STRUKTUROVANÉ PROGRAMOVÁNÍ

poznámky pro Základy programování 1

Petr Osička

Univerzita Palackého
v Olomouci

Tento text je doplňkovým učebním textem k semináři *Základy programování 1* vyučovanému na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci. Nedělá si ambice být úplnou učebnicí programování v jazyce C, pro to existuje jiná literatura. Ke čtení textu není potřebná předchozí znalost programování.

Text je průběžně upravován a doplňován. *Poslední změna: 26. září 2019*

Za připomínky děkuji následujícím kolegům: M. Kauer, T. Kühr, J. Zacpal, M. Trnečková za připominky.

Petr Osička
Olomouc, léto 2017

Obsah

1 Úvod

Co je program

Program je (zjednodušeně řečeno) sekvence příkazů pro počítač. Pokud program spustíme, počítač vykonává jednotlivé příkazy za sebou, od prvního k poslednímu. Ke svému běhu potřebuje program přístup k procesoru, který příkazy vykonává, a k paměti, kde jsou uložena data, na kterých procesor instrukce vykonává. Program také může přistupovat k zařízením pro vstup a výstup, např. klávesnici, monitoru, pevnému disku, síťové kartě apod.

Program obvykle obsahuje příkazy různých typů, aritmetické, logické, instrukce pro zápis a čtení z paměti apod. Důležitým příkazem je pak příkaz (podmíněného) skoku, který umožňuje (při splnění nějaké podmínky) zvolit, který příkaz bude vykonáván jako další (může být jiný, než příkaz nacházející se v programu za aktuálně vykonávaným příkazem).

Co je programování

Programování je proces vytváření programu. Technicky by bylo možné zapisovat program jako sekvenci příkazů z minulého odstavce. Vytváření programu takovým způsobem je ale pro člověka obtížné: je těžké převést představu o tom, co má program dělat, do takové sekvence příkazů. Snažším způsobem vytvoření program je využití nějakého programovacího jazyka. Programovací jazyky umožňují zapisovat program způsobem, který je člověku bližší. O převod z takového jazyka do sekvence příkazů se pak stará speciální program, kterému říkáme *překladač*.

Programovací jazyky poskytují programátorům možnost psát program s rozumnou strukturou. V programovacím jazyce C, kterým se v textu budeme zabývat, rozumné struktury dosáhneme například následujícím.

- Příkaz (podmíněného) skoku je nahrazen speciálními konstrukcemi pro řízení běhu programu. O takových konstrukcích se dočtete v kapitole 3.
- Program lze rozdělit na funkce. Můžeme si je představit jako „jednoduché miniprogramy," které pak můžeme vhodně kombinovat a vytvářet tak složitější programy (kapitola 5). Princip vykonávání příkazů tak, jak jdou za sebou, je zachován. Je to ale lokálním způsobem, sekvence příkazů se vyskytují ve funkcích.
- K datům nemusíme přistupovat pomocí adres v paměti, ale můžeme si je pojmenovat a přistupovat k nim pomocí jména (tj. používat proměnné, kapitola 2). Můžeme také spolu související data sdružovat do jednoho (pojmenovaného) celku (tj. používat pole a strukturované typy, kapitoly 4 a 6).

Jak vytvořit program

Technicky vypadá tvorba programu takto. Program zapíšeme ve vhodném textovém editoru (který danému programovacímu jazyku rozumí, tj. zvýrazňuje syntax jazyka, podporuje

dobré formátování, umožňuje rozumné vyhledávání apod.). Text programu označujeme jako zdrojový kód. Uložíme jej do textového souboru, typicky s koncovkou `.c`. Poté použijeme překladač, s jehož pomocí soubor se zdrojovým kódem transformujeme na spustitelný program.[1]

Při programování je učitečný i debugger. Je to program, který umožňuje zkoumat, co dělá jiný program, např. ten námi tvořený, za běhu a pomáhá tak s hledáním chyb. Chybám v programu se věnujeme v kapitole 7. Editor, překladač i debugger (a další programy, například profiler) lze integrovat do jednoho prostředí, které umožňuje s nimi pracovat jednotným a pohodlným způsobem. Taková prostředí jsou označována jako IDE (Integrated Development Environment). Je důležité vybrat si dobré IDE (nebo editor, překladač a debugger) a naučit se je používat.[2]

Jak vytvořit program II

Program tvoříme proto, aby za nás počítač něco spočítal. Začínáme tedy s nějakým problémem. Počítač ale neumí (a ani nikdy umět nebude) sám vymyslet způsob, jak daný problém vyřešit. To musí udělat programátor sám.[3] Tvorba programu se z tohoto pohledu dá zhruba rozdělit na dvě fáze. V první fázi je nutno vymyslet správný algoritmus (tj. přesný postup), který problém vyřeší. V druhé fázi pak tento algoritmus implementujeme, tj. zapíšeme jej v programovacím jazyce a vyřešíme detaily, které s tímto jazykem souvisejí.

Algoritmus můžeme vyjádřit s různou mírou přeciznosti. Lze pouze v několika bodech načrtnout, co bude algoritmus dělat a detaily vyřešit až při programování. Pro začátečníky je podle mého názoru ale vhodnější jiný postup. Algoritmus vyjádříme co nejpřesněji, například jej zapíšeme ve vhodném pseudokódu. Pseudokód se obvykle blíží nějaké formě programovacího jazyka, nemusí být ale tak přesný. Můžeme se spolehnout na to, že jej čte člověk, který si může některé věci domyslet (v pseudokódu tak lze používat i přirozený jazyk, matematické formule apod.). Nějaká forma pseudokódu je přítomna prakticky ve všech knihách o algoritmech, například [2]. Vždy obsahuje konstrukty, které se tento semestr naučíme používat v jazyce C: větvení, cykly, pole, funkce a strukturovaná data. Poté, co algoritmus zapíšeme v pseudokódu, si na papír zkusíme jeho běh na jednoduchých příkladech. Tím ověříme, jestli v něm nemáme chybu, a lépe mu porozumíme. Teprve až jsme s algoritmem spokojeni, pustíme se do jeho implementace.

Navrhovaný postup nemusíme uplatňovat pedanticky. S použitím programování lze s algoritmy (nebo jejich částmi) experimentovat. Navrhneme část algoritmu, a naprogramujeme ji. Na běžícím programu pak můžeme zkontrolovat, jestli daná část dělá to, co očekáváme.

[1] Ve skutečnosti je proces mírně kompikovanější (a má více fází), ale pro těď nám bude stačit tento jednoduchý pohled.

[2] Vhodné IDE vám doporučí vyučující na semináři

[3] Nebo si ho někde vyhledá.

2 Základy

2.1 První program

Začneme jednoduchým programem, který nedělá nic jiného, než že vypíše text `Ahoj svete` do konzole.

Program 2.1: První program

```
1 #include <stdio.h>
2
3 int main()
4 {
5    /* program vypise Ahoj svete */
6    printf("Ahoj svete");
7    return 0;
8 }
```

Díky prvnímu řádku můžeme v programu pracovat se standardním vstupem a standardním výstupem. Myslí se tím vstup a výstup z/do konzole (příkazové řádky).

Zbytek kódu je definicí *funkce* `main`. Pro teď si můžeme funkci představit jako miniprogram, kterému předáme vstupní data a on něco provede. Může to být výpočet nějaké hodnoty, vypsání textu do konzole apod. Více si o funkcích řekneme v kapitole 5. Na řádku 3 je hlavička funkce a mezi závorkami na řádcích 4 a 8 je tělo funkce. Funkce `main` je speciální v tom, že provedení celého programu je vlastně provedením těla této funkce. Až do kapitoly 5 budeme všechny programy psát tak, že změníme tělo funkce `main` předchozího programu.

Na řádku 5 se nachází *komentář*. Je to text, který na běh výsledného programu nemá žádný vliv. Slouží pro psaní poznámek. Komentáře začínají dvojicí znaků `/*` a končí dvojicí znaků `*/`, přičemž může může být dlouhý i více řádků. Na řádku 7 vracíme z funkce `main` jako výsledek hodnotu 0. Je to poslední příkaz v programu, který operačnímu systému říká, že program skončil bez chyby. Vlastní vypsání textu se provádí na řádku 6 a to pomocí funkce `printf`. Tuto funkci spustíme tak, že napíšeme její jméno a za to do kulatých závorek její vstupní data — text k vypsání. O takovémto spuštění říkáme, že funkci *voláme* s tímto textem jako *argumentem*. Podrobnosti o volání funkce `printf` si řekneme později v této kapitole. Všimněme si, že oba příkazy v těle funkce jsou ukončeny středníkem. To je obecné pravidlo: každý příkaz musí být ukončen středníkem.

Předtím, než můžeme program spustit, musíme jeho zdrojový kód přeložit pomocí překladače. Překladač je program, který (zjednodušeně řečeno) vezme soubor se zdrojovým kódem a vyrobí z něj spustitelný soubor. V tomto textu není prostor pro vysvětlování práce s překladačem. Pro ukázku si pouze ukážeme pomocí překladače `gcc` na unixovém operačním systému. Předpokládejme, že zdrojový kód je uložený v souboru `program1.c`.

```
$ gcc program1.c
```

Překladač vytvořil spustitelný soubor `a.out`, který po spuštění vypíše do konzole pořadovaný text.

```
$ ./a.out
Ahoj svete
```

Úkoly

1. Ve zvoleném vývojovém prostředí vytvořte, zkompilujte a spusťte program 2.1.
2. Program 2.1 upravte tak, aby vypsal jiný text.

2.2 Proměnné a typy

Pomocí proměnných v programu pracujeme s daty. Proměnná v programu zhruba odpovídá pojmu proměnná, jak jej používáme v matematice. Ve výrazu $x + 2$ můžeme za x uvažovat různá čísla. Pokud řekneme, že $x = 4$, pak je hodnota výrazu 6. Mužeme ovšem také říci, že $x = 10$, pak je hodnota výrazu 12. Pokud chceme znát hodnotu celého výrazu, musíme přiřadit hodnotu x. Nicméně tuto hodnotu můžeme změnit. V programu navíc potřebujeme hodnotu proměnné někam uložit, je jí proto přiřazeno místo v paměti, a musíme vědět, jaký *typ* hodnoty je v paměti uložen, abychom mohli daný obsah paměti správně interpretovat (paměť je jenom sekvence jedniček a nul).

Proměnná je pojmenované místo v paměti, kde je uložena hodnota. K proměnné se váže informace o jejím typu.

Každou proměnnou, kterou chceme v programu používat, je nutno nejdříve definovat. Syntax definice s inicializací[1] (tj. určením hodnoty) proměnné je následující

```
typ jmeno = hodnota;
```

Výrazy zapsané fialovou barvou nepíšeme přímo do kódu, ale na jejich místo dosadíme text odpovídající jejich významu. Například, pokud vytváříme proměnnou `foo`, v kódu na místo `jmeno` napíšeme `foo`. Podobně na místo `typ` napíšeme klíčové slovo označující požadovaný typ a na místo `hodnota` výraz reprezentující nějakou hodnotu. Konkrétní definice proměnné `foo` tedy může vypadat následovně.

```
int foo = 10;
```

Proměnná je typu `int` a její hodnota je nastavena na 10.

Definice proměnné s inicializací zajistí přidělení místa pro proměnnou, vytvoření vazby mezi jejím jménem a tímto místem, a inicializování tohoto místa na správnou hodnotu.

[1] Proměnné lze definovat i bez inicializace, pro začátek ovšem budeme proměnné vždy inicializovat. Neinicializovaná proměnná může totiž mít předem neurčenou (mohlo by se zdát, že i náhodnou) hodnotu. Pokud později v programu do takové proměnné zapomeneme přiřadit hodnotu, kterou chceme, můžeme tím způsobit chybu v programu.

Lze také definovat více proměnných stejného typu na jednom řádku. Stačí oddělit jména proměnných (včetně případné inicializace) čárkou. Pro přehlednost ovšem začátečníkům doporučuji mít definici každé proměnné na zvláštním řádku.

Programátor potom může k proměnné přistupovat pomocí jejího jména. Jména proměnných mohou obsahovat znaky abecedy, číslice a znak podtržítka, ovšem jméno nesmí začínat číslicí.

Jazyk C poskytuje několik základních typů a mechanismus, jak vytvářet typy nové. Všechny základní typy jsou typy číselné. Ke každému typu se vážou dvě důležité informace: klíčové slovo určující daný typ (toto slovo se často používá i jako jméno typu, už jsme se setkali s `int`) a rozsah hodnot, které mohou proměnné daného typu nabývat. Tento rozsah je určen velikostí daného typu v paměti. Typy můžeme rozdělit na celočíselné a s plovoucí čárkou, podle toho, zda hodnoty daného typu připouští desetinná čísla. Pokud napíšeme před klíčové slovo celočíselného typu slovo `unsigned`, je typ tzv. *bezznaménkový*, tedy hodnoty tohoto typu nemohou být záporné. S klíčovým slovem `signed`, nebo bez specifikace znaménkovosti, je typ vždy *znaménkový*. V následující tabulce je shrnutí všech základních typů.

Celočíselné typy
`char`, `int`, `short int`, `long int`, `long long`
Typy s desetinnou čárkou
`float`, `double`, `long double`

Velikosti některých typů nejsou pevně určeny a mohou se na počítačích různých hardwarových architektur lišit. Protože je často potřeba velikosti jednotlivých typů v programu znát, existuje způsob, jak tyto velikosti zjistit. Dělás se to pomocí operátoru operátoru `sizeof`, který pro svůj argument, kterým je buď typ nebo proměnná, vrátí jeho velikost v bytech. Hodnota, kterou `sizeof` vrátí je sama typu `size_t`. To je celočíselný bezznaménkový typ, který vznikl přejmenováním některého ze základních typů (na různých architekturách to může být různě). V následujícím kousku kódu je ukázka použití `sizeof`,

```
int k = 10;
float f = 12;

/*zjisteni velikosti typu float*/
size_t float_size = sizeof(float);

/*zjisteni velikosti promenne k*/
size_t int_size = sizeof(k);
```

Typ `char` je typ s velikostí 8 bitů a je tedy schopen uchovat 256 různých hodnot. Bezznaménkový `char` má rozsah $[0, 255]$, a znaménkový typicky $[-128, 127]$. V programu se `char` často používá k reprezentaci znaků abecedy, číslic a dalších základních znaků, které se vyskytují v ASCII tabulce (tabulka „číslující“ jednotlivé znaky). Znakové konstanty jsou typu `char` a zapisují se pomocí apostrofů před a za znakem, jako v následujícím příkladu.

```
/* promena foo ma hodnotu rovnu hodnote
   znaku a v ASCII tabulce */
char foo = 'a';
```

Je důležité si uvědomit, že konstanta `'a'` je ve skutečnosti číslo. Její hodnota je dána ASCII tabulkou, pro `'a'` je to hodnota 97.

Typ `short int` (zkráceně také `short`) má alespoň 16 bitů, `int` má typicky přirozenou velikost čísla na daném počítači, alespoň však 16 bitů, `long int` (zkráceně také `long`) má velikost alespoň 32 bitů, `long long` má alespoň 64 bitů. Uvedené typy jsou koncipovány tak, aby pokryly vhodný rozsah velikostí. Můžeme se spolehnout na to, že posloupnost velikostí typů v pořadí, v jakém jsme si je uvedli, je vždy neklesající. Celočíselné konstanty jsou považovány za typ `int`, pokud napíšeme za konstantu `l` nebo `L`, bude považována za typ `long`.

```
123456   /* typ int */
123456L  /* typ long */
```

Typ `float` je obvykle typ s tzv. jednoduchou přesností, to odpovídá zhruba číslům s 6 až 9 číslicemi, `double` má obvykle dvojitou přesnost, to odpovídá zhruba 15 až 17 číslicím, `long double` je obvykle číslo se čtyřnásobnou přesností, to je zhruba 33 až 36 číslic. Podobně jako u celočíselných typů se mohou velikosti typů z tohoto odstavce lišit v závisloti na konkrétní platformě. Desetinné konstanty zapisujeme buď s desetinnou tečkou, případně v exponenciálním zápise. Jsou považovány za typ `double`, přípona `f` nebo `F` indikují konstantu typu `float`, `l` nebo `L` konstanty typu `long double`

```
1.234    /* zapis s desetinnou teckou */
1e-3     /* exponencialni zapis */
1.2e-4   /* exponencialni zapis */
```

2.3 Operátory

Operátor představuje operaci, kterou lze provést s nějakými argumenty. Představme si například operaci sčítání, kterou lze provést se dvěma čísly. Ke každému operátoru je třeba vědět, na kolik argumentů jej lze aplikovat. Tomuto číslu se říká *arita*. Operátorům s aritou jedna říkáme *unární*, operátorům s aritou dva říkáme *binární* a operátorům s aritou tři říkáme *ternární*.

Pro základní operace s číselnými hodnotami slouží aritmetické operátory. Jejich souhrn je v následující tabulce

Operátor	Argumenty	Popis
-	1	mění znaménko argumentu
+	2	součet argumentů
-	2	rozdíl prvního a druhé argumentu
*	2	součin argumentů
/	2	podíl argumentů
%	2	zbytek po celočíselném dělení prvního argumentu druhým

Pro aritmetické operátory platí stejná pravidla jako pro jejich matematické protějšky, včetně toho, že nelze dělit nulou. Pokud vytváříme výraz, který obsahuje více typů operátorů, je nutné vědět, v jakém pořadí se budou operátory aplikovat. Tato pravidla jsou také stejná jako ta používaná v matematice. Například u výrazu

```
-1 + 2 * 3
```

nejdříve vynásobíme čísla 2 a 3 a k výsledku příčteme -1. Výraz by se tedy vyhodnotil na na 5. Pokud chceme operátory vyhodnotit v jiném pořadí, je nutné použít kulatých závorek (opět podobně jako v matematice).

```
(-1 + 2) * 3
```

V tomto případě bude hodnota výrazu 3, protože jako první se provede sčítání. Argumenty operátoru mohou být i proměnné, výsledek se spočítá z jejich hodnot. Například pro

```
int a = 1, b = 2, c = 3;
```

by se výraz

```
(a + b) * c
```

vyhodnotil na 9.

Operátor přiřazení = používáme jako

```
l-value = r-value
```

přičemž `l-value` musí být místo, do kterého se dá přiřadit hodnota, my známe zatím pouze proměnnou. Operátor zapíše na místo `l-value` hodnotu `r-value`, a `r-value` vrátí jako výsledek. Například

```
int a = 5;
int b = 6;
int c = 0;
int d = 0;

c = 7; /* hodnota c je ted 7*/
c = b; /* hodnota c je ted 6 */
b = 3; /* hodnota b je ted 3, hodnota c zustava 6!!*/
c = c - b; /* hodnota c je ted 3 */
```

Operátor přiřazení lze kombinovat s aritmetickými operátory, například `a += b` odpovídá `a = a + b`, podobně fungují i operátory `-=`, `*=`, `/=`, `%=`.

Operátor `++` zvyšuje hodnotu proměnné o 1. Lze jej použít před nebo za proměnnou. Při použití před proměnnou operátor vrací novou hodnotu, při použití za proměnnou vrací hodnotu před zvýšením.

```
int z = 0;
int b = 10;

/* po provedeni dalsiho radku: z = 10, b = 11*/
z = b++;
/* po provedeni dalsiho radku: z = 12, b = 12*/
z = ++b;
```

2.4 Druhý program

První program, který jsme vytvořili, nebyl moc užitečný. Pokusme se vytvořit něco zajímavějšího, například program na převedení teploty ve stupních Fahrenheita na teplotu ve stupních Celsia. Stačí použít vzoreček

$$c = \frac{5(f - 32)}{9},$$

kde c jsou stupně Celsia a f jsou stupně Fahrenheita, a nakonec výsledek vypíše do konzole.

Program 2.2: Převod stupňů Fahrenheita na stupně Celsius

```
1  #include <stdio.h>
2
3  int main()
4  {
5    /* prevedeme 120 stupnu fahrenheita*/
6    float fahrenheit = 120.0,
7    float celsius = 0;
8
9    /* spocteme (C) podle vzorce */
10   celsius = (5 * (fahrenheit - 32)) / 9;
11
12   /* vypiseme vysledek */
13   printf("Teplota je %f (C)",celsius);
14
15   return 0;
16 }
```

Jak vidíme, základní struktura prvního a druhého programu je stejná. Změnilo se pouze tělo funkce `main`. V programu potřebujeme reprezentovat teplotu ve stupních Fahrenheita a Celsia, za tímto účelem jsme na řádku 5 a 6 definovali proměnné `fahrenheit` a `celsius` schopné nabývat desetinných hodnot, a proměnnou `fahrenheit` inicializuje na hodnotu, pro kterou chceme spočítat teplotu ve stupních celsia.

Na řádku 10 spočítáme podle vzorce teplotu ve stupních Celsia a uložíme ji do proměnné `celsius`. Nakonec na řádku 13 vypíšeme výslednou teplotu do konzole.

V programu se objevilo několik již známých věcí: zavedli jsme proměnné pro desetinná čísla a použili jsme aritmetické operátory a operátor přiřazení. Ovšem funkci `printf` jsme použili novým způsobem. Popišme si ji tedy podrobněji.

2.5 Standardní výstup

Funkce `printf` slouží k výpisu do konzole. Akceptuje proměnný počet argumentů, můžeme jí předat jeden argument, jak jsme viděli na řádku 6 Programu 2.1, nebo více argumentů, jak jsme viděli například na řádku 15 Programu 2.2. Obecný předpis volání `printf` je následující:

```
printf(format, h1, h2,...);
```

Prvním argumentem je řetězec, která určuje text k vypsání. Tento řetězec může obsahovat speciální formátovací instrukce, které označují místa, na než se vypisují hodnoty předané jako další argumenty, a to postupně zleva doprava. Podíváme-li se například na řádek

```
printf("Teplota je %f (C)", celsius);
```

z programu 2.2, pak si můžeme všimnout `%f` ve formátovacím řetězci. To je právě ona formátovací instrukce, na jejíž místo se vypíše hodnota předaná jako další argument, tedy hodnota proměnné `celsius`. Formátovací instrukce vždy začíná znakem `%` následovaným jedním nebo více dalšími znaky, které určují, jaký typ hodnoty je třeba vypsat. V předchozím příkladě to byl znak `f` určující, že se má vypsat desetinné číslo. Následující tabulka zachycuje nejčasteji používané formátovací instrukce (ostatní lze najít v referenční příručce u funkce `printf`.)

Formátovací instrukce	Tištěná hodnota
`%c`	znak
`%d` nebo `%i`	celé číslo znaménkově
`%f`	desetinné číslo
`%e`	exponenciální zápis

2.6 Typy vs aritmetické operátory

Výsledek použití aritmetického operátoru je většinou toho typu, který má z typů všech jeho argumentů největší prioritu[2] Typy s plovoucí čátkou mají vyšší prioritu než typy celočíselné, v rámci jedné skupiny je pak priorita určena pomocí velikosti daného typu tak, že větší typy mají větší prioritu. U operátoru přiřazení je výsledek vždy toho typu, jakého typu je `l-value`. Pokud takto změníme typ hodnoty na typ, který je menší, dojde k zaokrouhlení nebo k jiné ztrátové úpravě této hodnoty. Dobře si projděte následující příklady.

```
int i = 10, j = 4, k;
float f, g = 10.0, h;

/* f a k budou mit hodnotu 2, protoze i a j jsou
   typu int, podil je tedy taky int. Vysledek podilu
   dostaneme odstranenim desetinne casti  */
k = i / j;
f = i / j;

/* f bude mit hodnotu 2.5, protoze g je
   float a tedy i podil g / j je float */
f = g / j;

/* k bude rovno 2, ztrati se desetinna cast */
k = f;
```

[2] Někdy se může stát, že výsledek je dokonce ještě většího typu, to je ovšem ve velmi specifických situacích, o kterých bychom se mohli dočíst ve standardu jazyka. Protože s progamováním začínáme, nebudeme se tím teď trápit.

```
/* h bude rovno 0.5 */
h = f - k;
```

Úkoly

Pokud v úkolech hovoříme o tom, že program má zpracovávat nějaký vstup, myslíme tím, že v řešitel si v programu vytvoří proměnné, které budou daný vstup obsahovat, a pod tím napíší kód odpovídající danému úkolu. Je-li například níže po studentu požadováno, aby spočítal a vypsal obsah čtverce, chce se po něm, aby vytvořil následující program

```
#include <stdio.h>

int main(){

  int strana_ctverce = 10;

  /* tady je kod, ktery spocita obsah ctverce,
     a vypise jej */
}
```

1. Upravte Program 2.2 tak, aby převáděl částku v USD na částku v českých korunách.
2. Upravte program 2.2 tak, aby převáděl úhel ve stupních na úhel v radiánech.
3. Napište program, který vypíše velikosti všech základních typů na vašem počítači.
4. V referenční příručce najděte a vyzkoušejte různé instrukce použitelné ve funkci `printf`, včetně instrukcí pro nový řádek a tabulátor.
5. Napište program, který vypočte hodnotu matematického výrazu

$$\frac{3}{2} + 12 - \frac{5^3 - 2}{6}$$

   a vypíše ji na obrazovku.
6. Napište program, který spočítá a vypíše obvod a obsah čtverce, známe-li jeho stranu.
7. Napište program, který spočítá průměr čísel 1, 2, 3, 4, 50, 100, 1002.14 a vypíše jej na obrazovku.
8. Napište program, který pro daná dvě čísla vypíše jejich součet, rozdíl a součin.
9. Napište program, který načte velké písmeno, a vypíše jej jako malé písmeno.
10. Napište program, který vypíše první a poslední číslici tříciferného čísla.
11. Napište program, vypíše osmimístné číslo, chápané jako datum ve tvaru DDMMYYYY jako opravdové datum. Například pro číslo 13122002 vypište `13. 12. 2002`

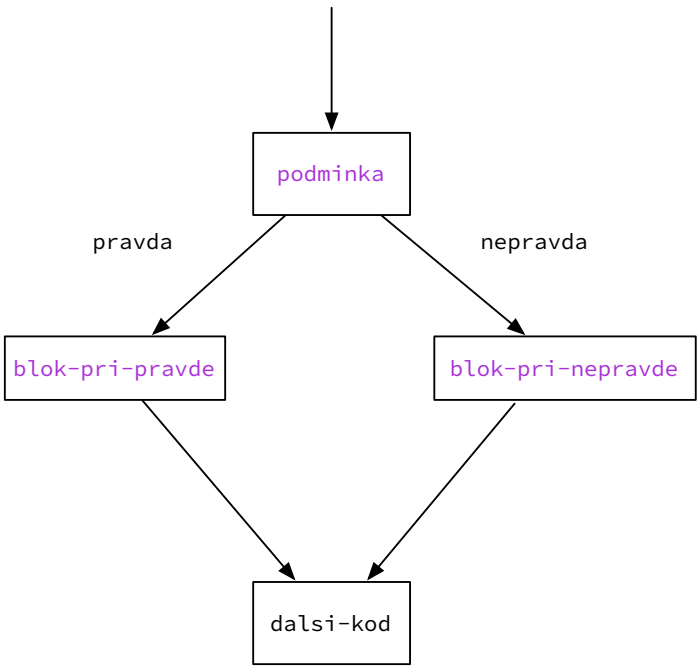
3 Kontrola běhu programu

3.1 Větvení programu

V programu 2.2 vůbec nekontrolujeme, jakou hodnotu uživatel vloží z klávesnice. Řekněmě, že budeme chtít, aby program převáděl jenom teploty větší než absolutní nula (-459.16 stupňů Fahrenheita). K tomu musíme mít možnost ověřit, jestli je proměnná `fahrenheit` větší než -459.16 a pokud ano, spočítat převod jako doposud, pokud ne, oznámit, že uživatel zadal nesmyslnou teplotu. Situaci, kdy se program (nebo obecně algoritmus) na základě nějaké podmínky rozhodne pro jednu z několika možností jak pokračovat, označujeme jako *větvení*. Základní konstrukci pro větvení v jazyce C je `if`, obecný zápis této konstrukce vypadá následovně.

```
if (podminka)
   blok-pri-pravde
else
   blok-pri-nepravde
```

Výraz `podminka` je brán jako logická hodnota. Logické hodnoty existují pouze dvě, buď pravda a nepravda. Pokud je podminka pravdou, provede se `blok-pri-pravde`, v opačném případě se provede `blok-pri-nepravde`. Blok kódu je buď jeden příkaz nebo souvislý kus kódu uzavřený mezi složené závorky. Na místa `blok-pri-pravde` a `blok-pri-nepravde` lze napsat právě jeden blok kódu. Část počínající `else` je nepovinná a lze ji vynechat. Větvení lze graficky zachytit pomocí diagramu.



Hodnotu 0 (jakéholiv typu) považujeme za nepravdu, vše ostatní za pravdu. Pro podmínky se hodí binární operátory porovnání `<`, `<=`, `>`, `>=`, `==`, `!=` (operátor `!=` je nerovnost), jejichž význam je intuitivní. Zdůrazněme pouze, že operátor pro test rovnosti `==` obsahuje dvě rovnítka a je různý od operátoru přiřazení `=`. S logickými hodnotami lze dále pracovat pomocí logických operátorů, které jsou protějšky základních logických spojek. Operátor `&&` odpovídá spojce *a*, operátor `||` spojce *nebo* a unární operátor `!` psaný před argument odpovídá *negaci*. Operátory mají stejné vlastnosti jako logické spojky, které reprezentují.

Logické operátory `&&` a `||` se vyhodnocují *líně*. Pokud se zjistí, že zleva první argument operátoru `&&` je nepravdivý, pravdivost druhého argumentu už se nezjišťuje (protože výsledek už je určitě nepravda). U `||` to funguje analogicky.

Program 3.1: Převod stupňů Fahrenheita na stupně Celsius

```c
#include <stdio.h>

int main()
{
  float fahrenheit, celsius;

  /* nacteme teplotu*/
  printf("Zadejte teplotu (F): ");
  scanf("%f", &fahrenheit);

  /* prevedeme na stupne celsia, pokud je ve spravnem rozsahu */
  if (fahrenheit >= -459.16) {
    celsius = (5 * (fahrenheit - 32)) / 9;
    printf("Teplota je %f (C)", celsius);
  }
  else
    printf("Teplota musi byt vyssi nez -459.16");

  return 0;
}
```

Kontrukci `if` lze vnořovat, tedy `blok-pri-pravde` i `blok-pri-nepravde` mohou obsahovat další `if`. Při vnořování je někdy potřeba použít závorky i pro vymezení bloků s jedním příkazem, abychom tak dali najevo, které `if` a `else` patří k sobě. Bez použití závorek patří `else` vždy k nejbližšímu `if`. V situacích, kde si nebudete jistí, radši používejte závorky. Následující dvě konstrukce jsou různé.

```c
int a = 5, b = 1, c = 3, foo = 10;

/* tady bude foo rovno 10*/
if (a > b) {
  if (b > c)
    foo = b;
}
else foo = a;
```

```
/* tady bude foo rovno 3*/
if(a > b)
  if (b > c)
    foo = b;
  else
    foo = c;
```

Můžeme to ověřit i pomocí diagramů. Diagram prvního použití `if` je na následujícím obrázku.

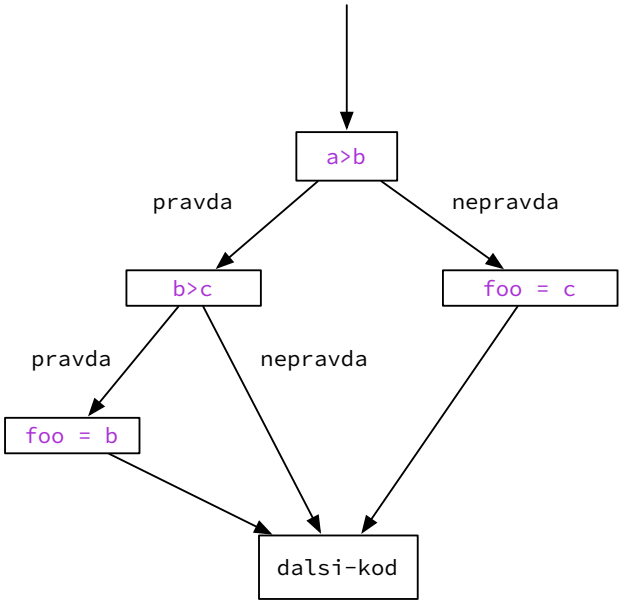
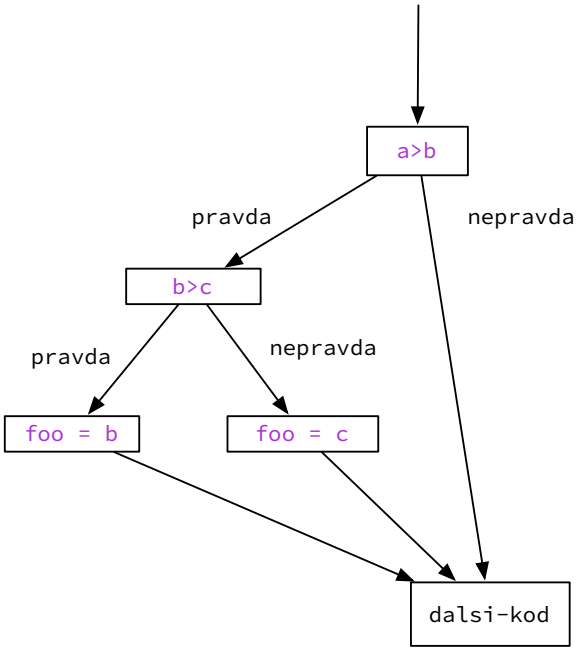


Diagram druhého použití je od toho prvního různý.



V následujícím příkladu lze některé (i všechny) závorky vynechat a kód bude dělat stejnou věc.

```
int a = 1, b = 2, c = 3, foo = 4;

 /* s plnym pouzitim zavorek, foo bude 3 */
if (a >= b && a >= c) {
  foo = a;
}
else {
  if (b >= c) {
  foo = b;
  }
  else {
    foo = c;
  }
}

/* nektere zavorky vynechany, foo bude 3 */
if (a >= b && a >= c) {
  foo = a;
}
else if (b >= c) {
  foo = b;
}
else {
    foo = c;
}

/* vsechny zavorky vynechany, foo bude 3 */
if (a >= b && a >= c)
  foo = a;
else if (b >= c)
  foo = b;
else
  foo = c;
```

V krátkosti uvedeme ještě další konstukce, pomocí kterých lze v jazyce C provést větvení. Pokrývají speciální případy, kdy by bylo použití `if` nešikovné nebo nepřehledné, museli bychom `if` použít mnohokrát, komplikovaně `if` vnořovat apod. Stejný program bychom mohli napsat i s pomocí `if`, ale z uvedených důvodů může být lepší použít jinou konstrukci.

Pokud v podmínkách konstrukcí `if` kontrolujeme rovnost s celočíselnou konstantou, tedy například

```
if (x == 1)
   printf("x je jedna");
else if (x == 2)
   printf("x je dva");
else if (x == 3)
   printf("x je tri")
else
   printf("x neni 1,2 nebo 3");
```

můžeme použít `switch`. Příklad nahoře bychom pomocí `switch` přepsali jako

```
switch (x){
  case 1:
    printf("x je jedna");
    break;
  case 2:
    printf("x je dva");
    break;
  case 3:
    printf("x je tri");
    break;
  default:
    printf("x neni 1,2 nebo 3");
}
```

Obecná syntax konstrukce `switch` je následující

```
    switch (vyraz) {
      case konstanta_1:
          prikazy_1
          break;
      case konstanta_2:
          prikazy_2
          break;
      ...

      default:
         prikazy
    }
```

Při provádění kódu odshora hledáme shodu hodnoty `vyraz` s některou z konstant `konstanta_1`, `konstanta_2` .... S `default` se výraz shoduje vždy. Při shodě začneme provádět příkazy počínaje prvním příkazem za shodnou konstantou a konče prvním příkazem `break`, na který se narazíme. Konstrukci `switch` si tedy můžeme představit jako blok příkazů obsahující řádky s `case` a příkazy `break`. Provedeme blok příkazů, který začíná prvním příkazem za `case` řádkem shodujícím se s hodnotou `vyraz`, a který končí příkazem `break` nebo koncem `switch`.

Pro jednoduchá větvení lze použít podmínkového operátoru `?:`. Tento operátor je ternární (bere tři argumenty). Jeho syntax je

```
    podminka ? vyraz1 : vyraz2;
```

Pokud je `podminka` pravdivá, je výsledkem hodnota, kterou obdržíme vyhodnocením `vyraz1`, v opačném případě je výsledkem hodnota, kterou obdržíme vyhodnocením `vyraz2`, viz následující příklad.

```
/* nasledujici pouziti if a operatoru ? vedou ke stejne hodnote x*/

if(a < b)
  x = a;
else
  x = b;
```

```
x = (a < b) ? a : b;
```

Úkoly

1. Upravte Program 2.2 tak, aby v případě, že načteme hodnotu větší než 400, program tuto hodnotu nepřeváděl, ale vypsal do konzole informaci, že je nutné zadat hodnotu menší než 400.
2. Napište program, který načte 3 čísla a poté vypíše nejmenší z nich.
3. Napište program, který rozhodne, zdali je daný rok přestupný.
4. Načtěte ze vstupu souřadnice bodu v na euklidovské ploše. Vypište, do kterého kvadrantu bod patří (nebo na které ose leží).
5. Načtete ze vstupu souřadnice středu kružnice a její poloměr. Vypište kvadranty, do kterých kružnice zasahuje.
6. Ze vstupu načtěte délky stran trojúhelníka. Rozhodněte, jestli se trojúhelník s danými delkami stran existuje. Pokud ano, rozhodněte, jestli je pravoúhlý, rovnostranný nebo obyčejný.
7. Napište program, který načte ze vstupu číslici a vypíše ji slovně.
8. Napište program, který rozhodne o známce na základě počtu bodů získaných v testu. Známkování je dáno následující tabulkou

Body	Známka
90 - 100	A
80 - 89	B
76 - 79	C
71 - 75	D
60 - 70	E
0 - 59	F

9. Napište program pro určení počtu a typu (reálné vs. komplexní) řešení kvadratické rovnice $ax^2 + bx + c = 0$.

3.2 Cykly

V programu je obykle potřeba opakovaně provádět blok kódu. Představme si například, že máme kus kódu, který umí z textového souboru přečíst jeden řádek. Pokud pak chceme přečíst celý soubor, můžeme to provést tak, že opakovaně provádíme kód pro čtení jednoho řádku, a to tak dlouho, dokud nenarazíme na konec souboru. Pokud provádíme opakovaně stejný blok kódu, řekneme že *cyklíme* nebo *iterujeme*. Konstrukce pro cyklení jsou obvykle realizovány tak, aby umožnili provádění bloku kódu tak dlouho, dokud platí nějaká podmínka, přesně tak jako u příkladu s načítáním souboru.

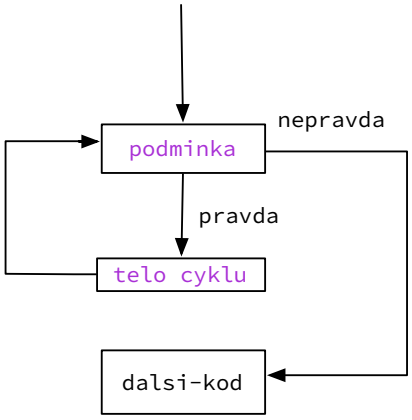
Řekněme, že chceme naprogramovat Euklidův algoritmus pro nalezení největšího společného dělitele dvou čísel. Pro čísla a a b, označíme jejich největšího společného dělitele jako $\gcd(a, b)$. V algoritmus využijeme toho, že $\gcd(a, 0) = a$ a $\gcd(a, b) = \gcd(b, a \bmod b)$, pokud je b nenulové ($a \bmod b$ je zbytek po dělení čísla a číslem b). Největšího společného dělitele a a b spočteme tedy tak, že dokud se b nerovná nule, opakujeme následující: do pomocné proměnné r spočítáme $a \bmod b$, do a přiřadíme b a do b přiřadíme r. V momentě, kdy b je nula, je výsledek v proměnné a. Program implementující Euklidův algoritmus vypadá následovně.

```
#include <stdio.h>

int main()
{
  int a,b,r;
  printf("Zadejte dvojici cisel: ");
  scanf("%i %i", &a, &b);

  while(b != 0) {
    r = a % b;
    a = b;
    b = r;
  }

  printf("Vysledek je %i\n", a);
}
```

K implementaci opakování kódu při platnosti podmínky `b != 0`, jsme použili konstrukci `while`.

```
  while (podminka)
    telo cyklu
```

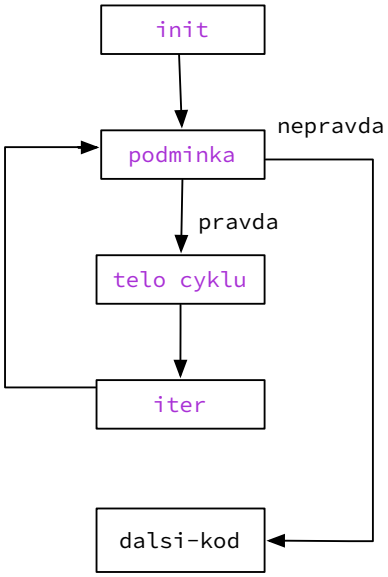
`podminka` je vyraz, který se vyhodnotí na logickou hodnotu, `telo cyklu` je blok kódu. Konstrukce dělá následující: pokud se výraz `podminka` vyhodnotí na pravdu, provede `telo cyklu`, jinak cyklus skončí. Po provedení těla cyklu se opět pokračuje testem pravdivosti výrazu `podminka`. Viz následující diagram.



Druhou konstrukcí pro iteraci je cyklus `for`.

```
for (init; podminka; iter)
  telo cyklu
```

`init` je seznam příkazů oddělených čárkou, které se provedou před započetím cyklu, `podminka` má stejný význam jako u cyklu `while`, `iter` je seznam příkazů oddělených čárkou, které se provedou vždy po provedení těla cyklu. Konstrukce dělá následující: provede příkazy v `init`, poté otestuje pravdivost vyrazu `podminka`. Pokud je pravdivý provede `telo cyklu`, a poté všechny přikazy v `iter`. V opačném případě cyklus končí. Po provedení těla cyklu a příkazů v `iter` přechází opět k testu pravdivosti výrazu `podminka`. Libovolné z `init`, `podminka`, `iter` lze vynechat, při vynechání `podminka` se vždy podmínka považuje za pravdivou. Viz následující diagram.



Typické použití `for` je s tzv. *krokovací proměnnou*. To je proměnná, kterou inicializujeme v části `init`, její hodnotu měníme (obvykle o konstantní hodnotu) v části `iter`, a obvykle vystupuje i v části `podminka`.

```
/* vypiseme cisla 0 az 9, j je krokovaci promenna */
int j;

for (j = 0; j < 10; j = j + 1) {
  printf("%i ", j);
}
```

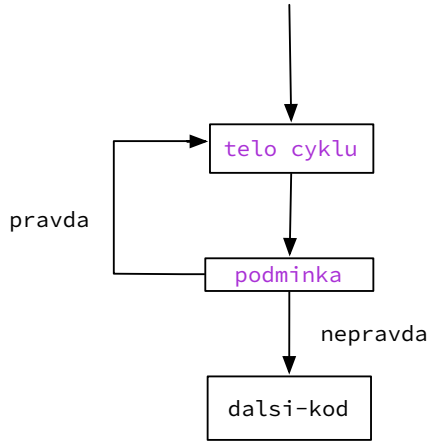
Pro lepší pochopení `for` si ukážeme, jak lze program přepsat s použitím `while`.

```
/* vypiseme cisla 0 az 9 */
int j;

j=0;
while(j < 10) {
  printf("%i ", j);
  j = j + 1;
}
```

Poslední konstrukcí pro cyklení je `do while`. Konstrukce je analogická cyklu `while`, pouze s tím rozdílem, že podmínku testujeme za tělem cyklu, nikoliv před tělem cyklu. To znamená, že u `do while` vždy proběhne tělo cyklu alespoň jednou. Syntax cyklu je následující

```
do
  telo cyklu
while (podminka)
```

Provádění cyklu je zachyceno na následujícím obrázku.



Příkaz `break` v těle cyklu tento cyklus okamžitě přeruší a program pokračuje vykonáváním kódu až za cyklem. Příkaz `continue` v těle cyklu přeruší vykonávání těla cyklu, program však cyklus neopustí, pouze skočí dopředu na konec těla. To znamená, že se přeskočí kód od `continue` do konce těla cyklu (u `for` to znamená, že `iter` se provede), jako v následujícím příkladu.

```
/* vypiseme cisla 0 az 9, vynechame nasobky 3 */
int j;
for (j = 0; j < 10; j = j + 1) {
  if(j % 3 == 0)
    continue;
  printf("%i ", j);
}
```

Pokud bychom na místo `continue` použili `break`, program by nevypsal nic. Číslo 0 je totiž násobkem 3 a cyklus by skončil už během prvního provádění těla cyklu.

Úkoly

1. Napište program, který načte celá čísla a a b a poté
   a) vypíše prvních a násobků čísla b,
   b) vypočítá a-tou mocninu čísla b,
   c) vypočítá a-té Fibonacciho číslo,
   d) určí, kolik číslic má číslo a.

e) sečte všechna celá čísla větší než a a menší než b.

2. Jak vytvoříte cyklus, který nikdy neskončí?
3. Napište program, který spočítá součet všech dvouciferných lichých čísel.
4. Vypište všechna čtyřciferná čísla, jejich součet číslic je dělitelný 7.
5. Načtěte čísla a a b. Vypište všechna čísla větší než a a menší než b, která se čtou stejne zezadu jako zepředu (např. 12321, 1001).
6. Napište program, který aproximuje hodnotu čísla π pomocí Gregory-Leibnitzovi aproximace jako součet prvních 100 členů sumy

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}.$$

7. Napište program, který pro přirozené číslo vypíše jeho rozklad na prvočísla.
8. Napište program, který pro číslo n vypíše trojúhelník podle vzorů v následujících příkladech.

a)
```
/*pro n = 3*/
1
12
123

/*pro n = 4*/
1
12
123
1234
```

b)
```
/*pro n = 3*/
1
2 3
4 5 6

/*pro n = 4*/
1
2 3
4 5 6
7 8 9 10
```

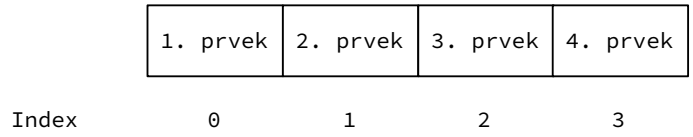
c)
```
/*pro n = 3*/
  *
 * *
* * *

/*pro n = 4*/
   *
```

```
  * *
 * * *
* * * *
```

4 Pole

Pole je kolekce za sebou uspořádaných hodnot stejného typu, které jsou i v paměti uloženy za sebou. K poli budeme přistupovat pomocí proměnné. Vztah mezi polem a proměnnou není stejný jako jako mezi proměnnou a číselnou hodnotou, je komplikovanější a úplně jej popíšeme až v příštím semestru. Nicméně *dočasně* si můžeme představit, že pole je *hodnota* uložená v dané proměnné.

Pomocí proměnné nebudeme (prozatím) přistupovat k celému poli, ale pouze k jeho jednotlivým prvkům. Učiníme tak pomocí proměnné, ve které je pole uloženo, a indexu daného prvku. Indexem myslíme pořadové číslo prvku v kolekci, přičemž s číslováním začneme od nuly: první prvek je na indexu 0, druhý prvek na indexu 1 atd.



Pole definujeme (tedy proměnnou „obsahující“ pole) podobně jako jednoduché proměnné, navíc však musíme určit kolik prvků bude pole obsahovat. Tomuto číslu budeme říkat *velikost* pole. Definice pole bez inicializace vypadá následovně:

```
typ jmeno[velikost];
```

`typ` a `jmeno` mají stejný význam jako u definice proměnné (určují typ prvků a jméno proměnné), `velikost` je kladná celočíselná konstanta (nemůže to být proměnná ani jiný výraz!) udávající počet prvků v poli. Pokud chceme pole inicializovat, přidáme na pravou stranu seznam hodnot ve složených závorkách.

```
typ jmeno[velikost] = {p1, p2, ..., pn};
```

Prvních n prvků pole nabude postupně hodnot `p1` až `pn`. Pokud chceme, aby byla velikost pole rovna počtu prvků ve výčtu, lze `velikost` vynechat.

```
/* neinicializovane pole typu int o velikosti 6 */
int foo[6];

/* pole typu float o velikosti 10 se 3 inicializovanymi prvky */
float bar[10] = {1.0f, 2.0f, 3.5f};

/* pole typu int o velikosti 4 se vsemi inicializovanymi prvky */
int baz[] = {1,10,-3,0};
```

K prvku pole přistujeme uvedením jména pole a za ně do hranatých závorek indexu daného prvku

```
jmeno[index]
```

`index` musí být celočíselná hodnota větší nebo rovna nule. Jak jsme již uvedli, prvky pole jsou indexovány právě od nuly, proto má n-tý prvek v poli index $n - 1$. K výrazu `jmeno[index]` lze přistupovat jako k proměnné, prvku v poli lze přiřadit novou hodnotu a také lze hodnotu prvku v poli využívat v jiných výrazech, viz následující příklad.

```
/* program vytvori pole nasobku cisla 3
   a pote vypise jeho obsah*/
int i;
int pole[20];

/* prvek pole[i] bude roven 3*i */
for(i=0; i<20; i+=1)
  pole[i] = 3 * i;

/* vypisu prvky pole*/
for(i=0; i<20; i+=1)
  printf("%i ", pole[i]);
```

V předchozím příkladě si můžeme všimnout, že pokud chceme postupně pracovat se všemi prvky pole, je výhodné použít cyklus `for`, jehož řídící proměnná vystupuje v roli indexu. Podotkněme také, že v příkladu není index nikdy není větší než 19. To je obecný princip. Přístup k prvku pole, který se v poli nevyskytuje, protože pole má méně prvků (index je větší nebo roven velikosti pole) je chybou, která se neprojeví jako chyba při překladu, ale při běhu programu může vést k jeho zhroucení nebo nechtěnému chování!

```
int foo[5] = {1,2,3,4,5};

foo[3] = 15;              /* OK */
foo[4] = foo[1] + foo[2]; /* OK */

foo[1] = foo[5];          /* SPATNE */
foo[7] = 10;              /* SPATNE */
```

Nakonec kapitoly si ukážeme ještě jeden příklad. Předpokládejme, že máme pole a chceme zjistit, jestli se v něm nachází (a případně na jakém indexu) předem daná hodnota.

```
int p[10];
int hledany;
int index;
int i;

/* tady pole naplnime cisly a priradime hodnotu promenne hledany */

/* v následucím kódu do promenne index vlozime index, na kterem se nachazi hledany,
   pripadne tam vlozime -1, pokud se prvek nenajde */

index = -1;

for(i=0; i<10; i+=1)
  if(p[i] == hledany) {
    index = i;
    break;
  }
```

Pomocí cyklu `for` procházíme prvky pole. Pro každý z nich kontrolujeme, jestli se rovná `hledany`. Pokud ano, přiřadíme `i` (tj. index aktuálně porovnávaného prvku) do `index` a cyklus ukončíme pomocí `break`. Pokud při průchodu pole nenarazíme na shodu, zůstane v proměnné `index` hodnota -1, kterou jsme tam vložili před cyklem.

Úkoly

1. Napište program, který vypočte průměr pole desetinných čísel.
2. Upravte program pro nalezení prvku v poli (řešený příklad nahoře), pokud můžeme předpokládat, že prvky jsou v poli setřízeny od nejmenšího.
3. Napište program, který rozhodne, jestli jsou dvě pole stejná (tj. mají na všech indexech stejné prvky). Můžete předpokládat, že pole mají stejný počet prvků.
4. Upravte program z předchozího příkladu tak, aby fungoval i v případě, že pole sice obsahují stejné prvky, ale mohou být v obou polích na různých indexech. Pozor, pole může obsahovat nějakou hodnotu vícekrát!
5. Napište program, který otočí pořadí prvků v poli. Tj. po provedení programu bude původně první prvek poslední, puvodně druhý prvek předposlední atd.
6. Napište program, který sečte všechna lichá čísla z pole.
7. Napište program, který ověří, zda-li jsou prvky v poli uspořádány vzestupně, sestupně či vůbec.
8. Napište program, který v poli nahradí všechny výskyty hodnoty `a` hodnotou `b` (kde `a` a `b` jsou hodnoty vhodného typu).

4.1 Textové řetězce

Už víme, že typu `char` lze využít k reprezentaci znaků abecedy podle ASCII tabulky. Textový řetezec, který chápeme jako sekvenci jednotlivých znaků, je pak přirozeně reprezentován polem typu `char`, jehož jednotlivé prvky jsou znaky řetězce. Řetězec je v poli ukončen prvkem s hodnotou 0 (pozor, ne znak `'0'`). Délku řetězce lze tedy spočítat jako počet nenulových prvků pole vyskytujících před prvkem s hodnotou 0.

```
   /* predpokladame, ze r je retezec, tedy pole typu
      char obsahujici prvek 0 */

   int delka=0;

   /* retezec je ukoncen prvkem rovnym 0, tedy false */
   while(r[delka])
     delka += 1;
```

Pole obsahující řetězec musí mít nejméně tolik prvků, kolik má řetězec znaků plus jeden prvek pro na nulu na konci. Nic však nebrání tomu, aby mělo prvků více. Textové řetezce lze inicializovat pomocí řetězové konstanty.

```
/* priklad inicializace retezu */
char foo[] = "Ahoj svete!";
```

Úkoly

1. Napište program, který převede všechny malé znaky v řetězci na velké.
2. Napište program, který určí maximální počet znaků, na kterých je konec řetězce `r1` stejný jako začátek řetězce `r2`. Například pro řetězce `ahoj` a `hojnost` jsou to tři znaky (shoda je na řetězci `hoj`), pro řetězce `ahoj` a `baba` je nula znaků.
3. Napište program, který určí, zda-li je řetězec palindromem (čte se stejně od konce jako od začátku).

5 Funkce

I nejjednoduší program v C obsahuje funkci `main`. Zdrojový kód programu obvykle obsahuje definice mnoha funkcí, které se mezi sebou volají. Pripomeňme, že funkci si představujeme jako miniprogram, kterému předáme nějaká vstupní data, on s nimi něco provede (něco spočítá, něco změní, vypíše na obrazovku apod.) a vrátí výsledek.

Programátor má možnost definovat vlastní funkce, nyní si řekneme jak. Definice funkce se skládá z hlavičky funkce a z těla funkce. Hlavička funkce má tvar

```
typ jmeno (typ1 a1, typ2 a2, ...,typn an)
```

V hlavičce určíme vše potřebné pro pohled na funkci zvnějšku. Pomocí `typ` určíme typ *návratové hodnoty*, tedy hodnoty, kterou funkce vrací jako svůj výsledek. Je možné vytvořit funkci, která žádnou návratovou hodnotu nemá (tj. nic nevrací), pak ke specifikaci typu použijeme klíčové slovo `void`. `jmeno` je jménem funkce, platí pro něj stejná pravidla jako pro jména proměnných. Za jménem funkce je v kulatých závorkách specifikace vstupních dat funkce. Jsou specifikována jako čárkami oddělený seznam dvojic typ a jméno, každé takové dvojici říkáme *argument* nebo *parametr*. K argumentům funkce přistupujeme v těle funkce tak, jako by to byly proměnné.

Tělo funkce je blok příkazů, tedy kus kódu ohraničený složenými závorkami, nacházející se pod hlavičkou funkce. Tento blok určuje, co funkce dělá. Pokud chceme z funkce vrátit návratovou hodnotu, uděláme tak příkazem

```
return hodnota;
```

Po provedení tohoto příkazu funkce okamžitě skončí a vrátí `hodnota`, přičemž `hodnota` může být libovolný výraz, jehož výsledkem je hodnota, například konstanta, výsledek aritmetických operací, výsledek zavolání funkce atd. Chceme-li se vrátit z funkce bez návratové hodnoty, výraz `hodnota` vynecháme. Pozor, pokud v hlavičce funkce specifikujeme typ návratové hodnoty jiný než `void`, musí funkce vždy nějakou hodnotu vracet! Uveďme si nyní několik příkladů definice funkce.

```
/* vypocet mocniny s prirozenym exponentem */
double mocnina(double zaklad, int exponent)
{
  double vysledek = 1.0;
  int i;

  for(i=0; i<exponent; i+=1)
    vysledek = vysledek * zaklad;

  return vysledek;
}

/* vypis prvnich n nasobku cisla m */
void vypis_nasobky(int m, int n)
{
```

```
    int i;

    for(i=1; i<=n; i+=1)
      printf("%i ", m * i);
  }
```

Funkce vždy definujme vně všech ostatních funkcí, tj. i vně funkce `main`!

Funkci voláme tak, že napíšeme její jméno a za ně do kulatých závorek seznam předaných hodnot argumentů, jejichž pořadí a počet odpovídá definici funkce. Před provedením těla funkce se na argumenty specifikované v hlavičce funkce navážou skutečné hodnoty předané při volání. Potom se provede tělo funkce. Pokud funkce vrací nějakou hodnotu, je výsledkem zavolání funkce její návratová hodnota (připomeňme, že ta je určena pomocí `return` v těle funkce). Zavolání funkce vždy způsobí provedení těla funkce, takže pokud je v těle funkce příkaz s vedlejším efektem (například výpis do konzole), tento vedlejší efekt při zavolání nastane.

Řekněme, že zavoláme funkci `mocnina` definovanou nahoře s parametry 2 a 3 a výsledek přiřadíme do proměnné `y`.

```
double y = mocnina(2,3);
```

Při provádění volání se nejdříve naváže hodnota 2 na argument `zaklad` a hodnota 3 na argument `exponent`. V těle funkce se tedy mužeme dívat na `zaklad` jako na proměnnou s hodnotou 2 a na `exponent` jako na proměnnou s hodnotou 3. Poté se provede tělo funkce a příkazem `return` vrátíme hodnot proměnné `vysledek`, tedy hodnotu 8. Tuto hodnotu poté přiřadíme (operátorem přiřazení) proměnné `y`. Další příklady volání funkce jsme už viděli mnohokrát, volali jsme funkce `printf` a `scanf`. Uveďme si přesto ještě příklad, ve kterém se pracuje i s návratovou hodnotou funkce.

```
/* vypocet mocniny s prirozenym exponentem */
  double mocnina(double zaklad, int exponent)
  {
    double vysledek = 1.0;
    int i;

    for(i=0; i<exponent; i+=1)
      vysledek = vysledek * zaklad;

    return vysledek;
  }

  /* suma prvnich n clenu posloupnosti 1/2^i */
  double suma(int n)
  {
    double vysledek = 0;
    int i;

    for(i=0; i<n; i+=1)
      vysledek += 1 / mocnina(2,i);

    return vysledek;
  }
```

```
int main()
{
  int m;
  printf("Kolik clenu posloupnosti secteme ");
  scanf("%i", &m);
  printf("Vysledek je %f\n", suma(m));
  return 0;
}
```

Program v příkladu počítá součet posloupnosti s prvky $1/2^i$ pro i od 0 do $n-1$. Tento součet počítá funkce `suma`. Pro výpočet mocniny 2^i v těle funkce `suma` voláme funkci `mocnina` a zpracujeme její návratovou hodnotu.

Předtím, než lze funkci volat, musí o ní program „vědět“. To znamená, že musí být před voláním buď definována nebo *deklarována*. Funkci deklarujeme zapsáním její hlavičky ukončené středníkem, přičemž jména argumentů lze vynechat, stačí pouze jejich typy. Tělo funkce vynecháme. Deklarací funkce sdělujeme, že funkce s danou hlavičkou existuje. Někde ovšem musí existovat i její definice, jinak by funkce po zavolání nemohla proběhnout. Umístění definice je složitější problém související s rozdělením kódu do více souborů, pro teď se spokojíme s umístěním definice kamkoliv za deklaraci. Funkci `mocnina` bychom deklarovali následovně.

```
double mocnina(double zaklad, double exponent);
```

Deklarace funkce se hodí například v situaci, kdy se funkce vzájemně volají v kruhu. Řekněme, že funkce A volá funkci B, funkce B volá funkci C a funkce C volá funkci A. Minimálně jednu z funkcí A,B,C musíme deklarovat před tím, než funkce definujeme, protože jinak bychom se vždy dostali do situace, ve které voláme nedefinovanou funkci. Můžeme například deklarovat funkci A a poté postupně definovat C, B a A.

5.1 Rozsah platnosti proměnných

Platnost proměnné určuje kus kódu, kde tato proměnná existuje. To znamená, že s ní můžeme pracovat (přiřazovat jí hodnotu a číst její hodnotu). Proměnné jsou platné od své definice[1] do konce bloku (= kus kódu mezi množinovými závorkami), ve kterém byli definovány. Pokud se v daném bloku pokusíme použít proměnnou, která v něm nebyla definována, je hledána v bloku přímo nadřazeném, to znamená bloku, který obsahuje náš blok. Pokud není nalezena ani tam, tak se situace opakuje, proměnnou hledáme ve stále nadřazenějším bloku. Nejvyšším blokem, který je nadřazen všem blokům, je samotný soubor (tj. prostor mimo definice funkcí). Není-li proměnná nalezena ani v nejvyšším bloku, nastane chyba překladu. Proměnným definovaným v nejvyšším bloku říkáme *globální proměnné* (jsou platné globálně v celém souboru od místa své definice). Proměnné definované v jiný blocích nazýváme *lokální proměnné* (jsou platné *jenom* ve svém lokálním bloku).

Výše popsaný mechanismus má několik důsledků. V programu mohou existovat dvě různé proměnné stejného jména. V následující situaci

```
int alpha()
{
```

[1] nebo deklarace. O deklaracích proměnných si ale řekneme až v druhém semestru. Totéž platí pro všechny ostatní definice proměnných v této kapitole.

```
   int vysledek = 0;
   /* fce neco pocita */
}

int beta()
{
   int vysledek = 1;
   /* fce neco pocita*/
}
```

spolu proměnné `vysledek` z funkcí `alpha` a `beta` vůbec nesouvisí, protože jsou v různých blocích (tělech funkcí) a žádný z bloků není vnořen do druhého z nich. Následující příklad je komplikovanější.

```
int foo = 5; /* globalni promenna */

void alpha()
{
  int foo = 6;  /* lokalni promenna prekryvajici globalni promennou */
  printf("%i ", foo);
}

void beta()
{
  foo = 7;  /* pristup ke globalni promenne */
  printf("%i ",foo);
}

int main()
{
   printf("%i ",foo);
   alpha();
   beta();
   printf("%i ",foo);
   return 0;
}
```

Pokud program spustíme, vypíše postupně čísla 5, 6, 7 a 7. Při prvním zavolání funkce `printf` ve funkci main `main` přistupujeme ke globální proměnné `foo` (v těle `main` není `foo` definována a tak je hledána v nadřazeném bloku). Vypíše se tedy hodnota 5. Ve funkci `alpha` definujeme lokální proměnnou `foo`, která překryje globální proměnnou `foo` (jsou to dvě různé proměnné se stejným jménem). Při výpisu hodnoty `foo` je nalezena lokální proměnná a je použita její hodnota (program vypíše 6). Od definice `foo` v těle `alpha` až do jejího konce nemáme ke globální proměnné vůbec přístup. Ve funkci `beta` opět pracujeme s globální proměnnou a změníme její hodnotu na 7, program proto potom vypíše dvakrát 7.

Globální proměnné bychom měli používat s rozvahou a pouze v případech, kdy je jejich použití nutné. Příliš liberální používaní globálních proměnných může vést k *špagety kódu*. To je program, kde jsou všechny části kódu spolu velmi těsně provázány (například pomocí globálních proměnných). Takový kód je velmi těžko srozumitelný (často i pro autora). Programovat bychom měli tak, aby implementace jednotlivých funkcí v programu (tj. kód v jejich tělech) na sobě byly co nejvíce nezávislé.

5.2 Předávání parametrů hodnotou

Co se stane, když ve funkci změníme hodnoty argumentů? Bude v následujícím příkladu hodnota proměnné a po zavolání funkce `zmen_na_5` 4 nebo 5?

```
void zmen_na_5(int cislo)
{
  cislo = 5;
}

int main()
{
  int a = 4;
  zmen_na_5(a);
  /* jaka je hodnota a? Je to 4 nebo 5? */
}
```

Správná odpověd je, že hodnota a po zavolání `zmen_na_5` je 4. Proc?

Parametry se funkcím *předávají hodnotou*. Při volání funkce se předané hodnoty argumentů překopírují na jiné místo v paměti (toto místo odpovídá argumentům funkce, které můžeme chápat jako lokální proměnné dané funkce) a případná změna argumentů v těle funkce pak proběhne na tomto místě, nikoliv na původním místě (v našem příkladu odpovídajícím proměnné a).

To neznamená, že v C nejde zařídit, aby funkce měnit hodnoty svých argumentů mohla. Lze to, ale řekneme si to až v druhém semestru.. Pro teď se spokojíme se speciálním případem tohoto mechanismu: *hodnoty prvků pole, které předáme funkci jako argument při jejím volání, může tato funkce měnit.* Viz následující příklad. Fakt, že pole může být argumentem funkce, je také nová věc, proto si v hlavičce funkce všimněte i způsobu, jakým se takový argument zapisuje.

```
void vynuluj_pole(int p[], int velikost)
{
  int i;
  for(i=0; i<velikost; i+=1)
    pole[i] = 0;
}

int main()
{
  int foo[5] = {1,2,3,4,5};
  vynuluj_pole(foo, 5);

  /* vsechny prvky pole jsou rovny 0 */
}
```

Úkoly

1. Přepište úlohy z minulých kapitol za pomoci funkcí. To znamená: naprogramujte funkci, která dělá požadovanou věc, a poté ji zavolejte z funkce `main`. Musíte vhodně zvolit argumenty a návratovou hodnotu funkcí.

2. Napište program, který nalezne všechna čtyřciferná čísla, která jsou dělitelná číslem, které dostaneme jako sumu čísla tvořeného první a druhou číslicí a čísla tvořeného třetí a čtvrtou číslicí. Např. číslo 3417 je dělitelné číslem 34 + 17. Vhodné části algoritmu realizujte pomocí funkcí.
3. Napište program, který pro dané přirozené číslo spočítá jeho rozdíl od nejbližšího většího prvočísla.
4. Napište program, který pro dané přirozené číslo n vypíše všechna prvočísla menší než n, jejichž součet číslic je také prvočíslem.
5. Napište program, který pro dané přirozené n spočítá sumu

$$1! + (1 + 2)! + (1 + 2 + 3)! + \cdots + (1 + 2 + 3 + \cdots + n)!$$

(Testujte pouze pro malá n, výsledek může být obrovské číslo a může dojít k přetečení).

6 Struktury

Pokud v programu potřebujeme pracovat se složenými objekty, tedy například potřebujeme nějak reprezentovat body v třírozměrném prostoru (k tomu potřebujeme tři souřadnice) nebo chceme naprogramovat program pro správu hudebních skladeb (a k jedné skladbě se váže její jméno, jméno interpreta, délka, rok nahrávky atd.), můžeme ke sdružení všech potřebných informací do jednoho celku použít struktury.

Struktura umožnuje spojit více hodnot, které mohou být různých typů, dohromady pod jediné jméno. Například, pokud chceme pracovat v programu s časem měřeným v hodinách, minutách a sekundách, můžeme pro to definovat následující strukturu.

```
struct cas {
  unsigned int hodin;
  unsigned int minut;
  float sekund;
};
```

Touto definicí zavedeme do programu nový typ se jménem `struct cas`. S tímto typem můžeme dále pracovat jako je obvyklé: lze definovat proměnné tohoto typu, funkce mohou brát strukturu jako argument, nebo ji vracet jako návratovou hodnotu atd.

Obecný předpis pro definici struktury vypadá následovně.

```
struct jmeno {
  polozka1
  polozka2
    ...
  polozkan
  } seznam promennych;
```

`polozka1` až `polozkan` jsou zápisy ve formě podobné definici proměnné, jako jsme viděli v předchozím příkladu. Zopakujme, že položky mohou být různých typů. Na místo `seznam promennych` lze uvést čárkami oddělenou sekvenci jmen proměnných s případnou inicializací (o té dále). Tyto proměnné jsou právě definovaného typu. `seznam promennych` lze vynechat (nezapomínejme, že proměnné typu právě definované struktury jdou později definovat obvyklým způsobem). Tedy například definicí

```
struct cas {
  unsigned int hodin;
  unsigned int minut;
  float sekund;
  } z1, z2, z3;
```

zavedeme proměnné `z1`, `z2` a `z3` typu `struct cas` (a samozřejmě zavedeme i tento typ). Lze vynechat i `jmeno` a pracovat pouze s proměnnými definovanými v části `seznam-promenných`. Samotný zápis `struct ...` je totiž zápisem typu, `jmeno` můžeme při dalším použití v programu chápat jako zkratku za tento zápis.

Při definici proměnné struktury lze jednotlivé položky inicializovat výčtem, podobně jako je tomu u polí.

```
/* do polozky hodin vlozime 10,
   do polozky minut vlozime 13,
   do polozky sekund vlozime 12.36 */
struct cas foo = { 10, 13, 12.36 };
```

Hodnoty v závorkách uvádíme v pořadí, v jakém jsme položky deklarovali. K jednotlivým položkám struktury lze přistupovat pomocí operátoru tečka

```
promenna.polozka
```

přičemž tento výraz můžeme chápat jako proměnnou (a provádět s ním operace proměnné příslušející: číst jeho hodnotu a přiřazovat mu hodnotu). Tedy například

```
struct cas bar;
bar.hodin = 10;
bar.minut = foo.minut + 5;
bar.sekund = 1.13;
```

Struktury lze přiřazovat. Při přiřazení se kopírují hodnoty sobě odpovídajících položek. Například provedení příkazu

```
foo = bar;
```

odpovídá sekvenci příkazů

```
foo.hodin = bar.hodin;
foo.minut = bar.minut;
foo.sekund = bar.sekund;
```

Odtud plyne i to, že struktury jsou funkcím předávány hodnotou (a že lze z funkce vrátit hodnotu lokální struktury). Všechny ostatní operace se strukturami musí programátor implementovat sám jako funkce. Práci se strukturami demonstrujeme v následujícím příkladu. Všimněte si, jak pracujeme se strukturami jako s argumenty a návratovými hodnotami funkcí, a inicializace pole struktur ve funkci `main`.

```
#include <stdio.h>

struct cas {
  unsigned int hodin;
  unsigned int minut;
  float sekund;
};

/* vrati cas v kanonickem tvaru */
struct cas kanonicky_tvar(struct cas t)
{
  struct cas r;
  int foo;

  foo = t.sekund / 60;
```

```
  r.sekund = t.sekund - (foo * 60);
  r.minut = (foo + t.minut) % 60;
  r.hodin = t.hodin + ((foo + t.minut) / 60);

  return r;
}

/* vrati -1 pokud je a kratsi nez b,
          0 pokud jsou casy stejne dlouhe,
          1 pokud je b kratsi */
int porovnej(struct cas a, struct cas b)
{
  struct cas ak = kanonicky_tvar(a);
  struct cas bk = kanonicky_tvar(b);

  /* v nasledujicim vyuzivame toho, ze (x > y) je
     rovno 0, pokud neplati, a 1, pokud plati */
  if(ak.hodin != bk.hodin)
      return -1 + 2 * (ak.hodin > bk.hodin);

  if(ak.minut != bk.minut)
      return -1 + 2 * (ak.minut > bk.minut);

  if(ak.sekund != bk.sekund)
      return -1 + 2 * (ak.sekund > bk.sekund);

  return 0;
}

int main()
{
  /* pri inicializaci pole struktur uvedeme za sebou
     inicializaci struktury na prvni pozici, pote na
     druhe pozici atd.  */
  struct cas pole[3] = { 1, 120, 2.3,
                         2, 4, 117.1,
                         3, 5, 10.3   };

  /* odhadnete, co program vypise*/
  printf("%i\n", porovnej(pole[0], pole[2]));

  return 0;
}
```

6.1 Struktura obsahující jinou strukturu

Struktura může jako položku obsahovat další strukturu. Toho dosáhneme jednoduše deklarováním takové položky ve struktuře. Například bychom mohli vytvořit strukturu pro účastníka závodu.

```
struct zavodnik {
  char jmeno[30];
```

```
  int cislo;
  struct cas vysledek;
};
```

K položkám vnitřní struktury lze stále přistupovat pomocí operátoru tečky. Máme-li proměnnou `z` struktury `zavodnik`, můžeme se dostat k položce `minut` pomocí dvojího použití operátoru tečka. První použití k přístupu k položce `vysledek`, druhé použití k přístupu k položce `minut`. Celý zápis tedy je `z.vysledek.minut`. Nemusíme použít závorky, protože operátor tečky se postupně vyhodnocuje zleva.

Položky struktury jsou v paměti uloženy za sebou (s potenciálními prázdnými místy mezi položkami, více dále). Pokud struktura A obsahuje jako položku strukturu B, jsou položky struktury B v paměti uloženy na místo odpovídající dané položce ve struktuře A. Například u struktury `zavodnik` jsou za sebou uloženy položky `jmeno`, `cislo`, `vysledek.hodin`, `vysledek.minut`, `vysledek.sekund`. Velikost struktury lze zjistit pomocí operátoru `sizeof`. Pozor, tato velikost se nemusí rovnat součtu velikostí jednotlivých položek, překladač se může rozhodnout velikost dané struktury zvětšit. (Může přidat volné místo mezi/za položky, aby byla struktura v paměti zarovnaná. Například může chtít, aby položky začínaly na adrese dělitelné čtyřmi.) Z předchozího plyne, že struktura nemůže obsahovat jako položku samu sebe. Například následující definice struktury je špatně.

```
struct foo {
  int i;
  struct foo dalsi; /* ilegalni polozka !!*/
};
```

Při pomyslném ukládání položky do paměti bychom se totiž nekonečně zanořovali do položky `dalsi` (z tohoto pohledu by struktura měla nekonečnou velikost).

6.2 Pojmenování typů

V jazyce C existuje nástroj na vytváření nových jmen (existujících) typů nazvaný `typedef`. Pokud bychom například chtěli vytvořit nové jméno pro typ `int`, můžeme to učinit následovně.

```
typedef int CisloPopisne;
```

Slovo `CisloPopisne` je nyní synonymem pro slovo `int` (které pořád zůstává platným klíčovým slovem) a lze jej v jazyce používat na všech místech, kde lze použít `int`. Nová jména jde ovšem vytvářet i pro komplikovanější typy. Například pomocí

```
typedef float Male_pole[3];
```

vytvoříme jméno `Male_pole` pro tříprvkové pole typu `float`. Tato nová jména můžeme využívat na místech, kde bychom používali původní zápisy daných typů, tedy například

```
Cislo i;
Male_pole p = { 1.3, 2.1, 1.4 };
```

Deklarace nového jména typu má vždy formu

```
typedef deklarace promenne;
```

a nové jméno typu se vždy objevuje na místě, kde se v deklaraci proměnné objevuje jméno proměnné. `typedef` lze s výhodou využít pro vytvoření jména pro strukturu a vyhnout se tak nutnosti psát slovo `struct`.

```
typedef struct {
  int data;
  int dalsi_data;
} foo;
```

Předchozím zápisem jsme vytvořili bezejmennou strukturu a pojmenovali jsme ji `foo`.

Nová jména typů vytváříme kvůli odstínění konkrétní vnitřní reprezentace objektů (například na jednom systému bychom chtěli, aby `CisloPopisne` bylo `int` a na jiném `long`, stačí pak změnit řádek, kde vytváříme nové jméno). Vytváření vhodných jmen pro typy také zlepšuje srozumitelnost a čitelnost kódu.

Úkoly

1. K předchozímu programu doprogramujte funkci pro výpočet rozdílu mezi dvěma časy. Funkce musí vracet také strukturu.
2. Definujte strukturu pro reprezentaci zlomku. Naprogramujte funkce pro aritmetické operace se zlomky a pro převod zlomku do základního tvaru.
3. Definujte strukturu pro reprezentaci bodu na ploše. Naprogramujte funkci, která pro pole bodů vrátí nejkratší vzdálenost mezi mezi body z pole.
4. Definujte strukturu pro datovou strukturu zásobník pomocí pole s kapacitou 100 prvků (vhodné položky vymyslete sami). Implementujte funkce pro vložení prvku na zásobník, odebrání prvku ze zásobníku, test toho, jestli je zásobník prázdný, a test toho, jestli je zásobník plný.
5. Proveďte totéž, co v předchozí úloze, tentokrát pro datovou strukturu fronta.
6. Definujte strukturu pro obdélník rovnoběžný s osami (vhodné položky vymyslete sami). Naprogramujte:
   a) funkci pro výpočet obsahu a obvodu obdélníku
   b) funkci, která určí, zda se dva obdélníky protínají
   c) funkci, která určí, zda je jeden obdélník uvnitř druhého obdélníku.

7 Hledání chyb v programu

7.1 Syntaktické chyby

Syntaktická chyba vznikne tehdy, pokud zdrojový kód neodpovídá syntaxi jazyka (formálním pravidlům říkajícím, jak se zdrojový kód zapisuje). Taková chyba je nalezena už při překladu programu. Překladač nevytvoří spustitelný soubor, ale vypíše seznam chyb. Každá z nich je popsána a často je uveden i řádek, na kterém se chyba vyskytuje. Mezi typické syntaktické chyby patří: zapomenutý středník, chybějící nebo přebývající závorka, překlep ve jménu proměnné, funkce nebo typu, atd. Syntaktickou chybu je obvykle velmi jednoduché najít a odstranit.

7.2 Chyby zjištěné za běhu programu

To, že se podařilo program přeložit, a lze jej spustit, není zárukou, že program funguje správně. Většina chyb se totiž projeví pouze za běhu programu. Důsledkem takové chyby může být havárie programu, případně jeho nechtěné či nepředpokládané chování. Chyby zjištěné za běhu programu můžeme zhruba rozdělit do dvou kategorií.

1. Chyby způsobené nesprávným použitím jazyka. V jazyce C mezi takové chyby mimo jiné patří: přístup k prvku pole pomocí nesprávného indexu (např. index větší než velikost pole), dělení nulou apod. Hodně chyb vzniká kvůli chybné práci s pamětí, například důsledkem neuvolňování paměti je tzv. leakování paměti a program zabírá víc paměti než potřebuje, díky přístupu k jiné části paměti, než zamýšlíme, může program dělat něco jiného než chceme apod. Práci s pamětí se budeme věnovat až v druhém semestru kurzu.

2. Chyba není způsobena tím, jak je kód napsán (není to chyba z předchozího bodu), ale přesto program nedělá to, co bychom chtěli. Taková chyba je způsobena buď tím, že jsme algoritmus špatně implementovali (tj. algoritmus je správně, jenom jsme ho špatně převedli na program), nebo je chyba v samotném algoritmu.

Prvním krokem k odstranění chyby je její nalezení a zjištění proč chyba vznikla. Proto je nutné zjistit, co se v programu za jeho běhu děje. Jaké informace potřebujeme k tomu, abychom řekli, že víme co v programu děje? Musíme vědět, který příkaz program vykonal naposled (na kterém řádku kódu se při vykonávání programu nacházíme), a obsah paměti, tedy hodnoty všech proměnných, polí, hodnoty argumentů předaných funkcím při volání, jejich návratové hodnoty, atd.

Co se v programu děje můžeme zjistit buď analýzou kódu, nebo pozorováním programu za běhu. Při analýze kódu jej opakovaně čteme a přemýšlíme nad tím, jak funguje. Odpovídáme si na otázky následujícího typu: Co stane při provedení tohoto příkazu (např. hodnoty, kterých proměnných se změní, a jak)? Jaká je návratová hodnota dané funkce pro dané hodnoty argumentů? Proč má proměnná jinou hodnotu, než by měla mít? V podstatě program, nebo jeho části, provádíme v hlavě (nebo na papír). S přibývajícími zkušenostmi a s tím, jak

se budete v programování zlepšovat, budete analýzou kódu schopni najít a odstranit čím dál tím větší část chyb. Je to přirozené, budete totiž programům, které píšete, rozumět víc a víc.

Jednoduchou formou zjištění toho, co program dělá za běhu, je vypisování hodnot proměnných, které nás zajímají, na vhodných místech v program pomocí `printf`. Program spustíme a pak se podíváme na to, co vypsal. Pokud bychom například chtěli zjistit, co přesně dělá program pro výpočet nejmenšího společného dělitele, mohli bychom jej upravit následovně.

```
#include <stdio.h>

int main()
{
  int a,b,r;
  printf("Zadejte dvojici cisel: ");
  scanf("%i %i", &a, &b);

  printf("pocitam gcd pro %i a %i\n", a, b);
  while(b != 0) {
    r = a % b;
    a = b;
    b = r;
    printf("a: %i, b: %i\n", a, b);
  }

  printf("Vysledek je %i\n", a);
}
```

Po spuštění a zadání čísel 128 a 78 program vypíše:

```
pocitam gcd pro 128 a 76
a: 76 b: 52
a: 52 b: 24
a: 24 b: 4
a: 4 b:0
Vysledek je 4
```

Vidíme hodnoty proměnných `a` a `b` před započetím cyklu `while` a na konci každé jeho iterace. Vypisování jako formu analýzy programu používá mnoho zkušených a úspěšných programátorů, viz například kniha [1]. Jistá forma vypisování je přítomná i v hotových programech, když program vypisuje do konzole nebo do logovacího souboru informace o tom, co zrovna dělá.

Další možností jak zkoumat program během jeho běhu je použití debuggeru, který umožňuje tzv. *krokování*. Debugger umí na programátorem zvoleném místě (řádku ve zdrojovém kódu) běh programu zastavit. Místu přerušení běhu se říká *breakpoint*. Potom lze na požádání provádět následující příkaz (řádek) v programu. V každém okamžiku, kdy je program zastavený, lze zjistit obsah libovolné části paměti, ke které má program v daném okamžiku přístup (tj. hodnoty proměnných, obsah polí apod), a seznam všech neukončených volání funkcí spolu s hodnotami jim předaných argumentů, tzv. *backtrace* nebo *stack trace*. Podrobněji v následujícím příkladu.

```
1 int alpha(int a)
2 {
```

```
3     int x = a+1;
4     return a;
5   }
6
7   int beta(int c)
8   {
9     return alpha(c + 5);
10  }
11
12  int gamma(int a)
13  {
14    return 42;
15  }
16
17  int main()
18  {
19    gamma(10);
20    beta(6);
21    return 0;
22  }
```

Pokud program zastavíme na řádku 3, pak backtrace programu bude vypadat podobně jako následující diagram.

```
main, arguments: none
  |
  beta, arguments: a 6
    |
    alpha, arguments: c 11
```

V diagramu můžeme vidět, že předtím než se program dostal k řádku 3, proběhlo zavolání funkce `main`, ve které došlo k zavolání funkce `beta`, z níž program volal funkci `alpha`. Všimněte si, že v diagramu není nic o funkci `gamma`. To je kvůli tomu, že volání této funkce už skončilo.

Pokud je dalším příkazem, který má program provést, volání funkce, nabízí debugger možnost provést celé toto volání a až potom program zastavit (anglicky *step over*) nebo skočit na první řádek těla volané funkce (anglicky *step in*). Pokud je program zastaven v těle funkce, lze v jednom kroku dokončit provádění těla této funkce a program zastavit až na řádku za jejím voláním (anglicky *step out*).

Způsob práce s konkrétním debuggerem nebudeme v textu probírat. Všechny věci, které jsme o debuggerech uvedli, umí většina z nich. Ovládání konkrétního debuggeru si ukážeme na semináři, případně je můžete najít v jeho manuálu.

Literatura

[1] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming.* Apress, 2009.

[2] T. Cormen, R. Leiserson, R. Rivest, Clifford Stein. *Introduction to algorithms.* MIT Press, 2009.

[3] K.W. Kernighan, Dennis M. Ritchie. *Jazyk C.* Computer Press, 2006.