

PB071 – Programování v jazyce C

Rekurze, funkční ukazatele,
knihovny, standardní knihovna

Organizační

- 21.4. Velikonoce
 - Přednáška a pondělní cvičení odpadají
 - Zbytek týdne normálně probíhá výuka (cvičení)
- 28.4. Zvaná přednáška – Juraj Michálek
 - spousta zajímavostí souvisejících s vývojem



WE WANT YOU



K-net Technical
International
Group



 **MONET+**

 **YSOFT**

Soutěž pro zapálené studenty

- Určeno pro studenty 2. a 3. semestru Bc studia
- 5 úkolů z informatiky, 24 hodin na vyřešení
 - můžete si vybrat, který řešit
- Po vyřešení pohovor na 5-7 pozic v laboratořích
 - není třeba vyřešit všechny úkoly
- 15. 4. 9:33 D1 – rozdání první série úkolů
- 16. 4. 9:44 kancelář B308 – druhá série úkolů
- Detaily na <https://www.fi.muni.cz/~xsvenda/soutez2014.pdf>

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;  
}
```

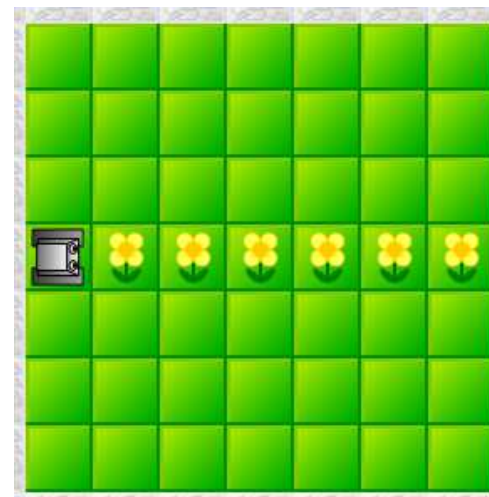
- Myšlenka řešení pomocí rekurze:

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

```
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



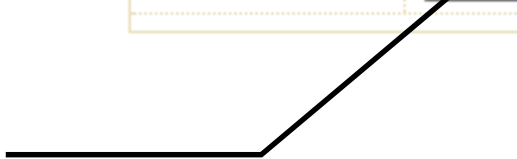
Robotanik - registrace

- Registrace může být anonymní
 - zaškrtněte ale skupinu **pb071_jaro2014**
 - umožníte nám sledovat celkovou úroveň

Pokud používáte Tutora na škole

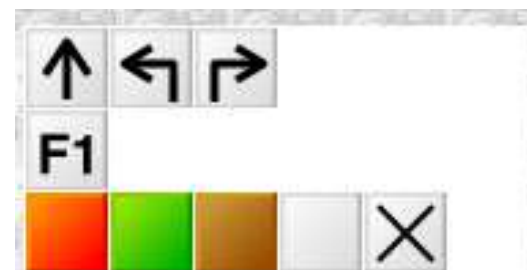
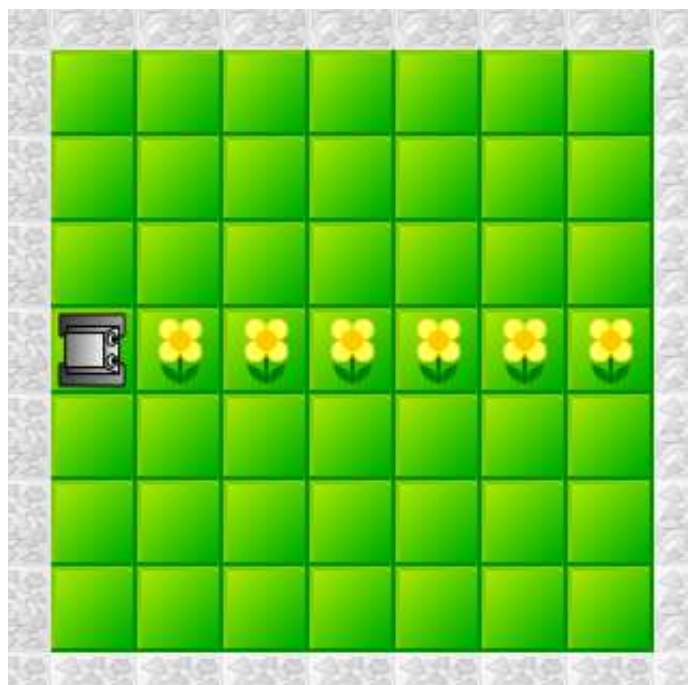
Přihlášení k výukovým skupinám	
Přidej skupinu	<input type="text" value="pb071"/> <input type="button" value="Dohledat"/>

**dodatečný
identifikátor
pb071_jaro2014**



Elementární rekurze

- Jak sebrat květinčky na dvě instrukce?

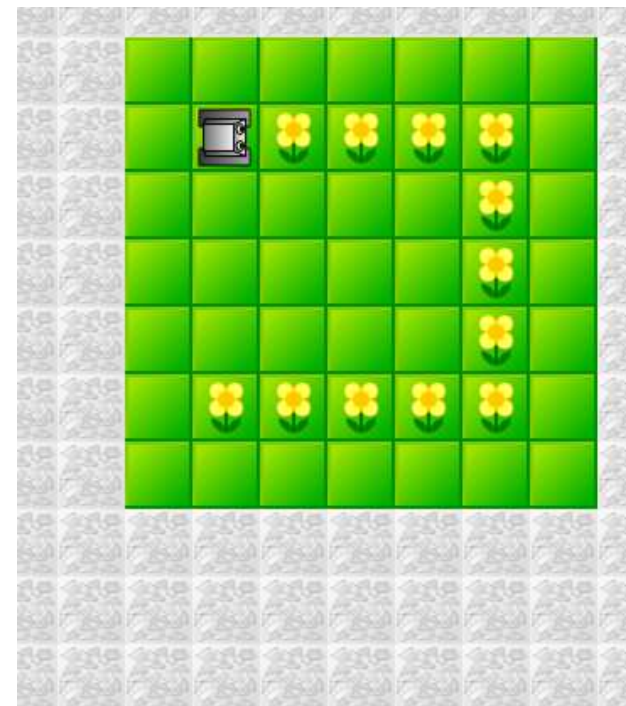


- Krok dopředu a opakovat 😊



Trénink funkcí

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



F1					
F2					

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 >= 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)

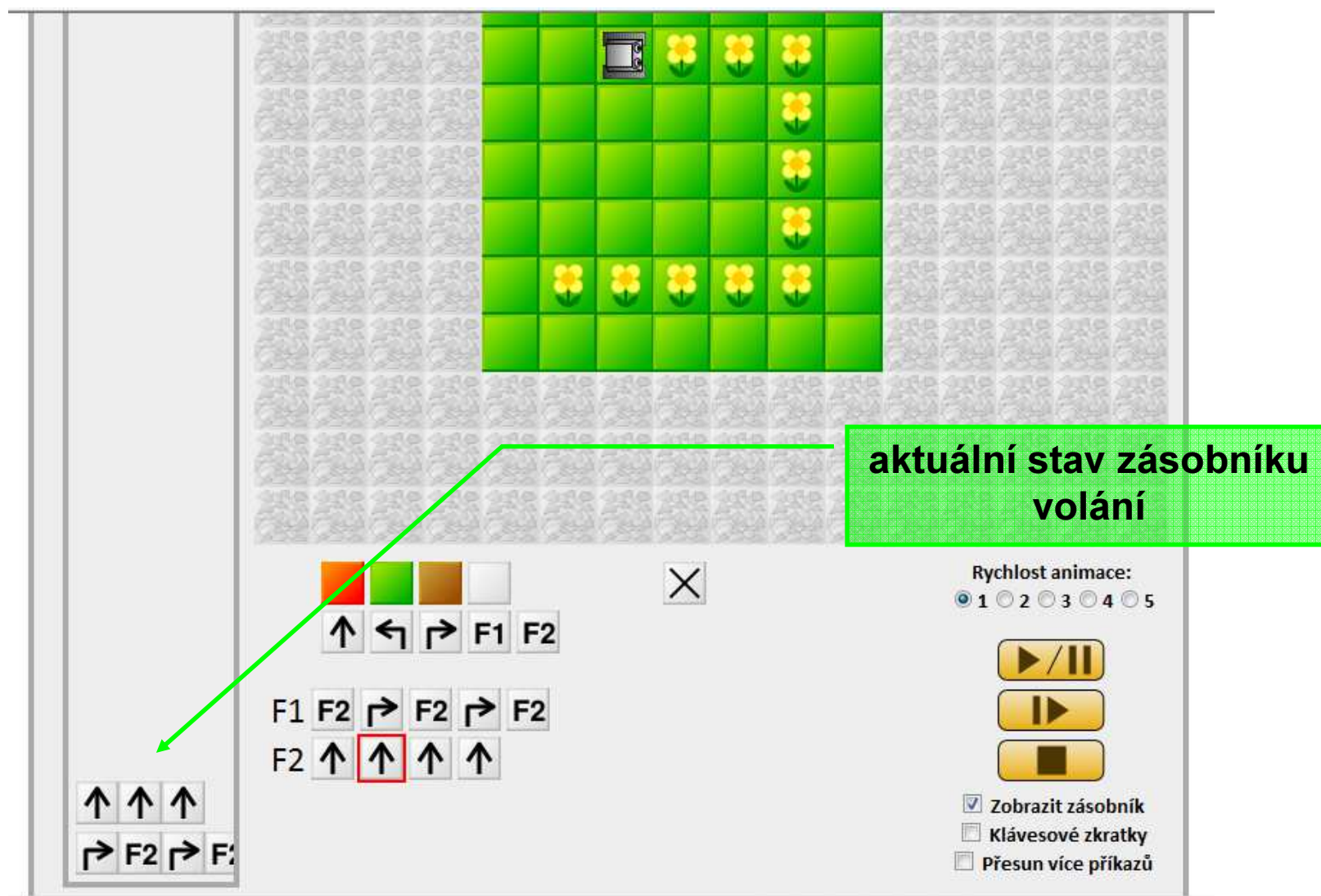
int N = 1

return_addr5

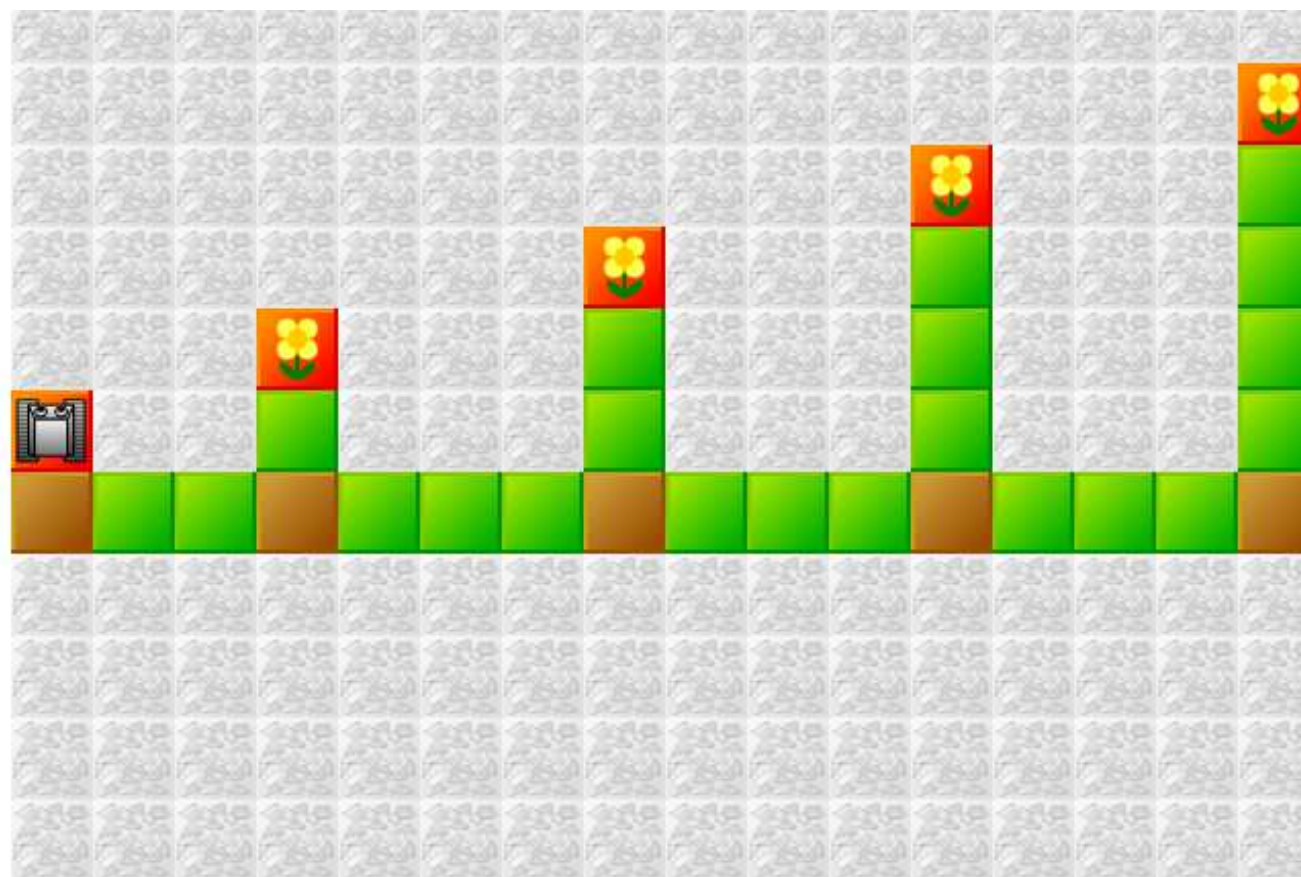
return 1;

fact(1)

Zásobník v Robotanikovi



Trochu složitější na pět



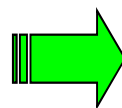
F1



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

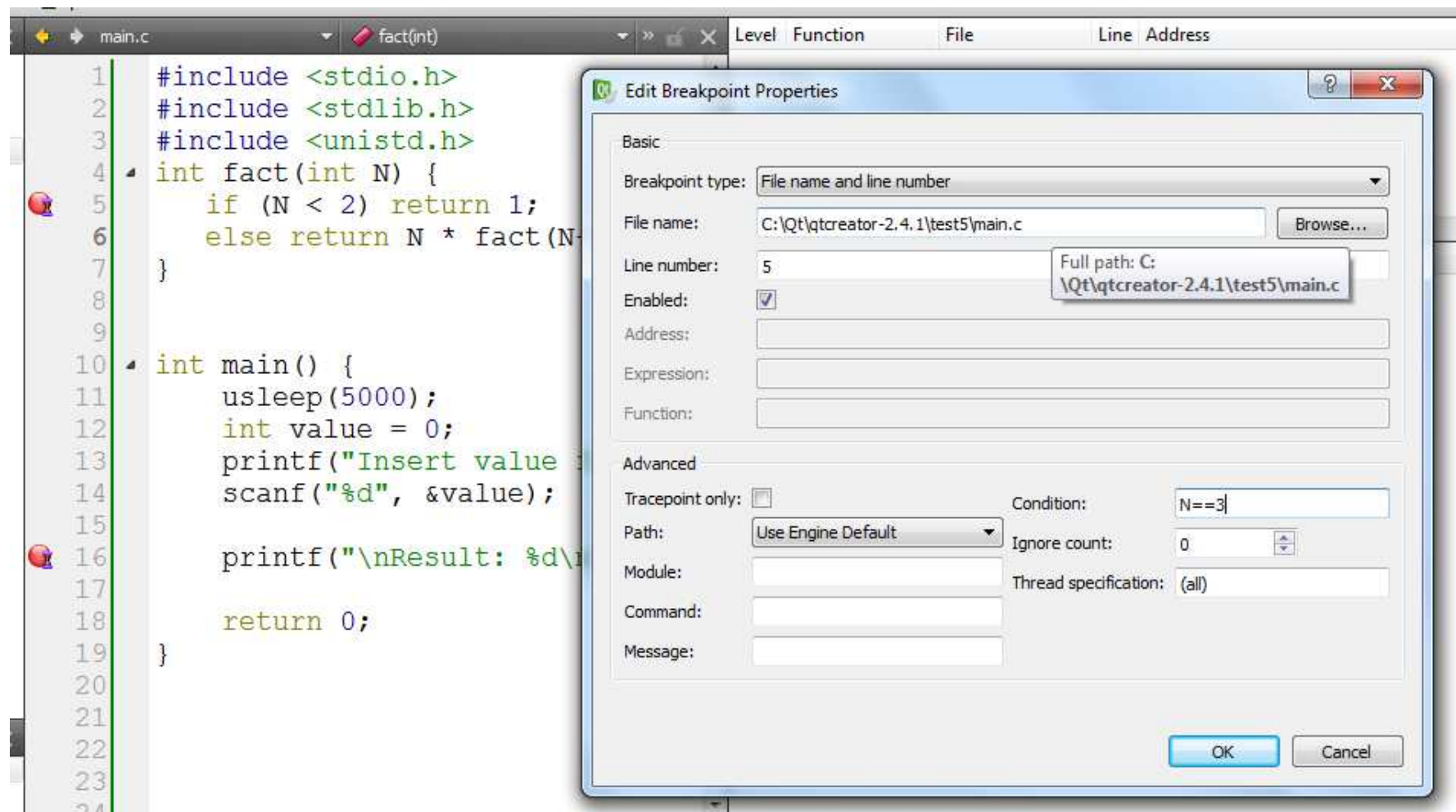
zásobník volání, aktuálně
jsme v 6. zanoření

The screenshot shows a debugger window with the source code of a C program on the left and the call stack on the right. The source code defines a recursive function `fact(int N)` and a `main` function. The call stack on the right shows the sequence of function calls. A green arrow points from the text box to the call stack entry for the 6th recursive call.

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

Name	Value	Type
N	3	int

Podmíněný breakpoint, $N == 3$



Funkční ukazatel

Funkční ukazatel - motivace

- Antivirus umožňuje provést analýzu souborů
 - funkce `int Analyze(const char* filePath);`
- Antivirus běží na pozadí a analyzuje všechny soubory při jejich čtení nebo zápisu na disk
- Jak ale antivirus zjistí, že má soubor analyzovat?
 - trvalé monitorování všech souborů je nemožné
 - vložení funkce `Analyze()` do všech programů nemožné
- Obsluha souborového systému čtení i zápis zná
 - antivirus si může zaregistrovat funkci `Analyze` na událost čtení/zápis
 - Jak registrovat? → **funkční ukazatel** (callback)

Funkční ukazatele

- Funkční ukazatel obsahuje adresu umístění kódu funkce
 - namísto běžné hodnoty je na adrese kód funkce
- Ukazatel na funkci získáme pomocí operátoru &
 - `&Analyze // bez kulatých závorek`
- Ukazatel na funkci lze uložit do proměnné
 - `návratový_typ (*jméno_proměnné) (typy argumentů)`
 - např. `int (*pAnalyze) (const char*) = &Analyze;`
- Ukazatel na funkci lze zavolat jako funkci
 - `pAnalyze("C:\\autoexec.bat");`
- Ukazatel na funkci může být parametr jiné funkce
 - `void foo(int neco, int (*pFnc) (float, float));`

Funkční ukazatel - signatura

- Důležitá je signatura funkce
 - typ a počet argumentů, typ návratové hodnoty, volací konvence
 - jméno funkce není důležité
- Do funkčního ukazatele s danou signaturou můžeme přiřadit adresy všech funkcí se stejnou signaturou

```
int Analyze (const char* filePath) {}  
int Ahoj (const char* filePath) {}  
int main(void) {  
    int (*pFnc) (const char*) = &Analyze;  
    pFnc("c:\\autoexec.bat");  
    pFnc = &Ahoj; // mozne take jako: pFnc = Ahoj;  
    pFnc("c:\\autoexec.bat");  
    return 0;  
}
```

- Neodpovídající signatury kontroluje překladač

Funkční ukazatel - využití

- Podpora „pluginů“ v systému (např. antivirus)
 - plug-in zaregistruje svůj callback na žádanou událost
 - např. antivirus na událost „přístup k souboru na disku“
 - při výskytu události systém zavolá zaregistrovaný callback
- V systému lze mít několik antivirů
 - seznam funkčních ukazatelů na funkce
 - seznam zaregistrovaných funkcí „`Analyze()`“
- Jak systém pozná, že není žádný antivirus?
 - žádný zaregistrovaný callback

Funkční ukazatel - využití

- Podpora abstrakce (např. I/O zařízení)
 - program si nastaví funkční ukazatel pro výpis hlášení
 - na obrazovku, na display, na tiskárnu, do souboru...
 - provádí volání nastaveného ukazatele, nikoli výběr funkce dle typu výstupu
 - (simulace pozdní vazby známé z OO jazyků)

```
//printScreen(), printLCD(), printPrinter(), printFile()...  
void (*myPrint) (const char*) ;  
  
myPrint = &printLCD;  
  
myPrint("Hello world");
```

Funkční ukazatel - využití

- Aplikace různých funkcí na interval prvků
 - např. simulace `for_each` známého z jiných jazyků
 - přehlednější a rychlejší než `switch` nad každým prvkem

```
int add2(int value) { return value + 2; }
int minus3(int value) { return value - 3; }

void for_each(int* array, int arrayLen, int(*fnc)(int)) {
    for (int i = 0; i < arrayLen; i++) {
        array[i] = fnc(array[i]);
    }
}

int main(){
    const int arrayLen = 10;
    int myArray[arrayLen] = {0};
    for_each(myArray, arrayLen, &add2);
    for_each(myArray, arrayLen, &minus3);

    return 0;
}
```

Funkční ukazatele – konvence volání

- Při funkčním volání musí být
 - někdo zodpovědný za úklid zásobníku (lokální proměnné)
 - definováno pořadí argumentů
- Tzv. konvence volání
 - uklízí volající funkce (cdecl) – default pro C/C++ programy
 - uklízí volaná funkce (stdcall) – např. Win32 API
- GCC: `void foo(int a, char b) __attribute__((cdecl));`
- Microsoft, Borland: `void __cdecl foo(int a, char b);`
- Při nekompatibilní kombinaci dochází k porušení zásobníku
- Více viz.
 - http://en.wikipedia.org/wiki/X86_calling_conventions
 - <http://cdecl.org/>

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 překladu
 - kód z knihovny je začleněn do kódu aplikace (např. `printf()`)
- **Dynamické linkování** – probíhá za běhu aplikace
 - v době překladu 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

Standardní knihovna C99

Standardní knihovna jazyka C (C99)

- Celkem 24 hlavičkových souborů
- 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) – řídicí 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() – „ticky“ 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



<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 (≥ 32767)
 - do rozsahu 0 až 9 převedeme pomocí rand() % 10
 - Pozor, rand není vhodný pro generování hesel apod.!
- Široké využití: výběr pivotu, 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 vrací < 0 , pokud $a > b$, tak vrací > 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);
```

100 000 hodnot v poli na seřazení

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

qsort a omezení rychlosti

- funkční ukazatele nelze inlinovat
- krátká porovnávací funkce má režii v podobě zavolání funkce z funkčního ukazatele

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`

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!



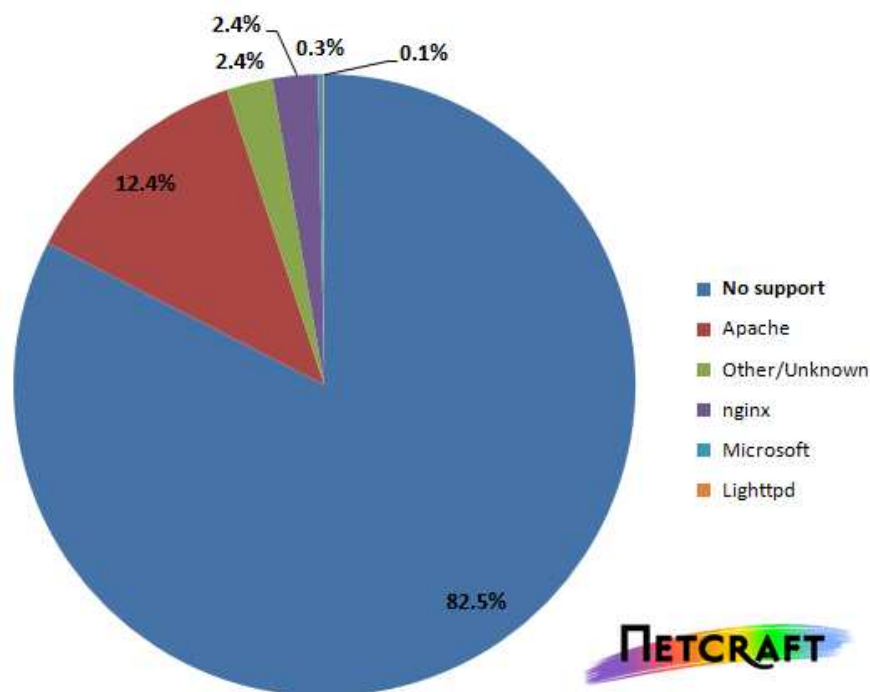
O jak závažnou chybu se jedná?



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	diff blob history
apps/s_cb.c	diff blob history
apps/s_client.c	diff blob history
apps/s_server.c	diff blob history

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/>)

