

GS Informatika

Daniel Rod

Jazyk C

- 1 O jazyku C
- 2 Hardware a principy počítačů
- 3 Kompilace programu v C (podrobněji)
- 4 Jazyk C - základy
- 5 Kontrolní struktury, pole
- 6 Ukazatele (pointery)

Jazyk C

- Low až Medium level jazyk
- Programování systémů (OS, embedded)
- Explicitní práce s pamětí
- ALGOL rodina jazyků

Jazyk C

- Low až Medium level jazyk
- Programování systémů (OS, embedded)
- Explicitní práce s pamětí
- ALGOL rodina jazyků

Kompilace

- Běžné jsou kompilované implementace
- Před spuštěním jej musíme převést do spustitelného souboru
- Velmi starý model kompilace - často vyžaduje explicitní deklarace a implementace
- Využití textových souborů pro psaní kódu

Typy souborů

- Hlavičkové soubory - běžně obsahují definice, přípona `.h`
- Zdrojové soubory - obsahují implementace, přípona `.c`
- Hlavičkové soubory nejsou nutností, hodí se ale pro
 - Organizaci
 - Modularitu
 - "Reusability"

Typy souborů

- Hlavičkové soubory - běžně obsahují definice, přípona `.h`
- Zdrojové soubory - obsahují implementace, přípona `.c`
- Hlavičkové soubory nejsou nutností, hodí se ale pro
 - Organizaci
 - Modularitu
 - "Reusability"

Definice a implementace

- Definice nám pouze říká co funkce zkonsumuje za datové typy a co nám za datový typ vrátí
- Pro malé programy stačí jen jeden zdrojový soubor, nemusíme nutně separovat implementaci a definice
- Starší kompilery mohou být hloupé - definice by měla předcházet použití (to jsme v BSL/ISL neměli!)

Definice a implementace

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int c = 1 << 3;
6      printf("%d\n", c);
7      int k = my_fn(5);
8      printf("%d", k);
9  }
10
11 int my_fn(int a) {
12     return a + 1;
13 }
```



```
<source>: In function 'main':
<source>:7:13: warning: implicit declaration of function 'my_fn' [-Wimplicit-function-declaration]
    7 |     int k = my_fn(5);
      |           ^~~~~~
      |           |
      |           ASM generation compiler returned: 0
<source>: In function 'main':
<source>:7:13: warning: implicit declaration of function 'my_fn' [-Wimplicit-function-declaration]
    7 |     int k = my_fn(5);
      |           ^~~~~~
      |           |
      |           Execution build compiler returned: 0
Program returned: 0
      8
      6
```

Syntax jazyka

- Jazyk C je procedurální - námi požadované operace se postupně provádějí, funkce jsou pak sady požadovaných operací
- Přiřazení hodnoty do proměnné je také operace!
- Validní identifikátory jsou omezenější oproti LISP/SCHEME dialektům
- Každá instrukce končí středníkem ;
- Kód je dělen na bloky instrukcí

Syntax jazyka

- Jazyk C je procedurální - námi požadované operace se postupně provádějí, funkce jsou pak sady požadovaných operací
- Přiřazení hodnoty do proměnné je také operace!
- Validní identifikátory jsou omezenější oproti LISP/SCHEME dialektům
- Každá instrukce končí středníkem ;
- Kód je dělen na bloky instrukcí

Program

- Každá program musí mít alespoň jednu funkci **main()**
- Při spuštění programu se provádí operace z funkce **main()**

První program

- Jednoduchý program který po spuštění vypíše *Program v jazyce C*, odřádkuje a ukončí se

První program

- Jednoduchý program který po spuštění vypíše *Program v jazyce C*, odřádkuje a ukončí se

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Program v jazyce C!\n");
6
7     return 0;
8 }
```

Jednoduchý program v C

První program

- Jednoduchý program který po spuštění vypíše *Program v jazyce C*, odřádkuje a ukončí se

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Program v jazyce C!\n");
6
7     return 0;
8 }
```

Kompilace

- Zdrojový soubor je nejprve zkompilován do tzv. objektového souboru (*s příponou .o*), kde se nachází *relativní adresy* na proměnné, volání funkcí a reference na funkce bez známé implementace

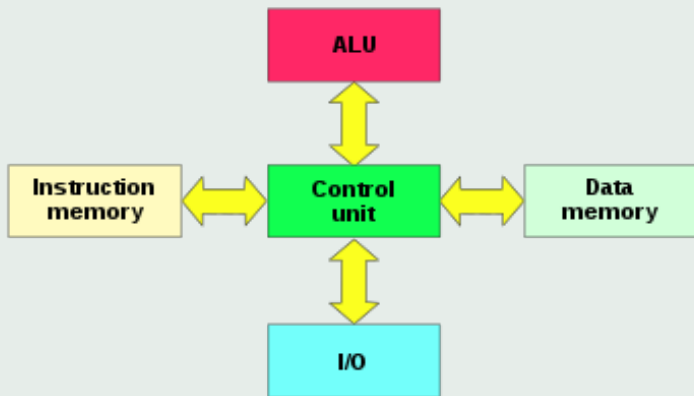
Od zdrojového souboru ke spustitelnému

- Zdrojový soubor *program01.c* je zkompilován pomocí kompilátoru (např. clang nebo gcc)

gcc program01.c

- To vytvoří nejprve objektový soubor, který následně převede na spustitelný soubor. Běžně nelze očekávat, že spustitelný soubor zkompilovaný na jednom počítači bude fungovat na jiném!

Zjednodušené schéma - Harvardská architektura



(ALU se dá brát jako součást Control Unit spolu s registry a counter)

Paměťový adresový prostor

- Data v paměti mají určitou lokaci - adresu
- Pokud používáme větší paměť (Instruction + Data), je třeba dostatečně velký adresový prostor
- Pro adresy o velikosti 32 bitů máme 4GB paměti které můžeme adresovat (x86)
- 64bitové procesory - používají datové jednotky co mají až 64 bitů - máme "k dispozici" adresy přesahující velikosti dnešních RAM

Paměťový adresový prostor

- Data v paměti mají určitou lokaci - adresu
- Pokud používáme větší paměť (Instruction + Data), je třeba dostatečně velký adresový prostor
- Pro adresy o velikosti 32 bitů máme 4GB paměti které můžeme adresovat (x86)
- 64bitové procesory - používají datové jednotky co mají až 64 bitů - máme "k dispozici" adresy přesahující velikosti dnešních RAM

RAM

- Random access memory - přistupujeme k libovolné hodnotě stejně rychle jako ke každé jiné (přibližně!)
- Bývá **volatile** - po vypnutí ztratí data

SRAM

- Static RAM
- Malá kapacita dat
- Rychlé - zejména sekvenční přístupy

SRAM

- Static RAM
- Malá kapacita dat
- Rychlé - zejména sekvenční přístupy

DRAM

- Dynamic RAM
- Levnější a větší (řádově GB)
- Pomalejší

WORD - jednotka velikosti

- Udává "jednotku přenosu" dat
- n-bitové zařízení - slovo má velikost n bitů
- Instrukce pracují s daty o velikosti slov
- Historicky - ve starém kódu se můžeme setkat s DWORD - 32b
- V x86 má například WORD 32b a adres space je 32b

Slovo (WORD) a instrukce

WORD - jednotka velikosti

- Udává "jednotku přenosu" dat
- n-bitové zařízení - slovo má velikost n bitů
- Instrukce pracují s daty o velikosti slov
- Historicky - ve starém kódu se můžeme setkat s DWORD - 32b
- V x86 má například WORD 32b a adres space je 32b

Instrukce

- Posloupnost n bytů
- Instrukční sada - které instrukce umí procesor (např. [základní známe x86 instrukce](#))
- Strojový kód - posloupnost instrukcí
- Instruction Pointer - pozice momentálně vykonávané instrukce (pokud je vícebytová typicky ukazuje na první byte)
- Zpravidla má stejnou velikost jako bloky code memory
- Opcode - typ instrukce, následují jej argumenty (adresy)

Vlajky(flags)

- flag je 1 bit informace (Ano / Ne)
- Příznakový registr pomocí flags uchovává informace o probíhajících výpočtech
- Ne všechny instrukce se dokončí v jednom *cyklu*
- Sčítání zabere 1-2 cykly (2 v případě velkých čísel, používá se carry flag - přenos z výsledku v předchozím cyklu a sign flag - jestli vyšel předchozí cyklus záporně)
- Násobení 32 bit čísel - 10 cyklů + carry flag + sign flag
- Násobení 64 bit čísel - 20 cyklů + carry flag + sign flag
- Dělení 32 bit čísel - 70 cyklů!

Instrukce čitelně

- Instrukce jsou posloupnost bytů (reprezentujeme dvojice jako HEX cifry) - špatně se čtou
- Assembler přiřazuje opcodes jména

```
1 #include <stdio.h>
2
3 int __attribute__((noinline)) add(int x1, int x2)
4 {
5     return x1 + x2;
6 }
7
8 int main(int argc, char **argv)
9 {
10     int c = add(1, 2);
11
12     printf("%d\n", c);
13 }
```

A ▾ Output ▾ Filter... ▾ Libraries + Add new... ▾ Add tool... ▾

```
printf@plt-0x10:
ff 35 e2 2f 00 00
401020 push 0x2fe2(%rip) # 404008 <_GLOBAL_OFFSET_TABLE_+0x8>
ff 25 e4 2f 00 00
401026 jmp *0x2fe4(%rip) # 404010 <_GLOBAL_OFFSET_TABLE_+0x10>
0f 1f 40 00
40102c nopl 0x0(%rax)
add:
8d 04 37
401126 lea (%rdi,%rsi,1),%eax
c3
401129 ret
main:
48 83 ec 08
40112a sub $0x8,%rsp
be 02 00 00 00
40112e mov $0x2,%esi
bf 01 00 00 00
401133 mov $0x1,%edi
e8 e9 ff ff ff
401138 call 401126 <add>
89 c6
40113d mov %eax,%esi
bf 04 20 40 00
40113f mov $0x402004,%edi
b8 00 00 00 00
401144 mov $0x0,%eax
e8 e2 fe ff ff
401149 call 401030 <printf@plt>
b8 00 00 00 00
40114e mov $0x0,%eax
48 83 c4 08
401153 add $0x8,%rsp
c3
401157 ret
0f 1f 84 00 00 00 00 00
401158 nopl 0x0(%rax,%rax,1)
```

Preprocessor

- Překlad maker - příkazy s prefixem `#`
- Vlastní makra - nahrazování "textu za text"
- Makra mohou mít argumenty

Preprocessor

- Překlad maker - příkazy s prefixem `#`
- Vlastní makra - nahrazování "textu za text"
- Makra mohou mít argumenty

Kompilátor

- V několika "passech" projede jednotlivé zdrojové soubory a vytvoří objektové soubory.
- Překládá námi napsané příkazy na instrukce
- Pracuje se soubory separátně - proto potřebujeme hlavičkové soubory! Soubor `prg1.c` neví nic o implementaci funkce z `prg2.c`, jen víme že je deklarovaná v `prg2.c`

Preprocessor

- Překlad maker - příkazy s prefixem `#`
- Vlastní makra - nahrazování "textu za text"
- Makra mohou mít argumenty

Kompilátor

- V několika "passech" projede jednotlivé zdrojové soubory a vytvoří objektové soubory.
- Překládá námi napsané příkazy na instrukce
- Pracuje se soubory separátně - proto potřebujeme hlavičkové soubory! Soubor `prg1.c` neví nic o implementaci funkce z `prg2.c`, jen víme že je deklarovaná v `prg2.c`

Linker

- Propojí jednotlivé objektové soubory - pokud volám z `prg1.c` funkci v `prg2.c`, až po projetí linkerem bude toto volání "funkční"
- Zajistí přiřazení objektových souborů referencovaných knihoven (BSL/ISL require)

Základní datové typy

Číselné datové typy

- Ze začátku budeme pracovat zejména s čísly
- C je striktně typovaný - každá proměnná musí mít deklarovaný typ
- signed / unsigned typy - určuje jestli mají "znaménko", signed je default
- Velikost v paměti **závisí na implementaci!** Zjistíme pomocí *sizeof(T)*

Celá čísla

- short
- int
- long
- long long

Desetinná čísla

- float
- double
- long double

Dodatek - char

- Char je nejmenší číselný typ, běžně se ale používá pro **ukládání textu** (jako 1-String)

Speciální typy

- `size_t` - unsigned typ běžně používaný pro "velikost" - hodnoty mají max velikost odpovídající maximální velikosti "objektů"
- `intptr_t` - (`#include <stdint.h>`) unsigned typ do kterého lze uložit validní pointer, používá se při pointer aritmetice (bude nás zajmat až později)

Ukázka C programu

```
1 #include <stdio.h> /* odkaz na hlavickovy soubor */
2 #define NUMBER 5 /* symbolicka konstanta - makro */
3
4 int compute(int a); /* deklarace funkce (hlavicka/prototyp) */
5 /* Funkce bere jeden argument typu "int" a vraci hodnotu typu "int" */
6
7 int main(int argc, char *argv[])
8 { /* main funkce */
9     int v = 10; /* deklarace promenne */
10    int r;
11    r = compute(v); /* volani funkce */
12    return 0; /* konec main funkce - vraci hodnotu 0 */
13 }
14
15 int compute(int a)
16 { /* implementace deklarovane funkce (definice) */
17     int b = 10 + a; /* telo funkce (body) */
18     return b; /* funkce vraci hodnotu 'b' */
19 }
```

Deklarace

- Deklarace obsahuje jen hlavičku funkce - jméno funkce, jaké parametry (a jakého typu) funkce má a jaký typ proměnné vrací
- Deklarace nejsou povinné, je ale vhodné funkce deklarovat před použitím (při čtení kódu "od shora")

```
1 float probability(int num_dice, int min_number_count);
```

Deklarace

- Deklarace obsahuje jen hlavičku funkce - jméno funkce, jaké parametry (a jakého typu) funkce má a jaký typ proměnné vrací
- Deklarace nejsou povinné, je ale vhodné funkce deklarovat před použitím (při čtení kódu "od shora")

```
1 float probability(int num_dice, int min_number_count);
```

Definice

- Zavádí implementaci funkce - říká, jak funkce procedurálně postupuje a jak dosáhne výsledku který může vrátit
- Uvnitř funkce máme implicitně local prostředí - můžeme zavádět proměnné, které budou "existovat" jen v rámci běhu funkce
- Nelze mít "funkci ve funkci"

Deklarace a definice funkce

```
1 float probability(int num_dice, int min_number_count)
2 {
3     float one_dice_prob = 1.0 / 6.0; /* Zavedeni promenne one_dice_prob typu
4     float */
5     float prob = one_dice_prob * min_number_count; /* Zavedeni promenne prob
6     */
7     return prob; /* Vraceni hodnoty ulozene v promenne prob */
8 }
```

Deklarace a definice funkce

```
1 float probability(int num_dice, int min_number_count)
2 {
3     float one_dice_prob = 1.0 / 6.0; /* Zavedeni promenne one_dice_prob typu
4     float */
5     float prob = one_dice_prob * min_number_count; /* Zavedeni promenne prob
6     */
7     return prob; /* Vraceni hodnoty ulozene v promenne prob */
8 }
```

Funkce co nic nevrací

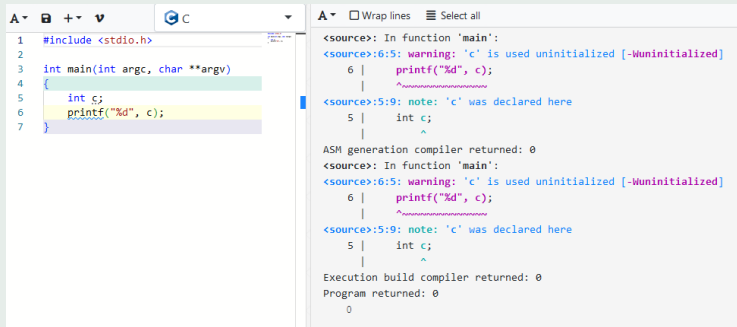
- V některých případech chceme, aby funkce nic nevracela
- Návratový "typ" **void** - funkce nevrací žádnou hodnotu (void znamená "žádný typ")
- Např. při vypisování

```
1 void print_probability(int num_dice, int min_number_count)
2 {
3     /* Zavedeni promenne prob typu float a ulozeni hodnoty kterou vraci fce
4     probability */
5     float prob = probability(num_dice, min_number_count);
6     printf("\nPravdepodobnost je: %f\n", prob); /* Vypsani hodnoty ulozene v
7     promenne prob */
8     /* Nevracime nic! */
9 }
```


DEMO

Deklarace

- Proměnné můžeme nejprve deklarovat, kompilér pak ví že má vyhradit místo v paměti pro tuto proměnnou
- Deklarovaná proměnná má tedy *adresu*, ale nemá "explicitní" hodnotu.



The screenshot shows a code editor with two panes. The left pane displays C code, and the right pane shows the compiler's output.

Left Pane (Code):

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int c;
6     printf("%d", c);
7 }
```

Right Pane (Compiler Output):

```
<source>: In function 'main':
<source>:6:5: warning: 'c' is used uninitialized [-Wuninitialized]
    6 |     printf("%d", c);
      |                   ^
<source>:5:9: note: 'c' was declared here
    5 |     int c;
      |         ^

ASM generation compiler returned: 0
<source>: In function 'main':
<source>:6:5: warning: 'c' is used uninitialized [-Wuninitialized]
    6 |     printf("%d", c);
      |                   ^
<source>:5:9: note: 'c' was declared here
    5 |     int c;
      |         ^

Execution build compiler returned: 0
Program returned: 0
0
```

Přřazení (assignment) a mutace / reassignment

- Deklarované proměnné můžeme přiřadit hodnotu (nebo provést zároveň deklaraci a přiřazení)
- Do paměti vyhrazené pro proměnnou se uloží data
- Proměnná je v C tzv. l-value -> má pevně stanovenou adresu!
- Data na této adrese můžeme upravit -> měníme hodnotu proměnné!

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int integer1; /* Deklarace */
6     integer1 = 5; /* Inicializace (prirazeni) */
7     int integer2 = 10; /* Deklarace a inicializace */
8     int integer3, integer4 = 20; /* Deklarace a inicializace */
9     integer2 = 400; /* Reassignment */
10
11     integer3 = integer1 + integer2 + integer3 + integer4; /* Reassignment */
12     return 0;
13 }
```

l-value

- Výraz který se po vyhodnocení odkazuje na **paměť**
- Může být na levé i pravé straně přiřazovacího operátoru =

r-value

- Výraz který se neodkazuje na žádnou adresu
- Jen na pravé straně přiřazovacího operátoru =
- Např. konstanta

Komentáře

- Hlavička funkce obsahuje informace o typech
- Stále je vhodné popsat co funkce dělá, případně jak
- Řádkové komentáře pomocí //
- Komentáře pomocí /* */

```
1 /*  
2 Tato funkce dela neco s parametry.  
3 Tento komentar je na vice radcich.  
4 */  
5 void fn(int a, int b)  
6 {  
7     // Do stuff - radkovy komentar  
8 }
```

Funkce a proměnné

Cílem je spočítat počet zrněk kávy na šachovnici podle následujícího vzoru: Na prvním čtverci je 1 zrnko, na druhém 2, na každém dalším pak dvojnásobek.

- 1 Deklarujte proměnnou *square_count* typu *int* v globálním scope.
- 2 Deklarujte funkci *square_grains* s návratovým typem *int* a jedním argumentem *square_number* typu *int*, funkci zatím neimplementujte.
- 3 Deklarujte funkci *total_grains* s návratovým typem *int* a jedním argumentem *total_squares* typu *int*, funkci zatím neimplementujte.
- 4 Implementujte funkci *main*, která spočítá počet zrněk na šachovnici s počtem polí *square_count* a vypíše tento počet. Funkce *main* následně vrátí hodnotu *"success"* - číslo 0. (Hint: pro vypsání použijte funkci *printf("%d\n", ...)*; z knihovny *stdio*)

Náš program zatím nejde zkompileovat, ale je korektní - implementace funkcí by totiž mohla klidně být v jiném zdrojovém souboru - chybu dostaneme až při kompilaci, kdy implementaci neposkytneme! Pojdme trochu prozkoumat naše data

- 1 Kolik polí má typická šachovnice? Upravte podle této znalosti typy proměnných a argumentů funkcí. (Hint: *int* je pro naše účely zbytečně "velký")
- 2 Nalezněte v kódu alespoň jednu l-value a dvě r-value.

Viditelnost a klíčové slovo extern

Když deklarujeme funkci (jako třeba v předchozím cvičení), C automaticky přidává klíčové slovo *extern*, které kompileru říká, že může implementaci najít jinde. Až linker nás zastaví a vyhodí chybu pokud takto deklarovanou funkci nenajde v žádných knihovnách použitých při kompilaci.

Modifikátor *extern* lze použít i na proměnné, tomu se ale zatím věnovat nebudeme.

Rozhodování - if

- Při běhu programu je třeba rozhodovat o hodnotách a podle toho vyhodnocovat různé větve logiky. K tomu může sloužit klíčové slovo *if*.

```
1 int main() {  
2     int my_value = 5;  
3     if (my_value == 5) {  
4         printf("my_value is 5\n");  
5     }  
6  
7     if (my_value != 7) {  
8         printf("my_value is not 7\n");  
9     }  
10  
11    if (my_value > 4) {  
12        printf("my_value is greater than 4\n");  
13    }  
14  
15    return 0;  
16 }
```

Rozhodněte

Co bude na výstupu tohoto kódu?

Rozhodování - else

- Kód se často větví na dvě možnosti, pak můžeme použít "if-else" přístup

```
1 int main()
2 {
3     int my_value;
4     scanf("%d", &my_value); // Nacteni hodnoty ze stdin
5
6     if (my_value == 5) {
7         printf("my_value is 5\n");
8     }
9     else {
10        printf("my_value is not 5\n");
11    }
12
13 }
```

Early return

- Pokud je to ale možné, je vhodnější tzv. early return přístup
- Rozhodování je separováno do samostatné funkce, nepoužíváme else ale při splnění podmínky rovnou vracíme
- Kód je pak více "lineární pro oči"

```
1 int main()
2 {
3     int my_value;
4     scanf("%d", &my_value); // Nacteni hodnoty ze stdin
5
6     if (my_value == 5) {
7         printf("my_value is 5\n");
8         return 0;
9     }
10
11     printf("my_value is not 5\n");
12     return 0;
13 }
```

Rozhodování - else if

- Pokud potřebujeme rozlišit mezi několika možnostmi které se vylučují, lze použít *else if* strukturu
- Opět většinou lze nahradit early returnem

```
1  if (my_value == 5) {  
2      printf("my_value is 5\n");  
3  }  
4  
5  else if (my_value == 7) {  
6      printf("my_value is 7\n");  
7  }  
8  
9  else if (my_value == 9) {  
10     printf("my_value is 9\n");  
11 }  
12  
13 else {  
14     printf("my_value is not 5, 7 or 9\n");  
15 }
```

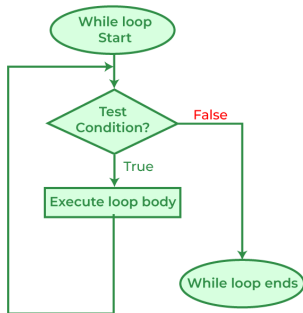
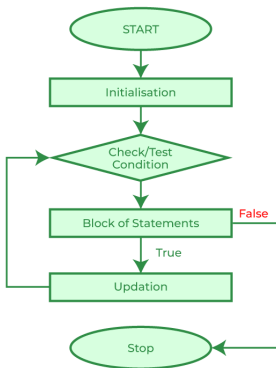
Rozhodování - switch

- Pokud rozhodování zakládáme na nějaké hodnotě integer nebo char hodnoty, je možné použít switch statement
- Často se přeloží do ASM jinak než if-else if-else bloky (jumptables - efektivnější)

```
1 int main()
2 {
3     int my_value;
4     scanf("%d", &my_value); // Nacteni hodnoty ze stdin
5
6     switch (my_value)
7     {
8     case 5:
9         printf("my_value is 5\n");
10        break;
11    case 6:
12    case 7:
13        printf("my_value is 6 or 7\n");
14        break;
15    default:
16        printf("my_value is not 5, 6 or 7\n");
17        break;
18    }
19    return 0;
20 }
```

Smyčky (loops)

- Pro potřeby opakování nějakého algoritmu používáme smyčky for a while



```
1 #include <stdio.h>
2
3 int main()
4 {
5     size_t count;
6     scanf("%d", &count); // Nacteni hodnoty ze stdin
7
8     // Pocatecni hodnota i; podminka cyklu; zmena promenne i po kazde iteraci
9     for (size_t i = 0; i < count; i++) {
10         printf("%d\n", i);
11     }
12
13     // Pouze podminka cyklu
14     while (count > 0) {
15         printf("%d\n", count);
16         count = count - 1;
17     }
18 }
```

Funkce a proměnné

Implementujte funkce z předchozího cvičení (*square_grains* a *total_grains*).

- 1 Určete jak musí jednotlivé funkce "postupovat"
- 2 Pomocí kontrolní struktur proveďte implementaci těchto postupů a otestujte
- 3 Výpočet lze zjednodušit použitím správných matematických funkcí. O jaké funkce se jedná? Nalezněte je v C/C++ [referenci](#)

Více dat v jedné proměnné

- Stejně jako v BSL/ISL, běžně potřebujeme pracovat s proměnnou s více hodnotami za sebou - s listem, v C máme **pole**
- V poli jsou hodnoty pouze jednoho typu, označíme jej pomocí [] za názvem proměnné
- Hodnoty jsou v paměti uloženy přímo za sebe
- Pozor na předávání do funkce! Předává se jako tzv. pointer (ukážeme si dále)! Musíme předat velikost pole jako parametr

```
1 #include <stdio.h>
2 int main()
3 {
4     #define SIZE 5
5     int x[SIZE]; // Deklarace pole se SIZE prvky
6
7     int y[] = { 3, 9, 27, 81, 243 }; // Deklarace pole s inicializací
8     int z[5] = {1, 2}; // Deklarace pole s částečnou inicializací
9
10    for (size_t index = 0; index < SIZE; index++) {
11        x[index] = y[index] * 2; // Prirazení do pole a přístup k prvku pole
12    }
13 }
```



```

1 #include <stdlib.h>
2 int sum(int arr[], size_t array_size)
3 {
4     int sum = 0;
5     for (size_t i = 0; i < array_size; i++) {
6         sum += arr[i];
7     }
8     return sum;
9 }

```

```

1 #include <stdio.h>
2 void fn(int arr[])
3 {
4     printf("Inside function: %d\n", sizeof(arr));
5 }
6
7 int main() {
8     int a[] = {1, 2, 3, 4, 5};
9     printf("Outside function: %d\n", sizeof(a));
10    fn(a);
11 }

```

```

<source>: In function 'fn':
<source>:4:13: warning: 'sizeof' on array function parameter 'arr' will return size of 'int *' [-Wsizeof-array-argument]
4 |     printf("Inside function: %d\n", sizeof(arr));
  |                                     ^
<source>:2:13: note: declared here
2 | void fn(int arr[])
  |          ~~~~~~
ASM generation compiler returned: 0
<source>: In function 'fn':
<source>:4:13: warning: 'sizeof' on array function parameter 'arr' will return size of 'int *' [-Wsizeof-array-argument]
4 |     printf("Inside function: %d\n", sizeof(arr));
  |                                     ^
<source>:2:13: note: declared here
2 | void fn(int arr[])
  |          ~~~~~~
Execution build compiler returned: 0
Program returned: 0
Outside function: 20
Inside function: 8

```

Pointer

- l-values mají místo v paměti
- Pomocí operátoru & lze obdržet adresu proměnné
- Tento operátor lze použít jen na l-values!

```
1 int x = 0;
2 short y = 0;
3
4 int* a = &x; // a je ukazatel na x
5
6 short* b = &y; // b je ukazatel na y
7
8 int* c = &12; // chyba - nelze adresovat r-value
```

Dereference

- Pointer je efektivně proměnná ukazující na oblast adresového prostoru
- Typ pointeru nám pak udává na "jak velkou oblast" ukazujeme
- Využíváme při interpretaci hodnoty na adrese
- Hodnotu uloženou na adrese dostaneme pomocí dereference pointeru

```
1 #include <stdio.h>
2 int main()
3 {
4     int x = 10;
5     int* px = &x; // px je pointer na adresu proměnné x
6     printf("Adresa: %x\n", px); // v px je uložená adresa
7     printf("Hodonota: %d\n", (*px)); // dereference px
8     // dereferencí dostáváme hodnotu na adrese pointeru
9 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
Adresa: 60b40394
Hodonota: 10
```

Intermezzo: Type casting

Změna typu

- V některých případech potřebujeme změnit interpretaci (typ) hodnoty uložené v paměti
- Např. dostaneme z nějaké funkce integer a víme že nepřekročí číslo 255, chceme ho tedy převést do charu *char*
- Dělení čísel

```
1 int extern foreign_fn(int, int);
2 int main() {
3     int result = foreign_fn(1, 2);
4     char c = (char) result;
5     return 0;
6 }
```

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 21;
5     int b = 8;
6     float f1 = a / b;
7     float f2 = (float) a / (float) b;
8     float f3 = (float) a / b;
9     printf("Bez castu: %f\n", f1);
10    printf("S castem obou: %f\n", f2);
11    printf("S castem jednoho: %f\n", f3);
12 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0

Bez castu: 2.000000
S castem obou: 2.625000
S castem jednoho: 2.625000
```

Cast pointeru

- Můžeme změnit i typ pointeru
- Např. při castu z *int* na *char* říkáme, že při dereferenci máme s obsahem paměti na dané adrese nakládat jako s charem

```
1 #include <stdio.h>
2 int main()
3 {
4     int x = 999999999;
5     int* px = &x; // px je pointer na adresu proměnné x
6     unsigned short* px_cast = (unsigned short*) px;
7     printf("Adresa: %x\n", px); // v px je uložená adresa
8     printf("Hodonota: %d\n", *px); // dereference px
9     printf("Hodonota short: %d\n", *px_cast);
10    // dereferencí dostáváme hodnotu na adrese pointeru
11 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
Adresa: d63814cc
Hodonota: 999999999
Hodonota short: 51711
```

Implicitní typecast

- Předchozí ukázky byly explicitní type cast
- Často není třeba - implicitní typecast při rozšiřování

```
1 int extern foreign_fn(int, int);
2 int main() {
3     short a = 21;
4     short b = 25;
5     int c = foreign_fn(a, b); // a, b implicitne pretypovano na int
6 }
```

Volání funkcí - pass by value

- Při předávání parametru funkci dochází k předání pomocí "pass by value"
- Předáváme hodnotu jako r-value (v lokálním scope funkce vytváříme kopii dat které do ní přichází jako parametry)

```
1  #include <stdio.h>
2  void not_modified(int a) {
3      a = 0;
4  }
5
6  int main() {
7      int a = 1;
8      not_modified(a);
9      printf("a: %d\n", a);
10     return 0;
11 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
a: 1
```

Volání funkcí - pass by value

- Při předávání parametru funkci dochází k předání pomocí "pass by value"
- Předáváme hodnotu jako r-value (v lokálním scope funkce vytváříme kopii dat které do ní přichází jako parametry)

```
1  #include <stdio.h>
2  void not_modified(int a) {
3      a = 0;
4  }
5
6  int main() {
7      int a = 1;
8      not_modified(a);
9      printf("a: %d\n", a);
10     return 0;
11 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
a: 1
```

Modifikace z funkce

- Jak vyřešit když potřebujeme modifikovat vnější data uvnitř funkce?

Modifikace z funkce

- Funkcionální přístup - všechny funkce budou *pure* (problém při velkém objemu dat)
- Globální scope - modifikované proměnné budou žít v globálním scope (problém s nepřehledností kódu, modifikovatelný globální stav ve větších programech přináší problémy)
- Využijeme pointery!

Modifikace z funkce

- Funkcionální přístup - všechny funkce budou *pure* (problém při velkém objemu dat)
- Globální scope - modifikované proměnné budou žít v globálním scope (problém s nepřehledností kódu, modifikovatelný globální stav ve větších programech přináší problémy)
- Využijeme pointery!

Přřazení dereferencovanému pointeru

- Dereference pointeru může být i na levé straně přiřazení!

```
1 #include <stdio.h>
2 int main() {
3     int x = 12;
4     int *px = &x; // px je pointer na x
5     *px = 42; // přiřazení dereferenci
6     printf("x: %d\n", x);
7 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
x: 42
```

Modifikace z funkce

- Místo hodnoty bude argumentem funkce pointer na danou hodnotu
- Hodnotu dostaneme dereferencí
- Hodnotu upravíme přiřazením dereferenci

```
1 #include <stdio.h>
2
3 void swap(int *a, int *b) {
4     int tmp = *a; // dereference do nove promenne (copy)
5     *a = *b; // prirazeni hodnoty na lokaci b do lokace a
6     *b = tmp; // prirazeni hodnoty tmp do lokace b
7 }
8
9 int main() {
10     int x = 10;
11     int y = 20;
12     swap(&x, &y); // do funkce posilame pointery (adresy)
13     printf("%d\n", x); // 20
14     printf("%d\n", y); // 10
15 }
```

Pointer vs array

TBD ...