí události a je třeba zabránit tomu, aby méně důležitá přerušení (úrovně) blokovala přerušení důležitější (vyšší úrovně). Je zřejmé, že možnost přerušit rutinu obsluhující jiné přerušení, by měla dovolena pouze přerušením, které mají vyšší prioritu (jsou důležitější) než přerušení právě obsluhované. To je zajišťováno prostřednictvím tzv. maskování priority nižších úrovní. V některých systémech pak existuje maskovací registr, který je možno naplnit programově a definovat tak, která přerušení mají být znemožněna.

10.5 VOLÁNÍ A NÁVRATY Z RUTIN PŘERUŠENÍ

Rutiny přerušení nemusíme nutně používat pouze k obsluze příslušných vnějších zařízení, ale také k různým jiným účelům, pro které se nám budou hodit přerušení od těchto zařízení. Typickým příkladem je například vyvolání našeho programu pomocí „horké klávesy“, tj. stisku určité klávesy, kterou chceme ovladači klávesnice „uzmout“ pro rozběhnutí našeho programu. Při potřebě volat program v pravidelných časových intervalech však využijeme přerušení od časovače.
Při jakémkoli předávání řízení původní rutině přerušení je vždy třeba obnovit registry CPU do takového stavu, v jakém byly prí vzniku přerušení.

10.6 RT JAZYKY

Prostředky pro programování RT-úloh, které usnadňují jejich realizaci jsou především

- operační systémy reálného času
- programovací RT – jazyky

Náročnost prostředí, v němž jsou úlohy řízení technologických procesů provozovány, klade na programové vybavení pro práci v reálném čase zvláštní nároky:

- Především musí být maximálně spolehlivé. Jeho selhání může znamenat značné ztráty materiální i lidské.
- RT – systémy jsou obvykle rozsáhlé a složité. To komplikuje a následkem toho mohou být náklady na vývoj a údržbu systému velmi vysoké.
- RT-systém musí dávat odezvu na řadu různých vnějších událostí v daném časovém limitu.
- RT-systém zahrnuje obvykle kromě konvenčních periferních zařízení i řadu zařízení nestandardních, jejichž obsluhu je třeba zajistit.
- RT-systém by měl být efektivní a měl by co nejlépe využívat příslušné programové vybavení.

Je třeba zdůraznit, že i když jsou RT – jazyky navrhovány tak, aby splňovaly především požadavky RT-systémů, není jejich použití omezeno pouze na oblast řízení procesů. Jsou dobrým nástrojem pro tzv. systémové programování, např. pro tvorby operačních systémů, kompilátorů, pro řešení simulace atp. Typickým příkladem tohoto je například programovací jazyk MODULA-2.

10.7 ZÁKLADNÍ VLASTNOSTI RT JAZYKŮ

10.7.1 Zabezpečenost

Zabezpečenost jazyka je dána tím, do jaké míry lze chyby v programu detekovat při kompilaci programu, nebo operačním systémem při chodu přeloženého programu.

Je lepší, když co nejvíce chyb možno detekovat již ve fázi kompilace programu, protože odhalování chyb při běhu RT – systému je podstatně náročnější, než pří provozování programového vybavení, které nepracuje v reálném čase. Je přirozené, že je tak možno zjišťovat pouze formální chyby, nebo chyby způsobené formálně sice správným avšak nevhodným použitím některých jazykových konstrukcí, zejména datových typů.
Hlavním předpokladem k zabezpečení jazyka na úrovni kompilace je proto kontrola dat. Ve fázi běhu programu pak lze provádět i zde kontrolu omezenou, spočívající například v kontrole indexu polí, testování přípustnosti argumentů standardních funkcí atp. Protože tyto kontroly stojí čas a místo v paměti, bývají RT- překladače vybaveny možností volby, zda tuto kontrolu při kompilaci zařadit, nebo vyřadit. Odladěné programy je možno používat bez kontrol tohoto typu, jsme-li přesvědčeni, že k uvedeným chybám nemůže dojít.
V žádném případě pak nelze ovšem odhalit chyby, které vznikly na základě nesprávného logického návrhu programu anebo logicky chybným naprogramováním. Velmi častým případem jedné z tisíců možných chyb tohoto druhu, která se velmi často vyskytuje, je např. nesprávná aplikace de Morganových pravidel, kterou programátor provede při úpravě vztahu disjunkce nebo konjunce, aby upravil nějaký výraz do tvaru vhodnějsšího pro testování v daném místě programu.

- Čitelnost

Čitelnost (readability) jazyka závisí na řadě faktorů od volby klíčových slov, možnosti volby identifikátorů až po prostředky modularizace programů. Účelem je poskytnou dostatečně jasnou notaci. která by umožňovala psaní programů, jejichž funkce by byla pochopitelná co nejvíce ze samotného zápisu, bez pomocné dokumentace.
Snadná čitelnost programu není pouze záležitostí jazyka, ale především programovacího stylu programátora; jazykové prostředky v tomto směru dávají lepší nebo horší předpoklady. K zvýšení čitelnosti programu může programátor přispět vhodným a logickým strukturováním programu do jednotlivých programovacích jednotek a funkcí, přehledným zápisem a úpravou jeho textu, okomentováním důležitých a méně jasných funkcí, mnemotechnicky vhodnou volbou identifikátorů, vhodnou volbou datových struktur atd.

10.7.2 Flexibilita

Flexibilita jazyka umožňuje programátorovi, aby mohlo programátor prostředky tohoto jazyka vyjádřit všechny potřebné operace a funkce, aniž by se musel uchylovat ke strojovému kódu nebo jiným prostředkům ležícím mimo sféru tohoto jazyka. Tento požadavek je zvláště u RT-jazyků,protože v RT – systémech je třeba programovat řadu nestandardních periferních zařízení. Flexibilita je však v rozporu s požadavkem zabezpečenosti, takže pokud jede o jazykové prostředky, je třeba volit rozumný kompromis.

10.7.3 Jednoduchost

Jednoduchostí se v tomto případě myslí jednoduchost pro osvojení, naučení se jazyka. Praxe ukazuje, že jednoduchosti se kupodivu nedosáhne tím, že se vyhneme složitějším jazykovým konstrukcím, ale spíše tak, že na jazyk neklademe žádná zbytečná dodatečná omezení. Například je vhodné, aby pole bylo možno utvářet se složek libovolného typu, i uživatelsky definovaných. Nevhodným omezením některých jazykových konstrukcí je pak třeba zapamatovat si řadu výjimek, ale kupodivu i méně efektivního kódu generovaného kompilátorem.

10.7.4 Protabilita

Portabilita znamená, že jazyk by měl být navržen tak, aby nezávisel na konkrétním technickém vybavení a aby programy napsané v jednom jazyce bylo možno přenášet na jiné počítače a tam je po po překladu provozovat, aniž by bylo nutno předtím dělat ve zdrojovém textu programu nějaké úpravy.

Již samotná existence programovacích jazyků je výrazem snahy o dosažení portability, V praxi je však dosažení portability tvrdým oříškem, zejména v případě RT-systémů, které využívají často programování na nízké úrovni (low-level programing).
Tento problém se řeší (např. C-jazyku, nebo v jazyku MODULA – 2) tím, že všechny strojově závislé funkce jsou vyjmuty ze sféry, kterou zajišťuje kompilátor a jsou přesunuty do knihoven, jejichž obsah a rozhraní je součástí definice jazyka, avšak které jsou realizovány specificky pro každý konkrétní typ počítače.

10.7.5 Efektivita

Efektivita je požadavek na vysokou výkonnost vytvořeného programového vybavení, případně na nízký objem přeloženého kódu, zejména u RT- aplikací lze očekávat velký důraz na tyto vlastnosti.

10.8 POŽADAVKY NA RT – JAZYKY

Pořadí výše požadovaných vlastností RT- jazyků vyjadřuje i jejich důležitost. Kromě toho existují i další požadavky na vlastnosti RT-jazyků. RT-systémy jsou velmi často velmi rozsáhlé a složité a proto je třeba, aby RT-jazyk řešil problematiku „programování ve velkém“ a poskytoval prostředky pro explicitní modularizaci.
Protože RT-systémy lze nejlépe popsat jako množinu paralelně probíhajících posloupností akcí (procesů), měl by RT-jazyk obsahovat prostředky pro paralelní programování.
Jedním z potřebných vlastností programového vybavení RT-procesů je možnost ovládání nestandardních periferních zařízení. Pro tyto řešení těchto úloh by proto měl RT- jazyk obsahovat potřebné vybavení.
Požadavek vysoké spolehlivosti vyžaduje v RT-jazycích, aby existovaly prostředky umožňující zotavení z chyb, kterým nelze ani při sebedokonalejším provedení aplikačního programového vybavení zabránit.

10.9 CHYBY A PORUCHY V RT-SYSTÉMECH

Identifikace a adekvátní ošetření chyb je nezbytnou podmínkou úspěšného provozování aplikačního programového vybavení, zejména v automatizovaných systémech řízení.
Tam, ke počítač pracuje nepřetržitě a v reálném čase, je ochrana proti poruchám a jejich následkům nutnou podmínkou pro úspěšné provozování takového systému.
Chyby (v tom nejobecnějším slova smyslu), provázejí aplikace počítačů v ASŘ od stadia jejich projekčního zrodu až do doby rutinního provozování, jsou nejrůznějšího původu, druhu a závažnosti. Jsou to chyby v zadání úlohy, v koncepci progra mového vybavení, ve výběru výpočetního systému a jeho prostředků, ve firemním programovém vybavení a jeho dokumentaci, chyby v uživatelských programech, chyby technického vybavení.
Stává se, že i v pokročilejších stádiích vývoje aplikačního programového vybavení objevujeme chyby, které mají původ ve stadiích předchozích. Jejich odstranění může být pak i značně pracné a nákladné.
Je tedy zřejmé, že spektrum zdrojů a situací, produkujících nejrůznější poruchové stavy a podmínky, je velmi široké, budeme-li pojem chyba chápat v tak širokém smyslu, jak jsme uvedli výše. Jaké poruchy a chyby nás především zajímají?

10.10 PORUCHY A CHYBY

10.10.1Definice

Porucha (dle ČSN 369 001) je částečná, nebo úplná ztráta schopnosti provozu soustavy nebo její části. Dle místa vzniku můžeme rozdělit poruchy na poruchy technologického zařízení

(dále HW porucha), poruchy programového vybavení (dále SW porucha) a poruchy v okolí systému. Porucha se navenek projevuje jako chyba. Chyba způsobená určitou poruchou je neshoda mezi správným (očekávaným) chováním soustavy a nesprávným (realizovaným) chováním této soustavy.
Je zřejmé, že počet SW – poruch v nějakém systému bude konstantní (neprovádí-li se úpravy programů), zatímco frekvence SW- chyb, projevů těchto poruch, bude kolísat v závislosti na čase, protože záleží na tom, kolikrát bude poruchová část aktivována. Na druhé straně HW-poruchy jsou kvalitativně i kvantitativně závislé na čase. Specifickým jevem jsou tzv. občasné HW- poruchy. Jsou to poruchy, které trvají jen omezenou dobu, po níž hardware dosáhne bezporuchového stavu bez vnějšího zásahu (bez opravy). Udává se, že 90 % operačních poruch je způsobeno občasnými HW- poruchami. Poruchy okolí mají pak příčinu ve vnějších vlivech (mimo systém počítače) a ve způsobu přístupu k systému.
Nás zajímají především druhy aplikací, v nichž počítač pracuje v reálném čase, neptržitě, komunikuje se svým lidksým a technickým okolím a událostu¨i a požadavky na komunikaci z tohoto okolí přicházejí náhodně a nekoordinovaně. ASŘ jistě patří do této kategorie.
Poruchy a chyby, s nimiž musíme počítat při vývoji programového vybavení, jeho relizaci a provozu, lze rozdělit asi do následujících čtyř okruhů:

10.11 IMPLEMENTAČNÍ ÚROVEŇ

1. Chyby identifikace na úrovni operačního systému (OS), ktré jsou též na úrovni OS nebo překladačem ošeteny.
2. Chyby identifikované na úrovni OS nebo překladače, jejichž ošetření je postoupeno volajícímu programu.

Aplikační úroveň

3. Programové chyby identifikované a ošetřené na úrovni aplikačních programů.
4. Formální a logické chyby v komunikaci (konverzaci) systému s okolím zajištěné a ošetřené na úrovni aplikačního programového vybavení.

Uvedené čtyři okruhy zachycují tu lepší situaci, protože chyby jsou ošetřené a identifikované. Horší situace nastává, jsou-li chyby neošetřené a nebo neidentifikované. I s takovými chybami je však nutno počítat.
Co se očekává od programového vybavení (AVP)?
Od programového vybavení se očekává, že bude odolné vůči poruchám. Zajistit stoprocentní odolnost systému vůči jakýmkoli poruchám není možné. Běžné systémy budou spíše částečně odolné; bude existovat konečná množina poruch, vůči nimž bude systém odolný. Má-li bát systém považován za odolný, je třeba, aby v případě poruchy v systému byly splněny následující podmínky:

- nejdi k ukončení výpočetního procesu zastavením výpočtu nebo k jeho samovolné změně
- nedojde k produkci chyby
- doba odezvy systému nepřesáhne stanovenou hranici
- požadavky na přidělení výpočetních prostředků nepřesáhnou možnosti systému.

Ukážeme nyní konkrétněji příklady situace, jak dochází ke vzniku uvedených poruch a chyb.

10.11.1Chyby 1.skupiny

První zdrojem chyb této skupiny jsou většinou přechodné HW-poruchy, řidčeji latentní SW poruchy. Tyto chyby jsou nejčastějším důvodem ke sporům mezi programátory a techniky a to, kde je chyba: v počítači nebo programu? Je to tím, že projevy těchto poruch jsou velmi podobné, včetně hlášení poskytované operačním systémem, většinou ve formě „havarijního“ výpisu registrů. V odladěném systému však půjde skoro vždy o (přechodné) HW- poruchy.

10.11.2Důsledky chyb

Všechny chyby tohoto typu mají, zejména v multiprogramovém prostředí, jeden velmi nepříjemný důsledek. Program, který je postižen ukončením, je vyřazen ze systému zpracování, cyklické programy ztrácejí cykličnost a nemohou být dále aktivovány. Je-li jediným projevem chyby výpis havarijního hlášení na monitoru a není-li k dispozici zvuková nebo světelná signalizace chyb, můžeme zjistit chybu až po delší době. Čím později, tím větší škody mohou vzniknout a tím obtížnější je zjištění příčiny chyby.
V těchto případech je třeba, abychom měli připraveny prostředky – programové vybavení – které léčí následky takových poruch. Minimálně je třeba obnovit činnost suspendovaných programů, další akce závisejí na konktétrních škodách a možnostech jejich programové likvidace. Žádoucí je identifikace chybných nebo chybějících výsledků.
Specifickou kategorii chyb (poruch) představují ty chyby, po nichž se zhroutí celý systém potichu, bez sebemenšího hlášení a bez možnosti pokračovat v jakékoli práci na počítači. (Například není možno zadat ani operátorské příkazy a zjistit minimum informací o stavu systému). Systém je pak třeba restartovat.

10.11.3Monitorování.

Zejména předchozí typy chyb zavdávají podněty ke konstrukci nejrůznějších systémů monitorování chodu aplikačních programů, aby po zhroucení programu nebo systému (říká se tomu post mortem, po smrti, protože už program skončil) bylo možno nahlédnout alespoň trochu do historie před výpadkem a pokusit se odhadnout pravděpodobnou příčinu chyby.
Použití takového systému je však náročné a problematické zároveň. Pro možnost přesné idintifikace příčiny chyby by bylo většinou nutno monitorovat daleko širší a případ od případu různorodější množinu dat, než je únosné, a sledování by bylo nutno provádět z úrovně operačního systému. Orientace v monitorovaných údajích a jejich vyhodnocení jsou potom také obtížné. Bude-li příčinou chyby nebo výpadku přechodná HW- porucha, zavede nás systém monitorování pravděpodobně na scestí.
Proto je velmi důležité, abychom při programování kladli velký důraz na spolehlivost vytvářeného aplikačního programového vybavení, na bezpečnost programování. Jde totiž především o to určit, zda v uvedených aplikacích je zdrojem poruchy HW nebo SW. A tak určitá jistota, že provozované programové vybavení neobsahuje latentní SW-poruchu, nám pomůže značně zůžit prostor hledání poruchy.
Druhý zdroj chyb 1. skupiny vzniká zanedbáním ošetření chybových situací v aplikačních programech, případně přenechání jejich ošetření operačnímu systému. To má většinou za následek vynucené ukončení programu operačním systémem.
Vynucené ukončení programu je to, čemu se zejména v ASŘTP musíme snažit vyhnout za každou cenu. Hlášení chyby, poskytnuté při takové příležitosti operačním systémem, nepředstavuje většinou snadno interpretovatelnou informaci, zejména používáme-li vyšších programovacích jazyků. Totiž výpisem nejrůznějších registrů počítače a kódů, jejichž význam někdy nebývá nikde uveden. I v případě chyb, jejichž kódy v manuálu uvedeny jsou, neinformuje jejich popis často o skutečné příčině chyby.
Ošetření chyby by mělo být tedy takové, aby bylo možno vytvořit srozumitelné hlášení o její příčině. To nemusí být vždy snadné. Cena za to je samozřejmě zvýšení pracnosti při vývoji programu. Náročnost dokonalého ošetření chyby není také často na první pohled zřejmá, zejména je-li ji nutno řešit na několika programových úrovních. To je typické zejména pro systémy pracující v reálném čase, kde je často řešení některých úkolů rozloženo mezi více programů. Z toho pak vyplývá nutnost rozložit ošetření chyby mezi programy podílející se na zpracování.
Řádné ošetření chyb však můžeme provést tehdy, jsme-li schopni všechny chyby identifikovat. Vyskytnou se i chyby,které jsou zapříčiněny občasnou HW-poruchou nebo latentní SW-poruchou. Již jsme uvedli, že pro takové případy je třeba mít připraveny

prostředky, které léčí následky, protože příčinu těchto výpadků nemusíme nikdy najít. Při jejich řídkém výskytu by takové hledání bylo statně nerentabilní, můžeme –li vytvořit programový aparát, který například obrazovku vyřazenou z konverzace systém znovu aktivuje, aniž by bylo třeba přerušovat konverzaci na dalších deseti obrazovkách pracující pod řízením konverzačního systému. Při téměř stoprocentní spolehlivosti obrazovek bude tento programový aparát téměř stoprocentně nevyužit.
Uvedené příklady ukazují, že odolnost systému lze dosáhnout zavedením nadbytečnosti (redundance) a to nejenom kvantitativně (zdvojováním stejných prvků) ale i kvalitativně. Dojdeme nakonec do zdánlivě paradoxní situace, že většina programového kódu, která bude v systému k dispozici, bude po drtivou většinu života systému téměř nebo vůbec nevyužita, zatímco po drtivou většinu života systému se bude využívat menší část tohoto kódu zajišťující jeho činnost ve standardních, nechybových situacích.

10.11.4Chyby 2. skupiny

Jsou-li náležitým způsobem ošetřeny, nepředstavují pro systém zvláštní nebezpečí. Vznikají především v případech, kdy se obracíme na operační systém se žádostí i provedení nějakých služeb, poskytnutí prostředků atp. Operační systém nás pak informuje navrácením stanovených kódů, s jakým výsledkem byla požadovaná služba provedena. Závisí pak na konkrétní situaci, které z ódů musíme považovat za chybu a které ne.
Uvedené schéma se uplatňuje v řadě situací., kdy žádáme operační systém o přidělení určitých prostředků nutných k provedení žádajícího programu. Může jít například o přidělení určité oblasti paměti, přidělení zdroje atp. Požadavky mohou přitom být dvojího druhu. Buď bezpodmínečně a pak bude čekat program ve stavu odložení tak dlouho, dokud operační systém nebude ve stavu jeho požadavku vyhovět. Anebo je požadavek podmínečný a pak, není-li mu možno vyhovět, vrátí operační systém programu informaci o nesplnění požadavku a je věcí žádajícího programu (programátora!), aby se zařídil, jak umí.
Tento přístup počítá s vynalézavostí programátora v konkrétní situaci a komplikuje strukturu programu. První přístup nechává vše na operačním systému, avšak může připustit situaci, že nebude možno uspokojit všechny požadavky kladené uživatelskými programy na výpočetní prostředky systému, což může vést k havarijnímu stavu.

10.11.5Chyby 3, skupiny . aplikační úroveň

Jsou to chyby, za jejichž vznik i likvidaci jsou plně odpovědni tvůrci aplikačního programového vybavení. Až snad na vyjímečné případy, kdy i tyto chyby mohou mít příčinu v HW-poruše. V dobře odladěném systému by se neměly za normálních okolností vyskytnout. Jde o chyby vznikající zasláním falešných zpráv nebo nesprávných parametrů jiným programům. Obrana spočívá především v důsledném testování hodnoty každého parametru, které program dostává cestou komunikace s jinými programy nebo jako vstupní data.
Způsob kontroly, případně ošetření chyby, bude záviset na charakteru vstupující informace a podle toho též chyba interpretována. Při zadávání dat, např. z displeje, spočívá kontrola dat v testování mezí a ve vhodně uvedeném způsobu konverzace. Například je účelné přijaté parametry znovu zobrazit, aby si zadávající uvědomil, že zadaná data (i když jsou v zadaných mezích) jsou skutečně ta, která chtěl zadat a přijmout tyto data doopravdy až teprve po opětovném potvrzení jejich správnosti.
U dat získávaných měřením (analogové vstupy, číslicové vstupy) bude způsob kontroly přirozeně jiný. Vezměme jako příklad zásobník uhlí, a vněmž jsou instalována tři čidla – číslicové vstupy – indikující tři úrovně zaplnění zásobníku; minimální, střední a maximální. Při zaplnění zásobníku na maximální úroveň musí být signalizováno zaplnění na obou nižších úrovních. Jestliže tomu tak není, půjde o poruchu některého z čidel. Může však výjimečně dojít k anomálnímu rozložení uhlí v těsném okolí čidla (je tam díra). To však programem

těžko zjistíme a jediné, co můžeme udělat, je vyslání varovného hlášení na obrazovku kompetentního pracoviště.
Popisovaná situace dovoluje provést logickou kontrolu sejmutých hodnot a tím do určité míry i testovat jejich správnost. Provedenou kontrolu však nejsme schopni pokrýt všechny možné poruchy, stejně jako nejsme schopni odlišit zcela normální, byť výjimečnou situaci – „díru“. Zdvojením, případně trojením příslušných čidel bychom vytvořili podmínky pro téměř dokonalou kontrolu. Je však otázkou za jakou cenu a zda v popisované situaci je to vůbec nutné a účelné. Takové případy je nutno řešit vždy individuálně.

10.11.6Chyby 4. skupiny

Jde o chyby technologické obsluhy způsobené za provozu systému zadáním nesprávných údajů nebo nesprávnou manipulací s přístroji. Zde musí programátor prokázat značnou dávku předvídavosti, aby správně předpověděl co nejvíce druhů chyb, kterých se obsluha v budoucnosti může dopustit s zajistit ošetření těchto chyb tak, aby se příliš negativně neprojevily v následném chodu systému.
Opomenout některou z možností je snadné. Podrobněji je vidět možné chybové situace až pří vlastním programování. Na hotovém programu můžeme přesněji vidět jeho slabá místa. Slabá v tom. že mohou být poznamenána nevhodnou konverzací nebo manipulací. I zde po celou dobu provozování programu nemusí k některým předpověděným chybám vůbec dojít a vytvořený chybový kód nebude nikdy ohlášen. (Spíše se vyskytnou chyby, na které se zapomnělo).
V případě, že obsluha doplňuje konverzační údaje, které nejsou časově závislé ani nezáleží na pořadí jejich vzájemného zadávání, nemusí být kontrola těchto údajů příliš složitá. I tak však, může část programu obsahující ošetření chyb značně převyšovat objem části zpracovávající bezchybně zavedené údaje.
V případě, že existuje časová závislost mezi zadáváním údajů (například z různých pracovišť), může to mít podstatný vliv nejen na náročnost zpracování chyb, ale i na normální způsob zpracování. Při normálním průběhu zpracování je pak totiž třeba sbírat a vytvářet další údaje, které se uplatní jen pro účely kontroly a ošetření chyb. Tím vzniká nadbytečnost údajů, potřebná pro vytvoření algoritmů, zajišťujících spolehlivost systému.
Podobná je situace při řešení úlohy restartu, která vyžaduje průběžný sběr dat v normální situaci redundantních, která však využijeme v případě, že je nutno systém z jakýchkoli důvodů restartovat.
K provedení potřebných algoritmů pro situace ošetření chyb i restartu v právě popsaných situacích je třeba důkladných znalostí jak technologie a organizace práce ve sledovaném provozu, tak možností a prostředků použité výpočetní techniky (a programování).
Zpracovat takovou analýzu před započetím prací na aplikačním programovém vybavení je ve složitějším případě neproveditelné. Takový systém je nutno budovat postupně, za současné práce na aplikačním programovém vybavení. Závazně musí být ale předem stanovena koncepce, jakým způsobem se bude postupně vytvářet a jaké společné programové prostředky v něm busou použity. Celou práci může usnadnit vhodně provedená dekomozice na programové subsystémy.

10.12 OŠETŘENÍ CHYB V RT- SYSTÉMECH

Již jsme řekli, že na rozdíl od běžných, dávkových vědeckotechnických nebo komerčních aplikací, musejí procesy a výpočty v RT-systémech probíhat nepřetržitě i v případech, kdy dojde k chybě. Při nesprávném chodu nějakého procesu s neošetřením všech možných chybových situací by totiž mohlo dojít nejenom k zastavení tohoto procesu, ale k zastavení celého systému RT-programů (t. j. procesů).

Tuto situaci není možné nechat dopustit nejen při řízení technologických procesů v reálném čase, ale není žádoucí ani v rozsáhlejších multiuživatelských systémech reálného času, např. informačních, protože zde může probíhat současně řada transakcí, které při výpadku budou ztraceny a jejich rekonstrukce může být velmi problematická, ne-li nemožná.
To znamená, že RT – systémy musejí obsahovat prostředky a programy pro ošetření chyb a že tedy tyto prostředky musejí být obsaženy v RT-jazycích.
Ošetření chyby má dvě fáze: detekci chyby a reakci na chybovou situaci.
Detekci chyby je možno provést, dle situace , na dvou úrovních: implementační a aplikační.
Na implementační úrovni lze detekovat chyby pomocí technických prostředků a ke jejich ošetření slouží pomocné sekvence instrukcí generovaných kompilátorem. Příkladem takové chyby může být dělení nulou (obecně přetečení overflow při operacích v pevné řádové čárce) nebo překročení indexu pole atp.
Chyby na aplikační úrovni jsou takové chyby, které se objeví ve vrstvě aplikačního programového vybavení, Jejich příčinou bude většinou logicky nesprávné zpracování programu programátorem., který opomene možnost výskytu určitých chybových situací, nebo nepředpokládá, že by se vůbec mohly určité chybové situace vyskytnout a neošetří je.
Na implementační úrovni programu se ošetřují chyby obtížněji, protože mohou vzniknout na různých místech programu. a je velmi obtížné zjistit adekvátní reakci programu na chybu.
Jestliže v programech dávkových je adekvátní reakcí na chybu výpis chybového hlášení obsahující její identifikaci, pokud možno i lokalizaci a ukončení programu, pak v RT – systémech tomu tak není. Zde musí zahrnout odpovídající odezva na chybu i tzv, zotavení systému, které by mělo spočívat v opravě chybou dotčených výsledků a zajištění další normální funkce systému.
Každý mechanismus ošetření chyb v programovacím jazyce by měl být:

a) jednoduchý a srozumitelný z hlediska použití
b) nepříliš rozsáhlý, aby nenarušil srozumitelnost a přehlednost těch částí programu, které vykonávají vlastní algoritmus
c) časově nenáročný, aby nevyžadoval příliš velkou časovou režii v případech, kdy se chyba nevyskytuje
d) umožňovat jednotné ošetření chyb, detekovaných na implementační i aplikační úrovni
e) poskytnout zotavovací prostředky pro naprogramování zotavovacích akcí

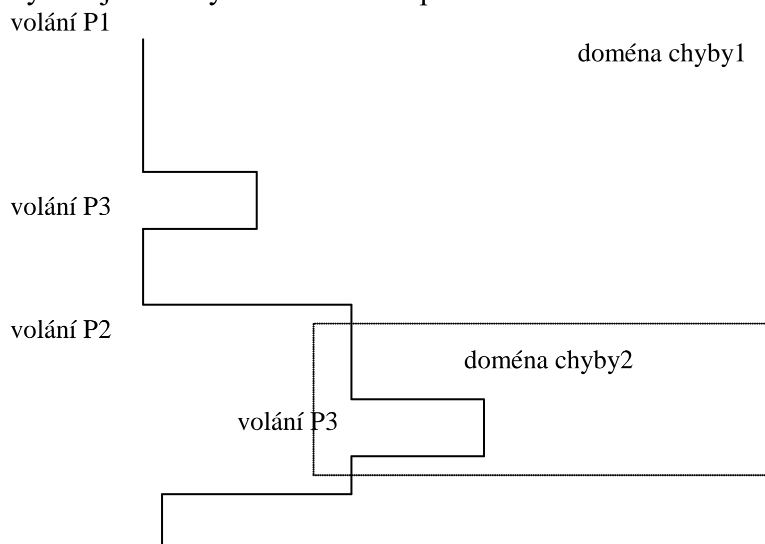
10.13 KLASICKÉ PROSTŘEDKY PRO OŠETŘENÍ CHYB

Jednou z nejjednodušších metod je využití parametrů procedury k předávání kódu chyby volajícímu programu. Po dokončení procedury se tento kód testuje a v závislosti na jeho hodnotě zajišťuje další zpracování chyby. Hlavní výhodou tohoto přístupu je jeho jednoduchost, která umožňuje při důsledném využití jednotný způsob zpracování chyb, jeho soustředění do jednoho nebo poměrně malého počtu jasně vymezených podprogramů a hlavně umožňuje to, že stejný druh chyby může být v různých podprogramech, nebo i částech programu ošetřen různým způsobem v závislosti na tom, jak je ta která chyba v daném místě závažná.

10.13.1Obsluhy výjimek

Jednoduchý příkaz GOTO v kombinaci s proměnnými typu návěští (LABEL) poskytuje efektivní mechanismus umožňující ošetření každé konkrétní chyby v závislosti na kontextu v jakém tato chyba nastala, avšak je to mechanismus s nízkou úrovní abstrakce a také je žádoucí nahradit použití příkazu GOTO „čištšími“ strukturovanými příkazy jako například IF, CASE, WHILE atd.
Předpokládejme, že při výpočtu vznikne chyba E. Pro ošetření chyby E může existovat několik alternativ obslužných příkazů – obsluh chyb. Předpokládejme nyní, že budeme mít k

dispozici mechanismus, který vyvolá po výskytu chyby odpovídající odezvu – obsluhu chyby. Předpokládejme dále, že jsme schopni s každou obsluhou chyby v programu asociovat určitou oblast programu, tzv. doménu, která specifikuje tu část výpočtu, v němž se při výskytu chyby aktivizuje k ošetření chyby právě tato obsluha.
Struktura systému ošetření chyb pak závisí především na pojmu doména chyby. Technicky pak spočívá problém v tom, jak vymezit jednoznačně a přesně oblast programu, doménu, svázanou s konkrétní obsluhou chyby. Je přirozeným požadavek, aby taková doména představovala určitou logickou část programu určenou pro zajištění určité logické funkce programu. Takovým přirozeným úsekem programu bývá funkce, podprogram, modul, resp. blok a proto bude přirozené vytvářet domény tak, aby odpovídaly těmto programovým strukturám i s tím důsledkem, že návrh programu bude třeba podřídit i hledisku ošetření chyb vyžadujícímu vymezení domén pomocí těchto struktur.



Je zřejmé, že chyba, která vznikne v proceduře P3 bude ošetřena jiným způsobem, bude-li procedura volána z procedury P1, než při volání P3 z procedury P2. Ošetření chyb má tedy určitý dynamický charakter, který se projevuje tak, že stejná chyba vzniklá v proceduře P3 je různým způsobem ošetřena v závislosti na tom, z jaké domény je podprogram volán.
Použití těchto prostředků předpokládá, že budou v jazyce zavedeny prostředky pro definici výjimek, definování domén pro jejich aktivaci. Z důvodů nedostupnosti konkrétní implementace však dále neuvádíme další podrobnosti a předchozí text považujeme pouze za informativní, informující o vývojových snahách v této oblasti.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Komunikace a synchronizace procesů

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

11 KOMUNIKACE A SYNCHRONIZACE PROCESŮ 3
11.1 Zasílání zpráv 4
11.2 Monitory 4
11.3 Realizace některých synchronizačních nástrojů. 4
11.3.1 Aktivní čekání 4
11.3.2 Pasivní čekání 5
11.4 Nepodmíněný zákaz přepínání kontextu 5
11.5 Podmíněný zákaz přepínání kontextu 5
11.6 Semafor 5
11.7 Signál 6
11.8 Kritické sekce, vzájemné vyloučení 6
11.9 Producent - konzument 6
11.10 Čtenáři - pisaři 7
11.11 Souběh (rendez-vous) 7
11.12 Uváznutí a stárnutí 8

11 KOMUNIKACE A SYNCHRONIZACE PROCESŮ

Obsah kapitoly:

Komunikace a synchronizace procesů

CÍL:

Po prostudování tohoto odstavce budete umět:

- definovat zasílaní zpráv, monitory, semafor, signál, kritické sekce vzájemného vyloučení, souběh, uváznutí a stárnutí
- popsat realizaci některých synchronizačních nástrojů, podmíněný a nepodmíněný zákaz přepínání kontextu

11.1 ZASÍLÁNÍ ZPRÁV

Technika zasíláni zpráv představuje poměrné jednoduchou abstrakci komunikace procesů spojenou s jejich synchronizací. Komunikace pomocí zasílání zpráv probíhá tak, že jeden proces vyšle jednou operací posloupnost bitů konečné délky (zašle zprávu) a druhý proces ji přijme. Potřebné synchronizační operace jsou tedy typicky dvě procedury

SEND MESSAGE(...) pro zaslání zprávy a
RECEIVE MESSAGE(...) pro přijetí zprávy.

Konkrétní operace se liší dle toho, jak je určen druhý proces s ním se komunikuje - tzv. adresací a dále tím, probíhá-li komunikace synchronně nebo asynchronně.
Adresace se nazývá symetrická, uvádí-li se při komunikaci jméno vysílajícího i přijímajících asymetrická, uvádí-li se jen jméno přijímajícího procesu. Jméno vysílajícího procesu se určí při přijetí zprávy dotazem přes parametr. Adresace se nazývá nepřímá, uvádí-li se jméno použité vyrovnávací paměti nebo komunikačního kanálu.
Asynchronní vysílání vyžaduje vyrovnávací paměť o určité kapacitě. Technika výběru se může realizovat bud' prostřednictvím schránek paměti, staticky, přičemž jsme omezeni předem zvolenou kapacitou vyrovnávací paměti. Rafinovaněji způsob představuje technika výběru pomocí fronty, dynamická, kde je kapacita vyrovnávací paměti omezena pouze kapacitou operační paměti dostupnou pro dynamické přidělování paměti.
Synchronní operace RECEIVE obsahuje čekání na příchod zprávy a operace SEND čekání na příchod potvrzení o přijetí vyslané zprávy, resp. potvrzení zahájení komunikace. Při tomto režimu komunikace stačí jedna vyrovnávací paměť s kapacitou maximálně jedné zprávy.

11.2 MONITORY

Základní myšlenkou monitoru je spojit deklaraci společné proměnné - resp. společné datové struktury - s explicitní definicí všech operací, které lze na ní provádět a současně stanovit, že tyto operace jsou zároveň vyloučeny. Koncepce monitoru vychází a myšlenky abstraktního datového typu a rozšiřuje ji o vzájemné vyloučení operací.
Při implementaci je třeba ještě dodat 2 operace realizované na úrovni operačního systému

vstup_do_monitoru(monitor M);
výstup_z monitoru(monitor M);

pomocí nichž synchronizujeme operace p1, p2,...,pn v monitoru tak, aby probíhaly ve vzájemném vyloučení. To je však velmi silná podmínka protože často stačí jestliže například pomocí zdroje synchronizujeme je některé operace.

11.3 REALIZACE NĚKTERÝCH SYNCHRONIZAČNÍCH NÁSTROJŮ.

Již jsme uvedli, že pro synchronizaci procesů se využívají tyto dva základní principy:

A. Aktivní čekání, kdy dochází k tomu, že do procesu vkládáme pomocné prázdné akce, čímž dojde ke "vzdálení" zakázané oblasti tvořené průnikem kritických oblastí.
B. Pasivní čekání, při němž dochází k pozastavení všech konkurujících virtuálních procesorů a tím k pozastavení jim odpovídajících procesů.

11.3.1 Aktivní čekání

Bylo již řečeno, že technika aktivního čekání předpokládá existenci pomocných prázdných akci, které vkládáme do procesu, abychom dosáhli odsunu zakázané oblasti. Do kterého procesu? Samozřejmě že do toho, který se hlásí o vstup do zakázané oblasti jako druhý. Tento druhý proces(a stejně tak další procesy, pokud jich je více) musí prvnímu procesu poskytnout čas, aby mohl dokončit operace v zakázané oblasti.

11.3.2 Pasivní čekání

Základní myšlenka techniky pasivního čekání je tato: Jsou-li virtuální procesory vytvářeny přepínáním kontextu, které proběhne při přerušení generovaném časovačem, dosáhneme cíle tím, že zamezíme přepínání kontextu. Zákaz lze realizovat ve dvou variantách: jako nepodmíněný nebo jako podmíněný.

11.4 NEPODMÍNĚNÝ ZÁKAZ PŘEPÍNÁNÍ KONTEXTU

Nejsilnější, avšak zároveň nejméně vhodný způsob realizace zákazu přepnutí kontextu, je vydání zákazu přerušení vůbec, protože je tím znemožněno i přerušení od časovače. DI (Disable Interrupt) a DI (Enable Interrupt).
Těchto funkcí však můžeme používat jen při psaní ovladačů periferních zařízení resp. při ovládání přerušovacího systému. Zákaz přerušení DI, tj. , odblokování celého přerušovacího systému, je možno vydat jen na minimální, nezbytně nutnou dobu, abychom neztratili přerušení od nezávisle pracujících periferií. Podrobněji je o tom pojednáno v kapitole o přerušovacím systému.
Druhá možnost spočívá v zákazu přepnutí kontextu, který si může vynutit proces. Ktomuto účelu jsou některé systémy vybaveny dvojicí funkcí Lock a Unlock. Proces, který vydá přikaz Lock způsobí ´uzamčení´ procesoru pro své vlastní potřeby na úkor jiných procesů a instrukcí Unlock dává procesor k dispozici ostatním procesům.
Funkce Lock a Unlock odpovídají přesně dvojici operacíK a Z. Na rozdíl od DI a EI však během uzamčení normálně funguje přerušovací systém, takže může například probíhat přenos dat po sériových linkách do té úrovně, pokud je příslušné zpracování součástí ovladače sériových portů.
I zde však je nutno dodržovat zásadu, že process uzamyká procesor na co nejkratší dobu, jaká je nezbytná pro zpracování určitého úseku kódu bez případné intervence jiného procesu. Pak je třeba procesor uvolnit a okamžitě o něj třeba hned požádat, následuje-li další nedělitelná sekvence instrukcí. Toto uvolnění je nezbytné proto, aby proces příliš nemonopolizoval procesor, což je neslučitelné s prioritní strategií přidělování procesoru jednotlivým procesům. Instrukcí Lock může totiž proces i s tou nejnižší prioritou dosáhnout na libovolně dlouhou dobu toho, aby byl procesor přidělen procesům s vyšší prioritou.
Současně vidíme další nevýhodu nepodmíněného zákazu přerušení: zbytečně se zastaví i ty procesy, které nemají nic společného se zakázanou oblastí do níž vstoupil první proces a tedy by čekat vlastně nemusely. To je důvodem pro zavedení techniky podmíněného zákazu přepínání kontextu.

11.5 PODMÍNĚNÝ ZÁKAZ PŘEPÍNÁNÍ KONTEXTU

Základní myšlenka podmíněného zákazu přepnutí kontextu spočívá v tom, že pro různé zakázané oblasti budou definovány a používány různé synchronizační prostředky. Pak lze postupem analogickým předešlému vynutit si zákaz přepnutí kontextu jen pro omezenou množinu procesů. Proto podmíněný zákaz. Procesy které nemají se zakázanou oblastí, která je právě " ve hře" nic společného, nemusejí být omezovány víc, než je dáno obecnými pravidly přidělování procesoru v daném systému plánování procesů.
Dva nejjednodušší synchronizační nástroje pra realizaci podmíněného zákazu přepnutí kontextu procesů představují Semafor a Signál.

11.6 SEMAFOR

Semafor je v podstatě dvouhodnotová proměnná, jejímž úkolem je udržovat informací o stavu jí přiřazené zakázané oblasti, tuto informaci, zda do ní některý proces vstoupil. Nechť tedy

například hodnota OBSAZENY znamená, že zakázaná oblast je "obsazena", hodnota VOLNY, že je do ní možno vstoupit.
Pro práci se semafory zavedeme dvě systémové operace SECURE a RELEASE jakožto konkrétní implementace obecných synchronizačních operací Z a K. Definujme nyní datový typ Semafor a operace SECURE a RELEASE.

11.7 SIGNÁL

Od signálu budeme očekávat především odstranění nežádoucí režie spojené s aktivním čekáním. Přesunutím kontroly a řízením operací se signály do nadřazené úrovně se vytváří možnost vytvořit prostředek obecnějšího použití a vlastností než má semafor.
Problematikou synchronizace se totiž doposud zabýváme na základě modelové situace představované kritickou, resp. zakázanou oblastí v souvislosti s problémem výlučného přístupu. O něco obecnější situaci, kdy je třeba synchronizovat činnost dvou neb více procesů představuje úloha, kdy jeden z procesů musí čekat než jiný proces provede nějakou akci na jejímž základě může teprve pokračovat v činnosti, Přitom první proces nemusí být vázán na, stav druhého procesu a může pracovat na něm nezávisle. Může tedy tuto akci provést v předstihu, v okamžiku, kdy na ni ještě druhý proces nečeká a ten se musí dovědět o tom, že akce byl, provedena v okamžiku, kdy se dostane do místa, stavu, kdy bude potřebovat její výsledek a kdy na ní bude muset čekat, pokud ne bude provedena.
V této situaci vystupuje jeden a procesů jako zákazník, klient, druhý proces jako obsluha, server (Latinsky servus znamená otrok). Provedením služby vytváří server událost, na níž čeká klient. Záleží na situaci, byla-li tato o událost klientem vyžádána anebo jde-li o událost na niž může čekat více procesů, více klientů.
Událost budeme reprezentovat synchronizačním nástrojem signál. Pro jeho zpracování vytvoříme tři operace. SEND pro ohlášení události, aktivaci signálu, WAIT pro čekání na aktivaci signálu a INIT pro počáteční nastaveni jeho hodnoty.

11.8 KRITICKÉ SEKCE, VZÁJEMNÉ VYLOUČENÍ

Kritická oblast programu je taková oblast programu, která má být provedena při vzájemném vyloučení s jinou kritickou oblastí jiného programu. Realizovat kritické sekce tedy znamená synchronizovat prováděni příkazů tak, aby došlo k požadovanému vzájemnému vyloučení. Petriho síť specifikující tuto úlohu je znázorněna na obrázku OBRxxi. Synchronizace je založena na této myšlence. Každý a procesů Pi odebere při přechodu VSTUPi ae strážního místa SM tečku. Pokud není v místě SM tečka, přechod se nemůže realizovat. Vidíme, že žádné dva z příkazů KSi se nemohou provést současně, takže jimi řízené procesy jsou vzájemně vyloučeny.
V programech je často třeba uplatnit princip kritických sekcí na více místech a my budeme nazývat všechny kritické sekce, které mají být provedeny při vzájemném vyloučení, sdruženými kritickými sekcemi.

11.9 PRODUCENT - KONZUMENT

V této úloze jde o asynchronní komunikaci procesů pomocí společné paměti konečné délky (výraz asynchronní znamená, že oba procesy na sobě při předávání dat nečekají).
Schéma úlohy je uvedeno na na následujícím obrázku. Producent a konzument mají společnou vyrovnávací paměť s kapacitu 3 položky.

1 1(4,7 ..)
Producent -----> ------> Konzument 3(6,9..) 2 2(5,8..)

Proces Producent sekvenčně vytváří jednotlivé položky očíslované 1 , 2, 3, 4, 5, 6, 7, 8, 9, . . . atd a zapisuje je postupné do vyrovnávací paměti do míst v pořadí 1 , 2, 3, 1 , 2, 3, 1 , 2, 3, . . . atd . Proces Konzument v témž pořadí tyto položky odebírá a zpracovává.
Synchronizaci procesů Producent a Konzument je třeba zabránit tomu, aby proces Konzument nepředběhl při výběru proces Producent, tj. aby neodebral ještě nevytvořenou položku. A naopak aby proces Producent nepředběhl proces Konzument "o jedno kolo", tj. aby nepřepsal ještě neodebranou položku.
Specifikace úlohy Producent-Konzument je na OBRxx2. Princip modelu spočívá v použití dvou pomocných míst VOLNÉ (případně ČEKÁ) a OBSAZENÉ, v nichž počet teček odpovídá počtu volných, případně obsazených míst. Proto proces Producent odebere před zápisem jednu tečku z místa volné a po zápisu ji přidá do místa obsazené . Podobně proces Konzument odebírá před čtením z vyrovnávací paměti tečku z místa OBSAZENÉ, případně zde čeká. Tečku vrátí do místa VOLNÉ.
Úloha p-K má výrazné praktické aplikace - je podstatou každé asynchronní komunikace procesů. například v ovladačích sekvenčně pracujících periferii, u kterých má smysl vyrovnávat rozdílnou rychlost vlastního zařízení a toku požadavků na provedení operace.

11.10 ČTENÁŘI - PISAŘI

Často se vyskytuje tato úloha v databázových systémech. Její podstatu lze formulovat takto: n procesů = čtenáři a m procesů - písaři má přistup ke společné datové struktuře. Čtenáři mají tento přístup jen operací čtení, písaři operaci zápisu. Je třeba synchronizovat tak, aby mohly operace čteni probíhat současně, ale přitom byly vzájemně vyloučeny s operacemi zápisu. Současné musejí být vyloučeny i zápisové operace.
Procesy je třeba synchronizovat tak aby operace čtení mohly probíhat současně, ale aby přitom byly vzájemně vyloučeny s operacemi zápisu. Vzájemně musí být také vyloučeny všechny zápisové operace. Jinak řečeno každý písař musí počkat na dokončení všech operaci čtení a zápisu.
Specifikace této úlohy Petriho sítí je na obrázku OBRxx3. Před každou operací čtení odebere čtenář jednu tečku ze strážního místa ZÁPIS_MOŽNÝ a po skončení operace ji sem vrátí. Každý proces PÍSAŘ odebere ze strážního místa před zápisem celkem n teček (n je počet 'čtenářů) .
Teček je tolik, kolik je čtenářů, aby mohli všichni najednou číst. Nespravedlivé je toto řešení a hlediska písaře, když čeká pak čtenáři mohou zahajovat operace čtení. Zároveň toto řešení předpokládá, že písařů není dvakrát více než čtenářů. Pak by nebyly vyloučeny například dvě operace zápisu, nebo by jinak musel proces písař odebírat více teček, než je čtenářů.

11.11 SOUBĚH (RENDEZ-VOUS)

Podstatou souběhu je společné provedení určitých úseků programů dvěma procesy. Procesy je nutno synchronizovat tak, aby před společným úsekem (provedením společných úseků) počkal jeden proces na druhý a ve společném úseku pak postupovaly oba procesy stejnou rychlostí. Některé programovací jazyky mají pro vyjádření souběhu přímo zabudovány prostředky, například ADA.
Specifikace této úlohy pomocí Petriho sítě má navodit představu, že zápis společného úseku je začleněn do programu řídicího jeden z procesů. V postupovém prostoru pak části odpovídající souběhu odpovídá úsečka, jejíž směrnice je vždy i, což z hlediska procesů vypadá jakoby nedocházelo k přepínáni kontextů.
Aplikace tohoto modelu se vyskytuje nejčastěji u paralelních procesů probíhajících v distribuovaném prostředí. Pro toto prostředí je charakteristická spolupráce více procesů pomoci komunikačních kanálů bez využívání společné paměti.

11.12 UVÁZNUTÍ A STÁRNUTÍ

K uváznutí procesu dochází tehdy, jestliže vlivem použití synchronizačních nástrojů nemůže proces dále postupovat. V této situaci se může ocitnou obecně více procesů.
Ke stárnutí procesu dochází tehdy, jestliže vlivem nesprávně použitých synchronizačních nástrojů některý z procesů je více předbíhán ostatními procesy a nepokračuje v postupu.
K uváznuti procesu může typicky dojít při nepozorném,. resp. nesprávném použití synchronizačních operaci, nepřiklad tehdy, když při výstupu z kritické oblasti jeden z procesů neuvolni semafor. Následkem toho druhý proces před vstupem do kritické oblasti uvázne. Typicky vznikají tyto situace při překryti dvojic sdružených kritických oblastí, kde vzniká požadavek ne současné použiti dvou společných prostředků.
Petriho sítě a jejich použíti pro znázornění synchronizace.
Jedním z nejdůležitějších pojmů, který se objevuje nejenom v souvislosti s paralelními výpočetními architekturami ale například v některých diskrétních simulačních jazycích, je pojem proces. jako dekompoziční jednotka systému tvořeného posloupností událostí.
Tato dekompoziční jednotka je zvláště vhodná pro takové systémy, v nichž se objevují synchronní i asynchronní činnosti v rámci vnitřního paralelismu jednotlivých často násobných, komponent systému. V těchto systémech, označovaných jako paralelní systémy, lze identifikaci elementárních procesů, případně stanovením jejich hierarchií a určením 2působu jejich komunikace, podstatně zjednodušit popis i značně komplexních systémů.
Úvahy o isomorfismu mezi subsystémy objektů reálného světa ne jedné straně a jejich popisem a počítačovým modelem na straně druhé jakož i praktické zkušenosti ukazují, že k současným i perspektivním výpočetním systémům je výhodné přistupovat právě jako k paralelním systémům.
Pojem paralelní systém je odvozen od způsobu dekompozice diskrétních systémů na relativné samostatné funkční jednotky procesy, které probíhají paralelné v čase a které spolu komunikují v rámci vzájemných interakcí.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Petriho sítě

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

12 PETRIHO SÍTĚ 3
12.1 Petriho sítě 4
12.2 Graf p-sítě 5
12.3 Evoluční pravidla 5
12.4 Synchronizační nástroje 10

12 PETRIHO SÍTĚ

Obsah kapitoly:

definujte evoluční pravidla

popište Petriho sítě

CÍL:

Po prostudování tohoto odstavce budete umět:

- definovat evoluční pravidla
- popsat Petriho sítě

12.1 PETRIHO SÍTĚ

Princip abstrakce, jenž je snad vůbec nejdůležitějším principem modelováni, vedl v sedmdesátých letech k vytvoření řady matematických modelů paralelních systémů. Účelem těchto modelů je popis abstrakce paralelního systému z hlediska koordinace a synchronizace procesů. Tato abstrakce se označuje jako řídicí struktura paralelního systému. K nejznámějším matematickým modelům paralelních systémů patři Petriho síť.

V této kapitole se seznámíme s řídicí strukturou paralelního systému ve tvaru p-sítě, jež představuje unifikovaný model řídicí struktury téměř identický s Petriho sítí. Tohoto modelu budeme využívat pro popis synchronizačních úloh, kde p-sít představuje jakýsi "vývojový diagram" synchronizační úlohy, jenž poskytuje návod pro její strukturalizaci programu i popis paralelnosti a násobnosti procesů.

Definice p-sítě

p-sít je pětice R = <P, T, ∑, S, F> kde
P = (P1, ..., pn) je množina míst
T = (ti, ..., tm) je množina přechodů
∑ = je abeceda operaci
S Є P   je počáteční místo
F Є P   je koncové místo.
Každý přechod tj Є T je uspořádaná trojice
tj = (σj, Ij, Oj)
kde   σj   je operace přislušející přechodu tj,
Ij   je multimnožina vstupních míst přechodu tj
Oj   je multimnožina výstupních míst přechodu tj

Multimnožina se od množiny liší tím, že připouští více výskytů prvků množiny. Typickým příkladem multimnožiny je rozklad přirozeného čísla na prvočinitele, kde prvky multimnožiny jsou jednotliví prvočinitelé. Počet výskytů prvku x v multimnožině M je dán funkci #(x, M).

Platí

#(x, M)≥0; je-li #(x, M) = 0, pak neplatí, že x Є M.

Příklad:

Uvažujeme p-sít Ri (která: modeluje řídicí strukturu jednoduchého počítačového systému).
R1 - < {p1, P2, P3, P4, P5, P6, P7, P8},
{ tl, t2, t3, t4, t5, t6},
{a, b, c, d, e, f},
P1, P8>
kde prvky množiny T jsou tyto trojice:
t1 = (a, {p1}, {p2, p7, p4})
t2 = (b, {p2}, {p2, p3})
t3 = (c, {p3, p4}, {p5})
t4 = (d, {p5}, {p6, p7})
t5 = (e, {p6, p7}, {p7})
t6 = (f, {p7}, {p8})

Druhými a třetími složkami přechodu tj jsou v tomto případě množiny, nikoliv multimnožiny.

Přiklad: Příklad přechodu s multimnožinou vstupních a výstupních míst

t = (σ, {p1, p1, p1, p2, p2} , {p3, p3})

Pro snažší referenci jednotlivých složek přechodu tj zavedeme trojici funkcí přechodu tj; vstupní funkci I, výstupní funkci O a vyhodnocovací funkci σ, jejichž hodnotami jsou multimnožina výstupních míst, resp. prvek operací ∑

I(tj) = Ij, O(tj) = Oj a σ(tj) = σj

Vstupní a výstupní funkce přechodu jsou definovány jako multimnožiny míst. Analogicky definujeme vstupní a výstupní funkce místa pk, I(pk) a O(pk), jako množiny, pro něž platí

(tj, I(pk)) = # (pk, O(tj))
(tj, O(pk)) = # (pk, I(tj))

pro všechna pk Є P a Tj Є T

Příklad:

Pro p-sít Ri (viz příklad 1) dostáváme tyto vstupní,výstupní a vyhodnocovací funkce přechodů a vstupní a výstupní funkce míst:

I(t1) = {p1}	O(t1) = {p2, p7, p4}	σ(t1) = a
I(t2) = {p2}	O(t2) = {p2, p3}	σ(t2) = b
I(t3) = {p3, p4}	O(t3) = {p5}	σ(t3) = c
I(t4) = {p5}	O(t4) = {p6, p4}	σ(t4) = d
I(t5) = {p5, p6}	O(t5) = {p7}	σ(t5) = e
I(t7) = {p7}	O(t6) = {p8}	σ(t1) = f
I(p1) = O	O(p1) = {t1}	
I(p2) = {t1, t2}	O(p2) = {t2}	
I(p3) = {t2}	O(p3) = {t3}	
I(p4) = {t1, t4}	O(p4) = {t3}	
I(p5) = {t3}	O(p5) = {t4}	
I(p6) = {t4}	O(p6) = {t5}	
I(p7) = {t1, t5}	O(p7) = {t5, t6}	
I(p8) = {t6}	O(p8) = 0	

12.2 GRAF P-SÍTĚ

Graf p-sítě je orientovaný multigraf, jehož množina uzlů je sjednoceni množiny míst P a množiny přechodů T, a jehož množina hran je definována takto:

1. pro každý výskyt místa pk v multimnožině O(tj) vede hrana z přechodu tj do místa pk - výstupni hrana přechodu tj
2. pro každý výskyt místa pk v multimnožiné I(tj) vede hrana z místa pk do přechodu tj - vstupní hrana přechodu tj

Graficky jsou uzly příslušející místům a přechodům p-síti rozlišeny: místa jsou označována kroužkem a přechody úsečkou.

Evoluční pravidla p-sítě

Provádění p-sítě jako abstraktního stroje je řízené existenci příznaků, které v určitém počtu příslušejí každému místu p-sítě. Graficky se příznaky označují jako černé tečky umístěné v místech p-sítě. Dříve, než tato pravidla uvedeme, je nutné definovat dva nové pojmy.

Přechod ty se nazývá realizovatelný přechod, jestliže pro každé pk (leží) P existuje alespoň #(pk,I(tj)) příznaků v místě pk. Tzn., že vstupní místa přechodu ty mají tolik příznaků, že lze každé vstupní hraně přiřadit jeden příznak.

Přechod ty je realizován, jestliže se odstraní #(pk, I(tj)) příznaků z každého vstupního místa a přidá #(pi, O(tj)) příznaků každému výstupnímu místu pi. Realizován může být pouze realizovatelný přechod.

12.3 EVOLUČNÍ PRAVIDLA

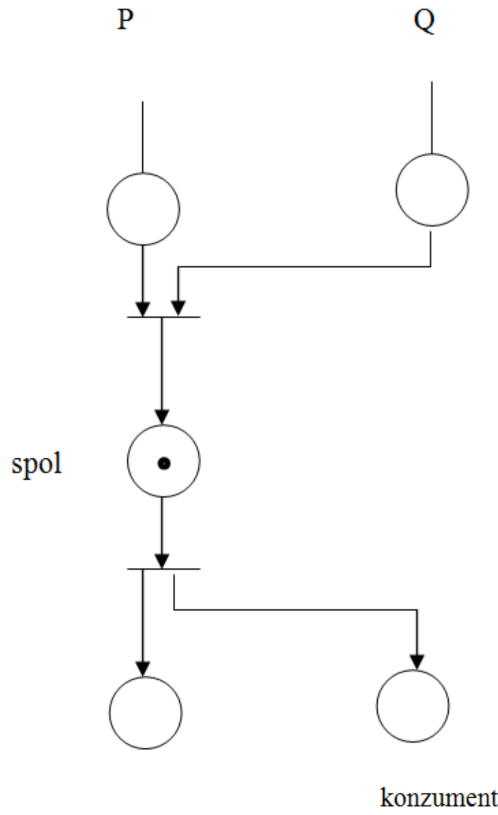
1. p-sít je inicializována umístěním jednoho příznaku do počátečního místa S
2. Vypočítá se množina realizovatelných přechodů W, W ≤ T
3. je-li W # 0, pak je vybrán jeden z realizovatelných přechodů z W a je realizován. V opačném případě provádění p-sítě skončilo.

Kroky (2) a (3) jsou opakovaně prováděny tak dlouho, až neexistuje žádný realizovatelný přechod. Kdykoliv je přechod realizován mění se počet nebo umístění příznaků v místech p-sítě a tím se mění stav p-sítě.
Příklad: Aplikaci evolučních pravidel ukážeme na příkladě p-sítě R2, zadané grafem p-sítě (obr.). Počáteční, resp. koncové místo je místo P1 , resp. p8
Modelováni prostřednictvím p-sítí p-sítě jako ředicí struktury modelů paralelních systémů se používají ve funkci modelů na takové hladině abstrakce, jež je vhodná pro popis a řízení interakci paralelních procesů. Základní komponenty p-síti, místa a přechody, popisuji podmínky a operace (abstrakce události) a jejich vzájemné vztahy.
Výskyt operace σj je obecně závislý na splnění určitých podmínek. Tyto podmínky jsou pro přechod tj modelující operaci specifikovány multimnožinou vstupních dat. Splnění podmínek pro výskyt operace σj odpovídá v předchozím odstavci definovanému pojmu - realizovatelnosti přechodu, tj. přechod je realizovatelný, právě když jsou splněny podmínky implikující možnost výskytu operace σj (vstupní místa mají dostatečný počet příznaků). Samotný výskyt operace pak odpovídá realizaci přechodu. Operace, která v systému nastala, může obecně vést ke splněni jiných podmínek, jež implikuji možnost výskytu jiných události. Změna platnosti podmínek systému po provedeni operace a~ je modelována přesunem příznaků ze vstupních míst Ij do výstupních míst Oj po realizaci přechodu tj.
Petriho sítě se používají pro znázornění paralelismu a synchronizace paralelních procesů. Místa se znázorňují kolečky a přechody čárkou kolmou na hrany. Místa a přechody se propojují orientovanými hranami tak, aby žádné dva sousední uzly nebyly téhož typu. Synchronizační úlohy budeme znázorňovat pomoci tzv. označkované Petriho sítě. V této síti je každému místu přiřazen určitý počet značek (znázorněny jako černé tečky). Přechodová pravidla pro označkovanou Petriho sít určují přípustné změny jejího značkováni. Každá změna značkováni se provede tak, že se provede jeden z jejich připravených přechodů. Přechod mezi místy p a q je připraven, jestliže každé misto p obsahuje alespoň tolik teček, kolik vede z místa p do t hran. Přechod se pak provede takto:

1. Z každého místa p se odeber tolik teček, kolik vede hran z místa p do místa t
2. Do každého místa q se přidá tolik teček, kolik vede hran z místa t do místa q.

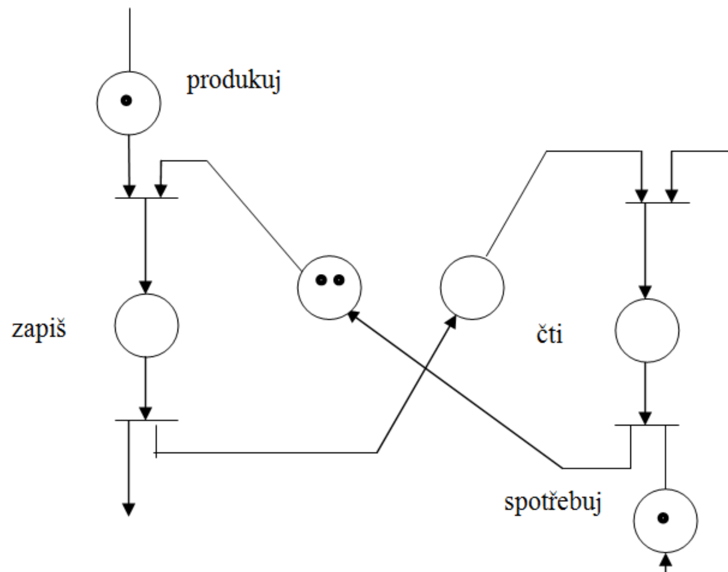
Při modelování paralelních procesů používáme Petriho sítě tak, že každému příkazu bude odpovídat nějaké místo sítě. Tečka pak bude symbolizovat prováděni odpovídajícího příkazu. Přechod z příkazu A na příkaz B bude v Petriho síti znamenat provedení přechodu, který je mezi místy A a B. Běh procesu se proto v Petriho síti jeví jako pohyb tečky. Paralelismus se projeví existencí více teček.
souběh

Souběh – rendezvous

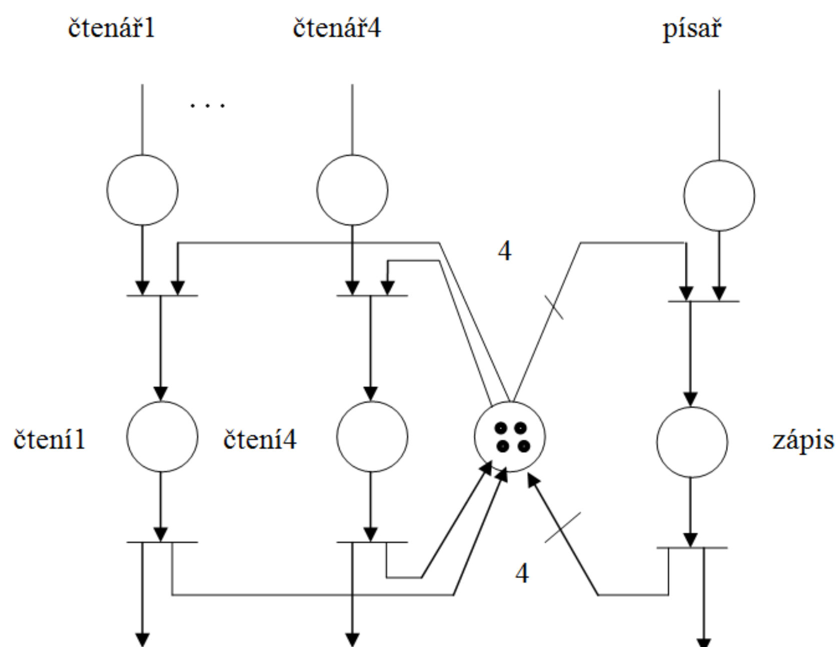


Producent a konzument

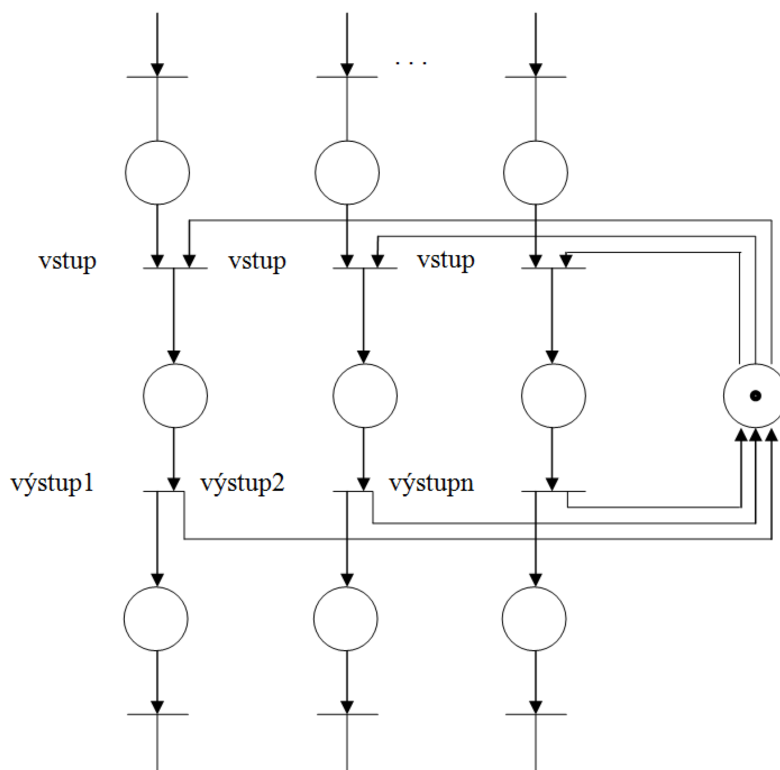
producent konzument



Čtenáři a písaři



Model kritických sekcí



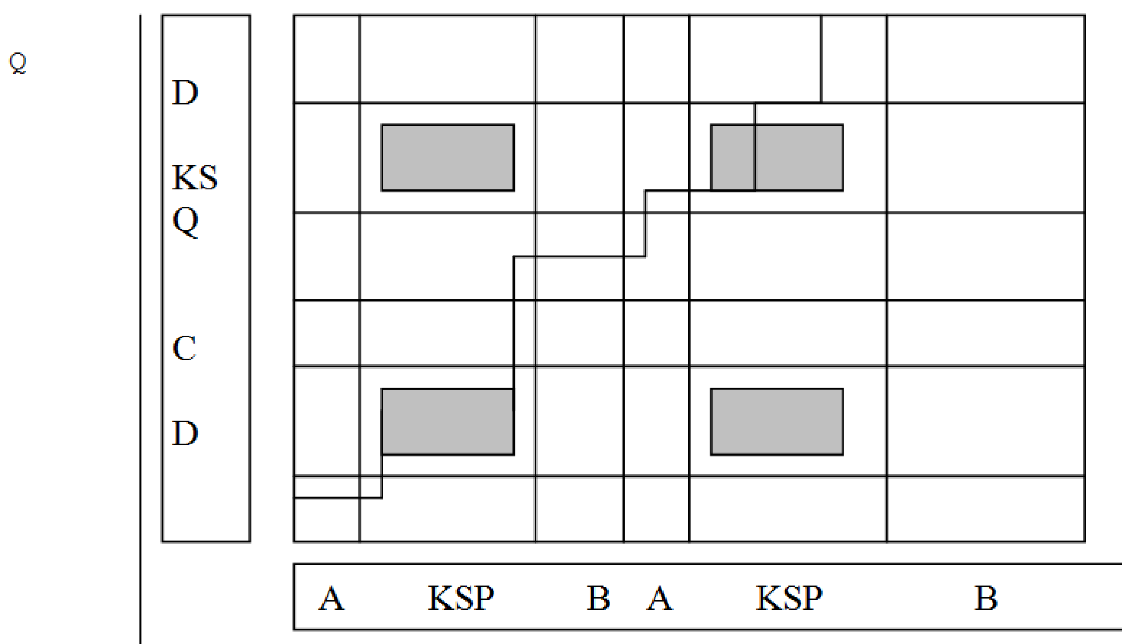
Úkol zní: Jak zabránit časové závislým chybám?
Podstata řešení spočívá v synchronizaci procesů, tj. v koordinaci jejich postupu. Pokud nejsou splněny Bernsteinovy podmínky, je třeba rozvážit jednotlivé případy časové závislosti a ty které jsou nevhodné odstranit synchronizací procesů.
Procesy synchronizujeme tak, že je doplňujeme o synchronizační operace.
Nejjednodušší synchronizační operace jsou ty, kterými koordinujeme postup procesu s tokem fyzikálního času. Příkladem takové operace je operace běžná ve všech RT - systémech, kterou často vyjadřujeme punkcí DELAY (T).
Jde o synchronizaci v čase absolutní, která způsobí pozastavení procesu na T časových jednotek (například milisekund). Použiti této instrukce je však problematické, neboť vyžaduje

znalost toho, na jak dlouho máme řešení procesu pozastavit a vlastně vyžaduje vědomost o tom, jak se chovají ostatní procesy, resp. jaké jsou jejich okamžité nároky na sdílené zdroje. To je požadavek naprosto nereálný, protože ho nelze vlastně vůbec odpovědět v důsledku velmi dynamicky se měnícího prostředí; ať již jde o prostředí operačního systému, nebo technologického procesu.
Častějším, velmi obecným případem je použití takových synchronizačních operací, kterými dosahujeme uspořádání relativní rychlosti postupu procesů. Než přejdeme k dalšímu výkladu, seznámíme se s pojmem kritické oblasti, který v tomto výkladu použijeme.

KRITICKÁ OBLAST,POSTUPOVÁ CESTA
Uvažujme o dvou procesech P a Q. Představme si, že mají následující cyklickou strukturu

Proces P:	Proces Q:
LOOP A: KSP: B: END	LOOP C: KSQ: D: END,

Příkazy A,KSP,B,C,KSQ,D zastupují úseky instrukcí programu a jak naznačeno, cyklicky se opakují. Znázorníme si časový postup procesů P a Q na vodorovné a svislé ose.



Při běhu procesů dochází opakovaně k situaci, že má proces vstoupit do oblasti KSP, resp. KSQ. Tak je označena v obou procesech ta část kódu, která pracuje se společnými proměnnými nebo objekty obou procesů a kde je nebezpečí, že jeden proces naruší chod druhého procesu. Tato část kódu programu se proto nazývá kritická sekce, nebo kritická oblast a ty kritické sekce všech procesů, které operuji na společných objektech se nazývají spražené kritické sekce, nebo spražené kritické oblasti.
Při souběžném chodu procesů dochází k přepínáni kontextu a k střídavému využívání procesoru procesy, v našem případě dvěma. To je na obrázku znázorněno klikatou čarou, která se nazývá postupová cesta a znázorňuje střídáni procesů při využiti procesoru. Je zřejmé, že je žádoucí aby postupová cesta neprocházela přes průnik kritických sekcí procesů, který je na obrázku vyznačen tmavě a který nazýváme zakázaná oblast.
Jak to zařídit je úkolem synchronizace. Podíváme-li se na obrázek, vidíme, že máme v podstatě dvě možnosti.

1. Postupová cesta vede tak, aby neprocházela zakázanou oblastí, což znamená, že musíme v určitém okamžiku omezit možnost přepínáni kontextu.
2. Zakázaná oblast se odsune jinam, aby postupová cesta prošla mimo ni.

Podstatou prvního přístupu je metoda tzv. pasivního čekání, zatímco v druhém případě jde o přistup, jehož podstatou je metoda tav. aktivního čekáni. Je nyní pouze otázkou, jakými prostředky dosáhneme efektu požadovaného v bodě 1) nebo 2). Těmito prostředky jsou synchronizační nástroje.

12.4 SYNCHRONIZAČNÍ NÁSTROJE

Synchronizační nástroje jsou technické nebo programové prostředky, kterými umožňujeme synchronizovaný postup kooperujících procesů v čase. Ukažme si nejdříve obecnou ideu, které stoji v základě jakéhokoliv synchronizačního prostředku.
Představme si, že máme synchronizovat dva procesy P a Q a dále, že máme k dispozici zatím ještě blíže nespecifikovanou spraženou dvojici operaci Z a K s vlastností, že jakmile některý proces provede operaci Z, nemůže jiný proces tuto operaci provést, dokud první proces neprovede operaci K.
Tyto operace můžeme použít k tomu, abychom zabránili současnému vstupu procesů do zakázané oblasti následovně. Dříve, než libovolný proces vstoupí do kritické oblasti, vykoná synchronizační operaci Z. Následkem toho se postup druhého (ostatních)procesů pozastaví. Po výstupu zkritické oblasti provede první proces operaci K, která povolí provedeni operace Z jinému procesu a tím umožní čekajícím procesům vstoupit do jejich kritických oblastí.
Jsou asi zřejmé dvě věci:

1. Předpokladem úspěšné synchronizace je, aby procesy (tj. programátoři !!!) postupovali korektně, ukázněně, aby používali synchronizační operace vždy při vstupu a výstupu do a z kritické oblastí.
2. Implementovat synchronizační operace Z a K nebude asi úplně jednoduché, protože minimálně operace Z, nebo alespoň její část musí být nedělitelná. To proto, aby během jejího provedení nemohl být procesor předělen jinému procesu, který by vstoupil do téže synchronizační operace. Současně je třeba řešit otázku co s procesem, který nemůže provést nebo dokončit operaci Z.

Není proto divu, že existuje více různých synchronizačních nástrojů, které řeší v podstatě tentýž problém. že to dáno různými přístupy k jejich implementaci a vyplývá 2 toho i to, že se v různých situacích mohou jednotlivé nástroje různě osvědčit. Ze shora uvedeného obecného modelu synchronizace však vychází většina universálních synchronizačních nástrojů.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Procesy, paralelní procesy, souběžné zpracování

Ing. Ivo Špička, Ph.D.

Ostrava 2013

© Ing. Ivo Špička, Ph.D.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

13 PROCESY, PARALELNÍ PROCESY, SOUBĚŽNÉ ZPRACOVÁNÍ ... 3

13.1 Procesy, paralelní procesy, soubéžné zpracování ... 4

13.2 Systémy reálného času ... 4

13.3 Paralelni systémy ... 4

13.4 Výpočetní Procesy ... 5

13.4.1 Rozdíl mezi procesem a programem ... 6

13.4.2 Popis výpočetní akce ... 6

13.5 Přepnuti kontextu ... 6

13.6 Problém časové závislosti ... 7

13.7 Vlastnosti OS reálného času ... 8

13.7.1 Deterministické chování ... 8

13.7.2 Rychlá odezva na vnitřní události ... 8

14 POUŽITÁ LITERATURA, KTEROU LZE ČERPAT K DALŠÍMU STUDIU ... 10

13 PROCESY, PARALELNÍ PROCESY, SOUBĚŽNÉ ZPRACOVÁNÍ

Obsah kapitoly:

Definujte multiprocessing, distribuované zpracování

Popište souběžné zpracování, paralelní procesy, systémy reálného času

CÍL:

Po prostudování tohoto odstavce budete umět:

- definovat multiprocessing, distribuované zpracování
- popsat souběžné zpracování, paralelní procesy, systémy reálného času

13.1 PROCESY, PARALELNÍ PROCESY, SOUBĚŽNÉ ZPRACOVÁNÍ

V následujících přednáškách budeme hovořit o problematice zpracováni sobžně pracujících úloh na výpočetním systému. Slovo souběžně je použito jako překlad anglického »concurrent« . Concurrent programming, concurrent processing. Co to znamená ?
Concurrent programming , neboli technika programováni souběžně pracujících úloh je technika tvorby programů, které umějí dělat více vécí »v témže čase <
Jak dosáhnout na počítači současného provádění více úloh současně? V principu ¨dvěma způsoby:

a) na jednom procesoru nebo
b) na více procesorech

Programování takových úloh, které vyžadují souběžné pracující programy souvisí dále se dvěma pojmy:

a) programování reálného času (real time programming)
b) Paralelní prgramování

13.2 SYSTÉMY REÁLNÉHO ČASU

Jsou to takové systémy, kde časové požadavky mají podstatnou váhu a hrají důležitou roli při specifikování funkcí systému. Časové požadavky se mohou projevit:

a) Jako poždavky na rychlost reakce na podněty přicházejecí z okolí počítače (typické pro řídicí systémy)
b) Jako podmínky stanovující časový limit pro zpracování nějakých úloh
c) Jako požadavky na automatické časové startování úloh a to buď cyklicky nebo v zadaných absolutních časech

Příklady:

- každý operační systém,i monoprogramový, obsahuje funkce, které mají RT-charakter: například IO subsystém.
- systémy zpracování transakcí
- systémy řízení průmyslových technologií
- vestavné systémy (embeded systems) . Jsou to systémy, které se zabudovávají do nejrůznéjšíchh technických výrobků a » prudce vylepšují jejich inteligenci « , jako jsou auta, pračky, CD disky atd.
- komunikačni systémy: telefonní ústředny, satelitní komunikace

13.3 PARALELNI SYSTÉMY

Jsou to systémy implementované na počítačích, resp. výpočetních systémech, se dvěma nebo více současně pracujícími procesory. Ale pozor: podmínkou pro to, abychom mohli považovat aplikaci za paralelní je, aby řešení nějaké logicky ucelené úlohy bylo distribuováno alespoň mezi dva procesory.
Řekneme-li tedy, že aplikace je paralelní, říkáme cosi o implementaci, řekneme-li, že jde o RT-aplikaci, říkáme něco o jejich vlastnostech požadovaných z hlediska časového zpracování. Přesto je zřejmé, že mezi paralelními a RT-systémy existuje souvislost daná minimálně tím, že bychom asi netvořili paralelní systémy, pokud bychom se nedostávali do časových potíží. Avšak i z metodického hlediska, jak pokud jde o analýzu úloh tak jejich implementaci mají oba přístupy mnoho společného.
Společným podstatným rysem obou typů systémů je jejich požadavek na spolehlivost a to jak v oblasti HW, tak v oblasti SW. Spolehlivost HW je dnes vysoká, daná jednak vysokou úrovní konstrukčních technologií, jednak dalšími možnostmi zabudovávání kontrolních obvodů a mechanismů na úrovni HW. V SW oblasti je situace o mnoho složitější. Neustále do

ní totiž vstupuje člověk se svými novými programy, které jsou plné jeho chyb: ať již jde o chyby ve specifikaci, její implementací atp. Na paralelních systémech pak je obtížným problémem ladění, protože je vlastně nemožné udržet kontrolu nad stavem všech paralelně, na různých procesorech nestejným tempem probíhajících úloh.
Návrh souběžné pracujících programů.
Jednoprocesorové, tj. normální počítače, pracují sekvenčně a většina programovacích jazyků dovoluje formulovat výpočetní postupy jako sekvenčně prováděné posloupnosti operací.
Souběžně pracující programy mají části, kterým se říká PROCESY, a které mohou pracovat souběžně. Avšak procesy samy jsou sekvenční. Souběžné provedeni procesů pak může být skutečně paralelní, nebo pouze quasiparalelní.
PROCES je část programu (nebo systému ,pak to může být i program), která je prováděna přísně sekvenčně (instrukce zapsaná v textu programu dále od začátku se provede v čase později) ale která může být provedena souběžně paralelně (reálně nebo virtuálně=zdánlivě) s ostatními procesy programu.
Rozděleni počítačových architektur z hlediska paralelismu.
Monoprocesorové architektury - dovolují pouze quasiparalelismus, paralelně na nich provozované systémy jsou označovány jako systémy pracující ve sdílení času (time sharing) nebo multiprogramové systémy, vlastní procesy pak slovem PROCES, nebo TASK nebo THREAD.
Víceprocesorové architektury

- MULTIPROCESSING provádí souběžně pracující procesy na více procesorech, které mohou přistupovat ke společně sdílené paměti a samy mohou mít vlastní lokální paměť.
- DISTRIBUOVANÉ ZPRACOVÁNÍ (distributed processing) zde jsou procesy prováděny na oddělených procesorech bez možnosti přístupu ke společně sdílené paměti. Paměť, pouze lokální, mají jednotlivé procesory a výměnu dat je nutno provádět pomocí zasílání zpráv po komunikačních kanálech.Typickým představitelem jsou zde systémy založené na transputerech.

Každodenní zkušenost nám dokazuje, že svět, který nás obklopuje je světem paralelních dějů. Dějů, které probíhají v čase současně a vzájemně se ovlivňují. Proto při řešení úloh modelujících děje probíhající v reálném čase hraje podstatnou úlohu paralelní zpracování (paralelismus).
Jako systémy pracující v reálném čase označujeme počítačové aplikace, které lze rozdělit zhruba do dvou skupin:

a) Nepřetržitě pracující výpočetní, informační nebo výrobní počítačové systémy, v nichž ke vstupu, zpracování a výstupu dat dochází v čase náhodně, částečně nepředvídaně a z více různých míst nebo zdrojů (terminály, jiné počítače atp.).
b) Systémy řízení technologických pochodů, které vyhovují předchozí charakteristice a navíc mají daleko přísnější požadavky na rychlost zpracování jednotlivých požadavků

Zatímco v běžných počítačových úlohách a výpočtech vystačíme s jediným programem, výpočetním procesem, v těchto systémech je zapotřebí ke splnění požadovaných funkcí ne jednoho programu, ale více programů, které musí pracovat souběžně. Přitom které programy a kdy budou pracovat souběžně není předem známo a závisí to na tom, jak se situace vyvíjí v "okolí" výpočetního systému, tj. například v technologii, která je systémem řízena.

13.4 VÝPOČETNÍ PROCESY

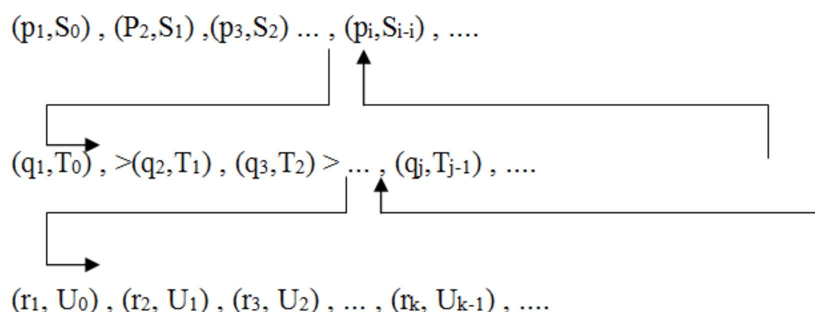
Používáme-li jednoprocesorový počítač, pak se určitá sekvence příkazů programů v nějakém programovacím jazyce (např. Pascalu, atp.) provede sekvenčně, jako posloupnost výpočetních akcí, operaci. Takové posloupnosti výpočetních akcí říkáme sekvenční výpočetní proces, krátce proces.

13.4.1 Rozdíl mezi procesem a programem

Technický objekt, část počítače, která výpočetní akce provádí se nazývá procesor. Každá výpočetní akce je určena příkazem a stavem objektů (proměnných), které v programu vystupují. Výpočetní akci je možno pak formálně popsat uspořádanou dvojici (p, S), kde p je příkaz a S stav objektů před provedením příkazu.

13.4.2 Popis výpočetní akce

(p, S)
p - Příkaz (instrukce programu)
S - Stav datových objektů před provedením
Provádění programu, neboli proces, lze pak chápat jako posloupnost uspořádaných dvojic
(P1, So), (P2, Si), ..., (pi, Si-1), ...
Tato posloupnost může být teoreticky nekonečná. Proces má dynamický charakter. Jednotlivé prvky posloupnosti se sekvenčně generují, stav Si vznikne provedením akce (pi, Si-1) . Provádí-li se některá akce procesu, říkáme, že proces běží. Naproti tomu program, ze kterého vybírá procesor příkazy určující jednotlivé akce se nemění. I když je možno si představit, že pi jsou příkazy vyššího programovacího jazyka, například Pascalu, bude pro pochopení dalšího výkladu lepší představovat si, že pi jsou instrukce na úrovni strojového kódu.
Dnes převážně používané počítače - až na výzkumné případy (např. stroje s objektovou architekturou) a speciální multiprocesorové stroje - jsou jednoprocesorové počítače von Neumannovy koncepce a tedy v principu prostředky sekvenční na úrovni architektury strojového kódu. Říkáme také, že tyto počítače mají vN architekturu.
Je proto na místě otázka, co pak rozumíme paralelismem provedení na sekvenčním počítači vN architektury. Již jsme naznačili, že to bude paralelismus zdánlivý, virtuální, a že ho docílíme rychlými přechody od řešení jedné úlohy (programu) k druhé, jak si hned ukážeme dále.
Nechť je program charakterizován dvojicí < P, S > kde P je konečná množina všech příkazů programu a S množina datových objektů, které se mohou měnit prováděním programu.
Mezi těmito-datovými objekty má význačné postavení určitá podmnožina a tvořená obsahy všech datových registrů procesoru, a která se mění provedením každé strojové instrukce. Tato množina údajů a se nazývá kontext programu a je tedy vlastně $S = \sigma + s$.
Souběžné, konkurenční, provádění více programů <P,S> , < Q, T> , <R, U> jakožto para1e1ních procesů, zpracovávaných jedním skutečným procesorem, pak může například vypadat tak, jak je znázorněno na dalším schématu.



13.5 PŘEPNUTI KONTEXTU

Vidíme že posloupnost provedení příkazů reálným procesorem je vlastně následující:
(p1, S0) , (p2, S1) , (q1, T0) , (r1, U0) , (r2, U1) , (r3,U2) , (q2,T2) , (q3,T2) , (p3,S2) , ...
Je přirozené, že při přepínání úloh je nutno uschovat kontext programu a při opětovném přechodu k řešení úlohy zajistit jeho obnovu tak, aby mohl program navázat přesně na stav v

němž bylo jeho prováděni přerušeno. Mechanizmu přechodu od řešení jedné úlohy k druhé se proto také říká přepínání kontextu programu.
Prakticky je přepnutí kontextu umožněno mechanismem hardwarového přerušení. Přerušovací systém počítače umožňuje přerušit prováděni běžícího programu na základě tzv. vnitřního přerušeni odvozeného od hodinových pulsů generovaných hardwarem počítače anebo na základě vnějších přerušeni generovaných souběžně pracujícími periferními zařízeními. Tato periferní zařízeni vydávají přerušovací signál pří dokončeni operací, které byly vyžádány procesorem. .
Tak může na jednom počítači probíhat souběžně několik paralelních procesů, které používají tytéž technické prostředky (periferní zařízení, operační paměť atp.)
A zde vzniká jeden z obecných problémů paralelismu: Korektní použití společného prostředku několika paralelními procesy. To znamená, že při použití společných prostředků více procesy nesmi dojít k tomu, aby vlivem přepínání kontextu mezi jednotlivými procesy došlo k nesprávné činnosti jednoho nebo více z nich. Je zřejmé, že jde o problém času, časové závislosti procesů, který by nevznikl kdyby mohl být proces proveden samostatně.

13.6 PROBLÉM ČASOVÉ ZÁVISLOSTI

Při zpracováni paralelních procesů se mohou vyskytnout dvě základní situace. Buď jsou tyto procesy (alespoň dva) vzájemně úplně nezávislé v tom smyslu, že nijakým způsobem nespolupracují, tj. nepodílejí se o společná data atp. Anebo je tomu naopak: existuje nějaký společný objekt, jako např. periferní zařízení, datová struktura, soubor atp., jehož společným používánim se mohou akce různých procesů vzájemně ovlivňovat.
Intuitivně je zřejmé, že časově závislé chyby jsou důsledkem nevhodného používání společných objektů paralelních procesů. Odstraněni těchto chyb proto musí spočívat v koordinovaném "promyšleném" používání společných objektů, což v konečném důsledku znamená přijmout určitá omezení v přístupu k nim.
Tato omezení je možno vyjádřit např. ve formě Bernsteinových podmínek. Formulujme Bernsteinovy podmínky pro dva procesy řízené příkazy P a Q, jejichž jedinými společnými objekty jsou nějaké proměnné. Bernsteinovy podmínky požaduji, aby procesy byly deterministické, tj. vzájemně na sobě nezávislé.

1 .	R(P)	&	W(Q)	-	O
2.	R(Q)	&	W(P)	=	O
3.	W(P)	&	W(Q)	-	O

Zde je:

R(S) množina proměnných	užitých v příkazech S	v operaci čtení z paměti
W(S) množina proměnných	užitých v příkazech S	v operaci zápisu do paměti

O prázdná množina
Vysloveno explicitně: Bernsteinovy podmínky požadují, aby množina společných proměnných užitých v příkazech čteni nebo zápisu jednoho procesu byla disjunktní s množinou proměnných užitých v příkazech zápisu druhého procesu, neboli aby procesy spolu nekomunikovaly prostřednictvím poměných k nímž má alespoň jeden z nich právo zápisu.
Tyto podmínky jsou tedy velmi silné, omezující a pro nás prakticky nezajímavé. Nás totiž především řešení právě situaci explicitního paralelismu, což je situace běžná při programování řídicích systémů a zde se bez přístupu různých procesů ke sopečných proměnných neobejdeme: právě naopak na něm je většinou založena idea dekompozice systému řízeni.
Bernsteinovy podmínky stačí k tomu, aby procesy byly deterministické. To znamená, že posloupnost akcí kterékoliv z procesů je jednoznačné určená počátečním stavem objektů So a příslušným programem. Takové procesy jsou pak defacto nezávislé, alespoň pokud jde o možnost vzájemného nežádoucího ovlivnění dat, které zpracovávají.

U paralelních procesů, které jsou nedeterministické, naopak dochází k jejich ovlivňováni s postupem času a tato vlastnost se nazývá časová závislost procesů.
Vlivem časové závislosti procesů pak může docházet k chybám, např. tím, že dojde k současnému použití společných proměnných oběma procesy.

13.7 VLASTNOSTI OS REÁLNÉHO ČASU

Nejdříve musíme přesněji specifikovat, co máme na mysli, hovoříme-li o operačním systému reálného času, resp. o vlastnostech nutných pro provozování RT-aplikací.
Budeme říkat, že operační systém má vlastnosti operačního systému reálného času, jestliže splňuje následující tři požadavky:

- deterministické chováni
- rychlá reakce na externí události
- rychlé reakce na vnitřní události

13.7.1 Deterministické chování

Jednou z nejdůležitějších charakteristik OS reálného času je zaručená krátká doba odezvy. O deterministickém chování pak hovoříme tehdy, jestliže doba odezvy je zaručena i při maximální zátěží systému zapříčiněné požadavky aplikačních úloh.
Rychlá odezva na externí události
Aby mohl systém reagovat na vnější událostí v nejkratším možném čase, musí být zajištěno efektivní řízeni procesů pod OS. Toho lze dosáhnout:

- přiřazením vhodného,resp. potřebného, programu vnější události (vnějšímu přerušeni)
- aktivaci tohoto programu v co nejkratším možném čase
- uložením procesu (odpovídajícího programu) trvale v paměti, jako programu rezidentního v paměti.

Toho lze dosáhnout pouze při použití prioritního systému plánování procesů využívající strategii tzv. preemptivního plánování.

13.7.2 Rychlá odezva na vnitřní události

Rychlá aktivace procesu není dostatečnou podmínkou pro rychlé zpracování RT úlohy. K tomu je třeba:

- účinné komunikační a synchronizační nástroje (funkce) pomocí nichž mohou být různé procesy rychle synchronizovány
- účinné prostředky pro rychlou výměnu dat mezi procesy

Standardní OS UNIX System V tyto požadavky nesplňuje anebo alespoň ne v takové míře, aby mohl být použit bez omezeni pro nejrůznější RT-aplikace.
Operační systém SORIX, který vyvinula fa Siemens AG obsahuje rozšířeni pro oblast reálného času nad rámec standardu UNIX umožňující dosaženi deterministického chování a tím pro aplikaci řízení v reálném čase.
Přehled RT-funkcí OS SORIX
RT - vlastnosti OS SORIX jsou založeny především na přerušitelných rutinách (funkcí) jádra (operačního systému) a na účinném řízení procesů, v dalším sledu pak na silných funkcích pro komunikaci mezi procesy, časovacích funkcích a na systému pro obsluhu souborů, který je optimalizován s ohledem na použiti v RT - prostředí.

- Přerušitelné rutiny jádra
  - proces může být přerušen i během volání resp. využívání systémových funkcí
- Procesy rezidentní v paměti
  - umožňuje se, aby časově kritické procesy byly trvale uloženy v operační paměti i v době,kdy nejsou aktivní. To umožní jejich rychlou aktivaci.

- Řízení procesů technikou preemtivního plánování
    - umožňuje aplikovat techniku aktivního procesu čistě na základě jeho priority s betzprostředním přerušením procesu s nižší prioritou aniž tento vyčerpal tzv. časovou dávku (time-slice).
- Mechanismus událostí s vícenásobným čekáním
    - umožňuje účinnou komunikaci mezi procesy s možností čekání na libovolnou kombinaci asynchronních událostí (multiple waiting).
- Rychlý semafor (quick semaphore)
    - pro rychlou syncronizaci nezávislých procesů.
- Časové funkce
    - umožňují plánování procesů (akci) v piředem zvolených absolutních časových okamžicích nebo cyklicky zvoleným časovým intervalem
- Optimalizovaný systém obsluhy souborů
    - s funkcemi minimalizujícími přístupové časy vstupně/výstupních operaci a s možností přímého vstupu/výstupu by passing rychlé vyrovnávací paměti (cache bugger). Kromě toho obsahuje OS SORIX další funkce, které sice nemají bezprostřední vztah k RT-funkcím,ale jsou užitečné pro automatizaci úloh.

14 POUŽITÁ LITERATURA, KTEROU LZE ČERPAT K DALŠÍMU STUDIU

[1] Špička, Ivo, Programování řídicích systémů. Studijní opora k předmětu. 2013.
[2] Pavel Herout, Učebnice jazyka C. 1. díl , 5. vyd. České Budějovice : Kopp, 2008 - 271, viii s. ISBN 978-80-7232-351-7 (brož.)
[3] Herout, Pavel, Učebnice jazyka C. 2. díl, 3. vyd. České Budějovice : Kopp, 2007 - vii, s. 272-437 ISBN 978-80-7232-329-6 (brož.)
[4] Qing Li, Caroline Yao, Real-Time Concepts for Embedded Systems, CRC Press; 2003, ISBN 1578201241
[5] Doporučená literatura:
[6] WIRTH, N.: Algoritmy a štruktúry údajov, Alfa Bratislava, 1988.
[7] JOSEPH, M.. Real-Time Systems Specification, Verification and Analysis. London : Prentice Hall, 1996. ISBN 0-13-455297-0.