

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)
- 7 Struktury
- 8 Dynamická alokace paměti

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 je nutné 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 zkonzumuje 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!)

Definire e implementare

```
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 }
```

main.c

```
<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

- Každý 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

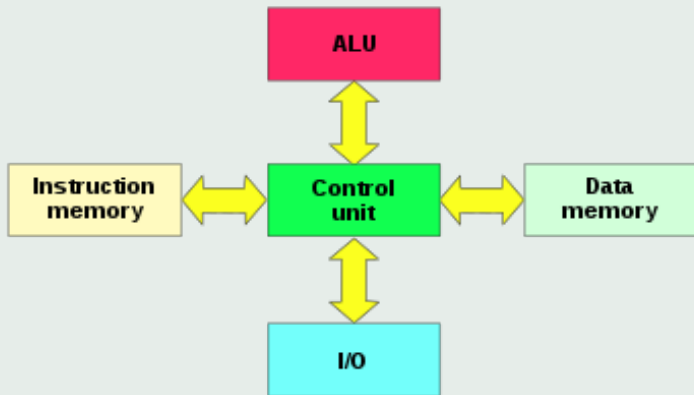
Od zdrojového souboru ke spustitelnému

- 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 číselně

- 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 35 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 hlavičkový soubor */
2 #define NUMBER 5 /* symbolická konstanta - makro */
3
4 int compute(int a); /* deklarace funkce (hlavička/prototyp) */
5 /* Funkce bere jeden argument typu "int" a vrácí hodnotu typu "int" */
6
7 int main(int argc, char *argv[])
8 { /* main funkce */
9     int v = 10; /* deklarace proměnné */
10    int r;
11    r = compute(v); /* volání funkce */
12    return 0; /* konec main funkce - vrácí hodnotu 0 */
13 }
14
15 int compute(int a)
16 { /* implementace deklarované funkce (definice) */
17     int b = 10 + a; /* tělo funkce (body) */
18     return b; /* funkce vrácí 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

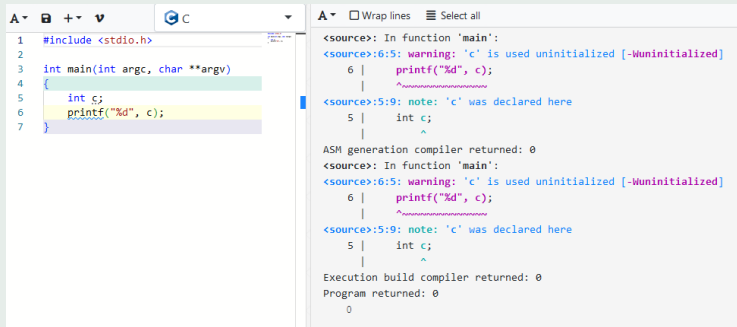
- V některých případech chceme, aby funkce nic nevracela
- Návrátový "typ" **void** - funkce nevrací žádnou hodnotu (void znamená "žádný typ")
- Např. při vypisování

```
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 displays a code editor with two panels. The left panel shows the source code of a C program, and the right panel shows the compiler's output, including warnings and notes.

Source Code (Left Panel):

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

Compiler Output (Right Panel):

```
<source>: In function 'main':
<source>:6:5: warning: 'c' is used uninitialized [-Wuninitialized]
    6 |     printf("%d", c);
      |           ^
      |           ~~~~~
      |           |
      |           v
<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);
      |           ^
      |           ~~~~~
      |           |
      |           v
<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 (přirazení) */
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 }
```

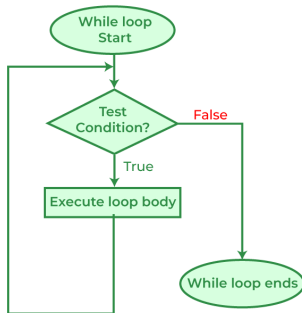
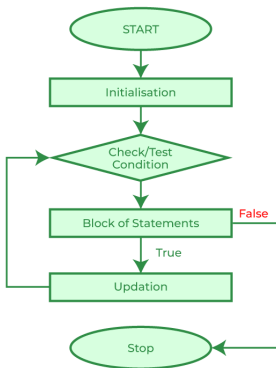
Rozhodování - switch

- 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 inicializaci
8     int z[5] = {1, 2}; // Deklarace pole s castecnou inicializaci
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>:4:43: 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>:4:43: 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

```

Ukazatele (pointery)

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 uložena 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 }
```

Konstanta

- Konstanta je hodnota která se za běhu programu nikdy nezmění
- Deklarujeme pomocí klíčového slova *const*

```
1 int main() {  
2     const int x = 5;  
3     x = 6; // error - lvalue není 'modifiable'  
4     return 0;  
5 }
```

Konstantní hodnoty

Konstanta

- Konstanta je hodnota která se za běhu programu nikdy nezmění
- Deklarujeme pomocí klíčového slova *const*

```
1 int main() {  
2     const int x = 5;  
3     x = 6; // error - lvalue není 'modifiable'  
4     return 0;  
5 }
```

Pointer na const

- Musíme deklarovat že je pointer na konstantu!
- Jinak může dojít k přepsání konstanty (!)

```
1 int main() {  
2     const int x = 5;  
3     int * px_mod = &x;  
4     const int * px_const = &x;  
5     *px_mod = 7; // bez erroru!  
6     *px_const = 9; // error  
7 }
```

Pointer na const

- `const int * identifier` deklaruje pointer na konstantní část paměti
- Lze použít i pro "zakázání" mutability uvnitř funkce!

```
1 int copy(const int * source, int * destination) {  
2     // Uvnitř funkce nelze menit hodnotu na adrese a  
3     *destination = *source;  
4     *source = 5; //error  
5 }  
6  
7 int main() {  
8     int a = 5; // Není deklarováno jako const  
9     int b;  
10    copy(a, b); // Funkce se k proměnné 'a' ale bude  
11                 // chovat jako ke konstantě  
12                 // a máme "jistotu" že zůstane nezměněna  
13 }
```

Konstantní pointer

- Pointer je také proměnná, která se může reassignovat!
- Můžeme deklarovat konstantní pointer - klíčové slovo `const` až za `"*"`
- `const int *` čteme jako "pointer na konstantní `int`"
- `int * const` čteme jako "konstantní pointer na `int`"
- Vše před hvězdičkou udává na jaký typ ukazujeme

```
1 int main() {  
2     int x[2] = {10, 20};  
3     int *x_ptr = x; // cast z array do pointeru (vysvetleno dale)  
4     printf("%i\n", *x_ptr); // 10  
5     x_ptr += 1; // modifikace lokace  
6     // (ne hodnoty na lokaci! není dereference!)  
7     printf("%i\n", *x_ptr); // 20;  
8     int * const x_cptra = x;  
9     x_cptra += 1; //error  
10 }
```

Pointer vs array

- Pointer je ukazatel na lokaci v paměti + informaci o datech uložených
- Těchto dat ale může být několik za sebou!
- Array je technicky ukazatel na první prvek v paměti (+ informace o celkové velikosti - sizeof)
- Array je *const* - hodnotu ukazatele **nelze** měnit (nelze "pohnout" s adresou)
- Array v hlavičce funkce je ale jen pointer, array také můžeme přiřazením převést na pointer

Inkrementace a dekrementace pointerů

- Nekonstantní pointer může měnit hodnotu
- Přičtením/odečtením měníme adresu na kterou ukazujeme
- Posouváme se v paměti o délku typu na který pointer ukazuje

Increment int pointer

```
1 #include<stdio.h>
2 int main() {
3     int x[2] = {10, 20};
4     int *x_ptr = x;
5     printf("%p\n", x_ptr);
6     x_ptr += 1;
7     printf("%p\n", x_ptr);
8 }
```

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
0x7ffff95db000
0x7ffff95db004

Increment long pointer

```
1 #include<stdio.h>
2 int main() {
3     long x[2] = {10, 20};
4     long *x_ptr = x;
5     printf("%p\n", x_ptr);
6     x_ptr += 1;
7     printf("%p\n", x_ptr);
8 }
```

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
0x7fff76c41d00
0x7fff76c41d08

Increment int pointer

```
1 #include<stdio.h>
2 int main() {
3     int x[2] = {10, 20};
4     int *x_ptr = x;
5     printf("%p\n", x_ptr);
6     x_ptr += 1;
7     printf("%p\n", x_ptr);
8 }
```

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
0x7ffff95db000
0x7ffff95db004

Increment long pointer

```
1 #include<stdio.h>
2 int main() {
3     long x[2] = {10, 20};
4     long *x_ptr = x;
5     printf("%p\n", x_ptr);
6     x_ptr += 1;
7     printf("%p\n", x_ptr);
8 }
```

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
0x7fff76c41d00
0x7fff76c41d08

Adresa se změnila vždy o délku (sizeof) typu!

Intermezzo: Post/Pre increment/decrement

Pre-increment/decrement

- Proveďte úpravu dat v paměti (přičtení/odečtení 1)
- Poté vrátí již upravenou hodnotu
- *++identifier / --identifier*

Post-increment/decrement

- Nejprve se vyhodnotí jako momentální hodnota v paměti
- Poté se provede úprava dat v paměti (přičtení/odečtení 1)
- *(identifier++) / identifier--*

```
1 #include <stdio.h>
2 int main() {
3     int i = 0;
4     while(++i < 3)
5         printf("%i", i); //12
6     i = 0;
7     printf("\n");
8     while(i++ < 3)
9         printf("%i", i); // 123
10 }
```

Porovnávání pointerů

- Pointery můžeme také porovnávat - má smysl při práci s jednou oblastí paměti (bounds checking)

Aritmetika pointerů

```
1 #include <stdio.h>
2 // Promenne urcujici ktere vypocty provest
3 short cmp1 = 1, cmp2 = 1, cmp3 = 1;
4
5 int compute1() {return 1;}
6 int compute2() {return 2;}
7 int compute3() {return 3;}
8
9 int main() {
10     long results[3];
11     long* head = results;
12     if (cmp1) *head++ = compute1();
13     if (cmp2) *head++ = compute2();
14     if (cmp3) *head++ = compute3();
15     // Ukazuje na zacatek array
16     long* tail = results;
17     while(tail < head) {
18         printf("%ld\n", *tail);
19         tail++;
20     }
21 }
```

C String

- Pro textová data se v C používá primárně *char* typ
- *char* obsahuje jeden znak (ASCII) - "něco jako 1-String"
- Textový řetězec je reprezentován typem *char array*, resp. *char **
- Jak program pozná kde string končí? Musíme předávat parametr o délce?

C String

- Pro textová data se v C používá primárně *char* typ
- *char* obsahuje jeden znak (ASCII) - "něco jako 1-String"
- Textový řetězec je reprezentován typem *char array*, resp. *char **
- Jak program pozná kde string končí? Musíme předávat parametr o délce?
- Každý C String je ukončen speciálním znakem s ASCII hodnotou 0.
- *Null-terminated byte string*
- Když má tedy C string 6 znaků, v paměti je vyhrazena oblast o 1 větší!

```
1 char letter = 'A'; // pismeno (1-String, ASCII)
2 char *firstname = "Jan"; // C string jako pointer
3 char surname[] = "Novak"; // C string jako array
```


Textová data

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|-----|----|-----|------------------------------------|-----|----|-----|-------|--------------|-----|----|-----|-------|----------|-----|----|-----|--------|------------|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 | ##32; | Space | 64 | 40 | 100 | ##64; | @ | 96 | 60 | 140 | ##96; | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ##33; | ! | 65 | 41 | 101 | ##65; | A | 97 | 61 | 141 | ##97; | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | ##34; | " | 66 | 42 | 102 | ##66; | B | 98 | 62 | 142 | ##98; | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | ##35; | # | 67 | 43 | 103 | ##67; | C | 99 | 63 | 143 | ##99; | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | ##36; | \$ | 68 | 44 | 104 | ##68; | D | 100 | 64 | 144 | ##100; | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | ##37; | % | 69 | 45 | 105 | ##69; | E | 101 | 65 | 145 | ##101; | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | ##38; | & | 70 | 46 | 106 | ##70; | F | 102 | 66 | 146 | ##102; | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ##39; | ' | 71 | 47 | 107 | ##71; | G | 103 | 67 | 147 | ##103; | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ##40; | (| 72 | 48 | 110 | ##72; | H | 104 | 68 | 150 | ##104; | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ##41; |) | 73 | 49 | 111 | ##73; | I | 105 | 69 | 151 | ##105; | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | ##42; | * | 74 | 4A | 112 | ##74; | J | 106 | 6A | 152 | ##106; | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | ##43; | + | 75 | 4B | 113 | ##75; | K | 107 | 6B | 153 | ##107; | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | ##44; | , | 76 | 4C | 114 | ##76; | L | 108 | 6C | 154 | ##108; | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | ##45; | - | 77 | 4D | 115 | ##77; | M | 109 | 6D | 155 | ##109; | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | ##46; | . | 78 | 4E | 116 | ##78; | N | 110 | 6E | 156 | ##110; | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | ##47; | / | 79 | 4F | 117 | ##79; | O | 111 | 6F | 157 | ##111; | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | ##48; | 0 | 80 | 50 | 120 | ##80; | P | 112 | 70 | 160 | ##112; | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | ##49; | 1 | 81 | 51 | 121 | ##81; | Q | 113 | 71 | 161 | ##113; | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | ##50; | 2 | 82 | 52 | 122 | ##82; | R | 114 | 72 | 162 | ##114; | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | ##51; | 3 | 83 | 53 | 123 | ##83; | S | 115 | 73 | 163 | ##115; | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | ##52; | 4 | 84 | 54 | 124 | ##84; | T | 116 | 74 | 164 | ##116; | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | ##53; | 5 | 85 | 55 | 125 | ##85; | U | 117 | 75 | 165 | ##117; | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | ##54; | 6 | 86 | 56 | 126 | ##86; | V | 118 | 76 | 166 | ##118; | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | ##55; | 7 | 87 | 57 | 127 | ##87; | W | 119 | 77 | 167 | ##119; | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | ##56; | 8 | 88 | 58 | 130 | ##88; | X | 120 | 78 | 170 | ##120; | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | ##57; | 9 | 89 | 59 | 131 | ##89; | Y | 121 | 79 | 171 | ##121; | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | ##58; | : | 90 | 5A | 132 | ##90; | Z | 122 | 7A | 172 | ##122; | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ##59; | ; | 91 | 5B | 133 | ##91; | [| 123 | 7B | 173 | ##123; | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | ##60; | < | 92 | 5C | 134 | ##92; | \ | 124 | 7C | 174 | ##124; | |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | ##61; | = | 93 | 5D | 135 | ##93; |] | 125 | 7D | 175 | ##125; | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | ##62; | > | 94 | 5E | 136 | ##94; | ^ | 126 | 7E | 176 | ##126; | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ##63; | ? | 95 | 5F | 137 | ##95; | _ | 127 | 7F | 177 | ##127; | DEL |

Source: www.LookupTables.com

Funkce pro práci se stringy

- V různých knihovnách
- Include hlavičky - *string.h*, *stdlib.h*, *ctype.h* a další
- [Seznam C-String funkcí](#)

strlen

- Jedna z nejdůležitějších funkcí pro práci se stringy - udává počet znaků ve stringu (čte string než narazí na null char)

strcmp

- Porovnávání 2 stringů
- Nelze jen porovnat proměnné! Jsou to pointery!
- DEMO - *string_cmp_wrong.c*
- Musí se porovnat charakter po charakteru (to dělá strcmp)
- Výjimky pro compile-time známé stringy (optimalizace)

Skládání dat

- Nechceme pracovat jen s primitivními daty - přidání struktury
- struct type - kompozitní typ, skládá se z několika jednodušších struktur/primitivů

```
1 struct Person {
2     char *firstname;
3     char *surname;
4     char *city;
5     int year_born;
6 };
7
8 int main() {
9     // Deklarace a inicializace struktury
10    struct Person p = {
11        "Jan",
12        "Novak",
13        "Praha",
14        1995
15    };
16    printf("%s", p.firstname); // Jan
17    printf("%s", p.city); // Praha
18 }
```

Přístup k datům

- Přímý přístup (pomocí member access operátoru ".")
- Přístup přes pointer (pomocí operátoru "->")

```
1 struct Point3D {  
2     int x, y, z;  
3 };  
4  
5 int get_x_pt(struct Point3D * pt) {  
6     return pt->x;  
7 }  
8  
9 int get_x(struct Point3D pt) {  
10     return pt.x;  
11 }
```

Struct jako argument funkce

- Parametry se předávají jako **value**
- Při předání struktury přímo se předá její **kopie**
- Pro velké struktury je pass-by-value drahá operace - hodně kopírování
- Předávání pointerem - předá se jen ukazatel na to, kde v paměti struktura je
- Při předání pointerem můžeme modifikovat
- Můžeme opět aplikovat const modifier - nelze pak modifikovat struct members

```
1 #include <stdio.h>
2 struct Point {
3     int x, y;
4 };
5
6 void set_x_wrong(struct Point pt, int new_x) {
7     pt.x = new_x;
8 }
9
10 void set_x(struct Point *pt, int new_x) {
11     pt->x = new_x;
12 }
13
14 int main() {
15     struct Point pt = {5, 5};
16     set_x_wrong(pt, 40); // Nedojde ke zmene, pt ve funkci je kopie!
17     printf("%i", pt.x); // 5
18     set_x(&pt, 40); // Dojde ke zmene, predavame pointer na "pt"
19     printf("%i", pt.x); // 40
20     return 0;
21 }
```

Inicializační funkce

- Často se setkáme s použitím inicializačních funkcí
- Pomáhají s parametrizací - díky hlavičce funkce lépe vidíme co hodnota znamená

```
1 struct Person {
2     char *firstname;
3     char *surname;
4     char *city;
5     int year_born;
6 };
7
8 void Person_init(
9     struct Person* const obj,
10    const char* const firstname,
11    const char* const surname,
12    const char* const city,
13    const int year_born) {
14    obj->firstname = firstname;
15    obj->surname = surname;
16    obj->city = city;
17    obj->year_born = year_born;
18 }
```

NULL

- Obsažen v standardní knihovně, hodnota `((void*)0)`
- Neukazuje na žádné místo v paměti!
- Používá se jako "speciální hodnota" nebo inicializační hodnota
- Garantovaná nerovnost s jakýmkoliv pointerem na proměnnou (nebo funkci)
- Dereference NULL pointeru způsobí chybu programu!!!
- Velmi častá (a kritická) chyba v C programech

```
1 #include <stdlib.h>
2 int main() {
3     int *ptr = NULL; // Deklarovano, zatím nemáme vyhrazenou pamet
4     int x = 5;
5     ptr = &x; // Teprve nyní pointer ukazuje na l-value
6 }
```


Ukázka - linked list

```
1 #include <stdio.h>
2 struct IntLinkedList {
3     int head;
4     struct IntLinkedList *tail; // Rekurzivni struktura
5 };
6
7 struct IntLinkedList cons(int head, struct IntLinkedList *tail) {
8     struct IntLinkedList ll = {head, tail};
9     return ll; // Pridavame novy head za tail
10 }
11
12 int first(struct IntLinkedList *list) { return list->head; }
13
14 struct IntLinkedList* rest(struct IntLinkedList *list) { return list->tail; }
15
16 int sum (struct IntLinkedList *list) {
17     if (list == NULL) return 0;
18     return first(list) + sum(rest(list));
19 }
20
21 int main() {
22     struct IntLinkedList ll1 = cons(10, NULL),
23         ll2 = cons(20, &ll1),
24         ll3 = cons(30, &ll2);
25     int result = sum(&ll3); // 60
26     return 0;
27 }
```

Stále není ideální - musíme explicitně říkat v jakém scope žijí prvky listu tím, že je tam deklarujeme - nelze psát `cons(30, cons(20, cons(10, NULL)))`

Budeme potřebovat dynamickou alokaci paměti - později

l-value scope

```
1 #include <stdio.h>
2 struct Point {
3     int x, y;
4 };
5
6 struct Point * new(int x, int y) {
7     struct Point p = {x, y};
8     return &p;
9 }
10
11 struct Point add(struct Point *a, struct Point *b) {
12     struct Point * pt_addr = new(a->x + b->x, a->y + b->y);
13     return *pt_addr;
14 }
15
16 int main() {
17     struct Point pt = add(new(1,1), new(2,2));
18     printf("%i", pt.x);
19     return 0;
20 }
```

```
<source>: In function 'new':
<source>:8:12: warning: function returns address of local variable [-Wreturn-local-addr]
      8 |     return &p;
        |           ^~
ASM generation compiler returned: 0
<source>: In function 'new':
<source>:8:12: warning: function returns address of local variable [-Wreturn-local-addr]
      8 |     return &p;
        |           ^~
Execution build compiler returned: 0
Program returned: 139
```

Union typy

- struct slouží jako product type (stejně jako v BSL/ISL), kombinuje více hodnot do jednoho typu
- V některých případech potřebujeme sum type - předat jednu hodnotu nabývající jednoho z více typů (v ISL např. Maybe Number - Number nebo #f)
- union typy - deklarují všechny možnosti, realizuje se pouze jedna

```
1 #include <stdio.h>
2 union IntOrChar {
3     int integer;
4     char character;
5 };
6
7 int main() {
8     union IntOrChar u;
9     u.integer = 32;
10    u.character = 'A';
11    printf("%i\n", u.integer); // 65 - corrupted
12    printf("%c\n", u.character); // A
13 }
```

Union typy

- Union typy tedy obsahují pouze jednu z možností (sum type)
- Jak poznat kterou? Obalení do struct s tagem udávající možnost
- Použití tzv. enum typu pro označení možnosti (tag)

Enum typy

- Výčtové typy
- Překládají se na integer
- Slouží ke zlepšení korektnosti za compile-time

```
1 enum WeekDay
2 {
3     Mon,
4     Tue,
5     Wed,
6     Thu,
7     Fri,
8     Sat,
9     Sun
10 };
```

Enum typy

- Lze explicitně specifikovat jaký integer bude přiřazen

```
1 enum PermissionLevel
2 {
3     User = 1,
4     Moderator = 10,
5     Admin = 50,
6     Owner = 100
7 };
```

DEMO: tagged_union.c

TBD