

Standardní knihovny C. Rekurze.

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 08

B0B36PRP – Procedurální programování

Část I

Část 1 – Standardní knihovny, čtení/zápis ze souboru

Přehled témat

- Část 1 – Standardní knihovny, čtení/zápis ze/do souboru

Standardní knihovny

Práce se soubory

Zpracování chyb

S. G. Kochan: kapitola 16, Appendix B

P. Herout: kapitola 6

Popis knihovních funkcí je ve 2. díle

- Část 2 – Rekurze

Faktoriál

Obrácený výpis

Hanojské věže

Rekurze

Fibonacciho posloupnost

- Část 3 – Zadání 7. domácího úkolu (HW07)

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`
Viz např. `ldd a.out`
 - **Hlavičkové soubory** – obsahují prototypy funkcí, definici typů, makra a konstanty a vkládají se do zdrojových souborů příkazem preprocesoru `#include <jmeno_knihovny.h>`

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
- `ctype.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)

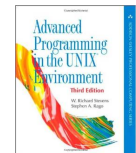
Komunikace s operačním systémem (OS)

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";
2
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,
4           c);
5     count++;
6 }

```

`lec08/read_file.c`

- Pokud nastane chyba nebo konec souboru vrací 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, ...);`

```

int main(int argc, char *argv[])
{
    char *fname = argc > 1 ? argv[1] : "out.txt";
    FILE *f;
    if ((f = fopen(fname, "w")) == NULL) {
        fprintf(stderr, "Error: Open file '%s'\n", fname);
        return -1;
    }
    fprintf(f, "Program arguments argc: %d\n", argc);
    for (int i = 0; i < argc; ++i) {
        fprintf(f, "argv[%d]='%s'\n", i, argv[i]);
    }
    if (fclose(f) == EOF) {
        fprintf(stderr, "Error: Close file '%s'\n", fname);
        return -1;
    }
    return 0;
}

```

`lec08/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);

```

`lec08/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 - Ctr+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 bajtů
- Pokud došlo k chybě nebo detekci konce souboru funkce vrací menší než očekávaný počet bajtů

## Příklad použití `errno`

- 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 }
```

lec08/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` logická 1 (`true`) program se ukončí a vypíše jméno zdrojového souboru a číslo řádku.
- Makro vloží příslušný kód do programu  
*Získáme tak relativně jednoduchý způsob indikace případné chyby, např. nevhodným argumentem funkce*
- Vložení makra lze zabránit kompilací s definováním makra `NDEBUG`  
[man assert](#)
- Příklad
 

```
#include <stdio.h>
#include <assert.h>

int main(int argc, char *argv[])
{
 assert(argc > 1);
 printf("program argc: %d\n", argc);
 return 0;
}
```

lec08/assert.c

## 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 -DNDEBUG assert.c -o assert
./assert
program start argc: 1
./assert 2
program start argc: 2
```

lec08/assert.c

## Část II

### Část 2 – Rekurse

## 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`*

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

## Výpočet faktoriálu

- Iterace

$$n! = n(n-1)(n-2)\dots 2 \cdot 1$$

```
int factorialI(int n)
{
 int f = 1;
 for (; n > 1; --n) {
 f *= n;
 }
 return f;
}
```

- Rekurse

$$n! = 1 \text{ pro } n \leq 1$$

$$n! = n(n-1)! \text{ pro } n > 1$$

```
int factorialR(int n)
{
 int f = 1;
 if (n > 1) {
 f = n * factorialR(n-1);
 }
 return f;
}
```

lec08/demo-factorial.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 posloupnost „*obrat posloupnost*“  
**pokračujeme ve čtení čísel**
    3. Vypiš číslo  
**vypíšeme uložené číslo**

## Příklad výpis posloupnosti 3/3

- `lec08/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 výpis posloupnosti 2/3

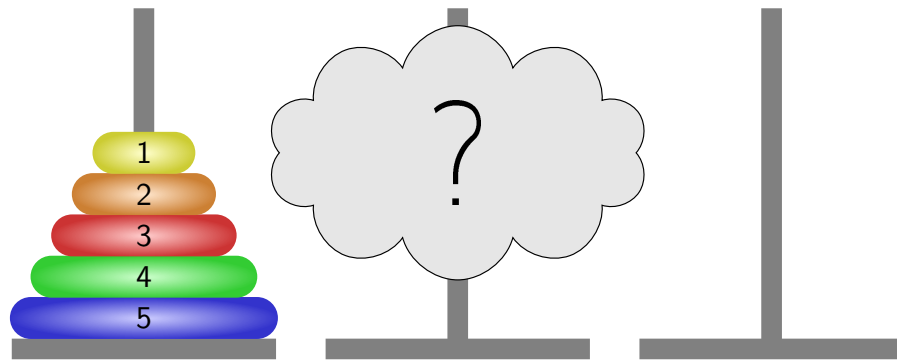
```
1 void reverse()
2 {
3 int v;
4 if (scanf("%i", &v) == 1) {
5 reverse();
6 printf("%3d ", v);
7 }
8 } Ctrl+D ~ EOT – End-Of-Transmission (konec přenosu)
9 int main(void)
10 {
11 fprintf(stderr, "Enter a sequence of numbers (use
12 Ctrl+D for the end of the sequence)\n");
13 reverse();
14 printf("\n");
15 return 0;
16 } lec08/demo-revert_sequence.c
```

## Příklad Hanojské věže



- Přemístit disky na druhou jehlu s použitím třetí (pomocné) jehly za dodržení 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



## Příklad řešení

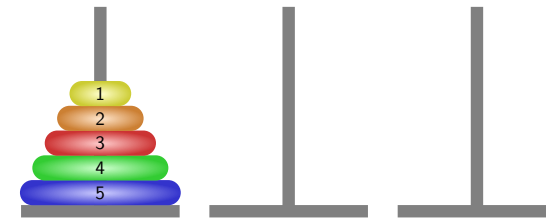
```

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 }

```

lec08/demo-towers\_of\_hanoi.c

## 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 výpisu

- `lec08/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 1
Move disc from 3 to 1 Move disc from 2 to 1
Move disc from 1 to 3 Move disc from 3 to 2
Move disc from 1 to 2 Move disc from 3 to 1
Move disc from 3 to 2 Move disc from 2 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 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>

## 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 rekurzivitou odstranit pomocí zásobníku.

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

## 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

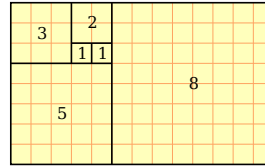
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

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:

$$f_0 = 1$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}, \text{ pro } n > 1$$

```

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 – 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 předpokladu
    - 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 – příklad 1/2

- Počet operací při výpočtu Fibonacciho čísla  $n$

```

1 long counter; // store number of individual operations
2
3 long fibonacciRecursive(int n) {
4 counter += 1; // jedno porovnání
5 return n < 2 ? 1 : fibonacciRecursive(n - 1) +
6 fibonacciRecursive(n - 2);
7 }
8 long fibonacciIterative(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 }
```

lec08/demo-fibonacci.c

## 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;
7
8 counter = 0; // reset counter
9 long fibI = fibonnaciIterative(n);
10 long counterI = counter;
11
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);
16
17 return 0;
18 }

```

lec08/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);
lec08/demo-fibonacci-stats.c

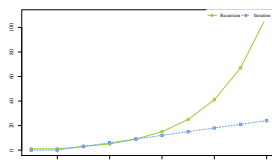
```

- Program zkompilujeme a spustíme

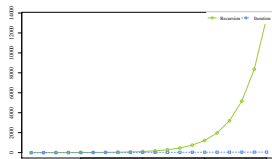
```
clang demo-fibonacci-stats.c
```

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

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



fibonacci-n10.pdf



fibonacci-n20.pdf

Přeměřovaný 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/>

```
lec08/demo-fibonacci-stats.c, lec08/fibonacci.sh
```

## Fibonacciho posloupnost – rekurzivně vs iteračně

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

```
lec08/demo-fibonacci-stats.c, lec08/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*

## Vsuvka – Kompilace s optimalizací

- Můžeme rekurzivní výpočet urychlit kompilací s optimalizacemi kódu?

|                                              |                                                                |
|----------------------------------------------|----------------------------------------------------------------|
| clang demo-fibonacci.c &&<br>time ./a.out 50 | clang -O2 -march=native demo-fibonacci.c<br>&& time ./a.out 50 |
| Fibonacci number recursive:<br>20365011074   | Fibonacci number recursive:<br>20365011074                     |
| Fibonacci number iteration:<br>20365011074   | Fibonacci number iteration:<br>20365011074                     |
| Counter recursive:<br>40730022147            | Counter recursive:<br>40730022147                              |
| Counter iteration: 147                       | Counter iteration: 147                                         |
| real 1m35.912s                               | real 1m16.042s                                                 |
| user 1m35.824s                               | user 1m15.968s                                                 |
| sys 0m0.000s                                 | sys 0m0.008s                                                   |

- Ano, můžeme, ale pouze zanedbatelně
- 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 (HW07)

## Shrnutí přednášky

## Zadání 7. domácího úkolu HW07

### Téma: Hledání textu v souborech

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

- **Motivace:** Dekomponovat výpočetní úlohu na dílčí výpočetní kroky
- **Cíl:** Osvojit si práci se soubory
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw07>
  - Zpracování vstupu po řádcích a detekce textového řetězce ve vstupním souboru
  - **Volitelné zadání** rozšiřuje úlohu o zpracování tří základních kvantifikátorů regulárních výrazů (pouze pro předcházející znak).
    - Znaky pro kvantifikátory: `?`, `*`, `+`
- **Termín odevzdání:** **01.12.2018, 23:59:59 PST**

## Diskutovaná témata

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