INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Úvod

Ing. Ivo Špička, Ph.D.

Ostrava 2013

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

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.

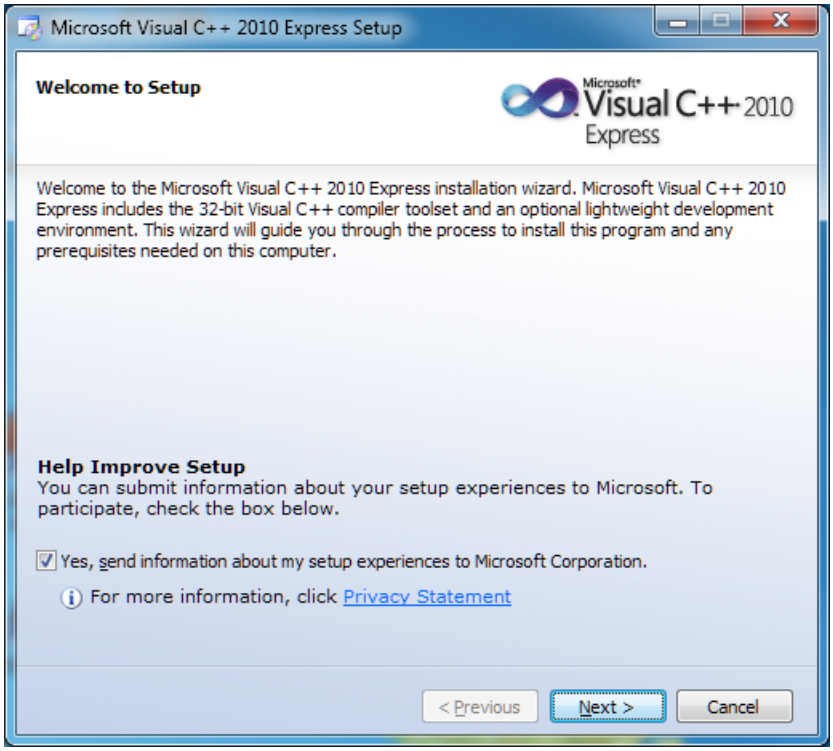
1.6.1 Řešené úlohy

```
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
```

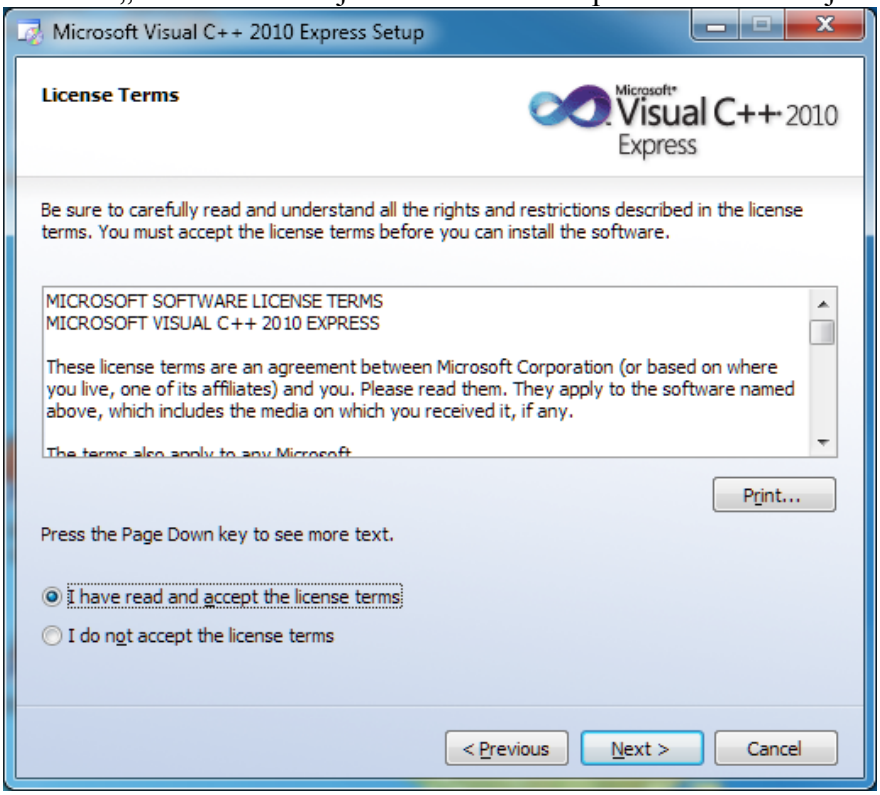
Nyní si rozebereme každý řádek zvlášť. Druhý řádek uvozený znakem # obsahuje direktivu. Direktiva #include říká překladači, že na toto místo má vložit soubor stdio.h. Soubory s příponou .h jsou tzv. hlavičkové soubory. Obsahují deklarace funkcí a datových typů a tak se na ně lze se na ně dívat jako na 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

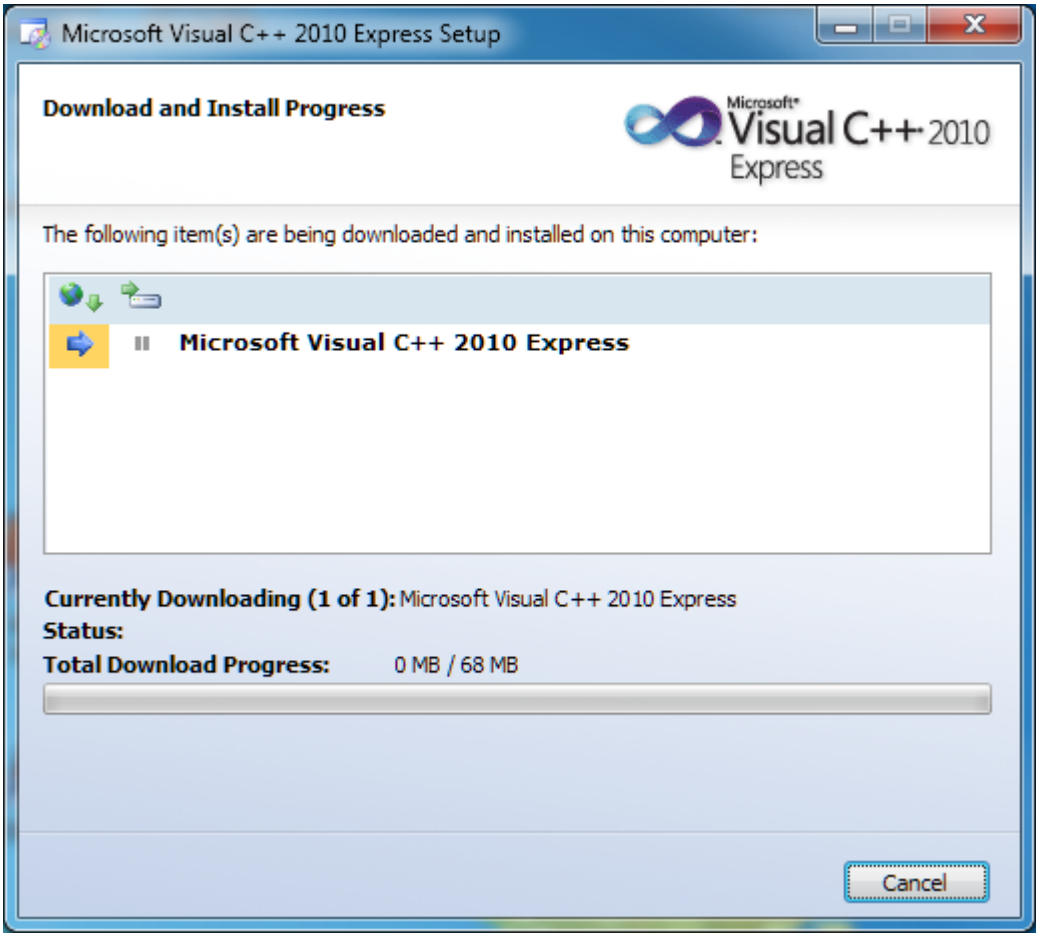
Pokud na počítači nemáme instalováno prostředí Microsoft Visual C++ Express, pak si jej nejprve musíme stáhnout z webu Microsoftu, například pomocí odkazu
http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express
Po spuštění nás povede dále průvodce instalací.



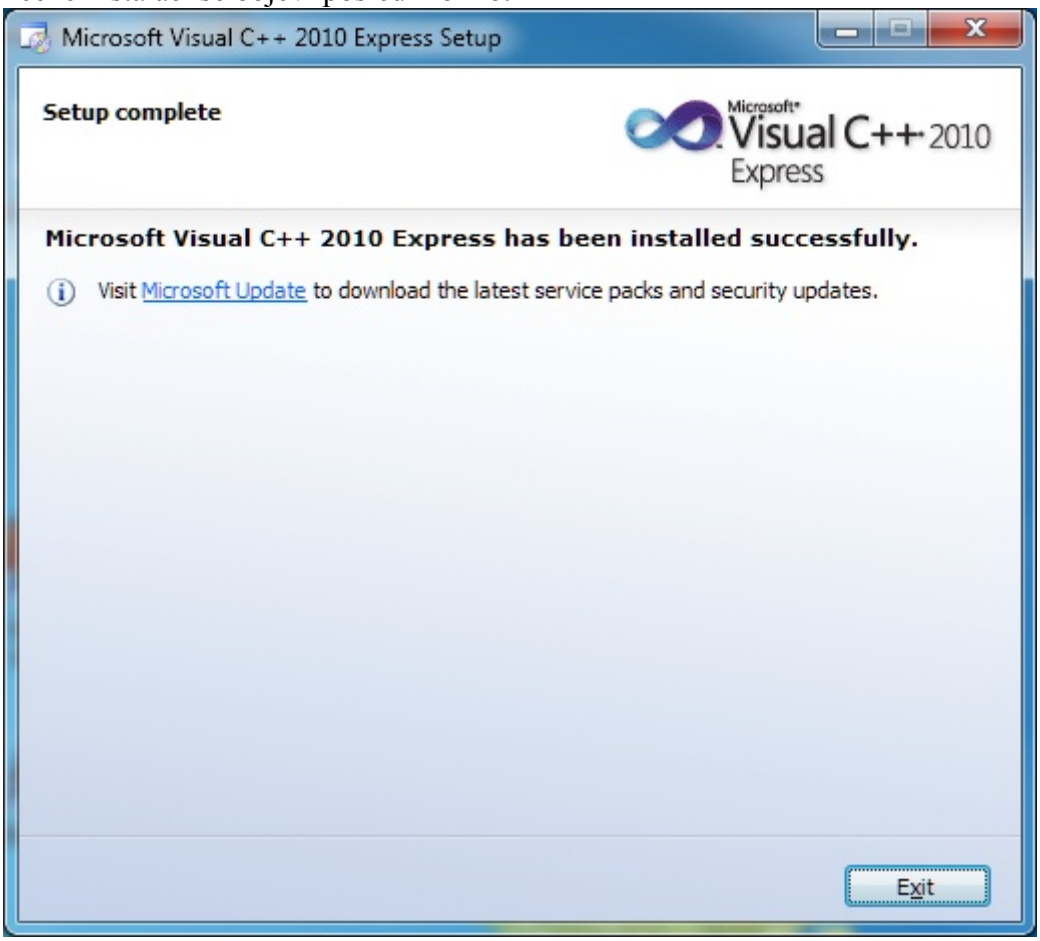
Po zmáčknutí tlačítka „Next“ v následujícím okně musíme potvrdit licenční ujednání



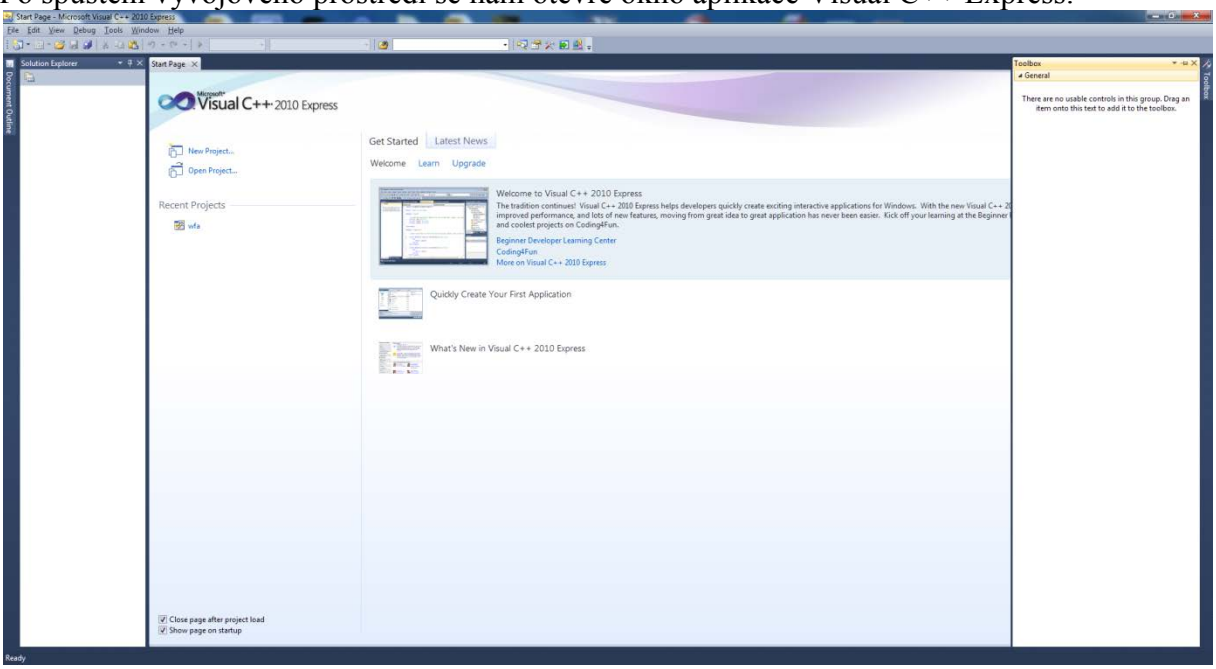
Dvakrát opět stiskneme tlačítko Next (cílový adresář nelze měnit) a spustí se instalace.



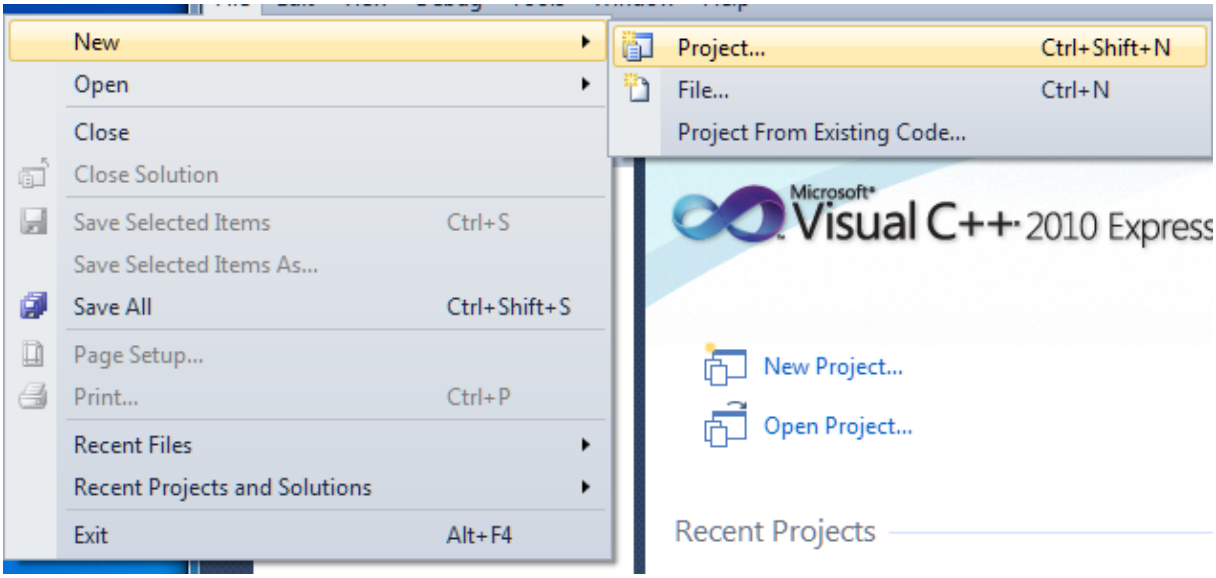
Po ukončené instalací se objeví poslední okno.



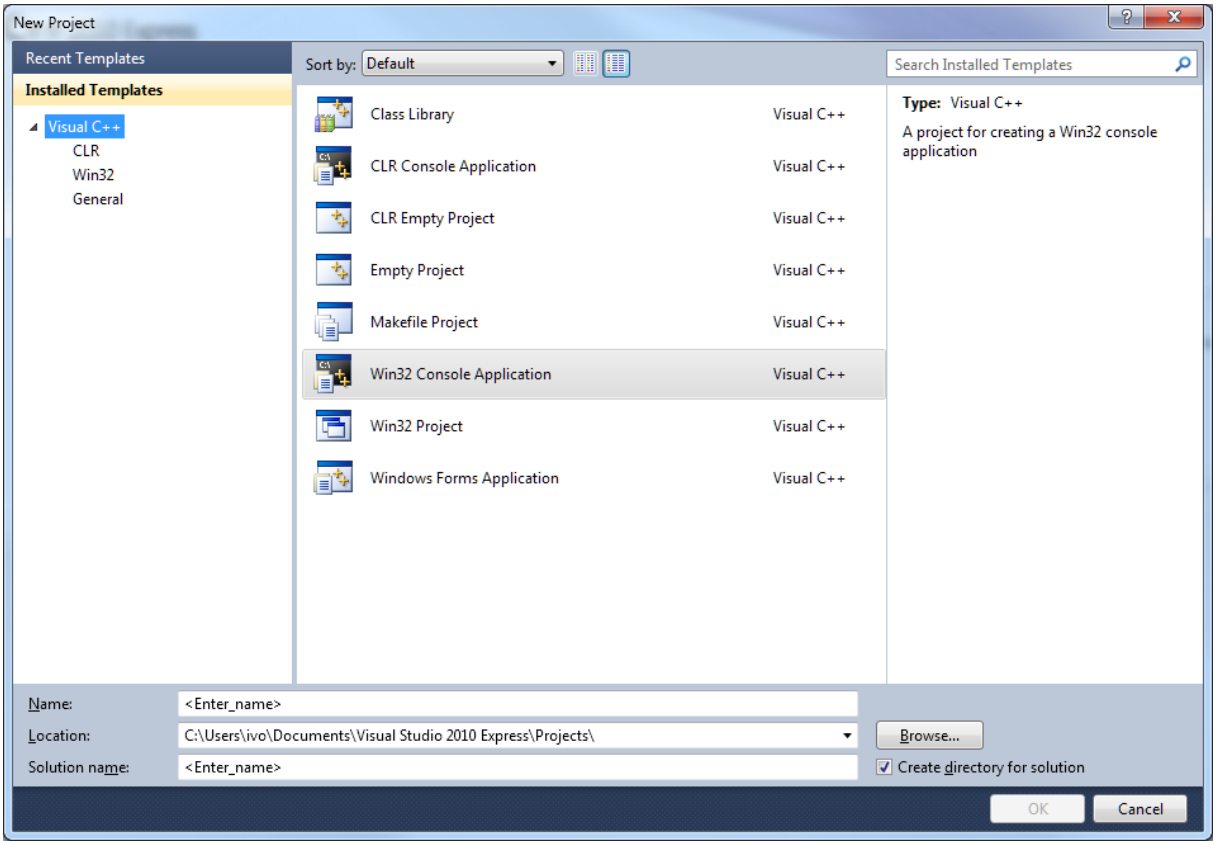
Spuštění vývojového prostředí.
Po spuštění vývojového prostředí se nám otevře okno aplikace Visual C++ Express.



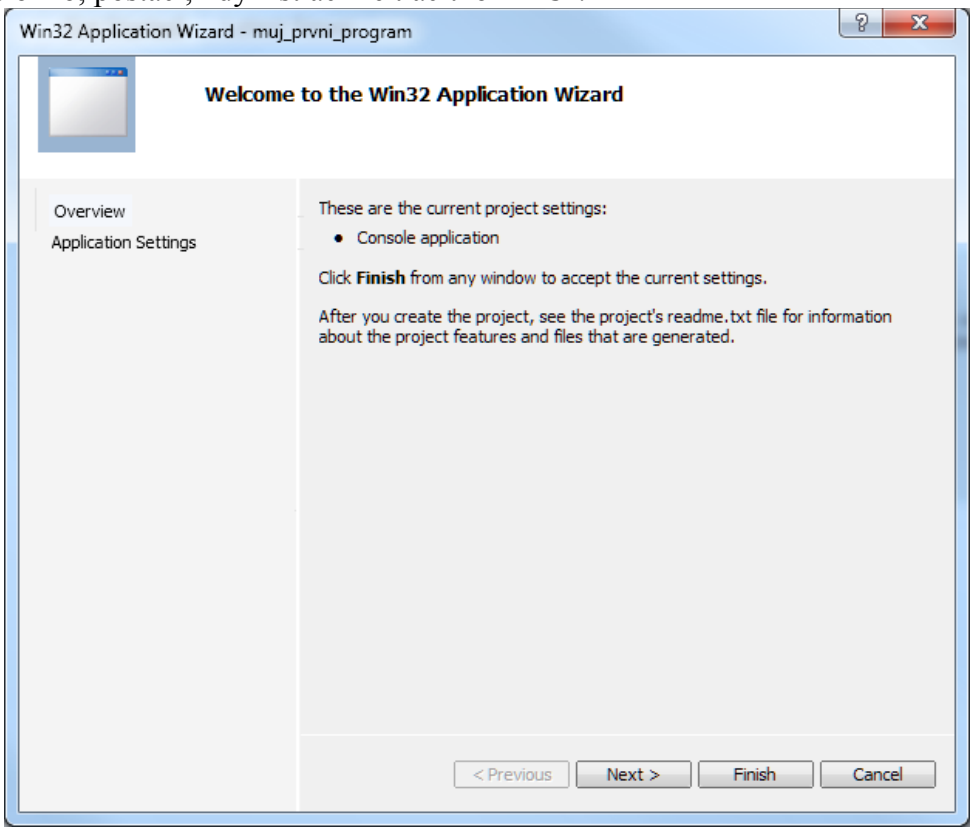
Nejprve se musí vytvořit nový projekt. To lze učinit několika způsoby. Pomocí příkazu NEW z menu



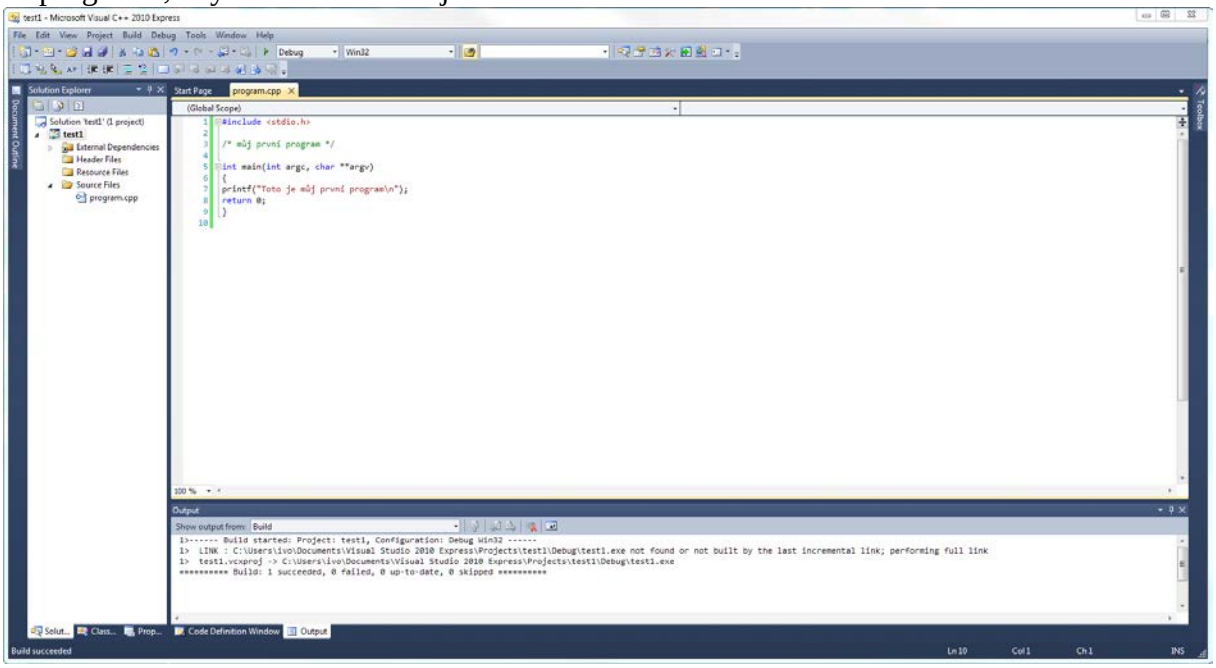
Otevře se nové dialogové okno, ve kterém musíme vybrat šablonu našeho projektu a jeho jméno.



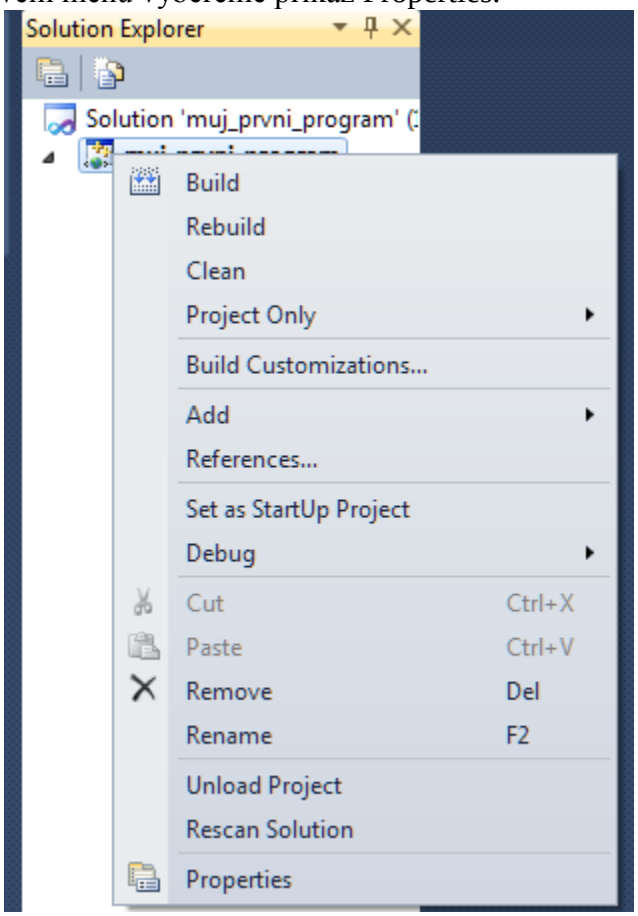
V dalším budeme v tomto kurzu používat konzolovou aplikaci. Do políčka Name napíšeme jméno našeho projektu, například muj_prvni_program. Pokud nechceme měnit standardní cestu ke složce s projekty, necháme nastavenou cestu, jak nám ji nabízí program. Nová cesta by se zadávala do pole Location. Po ukončeném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



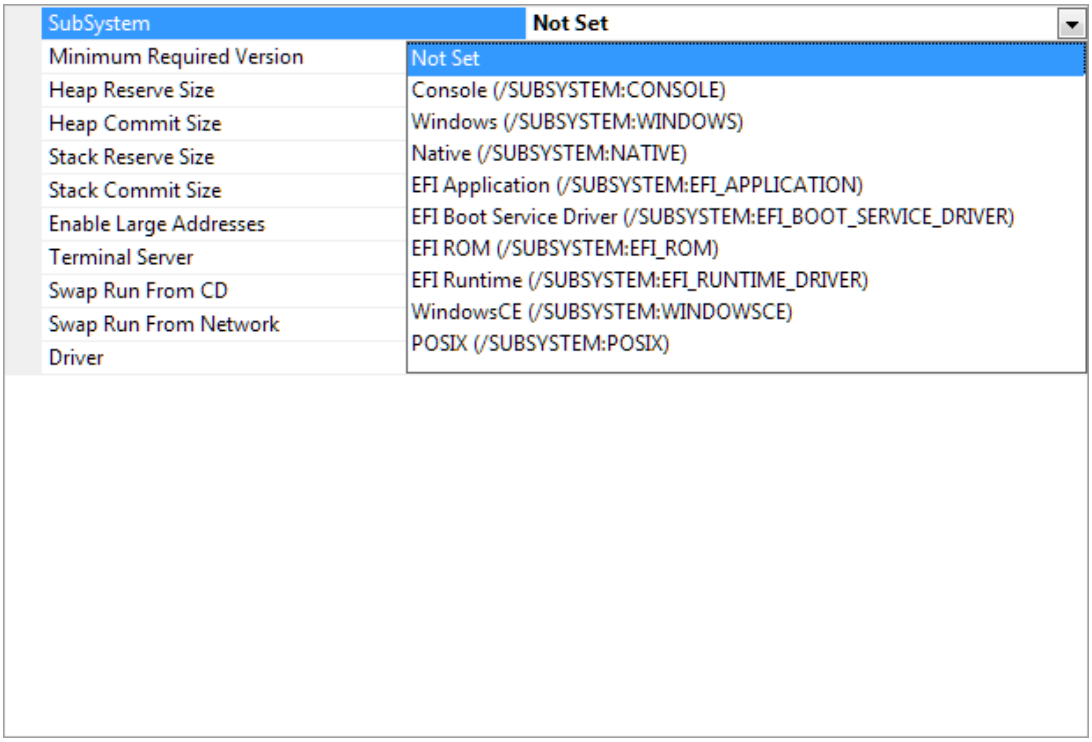
Abychom viděli, co vlastně náš program udělal, potřebujeme ještě nastavit některé vlastnosti našeho projektu. Jinak by nám problikne okno našeho programu, a ještě než bychom stačili cokoli přečíst, zase by se zavřelo. Existují samozřejmě možnosti dopsat vhodné příkazy přímo do programu, abychom dosáhli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



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

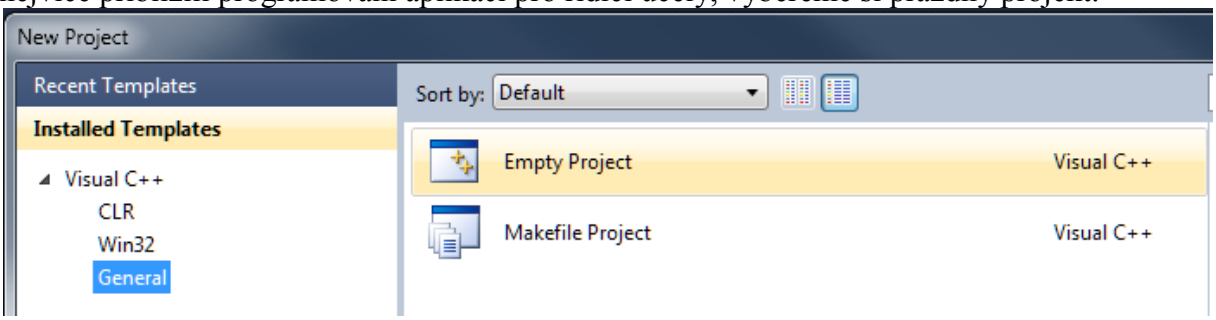


V nově otevřeném okně v prvé části rozbalíme položku Configuration Properties a pak položku Linker. Klikneme na řádek Systém a v prvém řádku v pravé části okna klikneme do řádku vpravo od názvu SubSystem.

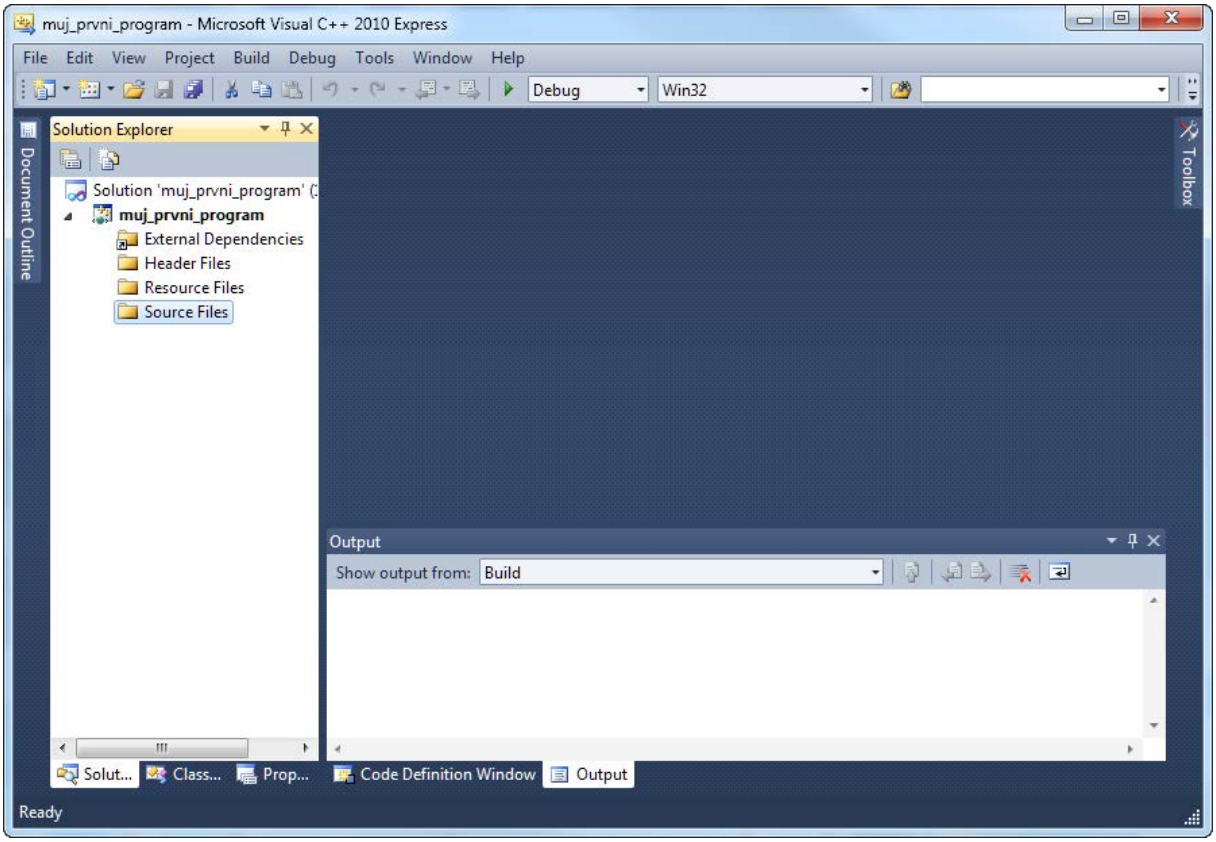


Z rozbalovacího menu vybereme hned druhou položku Console (/SUBSYSTEM:CONSOLE). Volbu musíme potvrdit tlačítkem OK.

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

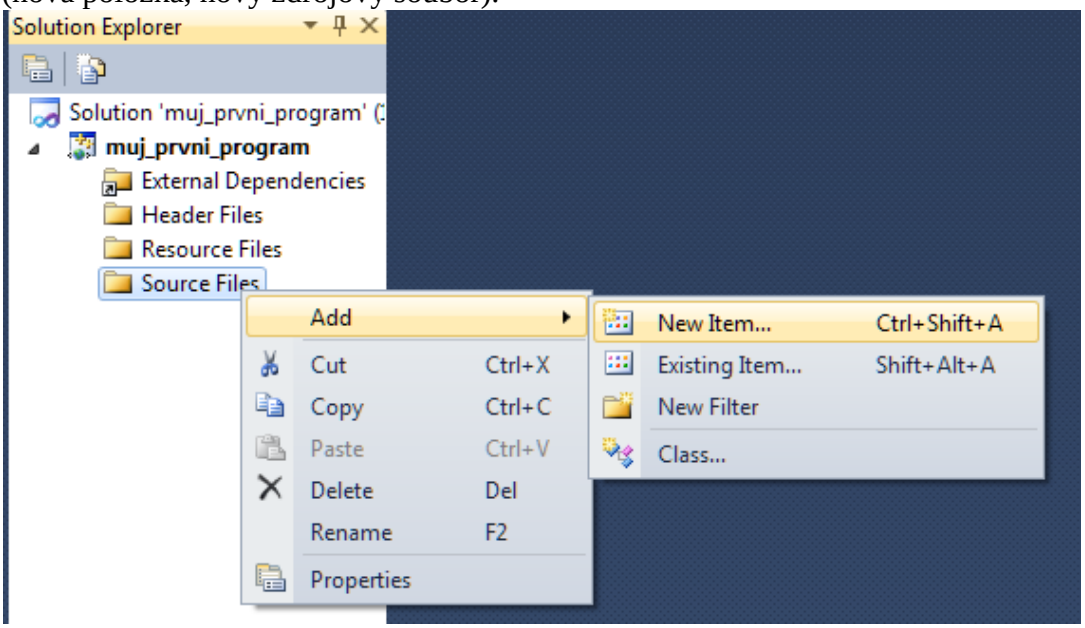


Strukturu prázdného projektu, kterou obdržíme po zadání jména projektu, ukazuje obrázek.

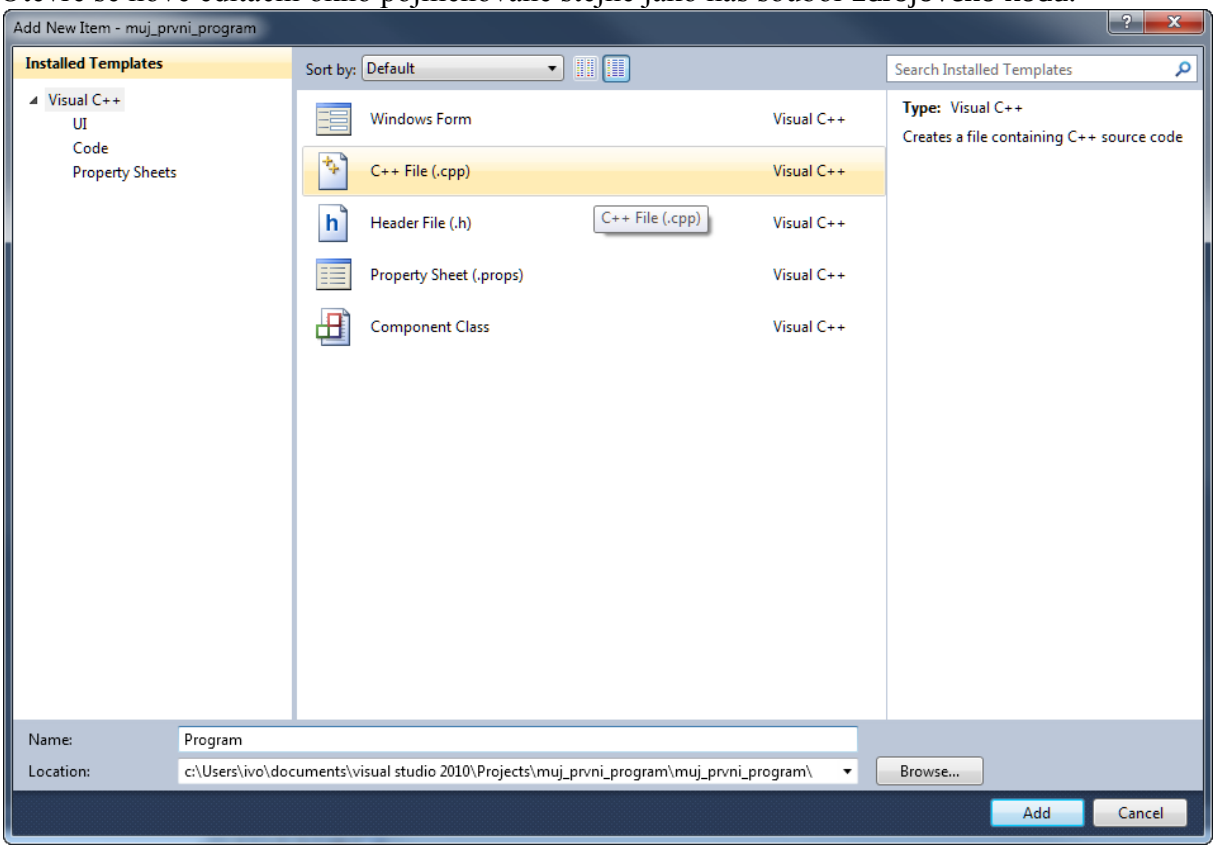


V tomto okamžiku máme zvolenou záložku Solution Explorer, okno výstupu Output. Projekt je tvořen zatím zcela prázdnými složkami External Dependencies, Header Files, Resourced Files a Source Files.

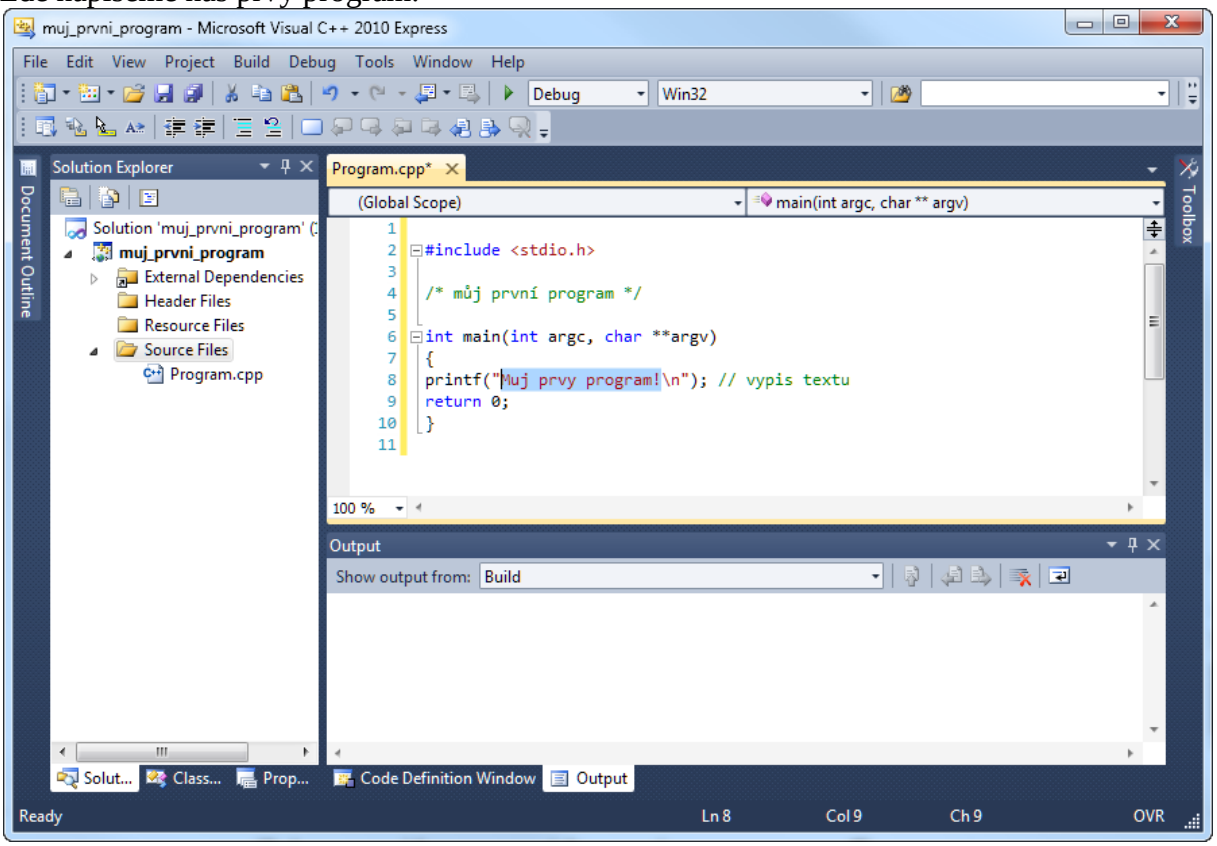
Jak si řekneme dále, každý program v jazyce C má alespoň jeden zdrojový soubor. Přesto, že my budeme psát programy v jazyce C, vývojové prostředí je určeno pro psaní programů v objektovém rozšíření jazyka C, tedy pro jazyk C++, tak přípona našich souborů zdrojových textů bude .CPP místo C. Nejprve tedy 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.

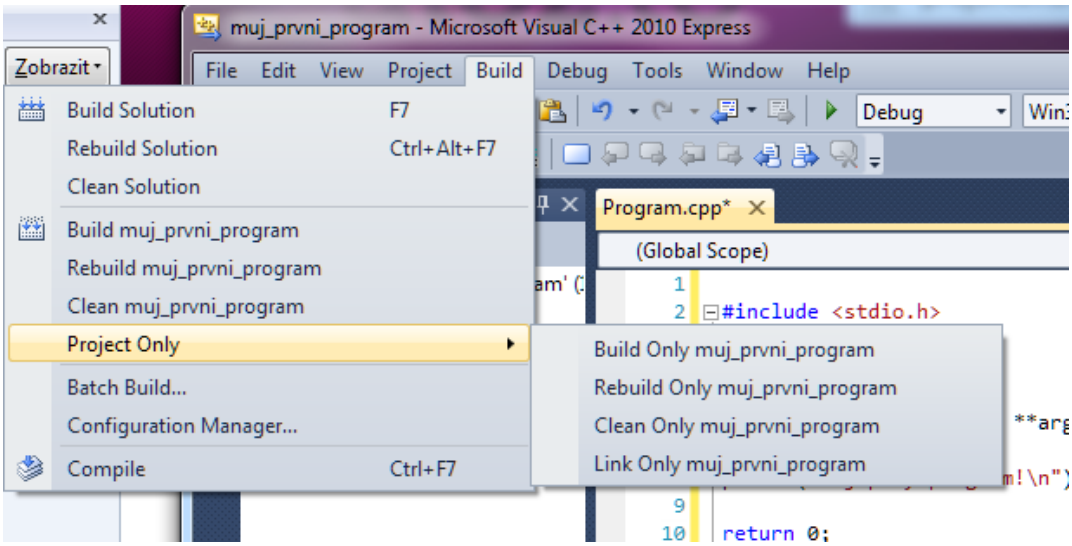


Zde napíšeme náš prvý program.



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.

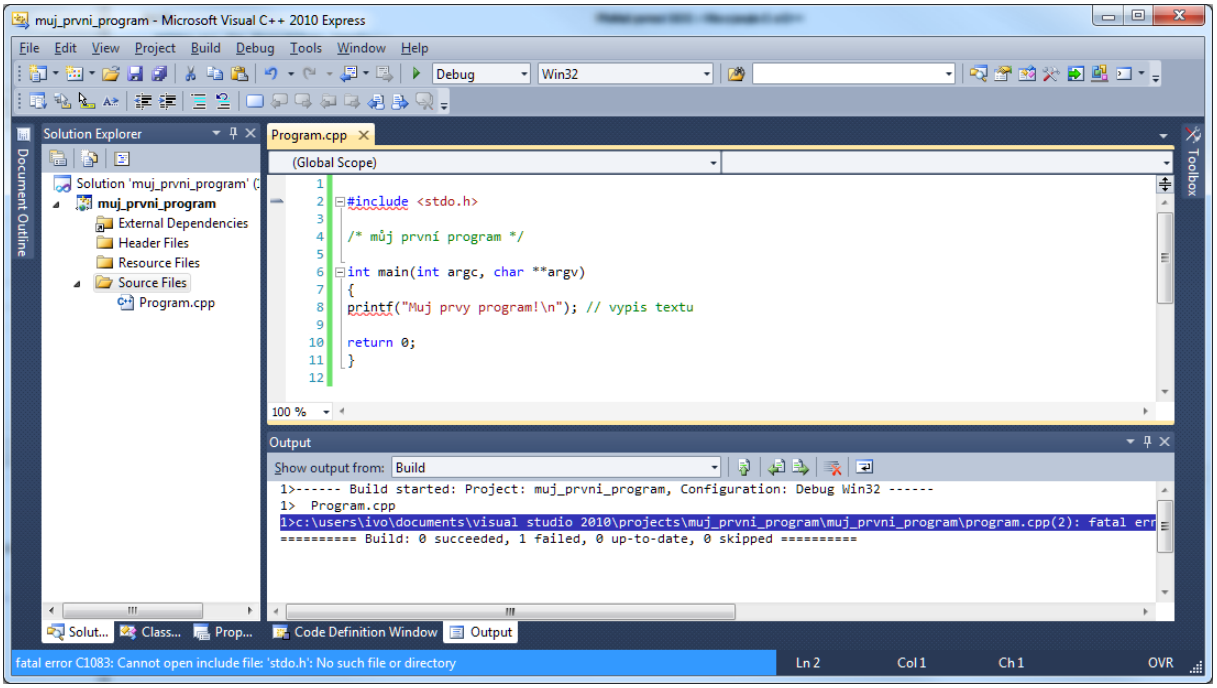
1.8 PŘEKLAD PROGRAMU, SESTAVENÍ PROGRAMU, SPUSTITELNÝ PROGRAM



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:



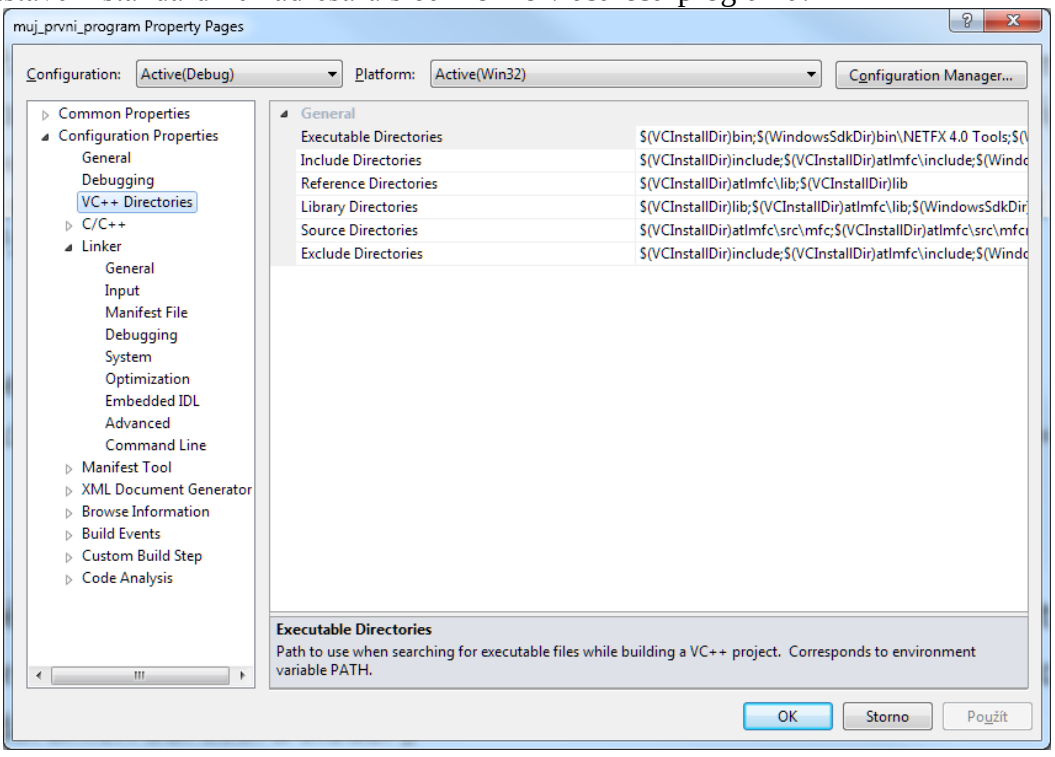
Poklepeme na řádku výpisu prvé chyby v okně output, v editačním okně se nám modrou značkou označí řádek s touto chybou. Zde se jedná o řádek č. 2. Výpis chyby v tomto případě zní:

„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“

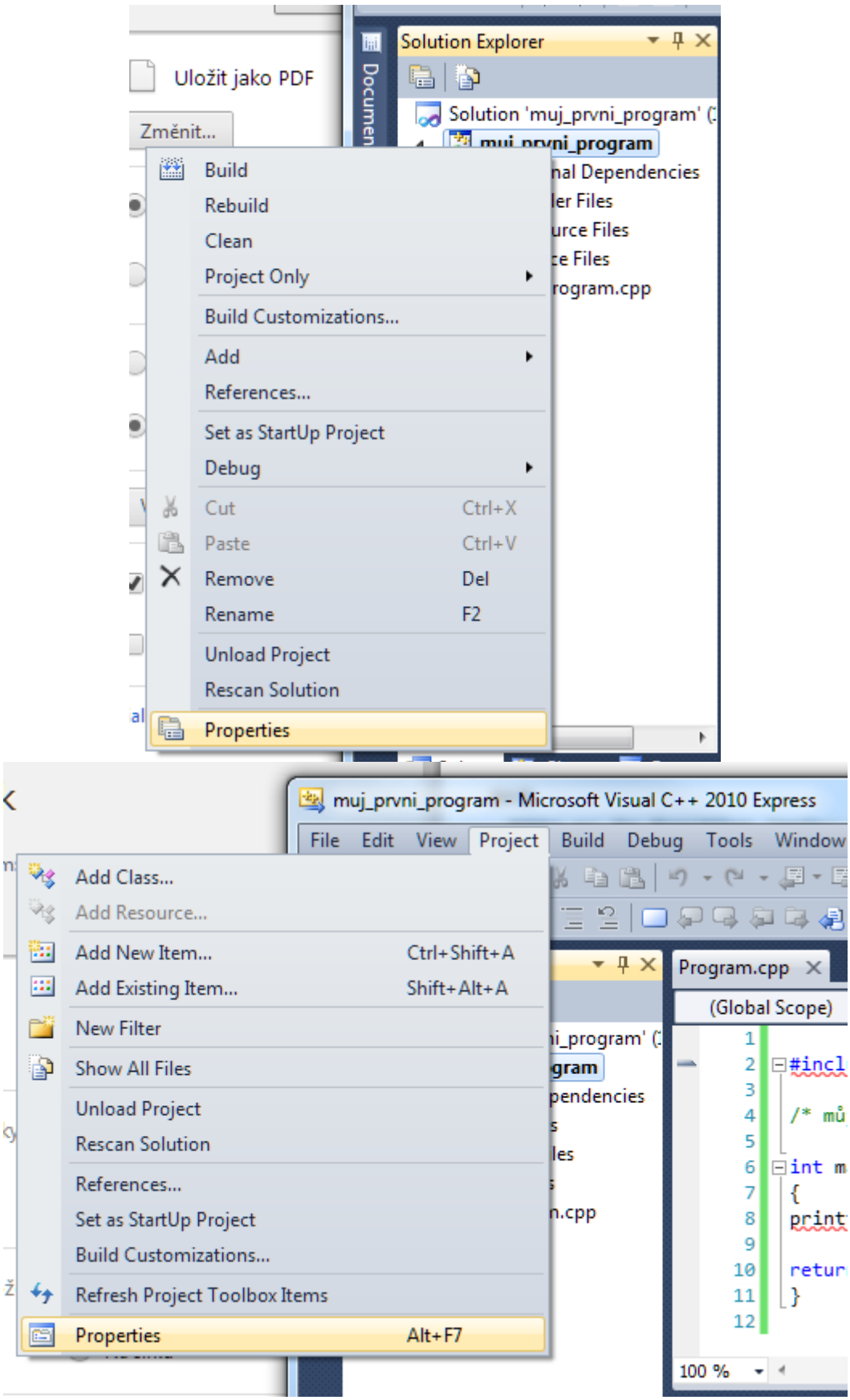
Program se nesestaví, viz poslední řádek výstupu:

„========== 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

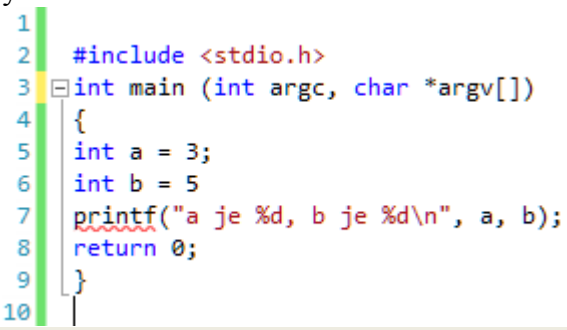


nebo příkazem Properties z menu Project nebo použitím klávesové zkratky Alt+F7.

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>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 části 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

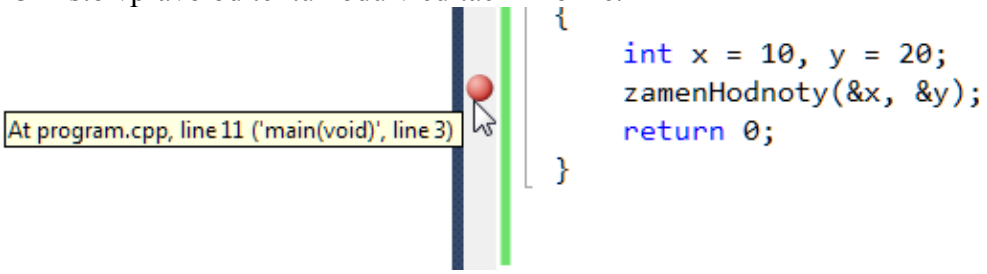
Navázání skutečného parametru na formální lze dosáhnout několika odlišnými způsoby. Ve většině dnešních programovacích jazyků se používají hlavně předávání parametrů hodnotou a odkazem:

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.

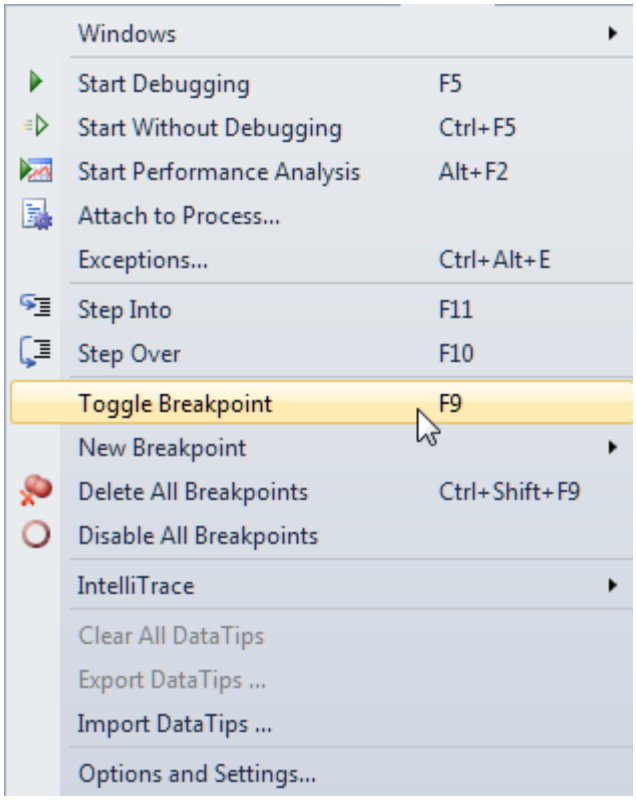
Funkce v jazyce C mohou mít několik parametrů a nejvýše jednu návratovou hodnotu, [1], [5]. Vstupní parametry funkcí je možné v jazyce C předávat pouze hodnotou. Na obr. 12 je zobrazena situace při volání funkce hodnotou.

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

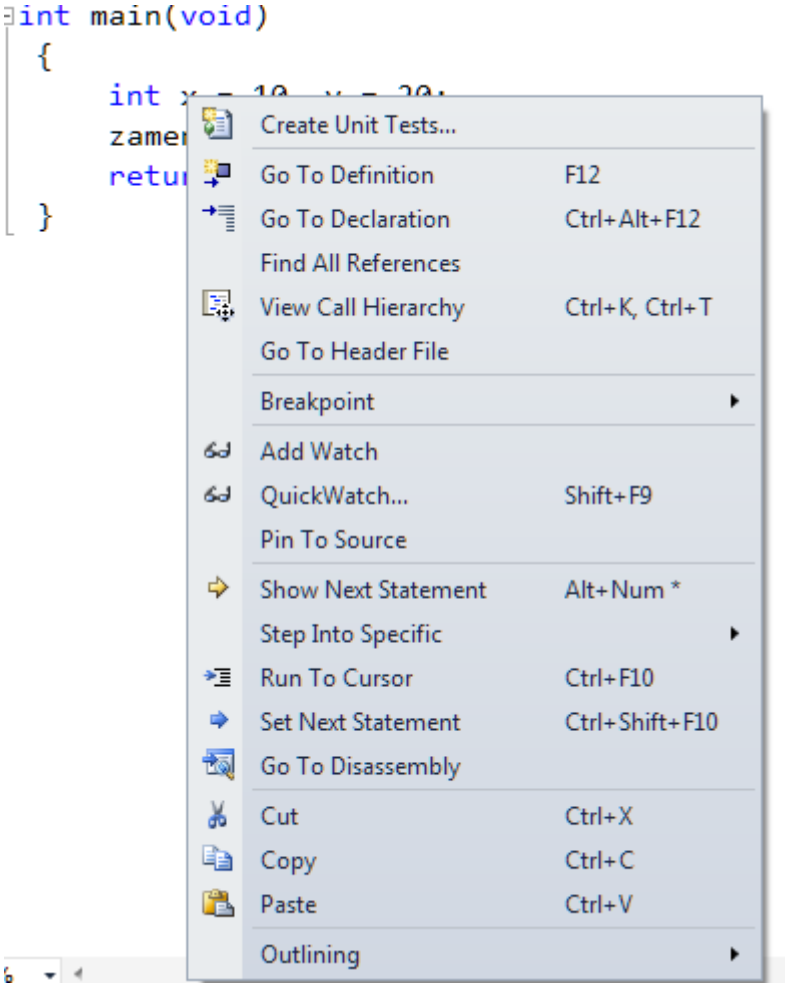
Abychom si názorně ukázali, jak se mění hodnoty proměnných, použijeme mocný ladící nástroj, a to debugger. V programu si nastavíme místo přerušení – breakpoint, na kterém se v režimu ladění zastaví vykovávání programu. To provedeme nejjednodušeji poklepáním na volné místo vpravo od textu kódu v editačním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



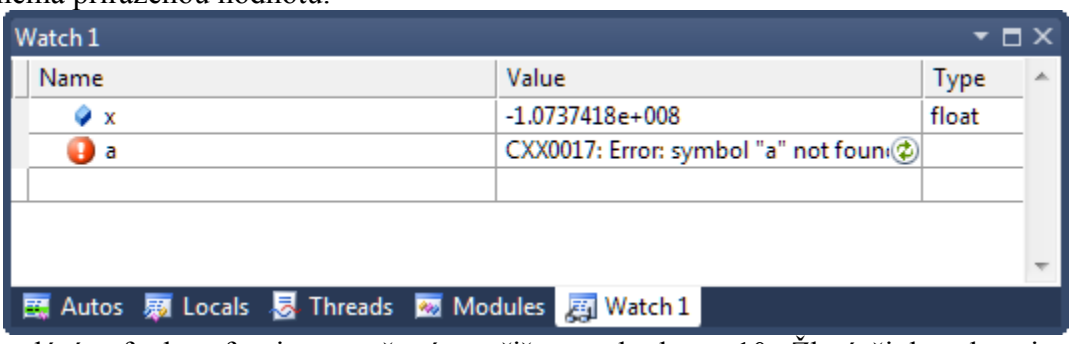
Hodnotu proměnných můžeme sledovat v okně Watch, které slouží pro výpis aktuálních stavů proměnných. Nové proměnné vložíme do sledování tak, že si v programu nastavíme kurzor na aktuální proměnnou a pomocí pravého tlačítka myši vyvoláme kontextovou nabídku. Zvolíme příkaz Add Watch.



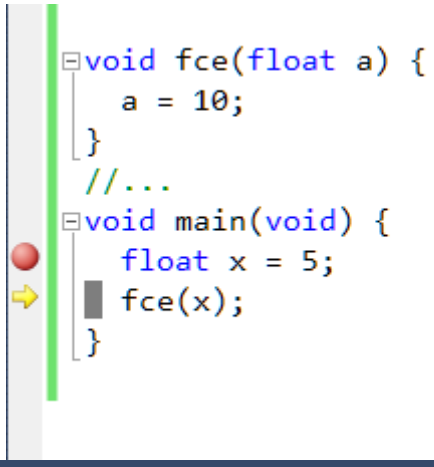
Totéž lze učinit tak, že v textu nastavíme kurzor na řádek kódu, před který chceme vložit bod přerušení. Pak použijeme příkaz z menu Debug -> New Breakpoint->Toggle Breakpoint nebo funkční klávesu F9.

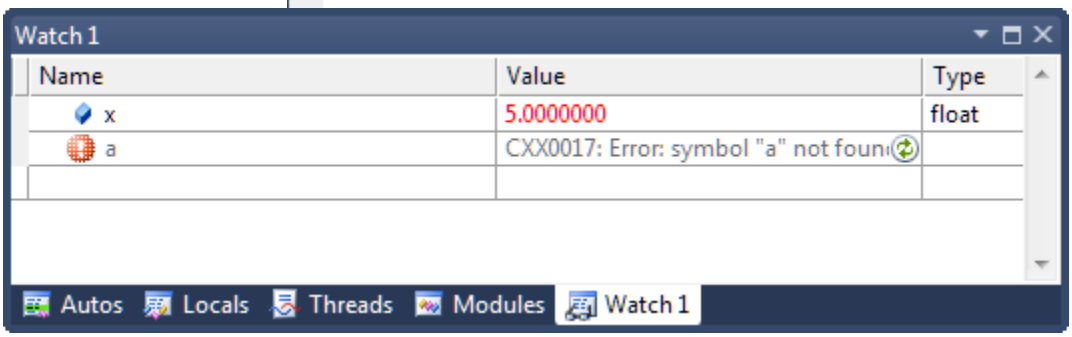


Ve funkci main je definována proměnná x, která je předána jako parametr funkce. Na obrázku ještě nemá přiřazenou hodnotu.

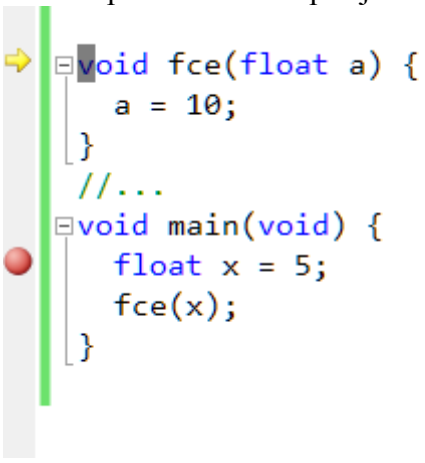


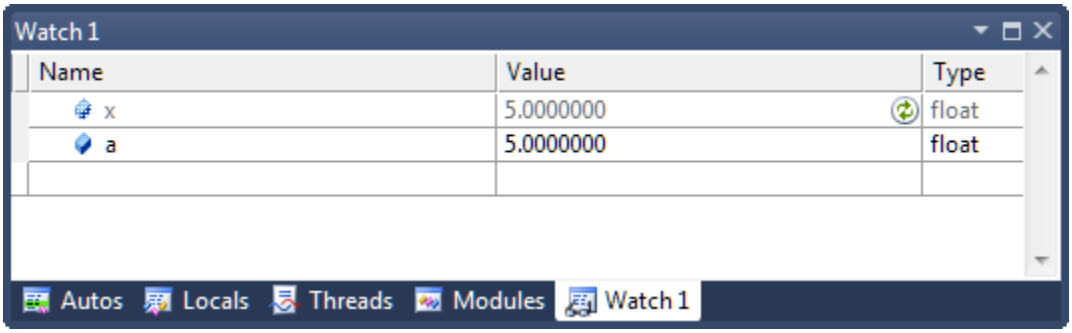
Před voláním funkce fce je proměnné x přiřazena hodnota 10. Žlutá šipka ukazuje místo aktuálně prováděného příkazu.





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.

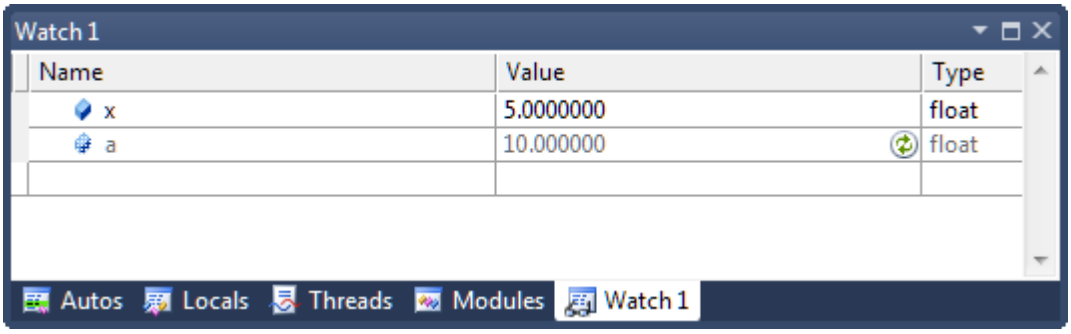




Po vykonání funkce fce a ještě před jejím ukončením je situace v paměti znázorněna na dalším obrázku. Proměnná a je naplněna hodnotou 10.

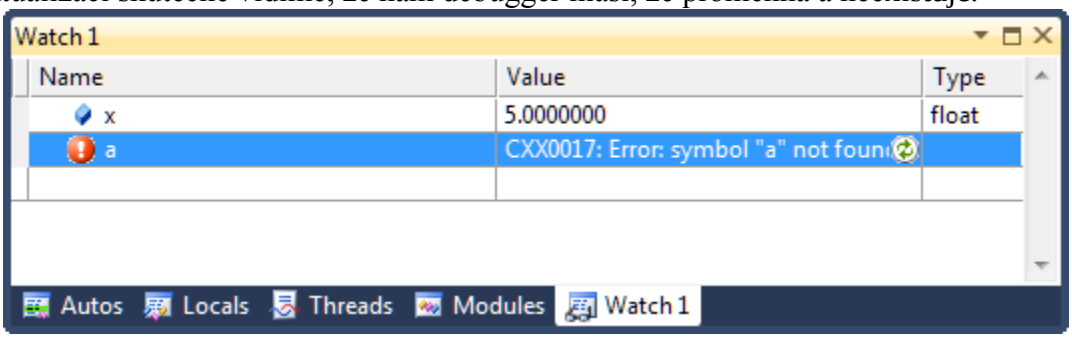

```
void fce(float a) {
   a = 10;
}
//...
void main(void) {
   float x = 5;
   fce(x);
}
```

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.



```
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.

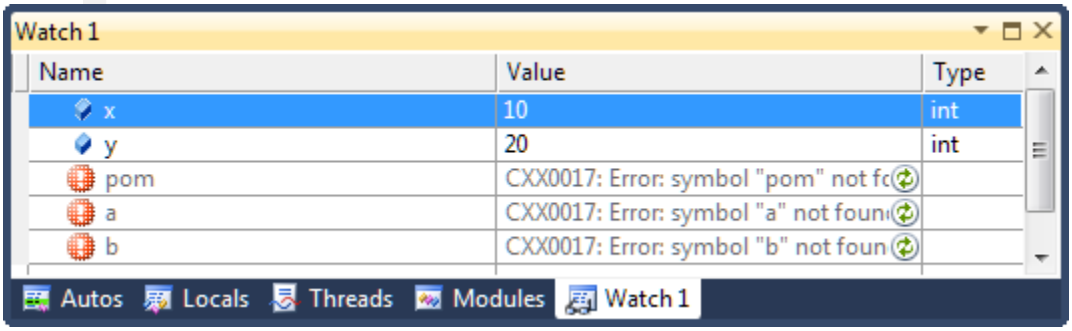
```
void zamenHodnoty(int a, int b) //zameni hodnoty v promennych
{
int pom;
pom = a;
a = b;
b = pom;
}
```

Tuto metodu budeme volat v programu následně:

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

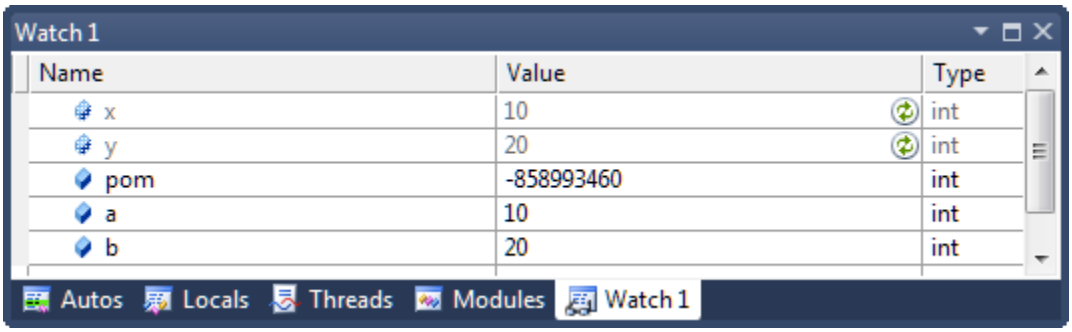
Obdobně, jako v příkladu výše si ukážeme, jaké hodnoty nabývají proměnné x, y a parametry funkce a a b při vykovávání programu. Stav proměnných před volání funkce zamenHodnoty.

```
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;
}
```



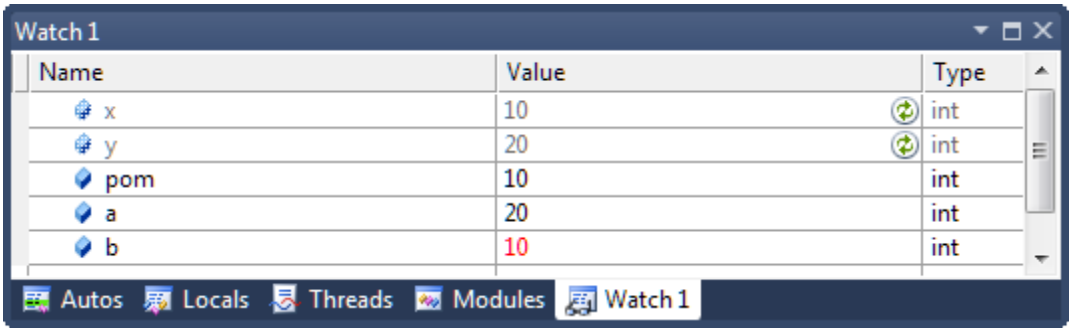
Stav proměnných na začátku vykonávání funkce zamenHodnoty.

```
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;
}
```



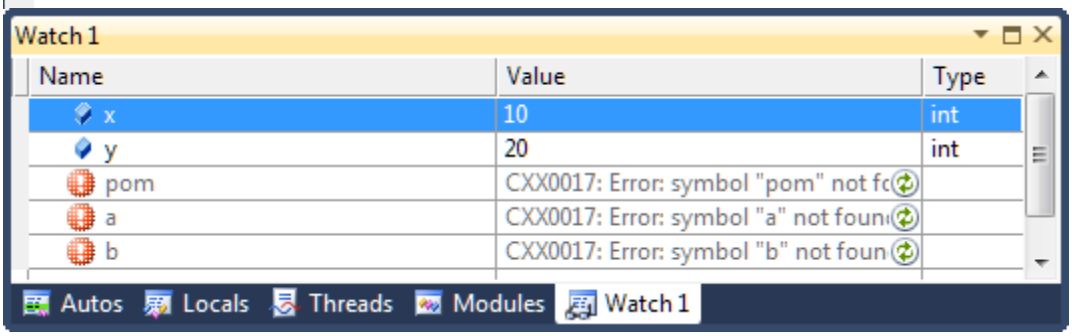
Hodnoty proměnných před opuštěním funkce zamenHodnoty.

```
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;
 }
```



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;
 }
```



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 fc	
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

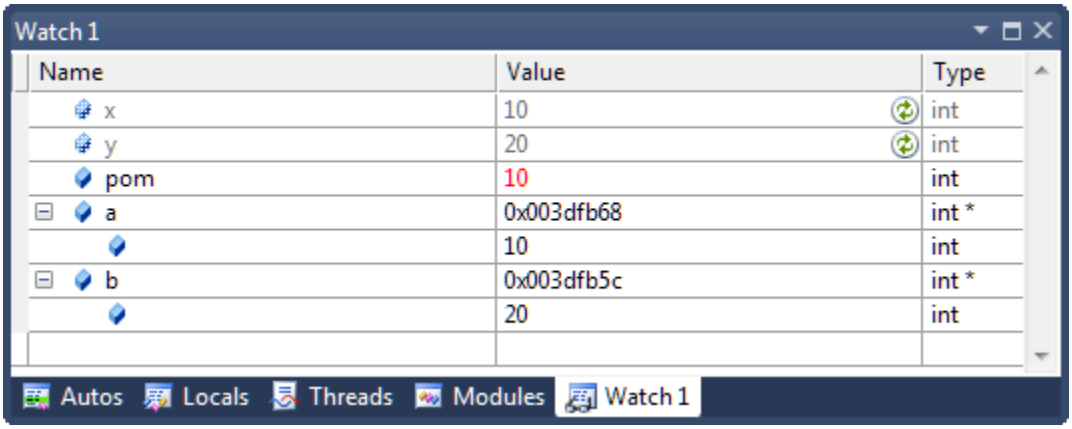
Name	Value	Type
x	10	int
y	20	int
pom	-858993460	int
a	0x003dfb68	int *
*a	10	int
b	0x003dfb5c	int *
*b	20	int

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.

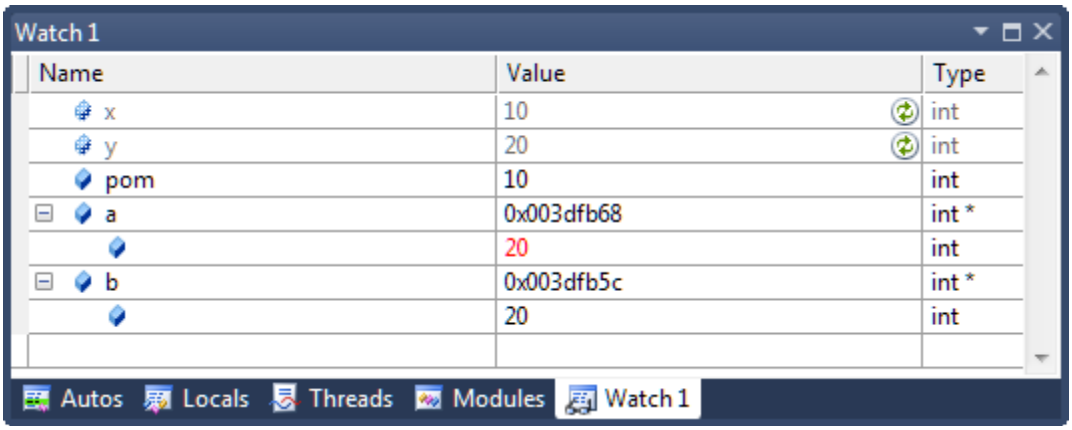
Druhý řádek funkce do proměnné pom neuloží obsah parametru a, ale pomocí operátoru dereference * se získá hodnota paměťového místa s adresou, kterou obsahuje ukazatel a, tudíž se získá hodnota proměnné x, která se uloží do pomocné proměnné pom.

```
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;
}
```

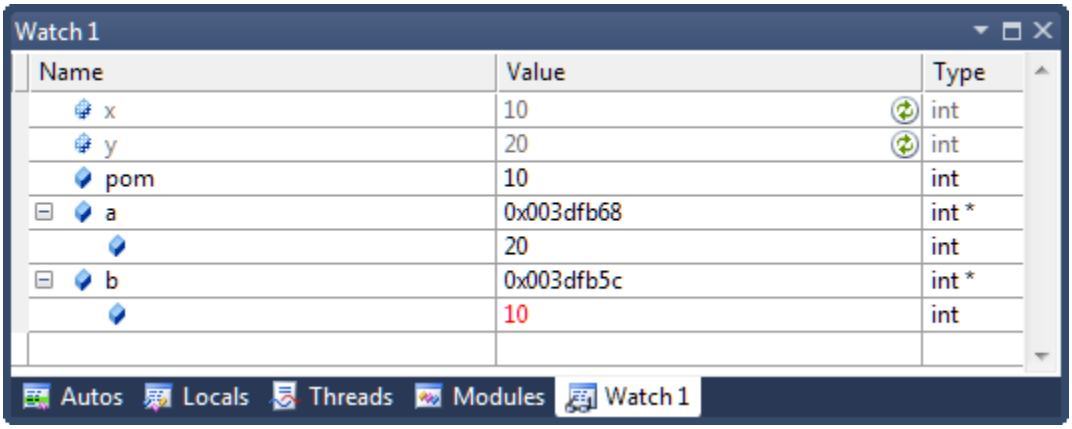


Podívejme se, jak bude vypadat situace po provedení příkazu *a = *b;

```
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;
}
```

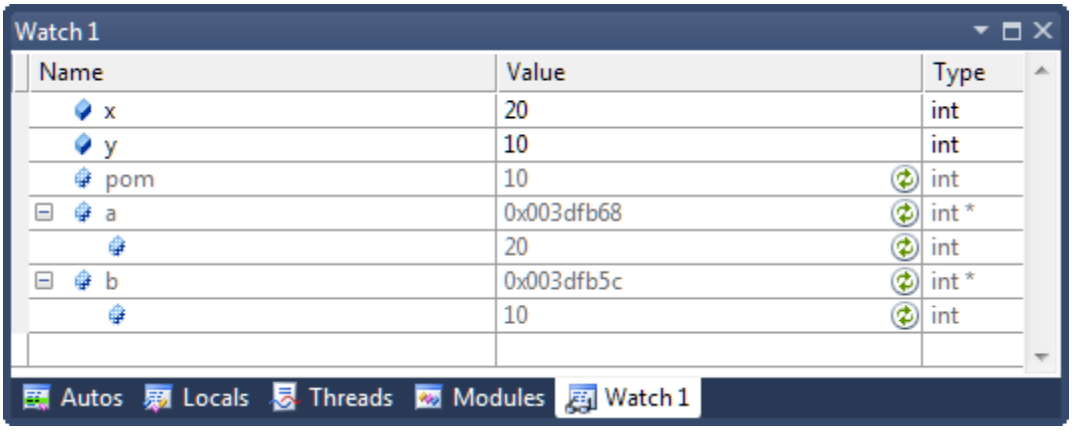


Hodnota na adrese 0x003dfb68 se změnila na hodnotu dvacet, kterou obsahuje proměnná místo s adresou uloženou v ukazateli b. Uloženou získáme nepřímo dereferencí ukazatele *b. Uvědomme si, že se jedná o hodnotu proměnné y.
Před návratem do funkce main bude stav následující:



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;
 }
```



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
```

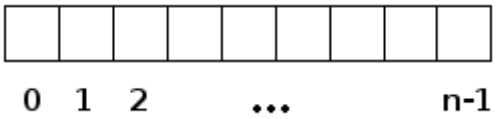
Všimněme si, že první parametr není parametr1, ale. /parametry, tedy vlastní jméno spouštěného programu včetně cesty.
První skutečný parametr má tedy index jedna a ne nula, jak bychom očekávali. Parametry se nijak neupravují a předávají se s pomlčkami i jiným formátováním. Při volání funkce jsou pouze 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 */
}
```