Programátorské minimum

Daniel Hládek

Table of Contents

Chapter 1. Úvod

Jazyk C pre jednoducho uvažujúcich.

Začínajúcich programátorov často odradí veľké množstvo pojmov ktoré je nutné sa naučiť a technologických problémov, ktoré je potrebné prekonať. Naučiť sa programovať znamená prekonanie týchto počiatočných prekážok. Cieľom tejto príručky je priblížiť jazyk C ľuďom bez predchádzajúcej skúsenosti s programovaním a uľahčiť prekonanie počiatočných prekážok.

Naučiť sa programovať sa nedá inak ako vyskúšaním "na vlastnej koži". V sérii niekoľkých tutoriálov Vás príručka naučí základné programátorské postupy. Nestačí "iba" čítať, dokonalý zážitok dosiahnete len tak že príklady v tutoriáloch si prepíšete a preložíte sám.

Predpokladáme, že máte počítač s nainštalovaným prekladačom, textovým editorom a viete spustiť príkazový riadok. Odporúčame si nainštalovať operačný systém Linux - obsahuje všetko potrebné pre vytváranie programov v jazyku C.

Naučíte sa za 4 hodiny

Programátorské minimum:

1. Napísať a preložiť triviálny program.
2. Vypísať správu.
3. Vytvoriť a inicializovať premennú.
4. Načítať číslo zo štandardného vstupu.
5. Navrhnúť funkciu s jedným parametrom, ktorá vracia jednu hodnotu.
6. Zavolať vlastnú funkciu s jedným parametrom a uložiť návratovú hodnotu.
7. Vypísať správu s parametrom.

Príklad

```c
#include <stdio.h>

float mocnina(float);

int main() {
    printf("Mocninová kalkulačka\n");
    float vysledok = 0;
    printf("Výsledok je zatiaľ %f\n",vysledok);
    char vstup[10];
    printf("Zadaj hodnotu na max. 10 miest:");
    fgets(vstup,10,stdin);
    printf("Zadali ste %s\n",vstup);
    float parameter = 0;
    sscanf(vstup,"%f",&parameter);
    printf("Hodnota parametra je %f\n",parameter);
    vysledok = mocnina(parameter);
    printf("Výsledok je %f\n",vysledok);
    return 0;
}

float mocnina(float arg){
    float parameter = arg * arg;
    return parameter;
}
```

Chapter 2. Ahoj Svet

Cieľom tohto tutoriálu je naučiť vás vytvoriť triviálny program v jazyku C.

Naučíte sa

- Zdrojový kód je zápis algoritmu v programovacom jazyku.
- Programovací jazyk je súbor pravidiel pre zápis algoritmov.
- Spustiteľný kód získame spracovaním (prekladom) zdrojového kódu
- Spustiteľný kód je plnohodnotný program

2.1. Vytvorenie zdrojového textu

V textovom editore si otvorme súbor, ktorý môžeme nazvať `hello.c` a do neho napíšeme:

```
#include <stdio.h>

int main(){
    printf("Ahoj svet\n");
    return 0;
}
```

Textový súbor uložíme a prvý krôčik máme za sebou. Práve sme vytvorili svoj prvý program.

2.2. Preklad

Náš program, je síce správny, ale počítač bude mať ťažkosti s jeho vykonaním, pretože rozumie iba číslam. Zdrojový text v jazyku C je pre človeka čitateľný. nie je priamo vykonateľný. Našou ďalšou úlohou bude preklad zdrojového textu do tvaru, ktorý sa dá spustiť. Zo zdrojového textu sa vytvorí binárny súbor, ktorý je tvorený číslami inštrukcií. Binárny súbor už je vykonateľný. Prekladač funguje podobne ako stroj na spracovanie textu. Vstupom do prekladača je zdrojový text programu, ktorý sa spracuje a premení do vykonateľnej podoby. Požitie kompilátora na zdrojový text, ktorý sme vytvorili je jednoduché:

```
gcc hello.c -o hello
```

Ak sme pri prepise neurobili žiadnu chybu, tak by preklad mal prebehnúť bez problémov a vznikne spustiteľný súbor `hello`.

2.3. Chybové hlásenia

Ak sme predsa len nejakú chybu urobili, prekladač napíše chybovú hlášku a preklad neprebehne. Pozorne si hlásenie prečítajte a skúste opraviť chybu. Chybové hlásenie (skoro) vždy obsahuje číslo riadku a znak, kde sa chyba nachádza a podľa toho sa ku chybe vieme vrátiť. Ak je chybových

hlásení viac, ignorujeme všetky okrem prvého. Opravíme prvú chybu, preložíme program a sledujeme, či oprava pomohla. Ak nie, postup opakujeme dovtedy, pokiaľ nie je preklad úspešný.

2.4. Spustenie programu

Ak preklad prebehol správne, môžeme vyskúšať výsledok. Najprv overíme situáciu, či sa tam spustiteľný súbor naozaj nachádza:

```
ls
hello.c hello
```

`ls` je príkazom interpretera BASH, ktorý sa stará o vykonanie príkazov v príkazovom riadku. Jeho ekvivalent v systéme Widows je príkaz `dir`.

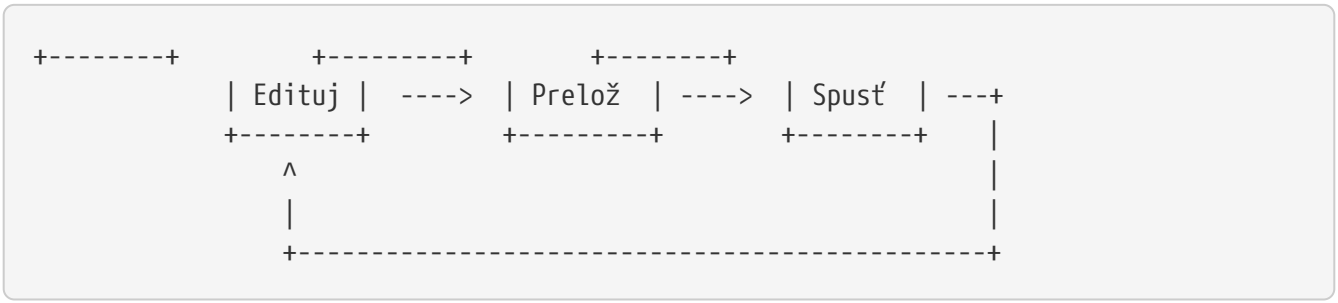
Príkaz `ls` by mal zobraziť zdrojový súbor aj spustiteľný súbor. Spustiteľný súbor môžeme spustiť:

```
./hello
Ahoj svet
```

2.5. Vývojový cyklus

Tento proces budeme opakovať stále dookola. V prvom kroku sme vytvorili alebo upravili existujúce zdrojové kódy. V druhom kroku sme sa pokúsili ich preložiť. Ak sa vyskytla chyba, tak sme sa museli vrátiť, znova upraviť zdrojové kódy a znova program preložiť. Posledným krokom vývojového cyklu je spustenie programu. Ak sme s výsledkom spokojní, nemusíme pokračovať a výsledný program môžeme odovzdať do používania. Ak nie, musíme začať znova.

Postup prác pri programovaní



2.6. Úprava programu

Ak spustenie prebehlo úspešne, môžem Vám gratulovať - práve ste sa stali (začínajúcim) programátorom. Náš prvý program je hotový.

Základy sú síce vybudované, ale výsledná stavba je malá. Zatiaľ vieme slušne pozdraviť, ale to je všetko. Skúsime využiť vedomosti ktoré máme aby sme výsledok trochu vylepšili. Okrem sveta skúsme pozdraviť aj niekoho iného.

Nájdime riadok v programe, ktorý pravdepodobne spôsobuje výpis správy na obrazovku a pozrime sa na neho bližšie:

```
①
printf("Ahoj svet\n")②
  ;③
```

Okrem správy "Ahoj svet" si všimneme niekoľko vecí, ktoré nám veľa povedia o jazyku C.

① Okrem správy, ktorá sa má vypísať tam vidíme `printf` a úvodzovky. Takýto zápis so zátvorkami nazývame **volanie funkcie**, Časť pred zátvorkou je **názov funkcie** a časť v zátvorke sú **argumenty funkcie** .

② Správa `"Ahoj svet"` je ohraničená úvodzovkami a zakončená znakom `'\n'`. Znaky ohraničené úvodzovkami nazývame **reťazec**. `'\n'` znamená nový riadok (line feed).

③ Riadok je zakončený bodkočiarkou. Takýto riadok nazývame **príkaz**.

Program v jazyku C sa skladá z definícií funkcií a ich volaní. Každú funkciu v jazyku C musíme pred volaním definovať, lebo jazyk C ako taký nepozná žiadne funkcie, iba spôsob ako ich definovať a volať.

V prípade funkcie `printf` je situácia jednoduchá, lebo funkciu už definoval niekto za nás. Nám už iba stačí "naučiť" kompilátor, čo znamená `printf` tým, že oznámime, kde definíciu funkcie nájde.

Na overenie týchto tvrdení použijeme tzv. *kreatívnu chirurgiu* a mierne zmrzačíme náš program. Upravíme niektorý riadok programu a zopakujeme proces prekladu. S vedeckým odstupom sledujeme čo sa stane:

```
#include <stdio.h> ③

int main(){
    printf("Ahoj svet\n"); ① ②
    return 0;
}
```

① Zmeňte `"svet\n"` na `"TUKE"` Zistili sme, že `"\n"` je zápis pre koniec riadku.

② Zmeňte `printf` na `print` Zistili sme, že takú funkciu systém nepozná. Opravte chybu a pokračujeme.

③ Vymažte riadok s `#include`. Zistili sme, že napriek očakávaniam kompilátor hlási chybu na riadku s `printf`.

Zoznam funkcií, ktoré môžeme pri práci použiť sa nazýva "Štandardná knižnica jazyka C".

2.7. Úlohy na precvičenie

Modifikujte program tak, aby do prvého riadka vypísal Vaše meno a do druhého riadka Vašu obľúbenú farbu napr.:

```
Moje meno je Chuck Norris
   a moja obľúbená farba je krvavo čierna.
```

Pozrite si dokumentáciu štandardnej knižnice a skúste využiť ľubovoľnú ďalšiu funkciu, napr. `sleep()`. Nezabudnite, že pred použitím musíte funkciu "naučiť" pomocou `#include`.

Chapter 3. Premeň ma! Premenné a dátové typy

V tomto bloku sa naučíme využívať pamäť počítača.

Naučíte sa

- Je možné uložiť hodnotu a tú neskôr použiť
- Každá hodnota má dátový typ.
- Dátové typy je možné meniť, má to ale vedľajšie efekty.
- Výpis premennej v rôznom formáte.
- Rovnaký typ je možné vypisovať rôznymi spôsobmi. — aj číslo s desatinnou čiarkou vieme zapisovať rôznym spôsobom.
- Čítať dokumentáciu funkcie `printf`.

3.1. Celočíselná premenná

V prvom kroku upravíme náš jednoduchý program:

```
#include <stdio.h>

int main(){
    int pocitadlo  = 7;
    printf("Ahoj svet\n",pocitadlo);
    return 0;
}
```

Pridali sme riadok, pomocou ktorého sme vytvorili pamäťové miesto s názvom `pocitadlo`. Číslo v pamäťovom mieste vieme podľa ľubovôle meniť, preto ho nazývame **premenná**.

Pozrime sa bližšie na príkaz pomocou ktorého vytvárame premennú:

```
int pocitadlo  = 7;
```

Prvá vec, ktorú si všimneme je kľúčové slovíčko `int`. Týmto slovíčkom označujeme **dátový typ** s ktorým pracujeme. Dátových typov je niekoľko, v tomto prípade pracujeme s celými číslami (integer). Celočíselný dátový typ nám umožňuje pracovať s celými číslami (ale nie s číslami s desatinnou čiarkou).

Druhým slovíčkom za dátovým typom je **názov premennej**. Názov premennej funguje podobne ako menovka na poštovej schránke alebo názov políčka vo formulári. Pomocou názvu premennej vykonávame všetky operácie.

Tretím slovíčkom na riadku je symbol `=`. V klasickom matematickom zápise to znamená znamienko

rovnosti. V zápise jazyka C je ale význam úplne iný a znamená operáciu **priradenia**. V tomto prípade sa do premennej s názvom `pocítadlo` priradí (zapíše) hodnota *7*.

3.2. Výpis premennej

To, že si vieme zapamätať nejaké číslo je síce skvelé, ale samo o sebe zbytočné. Aby boli premenné naozaj užitočné, potrebujeme s premennou vykonávať nejaké operácie. Pozrime sa bližšie na riadok s príkazom na výpis na obrazovku a modifikujeme ho tak, aby vypisoval aj hodnotu premennej:

```
printf("Počítadlo má hodnotu %d\n", pocitadlo);
```

V riadku s príkazom na výpis správy sme za čiarku pridali ďalší argument s menom premennej ktorú máme zahrnúť do výpisu. Znaky medzi úvodzovkami obsahujú správu ktorá sa má vypísať a znaky, ktoré zastupujú hodnotu premennej na výpis. Zástupné znaky vyjadrujú akým spôsobom sa má premenná vypísať, v tomto prípade `%d` znamená výpis vo forme celého čísla.

3.2.1. Úloha na vyriešenie

Preštudujte si možné formátovacie reťazce pre funkciu `printf` a vypíšte hodnotu premennej na minimálne zadaný počet miest, chýbajúce znaky doplňte nulami. Manuálovú stránku funkcie `printf` si zobrazíte príkazom `man printf` (v príkazovom riadku) alebo vyhľadáte na internete.

3.3. Inicializácia premennej

Skúste upraviť program nasledovným spôsobom (vynecháme operáciu priradenia):

Program so zle inicializovanou premennou

```
#include <stdio.h>

int main(){
    int pocitadlo;
    printf("Ahoj %d\n",pocitadlo);
    return 0;
}
```

Zistili sme, že na prvý pohľad síce program pracuje normálne, ale nie je jasné aká hodnota bude v premennej `pocítadlo`.

Zabudnutá inicializácia spôsobuje nepredvídateľné chovanie programu.

Keďže sme žiadnu hodnotu do premennej nepriradili a v ďalšom riadku túto neznámu hodnotu využívame, program sa správa nepredvídateľne. Ak do premennej žiadnu hodnotu nepriradíme tak jej hodnota závisí od toho, akú pamäť nám operačný systém pridelil. V premennej môže byť hodnota nula, ale aj akákoľvek iná hodnota ktorú nevieme predvídať. Na skutočnosť, že pracujeme s nedefinovanou hodnotou nás ale program nevie upozorniť a vyzerá to tak, že je všetko v poriadku.

Je zodpovednosťou programátora aby sa vyhol takýmto situáciám a využíval postupy a nástroje ktoré tomu zabránia.

```
#include <stdio.h>

int main(){
    int pocitadlo = 0; ①
    printf("Ahoj %d\n",pocitadlo);
    return 0;
}
```

① Táto premenná je inicializované správne a vieme že vždy bude nulová.

3.4. Dátový typ

Dátový typ definuje možnosti a obmedzenia premennej s ktorou pracujeme. Premenná funguje podobne ako obrazovka na kalkulačke. Obrazovka na kalkulačke má určité obmedzenia ktoré sa podobajú obmedzeniam dátového typu - na kalkulačke môžeme pracovať s číslami, ale nie s písmenami. Číslo na kalkulačke môžeme prečítať alebo ho zmeniť. Podľa typu kalkulačky s ním môžeme vykonávať sadu operácií - spočítavanie alebo násobenie.

Do teraz napísaný program nám umožňuje prácu iba s celočíselnými hodnotami. Do celočíselnej premennej nie je možné uložiť hodnotu s desatinnou čiarkou. Pokiaľ sa o to pokúsime, hodnota za desatinnou čiarkou sa zanedbá a dôjde k zaokrúhleniu smerom nadol. Na túto skutočnosť nás kompilátor nemusí upozorniť.

Modifikujme náš program:

Dochádza k automatickému zaokrúhľovaniu

```
#include <stdio.h>

int main(){
    int pocitadlo = 1.1;
    printf("Ahoj %d\n",pocitadlo);
    return 0;
}
```

Aká hodnota sa vypíše?

Zmeňme formát výpisu premennej na číslo s desatinnou čiarkou:

Dochádza k nesprávnemu formátu výpisu

```
#include <stdio.h>

int main(){
    float pocitadlo = 1.1;
    printf("Ahoj %d\n",pocitadlo);
    return 0;
}
```

Pozitívna zmena nenastala, lebo stále platia obmedzenia pre celočíselný dátový typ. Na to aby sme správne vedeli pracovať s číslom s desatinnu čiarkou, musíme použiť vhodný formátovací znak:

Správny typ premennej aj správny formát výpisu premennej.

```
#include <stdio.h>

int main(){
    float pocitadlo = 1.1;
    printf("Ahoj %f\n",pocitadlo); ①
    return 0;
}
```

① Všimnite si, že formátovacia značka `%d` (pre celé čísla) sa zmenila na `%f` (pre čísla s desatinnou čiarkou).

Dátový typ premennej a formát výpisu na obrazovku sú dve rozdielne veci. Aj keby sme zmenili formátovaciu značku na `%d`, obsah premennej `pocitadlo` to neovplyvní.

3.5. Operácie s dátovými typmi

Celočíselný dátový typ (`int`) je vhodný najmä ako počítadlo alebo na uchovávanie indexov. Nie je veľmi vhodný na matematické operácie ako napr. násobenie alebo delenie. Výsledkom delenia dvoch celých čísel je reálne číslo, a túto skutočnosť musíme brať do úvahy. Môžeme si to vyskúšať na jednoduchom matematickom programe, ktorý vykonáva operáciu delenia (`/`):

```
#include <stdio.h>

int main(){
    int vysledok = 5 / 2;
    printf("Ahoj %f\n",vysledok);
    return 0;
}
```

Keďže premenná `vysledok` má celočíselný dátový typ `int`, je jasné, že výsledok nebude správny. Navyše, nepomôže ani intuitívna oprava:

```
    float vysledok = 5 / 2;
    printf("Ahoj %f\n",vysledok);
```

Dôvodom je to, že podľa štandardu je výsledkom delenia dvoch celých čísel opäť celé číslo, takže automaticky dochádza k zaokrúhľovaniu. Tento proces, keď prekladač zmení dátový typ bez nášho dovolenia sa nazýva **implicitná dátová konverzia**. Do premennej, ktorá je vhodná na uloženia čísla s desatinnou čiarkou (typ `float`) sa uloží už zaokrúhlená hodnota. Prekladač jazyka C musíme presvedčiť, že delíme čísla s desatinnou čiarkou:

```
float vysledok = 5.0 / 2.0;
printf("Ahoj %f\n",vysledok);
```

Teraz to už funguje správne.

3.6. Typová konverzia

Matematické operácie vieme vykonávať aj s obsahom premennej, ktorý sme si zapamätali v predošlých krokoch. Stačí nahradiť číselné hodnoty v predošlom príklade názvami premenných, ktoré sme si inicializovali pred tým.

```
float podiel = delenec / delitel;
printf("Ahoj %f\n",podiel);
```

Samozrejme, musíme definovať premenné "deliteľ" a "delenec", t.j. priradiť im vhodný dátový typ a počiatočnú hodnotu. V tomto prípade by premenné mali mať typ, ktorý umožňuje prácu s desationnou čiarkou, t.j. `float` alebo `double`. Problém s dátovými typmi platí aj keď pracujeme s hodnotami, ktoré sú uložené v premenných.

Navyše, prvý pohľad na riadok, kde priradíme hodnotu do premennej "podiel", nám nič nepovie o tom, či operácia prebehne správne lebo si nemusíme pamätať aký typ sme priradili operandom "deliteľ" a "delenec". Aby sme si boli istí, že delenie prebehne správne, musíme splniť tieto predpoklady:

- obidva operandy musia mať typ s desatinnou čiarkou
- deliteľ nesmie byť nula.

Keby sme chceli správne deliť dve celé čísla, nemusíme dostať správny výsledok:

```
int delenec = 5;
int delitel = 2;
float podiel = delenec / delitel;
printf("Ahoj %f\n",podiel);
```

Oba operandy majú typ `int` a výsledok ich delenia je zase `int`, teda dôjde k implicitnej typovej

konverzii. Správny výsledok dostaneme až keď majú operandy správny typ:

```
float delenec = 5;
float delitel = 2;
float podiel = delenec / delitel;
printf("Ahoj %f\n",podiel);
```

3.6.1. Úloha na precvičenie

Čo sa stane ak bude deliteľ nulový? Čo sa stane, ak budeme ďalej narábať s výsledkom delenia nulou?

3.7. Vyhodnotenie operácií

Aby sme si boli istý, že delenie dvoch premenných bolo správne, musíme vykonať tzv. **explicitnú typovú konverziu** a overiť či deliteľ nie je nulový:

```
int delenec = 5;
int delitel = 2;
if (delitel == 0){
    printf("Nie je mozne delit");
}
else{
    float podiel = (float)delenec / (float)delitel;
    printf("Ahoj %f\n",podiel);
}
```

Konštrukcia `if/else` e určená na podmienené vykonanie. Zabráni tomu, aby sa vykonala operácia, kotrá nie je defimovaná (delenie nulou).

Typová konverzia je ďalšia operácia s premennými, ktorú pozná jazyk C. Do zátvorky uvedieme typ do ktorého chceme meniť. Prekladač vytvorí dočasné pamäťové miesto so zadaným dátovým typom do ktorého uloží hodnotu za zátvorkou. Toto dátové miesto s novým typom sa potom použije pri ďalších operáciách vo výraze.

Operácia priradenia (`=`) je vyhodnocovaná v určitom poradí. Najprv sa vyhodnotí výraz (*expression*) na pravej strane a výsledok sa uloží do premennej, ktorej názov sa nachádza na ľavej strane (*lvalue*). Iné pravidlá platia pre vyhodnocovanie operácie delenia. Najprv sa vyhodnotí výraz na ľavej strane, potom na pravej strane.

3.8. Kopírovanie hodnoty

Operácia priradenia je ekvivalentná operácii kopírovaniu hodnoty. Hodnotu z jednej premennej môžeme nakopírovať do druhej premennej a tam s ňou pracovať bez toho, aby to ovplyvnilo pôvodnú premennú. Môžeme napísať:

```
int povodna = 5;
int nova = povodna;
printf("Ahoj %d\n",nova);
```

Zistili sme, že hodnota premennej `povodna` sa nakopírovala do premennej `nova`. Že ide naozaj o operáciu kopírovania si môžeme overiť tým, že zmeníme hodnotu pôvodnej premennej a sledujeme, či to ovplyvní hodnotu novej premennej.

```
int povodna = 5;
int nova = povodna;
printf("Nova hodnota je  %d\n",nova);
printf("Povodna hodnota je  %d\n",povodna);
povodna = 4;
printf("Po priredeni je povodna hodnota  %d\n",povodna);
printf("a nova hodnota stale je  %d\n",nova);
```

Chapter 4. Funguj - Návrh a používanie vlastnej funkcie

Naučíme sa vytvárať a využívať vlastné funkcie.

Naučíte sa:

- volanie funkcie bez parametrov,
- parametre funkcie,
- volanie funkcie odovzdávanie parametrov kópiou,
- návratová hodnota funkcie a jej využitie.

4.1. Definícia funkcie

S vytváraním a využívaním funkcií v jazyku C sme sa už stretli, možno aj bez toho aby ste to vedeli. V našom hračkovom programe sme si už definovali vlastnú funkciu, ktorej sme dali meno `main`.

```
#include <stdio.h>

int main(){ ①
    printf("Ahoj svet\n");
    return 0;
}
```

① Definícia funkcie `main`.

Definícia funkcie sa skladá z nasledovných častí:

- návratový typ (v tomto prípade `int`)
- názov funkcie (v tomto prípade `main`)
- argumenty funkcie (uvedené v okrúhlych zátvorkách, v tomto prípade žiadne)
- telo funkcie (časť v zložených zátvorkách)

Ak určíme nejaký návratový typ, tak sľubujeme prekladaču, že pomocou príkazu `return` vrátime hodnotu daného typu. Príkaz `return` spôsobí okamžité prerušenie vykonávania funkcie a funkcia vráti danú hodnotu, ktorú je možné ďalej spracovávať. Posledný príkaz

```
    return 0;
```

ukončí vykonávanie hlavnej funkcie a operačnému systému odovzdá hodnotu `0`. Ak ako návratový typ určíme `void`, neočakávame žiadnu návratovú hodnotu a príkaz `return` nemusíme uviesť. Tieto znalosti môžeme využiť a definovať si vlastnú funkciu pre výpis správy.

Definícia vlastnej funkcie `pozdrav`

```
#include <stdio.h>


void pozdrav(){ ①
    printf("Dobry den\n");
}

int main(){ ②
    printf("Ahoj svet\n");
    return 0;
}
```

① Definícia funkcie `pozdrav`.

② Definícia funkcie `main`.

Čo sa stane? Preklad programu prebehne úspešne, to znamená, že program je syntakticky správny. Napriek očakávaniu ale zdvorilý pozdrav nebude vypísaný.

4.1.1. Úlohy na precvičenie

Ćo sa stane, keď vynecháme príkaz `return` z hlavnej funkcie? Čo sa stane, keď hlavnej funkcii určíme návratový typ `void`? Modifikujte príklad a zistite to.

4.2. Volanie funkcie

Definícia funkcie je určitým spôsobom ekvivalentná k definícii slova v slovníku. Ak poznáme slovo, to ešte neznamená, že sme ho použili. Ak chceme definovanú funkciu aj použiť, musíme ju **zavolať**. Volanie funkcie bez parametrov vykonáme zapísaním mena funkcie a prázdnych okrúhlych zátvoriek. Každý príkaz musíme ukončiť bodkočiarkou.

```
#include <stdio.h>

void pozdrav(){
    printf("Dobry den\n");
}

int main(){
    pozdrav(); ①
    printf("Ahoj svet\n");
    return 0;
}
```

① Volanie vlastnej funkcie `pozdrav`

Zistili sme, že volanie vlastnej funkcie sa neodlišuje od volania funkcie zo štandardnej knižnice. Náš jednoduchý program neobsahuje príkaz na vykonanie funkcie `main`. Podľa dohody sa o niečo také

stará operačný systém pri spustení programu. Operačný systém pri spustení programu vyhľadá definíciu funkcie `main` a tej odovzdá riadenie. Až potom sa vykonávajú funkcie, ktoré určil programátor.

4.3. Návratová hodnota a argumenty funkcie

```
void pozdrav(){
    printf("Dobry den\n");
}
```

Takto definovaná funkcia nemá definovanú návratový hodnotu. Výpis správy, ktorý funkcia vykoná je možné brať ako **vedľajší efekt**, ktorý nemá priamy súvis so vstupom funkcie.

Funkciu si môžeme predstaviť ako kuchynského robota, ktorý vie spracovať určitý druh potravín. Vstupné argumenty sú suroviny, ktoré vkladáme do kuchynského robota. Návratový typ je výsledok spracovania vstupných surovín. Napríklad možným vstupom do mlynčeka na mäso je kus mäsa a výstupom je mleté mäso. Automatický mlynček na mäso bude mať problémy s niektorými surovinami. Napríklad orechy alebo hrozno na vstupe môžu spôsobiť zničenie robota alebo výstup nebude mať očakávané vlastnosti.

Funkcia s vedľajším efektom

```
+---------+
       vstup ---> | funkcia | ---> výstup
                  +----+----+
                       |
                       v
                  vedľajší efekt
```

Funkcia, ktorú sme definovali vyššie nemá definované vstupné argumenty ani výstupný typ. Funkciu si môžeme definovať tak, že výstupná hodnota závisí od vstupných argumentov. Vstupné argumenty sú definície premenných do ktorých sa priradí hodnota počas volania funkcie. Vstupné premenné uvedieme počas definície do okrúhlych zátvoriek a oddelíme ich čiarkou.

```
int spocitaj(int a,int b){
    return a + b;
}
```

Týmto kusom kódu sme definovali funkciu, ktorá vezme dve čísla, spočíta ich a vráti výsledok ako návratovú hodnotu. Takúto funkciu vieme použiť na ľubovoľné premenné alebo hodnoty, stačí ich zadať ako argumenty funkcie:

```c
#include <stdio.h>

int spocitaj(int a,int b){
    return a + b;
}

int main(){
    spocitaj(2,3); ①
    return 0;
}
```

① návratovú hodnotu hneď zabudneme.

Ak si vyskúšame takýto program, zistíme, že nie je veľmi užitočný. Napriek tomu, že vykonáme operáciu spočítania, výsledok sa nedostaví. Je to z toho dôvodu, že sme na to nedali príkaz. Program poslušne vykoná to, o čo sme ho požiadali, ale nič viac.

Na zobrazenie hodnoty premennej je potrebné použiť funkciu `printf()`. Ako jej druhý argument uvedieme názov premennej, ktorej obsah sa má vypísať. Dávajme ale pozor, aby formátovacia značka v prvom argumente sedela s typom premennej:

```c
#include <stdio.h>

int spocitaj(int a,int b){
    return a + b;
}

int main(){
    int vysledok = spocitaj(2,3);
    printf("Vysledok spocitania 2 + 3 je %d\n",vysledok);
    return 0;
}
```

Tento program je oveľa užitočnejší. Najprv si definujeme premennú `vysledok` a do nej si uložíme návratovú hodnotu funkcie `spocitaj`. Funkcia `printf` spôsobí výpis správy na obrazovku ako vedľajší efekt vykonania. Návratová hodnota funkcie `printf` nás nezaujíma, preto ju ignorujeme. Návratová hodnota celého programu bude *0*, čo je signál operačnému systému, že je všetko v poriadku.

Keďže už vieme definovať vlastné funkcie s argumentami aj pracovať s návratovými hodnotami, môžeme si definovať funkciu, ktorá spočíva ľubovoľné dve čísla a výsledok vypíše na obrazovku. Využijeme kód, ktorý už ovládame:

```c
#include <stdio.h>

int spocitaj(int a,int b){ ①
    return a + b;
}

void vypis_sucet(int a,int b){ ②
    int vysledok = spocitaj(a,b);
    printf("Vysledok spocitania %d + %d je %d\n",a,b,vysledok);
}

int main(){ ③
    vypis_sucet(4,5);
    return 0;
}
```

① Argumenty funkcie sú dve celé čísla. Návratová hodnota funkcie `spocitaj` je celé číslo, ktoré je súčtom argumentov. Funkcia nemá vedľajšie efekty.

② Argumenty funkcie `vypis_sucet` sú dve celé čísla. Funkcia má prázdny návratový typ (nemusí mať `return`). Jej vedľajším efektom je výpis na obrazovku.

③ Funkcia `main` nemá argumenty a volá ju operačný systém pri spustení programu. Má celočíselný návratový typ, ktorého hodnotu prevezme operačný systém.

4.4. Úloha na precvičenie

1. Aký bude výsledok volania funkcie `vypis_sucet(4.5,4.5)` ?
2. Aká bude návratová hodnota tejto funkcie a čo sa vypíše na obrazovku ?

Vytvorte program a zistite to.

Chapter 5. Hoď to do stroja - Jednoduchý vstup z klávesnice a podmienky

Naučíte sa

- Používať funkciu `scanf` na načítanie do celočíselnej premennej.
- Vytvárať časti kódu, ktoré sa vykonajú iba v prípade ak je splnená podmienka.
- Overiť si výsledok načítania.
- Zistíte, že niektoré výrazy sú pravdivé alebo nie podľa okolností.

V predošlom tutoráli sme sa naučili vytvoriť funkciu na spočítanie dvoch čísel a na výpis výsledku. V tomto bloku sa naučíme, ako vytvoriť interaktívny program, ktorý bude schopný požiadať používateľa o vstup, načítať zadanú hodnotu do premennej a vypísať výsledok. Naučíme sa aj bojovať s nezodpovednými používateľmi, ktorí testujú našu pozornosť a trpezlivosť a zadávajú niečo iné ako očakávame. Výsledkom bude kalkulačka, vhodná aj pre malé deti.

5.1. Načítanie z klávesnice

Doteraz vytvorený program už je použiteľný ako jednoduchá kalkulačka, ale iba pre nás programátorov. Tento postup naozaj nie je vhodný pre každého. Bolo by fajn, keby s našou kalkulačkou vedela pracovať aj moja babka.

Do teraz sme počiatočné hodnoty premenných určovali priamo v programe. Na to aby sme zmenili čísla na spočítanie, musíme zmeniť náš program na správnom mieste a zopakovať proces prekladu a spustenia. Slovo premenná je od slova *meniť* a preto skúsime zmeniť náš program tak, aby sa premenná vedela meniť počas behu. Požiadame používateľa o vstup krátkou správou, v ktorej mu vysvetlíme, čo od neho chceme:

Žiadosť o vstup

```
#include <stdio.h>

int main(){i
    printf("Súčtová kalkulačka\n");
    printf("Prosím zadajte prvý argument\n");
    return 0;
}
```

Vstup od používateľa sa v jazyku C dá získať rôznymi spôsobmi. Najjednoduchším (a zároveň najhorším [1]) spôsobom je využitie funkcie `scanf`. Zhodou okolností (ale nie náhodou) je použitie funkcie `scanf` veľmi podobné použitiu funkcie `printf`. Ako prvý argument šablónu reťazca ktorý očakávame a druhý argument je **adresa premennej**, kde sa má uložiť výsledok. Adresu premennej získame pomocou operátora `&` (ampersand).

Vstup celočíselnej premennej

```
int vstup = 0;
scanf("%d",&vstup);①
```

① Druhý argument funkcie `scanf` očakáva adresu, preto musíme použiť operátor `&` na získanie adresy premennej

Takto si definujeme celočíselnú premennú `vstup`, do ktorej si uložíme počiatočnú hodnotu nula. V druhom príkaze voláme funkciu `scanf`, ktorá spôsobí to, že program čaká na celočíselný vstup od používateľa z klávesnice a ak je to možné, tak výsledok uloží na zadanú adresu premennej. Celý program s kalkulačkou môže vyzerať takto:

```
#include <stdio.h>

int spocitaj(int a,int b){
    return a + b;
}

void vypis_sucet(int a,int b){
    int vysledok = spocitaj(a,b);
    printf("Vysledok spocitania %d + %d je %d\n",a,b,vysledok);
}

int main(){
    printf("Súčtová kalkulačka\n");
    printf("Prosím zadajte prvý argument\n");
    int a = 0;
    scanf("%d",&a);
    printf("Prosím zadajte druhý argument\n");
    int b = 0;
    scanf("%d",&b);
    vypis_sucet(a,b);
    return 0;
}
```

5.1.1. Úloha na precvičenie

Preložte program, spustite ho a dajte vyskúšať Vašej babke alebo sedemročnému dieťaťu. Za akých podmienok program funguje správne a kedy nastáva zlyhanie?

5.2. Ošetrenie vstupu

Po preskúšaní programu sme zistili, že program síce funguje správne, ale iba v prípadoch keď zadaný vstup vyhovuje formátovacej značke `%d`, teda celé číslo. Ak používateľ zadá niečo iné, výsledky nie sú podľa očakávania. Program môže vyzerať, že funguje v poriadku, ale pracuje s nesprávnymi hodnotami bez toho aby nás na to upozornil.

Jedným z možných riešení je používateľa upozorniť na možmé problémy a poprosiť, aby zadával iba správne hodnoty:

```
printf("Súčtová kalkulačka celých číslek\n");
printf("Prosím zadajte prvý argument (ak nezadáte celé číslo, tak sa kalkulačka pokazí
)\n");
int a;
scanf("%d",&a);
```

Kalkulačka by sa mala správať tak, ako to používateľ očakáva, inak vznikne nespokojnosť na strane spotrebiteľa, môjho šéfa a v konečnom dôsledku aj moja nespokojnosť. Ani by som nebol veľmi šťastný, keby som si takúto kalkulačku kúpil. Pripomína mi to prístup niektorých úradov k oprave ciest. Namiesto vyasfaltovania dier pribudne značka s chrbátom dvojhrbej ťavy alebo s obrázkom uja s lopatou.

Ťava podľa http://ascii.co.uk/art/camel

```
,,_
                (=-'
           /\/\  ))
         ~/    \/ |
         | )___(  |
         |/     \||
   ejm98 |'      |'
```

Skúsme urobiť kalkulačku odolnejšiu voči neočakávanému vstupu a overiť, či načítanie prebehlo úspešne. Aby sme to vedeli urobiť, musíme si bližšie prečítať dokumentáciu funkcie `scanf`. Zistíme, že návratová hodnota funkcie `scanf` nie je až taká nezaujímavá ako v prípade funkcie `printf`.

Technickú dokumentáciu funkcie `scanf`. zobrazíme príkazom `man scanf` alebo na internete.

Funkcia `scanf` vráti počet úspešne načítaných hodnôt, čo je v našom prípade 1. Túto skutočnosť môžeme využiť na to aby sme našu kalkulačku urobili odolnejšou voči neočakávanému vstupu. Ak používateľ nezadal správny reťazec, môžeme ho na túto skutočnosť upozorniť. Zapamätáme si návratovú hodnotu a môžeme ju vypísať:

```
int a = 0;
int pocet_hodnot = scanf("%d",&a);
printf("Pocet uspesne nacitanych hodnot je %d\n",pocet_hodnot);
printf("Nacitana hodnota je %d\n",a);
```

Vieme síce zistiť, či nastal nesprávny vstup, ale nevieme s tým nič robiť. Aby sme vedeli ošetriť nesprávny vstup od používateľa, musíme napísať kód, ktorý sa vykoná iba v prípade, že používateľ zadal nesprávny vstup. Na to použijeme podmienku **if**.

5.2.1. Úloha na precvičenie

Čo sa stane ak namiesto celého čísla zadám reťazec alebo číslo s desatinnou čiarkou?

5.3. Podmienka if

Jazyk C nám umožňuje napísať taký kód, ktorý sa spustí iba v prípade, že je splnená určitá podmienka. Používame na to kľúčové slovíčko **if** za ktorým do okrúhlych zátvoriek napíšeme podmienku. Nasledujúci príkaz alebo blok príkazov sa vykoná iba v prípade, že je podmienka pravdivá. Podmienka v zátvorke je pravdivá práve vtedy keď je nenulová. Tento kód ilustruje vytvorenie bloku kódu pomocou podmienky **if**, ktorý sa vykoná vždy:

```
if (2) {
    printf("Vykonam sa vzdy\n");
}
```

Ak je výraz v zátvorke nulový, podmienka sa nevykoná. Nasledovný blok kódu v podmienke **if** sa nevykoná nikdy:

```
if (0) {
    printf("Nevykonam sa nikdy\n");
}
```

Namiesto hodnoty môžeme do zátvorky napísať výraz, ktorý sa vyhodnotí na nulovú alebo nenulovú hodnotu:

```
if (1 == 2) {
    printf("Nevykonam sa lebo 1 sa nerovna 2\n");
}
```

Podmienka **if** môže byť nasledovaná blokom **else**, ktorý sa vykoná iba v prípade, že podmienka nie je splnená:

```
if (1 == 2) {
    printf("Nevykonam sa lebo 1 sa nerovna 2\n");
}
else {
    printf("Vzdy viem ze 1 sa nerovna 2\n");
}
```

Operátor `==` znamená operáciu porovnania, ktorá ak je pravdivá tak vracia nenulovú hodnotu a ak je nepravdivá, tak vracia nulu. Operácia `!=` (nerovná sa) sa správa opačne ako operácia `==`, vráti *1* v prípade, že dve hodnoty nie sú rovnaké a nulu v iných prípadoch.

5.4. Operátor porovnania a operátor priradenia

Je rozdiel medzi operáciou `=` a `==`. Operácia priradenia `=` spôsobí vloženie hodnoty do premennej a vrátu pravdivú hodnotu. Operácia porovnania `==` vráti pravdivú hodnotu iba vtedy, ak je hodnota na ľavej strane rovnaká ako na pravej strane.

Je ľahké si zameniť tieto dve operácie. Použitie priradenia namiesto porovnania spôsobí, že neočakávané správnaie programu a kompilátor nás na to nemusí upozorniť.

Operácia priradenia vždy vracia nenulovú hodnotu a ako vedľajší efekt spôsobí kopírovanie do premennej na ľavej strane. Nasledovný kód sa teda nebude správať podľa očakávania:

```
int pocet_hodnot = 0;
if (pocet_hodnot = 1){ ①
    printf("Pocet nacitanych hodnot je 1\n",a);
}
else {
    printf("Nenacitala sa ziadna hodnota\n",a);
}
```

① Výraz `pocet_hodnot = 1` vráti nenulovú hodnotu a jeho vedľajší efekt je priradenie hodnoty do premennej.

Pozor, v jazyku C je operátor porovnania `==` rozdielny ako operátor priradenia `=`.

Blok kódu v podmienke sa vykoná vždy a prepíše sa hodnota premennej. Takto napísaný kód je nesprávny. Oveľa lepšie je vždy v podmienke `if` využívať operáciu porovnania:

```
if (pocet_hodnot == 1){
    printf("Hodnota pocet_hodnot je 1\n",a);
}
```

Operácia porovnania funguje správne iba na celočíselné hodnoty. Hodnoty s desatinnou čiarkou nemusi byť vnútorne reprezentované rovnako aje keď rovnako vyzerajú. Operácia porovnania nefunguje ani na polia, reťazce a iné zložitejšie dátové typy.

5.4.1. Úlohy na precvičenie:

Vykoná sa nasledovný blok kódu?

```
if (-1) {
    printf("Je -1 pravdivy vyraz?\n");
}
```

Čo vypíše takéto volanie funkcie `printf`?

```
printf("Hodnota pravdiveho vyrazu je %d",1==1);
```

5.5. Vstup s ošetrením

Už vieme napísať kód, ktorý sa vykoná iba ak je vstup od používateľa nesprávny. Ak bude zadaná nejaká nesprávna hodnota, tak vieme napísať upozornenie a prerušiť program ak je to potrebné.

Operátor porovnania môže využívať aj premenné, takže môžeme napísať:

```
#include <stdio.h>

int main(){
    printf("Súčtová kalkulačka\n");
    printf("Prosím zadajte prvý argument\n");
    int a = 0;
    // Nacita hodnotu pod pouzivatela do premennej a
    // a vrati pocet nacitanych hodnot do pocet_hodnot
    int pocet_hodnot = scanf("%d",&a);
    printf("Pocet uspesne nacitanych hodnot je %d\n",pocet_hodnot);
    if (pocet_hodnot != 1){
        printf("")
    }
    printf("Nacitana hodnota je %d\n",a);
    return 0;
}
```

V tomto príklade vieme vykonať nejaký blok príkazov iba v prípade, že bola zadaná nesprávna hodnota. Celý príklad s ošetrením vstupu bude trochu zložitejší:

```c
#include <stdio.h>

int spocitaj(int a,int b){
    return a + b;
}

void vypis_sucet(int a,int b){
    int vysledok = spocitaj(a,b);
    printf("Vysledok spocitania %d + %d je %d\n",a,b,vysledok);
}

int main(){
    printf("Súčtová kalkulačka\n");
    printf("Prosím zadajte prvý argument\n");
    int a = 0;
    int pocet_hodnot = 0;
    pocet_hodnot = scanf("%d",&a);
    if (pocet_hodnot == 1){
        printf("Prosím zadajte druhý argument\n");
        int b = 0;
        int pocet_hodnot = 0;
        pocet_hodnot = scanf("%d",&b);
        if (pocet_hodnot == 1){
            vypis_sucet(a,b);
        }
        else {
            printf("Zle ste zadali druhy argument. Cakal som cele cislo.\n");
        }
    }
    else {
        printf("Zle ste zadali prvy argument. Cakal som cele cislo.\n");
    }
    return 0;
}
```

Tento program má o niečo lepšie vlastnosti. Pomocou vetvenia **if-else** sme dosiahli, že funkcia na sčítanie sa bude volať iba v prípade, že obidve hodnoty boli zadané správne. Zabráni sa "pokazeniu" kalkulačky - vylúčili sme situáciu, keď bude program fungovať, ale nesprávne. Vďaka tomu si môžeme byť istý, že ak dostaneme výsledok tak bude správny.

5.6. Úlohy na precvičenie

Upravte kalkulačku tak, aby vedela pracovať aj s číslami s desatinnou čiarkou.

[1] vysvetlíme neskôr

Chapter 6. Bicyklová reťaz: cykly a reťazce

Naučíte sa

- Vytvoriť cyklický kód, ktorý sa opakuje dovtedy kým platí určitá podmienka.
- Načítať reťazec a celé číslo s ošetrením.
- Používať funkciu `fgets` na načítanie reťazc a `sscanf` na konverziu reťazca na číslo.

Pomocou našich zázračných programátorských schopností sme boli schopní vo veľmi krátkom čase navrhnúť a implementovať kalkulačku vhodnú aj pre netechnické typy. Stále tam však zostávajú viaceré nedostatky. V prípade, že zadáme nesprávny vstup, musíme celý proces začať od znova lebo program sa pokazí.

6.1. Cyklus while

V tejto kapitole upravíme kalkulačku tak, aby sa nevzdávala pri prvom neúspechu, ale vytrvalo prosila o správny vstup až kým sa to nepodarí. Na to využijeme cyklus typu **while**, ktorý bude prebiehať dovtedy, pokiaľ je splnená podmienka:

```
while(1){
    printf("Ja neprestanem\n");
}
```

Podmienka `1` je splnená ak je vyhodnotená na nenulovú hodnotu. Číslo 1 bude v našich podmienkach nenulové vždy, takže takýto cyklus bude s prebiehať do nekonečna (alebo až kým nevypnú elektrický prúd). Na koniec ktorý nenastane nemusíme čakať, ale program vypneme pomocou klávesovej skratky Ctrl+C.

Ak si to rozmeníme na drobné, program sa najprv spýta či je podmienka v okrúhlej zátvorke splnená. Ak je, tak sa vykoná časť v zložených zátvorkách a pokračuje novým cyklom, ktorý opäť zisťuje, či má pokračovať.

Zápis cyklu typu **while** je veľmi podobný podmienke **if**, hlavný rozdiel je v tom, že telo cyklu **while** môže byť vykonané viac krát (pokiaľ platí podmienka).

6.2. Načítanie hodnoty s ošetrením

Ak vieme napísať cyklus, môžme skúsiť vytvoriť taký kód, ktorý bude žiadať o vstup dovtedy pokým nebude správny. Inými slovami, ak bude vstup nesprávny tak sa bude požiadavka opakovať.

Na zastavenie cyklu vo vhodnom momente využijeme fakt, že môžme zmeniť hodnotu premennej.

Aby nám cyklus prebehol aspoň raz, nastavíme počiatočnú hodnotu premennej `pocet_hodnot` na nulovú hodnotu. Ak v priebehu vykonávania tela cyklu zmeníme hodnotu premennej vhodným spôsobom, podmienka cyklu nebude splnená a cyklus viac nebude prebiehať.

Zatiaľ na účel načítania poznáme funkciu `scanf`.

```
int pocet_hodnot = 0;
int a = 0;
while (pocet_hodnot != 1){
    printf("Prosím zadajte prvý argument\n");
    // Pri druhom volaní sa funkcia scanf nechova podla ocakavania
    pocet_hodnot = scanf("%d",&a);
}
printf("Načítal som hodnotu %d\n",a);
```

Vyskúšajte si tento kód. Čo sa stane, ak namiesto vstupu napíšete písmeno? Ćo sa stane, ak súčasťou vstupu bude špeciálny znak pre koniec vstupu (`EOF` - `Ctrl+D`). Skúste zistiť, prečo sa program v prípade nesprávneho vstupu správa nečakane. Modifikujte tento program tak, aby v prípade nesprávneho vstupu o tom vypísal správu a skončil porgram.

Zistili sme, že takto napísaný program síce funguje skvele, ale iba v prípade že sa nevyskytnú "neočakávané" okolnosti (nesprávny vstup od používateľa). Dôvodom je to, že funkcia `scanf` nie je veľmi užitočná. Obsahuje vnútorný buffer (pomocné pole znakov), ktorý sa vyprázdni iba v prípade, že vstup z klávesnice bol správny. V prípade, že bol vstup nesprávny tak tam nesprávna hodnota ostane, až pokiaľ nebude spracovaná. Ďalšie volania funkcie `scanf` potom namiesto vstupu od používateľa stále pracujú s pôvodným, nesprávnym vstupom.

Vyplýva z toho, že funkcia `scanf` je použiteľná iba na veľmi jednoduché príklady, ale nie je vhodná na reálne použitie. Našťastie, riešenie je pomerne jednoduché - naprogramovať si vlastný "buffer" (miesto pre dočasné uloženie údajov), do ktorého budeme ukladať vstup od používateľa. Premenu vstupu z klávesnice na číslo budeme vykonávať osobitne na vlastnom buffri.

6.3. Pole a reťazec

Zmena nášho programu bude taká, že namiesto celého čísla, ktoré je možné zapísať nesprávne budeme očakávať všeobecnejší typ **reťazec**. Do reťazca si poznačíme všetko, čo používateľ zadal (ktoré klávesy stlačil) a premenu na celé číslo vykonáme neskôr.

Reťazec je sada znakov ukončená nulou. Každý znak je v pamäti reprezentovaný jedným kódom (číslom v rozsahu 0 až 255). Posledným znakom je vždy nula ktorá vyznačuje koniec reťazca. Hodnoty znakov sú zakódované do číselnej podoby pomocou ASCII tabuľky. Na uloženie reťazca (viacerých nakov naraz) budeme potrebovať si zapamätať viac čísel naraz.

Klasické celočíselné premenné v jazyku C umožňujú uloženie iba jednej hodnoty. Aby sme si mohli do premennej uložiť viac hodnôt, musíme o tom prekladaču povedať. Napríklad ak chceme "rozšíriť" celočíselnú premennú na viacero miest, môžeme napísať:

```
int pole[4];
```

Vyhradeniu viacerých pamäťových miest rovnakého typu vedľa seba vravíme **pole**. Celé si to vieme predstaviť ako rebrík:

Pole v pamäti

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


    index:    0     1       2       3
             +------+------+-------+-------+
    hodnota: |  4   |  3   |  2    |   1   |
             +------+------+-------+-------+
    adresa:  \#10   \#14   \#18    \#22
```

Rozdiel od klasického zápisu deklarácie premennej (príkazu na vyhradenie pamäťového miesta) je použitie hranatých zátvoriek. Pomocou hranatých zátvoriek v deklarácii premennej vravíme prekladaču, aby vyhradil viac miest naraz. Pole si môžeme zostaviť z ľubovoľného dátového typu, napr. `float`.

Na uloženie znaku je v jazyku C najvhodnejší typ `char` (znakový typ). Jeden znak je v pamäti uložený vo forme ASCII kódu, teda celého čísla. Typ `char` nie je veľmi odlišný od typu `int`, je ale menší. Do jednej premennej typu `char` sa zmestí 256 rôznych hodnôt. Na uloženie znaku to stačí a šetrí to pamäť.

Reťazec je pole znakov zakončené nulou, pamäť pre uchovanie desať znakového reťazca si vyhradíme takto:

```
char retazec[11];
```

Posledné políčko poľa musíme vyhradiť pre zápis nuly na konci.

Vždy musíme počítať s tým, že do poľa sa musí zmestiť aj nula na konci. Na uloženie `n` ASCII znakov nám treba minimálne `n+1` bajtov. Situácia je ešte zložitejšia ak chceme do pamäte uložiť aj znaky s dĺžňom alebo mäkčeňom. Jeden takýto znak môže zabrať dva alebo viac bajtov. Ak nevieme, aké znaky v pamäti budú, je celkom ťažké presne povedať, koľko pamäte budeme potrebovať. Nemalo by to byť viack ako trojnásobok zamýšľaného počtu znakov.

Ak si chceme zapamätať ľubovoľný vstup od používateľa (menší ako *100* ASCII znakov) a uložiť ho do poľa, môžeme zapísať:

```
char retazec[100];
printf("Zadajte hocijaký vstup:");
fgets(retazec,100,stdin);
printf("Napísali ste: %s\n");
```

Tento program počká na vstup z klávesnice a ak používateľ stlačí enter, vypíše ho.

Prvým argumentom funkcie `fgets()` je názov poľa, kam sa má uložiť výsledok. Druhým argumentom je veľkosť poľa, ktoré máme k dispozícii. Tretím argumentom je súbor z ktorého sa má

načítavať, v tomto prípade načítavame z klávesnice ako keby to bol otvorený súbor.

Premenná `stdin` je globálna premenná ktorá reprezentuje klávesnicu. Funkciu `fgets` je možné použiť aj na načítanie zo súboru.

Reťazec v pamäti. Posledná hodnota s indexom 3 môže byť ľubovoľná.

```
char reazec[4] = "14";

    index:    0     1       2       3
             +------+------+-------+-------+
    hodnota: |  '1' |  '4' |   0   |   x   |
             +------+------+-------+-------+
    adresa:  \#10   \#11   \#12    \#13
```

Operácia načítania má neistý výsledok. Ak sa načítanie nepodarí (napr. v súbore sa už nenachádza žiadny reťazec) tak program na túto skutočnosť neupozorní a pokračuje ďalej, hoci je v cielovom poli `retazec` nejaká predošlá hodnota. Vždy je potrebné si overiť návratovú hodnotu fukcie pre načítanie aby sme vedeli čo robiť v prípade, že načítanie zlyhá.

Funkcia `fgets()` nás vie s pomocou špeciálnej návratovej hodnoty `NULL` (nula) informovať o tom, že sa jej nepodarilo načítať nič. Ak je jej návratová hodnota nulová tak sa načítanie reťazca nepodarilo.

```
char retazec[100];
printf("Zadajte hocijaký vstup:");
if (fgets(retazec,100,stdin) != NULL) { ①
    printf("Napísali ste: %s\n");
}
else {
    printf("Nenapísali ste nič.\n");
}
```

① Funkcia `fgets()` vraciu nulu ak sa nič nepodarilo načítať. Kvôli dátovým typom sa nula v tomto prípade zapisuje ako `NULL`, inak to je obyčajná nula.

Načítanie prázdneho riadka sa nepovažuje za chybu - aj prázdny riadok je riadok a `fgets()` ho poslušne načíta do poľa. Chyba, teda nemožnosť ďalšieho vstupu môžeme simulovať stlačením `Ctrl+D`, čo je špeciálny znak pre koniec vstupu.

6.4. Konverzia reťazca na číslo

Zistili sme, že v jazyku C sú reťazec a číslo dve rozdielne veci. Zjednodušene môžme povedať, že reťazec je viac hodnôt vedľa seba, číslo je iba jediná hodnota. Na to aby sme so zadaným reťazcom vedeli vykonávať matematické operácie, musíme si ho premeniť na číslo. Tento krok do teraz robila za nás funkcia `scanf`, ktorá premieňala vstup z klávesnice vo forme reťazca bez toho, aby sme o nejakých reťazcoch tušili. Zistili sme ale, že má určité obmedzenia, ktoré je potrebné obísť.

Riešenie je našťastie jednoduché - konverziu na číslo je možné vykonávať nie len z klávesnice, ale aj z hodnôt, ktoré sme si poznačili pred tým do poľa. Slúži na to funkcia `sscanf`, ktorá sa správa rovnako ako funkcia `scanf`, ale namiesto klávesnice pracuje s reťazcom, ktorý jej zadáme ako argument. Netrpí teda problémom so zasekávajúcim sa buffrom.

Konverziu reťazca na číslo pomocou funkcie `sscanf` vykonávame takto:

```
int cislo = 0;
int pocet_hodnot = 0;
char retazec[10] = "14";
pocet_hodnot = sscanf(retazec,"%d",&cislo);
printf("Vase cislo je %d",);
```

Celé číslo v pamäti

```
int cislo = 14;

            +------+
    hodnota: | 14   |
            +------+
    adresa:  \#10   \#14
```

Už vieme správne vykonať konverziu reťazca na číslo, ale jeho načítanie z klávesnice je stále problém, lebo funkcia `scanf` sa zasekáva. Namiesto nej radšej použime funkciu `fgets`, ktorá slúži na čítanie znakov zo súboru a netrpí "zasekávaním". Prvý argument tejto funkcie je pole, do ktorého budeme načítavať. Druhým argumentom je maximálny počet znakov ktorý budeme načítavať. Počet načítavaných znakov nesmie byť väčší ako pole, do ktorého načítavame. Tretím argumentom je súbor, z ktorého budeme načítavať. Návratová hodnota je počet znakov, ktorý sa nám podarilo načítať.

Moment - načítanie zo súboru? Načítavame predsa z klávesnice. Toto nie je omyl - štandardná knižnica jazyka C nerobí rozdiel medzi súborom, sieťovou kartou alebo klávesnicou. Princíp načítavanie je pri všetkých zariadeniach rovnaký - sadu hodnôt ukladáme do poľa. Na tento účel slúži globálna premenná `stdin` o ktorú sa stará prekladač a ktorú vieme použiť na čítanie.

6.5. Celý program

Celý program bude potom vyzerať takto:

```c
#include <stdio.h>

int spocitaj(int a,int b){
    return a + b;
}

void vypis_sucet(int a,int b){
    int vysledok = spocitaj(a,b);
    printf("Vysledok spocitania %d + %d je %d\n",a,b,vysledok);
}

int main(){
    printf("Súčtová kalkulačka\n");
    int pocet_hodnot = 0;
    int a = 0;
    char buffer[100];
    char* mam_riadok = NULL;
    while (pocet_hodnot == 0){
        printf("Prosím zadajte prvý argument\n");
        mam_riadok = fgets(buffer,100,stdin);
        if (mam_riadok == 0){
                return 0;
        }
        pocet_hodnot = sscanf(buffer,"%d",&a);
    }
    int b = 0;
    pocet_hodnot = 0;
    mam_riadok = NULL;
    while (pocet_hodnot == 0){
        printf("Prosím zadajte druhý argument\n");
        mam_riadok = fgets(buffer,100,stdin);
        if (mam_riadok == 0){
            return 0;
        }
        pocet_hodnot = sscanf(buffer,"%d",&b);
    }
    vypis_sucet(a,b);
    return 0;
}
```

Cyklus pre načítanie sa zopakuje vždy, keď používateľ zadá neplatnú hodnotu a nepodarí sa konverzia reťazca na číslo. Ak nastane koniec vstupu, program sa skončí.

6.6. Úloha na precvičenie

Modifikujte program tak, aby vypísal chybové hlásenie o tom, že sa nebol zadaný platný vstup.

Modifikujte program, tak aby napísal chybové hlásenie v prípade že výpočet nebol vôbec vykonaný.

Chapter 7. Na správnej adrese

Naučíte sa

- Čo je to adresa premennej a ako ju zistíme.
- Ako pracovať s adresou premennej.
- Vytvoriť si funkciu ktorá zapisuje do premennej.

Dostali sme sa do stavu, že automatická kalkulačka funguje vynikajúco. Vieme načítať viaceré čísla z klávesnice a vieme s nimi urobiť aritmetické operácie. Nevýhodou tohto programu je, že sa v ňom stále opakuje tá istá časť - načítanie celého čísla do premennej.

V prípade krátkeho programu to nie je problém, ale v prípade dlhšieho program to komplikuje situáciu. Ak sme sa pomýlili v jednej časti, musíme opraviť aj všetky ostatné kópie.

Je výhodné zabezpečiť, aby rovnakú vec riešila práve jedna časť kódu. Vytvoríme si preto funkciu, ktorá sa bude starať o načítanie čísla z klávesnice. Túto funkciu bude možné použiť na viacerých miestach programu podľa potreby. Kód na načítanie čísla bude iba na jednom mieste a nemusíme stále kopírovať túto časť, stačí zavolať funkciu, ktorú už máme hotovú. Program sa bude možné jednoduchšie opraviť alebo vylepšiť, lebo nemusíme hľadať kde všade sa vyskytuje podobný kód.

7.1. Lokálna premenná

Návrh vlastnej funkcie pre načítanie do premennej nie je celkom triviálny. V predošlého textu vieme, že návratová hodnota sa väčšinou využíva na chybové hlásenie o tom, či načítanie a spracovanie prebehlo úspešne, podobne ako to je vo funkciách `fgets` alebo `sscanf`. Výsledok načítania a spracovania sa ukladá do jednej z argumentov funkcie.

Funkcie na načítanie a premenu vstupu z klávesnice zapisujú výsledok do jedného z argumentov a využívajú návratovú hodnotu na hlásenie výsledkov.

Vytvorme si vlastnú funkciu na načítanie z klávesnice. Budeme potrebovať pole na uloženie načítaného reťazca, premennú na uloženie výsledku a pomocnú premennú na overenie si, či bolo načítanie a konverzia úspešná.

Najprv načítame reťazec. Ak je načítanie úspešné,pokúsime sa premeniť reťazec na číslo. Ak je premena reťazca na číslo úspešná vrátime výsledok. V inom prípade vrátime nulu.

```
int nacitaj_cele_cislo(){
   int vysledok = 0;
   char vstup[20];
   int r = 0;
   r = fgets(vstup,20,stdin);
   if (r > 0){
      r = sscanf(vstup,"%d",&vysledok);
      if (r == 1){
          return vysledok;
      }
   }
   return 0;
}
```

Premenné definované vo funkcii `nacitaj_cele_cislo` nazývame lokálne premenné. Lokálna premenná platí iba v bloku v ktorom bola napísaná. Ak sa ju pokúsíme využiť v inom bloku, napr. vo funkcii `main` tak nám to prekladač nedovolí.

Takáto funkcia fungujee skvele, vyskúšajte:

```
int main(){
  int c = nacitaj_cele_cislo();
  // Dalsi riadok nebude fungovať, lebo vstup je lokálne premenná
  // prntf("Nacitaany retazec je %s\n",vstup);
  printf("Nacital som %d\n",c);
  return 0;
}
```

Úloha na precvičenie

Ćo sa stane ak definujete premennú v rámci cyklu alebo podmienky?

Lokálne premenná možno vyzerajú dosť neprakticky. Premenné v rôznych funkciách a blokoch musíáme definovať stále znovu, a top je "veľa" roboty aj pre priemerne lenivého programátora.

7.2. Kopírovanie argumentov funkcie

Argument funkcie je lokálna premenná, ktorá platí pošas behu funkcie. Volanie funkcie vždy spôsobí vytvorenie nových premenných do ktorých sa skopíruje hodnota argumentov funkcie. Po skončení sa argumenty správajú ako iné lokálne premenné a hodnoty v nich uložené sa zabudnú.

V jazyku C sú funkcie volané vždy s kópiou argumentov. Keď napíšeme takýto program:

```
int mocnina(int argument){
    return argument * argument;
}

int main() {
    int argument = 7;
    int vysledok = mocnina(argument);
    printf("Vysledok je %d",vysledok);
    return 0;
}
```

Stane sa niekoľko vecí:

1. Do premennej s názvom `argument` sa uloží hodnota 7.
2. Hodnota, ktorá je uložená v premennej `argument` vo funkcii main sa skopíruje do premennej argument vo funkcii `mocnina`. Tieto dve premenné sú rozdielne, hoci sa volajú rovnako. Každá premenná má svoj rozsah platnosti - prvá premenná `argument` platí v rámci funkcie `main` a druhá premenná `argument` platí iba v rámci funkcie `mocnina`.
3. V rámci funkcie `mocnina` sa vypočíta návratová hodnota.
4. Riadenie programu sa vráti do funkcie `main`, návratové hodnota funkcie `mocnina` sa skopíruje do premennej `vysledok`.

Vidíme, ak funkcii odovzdáme hodnotu premennej, funkcia nie je schopná zapísať do pôvodnej premennej. Namiesto toho sa urobí kópia a výsledok spracovania kópie môžeme využiť ako návratovú hodnotu funkcie. Výhoda tohto prístupu je, že sme si istý, že funkcia nám nezmení hodnotu premennej. Nevýhoda je taká, že toho kopírovania je niekedy príliš veľa, čo nemusí vyhovovať, ak sú dáta s ktorými pracujeme príliš veľké. Druhá nevýhoda je, že dokážeme napísať funkciu, ktorá vracia maximálne jednu hodnotu.

7.2.1. Cvičenie

Aby sme si overili tvrdenie že pri volaní funkcie dochádza ku kopírovaniu, môžeme skúsiť modifikovať tento program a overiť si, že premenné s názvom `argument` sú v skutočnosti dve rôzne premenné. Modifikujeme hodnotu druhej premennej a sledujeme, či sa zmena prejaví v prvej premennej. Ak sú premenné rovnaké, nastane zmena v prvej premennej. Ak sú rôzne, znema sa neprejaví.

```
int mocnina(int argument){
    argument = 2;
    return argument * argument;
}

int main() {
    int argument = 7;
    int vysledok = mocnina(argument);
    printf("Vysledok je %d",vysledok);
    printf("Hodnota premennej argument je %d",argument);
    return 0;
}
```

1. Akú hodnotu premennej `argument` vypíše program?
2. Vypíšte adresu pamäte všetkých premenných s názvom `argument`.

7.3. Globálna premenná

Ćo stane ak do funkcie na načítanie namiesto čísla zadáme písmená? Takto napísaný program nie je schopný rozlíšiť chybne zadaný vstup od správne zadanej nuly. Takáto funkcia nie je použiteľná na našu slávnu kalkulačku - spôsobovala by neočakávané správanie v prípade nesprávneho vstupu. V prípade, že sme sa pomýlili by predstierala, že bola správne načítaná nula a spokojne by napísala výsledok. Potrebovali by sme ešte jeden výstup funkcie ktorý by sme využili na hlásenie o úsešnosti operácie.

Možno by sa dal využiť "centrálny odkladací prestor" - pamäť dostupná z každého miesta programu. Globálne premenné nie sú závislé od aktuálneho bloku a platia v každom mieste programu. Použitie globálnej premennej na hlásenie o výsledku operácie môže vyzerať takto:

Definícia globálnej premennej vyzerá takto:

```
#include <stdio.h>

int vysledok;
char vstup[20];
int r;

int nacitaj_cele_cislo(){
   r = fgets(vstup,20,stdin);
   if (r > 0){
      r = sscanf(vstup,"%d",&vysledok);
      if (r == 1){
          return vysledok;
      }
   }
   return 0;
}
```

Globálna premenná je jedným zo spôsobov na zisťovanie, či funkcia prehela v poriadku. Umožňuje obísť obmedzenie na práve jednu návratovú hodnotu. V tomto príklade je správa o úspešnosti operácie načítanie uložená v premennej `r`.

```
int main(){
  // c je lokálna premenná
  int c = nacitaj_cele_cislo();
  //  Dalsi riadok bude fungovať, lebo vstup je globálna premenná
  printf("Nacitaany retazec je %s\n",vstup);
  if (r == 0){
    printf("Načitanie bolo asi úspešné (ale možno aj nie)\n");
  }
  printf("Nacital som %d\n",c);
  return 0;
}
```

Ich použitie so sebou nesie niekoľko rizík: Funkcie, ktoré používajú globálne premenné sú nevypočítateľné - ich výsledok môže závisieť od predchádzajúceho behu programu. Globálna premenná vo funkcii môže fungovať ako "zadné dvierka" - premenná ktorá nie je vstupná ani výstupná. ale napriek tomu ju vieme využívať na ovplyvnenie výstupu alebo na odčítanie výsledku.

Globálna premenná má ale niekoľko nevýhod:

- Existencia "bočnej" premennej je utajená - nie je súčasťou predpisu funkcie.
- Na to, aby sme vedeli funkciu s globálnou premennou využívať, musíme poznať jej zdrojový kód.
- Počiatočný stav globálnej premennej nie je jasne definovaný, môže sa ľahko stať že nebude správne inicializovaná.
- Ľahko sa môže stať, že rovnakú globálnu premennú využíva aj iná funkcia.
- Ďalšie problémy vzniknú ak globálnu premennú využíva viac procesov naraz.

Program využívajúci globlne premenné sa ľahko môže premeniť na programátorské "peklo" - nepredvídateľný a nečitateľný kód. Ak si myslíte že práve Vy by ste to zvládli.... Cesta do pekla býva široká.

INFO: Globálnu premennú využívame iba ak sa problém nedá riešiť inak.

Príkladom globálnej premennej s ktorým ste sa už stretli je premenná `stdin`. O inicializáciu sa stará prekladač a je prístupná počas celého behu programu.

7.4. Úloha na precvičenie

Aký bude výsledok tohto programu?

```
void spocitaj(int a, int b,int vysledok){
   vysledok = a + b;
   printf("Výsledok 1+2 je %d\n");
}
int main() {
   int vysledok = 0;
   int a = 1;
   int b = 2;
   spocitaj(1,2,vysledok);
   printf("Výsledok %d+%d v inej funkcii je %d\n",a,b,vysledok);
}
```

Zistili sme, že výsledok je iný ako by sme očakávali na prvý pohľad. Premenná `vysledok` vo vnútri funkcie `spocitaj` je po operácii spočítania odlišná od premennej `vysledok` v funkcii `main`.

7.5. Adresa premennej

Globlna premená nie je ideálny spôsob ako vytvoriť funkciu s viacerými výstupnými hodnotami. Dodatočná výstupná hodnota by mala byť jasne dekladrovaná ako súčasť predpisu funkcie. Ale ako na to, keď jazyk C pozná iba jednu návratovú hodntu a všetky vstupné argumenty sú vlastne lokálne premenné ktoré sa "zabudnú" hneď ako funkcia skončí?

Musíme využiť ďalšiu fintu - adresu premennej. Každá poremenná v pamäti má svoje číslo, ktoré vyjadrujú miesto kde je údaj uložený. Toto číslo voláme adresa alebo smerník. Je podobná klasickej poštovej adrese - vieme podľa nej vyjadriť cieľ "doručenia", alebo "odosieľateľa". Adresa premennej je číslo, ktoré slúži rovnako dobre, ako meno premennej.

Pomocou adresy premennej "simulovať" výstupnú premennú pomocou vstupného argumentu. Namiesto vstupnej hodnoty funkcii posunieme adresu lokálnej premennej, kde sa má zapísať výsledok. Funkcia zapíše výsledok na zadané mniesto a nebude vadiť, že hodnotu adresy zabudne - výsledok bude zapísaný na mieste, kde adresa ukazuje.

Pamäť počítača si vieme predstaviť ako obrovskú policu, rozdelenú na rovnako veľké priehradky. Priehradky sú očíslované od 1 až po `N`, kde `N` je dĺžka police. Do jednej priehradky sa dá uložiť práve jedna správa.

Náš kamarát procesor je veľmi šikovný a dokáže spracovať čokoľvek, čo je uložené v polici, stačí mu povedať kde je to uložené. Komunikácia s procesorom by mohla vyzerať takto:

```
Zober hodnotu z priehradky číslo 7 a pripočítaj k nej hodnotu v priehradke číslo 20. V
ýsledok ulož do priehradky číslo 40.
```

Vyber hodnoty z priehradky 17, odpočítaj sedem a výsledok ulož do priehradky 34. Tento spôsob komunikácie je pre procesor ľahko zrozumiteľný - jediné čomu rozumie sú čísla.

INFO: ituácia bude ešte o niečo zložitejšia ak sa jedna správa nezmestí do jednej priehradky. Niektorý typ správy si môže vyžadovať viac miesta ako iný - napr. na uloženie rodného čísla

budeme určite potrebovať viac miesta ako na uloženie veku osoby. Pri každom údaji potom musíme oznámiť koľko miesta na polici zaberá. Veľkosť jedného dátového miesta a zvyčajný spôsob práce s ním nazývame **dátový typ**.

Kompilátor dokáže adresám v pamäti prideliť symbolické mená - prideliť názov premennej. Tento názov potom môžeme používať v programe. Komunikácia s procesorom potom môže vyzerať o niečo lepšie:

```
Priehradka číslo 7 sa bude volať výsledok. Priehradka 6 sa bude volať argument. Zober
argument, vynásob ho samým sebou a ulož do výsledku.
```

Výsledok operácie druhá mocnina z argumentu bude uložený v priehradke s názvom "výsledok". Prekladač sa stará aj o veľkosť potrebnú pre uloženie jedného údaju.

Operačný systém sa stará o to, aby každý program mal prístup iba k svojej časti pamäte (vyhradí priehradky pre pracovníka) a nezasahoval do práce iných programov a nezasahoval do práce iných programov. Zabráni tým tomu, aby jeden program modifikoval pamäť, ktorá mu nepatrí. Ak "poštový úrad" patrí k tým väčším (napr. Pošta 1) a pri polici sa točí viacero pracovníkov, rýchlo stratíme prehľad o situácii. Ak sa poštár pomýli tak ľahko "primieša" časti hodnôt ktoré patria inému pamäťovému miestu. Ak je ukladaná hodnota príliš veľká, prepíše údaje, ktoré sú uložené vedľa.

Schopnosť poznamenať si pamäťovú adresu niektorej premennej na prvý pohľad nie je veľmi užitočná. (Niekoho síce môže hriať pri srdci možnosť sa pozrieť "pod kapotu" a ručne skontrolovať, či sa prekladač nepomýlil a nepridelil tú istú pamäť dva krát, ale to asi nebude náš prípad). Adresa premennej je dôležitá, ak chceme zapisovať do argumentov funkcie.

7.6. Smerníková premenná

V jazyku C adresu v pamäti (číslo priehradky v polici) nazývame smerník. Adresa premennej je celé číslo a je možné ho uložiť do premennej ako každé iné celé číslo.

Na zápis do argumentu funkcie musíme využiť adresu premennej.

Skúsme si jednoduchý program na výpočet mocniny čísla s pozrieme sa na pamäťové adresy, ktoré nám priradí prekladač. Hodnotu pamäťovej adresy zistíme pomocou operátora `&`. Hodnotu adresy môžeme vypísať v hexadecimálnom tvare pomocou formátovacieho znaku `%p`.

```
int argument = 7;
printf("Premenná argument je uložená na adrese %p.\n",&argument);
```

```
Premenná argument je uložená na adrese `0xffffffffef`.
```

Vidme že pamäťová adresa je nejaké číslo, ktoré je priradené konkrétnemu pamäťovému miestu. Dostupné pamäťové miesta sú zoradené za sebou a vzostupne očíslované - podobne ako priehradky na polici na pošte.

Hodnotu adresy v pamäti si môžeme uložiť do premennej. Takúto premennú nazývame smerníková premenná a jej typ obohatíme znakom . Smerníková premenná sa vždy viaže na typ pôvodnej premenej. Spolu s adresou máme poznačený aj typ na ktorý adresa ukazuje. Ak si chceme uložiť smerník na dátový typ int, zapíšeme to ako int:

```
int argument = 7;
int* adresa_argumentu = &argument;
printf("Premenná argument je uložená na adrese %p",adresa_argumentu);
```

```
Premenná argument je uložená na adrese `0xfffffffef`.
```

Do premennej adresa_argumentu typu int* si môžeme uložiť adresu pamäťového miesta typu int. Smerníkovú premennú na určitý typ môžeme využívať podobne ako klasickú premennú, musíme ale pamätať, že v nej uchovávame nie hodnotu ale adresu kde je hodnota uložená. K hodnote, ktorá je uložená na zadanej adrese sa dostaneme pomocu operátora *.

Operátor pomocou ktorého získame adresu je rozdielny od ktorou vyznačujeme smerníkovú premennú.

Pomocou operátora * môžeme zo smerníkovej premennej získať hodnotu, na ktorú smerník ukazuje:

```
int argument = 7;
int* adresa_argumentu = &argument;
printf("Premenná argument je uložená na adrese %p a obsahuje hodnotu
%d",adresa_argumentu,*adresa_argumentu);
```

```
Premenná argument je uložená na adrese 0xfffffffef.
```

7.7. Funkcia na zápis do premennej

Ak vieme adresu premennej, vieme aj zapisovať aj do samotnej premennej. To vieme využiť a vytvoriť funkciu, ktorá ako argument prijme adresu premennej a je schopná do nej zapisovať. Tak obídeme fakt, že hodnota argumentu funkcie sa zabudne po jej skončení.

Pomocou smerníkovej premennej môžeme definovať taký vstupný argument (alebo viac argumentov), ktorý sa bude správať ako návratová hodnota. Program na mocniny môžeme prepísať tak, aby využívali smerníkovú vstupnú premennú na uloženie výsledku:

Navrhnmie si funkciu, ktorá načíta reťazec a premení ho na celé číslo. Ak bolo načítanie úspešné, funkcia vráti hodnotu nula a výsledok zapíše do vstupneho argumentu. Ak bolo načítanie neúspešné, funkcia vráti nenullvú hodnotu.

```
int nacitaj_cele_cislo(int* vysledok){
   printf("Zadajte celé číslo\n");
   char buffer[BUFSIZE];
   int r = fgets(buffer,100,stdin);
   if (r == NULL){
      printf("Načítanie sa nepodarilo.\n");
      return 1;
   }
   r = sscanf(buffer,"%d",vysledok);
   if (r != 1){
      printf("Zadaná hodnota nie je celé číslo.\n");
      return 2;
   }
   return 0;
}
```

Práve sme prišli na nový spôsob ako zapísať tú istú funkciu. Znak `&` pri argumente nám naznačuje, že funkcia má moc zapísať do premennej, ktorej adresu odovzdáme. Zápis do pamäťového miesta na ktoré ukazuje smerníková premenná `adresa_vysledku` sa deje s pomocou operátora `*`, ktorý sme využili na pravej strane výrazu. Ak vám volanie novej funkcie pripomína funkciu `scanf()`, nie je to náhoda. Návratová hodnota funkcie `scanf()`, ktorá sa odovzdáva kopírovaním sa používa na signalizovanie chybového stavu (načítanie do pamäťového miesta nemuselo prebehnúť v poriadku).

Keď máme hotovú funkciu na načítanie, program Kalkulačka sa stáva oveľa krajší:

```
int main(){
    printf("Súčtová kalkulačka\n");
    int a = 0;
    int b = 0;
    int r = 0;
    while (1){
// TODO -- ZLE
       r = nacitaj_cele_cislo(&a);
    }
    while (1){
       r = nacitaj_cele_cislo(&b);
    }
    vypis_sucet(a,b);
    return 0;
}
```

Otázka je, či je lepšie využiť návratovú hodnotu alebo smerníkový argument. O tom by mal rozhodnúť autor programu tak, aby funkcia bola dobre čitateľná a efektívna. Ako vodítko môťeme využiť tieto pravidlá:

- ak je výsledkom jediná hodnota, použijeme návratovú hodnotu.
- Návratová hodnota sa často používa na signalizovanie chybového stavu, napr. funkcia vrátu hodnotu 0 v prípade, že prebehla v poriadku alebo kód chyby ak nastala.

- V prípade, že funkcia má mať viac návratových hodnôt, použijeme smerníkové premenné.
- Ak je argument funkcie malý, odovzdávame ho klasicky kopírovaním pomocou návratovej hodnoty.
- Ak je argument funkcie veľký, použijeme smerníkovú premennú aby sme zjednodušili kopírovanie. Napr. reťazce sa väčšinou odovzdávajú iba pomocou smerníka, ktorý označuje začiatok reťazca.
- Pole vždy odovzdávame pomocou smerníkovej premennej ako adresu jeho začiatku.
- Ak dopredu nevieme, aký veľký bude argument, použijeme smerníkovú premennú a premennú s veľkosťou.

7.8. Úloha na precvičenie

- Navrhnite a napíšte funkciu, ktorá má viac návratových hodnôt

Chapter 8. Štruktúra bublikovej fólie v poli

Naučíte sa

- Pracovať so zložitejšími dátami a ukladať ich do polí v štruktúrovanej podobe.
- Triediť zoznam položiek podľa zvoleného atribútu.

Program v jazyku C vieme využiť aj na prácu so zložitejšími dátami ako sú celé alebo desatinné čísla. V tejto časti sa naučíme vytvoriť program na načítanie databázy a jej zotriedenie podľa zvolenej položky.

Prevtelíme sa do úlohy zamestnanca firmy pracujúcej v oblasti poľnohospodárstva. V našej biofarme sa profesionáne pestovaniu rôznych zdravých produktov na rozľahlých poliach. Od šéf asme práve dostali zadanie vytvoriť databázu poľnohospodárskych strojov v našej biofarme. Každý stoj má svoj opis a počet najazdených kilometrov.

8.1. Štruktúry

Využijeme svoju znalosť jazyka C a informácie o strojoch v garáži si uložíme do premenných. Prvou časťou návrhu je rozmýšľanie o tom, aké dátové typy budeme potrebovať. Pre každý stroj v našej garáži musíme evidovať jeho meno a počet najazdených kilometrov. Pre meno stroja sa javí najvýhpodnejší dátový typ reťazec maximálne 19 znakov (`char[10]`). Pre stav tachometra by mohla stačiť celočíslená premenná (`int`).

Vložme si všetky dostupné informácie o poľnohospodárkkych strojoch do krátkeho programu. Premenné si môžeme ľahko vypsovať na obrazovku, kedykoľvek si spustíme náš výtvor:

```
#include <stdio.h>
int main(){
    char meno1[10]="Traktor";
    int tachometer1 = 20000;

    char meno2[10]="Retro Traktor";
    int tachometer2 = 200000;

    char meno3[10]="Kombajn";
    int tachometer3 = 123456;

    printf("V garazi mame:\n");
    printf("%s, najazdenych %d km\n",meno1,tachometer1);
    printf("%s, najazdenych %d km\n",meno2,tachometer2);
    printf("%s, najazdenych %d km\n",meno3,tachometer3);

    return 0;
}
```

Program je veľmi pekný a funguje výborne. Takmer sme od šéfa za neho dostali pochvalu. Nanešťastie si spomenul, že by bolo fajn, keby sme zistili, ktorý stroj má toho najzdené najviac.

Samozrejme, dalo by sa to vyriešiť medódou pozriem - vidím, ale to nie je hodné pracovníka IT oddelenia farmy "Zdravý život". A je už vybavené, že naša farma dostane eurodotáciu, a dokúpine ďalších 70 strojov.

Informácie o jednom stroji by bolo dobré mať v jednej premennej, aby sme vedeli spojiť informácie o mene a počte najazdených kilometrov. Poznáme typ `int` vhodný na uloženie celých čísel a typ pole znakov `char[]`, ktoré je vhodné na uloženie mena. Nepoznáme ale typ `traktor` do ktorý by bol vhodný na uloženie informácií o traktoroch.

Našťastie si môžeme definovať vlastný dátový typ: Nový dátový typ je niečo návod na usporiadanie miesta v pamäti.

```
struct traktor {
    char meno[10];
    int tachometer;
};
```

Týmto zápisom sme si zadefinovali vlastný dátový typ, ktorý sa bude volať `struct traktor`. Pomocou štruktúry sme prekladaču vysvetlili, že pod pojopm "struct traktor" rozumeme dvojicu atribútov `meno` a `tachometer`. Tieto dva atribúty sa budú vyskytovať a spracovávať spolu.

V novom dátovom type definujeme druh "priehradiek", z ktorých sa bude skladať. Jednu priehradku nazývame "člen" štruktúry alebo "atribút". Člen štruktúry môže byť ľubovoľný existujúci dátový typ, dokonca aj iná štruktúra.

Vymyslite štruktúru vhodnú na uloženie informácií o kuchynských mixéroch.

8.2. Statická inicializácia

Pomocou štruktúry `struct traktor` sme vysvetli, ako vyzerá traktor vo všeobecnosti. Traktor v našom progame má zatiaľ iba tachometer a ceduľu s menovkou, ale nebude problém uvažovať aj nádrž, objem motora alebo počet kolies. Stále sa ale pohybujeme v rovine úvah o tom, aký druh strojov chceme mať v garáži.

Ak chceme predstvy premeniť na skutočnosť, musíme pristúpiť k ich realizácii - akvizícii strojovej techniky. V reálnom svete by to znamenalo transfer finančných prastriedkov smerom k predajcovi a trasnfer traktora smerom k nám. V počítačovom programe zase musíme vyhradiť pamäťové miesto, kde si uložíme informácie o novom stroji.

Definícia štruktúry sa podobá na definíciu funkcie. Ak definujeme funkciu, neznamené to, že sme ju vykonali - iba sme vysvetlili ako sa má vykonať. Na vykonanie je potrebný špeciálny príkaz na vykoannie funkcie. Podobne aj pri štruktúrach potrebujeme špeciálny príkaz na vytvorenie pamäťového miesta s dátovým typom štruktúra:

```
struct traktor zetor;
```

Tento príkaz vytvorí nové miesto pre uloženie informácií o traktore `zetor`.

Celý program kde zadefinujeme štruktúru `struct traktor` a vytvoríme jedno pamäťové miesto na uloženie infomrácií o jednom konkrétnom traktore bude vyzerať takto:

```
// Definícia nového dátového typu
struct traktor {
    char meno[10];
    int tachometer;
};

int main() {
    // Použitie nového dátového typu v novej premennej
    struct traktor zetor;
    return 0;
}
```

Pozor

Informácie o nových dátových typoch píšeme na rovnakú úroveň ako funkciu main(), nikdy nie do vnútra funkcie.

```
int main() {
    // Definícia nového dátového typu platí iba pre túto funkciu
    struct traktor {
        char meno[10];
        int tachometer;
    };

    // Použitie nového dátového typu v novej premennej
    struct traktor zetor;
    return 0;
}
```

Bolo by fajn, keby sme do novej premennej nového dátového typu vedeli uložiť nejaké informácie. Klasický postup s operátorom `=` asi fungovať nebude. Na rozlíšenie toho, ktorý atribút premennej chceme využívať použijeme operátor `.` .

```
zetor.tachmoter = 123;
```

Menší problém nastane pri pokuse o určenie mena nového traktora. Príkaz na priradenie reťazca nefunguje podľa očakávania:

```
zetor.meno = "Zetor 1000";
```

Zistite, aká hodnota najazdených kilometrov sa nachádza tesne po vytvorení premennej.

Namiesto toho môžme vykonať kopírovanie reťazce ručne. Využijeme fakt, že reťazec je vždy zakončený nulou.

```
int i = 0;
char meno[10] = "Zetor 1000";
while(meno[i] != 0){
    zetor.meno[i] = meno[i];
    i += 1;
}
zetor.meno[i] = 0;
```

môžME využiť funkciu zo štandardnej knižnice `strcpy` na skpírovanie reťazca, ktorá kopíruje reťazce. Musíme dbať na to, aby v mieste kde kopírujeme bolo dosť miesta. Pre použitie funkcie `strcpy` musíme priložiť hlavičkový súbor `<string.h>`.

```
strcpy(zetor.meno,"Zetor 1000");
```

8.2.1. Statická inicializácia štruktúry

Druhou možnosťou je alokácia a inicializácia v jednom kroku. O kopírovanie sa postará kompilátor. Na tento učel existuje špeciálny zápis, kotrý sa podobá na inicializáciu poľa. Musíme len špecifikovať mená členov, kotré inicializujeme:

```
struct traktor zetor = {.meno:"Zetor 1000",.tachometer:123};
```

tento zápis môžme použiť iba pri inicializácii premennej, nie v ďalšom programe.

Celý program bude vyzerať trochu inak. Hlavným rozdielom je to, že informácie o jednom poľnohospodáskom stroji sú uložené v jednej premennej.

```c
#include <stdio.h>
struct traktor {
    char meno[10];
    int tachometer;
}
int main(){
    struct traktor stroj1 = {.meno:"Traktor",.tachometer:123};
    struct traktor stroj2 = {.meno:"Retro Traktor",.tachometer:10000};
    struct traktor stroj3 = {.meno:"Kombajn",.tachometer:3454};

    printf("V garazi mame:\n");
    printf("%s, najazdenych %d km\n",stroj1.meno,stroj1.tachometer);
    printf("%s, najazdenych %d km\n",stroj2.meno,stroj2.tachometer);
    printf("%s, najazdenych %d km\n",stroj3.meno,stroj3.tachometer);
    return 0;
}
```

8.3. Polia štruktúr

Počet riadov programu ktorý využíva štruktúry sa takmer nezmenil a získali sme to, že informácie o jednom stroji sú v jednej premennej. Keďže sme firma, ktorá sa profesionálne venuje poliam, databázu traktorov bude uložená v poli.

```c
struct traktor stroje[3] = {
     {.meno:"Traktor",.tachometer:123},
     {.meno:"Retro Traktor",.tachometer:10000},
     {.meno:"Kombajn",.tachometer:3454}
};
```

Na prechádzanie poľa je špeciálne vhodný cyklus for. Vypís všetkých strojov v databáze je potom geniálne stručný:

```c
for (int i = 0; i < 3; i++) {
    printf("%s, najazdenych %d km\n",stroje[i].meno,stroje[i].tachometer);
}
```

Výsledný program bude takýto:

```c
#include <stdio.h>
struct traktor {
    char meno[10];
    int tachometer;
};
int main(){
    struct traktor stroje[3] = {
        {.meno:"Traktor",.tachometer:123},
        {.meno:"Retro Traktor",.tachometer:10000},
        {.meno:"Kombajn",.tachometer:3454}
    };
    printf("V garazi mame:\n");
    for (int i = 0; i < 3; i++) {
        printf("%s, najazdenych %d km\n",stroje[i].meno,stroje[i].tachometer);
    }
    return 0;
}
```

8.4. Typedef

Program môžeme "vylepšiť" ešte viac. Nemusíme stále vypisovať `struct traktor`. Pomocou zápisu `typedef` si môžeme definovať alias `traktor` pre názov typu `struct traktor`:

```c
typedef struct traktor {
    char meno[10];
    int tachometer;
} traktor;
```

Takto sme si vytvorili nový typ `traktor`. Zlepšená verzia môže vyzerať takto:

```c
#include <stdio.h>
#define POCET_STROJOV

typedef struct traktor {
    char meno[10];
    int tachometer;
} traktor;

int main(){
    traktor stroje[POCET_STROJOV] = {
        {.meno:"Traktor",.tachometer:123},
        {.meno:"Retro Traktor",.tachometer:10000},
        {.meno:"Kombajn",.tachometer:3454}
    };
    printf("V garazi mame:\n");
    for (int i = 0; i < POCET_STROJOV; i++) {
        printf("%s, najazdenych %d km\n",stroje[i].meno,stroje[i].tachometer);
    }
    return 0;
}
```

8.5. Bublinkové triedenie

Prvý krok sme zvládli - traktory sú úspešne na poli. Stačí iba ich presvedčiť nech sa zoradia podľa počtu najazdených kilometrov.

Milujeme bublinky. Každý má rád bublinky a tak na zotriedenie poľa použijeme "bublinkové triedenie".

Bublinky vo forme bublikového plastového obalu môžeme použiť aj na ukľudnenie nepokojnej nálady ich postupným praskaním.

Podstatou bublinkového triedenie je, že prvky poľa necháme "prebublávať" na koniec až pokiaľ sme si istý, že sú zoradené od najmenšieho po najväčší. Výsledkom prvého kola "prebublávania" bude to, že na konci určite bude najväčší prvok. V ďalšom kole už nemusíme "prebublávať" všetky prvy, ale môžeme vynechať ten na konci. Pokračujeme až spracujeme celé pole. "Prebublávanie" je realizované výmenou dvoch susediacich prvkov tak, aby ten väčší bol na pravo.

Jedno kolo prebublávania vyzerá takto:

```
for (int i=1; i < POCET_STROJOV; i++){
    // Ak je ma prvy stroj viac kilometrov
    if (stroje[i-1].tachometer > stroje[i].tachometer){
        // Vymenime ich tak, aby druhy stroj mal viac kilometrov
        traktor pomocny = stroje[i];
        stroje[i] = stroje[i-1];
        stroje[i-1] = pomocny;
    }
}
```

Na konci cyklu sme si istý, že na konci je stroj s najväčším počtom kilometrov. Aby sme zotriedili celé pole, stačí postup opakovať, ale na menšej oblasti poľa. Použijeme na to pomocnú premennú `j`, ktorá sa bude postupne zväčšovať.

```
for (int j=0; j < POCET_STROJOV -1; j++){
    for (int i=1; i < POCET_STROJOV -j ; i++){
        // Ak je ma prvy stroj viac kilometrov
        // Vymenime ich tak, aby druhy stroj mal viac kilometrov
    }
}
```

Finálny produkt bude:

```c
#include <stdio.h>
#define POCET_STROJOV

typedef struct traktor {
    char meno[10];
    int tachometer;
} traktor;

int main(){
    traktor stroje[POCET_STROJOV] = {
        {.meno:"Traktor",.tachometer:123},
        {.meno:"Retro Traktor",.tachometer:10000},
        {.meno:"Kombajn",.tachometer:3454}
    };
    for (int j=0; j < POCET_STROJOV -1; j++){
        for (int i=1; i < POCET_STROJOV -j ; i++){
            // Ak je ma prvy stroj viac kilometrov
            if (stroje[i-1].tachometer > stroje[i].tachometer){
                // Vymenime ich tak, aby druhy stroj mal viac kilometrov
                traktor pomocny = stroje[i];
                stroje[i] = stroje[i-1];
                stroje[i-1] = pomocny;
            }
        }
    }
    printf("V garazi mame:\n");
    for (int i = 0; i < POCET_STROJOV; i++) {
        printf("%s, najazdenych %d km\n",stroje[i].meno,stroje[i].tachometer);
    }
    return 0;
}
```

8.6. Úloha na precvičenie

Vyskúšajte si program pre väčšie množsto traktorv, kktorých parametri si vymyslite.

Modifikujte program pre triedenie iného druhu tovaru na sklade.