PB071 – Principy nízkoúrovňového programování

Rekurze, knihovny a licence, standardní knihovna, C11

Slidy pro komentáře (děkuji!):
https://drive.google.com/file/d/1fzWs4za6QSFDEx7VRRxqFq96V43GDc1U/view?usp=drive_link

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)*...*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 x (N-1)! a 0! == 1

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

Poznámka: časný návrat z funkce

- Rekurzivní funkci fact() z předchozího příkladu můžete přepsat i jako:

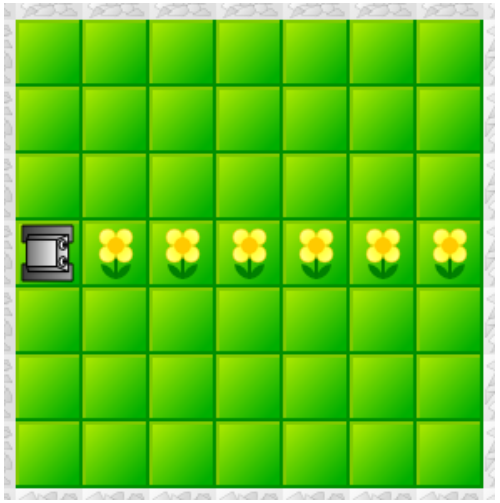
```
int fact(int N) {
    if (N <= 0) return 1;
    return N * fact(N-1);
}
```

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

- Stále se jedná o rekurzivní funkci - uvedený přepis s rekurzí nesouvisí a může být čitelnější

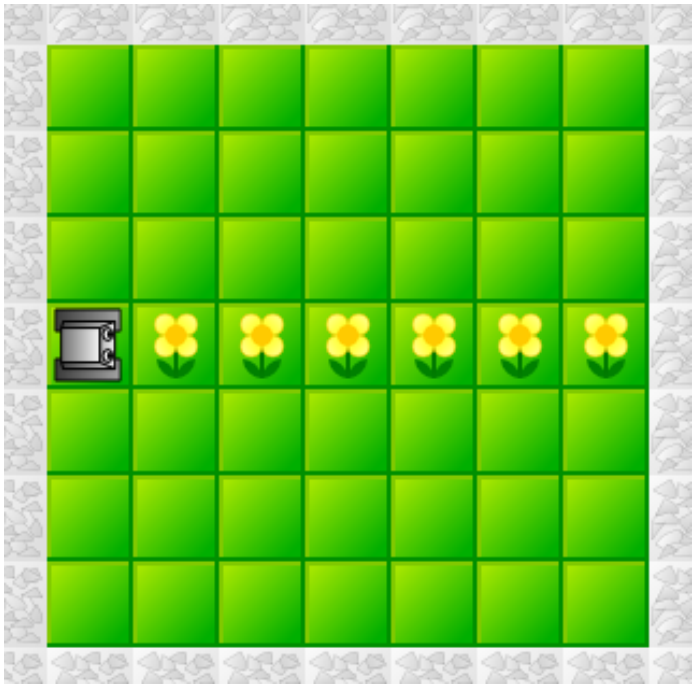
Robotanik - https://www.umimeprogramovat.cz/robotanik

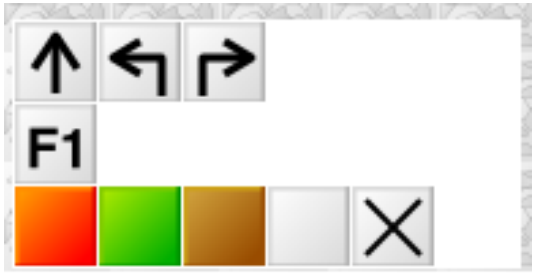
- Nástroj pro cvičení funkcí a rekurze
- Robot s omezenou sadou instrukcí
  - Krok dopředu 
  - Otočení doleva  / doprava 
  - Přebarvení pole 
  - Zavolání některé z funkcí  
    - volání může být i rekurzivní
  - Podmínění instrukce barvou pole 
- Cíl je sebrat květinky na daný počet instrukcí
- Zaregistrujte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



Elementární rekurze

- Jak sebrat květinky na dvě instrukce?



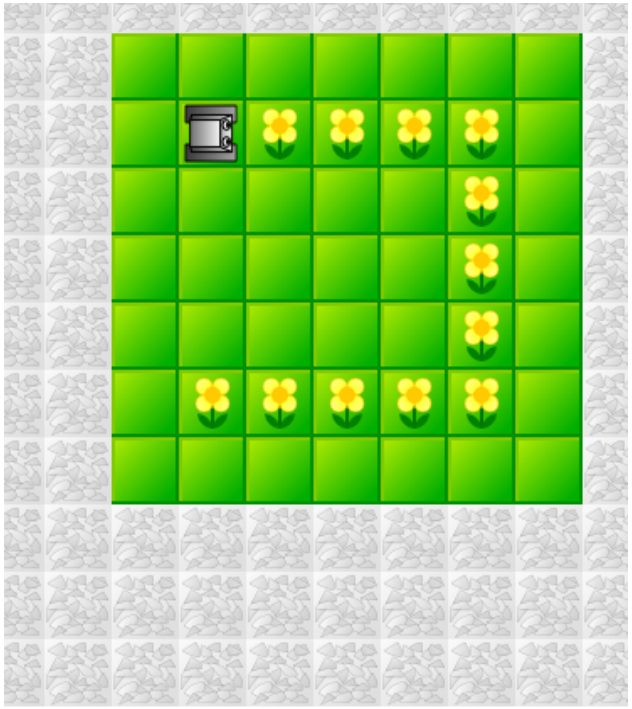


- Krok dopředu a opakovat ☺

Trénink funkcí

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

- Řešení?



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

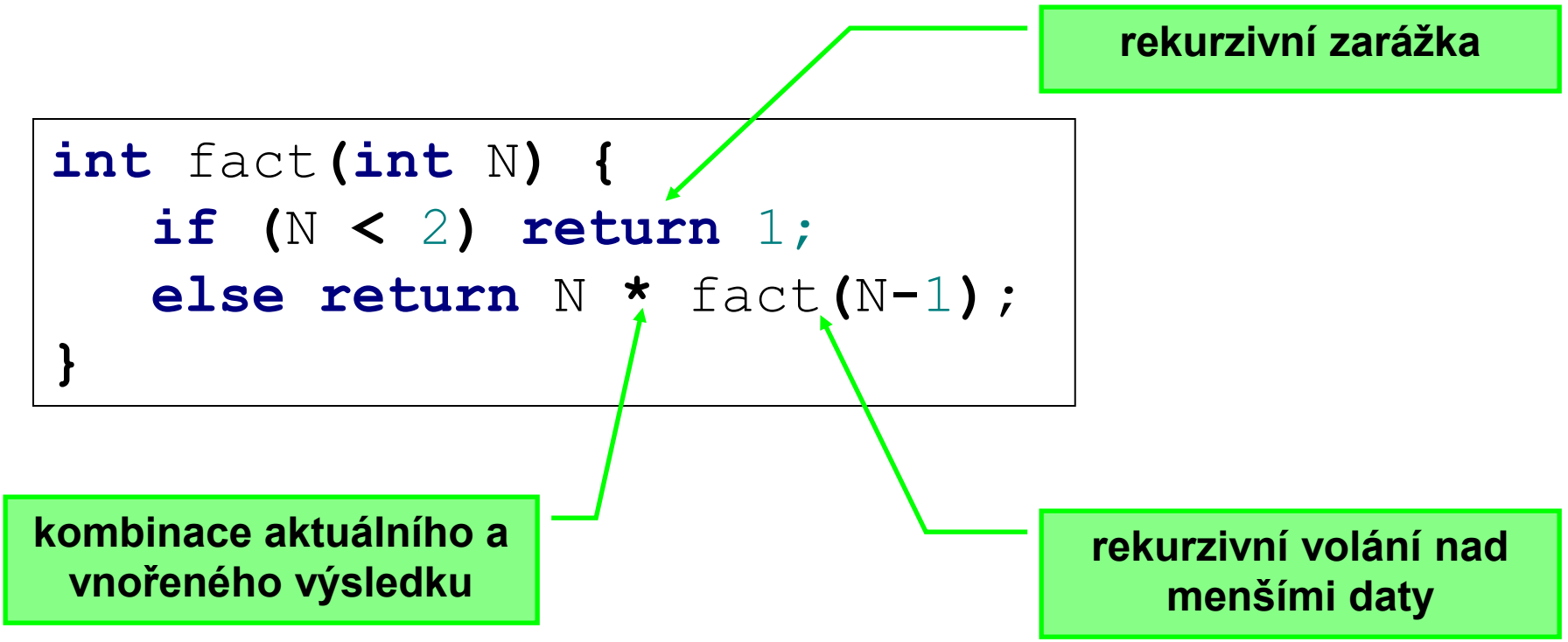
```
#include <stdio.h>
void tisk(int* array, int offset, int length) {
    if (offset < length) {
        printf("%d ", array[offset]);
        tisk(array, offset + 1, length);
    }
}

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;
}
```

rekurzivní zarážka
pokud offset >= length, tak se neprovede další vnoření

výpis aktuálního prvku pole

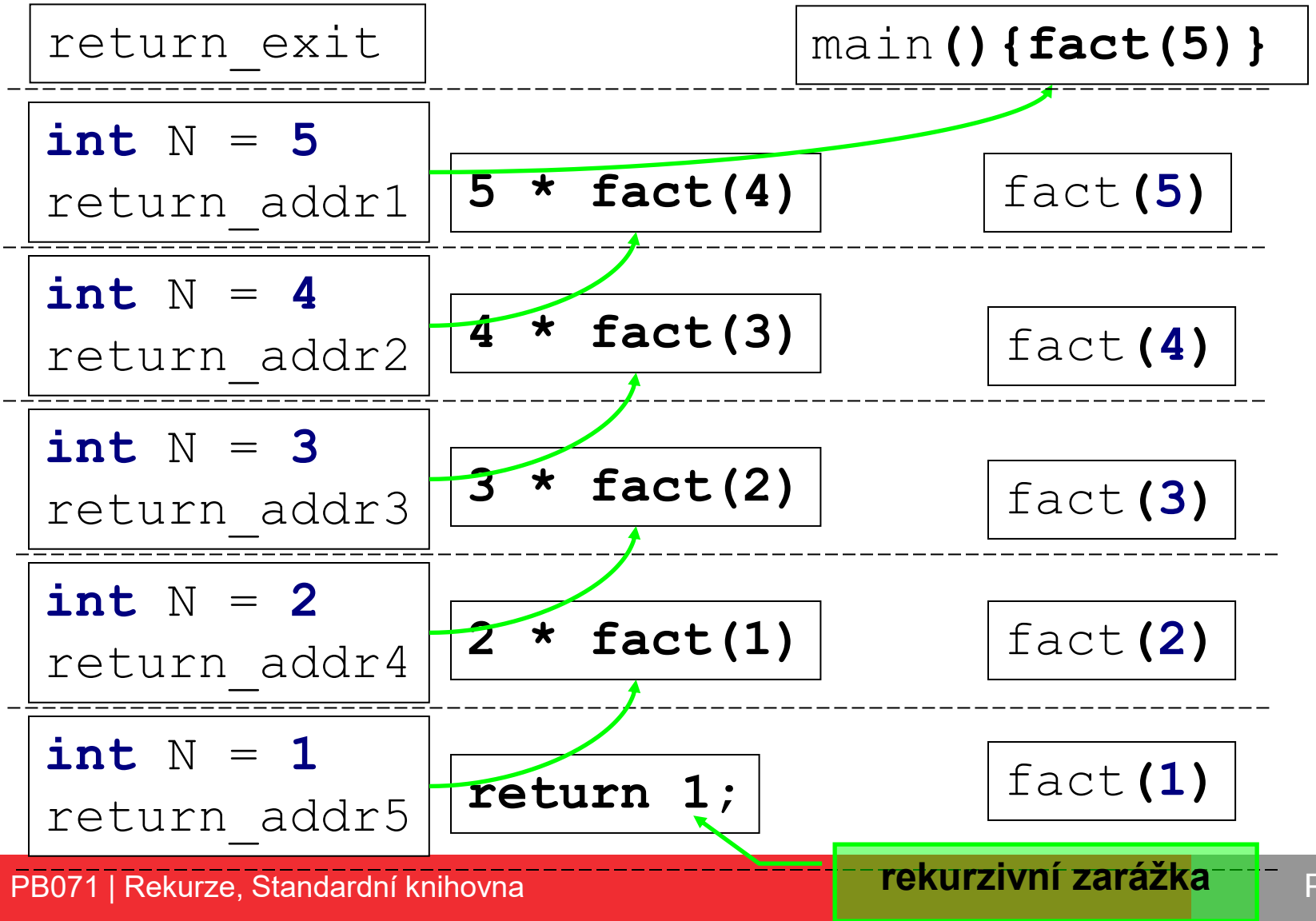
rekurzivní volání nad menšími daty

Faktoriál – části programu



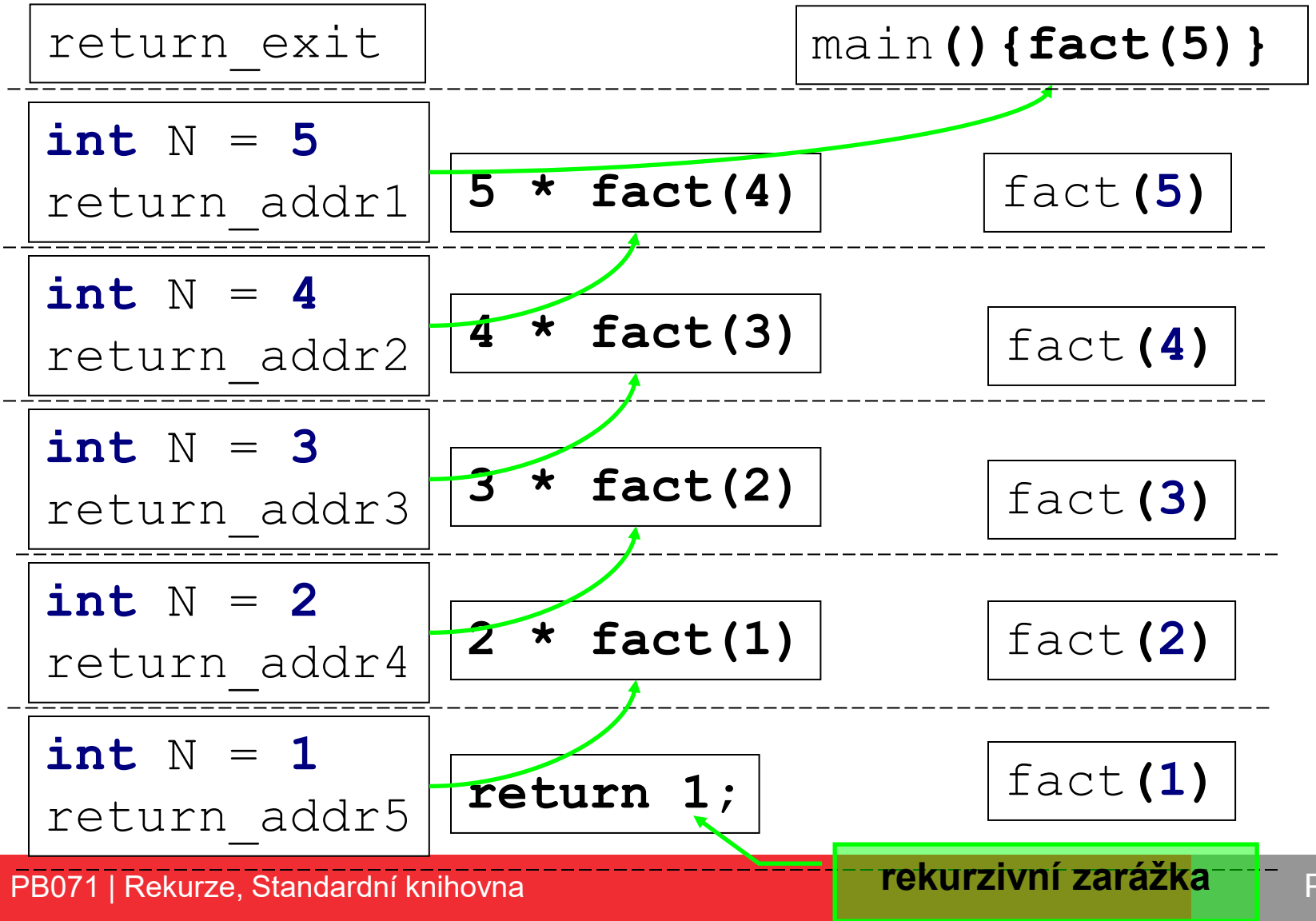
Zásobník (a rekurze)

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

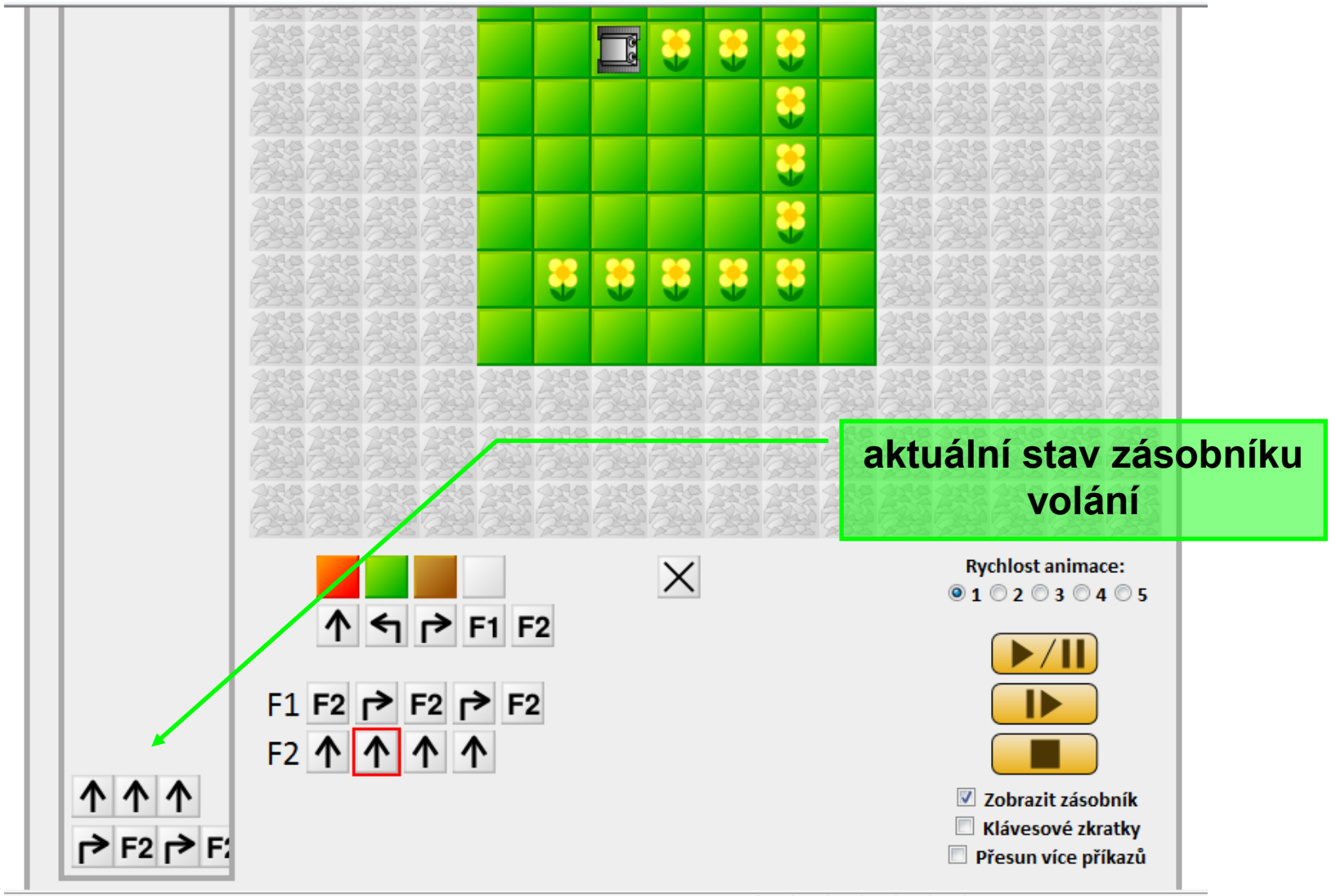


Zásobník (a rekurze)

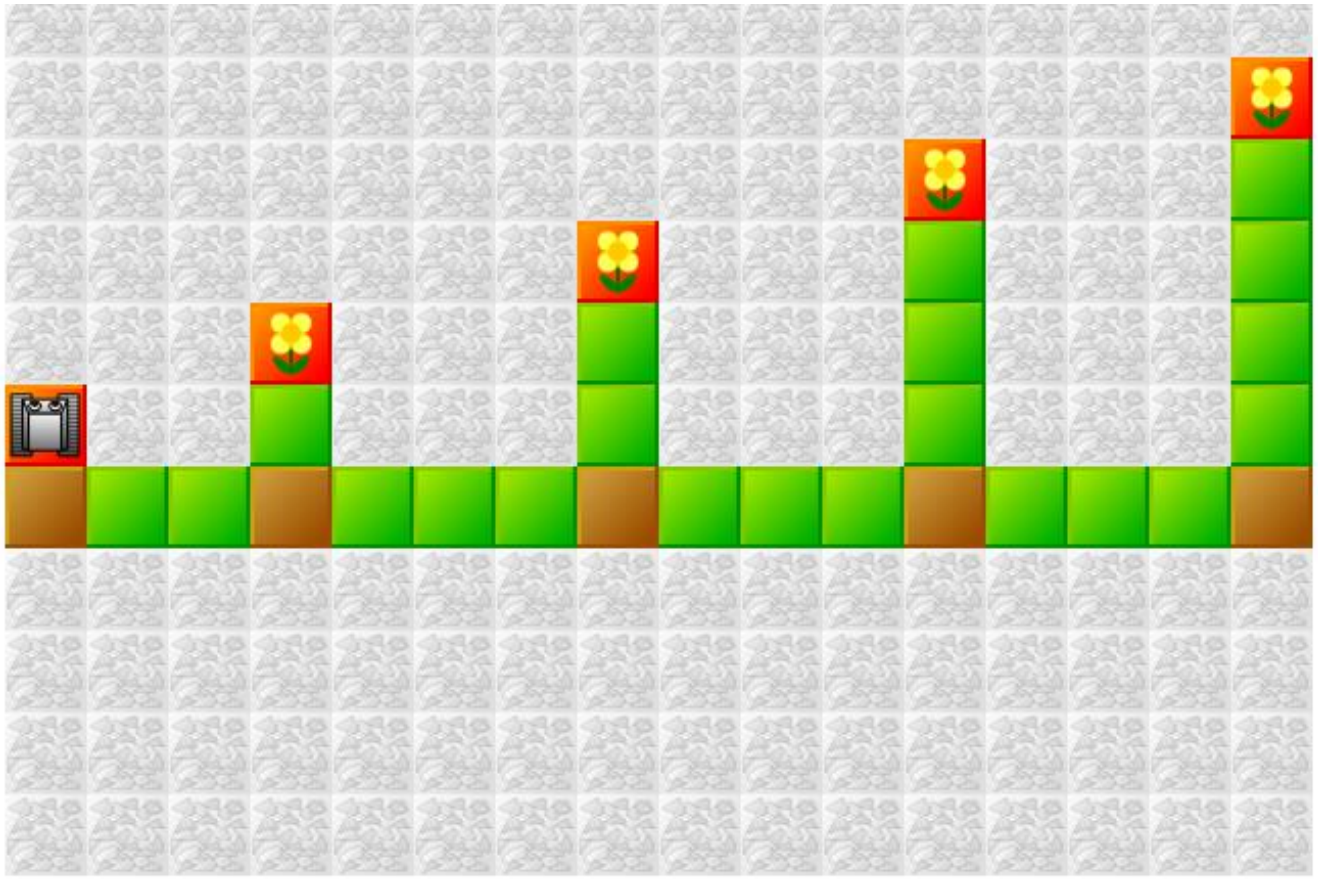
```
int fact(int N) {
    if (N < 2) return 1;
    else return N * fact(N-1);
}
```



Zásobník v Robotanikovi



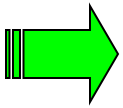
Trochu složitější na pět



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;
}
```

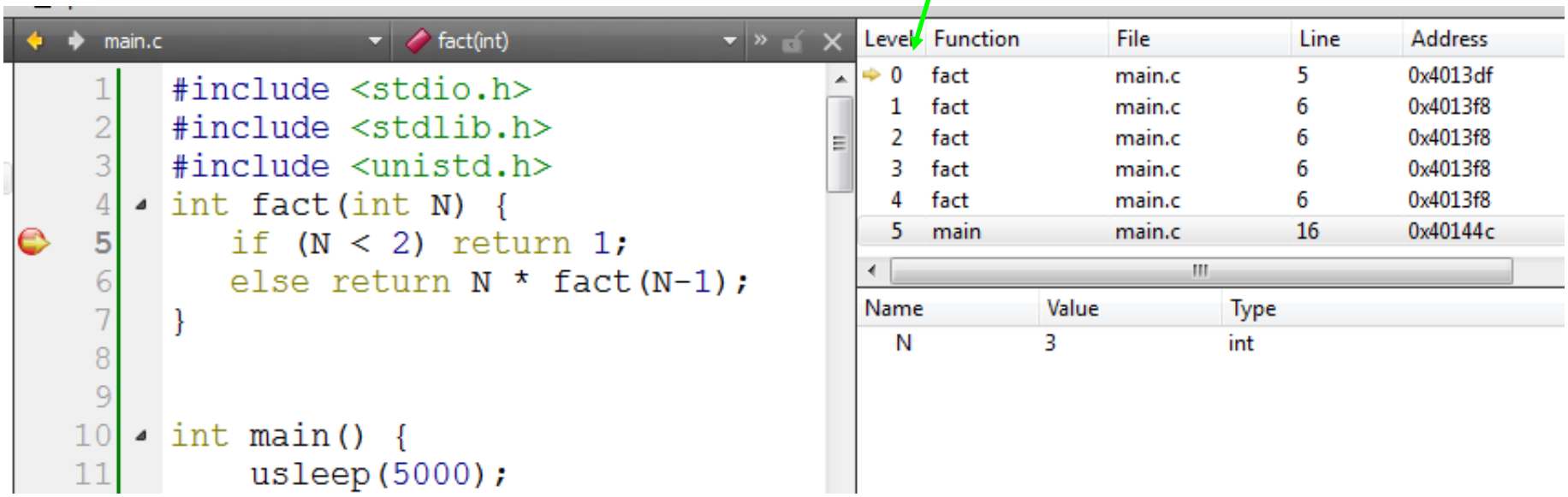


```
int fact(N) {
  int res = 1;
  for (int i=N; i > 1; i--)
     res *= i;
  return res;
}
```

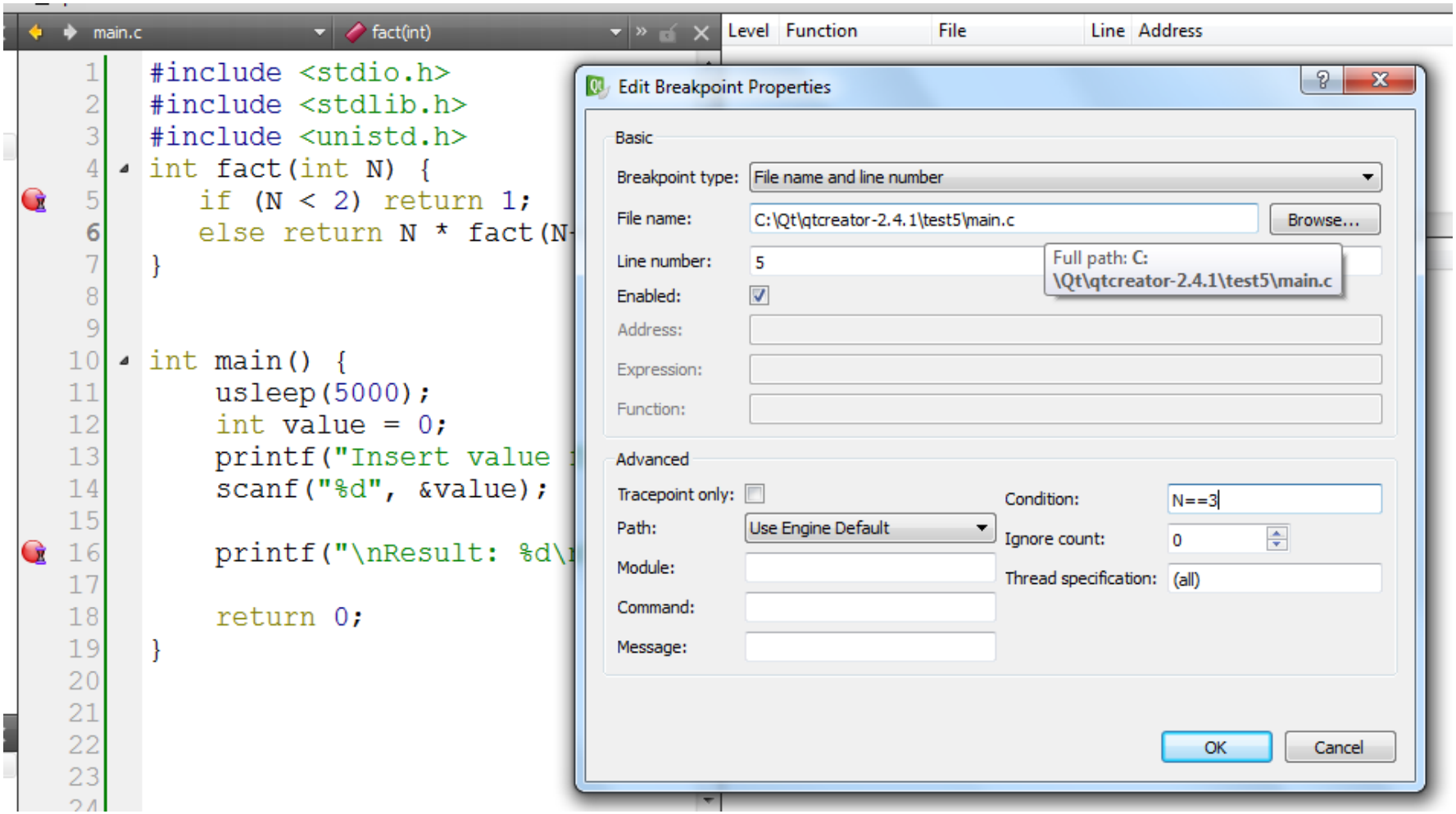
Ladění rekurzivních funkcí

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

zásobník volání, aktuálně jsme v 6. zanoření



Podmíněný breakpoint, N == 3



Využití knihoven

Koncept využívání knihoven

1. Kód z knihovny je přímo zahrnut do kódu aplikace
   - každá aplikace bude obsahovat duplicitně kód knihovních 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
- 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 knihovních funkcí (nespustí se, pokud knihovnu nenalezne)
  - Windows: library.dll, Unix: library.so
- Dynamické nahrávání sdílených knihoven
  - aplikace sama otevírá externí soubor se spustitelným kódem knihovních funkcí (a může reagovat pokud knihovnu nenalezne)
  - Windows (library.dll): **`LoadLibrary()`**,**`GetProcAddress()`**,**`FreeLibrary()`**
  - Unix (library.so): **`dlopen()`**, **`dlsym()`**, **`dlclose()`**

Jakou knihovnu vybrat?

- Preferujte funkce ze standardní knihovny
  - snadno přístupné a přenositelné
- Vyváženost použité vs. nepoužité funkčnosti
  - OpenSSL pouze pro výpočet SHA2-256 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
- Vyberte si licenci: https://choosealicense.com/

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) – 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 řetězci (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 (stejné => 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 spočtenu
- 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

```c
#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
```

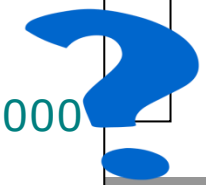
Měření času operace

- 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...)

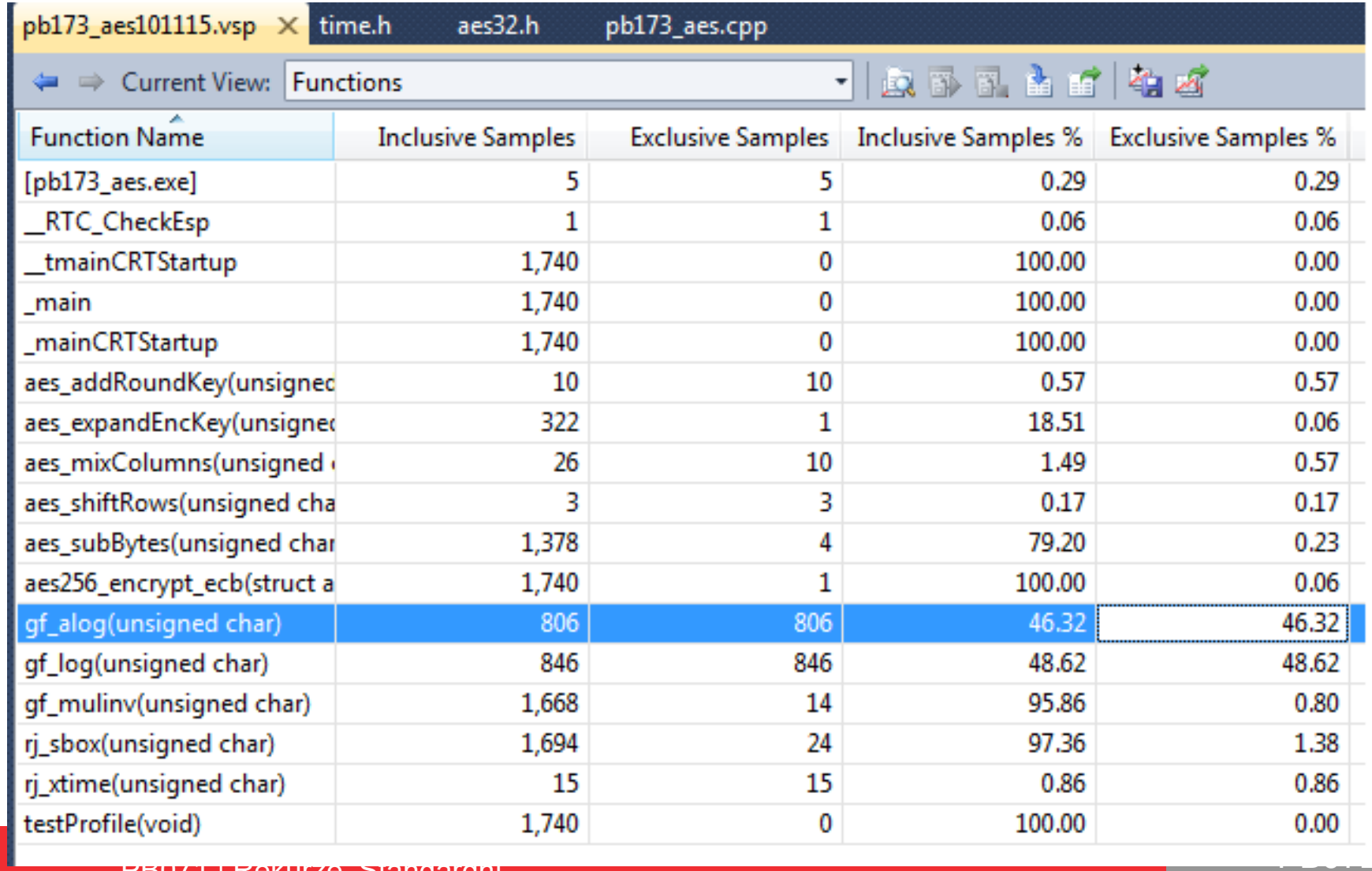
```c
#include <stdio.h>
#include <time.h>

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



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(unsignec	10	10	0.57	0.57
aes_expandEncKey(unsigne	322	1	18.51	0.06
aes_mixColumns(unsigned	26	10	1.49	0.57
aes_shiftRows(unsigned cha	3	3	0.17	0.17
aes_subBytes(unsigned char	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct a	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

		Hit	Total	Coverage
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			
	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 (>=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

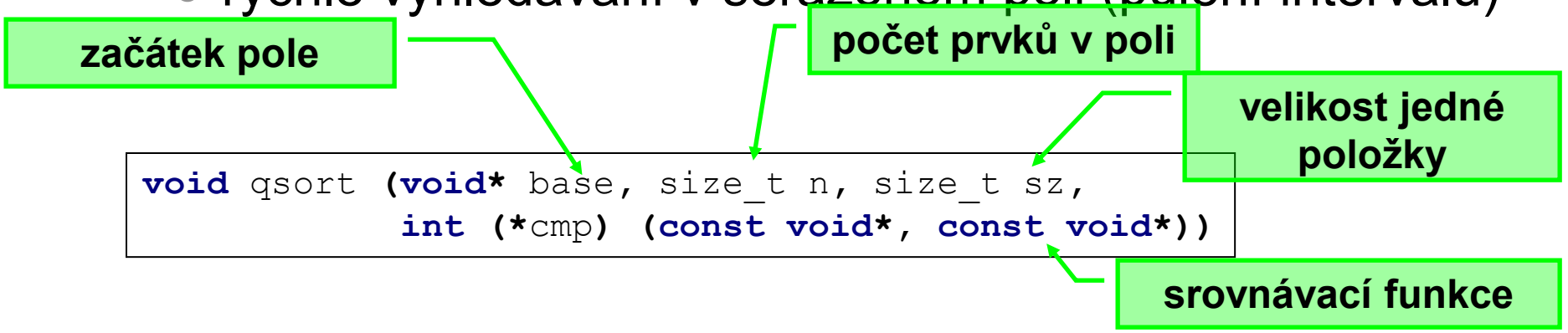
- Trik na zjištění obsahu nadřazeného adresáře

```
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)

Pozn. qsort() nemusí být implementován práve quicksortem



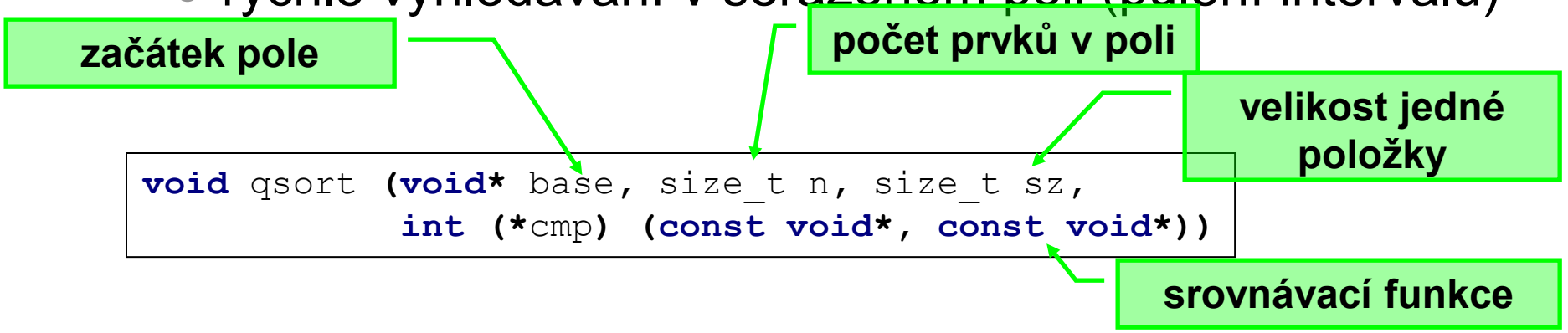

```
void qsort (void* base, size_t n, size_t sz,
            int (*cmp) (const void*, const void*))
```

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

Řadící a vyhledávací funkce

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

Pozn. qsort() nemusí být implementován práve quicksortem




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

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

Řadící funkce `qsort`

Pozn. qsort() nemusí být implementován právě quicksortem, může mít i jinou složitost, záleží na implementaci standardní knihovny (často ale bude použit).

- qsort() je řadící funkce implementující quick sort algoritmus s průměrnou složitostí O(n*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

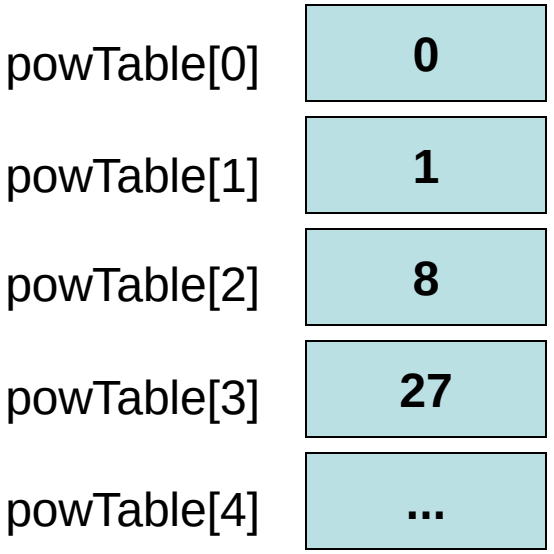
Vyhledávací funkce bsearch

- “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)



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`

ISO/IEC 9899:2011 (C11)

ISO/IEC 9899:2011 (C11)

Pozn. Mělo by být C23
https://en.cppreference.com/w/c/23

- Defacto nejnovější verze standardu (2011)
  - Novější je C17 (2017, ale ta jen odstraňuje defekty C11)
  - 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/talks/data/c1x.pdf
- Vyzkoušení na Aise:
  - module add gcc-10.2
  - gcc -std=c11
- (Zatím nejrozšířenější zůstává použití C99)

Vlákna

- #include <threads.h>
- Velmi podobné vláknům v POSIXu (snadný přechod)
  - pthread_create -> thrd_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_{weak,strong,…}

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?
  - Chybný postup:
    - 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", "w**x**")
  - 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)
```

https://en.cppreference.com/w/c/numeric/tgmath

Typově proměnná makra - ukázka

Type-generic macro	Real function variants			Complex function variants		
	float	double	long double	float	double	long double
fabs	fabsf	fabs	fabsl	cabsf	cabs	cabsl
exp	expf	exp	expl	cexpf	cexp	cexpl
log	logf	log	logl	clogf	clog	clogl
pow	powf	pow	powl	cpowf	cpow	cpowl
sqrt	sqrtf	sqrt	sqrtl	csqrtf	csqrt	csqrtl
sin	sinf	sin	sinl	csinf	csin	csinl

- Např. funkce **`pow()`** se přeloží s využitím typově generického makra na **`cpowf()`** pro argument typu **`float`**

Vylepšená podpora Unicode (UTF-16/32)

- char16_t and char32_t
- <uchar.h>

Makra pro zjištění možností prostředí

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

Shrnutí

- Rekurze
  - Důležitý mentální koncept
  - Často využíván ve funkcionálních jazycích
- Způsob začlenění knihoven je různý
  - ovlivňuje způsob použití
- Standardní knihovna C99
  - velká řada běžných funkcí (včetně řazení a vyhledávání v poli)
  - přenositelnost

DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU