

# **PB071 – Principy nízkoúrovňového programování**

Rekurze, funkční ukazatele,  
knihovny, standardní knihovna

# Rekurzivně volané funkce

# Rekurzivní funkce - motivace

- Některé algoritmy  $M$  na vstupních datech  $X$  lze přirozeně vyjádřit jako sérii  $M$  nad menšími instancemi dat  $X$

- Faktoriál:  $N! == N * (N-1) * (N-2) * \dots * 2 * 1$

- imperativní řešení:











```
int fact(int N) {  
    int res = 1;  
    for (int i=N; i > 1; i--){  
        res *= i;  
    }  
    return res;  
}
```

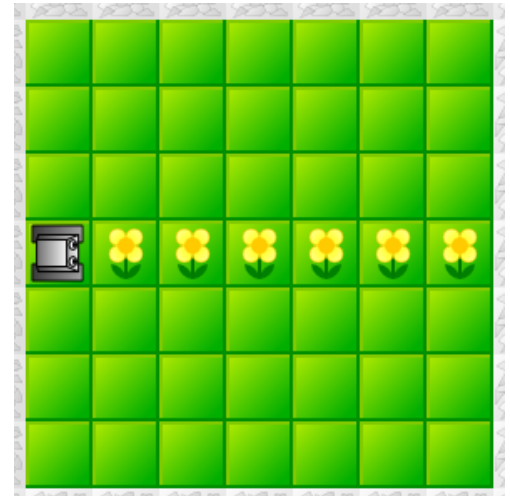
- Myšlenka řešení pomocí rekurze:

- $N! == N \times (N-1)!$  a  $0! == 1$

```
int fact(int N) {  
    if (N > 1) return N * fact(N-1);  
    else return 1;  
}
```

# Robotanik - <http://tutor.fi.muni.cz/>

- Nástroj pro cvičení funkcí a rekurze
- Robot s omezenou sadou instrukcí
  - Krok dopředu 
  - Otočení doleva  / doprava 
  - Přebarvení pole 
  - Zavolání některé z funkcí  
    - volání může být i rekurzivní
  - Podmínění instrukce barvou pole    
- Cíl je sebrat květiny na daný počet instrukcí
- Zaregistrujte se, budeme využívat na cvičení



# Robotanik - registrace

- Registrace může být anonymní
  - zaškrtněte ale skupinu **pb071\_jaro2016**
  - umožníte nám sledovat celkovou úroveň

**Pokud používáte Tutora na škole**

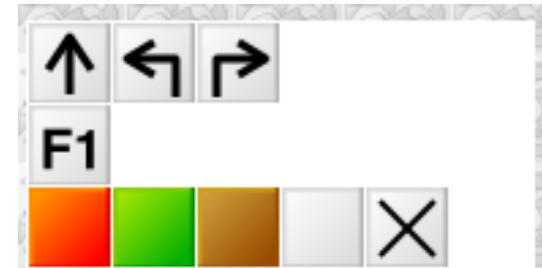
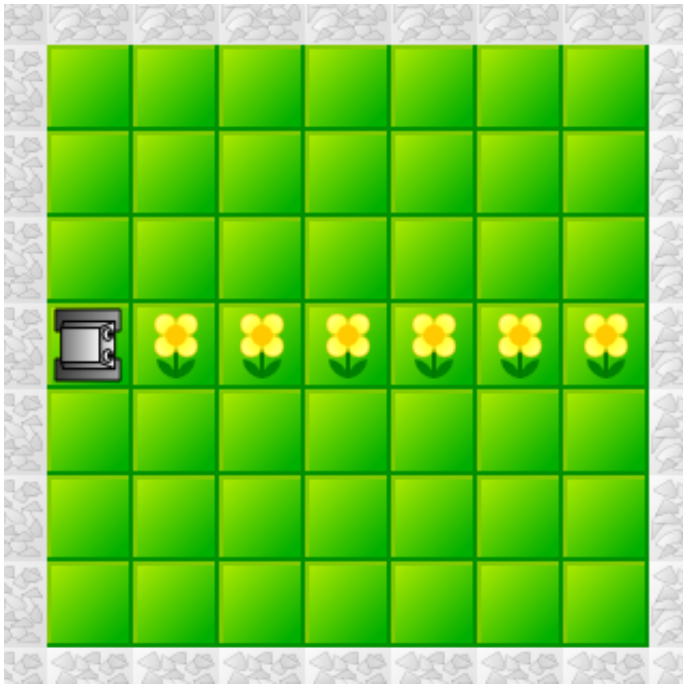
Přihlášení k výukovým skupinám	
Přidej skupinu	<input type="text" value="pb071"/> <input type="button" value="Dohledat"/>

**dodatečný  
identifikátor  
pb071\_jaro2016**

# Elementární rekurze

- Jak sebrat květinčky na dvě instrukce?

F1 

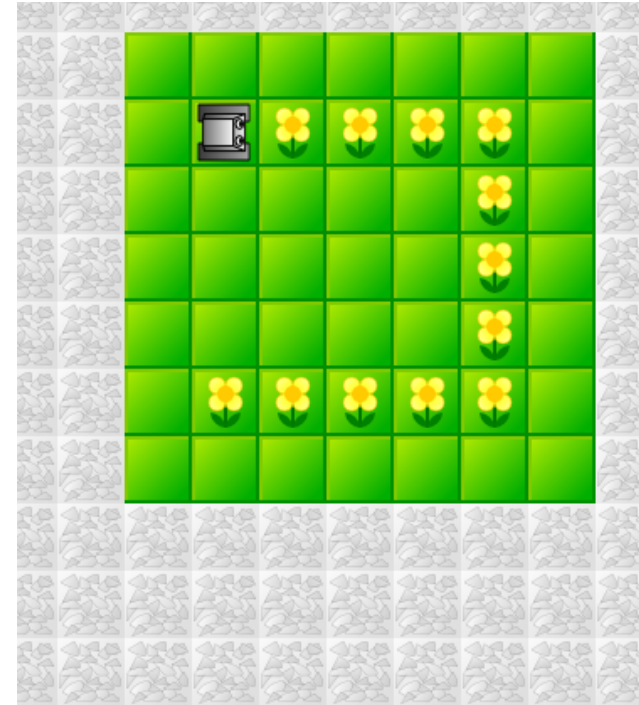


- Krok dopředu a opakovat 😊

F1  F1

# Trénink funkcí

- Funkce F2 bude provádět postup vpřed
- Funkce F1 se postará o zatočení a zavolání F2
- Řešení?



# Rekurzivní funkce

- Rekurzivní funkce je **normální funkce**
  - rozdíl není syntaktický, ale pouze “architektonický”
- Rekurzivní funkce ve svém těle **volá sebe samu**
  - typicky ale **s upravenými argumenty**
  - dojde k vytvoření samostatného rámce na zásobníku
    - separátní kopie lokálních proměnných pro každé funkční volání
  - po skončení vnořené funkce se pokračuje v aktuální
  - vzniká opakované rekurzivní zanoření volání
- Je nutné vnoření zastavit - **rekurzivní zarážka**
  - volání funkce, která již nezpůsobí opětovné vnořené volání
  - pokud nezastavíme, dojde k vyčerpání paměti a pádu programu
- Často kombinace výsledku z vnořené funkce s aktuálním



# Výpis pole rekurzivně

rekurzivní zarážka  
pokud offset  $\geq$  length, tak  
se neprovede další vnoření

```
#include <stdio.h>
void tisk(int* array, int offset, int length) {
    if (offset < length) {
        printf("%d ", array[offset]);
        tisk(array, offset + 1, length);
    }
}

int main() {
    const int len = 5;
    int array[len];
    for (int i = 0; i < len; ++i) array[i] = i;
    tisk(array, 0, len);
    return 0;
}
```

výpis aktuálního prvku pole

rekurzivní volání nad  
menšími daty

# Faktoriál – části programu

```
int fact(int N) {  
    if (N < 2) return 1;  
    else return N * fact(N-1);  
}
```

rekurzivní zarážka

kombinace aktuálního a  
vnořeného výsledku

rekurzivní volání nad  
menšími daty

# Zásobník (a rekurze)

```
int fact(int N) {  
    if (N < 2) return 1;  
    else return N * fact(N-1);  
}
```

return\_exit

main() {fact(5)}

int N = 5  
return\_addr1

5 \* fact(4)

fact(5)

int N = 4  
return\_addr2

4 \* fact(3)

fact(4)

int N = 3  
return\_addr3

3 \* fact(2)

fact(3)

int N = 2  
return\_addr4

2 \* fact(1)

fact(2)

int N = 1  
return\_addr5

return 1;

fact(1)

rekurzivní zarážka

# Zásobník v Robotanikovi

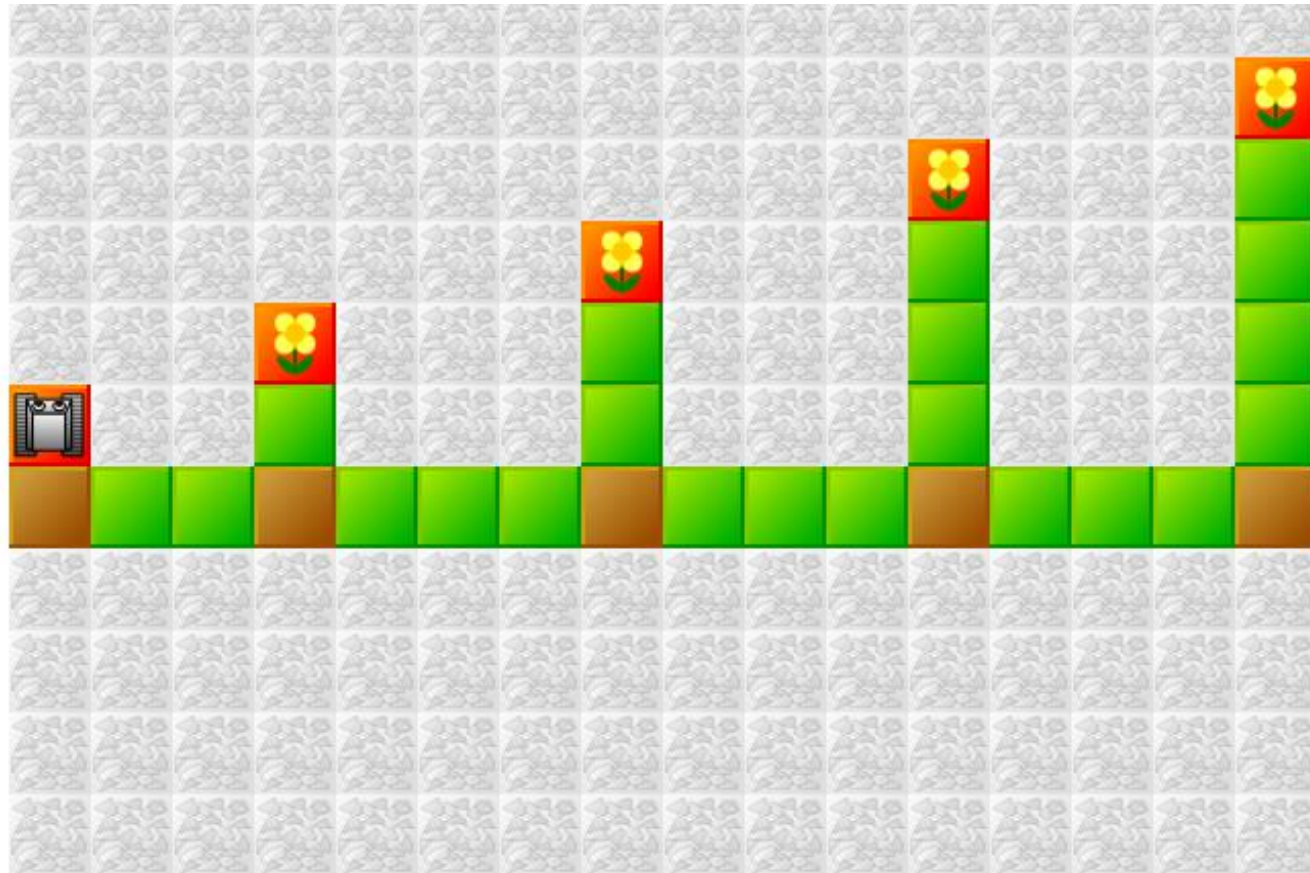
The screenshot shows the Robotanik game interface. At the top is a 10x10 grid. A robot is at (1, 5) and flowers are at (1, 6), (1, 7), (1, 8), (2, 8), (3, 8), (4, 8), (5, 8), (5, 9), (5, 10), and (6, 9). Below the grid is a command stack with four rows of buttons: Row 1: Up, Left, Right, F1, F2; Row 2: F1, F2, Right, F2, Right, F2; Row 3: F2, Up, Up (highlighted with a red box), Up, Up; Row 4: Right, F2, Right, F2. A green arrow points from the text 'aktuální stav zásobníku volání' to the highlighted 'Up' button. To the right of the grid is a 'Rychlost animace:' slider set to 1, and three buttons: Play/Pause, Step, and Stop. At the bottom right are three checkboxes: 'Zobrazit zásobník' (checked), 'Klávesové zkratky' (unchecked), and 'Přesun více příkazů' (unchecked).

aktuální stav zásobníku volání

Rychlost animace:  
1 2 3 4 5

Zobrazit zásobník  
Klávesové zkratky  
Přesun více příkazů

# Trochu složitější na pět



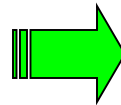
F1



# Rekurzivní funkce – kdy použít

- Funkční volání vyžaduje režii (paměť a CPU)
  - předání argumentů, předání návratové hodnoty, úklid zásobníku...
  - při opakovaném hlubokém zanoření výrazné
- Rekurzivní funkci lze přepsat do nerekurzivního zápisu
  - může se snížit přehlednost
  - typicky se zvýší rychlost a sníží paměťová náročnost za běhu
  - a naopak (mají stejnou vyjadřovací sílu)

```
int fact(int N) {  
    if (N > 1) return N * fact(N-1);  
    else return 1;  
}
```



```
int fact(N) {  
    int res = 1;  
    for (int i=N; i > 1; i--)  
        res *= i;  
    return res;  
}
```

# Ladění rekurzivních funkcí

- Využití zásobníku volání (call stack)
- Využití podmíněných breakpointů
  - na zajímavou vstupní hodnotu

**zásobník volání, aktuálně  
jsme v 6. zanoření**

The screenshot shows a debugger window with the following components:

- Code Editor:** Displays the source code of `main.c`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 int fact(int N) {
5     if (N < 2) return 1;
6     else return N * fact(N-1);
7 }
8
9
10 int main() {
11     usleep(5000);
```
- Call Stack:** A table showing the sequence of function calls.

Level	Function	File	Line	Address
0	fact	main.c	5	0x4013df
1	fact	main.c	6	0x4013f8
2	fact	main.c	6	0x4013f8
3	fact	main.c	6	0x4013f8
4	fact	main.c	6	0x4013f8
5	main	main.c	16	0x40144c
- Variable Window:** Shows the current state of variables.

Name	Value	Type
N	3	int

# Podmíněný breakpoint, N == 3

The image shows a Qt Creator IDE window with a C program and an 'Edit Breakpoint Properties' dialog box.

**Code in main.c:**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 int fact(int N) {
5     if (N < 2) return 1;
6     else return N * fact(N-1);
7 }
8
9
10 int main() {
11     usleep(5000);
12     int value = 0;
13     printf("Insert value: ");
14     scanf("%d", &value);
15
16     printf("\nResult: %d\n", fact(value));
17
18     return 0;
19 }
```

**Edit Breakpoint Properties Dialog:**

- Basic**
  - Breakpoint type: File name and line number
  - File name: C:\Qt\qtcreator-2.4.1\test5\main.c
  - Line number: 5
  - Enabled: ☒
  - Address:
  - Expression:
  - Function:
- Advanced**
  - Tracepoint only: ☐
  - Path: Use Engine Default
  - Module:
  - Command:
  - Message:
  - Condition: N==3
  - Ignore count: 0
  - Thread specification: (all)

Buttons: OK, Cancel



# Využití knihoven

# Koncept využívání knihoven

1. Kód z knihovny je přímo zahrnut do kódu aplikace
  - každá aplikace bude obsahovat duplicitně kód knihovných funkcí
  - aplikace nevyžaduje ke svému spuštění dodatečné soubory s knihovními funkcemi
2. Přeložený kód knihovny je v samostatném souboru, aplikace jen volá funkce
  - knihovna se nahrává implicitně při spuštění aplikace
  - knihovna se nahrává explicitně v kódu

# Proces linkování

- **Statické linkování** - probíhá během překladač
  - kód z knihovny je začleněn do kódu aplikace (např. `printf()`)
- **Dynamické linkování** – probíhá za běhu aplikace
  - v době překladač nemusí být znám přesný kód, který bude spuštěn
- **Statické linkování sdílených knihoven**
  - aplikace očekává přítomnost externích souborů se spustitelným kódem knihovnických funkcí
  - Windows: `library.dll`, Unix: `library.so`
- **Dynamické nahrávání sdílených knihoven**
  - aplikace sama otevírá externích soubor se spustitelným kódem knihovnických funkcí
  - Windows:  
`LoadLibrary()`, `GetProcAddress()`, `FreeLibrary()`
  - Unix: `dlopen()`, `dlsym()`, `dlclose()`

# Jakou knihovnu vybrat?

- Preferujte funkce ze standardní knihovny
  - snadno přístupné a přenositelné
- Vyváženost použité vs. nepoužité funkčnosti
  - OpenSSL pouze pro výpočet MD5 je zbytečné
- Preferujte knihovnu s abstrakcí odpovídající vašim požadavkům
  - pro stáhnutí http stránky není nutné využívat detailní API
- Preferujte menší počet použitých knihoven
  - každá knihovna má vlastní styl API => dopad na čitelnost vašeho kódu
- Pro malou konkrétní úlohu může být nejsnazší vyjmout a upravit kód

# Ověřte vhodnost licence u knihovny!

- BSD-like, MIT, public domain
  - můžete použít téměř libovolně, uvést jméno autora
- GPL, Lesser-GPL
  - pokud použijete, musíte zveřejnit i vaše upravené zdrojové kódy
- Proprietární licence
  - dle podmínek licence, typicky není nutné zveřejnit váš kód
- Duální licencování
  - kód typicky dostupný jako open source (např. GPL), při poplatku jako proprietární licence
- [http://en.wikipedia.org/wiki/Comparison\\_of\\_free\\_software\\_licenses](http://en.wikipedia.org/wiki/Comparison_of_free_software_licenses)

# Standardní knihovna C99

# Standardní knihovna jazyka C (C99)

- Celkem 24 hlavičkových souborů
- [http://en.wikipedia.org/wiki/C\\_standard\\_library](http://en.wikipedia.org/wiki/C_standard_library)
  - <stdio.h>
  - <stdlib.h>
  - <ctype.h>
  - <assert.h>
  - <stdarg.h>
  - <time.h>
  - <math.h>
  - ...

# <limits.h> konstanty limitů datových typů

- <http://en.wikipedia.org/wiki/Limits.h>
- Potřebné pro kontrolu rozsahů hodnot primitivních datových typů
  - použití pro kontrolu přetečení čítačů, mezivýsledků, očekávatelná přesnost výpočtu...
  - na různých platformách se mohou lišit
- Rozdíl mezi znaménkovým a neznaménkovým typem
  - např. `UINT_MAX` (+4,294,967,295) vs. `INT_MAX` (+2,147,483,647)
- `CHAR_BIT` – počet bitů v jednom `char`
  - typicky 8 bitů, ale např. DSP mají často 16 a víc
- Možný rozdíl mezi překladem pro 32/64 bitovou architekturu
  - `LONG_MAX` (32bit) == +2,147,483,647
  - `LONG_MAX` (64bit) == +9,223,372,036,854,775,807



# <ctype.h> - zjištění typu znaku

- isalnum(c) – číslo nebo písmeno
- iscntrl(c) – řídící znaky
  - ASCII kód 0x00 - 0x1f, 0x7f
- isdigit(c) – čísla
- islower(c) – malá písmena ('a' – 'z')
- isprint(c) – tisknutelné znaky
- ispunct(c) – interpunkční znamínka (, ; ! ...)
- isspace(c) – bílé znaky (mezera, \t \n ...)
- isupper(c) - velká písmena ('A' – 'Z')
- Některé znaky mohou splňovat více podmínek
  - např. iscntrl('\n'), isspace('\n')
- Knihovna <wctype.h> pro široké znaky (wisalnum()...)

# <string.h> - funkce pro práci s pamětí

- string.h obsahuje funkce pro práci s řetězcí (strcpy, strlen...)
- Navíc obsahuje rychlé funkce pro práci s pamětí
  - operace nad pamětí začínající na adrese X o délce Y bajtů
- **void \*memset(void \*, int c, size\_t n);**
  - nastaví zadanou oblast paměti na znak C
- **void \*memcpy(void \*dest, const void \*src, size\_t n);**
  - zkopíruje ze src do dest n bajtů
- **int memcmp(const void \*s1, const void \*s2, size\_t n);**
  - porovná bloky paměti na bajtové úrovni (stejně => 0)
  - *?Jak se liší memcpy a strcpy?*
- Pracuje na úrovni bajtů – je nutné spočítat velikost položek
  - **memset(intArray, 0, intArrayLen \* sizeof(int))**
  - při dynamické alokaci máte délku v bajtech typicky spočtenou
- Výrazně rychlejší než práce v cyklu po prvcích
  - kopírování paměťových bloků bez ohledu na sémantiku uložené hodnoty

# <time.h> Práce s časem

```
struct tm {  
    int tm_sec; /* Seconds: 0-59 (K&R says 0-61?) */  
    int tm_min; /* Minutes: 0-59 */  
    int tm_hour; /* Hours since midnight: 0-23 */  
    int tm_mday; /* Day of the month: 1-31 */  
    int tm_mon; /* Months *since* january: 0-11 */  
    int tm_year; /* Years since 1900 */  
    int tm_wday; /* Days since Sunday (0-6) */  
    int tm_yday; /* Days since Jan. 1: 0-365 */  
    int tm_isdst; /* +1 Daylight Savings Time, 0 No  
                  DST, -1 don't know */};
```

- Funkce pro získávání času
  - time() – počet sekund od 00:00, 1. ledna, 1970 UTC
  - clock() – „tiky“ od začátku programu (obvykle ms)
- Funkce pro konverzi času (sekundy → struct tm)
  - gmtime(), mktime()
- Formátování času a výpis
  - ctime() – lidsky čitelný řetězec, lokální čas
  - asctime() – lidsky čitelný řetězec, UTC
  - strftime() – formátování do řetězce dle masky
    - %M minuty ...

# Zobrazení času - ukázka

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

int main(void) {
    time_t timer = time(NULL);
    printf("ctime is %s\n", ctime(&timer));

    struct tm* tmTime = gmtime(&timer);
    printf("UTC is %s\n", asctime(tmTime));

    tmTime = localtime(&timer);
    printf("Local time is %s\n", asctime(tmTime));

    const int shortDateLen = 20;
    char shortDate[shortDateLen];
    strftime(shortDate, shortDateLen, "%Y/%m", tmTime);
    puts(shortDate);

    return 0;
}
```

```
ctime is Mon May 16 09:36:05 2011
UTC is Mon May 16 07:36:05 2011
Local time is Mon May 16 09:36:05 2011
2011/05
```

# Měření času operace

- `time()` vrací naměřený čas s přesností 1 sekundy
  - není většinou vhodné na přesné měření délky operací
- Přesnější měření možné pomocí funkce `clock()`
  - vrátí počet "tiků" od startu programu
  - makro `CLOCKS_PER_SEC` definuje počet tiků za sekundu
    - např. 1000 => přesnost řádově na milisekundy
- Pro krátké operace to většinou nestačí
  - lze řešit provedením operace v cyklu s velkým množstvím iterací
- Pozor na optimalizace překladače
  - chceme měřit rychlost v Release
  - některé části kódu mohou být zcela odstraněny (nepoužité proměnné) nebo vyhodnoceny při překladu (konstanty)
- Pozor na vliv ostatních komponent (cache...)

```

#include <stdio.h>
#include <time.h>

int main(void) {
    const int NUM_ITER = 1000000000;
    double powValue = 0;
    double base = 3.1415;
    // Run function once to remove "first run effects"
    powValue = base * base * base;

    // Run function in iteration
    clock_t elapsed = -clock();
    for (int i = 0; i < NUM_ITER; i++) {
        powValue = base * base * base;
    }
    elapsed += clock();

    printf("Total ticks elapsed: %ld\n", elapsed);
    printf("Ticks per operation: %f\n", elapsed / (double) NUM_ITER);
    printf("Operations per second: %ld\n",
        round(CLOCKS_PER_SEC / (elapsed / (double) NUM_ITER)));

    return 0;
}

```

#### DEBUG build

Total ticks elapsed: 2995

Ticks per operation: 0.000003

Operations per second: 402653184

#### RELEASE build

Total ticks elapsed: 0

Ticks per operation: 0.000000

Operations per second: 0

# Pokročilé profilery: MS Visual Studio

pb173_aes101115.vsp × time.h aes32.h pb173_aes.cpp				
Current View: Functions				
Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
[pb173_aes.exe]	5	5	0.29	0.29
__RTC_CheckEsp	1	1	0.06	0.06
__tmainCRTStartup	1,740	0	100.00	0.00
_main	1,740	0	100.00	0.00
_mainCRTStartup	1,740	0	100.00	0.00
aes_addRoundKey(unsigned char *, unsigned char *)	10	10	0.57	0.57
aes_expandEncKey(unsigned char *, unsigned char *)	322	1	18.51	0.06
aes_mixColumns(unsigned char *, unsigned char *)	26	10	1.49	0.57
aes_shiftRows(unsigned char *, unsigned char *)	3	3	0.17	0.17
aes_subBytes(unsigned char *, unsigned char *)	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct aes256_ctx *, unsigned char *, unsigned char *)	1,740	1	100.00	0.06
gf_alog(unsigned char)	806	806	46.32	46.32
gf_log(unsigned char)	846	846	48.62	48.62
gf_mulinv(unsigned char)	1,668	14	95.86	0.80
rj_sbox(unsigned char)	1,694	24	97.36	1.38
rj_xtime(unsigned char)	15	15	0.86	0.86
testProfile(void)	1,740	0	100.00	0.00

# LCOV - code coverage report

Current view: [top level](#) - [example/methods](#) - [iterate.c](#) ([source](#) / [functions](#))

Test: [Basic example](#) ( [view descriptions](#) )

Date: 2012-10-12

Legend: Lines: hit not hit | Branches: + taken - not taken # not executed

	Hit	Total	Coverage
Lines:	8	8	100.0 %
Functions:	1	1	100.0 %
Branches:	4	4	100.0 %

	Branch data	Line data	Source code
1		:	: /*
2		:	: * methods/iterate.c
3		:	: *
4		:	: * Calculate the sum of a given range of integer numbers.
5		:	: *
6		:	: * This particular method of implementation works by way of brute force,
7		:	: * i.e. it iterates over the entire range while adding the numbers to finally
8		:	: * get the total sum. As a positive side effect, we're able to easily detect
9		:	: * overflows, i.e. situations in which the sum would exceed the capacity
10		:	: * of an integer variable.
11		:	: *
12		:	: */
13		:	:
14		:	: #include <stdio.h>
15		:	: #include <stdlib.h>
16		:	: #include "iterate.h"
17		:	:
18		:	:
19		3 :	int iterate_get_sum (int min, int max)
20		:	{
21		:	int i, total;
22		:	:
23		3 :	total = 0;
24		:	:
25		:	/* This is where we loop over each number in the range, including
26		:	both the minimum and the maximum number. */
27		:	:
28	[ + + ]	67548 :	for (i = min; i <= max; i++)
29		:	{
30		:	/* We can detect an overflow by checking whether the new
31		:	sum would become negative. */
32		:	:
33	[ + + ]	67546 :	if (total + i < total)
34		:	{
35		1 :	printf ("Error: sum too large!\n");
36		1 :	exit (1);
37		:	}
38		:	:
39		:	/* Everything seems to fit into an int, so continue adding. */
40		:	:
41		67545 :	total += i;

Taken from <http://ltp.sourceforge.net/coverage/lcov/output/example/methods/iterate.c.gcov.html>



# <stdlib.h>

- Funkce pro konverzi typů z řetězce do čísla
  - atoi, atof...
- Generování pseudonáhodných sekvencí
  - deterministická sekvence čísel z počátečního semínka
  - **void** srand(**unsigned int** seed); - nastavení semínka
    - často srand(time(NULL))
  - **int** rand(**void**); - další náhodné číslo v sekvenci
    - rand() vrací číslo z rozsahu 0 do RAND\_MAX ( $\geq 32767$ )
    - do rozsahu 0 až 9 převedeme pomocí rand() % 10
  - Pozor, rand není vhodný pro generování hesel apod.!
- Široké využití: výběr pivota, náhodné doplnění (síťové pakety), IV, generování bludiště...
- Dynamická alokace (malloc, free...)

# <stdlib.h>

- Funkce pro spouštění a kontrolu procesů
  - **int** system (**const char\*** command);
    - vykoná externí příkaz, jako kdyby bylo zadáno v příkazové řádce, např. system("cls")
  - **char\*** getenv (**const char\*** name);
    - získá systémovou proměnnou, např. getenv("PATH")
    - nebo libovolnou nastavenou uživatelem
- Některé matematické funkce
  - abs(), div()
  - více v knihovně math.h

# Ukázka system() – zjištění obsahu adresáře

- Trik na zjištění obsahu nadřazeného adresáře

```
void printDir() {
    // Parent directory printing for poor (without POSIX)
    // NOTE: Posix functions provides significantly better way
    system("cd .. & dir > list.tmp"); // use ls on linux
    FILE* file = NULL;
    char mystring[100];

    if ((file = fopen("../list.tmp", "r")) != NULL) {
        while (fgets(mystring, 100, file) != NULL) {
            // Parsing would be necessary to obtain dirs
            // We just output the line
            puts(mystring);
        }
        fclose(file);
    }
    system("notepad.exe ../list.tmp");
}
```

# Řadící a vyhledávací funkce

- Standardní knihovna obsahuje řadící a vyhledávací funkci
  - quicksort – řadí posloupnost prvků v poli
  - rychlé vyhledávání v seřazeném poli (půlení intervalu)

začátek pole

počet prvků v poli

velikost jedné položky

```
void qsort (void* base, size_t n, size_t sz,  
            int (*cmp) (const void*, const void*))
```

srovnávací funkce

```
void *bsearch(const void*key, const void*base, size_t nmemb, size_t size,  
              int (*cmp) (const void *, const void *));
```

# Řadící funkce qsort

- qsort() je řadící funkce implementující quick sort algoritmus s průměrnou složitostí  $O(n \cdot \log(n))$
- Algoritmus je nezávislý na řazených položkách
  - můžeme řadit pole struktur, komplexní čísla, řetězce...
  - proto je třetí argument délka položky (qsort nemusí „rozumět“ položkám)
- Je nutné poskytnout funkci pro srovnání dvou položek (callback)
  - hlavička funkce je vždy **int xxx(const void \* a, const void \* b)**
    - ukazatel na položku použit pro zamezení kopírování velké hodnoty
    - **const void \*** pro srovnání libovolné hodnoty (přetypujete si)
  - pokud  $a < b$ , tak vrátí  $< 0$ , pokud  $a > b$ , tak vrátí  $> 0$

```
int compareInt(const void * a, const void * b) {  
    return ( *(int*)a - *(int*)b );  
}  
  
int compareString(const void * a, const void * b) {  
    return (strcmp((char*)a, (char*)b));  
}  
  
qsort(values, arrayLength, sizeof(int), compareInt);
```

# Demo quicksort(qsort) vs. bubblesort $O(n^2)$

```
// NOTE: Code excerpt only!!!
const int arrayLength = 100000;
int compare (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}
int bubblesort(int* array, int low, int high) {
    // ...Two nested for cycles
    return 0;
}
int main () {
    const int arraySize = arrayLength * sizeof(int);
    int* values = NULL;
    values = (int*) malloc(arraySize);

    // Generate random array
    srand((unsigned)time(NULL));
    for (int i = 0; i < arrayLength; i++) values[i]=rand();

    // Measure real time
    clock_t elapsed = -clock();
    bubblesort(values, 0, arrayLength-1);
    elapsed += clock();
    // ... print results (see full source code)
    // ... restore original values array etc.
    elapsed = -clock();
    qsort(values, arrayLength, sizeof(int), compare);
    elapsed += clock();
    // ... print results (see full source code)
    if (values) delete[] values;
    return 0;
}
```

Total ticks elapsed: 20878

Ticks per sorted item: 0.208780

Sorted items per second: 4790

Total ticks elapsed: 19

Ticks per sorted item: 0.000190

Sorted items per second: 5263158

# Vyhledávací funkce bsearch

- “Binární” vyhledávání
  - předpokládá seřazenou posloupnost prvků v poli
    - např. pomocí qsort
- Hledá půlením intervalu (proto je rychlé)
  - a proto musí být posloupnost seřazena
  - *jaká je očekávaná složitost?*
- Hledaný prvek je zadán ukazatelem void\*
  - pro typovou nezávislost a rychlost předání
- Opět využití callback funkce pro porovnání prvků
  - stejně jako u qsort

```
void *bsearch(const void*key, const void*base, size_t nmemb, size_t size,  
             int (*cmp)(const void *, const void *));
```

# <math.h> - matematické funkce

- Matematické konstanty
  - M\_PI, M\_SQRT2, M\_E ...
- Základní matematické funkce
  - sin, cos, pow, sqrt, log, log10...
- Zaokrouhlovací funkce
  - ceil, floor, round
- Od C99 další rozšíření
  - trunc, fmax, fmin...
  - rozšíření návratového typu až na long long (llround)
- Implementace budou pravděp. rychlejší, než vaše vlastní
  - ale např. pow(a, 3) pro a == 8 bitů lze rychleji
  - (precomputed tables)

powTable[0]	0
powTable[1]	1
powTable[2]	8
powTable[3]	27
powTable[4]	...



# Přepínač -l pro začlenění knihovny

- Knihovna math vyžaduje explicitní začlenění při linkování
- Parametr při překladu `-lknihovna`
  - např. pro začlenění `gcc -std=c99 hello.c -lm`
- Externí knihovny vyžadují začlenění při linkování
  - např. knihovna pro práci s "grafickým" textem PDCourses (`-lpdcurses`)
- Nastavení v QT Creator
  - `project.pro` -> `LIBS += -lpdcurses`
- Nastavení v Code::Blocks
  - Settings-> Compiler and debugger... -> Linker settings-> Add

# PB071 Prednaska 10 – rekurze, stdlib



# ISO/IEC 9899:2011 (C11)

# ISO/IEC 9899:2011 (C11)

- Nejnovější verze standardu (2011)
  - [http://en.wikipedia.org/wiki/C11\\_\(C\\_standard\\_revision\)](http://en.wikipedia.org/wiki/C11_(C_standard_revision))
  - přidány drobné rozšíření jazyka
  - přidány některé funkce dříve dostupné jen v POSIXu
- Pěkný souhrn motivací a změn
  - <http://www.jauu.net/data/pdf/c1x.pdf>
- Vyzkoušení na Aise:
  - module add gcc-4.7.2
  - gcc -std=c11
  - GCC zatím nepodporuje všechny nové vlastnosti
- (Zatím nejrozšířenější zůstává použití C99)

# Vlákná

- `#include <threads.h>`
- Velmi podobné vláknům v POSIXu (snadný přechod)
  - `pthread_create` -> `thr_create`
  - `pthread_mutex_init` -> `mtx_init`
- Specifikace lokální proměnné ve vlákně `_Thread_local`
  - lokální proměnná ve funkci spuštěné paralelně v několika vláknech
- `_Thread_local` storage-class
- `_Atomic` type qualifier, `<stdatomic.h>`
- Metody pro synchronizaci
- Atomické operace `_Atomic int foo;`

# Atomičnost operací a paměti

- Je `i++` atomické?
  - není, je nutné načíst, zvětšit, uložit
  - u vícevláknového programu může dojít k prolnutí těchto operací
    - načte se hodnota, která ale již nebude po zvětšení aktuální – jiné vlákno uložilo zvětšenou hodnotu `i`
- `atomic_{load,store,exchange}`
- `atomic_fetch_{add,sub,or,xor,and}`
- `atomic_compare_exchange_`

# Exkluzivní otevření souboru - motivace

- Např. MS Word při editaci souboru soubor.doc vytváří ~\$soubor.doc
  - při pokusu o otevření souboru soubor.doc vždy kontroluje, zda se mu podaří vytvořit a otevřít ~\$soubor.doc
  - pokud ne, soubor soubor.doc je již editován
- Po ukončení programu se zámek ruší
  - korektní ukončení většinou smaže i soubor se zámkem
  - náhlé ukončení ponechá soubor, ale již neblokuje přístup

# Exkluzivní režim otevření souboru

- Jak zjistíme, že soubor (zámku) již existuje?
  - otevři pro čtení
  - když selže, tak vytvoř a otevři pro zápis
  - race condition mezi čtením a vytvořením
  - potřebovali bychom “selži pokud existuje, jinak otevři na zápis”
- Dodatečný režim otevření souboru `fopen("cesta", "wX")`
  - vytvoř a otevři exkluzivně
  - selže pokud již existuje a někdo jej drží otevřený
- Typické využití pro soubory signalizující zámek (lock files)
  - pokud je aplikace spuštěna vícekrát, tak detekuje soubory, které jsou již používány
- Ekvivalentní POSIX příkazu `open(O_CREAT | O_EXCL)`



# Typově proměnná makra

- Vyhodnocení makra v závislosti na typu proměnné (Type-generic expressions)
- klíčové slovo **\_Generic**

```
#define FOO(X) myfoo(X)
```

```
#define FOO(X)  
    _Generic((X), long: fool, char: fooc, default foo) (X)
```

# Vylepšená podpora Unicode (UTF-16/32)

- `char16_t` and `char32_t`
- `<uchar.h>`

# Bezpečné varianty některých funkcí

- Funkce z (Secure C Library)
  - [http://docwiki.embarcadero.com/RADStudio/XE3/en/Secure\\_C\\_Library](http://docwiki.embarcadero.com/RADStudio/XE3/en/Secure_C_Library)
  - <http://msdn.microsoft.com/en-us/library/8ef0s5kh%28v=vs.80%29.aspx>
  - <http://www.drdobbs.com/cpp/the-new-c-standard-explored/232901670>
- fopen\_s, fprintf\_s, strcpy\_s, strcat\_s, gets\_s...
  - typicky kontrola délky paměti na ochranu před zápisem za konec alokované paměti (buffer overflow)
- gets() deprecated v C99, nyní úplně odstraněna

# Makra pro zjištění možností prostředí

- `__STDC_VERSION__`
  - makro pro zjištění verze, 201112L je C11

# Shrnutí

- **Rekurze**
  - Důležitý mentální koncept
  - Často využíván ve funkcionálních jazycích
- **Funkční ukazatele a callback**
  - důležitý a široce používaný koncept
  - umožňuje ošetřovat asynchronní události
  - obsahuje typovou kontrolu (argumenty a návrat. hodnota)
- **Způsob začlenění knihoven je různý**
  - ovlivňuje způsob použití
- **Standardní knihovna C99**
  - velká řada běžných funkcí (včetně řazení a vyhledávání v poli)
  - přenositelnost

# Bonus 😊

# Webová služba: opakovač paketů

```
network_receive(in_packet, &in_packet_len); // TLV packet
in = in_packet + 3;
```

`unsigned char* in`



Type [1B]

length [2B]

Payload [length B]

```
out_packet = malloc(1 + 2 + length);
out = out_packet + 3;
```

```
memcpy(out, in, length);
```

`unsigned char* out`



Type [1B]

length [2B]

Payload [length B]

```
network_transmit(out_packet);
```

# Problém?

```
network_receive(in_packet, &in_packet_len); // TLV packet  
in = in_packet + 3;
```

`unsigned char* in`

Type [1B]

0xFFFF [2B]

Payload [1B]

... Heap memory ...

```
out_packet = malloc(1 + 2 + length);  
out = out_packet + 3;
```

```
memcpy(out, in, length);
```

`in_packet_len != length + 3`

`unsigned char* out`

Type [1B]

0xFFFF [2B]

Payload [1B]

Heap memory (klíče, hesla...)

```
network_transmit(out_packet);
```

## Problém!





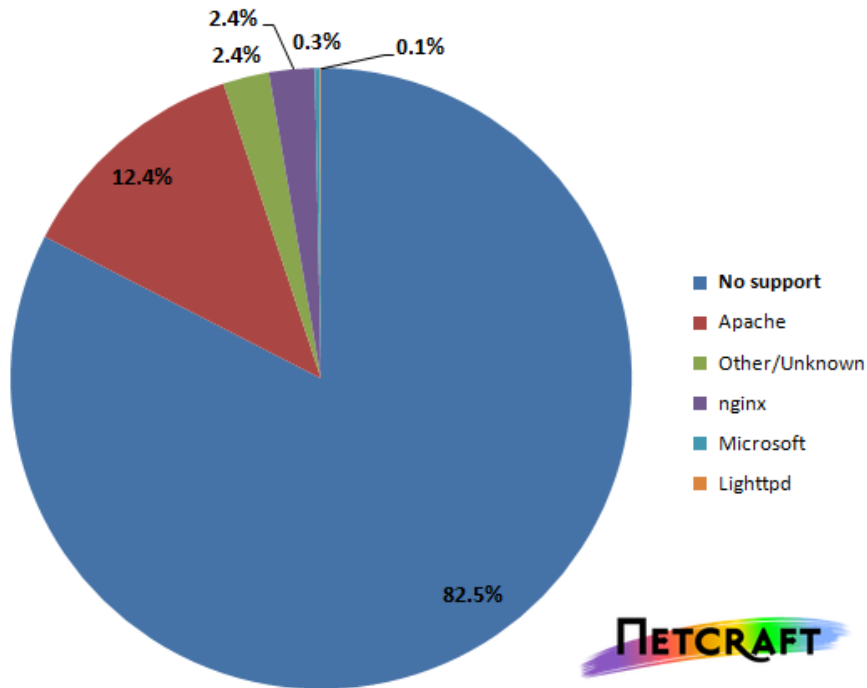
# O jak závažnou chybu se jedná?



**17% SSL web serverů (OpenSSL 1.0.1)**

[Twitter](#), [GitHub](#), [Yahoo](#), [Tumblr](#), [Steam](#), [DropBox](#), [DuckDuckGo](#)...  
<https://seznam.cz>, <https://fi.muni.cz> ...

TLS Heartbeat Extension Support by IP Address



- <http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html>

# Ponaučení

- Vždy VELMI rigidně kontrolujte vstupní argumenty
- Nebezpečný není jen zápis za konec pole, ale i čtení
- Nedůvěřujte informacím od klienta
  - Ani když jste vy sami jeho tvůrci (změna na síťové vrstvě)
- Pro síťové aplikace preferujte jiné jazyky než C
  - Např. automatická kontrola mezí polí (Java, C#)
  - Nenahrazuje kontrolu argumentů!
- Open-source sám o sobě nezajišťuje kód bez chyb
  - "given enough eyeballs, all bugs are shallow" L. Torvalds
- (Nedělejte commity ve spěchu před oslavou)



## [projects](#) / [openssl.git](#) / commit

[summary](#) | [shortlog](#) | [log](#) | [commit](#) | [commitdiff](#) | [tree](#)  
(parent: [84b6e27](#)) | [patch](#)

### PR: 2658

```
author      Dr. Stephen Henson <steve@openssl.org>
            Sat, 31 Dec 2011 22:59:57 +0000 (22:59 +0000)
committer   Dr. Stephen Henson <steve@openssl.org>
            Sat, 31 Dec 2011 22:59:57 +0000 (22:59 +0000)
commit      4817504d069b4c5082161b02a22116ad75f822b1
tree        7a85f6af852e34e5b80080b50d80741f6ab36c5a    tree | snapshot
parent      84b6e277d4f45487377d0159e82c356d750e1218    commit | diff
```

PR: 2658  
Submitted by: Robin Seggelmann <seggelmann@fh-muenster.de>  
Reviewed by: steve

Support for TLS/DTLS heartbeats.

### 20 files changed:

CHANGES	<a href="#">diff</a>   <a href="#">blob</a>   <a href="#">history</a>
apps/s_cb.c	<a href="#">diff</a>   <a href="#">blob</a>   <a href="#">history</a>
apps/s_client.c	<a href="#">diff</a>   <a href="#">blob</a>   <a href="#">history</a>
apps/s_server.c	<a href="#">diff</a>   <a href="#">blob</a>   <a href="#">history</a>

# Reference

- Všeobecné informace
  - <http://heartbleed.com/>
- Testování zranitelnosti konkrétní stránky
  - <https://filippo.io/Heartbleed/>
- Analýza problému na úrovni zdrojáku
  - <http://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed>
  - <http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html>

# O jak závažnou chybu se jedná? ☺

- XKDC (<https://xkcd.com/1353/>)

