

# Standardní knihovny C. Rekurze.

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 07

**B0B36PRP – Procedurální programování**

## Přehled témat

- Část 1 – Standardní knihovny, čtení/zápis ze/do souboru  
Standardní knihovny  
Práce se soubory  
Zpracování chyb
- Část 2 – Rekurze  
Faktoriál  
Obrácený výpis  
Hanojské věže  
Rekurze  
Fibonacciho posloupnost
- Část 3 – Zadání 7. domácího úkolu (HW7)

*S. G. Kochan: kapitola 16, Appendix B*

## Část I

### Část 1 – Standardní knihovny, čtení/zápis ze souboru

## Standardní knihovny

- Jazyk C sám osobě neobsahuje prostředky pro vstup/výstup dat, složitější matematické operace ani:
  - práci z textovými řetězci;
  - správu paměti pro dynamické přidělování;
  - vyhodnocení běhových chyb (run-time errors).
- Tyto a další funkce jsou obsaženy ve standardních knihovnách, které jsou součástí překladače jazyka C.
  - **Knihovny** – přeložený kód se připojuje k programu, např `libc.so`.
  - **Hlavičkové soubory** – obsahují prototypy funkcí, definice typů, makra a konstanty a vkládají se do zdrojových souborů příkazem preprocesoru `#include <jmeno_knihovny.h>`.

*Viz např. `ltd a.out`.*

*Např. `#include <stdio.h>`*

## Standardní knihovny

- `stdio.h` – Vstup a výstup (formátovaný i neformátovaný)
- `stdlib.h` – Matematické funkce, alokace paměti, převod řetězců na čísla, řazení (`qsort`), vyhledávání (`bsearch`), generování náhodných čísel (`rand`)
- `limits.h` – Rozsahy číselných typů
- `math.h` – Matematické funkce
- `errno.h` – Definice chybových hodnot
- `assert.h` – Zpracování běhových chyb
- `cctype.h` – Klasifikace znaků (`char`)
- `string.h` – Řetězce, blokové přenosy dat v paměti (`memcpy`)
- `locale.h` – Internacionalizace
- `time.h` – Datum a čas

## Základní práce se soubory – otevření souboru

- Knihovna `stdio.h`.
- Přístup k souboru je prostřednictvím ukazatele `FILE*`.
- Otevření souboru `FILE *fopen(char *filename, char *mode);`.
- Práce s textovými a binárními (*modifikátor "b"*) soubory.
- Soubory jsou čteny/zapisovány sekvenčně.
  - Se soubory se pracuje jako s proudem dat — postupné načítání/zápis.
    - Aktuální „pozici“ v souboru si můžeme představit jako kurzor.
    - Při otevření souboru se kurzor nastavuje na začátek souboru.
- Režim práce se souborem je dán hodnotou proměnné `mode`
  - "r" – režim čtení; *"r" – čtení textového souboru, "rb" – čtení binárního souboru*
  - "w" – režim zápisu; *Vytvoří soubor, pokud neexistuje, jinak smaže obsah souboru*
  - "a" – režim přidávání do souboru. *Kurzor je nastaven na konec souboru.*
- Můžeme kombinovat s dalšími režimy otevření souboru, např. "r+" pro otevření souboru pro čtení i zápis. *viz man fopen*

## Standardní knihovny (POSIX)

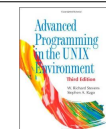
Komunikace s operačním systémem (OS).

*POSIX – Portable Operating System Interface*

- `stdlib.h` – Funkce využívají prostředků OS
- `signal.h` – Asynchronní události, vlákna
- `unistd.h` – Procesy, čtení/zápis souborů, ...
- `pthread.h` – Vlákna (POSIX Threads)
- `threads.h` – Standardní knihovna pro práci s vlákny (C11)



[Advanced Programming in the UNIX Environment, 3rd edition, W. Richard Stevens, Stephen A. Rago Addison-Wesley, 2013, ISBN 978-0-321-63773-4](#)



## Testování – otevření/zavření souboru

- Testování otevření souboru.

```

1 char *fname = "file.txt";

3 if ((f = fopen(fname, "r")) == NULL) {
4     fprintf(stderr, "Error: open file '%s'\n", fname);
5 }

■ Zavření souboru – int fclose(FILE *file);

1 if (fclose(f) == EOF) {
2     fprintf(stderr, "Error: close file '%s'\n", fname);
3 }

■ Dosažení konce souboru – int feof(FILE *file);
```

## Příklad – čtení souboru znak po znaku

- Čtení znaku: `int getc(FILE *file);`
- Hodnota znaku (`unsigned char`) je vrácena jako `int`.

```
1 int count = 0;
2 while ((c = getc(f)) != EOF) {
3     printf("Read character %d is '%c'\n", count, c);
4     count++;
5 }
```

lec07/read\_file.c

- Pokud nastane chyba nebo konec souboru vrátí funkce `getc` hodnotu `EOF`.
- Pro rozlišení chyby a konce souboru lze využít funkce `feof()` a `ferror()`.

## Zápis do textového souboru

- Zápis po znaku – `int putc(int c, FILE *file);`
- Formátovaný výstup `int fprintf(FILE *file, const *format, ...);`

```
1 int main(int argc, char *argv[])
2 {
3     char *fname = argc > 1 ? argv[1] : "out.txt";
4     FILE *f;
5     if ((f = fopen(fname, "w")) == NULL) {
6         fprintf(stderr, "Error: Open file '%s'\n", fname);
7         return -1;
8     }
9     fprintf(f, "Program arguments argc: %d\n", argc);
10    for (int i = 0; i < argc; ++i) {
11        fprintf(f, "argv[%d]='%s'\n", i, argv[i]);
12    }
13    if (fclose(f) == EOF) {
14        fprintf(stderr, "Error: Close file '%s'\n", fname);
15        return -1;
16    }
17    return 0;
18 }
```

lec07/file\_printf.c

- Identicky k `stderr` lze použít `stdout` a `stdin` pro čtení.

## Formátované čtení z textového souboru

- `int fscanf(FILE *file, const char *format, ...);`
- Analogie formátovanému vstupu.

*Pro vyplnění hodnot proměnných předáváme ukazatel.*

- Vrací počet přečtených položek, například pro vstup  
record 1 13.4
- příkaz: `int r = fscanf(f, "%s %d %lf\n", str, &i, &d);`
- vrátí v případě úspěšného čtení hodnotu proměnné  
`r == 3.`

- Při čtení textového řetězce je nutné zajistit dostatečný paměťový prostor pro načítaný textový řetězec, např. omezením velikosti řetězce.

```
char str[10];
int r = fscanf(f, "%9s %d %lf\n", str, &i, &d);
```

lec07/file\_scanf.c

## Náhodný přístup k souborům – `fseek()`

- Nastavení pozice kurzoru v souboru relativně vůči `whence` v bajtech.
- `int fseek(FILE *stream, long offset, int whence);`,  
kde `whence`
  - `SEEK_SET` – nastavení pozice od začátku souboru;
  - `SEEK_CUR` – relativní hodnota vůči současné pozici v souboru;
  - `SEEK_END` – nastavení pozice od konce souboru.
- `fseek()` vrací 0 v případě úspěšného nastavení pozice.

- Nastavení pozice v souboru na začátek.  
`void rewind(FILE *stream);`

## Binární čtení/zápis z/do souboru

- Otevření souboru s příznakem "b".

*Vliv na řetězce, řídicí znaky např. "\0", \n nebo EOF či EOT – Ctrl+Z.*

- Pro čtení a zápis bloku dat můžeme využít funkce `fread()` a `fwrite()` z knihovny `stdio.h`.
- Načtení `nmemb` prvků, každý o velikosti `size` bajtů.  
`size_t fread(void* ptr, size_t size, size_t nmemb, FILE *stream);`
- Zápis `nmemb` prvků, každý o velikosti `size` bajtů.  
`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
- Funkce vrací počet přečtených/zapsaných prvků o velikosti `size`.
- Pokud došlo k chybě nebo detekci konce souboru, funkce vrací menší než očekávaný počet bajtů.

Příklad použití `errno` (chyba při otevření souboru)

```

1 #include <stdio.h>
2 #include <errno.h>
3 #include <string.h>
4
5 int main(int argc, char *argv[])
6 {
7     FILE *f = fopen("soubor.txt", "r");
8     if (f == NULL) {
9         int r = errno;
10        printf("Open file failed errno value %d\n", errno);
11        printf("String error '%s'\n", strerror(r));
12    }
13    return 0;
14 }
```

lec07/errno.c

- Výstup při neexistujícím souboru.  
Open file failed errno value 2  
String error 'No such file or directory'
- Výstup při pokusu otevřít soubor bez práv přístupu k souboru.  
Open file failed errno value 13  
String error 'Permission denied'

## Zpracování chyb

- Základní chybové kódy jsou definovány v `<errno.h>`.
- Tyto kódy jsou ve standardních C knihovnách používány jako příznaky nastavené v případě selhání volání funkce v globální proměnné `errno`.
- Například otevření souboru `fopen()` vrací hodnotu `NULL`, pokud se soubor nepodařilo otevřít.
- Z této hodnoty, ale nepoznáme proč volání selhalo.
- Pro funkce, které nastavují `errno`, můžeme podle hodnoty identifikovat důvod chyby.
- Textový popis číselných kódů pro standardní knihovnu C je definován v `<string.h>`.
- Řetězec můžeme získat voláním funkce

```
char* strerror(int errnum);
```

Testovací makro `assert()`

- Do kódu lze přidat podmínky na nutné hodnoty proměnných.
- Testovat můžeme makrem `assert(expr)` z knihovny `<assert.h>`.
- Pokud není `expr true`, program se ukončí a vypíše jméno zdrojového souboru a číslo řádku.
- Makro vloží příslušný kód do programu.  
*Relativně jednoduchý způsob indikace chyby, např. nevhodným argumentem funkce, posloupností volání, ale jako strukturální chyba programu, nikoliv hodnot definovaných za běhu.*
- Vložení makra lze zabránit kompilací s definováním makra `NDEBUG`. [man assert](#)
- Makro `assert` má význam zejména při ladění program.

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 int main(int argc, char *argv[])
5 {
6     assert(argc > 1);
7     printf("program argc: %d\n", argc);
8     return 0;
9 }
```

lec07/assert.c

- Uvedený příklad pouze demonstruje použití `assert()`.
- Není vhodné testovat proměnné definované za běhu programu.
- Test `assert()` nebude do programu vložen při kompilaci s `NDEBUG`.

## Příklad použití makra assert()

- Kompilace s makrem a spuštění programu bez/s argumentem.

```
$ clang assert.c -o assert
$ ./assert
Assertion failed: (argc > 1), function main, file assert.c, line 5.
zsh: abort      ./assert
```

```
$ ./assert 2
start argc: 2
```

- Kompilace bez makra a spuštění programu bez/s argumentem.

```
$ clang -DDEBUG assert.c -o assert
$ ./assert
program start argc: 1
$ ./assert 2
program start argc: 2
```

lec07/assert.c

## Test alokace paměti a předčasné ukončení programu

- Dynamické přidělení paměti (`malloc`) je vhodné vždy kontrolovat.
- Pragmaticky můžeme očekávat typický průběh program bezchybný, včetně dynamické alokace paměti.
- Pak může být vhodné přidělení paměti kontrolovat, ale předčasně program ukončit v případě chyby.
- Nicméně stále je vhodné dát uživateli možnost dozvědět se, proč se program předčasně ukončil.
- Můžeme si tak napsat vlastní funkci (makro) `my_assert`, které ovšem nelze vyřadit kompilací s `-NDEBUG`.

```
1 #ifndef __MY_ASSERT_H__
2 #define __MY_ASSERT_H__
3
4 #include <stdio.h> //because of fprintf()
5 #include <stdlib.h> //because of exit() and malloc

7 #define my_assert(x, line, file) \
8     if (!(x)) {\
9         fprintf(stderr, "my_assert fail, line: %d,\
10            file %s\n", line, file);\
11            exit(-1);\
12        }

1 #include "my_assert.h"
2 ...
3 int *a = malloc(SIZE * sizeof(*a));
4 my_assert(a, __LINE__, __FILE__);
5 ...

lec07/demo-my_assert.c
```

- Při chybě dokážeme ve zdrojovém souboru dohledat místo a důvod chyb.

lec07/my\_assert.h

## Příkazy dlouhého skoku

- Příkaz `goto` je možné použít pouze v rámci jedné funkce.
- Knihovna `<setjmp.h>` definuje funkce `setjmp()` a `longjmp()` pro skoky mezi funkcemi.
- `setjmp()` uloží aktuální stav registrů procesoru a pokud funkce vrátí hodnotu různou od 0, došlo k volání `longjmp()`.
- Při volání `longjmp()` jsou hodnoty registrů procesoru obnoveny a program pokračuje od místa volání `setjmp()`.

*Kombinaci `setjmp()` a `longjmp()` lze využít pro implementaci ošetření výjimečných stavů podobně jako `try-catch` v jiných programovacích jazycích.*

```
1 #include <setjmp.h>
2 jmp_buf jb;
3 int compute(int x, int y);
4 void error_handler(void);
5 if (setjmp(jb) == 0) {
6     r = compute(x, y);
7     return 0;
8 } else {
9     error_handler();
10    return -1;
11 }

13 int compute(int x, int y) {
14     if (y == 0) {
15         longjmp(jb, 1);
16     } else {
17         x = (x + y * 2);
18         return (x / y);
19     }
20 }
21 void error_handler(void) {
22     printf("Error\n");
23 }
```

## Část II

### Část 2 – Rekurse

## Výpočet faktoriálu

```

■ Iterace
    n! = n(n-1)(n-2)...2·1

1 int factorialI(int n)
2 {
3     int f = 1;
4     for (; n > 1; --n) {
5         f *= n;
6     }
7     return f;
8 }

■ Rekurze
    n! = 1 pro n ≤ 1
    n! = n(n-1)! pro n > 1

1 int factorialR(int n)
2 {
3     int f = 1;
4     if (n > 1) {
5         f = n * factorialR(n-1);
6     }
7     return f;
8 }

    lec07/demo-factorial.c
    
```

## Příklad výpis posloupnosti 2/3

```

1 void reverse(void);

3 int main(void)
4 {
5     fprintf(stderr, "Enter a sequence of numbers (use Ctrl+D for the end of
6     the sequence)\n");
7     reverse();
8     printf("\n");
9     return 0;
10 }
11 void reverse(void)
12 {
13     int v;
14     if (scanf("%i", &v) == 1) {
15         reverse();
16         printf("%3d ", v);
17     }
18 }

    Ctrl+D ~ EOT – End-Of-Transmission (konec přenosu)

    lec07/demo-revert_sequence.c
    
```

## Příklad výpis posloupnosti 1/3

- Vytvořte program, který přečte posloupnost čísel a vypíše ji v opačném pořadí.
- Rozklad problému:
  - Zavedeme abstraktní příkaz: „*obrat posloupnost*“.
  - Příkaz rozložíme do tří kroků:
    1. Přečti číslo; **Číslo uložíme pro pozdější „obracený“ výpis.**
    2. Pokud není detekován konec posloupnosti „*obrat posloupnost*“; **Pokračujeme ve čtení čísel.**
    3. Vypiš číslo. **Vypíšeme uložené číslo.**

## Příklad výpis posloupnosti 3/3

```

■ lec07/demo-revert_sequence.c
■ Vytvoření posloupnosti.

./generate_numbers.sh | tr '\n' ' ' | cat > numbers.txt
clang demo-revert_sequence.c
./a.out < numbers.txt 2>/dev/null > numbers-r.txt
./a.out < numbers-r.txt 2>/dev/null > numbers-rr.txt

■ Příkaz pro výpis obsahu souborů.

for i in numbers.txt numbers-r.txt numbers-rr.txt; do echo "$i"; cat $i;
echo ""; done

■ Výpis obsahu souborů.

numbers.txt
10 4 20 8 8 5 18 6 7 7
numbers-r.txt
7 7 6 18 5 8 8 20 4 10

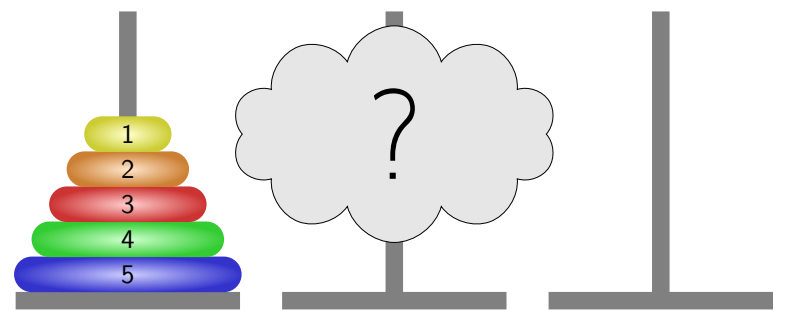
numbers-rr.txt
10 4 20 8 8 5 18 6 7 7
    
```

### Příklad Hanojské věže

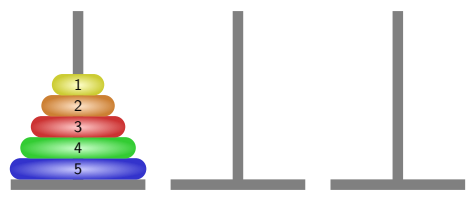


- Přemístit disky na druhou jehlu s použitím třetí (pomocné) jehly za dodržení následujících pravidel.
  1. V každém kroku můžeme přemístit pouze jeden disk a to vždy z jehly na jehlu.  
*Disky nelze odkládat mimo jehly.*
  2. Položit větší disk na menší není dovoleno.

### Hanojské věže – 5 disků



### Návrh řešení



- Zavedeme abstraktní příkaz `moveTower(n, 1, 2, 3)` realizující přesun  $n$  disků z jehly 1 na jehlu 2 s použitím jehly 3.
- Pro  $n > 0$  můžeme příkaz rozložit na tři jednodušší příkazy:
  1. `moveTower(n-1, 1, 3, 2);` Přesun  $n - 1$  disků z jehly 1 na jehlu 3.
  2. „přenes disk z jehly na jehlu 2“; *Přesun největšího disku na cílovou pozici.*  
*abstraktní příkaz*
  3. `moveTower(n-1, 3, 2, 1);` Přesun  $n - 1$  disků na cílovou pozici.

### Příklad řešení

```

1 void moveTower(int n, int from, int to, int tmp)
2 {
3     if (n > 0) {
4         moveTower(n-1, from, tmp, to); //move to tmp
5         printf("Move disc from %i to %i\n", from, to);
6         moveTower(n-1, tmp, to, from); //move from tmp
7     }
8 }
9
10 int main(int argc, char *argv[])
11 {
12     int numberOfDiscs = argc > 1 ? atoi(argv[1]) : 5;
13     moveTower(numberOfDiscs, 1, 2, 3);
14     return 0;
15 }

```

## Příklad výpisu

### ■ `lec07/demo-towers_of_hanoi.c`

```
clang demo-towers_of_hanoi.c      clang demo-towers_of_hanoi.c
./a.out 3                          ./a.out 4
Move disc from 1 to 2              Move disc from 1 to 3
Move disc from 1 to 3              Move disc from 1 to 2
Move disc from 2 to 3              Move disc from 3 to 2
Move disc from 1 to 2              Move disc from 1 to 3
Move disc from 3 to 1              Move disc from 2 to 1
Move disc from 3 to 2              Move disc from 2 to 3
Move disc from 1 to 2              Move disc from 1 to 3
Move disc from 1 to 2              Move disc from 1 to 2
Move disc from 3 to 2              Move disc from 3 to 2
Move disc from 3 to 1              Move disc from 3 to 1
Move disc from 2 to 1              Move disc from 2 to 1
Move disc from 3 to 2              Move disc from 3 to 2
Move disc from 1 to 3              Move disc from 1 to 3
Move disc from 1 to 2              Move disc from 1 to 2
Move disc from 3 to 2              Move disc from 3 to 2
```

## Rekurzivní vs iteračními algoritmy

- Nevýhodou rekurzivních algoritmů může být časová náročnost způsobená např. zbytečným opakováním výpočtu.
- Řadu rekurzivních algoritmů lze nahradit iteračními, které počítají výsledek „**zdola nahoru**“, tj. od menších (jednodušších) vstupních dat k větším (složitějším).
- Pokud algoritmus výpočtu „**zdola nahoru**“ nenajdeme, např. při řešení problému Hanojských věží, lze rekurzi odstranit využitím zásobníku.

*Např. zásobník využijeme pro uložení stavu řešení problému.*

## Rekurzivní algoritmy

- Rekurzivní funkce jsou přímou realizací rekurzivních algoritmů.
- Rekurzivní algoritmus předepisuje výpočet „**shora dolů**“.
- V závislosti na velikosti vstupních dat je výpočet předepsán:
  - Pro nejmenší (nejjednodušší) vstup je výpočet předepsán přímo;
  - Pro obecný vstup je výpočet předepsán s využitím téhož algoritmu pro menší vstup.
- Výhodou rekurzivních funkcí je jednoduchost a přehlednost.

## Rekurze

*“To iterate is human, to recurse divine.”*

L. Peter Deutsch

<http://www.devtopics.com/101-great-computer-programming-quotes>



## Elegance vs obtížnost rekurze

*I've often heard people describe understanding recursion as one of those "got it" moments, when the universe opened its secret stores of knowledge and gifted the mind of a burgeoning developer with a very powerful tool. For me, recursion has always been hard. Each time I'm able to peer more into its murky depths, I am humbled to see how little I feel like I really appreciate and understand its power and elegance.*

Rick Winfrey, 2012

<http://selfless-singleton.rickwinfrey.com/2012/11/27/to-iterate-is-human-to-recurse-divine>

## Fibonacciho posloupnost – historie

- Indičtí matematici (450 nebo 200 BC).
- Leonardo Pisano (1175–1250) popis růstu populace králíků.
  - Italský matematik známý také jako Fibonacci.*
  - $F_n$  – velikost populace po  $n$  měsících za následujících předpokladů.
    - První měsíc se narodí jediný pár.
    - Narozené páry jsou produktivní od 2. měsíce svého života.
    - Každý měsíc zplodí každý produktivní pár jeden další pár.
    - Králíci nikdy neumírají, nejsou nemocní atd.
- Henry E. Dudeney (1857–1930) – popis populace krav.
  - „Jestliže každá kráva vyprodukuje své první tele (jalovici) za rok a poté každý rok jednu další jalovici, kolik budete mít krav za 12 let, jestliže žádná nezemře a na počátku budete mít jednu krávu?“
  - Po 12 let je k dispozici jeden či více býků.*

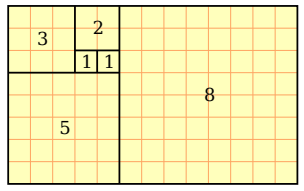
## Fibonacciho posloupnost

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Nebo 0, 1, 1, 2, 3, 5, ...

- $F_n = F_{n-1} + F_{n-2}$
- pro  $F_1 = 1, F_2 = 1$

Nebo  $F_1 = 0, F_2 = 1$



- Nekonečná posloupnost přirozených čísel, kde každé číslo je součtem dvou předchozích.
- Limita poměru dvou následujících čísel Fibonacciho posloupnosti je rovna **zlatému řezu**.
  - Sectio aurea – ideální poměr mezi různými délkami.
  - Rozdělení úsečky na dvě části tak, že poměr větší části ku menší je stejný jako poměr celé úsečky k větší části  $\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618\ 033\ 988\ 749\ 894\ 848\ \dots$

## Fibonacciho posloupnost – rekurzivně

- Platí:
  - $f_0 = 1$
  - $f_1 = 1$
  - $f_n = f_{n-1} + f_{n-2}$ , pro  $n > 1$

```

1 int fibonacci(int n) {
2     return n < 2
3         ? 1
4         : fibonacci(n - 1) + fibonacci(n - 2);
5 }

```

Zápis je elegantní, jak je však takový výpočet efektivní?

## Fibonacciho posloupnost – příklad 1/2

```

■ Počet operací při výpočtu Fibonacciho čísla n.
1 long counter; // store number of individual operations

3 long fibonnaciRecursive(int n) {
4     counter += 1; // jedno porovnání
5     return n < 2 ? 1 : fibonnaciRecursive(n - 1) + fibonnaciRecursive(n - 2);
6 }

8 long fibonnaciIterative(int n) {
9     long fibM2 = 1;
10    long fibM1 = 1;
11    long fib = 1;
12    for (int i = 2; i <= n; ++i) {
13        fibM2 = fibM1;
14        fibM1 = fib;
15        fib = fibM1 + fibM2;
16        counter += 3; // dvě přiřazení, jeden součet
17    }
18    return fib;
19 }
    
```

lec07/demo-fibonacci.c

## Fibonacciho posloupnost – rekurzivně vs iteračně

- Rekurzivní výpočet:
  - Složitost roste exponenciálně s  $n \sim 2^n$ .
- Iterační algoritmus:
  - Počet operací je proporcionální  $n \sim 3n$ .

lec07/demo-fibonacci-stats.c, lec07/fibonacci.sh
- Skutečný počet operací závisí na konkrétní implementaci, programovacím jazyku, překladači a hardware.
- Složitost algoritmů proto vyjadřujeme asymptoticky jako funkci velikosti vstupu.
  - Například v tzv. „Big O“ notaci:
    - rekurzivní algoritmus výpočtu má složitost  $O(2^n)$ ;
    - iterační algoritmus výpočtu má složitost  $O(n)$ .

*Efektivní algoritmy mají polynomiální složitost.*

## Fibonacciho posloupnost – rekurzivně 2/2

```

1 int main(int argc, char *argv[])
2 {
3     int n = argc > 1 ? atoi(argv[1]) : 25;
4     counter = 0; // reset counter
5     long fibR = fibonnaciRecursive(n);
6     long counterR = counter;

8     counter = 0; // reset counter
9     long fibI = fibonnaciIterative(n);
10    long counterI = counter;

12    printf("Fibonacci number recursive: %li\n", fibR);
13    printf("Fibonacci number iteration: %li\n", fibI);
14    printf("Counter recursive: %li\n", counterR);
15    printf("Counter iteration: %li\n", counterI);

17    return 0;
18 }
    
```

lec07/demo-fibonacci.c

```

$ clang demo-fibonacci.c && ./a.out 30
Fibonacci number recursive: 1346269
Fibonacci number iteration: 1346269
Counter recursive: 2692537
Counter iteration: 87
    
```

## Vsuvka – Vykreslení grafu

- Jednoduchou úpravou vypíšeme počty operací na řádek.
 

```
printf("%u\t%6.3e\t%6.3e\n", n, (double)counterR, (double)counterI);
```

lec07/demo-fibonacci-stats.c
  - Program zkompilujeme a spustíme.
 

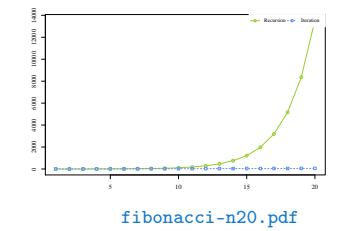
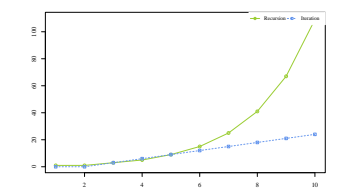
```
clang demo-fibonacci-stats.c
```

```
$. /a.out 10 > fibonacci.dat &&
$. /fibonacci.sh fibonacci.dat fibonacci-n10.pdf
```

fibonacci-n10.pdf

```
$. /a.out 20 > fibonacci.dat &&
$. /fibonacci.sh fibonacci.dat fibonacci-n20.pdf
```

fibonacci-n20.pdf
- Přesměrovaný standardní výstup do souboru fibonacci.dat představuje tabulku hodnot, která je vykreslena do grafu skriptem fibonacci.sh s využitím nástroje R – <https://www.r-project.org/>.



## Vsuvka – Kompilace s optimalizací

- Můžeme rekurzivní výpočet urychlit kompilací s optimalizacemi kódu?

```
$ clang demo-fibonacci.c && time ./a.out 50
Fibonacci number recursive: 20365011074
Fibonacci number iteration: 20365011074
Counter recursive: 40730022147
Counter iteration: 147

real    1m35.912s
user    1m35.824s
sys     0m0.000s

$ clang -O2 -march=native demo-fibonacci.c && time ./a.out 50
Fibonacci number recursive: 20365011074
Fibonacci number iteration: 20365011074
Counter recursive: 40730022147
Counter iteration: 147

real    1m16.042s
user    1m15.968s
sys     0m0.008s
```

- Ano, můžeme. V tomto případě je však zrychlení zanedbatelné.
- V tomto případě je rozhodující asymptotická složitost  $O(2^n)$  vs  $O(n)$ .

*Obecně se pro odladěné a výpočetně náročné programy vyplatí kompilovat s optimalizací. Nárůst výkonů může být i několikanásobný.*

# Část III

## Část 2 – Zadání 7. domácího úkolu (HW7)

Jan Faigl, 2024      BOB36PRP – Přednáška 07: Standardní knihovny C. Rekurze.      50 / 53

## Zadání 7. domácího úkolu HW7

### Téma: Kruhová fronta v poli

Povinné zadání: **3b**; Volitelné zadání: **4b**; Bonusové zadání: *není*

- **Motivace:** Práce s pamětí a datovými strukturami.
- **Cíl:** Prohloubit si znalost paměťové reprezentace a dynamické alokace paměti s uvolňováním.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw7>
  - Implementace kruhové fronty s využitím předalokovaného pole pro vkládané prvky.
  - **Volitelné zadání** rozšiřuje úlohu o dynamické zvětšování a zmenšování kapacity fronty podle aktuálních požadavků na počet vkládaných/odebíraných prvků.
- **Termín odevzdání:** 27.04.2024, 23:59:59 PDT.

Diskutovaná témata

# Shrnutí přednášky

Jan Faigl, 2024      BOB36PRP – Přednáška 07: Standardní knihovny C. Rekurze.      52 / 53

## Diskutovaná témata

- Standardní knihovny C
- Čtení a ukládání z/do souboru
- Ošetření chybových stavů - `assert()`, `errno`, `setjmp()`, `longjmp()`
- Rekurse a rekurzivní řešení problémů
- Kompilace programu s optimalizacemi
- **Příště: Spojové struktury.**

## Kódovací příklad – Ne úplně čitelné použití goto

- Použití `goto` souvisí zejména s čitelností kódu, může jej využít reakci na návratové hodnoty.

```

1 #include <stdio.h>

3 int main(void)
4 {
5     int c = getchar();
6     if (c >= '0' && c <= '9') {
7         goto print_digit;
8     }
9     goto print_error;
10 print_digit:
11     printf("User input '%c' that has ASCII code value %d\n", c, c);
12     goto leave;
13 print_error:
14     fprintf(stderr, "ERROR: Expected input value is '0'--'9'\n");
15 leave:
16     return 0;
17 }

```

[Low Level Learning: why is it illegal to use "goto"?](#)

- Uvedený příklad je funkční, ale načítelnosti úplně nepřidává.

V našem případě se plně obejdeme bez `goto`, obecně ale může být jeho použití užitečné, např. pokud testujeme postupně volání funkcí.

## Část V Appendix

## Kódovací příklad – struct 1/3

- Implementujeme složený typ s dvěma položkami typu pole znaků `username` a `number`, kde první položku chceme interpretovat jako textový řetězec (*null terminated*), ale ve druhém případě pouze jako pole znaků.

*Ukázkový příklad, jehož hlavní motivace je uložení paměti do souboru a náhled na obsah souboru.*

```

1 #define USERNAME_SIZE 8
2 #define NUMBER_DIGITS 4

3 struct record {
4     char username[USERNAME_SIZE + 1];
5     char number[NUMBER_DIGITS];
6 };

7

9 int main(void) {
10     struct record records[] = {
11         { .username = "user01" },
12         { .username = "user02" },
13         { .username = "admin" },
14         { .username = "root" },
15         { .username = '\0' } // null terminating array
16     };

18     fprintf(stderr, "DEBUG: size of struct %lu\n",
19             sizeof(struct record));
20     fprintf(stderr, "DEBUG: size of records %lu\n",
21             sizeof(records));

```

- Položka `username` je o jeden znak delší, uložení `'\0'`.
  - Položka `number` je zápis čísla o maximální hodnotě 9999 (počet řádů 4) v textové podobě (0000–9999).
  - Velikost složeného typu je dána velikostí jednotlivých položek.
  - Pole záznamů inicializujeme se zarážkou (poslední prvek obsahuje prázdný řetězec v položce `username`).
- ```

$ clang struct.c && ./a.out
DEBUG: size of struct 13
DEBUG: size of records 6

```
- Na příkladu si ukážeme, jak převést celé číslo na textovou reprezentaci, k čemuž použijeme několik pomocných funkcí.
  - Implementujeme si funkce pro tisk záznamu a pole záznamů.
  - Položku `number` naplníme programově z celého čísla s kontrolou, zdali se číslo vejde do `NUMBER_DIGITS`.
  - Složený typ a implementaci funkcí realizujeme v samostatném modulu `record.h` a `record.c`.

## Kódovací příklad – struct 2/3

```

1 #ifndef __RECORD_H__
2 #define __RECORD_H__
3
4 #define USERNAME_SIZE 8
5 #define NUMBER_DIGITS 4
6
7 struct record {
8     char username[USERNAME_SIZE + 1];
9     char number[NUMBER_DIGITS];
10 };
11
12 void print_record(const struct record * record);
13 void print(const struct record * const records);
14
15 unsigned int fill_numbers(struct record * const records);
16
17 #endif
                                record.h

```

- V C nemůžeme přetěžovat jména funkcí, proto máme funkci `print_record()` a `print()`.
- Funkce `fill_numbers()` vyplní položky `numbers` v posloupnosti hodnot typu `struct record`.

```

1 #include <stdio.h>
2 #include <limits.h>
3 #include <assert.h> // strukturální testy
4
5 #include "record.h"                                record.c
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33 void print_record(const struct record * record)
34 {
35     if (record) {
36         printf("Record\n");
37         printf("|- username: \"%s\"\n", record->username);
38         printf("|- number: ");
39         print_chars(NUMBER_DIGITS, record->number);
40         printf("\n\n");
41         // printf("|- number: %s\n"); // Vyzkoušejte!
42     }
43 }
44
45 void print(const struct record * const records)
46 {
47     struct record const *cur = records;
48     while (cur && cur->username[0]) {
49         print_record(cur);
50         cur += 1; // pointer arithmetic
51     }
52 }
                                record.c

```

## Kódovací příklad – struct 3/3

```

7 static unsigned short get_max_number(unsigned int digits)
8 {
9     unsigned int number = 1;
10    assert(sizeof(short) < sizeof(int)); // 16 vs. 32?
11    for (unsigned int i = 0; i < digits; ++i) {
12        number *= 10;
13    }
14    assert(number <= USHRT_MAX); // short
15    return number - 1;
16 }
17
18 static void fill_record_number(unsigned short n, int
19 digits, char *number)
20 { //number needs to be at least digits large
21     for (int i = digits - 1; i >= 0; --i) {
22         number[i] = (n % 10) + '0';
23         n = n / 10;
24     }
25 }
                                record.c

```

- V programu máme snahu testovat velikost paměťové reprezentace.

```

26 static void print_chars(size_t digits, const char *number)
27 { // number je ukazatel na konstantní hodnotu
28     for (size_t i = 0; i < digits; ++i, ++number) {
29         putchar(*number);
30     }
31 }
                                record.c
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54 unsigned int fill_numbers(struct record * const records)
55 {
56     struct record *cur = records;
57     unsigned short n = 0;
58     const short max_number = get_max_number(NUMBER_DIGITS);
59
60     while (cur && cur->username[0]) {
61         assert(n <= max_number); // Vyplňujeme programově
62         fill_record_number(n, NUMBER_DIGITS, cur->number);
63         n += 1;
64         cur += 1;
65     }
66     return n;
67 }
                                record.c

```

- Lokální pomocné funkce jsou `static`.
- Hodnoty položky `number` vyplňujeme programově inkrementálně od hodnoty 0000.

## Kódovací příklad – Načítání a ukládání struct 1/4

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "record.h"
5
6 int main(void) {
7     int ret = EXIT_SUCCESS;
8
9     struct record records[] = {
10         { .username = "user01" },
11         { .username = "user02" },
12         { .username = "admin" },
13         { .username = "root" },
14         { .username = "jf" },
15         { .username = '\0' } // null terminating array
16     };
17
18     fprintf(stderr, "DEBUG: size of struct %lu\n",
19             sizeof(struct record));
20     fprintf(stderr, "DEBUG: size of records %lu\n",
21             sizeof(records));
22
23     unsigned int n = fill_numbers(records); // number!
24     print(records);

```

```

24     const char *fname = "records.dat";
25     FILE *fd = fopen("records.dat", "w"); // uložení records
26     if (fd) {
27         size_t saved = fwrite(records, sizeof(struct record), n, fd);
28         size_t size = n * sizeof(struct record);
29         printf("DEBUG: saved bytes %lu out of %lu\n", saved * sizeof(struct
30 record), size);
31     } else {
32         fprintf(stderr, "ERROR: Cannot open \"%s\" for writing\n", fname);
33         ret = EXIT_FAILURE;
34     }
35     return ret;
                                save_struct.c
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

$ clang -c record.c -o record.o
$ clang save_struct.c record.o -o save && ./save
DEBUG: size of struct 13
DEBUG: size of records 78
Record
|- username: "user01"
|- number: 0000
...
Record
|- username: "jf"
|- number: 0004
DEBUG: saved bytes 65 out of 65

```

## Kódovací příklad – Načítání a ukládání struct 2/4

- Obsah uloženého souboru můžeme zkusit přímo otevřít v textovém editoru nebo použijeme program `hexdump`.

```

$ clang -c record.c -o record.o
$ clang save_struct.c record.o -o save
$ ./save 1>/dev/null
DEBUG: size of struct 13
DEBUG: size of records 78
$ hexdump -C records.dat
00000000 75 73 65 72 30 31 00 00 00 30 30 30 30 75 73 65 |user01...0000use|
00000010 72 30 32 00 00 00 30 30 30 31 61 64 6d 69 6e 00 |r02...0001admin.|
00000020 00 00 00 30 30 30 32 72 6f 6f 74 00 00 00 00 00 |...0002root....|
00000030 30 30 30 33 6a 66 00 00 00 00 00 00 30 30 30 |0003jf.....000|
00000040 34
00000041 |4|

```

- V případě, že vytvořenému souboru `records.dat` odebereme práva zápisu, např. `chmod 0 records.dat`, program selže.

```

$ chmod 0 records.dat
$ ./save 1>/dev/null; echo $?
DEBUG: size of struct 13
DEBUG: size of records 78
ERROR: Cannot open file "records.dat" for writing
1

```

## Kódovací příklad – Načítání a ukládání struct 3/4

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "record.h"
5
6 #define NUM_RECORDS 2
7
8 int main(void) {
9     const char *fname = "records.dat";
10
11     FILE *fd = fopen(fname, "r");
12     if (!fd) {
13         perror("Error open file");
14         goto error;
15     }
16     struct record *records = malloc(NUM_RECORDS
17     * sizeof(struct record));
18     if (!records) {
19         perror("Error malloc");
20         fclose(fd); // Korektně soubor zavíráme.
21         goto error;
22     }
23     ssize_t loaded;

```

```

23     while ((loaded = fread(records, sizeof(struct record), NUM_RECORDS, fd)) {
24         fprintf(stderr, "DEBUG: loaded records %lu\n", loaded);
25         for (size_t i = 0; i < loaded; ++i) {
26             print_record(&records[i]);
27         }
28     }
29     free(records);
30     fclose(fd);
31     goto leave;
32 error:
33     fprintf(stderr, "ERROR: during reading from the file \"%s\"\n", fname);
34     return 1;
35 leave:
36     return 0;
37 }

```

```
$ clang load_struct.c record.o -o load && ./load
```

```

DEBUG: loaded records 2
...
Record
|- username: "root"
|- number: 0003

```

```

DEBUG: loaded records 1
Record
|- username: "jf"
|- number: 0004

```

save\_struct.c

- Načtení bloku dat funkcí `fread()`.

## Kódovací příklad – Načítání a ukládání struct 4/4 (lépe)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "record.h"
5
6 #define NUM_RECORDS 2
7
8 enum { ERROR_FILE = 101, ERROR_MEM = 102};
9
10 void print_error(int error);
11
12 int main(void) {
13     int ret = EXIT_SUCCESS;
14     const char *fname = "records.dat";
15
16     FILE *fd = fopen(fname, "r");
17     if (!fd && (ret = ERROR_FILE)) { // ret!
18         goto leave;
19     }
20     struct record *records = malloc(
21     NUM_RECORDS * sizeof(struct record));
22     if (!records && (ret = ERROR_MEM)) { // ret!
23         goto leave;
24     }
25     ssize_t loaded;

```

```

25     while ((loaded = fread(records, sizeof(struct record), NUM_RECORDS, fd)) {
26         fprintf(stderr, "DEBUG: loaded records %lu\n", loaded);
27         for (size_t i = 0; i < loaded; ++i) {
28             print_record(&records[i]);
29         }
30     }
31     free(records);
32
33 leave:
34     if (fd) {
35         fclose(fd);
36     }
37     print_error(ret);
38     return ret;
39 }
40
41 void print_error(int error)
42 {
43     switch (error) {
44     case ERROR_FILE:
45         fprintf(stderr, "ERROR: open file\n");
46         break;
47     case ERROR_MEM:
48         fprintf(stderr, "ERROR: mem allocation\n");
49         break;
50     }
51 }

```