

Přesnost a rychlost výpočtu

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 10

B3B36PRG – Programming in C

Přehled témat

- Část 1 – Přesnost výpočtu
Přesnost výpočtů a numerická stability
- Část 2 – Rychlost výpočtu (programu)
Maticové násobení
Rychlost výpočtu
Paralelní výpočet
- Část 3 Kódovací příklady
Kódovací příklad – NATO Abeceda
NATO Abeceda („jinak“)
Kódovací příklad – struct
Kódovací příklad – Načítání a ukládání složeného typu struct

Part I

Část 1 – Přesnost výpočtu

Přesnost výpočtu - Příklad součtu dvou čísel

```
1 #include <stdio.h>
3 int main(void)
4 {
5     double a = 1e+10;
6     double b = 1e-10;
8     printf("a : %24.121f\n", a);
9     printf("b : %24.121f\n", b);
10    printf("a+b: %24.121f\n", a + b);
12    return 0;
13 }
15 clang sum.c && ./a.out
16 a : 10000000000.000000000000
17 b :          0.000000000100
18 a+b: 10000000000.000000000000
```

lec10/sum.c

Přesnost výpočtu - Příklad dělení dvou čísel

```

1  #include <stdio.h>
3  int main(void)
4  {
5      const int number = 100;
6      double dV = 0.0;
7      float fV = 0.0f;
9      for (int i = 0; i < number; ++i) {
10         dV += 1.0 / 10.0;
11         fV += 1.0 / 10.0;
12     }
14     printf("double value: %lf ", dV);
15     printf(" float value: %lf ", fV);
17     return 0;
18 }
20 clang division.c && ./a.out
21 double value: 10.000000 float value: 10.000002          lec10/division.c

```

Přesnost výpočtu - strojová přesnost

- Strojová přesnost ϵ_m - nejmenší desetinné číslo, které přičtením k 1.0 dává výsledek různý od 1, pro $|v| < \epsilon_m$, platí

$$v + 1.0 == 1.0.$$

Symbol == odpovídá porovnání dvou hodnot (test na ekvivalenci).

- Zaokrouhlovací chyba - nejméně ϵ_m .
- Přesnost výpočtu - aditivní chyba roste s počtem operací v řádu $\sqrt{N} \cdot \epsilon_m$.
 - Často se však kumuluje preferabilně v jedno směru v řádu $N \cdot \epsilon_m$.

Zdroje a typy chyby

- Chyby matematického modelu - matematická aproximace fyzikální situace.
- Chyby vstupních dat.
- Chyby numerické metody.
- Chyby zaokrouhlovací.
- Absolutní chyba aproximace

$$E(x) = \hat{x} - x, \hat{x} \text{ přesná hodnota, } x \text{ aproximace.}$$
- Relativní chyba $RE(x) = \frac{\hat{x} - x}{x}$.

Podmíněnost numerických úloh

- Podmíněnost úlohy $C_p = \frac{\text{relativní chyba výstupních údajů}}{\text{relativní chyba vstupních údajů}}$
- Dobře podmíněná úloha $C_p \approx 1$.
- Výpočet je dobře podmíněný, je-li málo citlivý na poruchy ve vstupních datech.
- Numericky stabilní výpočet - vliv zaokrouhlovacích chyb na výsledek je malý.
- Výpočet je stabilní, je-li dobře podmíněný a numericky stabilní.

Možnosti zvýšení přesnosti

- Reprezentace racionálních čísel - podíl dvou celočíselných hodnot, např. *Homogenní souřadnice*.
- „Libovolná přesnost“ - speciální knihovny, např. gmp až do výše volné paměti.
 - Souřadnice x,y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.

<https://gmplib.org/manual/index>

Součin dvou velkých čísel knihovnou gmp - 1/2

- V HW04B je uveden příklad $(995663 \cdot 995669)^8$ jako prvočíselný rozklad čísla 932865073719992059629773513614789388266580305083920591925740371392254317064584855785088915745761.
 - <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw02>
- Použijme knihovnu gmp pro mocninu a součin dvou čísel, `#include <gmp.h>`.
 - Typ celých čísel `mpz_t`, pomocné funkce `mpz_init_set_str()`, `mpz_init()`, `gmp_printf()` a `mpz_clears()` a operace `mpz_pow_ui()` a `mpz_mul()`.
 - Mocnina `unsigned integer` a násobení - multiplication.
 - Knihovna nemusí být součástí operačního systému, proto může být nutné specifikovat cestu k hlavičkovému souboru a vlastní knihovně (`-lgmp`).
 - Můžeme zadat cestu ručně při kompilaci (nebo do `Makefile`).
 - Alternativně můžeme použít nástroj `pkg-config` (nebo `pkgconf`).
 - <https://www.freedesktop.org/wiki/Software/pkg-config/> <http://pkgconf.org/>
 - Argumenty pro překlad (`CFLAGS`).
 - Argumenty pro linkování (`LDFLAGS`).

```
$ pkgconf --cflags gmp
-I/usr/local/include
```

```
$ pkgconf --libs gmp
-L/usr/local/lib -lgmp
```

Součin dvou velkých čísel knihovnou gmp - 2/2

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <gmp.h>
4
5 const char* resultSrc =
6 "932865073719992059629773513614789388266580305083"
7 "920591925740371392254317064584855785088915745761";
8
9 int main(int argc, char *argv[])
10 {
11     int ret = EXIT_SUCCESS;
12     mpz_t n1, n2, result;
13     mpz_init_set_str(n1, "995663", 10);
14     mpz_init_set_str(n2, "995669", 10);
15     mpz_init(result);
16
17     gmp_printf("n1: %Zd\n", n1);
18     gmp_printf("n2: %Zd\n", n2);
19     gmp_printf("%Zd~%d x %Zd~%d\n", n1, 8, n2, 8);
20     mpz_pow_ui(n1, n1, 8);
21     mpz_pow_ui(n2, n2, 8);
22
23     gmp_printf("%Zd x %Zd\n", n1, n2);
24     mpz_mul(result, n1, n2);
25     gmp_printf("%Zd\n", result);
26     printf("Result from HW04\n%s\n", resultSrc);
27     mpz_clears(n1, n2, result, NULL);
28     return ret;
29 }
```

```
$ ./demo-gmp-mpz
n1: 995663
n2: 995669
995663^8 x 995669^8
965826124294607867982699926255695296863400309121 x
965872686868261151537037082260231566481047775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
Result from HW04
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
lec10/gmp/demo-gmp-mpz.c
```

Racionální čísla knihovny gmp - 1/3

- „Libovolné přesnosti“ reprezentace, např. souřadnic v rovině jako výsledek operací výpočetní geometrie, můžeme realizovat podílem dvou („libovolně velkých“) celých čísel.
 - Souřadnice x,y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.
- Knihovna gmp k tomuto účelu poskytuje typ `mpq_t`, kromě typu necelého čísla `mpf_t`, který využijeme pro převod `mpq_t` na necelé číslo typu `double`.

```
49 double mpq2d(const mpq_t *op)
50 {
51     double ret;
52     mpf_t v;
53     mpf_init(v);
54     mpf_set_q(v, *op);
55     ret = mpf_get_d(v);
56     mpf_clear(v);
57     return ret;
58 }
```

<lec10/gmp/demo-gmp-mpq.c>

Racionální čísla knihovny gmp - 2/3

```

1 #include <stdio.h>
2 #include <stdlib.h>
4 #include <gmp.h>
6 double mpq2d(const mpq_t *op);
8 int main(int argc, char *argv[])
9 {
10     int ret = EXIT_SUCCESS;
12     unsigned long xl = 7511641767681;
13     unsigned long yl = 3468686699521;
14     unsigned long denl = 37395671041;
15     const unsigned int digits = 21;
17     double xd = 1. * xl;
18     double yd = 1. * yl;
19     double dend = 1. * denl;
21     printf("unsigned long: %lu %lu %lu\n", xl, yl, denl);
22     printf("double: %.01f %.01f %.01f\n", xd, yd, dend);
24     printf("double x,y (.2): %.2lf %.2lf\n", xd/dend, yd/dend);
25     printf("double x,y (.46): %.46lf %.46lf\n", xd/dend, yd/dend);
27     mpq_t x, y;
28     mpq_inits(x, y, NULL);
29     mpq_set_ui(x, xl, denl);
30     mpq_set_ui(y, yl, denl);
32     mpq_canonicalize(x);
33     mpq_canonicalize(y);
35     mpf_t xmpf, ympf;
36     mpf_inits(xmpf, ympf, NULL);
37     mpf_set_q(xmpf, x);
38     mpf_set_q(ympf, y);
40     gmp_printf("mpq x,y (canonical form): %Qd %Qd\n",
41                x, y);
42     gmp_printf("mpf x,y (to %d decimal digits): %.46f\n",
43                digits, digits, xmpf, ympf);
44     gmp_printf("mpq x,y (double .46): %.46lf %.46lf\n",
45                mpq2d(x), mpq2d(y));
46     mpf_clears(x, y, NULL);
47     mpf_clears(xmpf, ympf, NULL);
48     return ret;
49 }
50
51 lecl0/gmp/demo-gmp-mpq.c

```

Racionální čísla knihovny gmp - 3/3

- Souřadnice x,y - 751164176768 346868669952 3739567104 ~ 2008,57; 92,76.

```

$ make
clang -c -I/usr/local/include -g demo-gmp-mpq.c -o demo-gmp-mpq.o
clang demo-gmp-mpq.o -L/usr/local/lib -lgmp -o demo-gmp-mpq
clang -c -I/usr/local/include -g demo-gmp-mpz.c -o demo-gmp-mpz.o
clang demo-gmp-mpz.o -L/usr/local/lib -lgmp -o demo-gmp-mpz
$ ./demo-gmp-mpq
unsigned long: 751164176768 346868669952 3739567104
double: 751164176768 346868669952 3739567104
double x,y (.2): 2008.57 92.76
double x,y (.46): 2008.5651541681761500512948259711265563964843750000
92.7563700036227487544238101691007614135742187500
mpq x,y (canonical form): 399190273/198744 1536231/16562
mpf x,y (to 21 decimal digits): 2008.565154168176146200000 92.756370003622750875500
mpq x,y (double .46): 2008.5651541681759226776193827390670776367187500000
92.7563700036227487544238101691007614135742187500
lecl0/gmp/demo-gmp-mpq.c

```

Makefile s pkg-config a gmp

```

1 CFLAGS+=$(shell pkg-config --cflags gmp)
2 LDFLAGS+=$(shell pkg-config --libs gmp)
4 CFLAGS+=-g
6 DEMO_MPQ=demo-gmp-mpq
7 DEMO_MPZ=demo-gmp-mpz
9 TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ)
11 bin: $(TARGETS)
13 info:
14 @echo $(CFLAGS)
15 @echo $(LDFLAGS)
17 $(DEMO_MPQ): $(DEMO_MPQ).o
18 $(CC) $< $(LDFLAGS) -o $@
20 $(DEMO_MPZ): $(DEMO_MPZ).o
21 $(CC) $< $(LDFLAGS) -o $@
23 %.o : %.c
24 $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
26 clean:
27 $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o
lecl0/gmp/Makefile

```

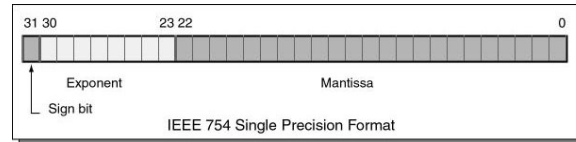
```

$ gmake
clang -c -I/usr/local/include -g demo-gmp-mpq.c -o demo-gmp-mpq.o
clang demo-gmp-mpq.o -L/usr/local/lib -lgmp -o demo-gmp-mpq
clang -c -I/usr/local/include -g demo-gmp-mpz.c -o demo-gmp-mpz.o
clang demo-gmp-mpz.o -L/usr/local/lib -lgmp -o demo-gmp-mpz

```

Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru $x = (-1)^s \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$. Základ 2.
IEEE 754, ISO/IEC/IEEE 60559:2011
- Mantisa je **normalizována** na první jedničku vlevo (v soustavě o dvojkovém základu).
- **float** – 32 bitů (4 bajty): s – 1 bit znaménko (+ nebo –), **exponent** – 8 bitů, tj. 256 možností. **mantisa** – 23 bitů $\approx 16,7$ milionu možností.



- **double** – 64 bitů (8 bajtů).
 - s – 1 bit znaménko (+ nebo –).
 - **exponent** – 11 bitů, tj. 2048 možností.
 - **mantisa** – 52 bitů $\approx 4,5$ biliony možností (4 503 599 627 370 495).
- **bias** umožňuje reprezentovat exponent vždy jako kladné číslo.

Lze zvolit, např. $\text{bias} = 2^{eb-1} - 1$, kde eb je počet bitů exponentu.
<http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky>
 B3B36PRG – Lecture 10: Přesnost a rychlost výpočtu

Příklad reprezentace float hodnot dle IEEE 754

The screenshot shows two instances of the IEEE 754 Converter. The top instance shows the conversion of -256.75. The sign is -1, the exponent is 135 (2^8), and the mantissa is 1.0029296875. The binary representation is 11000011100000000110000000000000 and the hexadecimal is 0xc3806000. The bottom instance shows the conversion of -256.74. The sign is -1, the exponent is 135 (2^8), and the mantissa is 1.0028905868530273. The binary representation is 110000111000000001011101011000 and the hexadecimal is 0xc3805eb8.

- Chyba reprezentace -256.75 vs -256.74.
- Infinity (0xf800000), -Infinity (0xff800000), a NaN (0x7fffffff).

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>
 B3B36PRG – Lecture 10: Přesnost a rychlost výpočtu 18 / 58

Příklady reprezentace hodnot typu float

Reprezentace čísla 85,125 (float)

- 85 odpovídá 1010101₍₂₎.
- 0,125 odpovídá 001
 - 0,125/2⁻¹ = 0,25 | 0
 - 0,125/2⁻² = 0,50 | 0
 - 0,125/2⁻³ = 1,00 | 1
- 85,125 odpovídá 1010101,001₍₂₎ = 1,010101001₍₂₎ × 2⁶,
- Bias pro float je 127.
- Exponet je 127 + 6 = 133
- 133 odpovídá 10000101₍₂₎.
- Normalizovaná mantisa je 010101001₍₂₎, kterou doplníme nulami na 23 bitů (zprava, je to desetinné číslo).
- 0 - 1000 0101 - 0101 0100 1000 0000 0000 000.
- 01000010 10101010 01000000 00000000.
- V šestnáctkové soustavě 0x42 0xaa 0x40 0x00, tedy 0x42aa4000.

Reprezentace čísla 0,1 (float)

- 0,1 má periodický rozvoj

1.	0,1 * 2 = 0,2		0
2.	0,2 * 2 = 0,4		0
3.	0,4 * 2 = 0,8		0
4.	0,8 * 2 = 1,6		1
5.	0,6 * 2 = 1,2		1
6.	0,2 * 2 = 0,4		0
- Opakuje se 0011, 23-bitů tak reprezentuje menší hodnotu.
- 0,1₍₁₀₎ ~ 0,0001 1001 1001 1001 1001 1001 100₍₂₎ = 1,1001 1001 1001 1001 1001 100₍₂₎ × 2⁻⁴.
- Exponet je 127 - 4 = 123 odpovídá 0111 1011₍₂₎.
- Normalizovaná mantisa 1,100 1100 1100 1100 1100 1100.
- 0 - 0111 1011 - 100 1100 1100 1100 1100 1100.
- 0011101 11001100 11001100 11001100.
- V šestnáctkové soustavě to je 0x3d 0xcc 0xcc 0xcc, tedy 0x3dcccc.
- Prakticky je 0,1 převedeno na o něco větší číslo 0x3dccccd, protože absolutní chyba je menší.

<lec10/floats.c>
 B3B36PRG – Lecture 10: Přesnost a rychlost výpočtu 19 / 58

Sčítání mnoha malých necelých čísel - 1/2

- Na příkladu součtu dvou velmi odlišných čísel (např. 1 × 10¹⁰ + 1 × 10⁻¹⁰) dochází z důvodu omezené reprezentace mantisy k zaokrouhlovací chybě.
- V případě naivní implementace součtu velkého počtu (např. 2³⁰) velmi malých hodnot (např. 1 × 10⁻²⁰) může dojít vlivem zaokrouhlování k významné chybě.

```

// small value to be sum
float v = 1e-20f; //float literal
// 1073741824 is 2^30 values (1e9)
const size_t power = 30;
size_t n = 1l<<power;
// multiplication factor for print
const double k = 1e11;
float *values = init_values(n, v);

double sum1 = v*n * k;
Prímé násobení - výsledek
1.073 741 789 925 364 287 228 1.

float sum_naive(size_t n, float *v)
{
  float r = 0;
  for (size_t i = 0; i < n; ++i) {
    r += v[i];
  }
  return r;
}
double sum2 = sum_naive(n, values) * k;
Naivní součet - výsledek
0.022 737 367 544 323 205 947 9.

float sum_alter(size_t n, float *v, size_t power)
{
  float r = 0;
  const size_t order = power - 1;
  size_t k = 2;
  for (size_t l = 1; l < order; ++l, k *= 2) {
    for (size_t i = 0; i < n; i += k) {
      v[i] = v[i] + v[i+k/2];
    }
  }
  k /= 2;
  for (size_t i = 0; i < n; i += k) {
    r += v[i];
  }
  return r;
}
float sum3 = sum_alter(n, values, power) * k;
Sčítání po dvojicích - výsledek
1.073 741 793 632 507 324 218 8.

```

<lec10/addition.c>
 B3B36PRG – Lecture 10: Přesnost a rychlost výpočtu 20 / 58

Sčítání mnoha malých necelých čísel - 2/2

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 float* init_values(size_t n, float v);
4 float sum_naive(size_t n, float *v);
5 float sum_alter(size_t n, float *v, size_t power);
6 int main(void)
7 {
8   float v = 1e-20f; // small value to be sum
9   const size_t power = 30; // try 3 vs. 30
10  size_t n = 1l<<power; // 1073741824 is 2^30 values
11  const double k = 1e11;
12  float *values = init_values(n, v);
13  double sum1 = v * n * k;
14  double sum2 = sum_naive(n, values) * k;
15  float sum3 = sum_alter(n, values, power) * k;
16  printf("Sum of %lu numbers of the value %.22lf\n", n, v);
17  printf("Sum1 (multiplication): %.22lf\n", sum1);
18  printf("Sum2 (naive) : %.22lf\n", sum2);
19  printf("Sum3 (alter) : %.22lf\n", sum3);
20  free(values);
21  return EXIT_SUCCESS;
22 }
23
24 float* init_values(size_t n, float v)
25 {
26   float *r = malloc(n * sizeof(float));
27   if (!r) {
28     fprintf(stderr, "ERROR: MEM_ALLOC\n");
29     exit(-1);
30   }
31   for (size_t i = 0; i < n; ++i) {
32     r[i] = v;
33   }
34   return r;
35 }
36
37 $ clang addition.c -o addition && ./addition
38 Sum of 1073741824 numbers of the value 1e-20
39 Sum1 (multiplication): 1.0737417899253642872281
40 Sum2 (naive) : 0.0227373675443232059479
41 Sum3 (alter) : 1.0737417936325073242188
42
43 $ calc "1e-20 * 2^30 * 1e11"
44 1.073741824

```

<lec10/addition.c>
 B3B36PRG – Lecture 10: Přesnost a rychlost výpočtu 21 / 58

Part II

Část 2 – Rychlost výpočtu (programu)

Maticové násobení - Naivně s transpozicí

```

1 void simple_multiply_trans(const int n, const double *a, const double *b, double *c)
2 {
3     double * bT = create_matrix(n); // allocate memory for transposed matrix
4     for (int i = 0; i < n; ++i) {
5         bT[i*n + i] = b[i*n + i];
6         for (int j = i + 1; j < n; ++j) {
7             bT[i*n + j] = b[j*n + i];
8             bT[j*n + i] = b[i*n + j];
9         }
10    }
11    for (int i = 0; i < n; ++i) {
12        for (int j = 0; j < n; ++j) {
13            double tmp = 0;
14            for (int k = 0; k < n; ++k) {
15                tmp += a[i*n + k] * bT[j*n + k];
16            }
17            c[i*n + j] = tmp;
18        }
19    }

```

Maticové násobení - Naivně

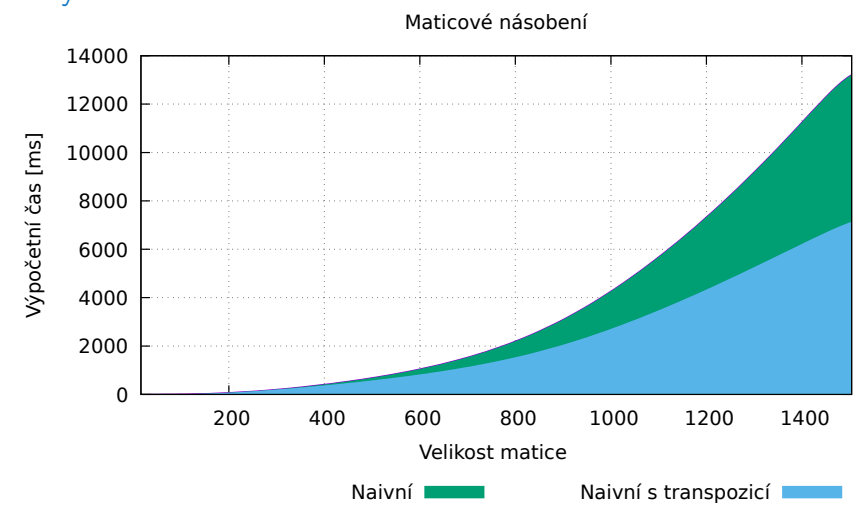
```

1 void simple_multiply(const int n, const double *a, const double *b, double *c)
2 {
3     for (int i = 0; i < n; ++i) {
4         for (int j = 0; j < n; ++j) {
5             double prod = 0;
6             for (int k = 0; k < n; ++k) {
7                 prod += a[i * n + k] * b[k * n + j];
8             }
9             c[i * n + j] = prod;
10        }
11    }
12 }

```

- Pro přehlednost předpokládáme kompatibilní rozměry matic a správně alokované.

Porovnání rychlosti násobení matic



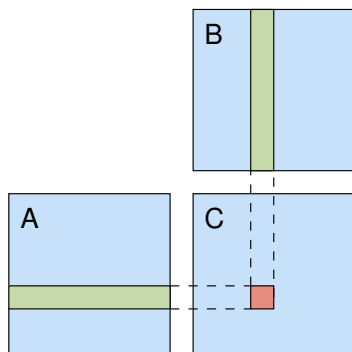
Architektura procesoru a způsob výpočtu

- Příklad násobení matic a násobení transponované matice ukazuje, že kromě instrukcí má zásadní vliv **organizace dat a přístup do paměti**.
- V moderních procesorech hraje **cache** zásadní roli společně s řetězením instrukcí, tzv. **pipelining**, a využitím specifických instrukcí.

SIMD - Single Instruction Multiple Data

- Proniknutí do detailů fungování cache a řetězení instrukcí je náplní předmětu **Architektura počítačů (BOB35AP0)**, kde máte možnost se seznámit s přicházející architekturou RISC V.

<https://cw.felk.cvut.cz/wiki/courses/b35apo/>



Optimalizace kódu

- Kromě optimalizace výsledného spustitelného kódu při překladu, je možné optimalizovat kódu za běhu nebo již existujících binární (přeložené) soubory.
 - **BOLT** - Binary Optimization and Layout Tool, zrychlení o až 20%–50%.
 - <https://arxiv.org/abs/1807.06735>
 - <https://dl.acm.org/doi/10.5555/3314872.3314876>
- Využití speciálních instrukcí v základních funkcích může výpočty (programy) výrazně urychlit, zejména pokud se funkce používají masivně.
 - AVX2 a EVEX instrukce (ze sady SSE4.2) ve funkcích porovnání řetězců `str{n}casecmp()` – až o 38% méně potřebného času.
 - 03/2022 - <https://www.phoronix.com/news/Glibc-strcasecmp-AVX2-EVEX>
- *V obou případech (a obecně) je vhodné rozumět principu a využít instrukce Assembleru.*

Informativní

Compiler Explorer

<https://godbolt.org/z/K9r1eWqcd>

Informativní

Compiler Explorer – Analýza optimalizovaného kódu

- Vliv optimalizace `-O2` na výsledný kód, který obsahuje nedefinované chování, přetečení celého čísla.

Příloha 3. přednášky.

<https://godbolt.org/z/G3GEz4vbw>

Příklad použití OpenMP - Maticové násobení 1/2

- Open Multi-Processing (OpenMP) - aplikační programové rozhraní (API) multiplatformních výpočtů se sdílenou pamětí. <http://www.openmp.org>
- Direktivou preprocesoru můžeme instruovat kompilátor k vytvoření kódu paralelního výpočtu, např. paralelizace přes vnější proměnnou `i`.

```

1 void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
2 {
3     int i;
4     #pragma omp parallel private(i)
5     #pragma omp for schedule (dynamic, 1)
6     for (i = 0; i < n; ++i) {
7         for (int j = 0; j < n; ++j) {
8             c[i][j] = 0;
9             for (int k = 0; k < n; ++k) {
10                c[i][j] += a[i][k] * b[k][j];
11            }
12        }
13    }
14 }
```

`lec10/demo-omp-matrix.c`

Pro přehlednost uvažujeme čtvercové matice stejných rozměrů.

Part III

Část 3 – Kódovací příklady

Příklad použití OpenMP - Maticové násobení 2/2

- Příklad násobení matic 1000×1000 s využitím OpenMP na iCore5 (2 jádra s HT ~ 4x výpočetní jednotky).
- Násobení matic 5000×5000 (Ryzen 9 5950X).

```

1 gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
2 ./demo-omp 1000
3 Size of matrices 1000 x 1000 naive
4     multiplication with O(n^3)
5 c1 == c2: 1
6 Multiplication single core 9.33 sec
7 Multiplication multi-core 4.73 sec
9 OMP_NUM_THREADS=2 ./demo-omp 1000
10 Size of matrices 1000 x 1000 naive
11     multiplication with O(n^3)
12 c1 == c2: 1
13 Multiplication single core 9.48 sec
14 Multiplication multi-core 6.23 sec
```

- i7-6700K:

- 1x vlákno 0.80s;
- 2x vlákna 0.39s;
- 4x vlákna 0.24s.

```

1 OMP_NUM_THREADS=16 ./demo-omp 5000
2 Size of matrices 5000 x 5000 naive
3     multiplication with O(n^3)
4 Multiplication single core 256.00 sec
5 Multiplication multi-core 18.05 sec
```

```

1 OMP_NUM_THREADS=16 ./demo-omp 5000
2 Size of matrices 5000 x 5000 naive
3     multiplication with O(n^3)
4 Multiplication single core 256.00 sec
5 Multiplication multi-core 18.05 sec
```

`lec10/demo-omp-matrix.c`

Kódovací příklad – NATO Abeceda – 1/4

- Implementujme program, který převede vstupní text (ASCII, znaky A–Z a a–z) do NATO abecedy, ve které jsou písmena hláskována prostřednictvím následujících jmen.
- Očekávaný výstup pro vstup `in.txt`.

```

$ cat in.txt
I like PRG and programming in C.
$ clang nato-alphabet.c && ./a.out < in.txt 2>/dev/null
India Lima India Kilo Echo Papa Romeo Golf Alpha November Delta Papa
Romeo Oscar Golf Romeo Alpha Mike Mike India November Golf
India November Charlie
```

- Alpha, Bravo, Charlie, Delta, Echo, Foxtrot, Golf, Hotel, India, Juliett, Kilo, Lima, Mike, November, Oscar, Papa, Quebec, Romeo, Sierra, Tango, Uniform, Victor, Whiskey, X-ray, Yankee, Zulu.

- V programu definujeme pole ukazatelů na textové literály s jednotlivými slovy.
- Programově otestujeme, že slova odpovídají počátečním písmenům A–Z.

- Implementujme testovací funkce.

```

1 static char *words[] = { // static to be "private"
2     "Alpha", "Bravo", "Charlie", "Delta", "Echo", "Foxtrot", "
3     "Golf", "Hotel", "India", "Juliett", "Kilo", "Lima", "Mike
4     ", "November", "Oscar", "Papa", "Quebec", "Romeo", "
5     Sierra", "Tango", "Uniform", "Victor", "Whiskey", "X-ray"
6     , "Yankee", "Zulu", NULL
7 }; // it is an array of pointers to text literals
8
9 int count_words_array(char *words[]); // Using array
10 int count_words(char **words); // Pointer to pointers
11 bool check_alphabet_words(char *words[]);
```


Kódovací příklad – NATO Abeceda – 2/4

```
1 // array is terminated by NULL used for counting
2 static char *words[] = { "Alpha", ..., "Zulu", NULL };
3 // array-like variant
4 int count_words_array(char *words[])
5 {
6     int n = 0;
7     while(words[n] != NULL) {
8         fprintf(stderr, "DEBUG: \"%s\\n\"", words[n]);
9         n += 1;
10    }
11    return n;
12 }
13 // pure pointer variant
14 int count_words(char **words)
15 {
16     int n = 0;
17     char **cur = words;
18     while (*cur) {
19         fprintf(stderr, "DEBUG: \"%s\\n\"", *cur);
20         cur++;
21         n += 1;
22     }
23     return n;
24 }
25 }
26 }
```

```
26 bool check_alphabet_words(char *words[])
27 {
28     bool ret = true; // true is from #include <stdbool.h>
29     char c = 'A'; // char is an integer ASCII code number
30     char **cur = &words[0]; // there is always at least one item
31     while (*cur) {
32         fprintf(stderr, "DEBUG: check %s[0] for '%c\\n", *cur, c);
33         if (c != *cur[0]) { // the first letter needs to match
34             ret = false; // false is from #include <stdbool.h>
35             break;
36         } else {
37             c += 1;
38             cur += 1;
39         }
40     }
41     return ret;
42 }
```

- Pole `words` je posloupnost prvků stejného typu (ukazatel na `char` – textový řetězec).
- Hodnota `&words[0]` je identická adresa jako hodnota `words`.

Kódovací příklad – NATO Abeceda – 4/4

```
1 ...
2 static const char * const words[] = { "Alpha", ..., NULL };
3 ...
4 char my_toupper(char c);
5 int main(void)
6 {
7     ...
8     int c;
9     while ((c = getchar()) != EOF) {
10        c = my_toupper(c); // or toupper() from <ctype.h>
11        if (c >= 'A' && c <= 'Z') {
12            printf("%s ", words[c - 'A']); // always print space
13        }
14    }
15    ...
16 }
17 char my_toupper(char c) // or use toupper() from <ctype.h>
18 {
19     if (c >= 'a' && c <= 'z') {
20         c = c - 'a' + 'A';
21     }
22     return c;
23 }
24 }
25 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <stdbool.h>
5
6 static const char * const words[] = { "Alpha", ..., NULL };
7 ...
8 int count_words_array(const char * const words[]);
9 bool check_alphabet_words(const char * const words[]);
10
11 int main(void)
12 { // assert macro debug and development only, see -DNDEBUG
13     assert(count_words_array(words) == 'Z' - 'A' + 1);
14     assert(check_alphabet_words(words));
15     int c;
16     while ((c = getchar()) != EOF) {
17         c = (c >= 'a' && c <= 'z') ? c = c - 'a' + 'A' : c;
18         if (c >= 'A' && c <= 'Z') {
19             printf("%s\\n", words[c - 'A']);
20         }
21     }
22     return EXIT_SUCCESS;
23 }
```

- Funkci `my_toupper()` můžeme nahradit použitím ternárního operátoru.
- V rámci zpřehlednění můžeme překlad (řádky 15–21) dát do samostatné funkce `void translate(const char * const words[])`.

Kódovací příklad – NATO Abeceda – 3/4

■ Můžeme použít `const`.

```
1 static const char * const words[] = { "Alpha", ..., NULL };
2 int count_words_array(const char * const words[])
3 {
4     int n = 0;
5     while(words[n] != NULL) {
6         n += 1;
7     }
8     return n;
9 }
10
11 int count_words(const char * const * const words)
12 {
13     int n = 0;
14     // cur je ukazatel na data typu konstantní
15     // ukazatel na konstantní textový řetězec
16     // (na konstantní ukazatel na konstantní hodnoty char).
17     const char * const * cur = words; // cur chceme měnit
18     while (*cur) {
19         cur++; // cur není konstantní ukazatel
20         n += 1;
21     }
22     return n;
23 }
24 }
```

```
26 #include <stdio.h>
27 #include <stdbool.h>
28
29 static const char * const words[] = { "Alpha", ..., NULL };
30 int count_words_array(const char * const words[]);
31 int count_words(const char * const * const words);
32 bool check_alphabet_words(const char * const words[]);
33
34 int main(void)
35 {
36     int ret = EXIT_SUCCESS;
37     fprintf(stderr, "DEBUG: size %lu\\n", sizeof(words));
38     int n = count_words_array(words);
39     fprintf(stderr, "DEBUG: no. of words: %i\\n", n);
40     n = count_words(&words[0]);
41     fprintf(stderr, "DEBUG: no. of words: %i\\n", n);
42     bool checked = check_alphabet_words(words);
43     fprintf(stderr, "DEBUG: check_alphabet_words passed [%s]\\n", checked ? "OK" : "FAIL");
44     return ret;
45 }
46 }
```

Kódovací příklad – NATO Abeceda („jinak“) – 1/2

- Slova abecedy uložíme jako řetězec `alphabet` všech slov spojených bez mezery, do kterého budeme odkazovat na jednotlivá slova polem ukazatelů na textové řetězce (`words`).
 - Slovo je `'Z' - 'A' + 1`, ale řetězec je posloupnost znaků zakončená `'\0'`.
 - První písmeno slova abecedy používáme k indexaci, např. `'C'harlie` je odkazované ukazatelem `words['C' - 'A']`. První znak slova tak můžeme v abecedě `alphabet` nahradit znakem `'\0'` získáme textové řetězce. *Bez prvního znaku!*

```
1 //Ukazatel na textový literál. Literál nemůžeme měnit!
2 //static char *alphabet = "AlphaBravoCharlie...";
3 static char alphabet[] =
4     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
5     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
6     "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
7
8 //pole ukazatelů na textové řetězce
9 static char *words['Z' - 'A' + 1] = { [0] = NULL };
10 int fill_words(char* str, char *words[]);
11
12 int main(void)
13 {
14     int ret = fill_words(alphabet, words);
15     if (!ret) {
16         for (char c = 'A'; c <= 'Z'; ++c) {
17             fprintf(stderr, "DEBUG: %02d. '%c' - %c\\n",
18                 c, c, c, words[c - 'A']);
19         } //         ↳ První písmeno slova abecedy.
20     }
21     return ret;
22 }
```

```
24 int fill_words(char* alphabet, char *words[])
25 {
26     int ret = EXIT_SUCCESS;
27     char *cur = alphabet; // kurzor do pole s písmeny abecedy
28     for (char c = 'A'; c <= 'Z'; ++c) {
29         assert(words[c - 'A'] == NULL); // nemá být nastaveno
30         cur = strchr(cur, c); // vyhledání řetězce začínající c
31         assert(cur); // písmeno c musí být v abecedě
32         *cur = '\0';
33         words[c - 'A'] = ++cur; // nastavení a posun kurzoru
34         assert(words[c - 'A']); //it should be set now
35     }
36     return ret; // pragmaticky vždy EXIT_SUCCESS nebo assert.
37 }
```

- V implementaci použijeme (makro) `assert()` k testování správné inicializace datových struktur. *Makro slouží pro ladění, viz man assert.*

Kódovací příklad – NATO Abeceda („jinak“) – 2/2

- Přidáme překlad znaků načítaných ze `stdin` a implementaci zprehledníme.

```
1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
4     "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
5 static char *words['Z' - 'A' + 1] = { [0] = NULL };
6
7 int fill_words(char* str, char *words[]);
8 int main(void)
9 {
10     int ret = fill_words(alphabet, words);
11     if (!ret) {
12         for (char c = 'A'; c <= 'Z'; ++c) {
13             fprintf(stderr, "DEBU: %02d. '%c' - %c\n",
14                 c, c, words[c - 'A']);
15         } //          ↳ První písmeno slova abecedy.
16     }
17     int c;
18     while ((c = getchar()) != EOF) {
19         c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
20         // ↳ volání funkce toupper() bude přehlednější!
21         if (c >= 'A' && c <= 'Z') {
22             printf("%c%s ", c, words[c - 'A']);
23         }
24     }
25     return ret;
26 }
```

```
1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
4     "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
5 static char *words['Z' - 'A' + 1] = { [0] = NULL };
6
7 void fill_words(char* str, char *words[]);
8 void translate(char *words[]);
9
10 int main(void)
11 {
12     fill_words(alphabet, words);
13     translate(words);
14     return EXIT_SUCCESS;
15 }
16
17 void translate(char *words[])
18 {
19     int c;
20     while ((c = getchar()) != EOF) {
21         c = toupper(c); // funkce z #include<ctype.h>
22         if (c >= 'A' && c <= 'Z') {
23             printf("%c%s ", c, words[c - 'A']); // první znak!
24         }
25     }
26 }
27 }
```

- Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.

Kódovací příklad – struct 2/3

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

```
1 #include <stdio.h>
2 #include <limits.h>
3 #include <assert.h> // strukturální testy
4 #include "record.h"
5
6 record.c
```

```
33 void print_record(const struct record * record)
34 {
35     if (record) {
36         printf("Record\n");
37         printf("|- username: %s\n", record->
38             username);
39         printf("|- number: ");
40         print_chars(NUMBER_DIGITS, record->number);
41         printf("\n");
42         // printf("|- number: %s\n"); // Vyzkoušejte!
43     }
44 }
45
46 void print(const struct record * const records)
47 {
48     struct record const *cur = records;
49     while (cur && cur->username[0]) {
50         print_record(cur);
51         cur += 1; // pointer arithmetic
52     }
53 }
```

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

Kódovací příklad – struct 1/3

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

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

```
1 #define USERNAME_SIZE 8
2 #define NUMBER_DIGITS 4
3
4 struct record {
5     char username[USERNAME_SIZE + 1];
6     char number[NUMBER_DIGITS];
7 };
8
9 int main(void) {
10     struct record records[] = {
11         { .username = "user01" },
12         { .username = "user02" },
13         { .username = "admin" },
14         { .username = "root" },
15         { .username = '\0' } // null terminating
16         array
17     };
18     fprintf(stderr, "DEBU: size of struct %lu\n",
19         sizeof(struct record));
20     fprintf(stderr, "DEBU: size of records %lu\n",
21         sizeof(records));
22 }
```

- Položka `username` je o jeden znak delší, uložení `'\0'`.
 - Položka `number` je zápis čísla o maximální hodnotě 9999 (počet řádů 4) v textové podobě (0000–9999).
 - Velikost složeného typu je dána velikostí jednotlivých položek.
 - Pole záznamů inicializujeme se zarážkou (poslední prvek obsahuje prázdný řetězec v položce `username`).
- ```
$ clang struct.c && ./a.out
DEBU: size of struct 13
DEBU: size of records 6
```
- Na příkladu si ukážeme, jak převést celé číslo na textovou reprezentaci, k čemuž použijeme několik pomocných funkcí.
  - Implementujeme si funkce pro tisk záznamu a pole záznamů.
  - Položku `number` naplníme programově z celého čísla s kontrolou, zdali se číslo vejde do `NUMBER_DIGITS`.
  - Složený typ a implementaci funkcí realizujeme v samostatném modulu `record.h` a `record.c`.

## Kódovací příklad – struct 3/3

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

```
26 static void print_chars(size_t digits, const char *number)
27 { // number je ukazatel na konstantní hodnotu
28 for (size_t i = 0; i < digits; ++i, ++number) {
29 putchar(*number);
30 }
31 }
32
33 record.c
```

```
54 unsigned int fill_numbers(struct record * const records)
55 {
56 struct record *cur = records;
57 unsigned short n = 0;
58 const short max_number = get_max_number(NUMBER_DIGITS);
59 while (cur && cur->username[0]) {
60 assert(n <= max_number); // Vyplňujeme programově
61 fill_record_number(n, NUMBER_DIGITS, cur->number);
62 n += 1;
63 cur += 1;
64 }
65 return n;
66 }
67
68 record.c
```

- V programu máme snahu testovat velikost paměťové reprezentace.

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

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

```

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

```

```

24 const char *fname = "records.dat";
25 FILE *fd = fopen("records.dat", "w"); // uložení records
26 if (fd) {
27 size_t saved = fwrite(records, sizeof(struct record), n, fd);
28 size_t size = n * sizeof(struct record);
29 printf("DEBUG: saved bytes %lu out of %lu\n", saved * sizeof(
30 struct record), size);
31 } else {
32 fprintf(stderr, "ERROR: Cannot open \"%s\" for writing\n",
33 fname);
34 ret = EXIT_FAILURE;
35 }
36 return ret;
37
38 };
39
40 save_struct.c

```

```

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

```

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

```

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

```

- Načtení bloku dat funkcí fread().

```

23 while ((loaded = fread(records, sizeof(struct
24 record), NUM_RECORDS, fd)) {
25 fprintf(stderr, "DEBUG: loaded records %lu\n",
26 loaded);
27 for (size_t i = 0; i < loaded; ++i) {
28 print_record(&records[i]);
29 }
30 free(records);
31 fclose(fd);
32 goto leave;
33 }
34 error:
35 fprintf(stderr, "ERROR: during reading from the
36 file \"%s\"\n", fname);
37 return 1;
38 }
39
40 leave:
41 return 0;
42 }
43
44 save_struct.c

```

```

$ clang load_struct.c record.o -o load && ./load
DEBUG: loaded records 2
...
Record
|- username: "root"
|- number: 0003
DEBUG: loaded records 1
Record
|- username: "jf"
|- number: 0004

```

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

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

```

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

```

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

```

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

```

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "record.h"
4 #define NUM_RECORDS 2
5 enum { ERROR_FILE = 101, ERROR_MEM = 102 };
6 void print_error(int error);
7 int main(void) {
8 int ret = EXIT_SUCCESS;
9 const char *fname = "records.dat";
10 FILE *fd = fopen(fname, "r");
11 if (!fd && (ret = ERROR_FILE)) { // ret!
12 goto leave;
13 }
14 struct record *records = malloc(
15 NUM_RECORDS * sizeof(struct record));
16 if (!records && (ret = ERROR_MEM)) { //
17 ret!
18 goto leave;
19 }
20 while ((loaded = fread(records, sizeof(struct
21 record), NUM_RECORDS, fd)) {
22 fprintf(stderr, "DEBUG: loaded records %lu\n", loaded);
23 for (size_t i = 0; i < loaded; ++i) {
24 print_record(&records[i]);
25 }
26 }
27 free(records);
28 return ret;
29 }
30
31 leave:
32 if (fd) {
33 fclose(fd);
34 }
35 print_error(ret);
36 return ret;
37 }
38
39 void print_error(int error) {
40 switch (error) {
41 case ERROR_FILE:
42 fprintf(stderr, "ERROR: open file\n");
43 break;
44 case ERROR_MEM:
45 fprintf(stderr, "ERROR: mem allocation\n");
46 break;
47 }
48 }
49
50 }
51 }

```

## Shrnutí přednášky

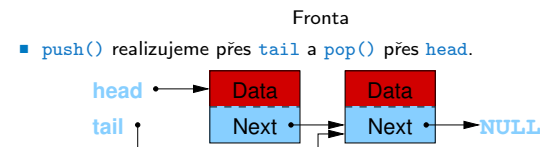
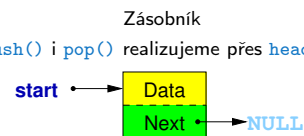
Part V  
Appendix

## Diskutovaná témata

- Numerická přesnost.
- Knihovna `gmp`.
- Maticové násobení a organizace paměti.
- Rychlost výpočtu a architektura procesoru.
- Paralení výpočty `OpenMP`.

## ADT – zásobník a fronta

- Obě datové struktury mají stejné rozhraní, např. `push()`, `pop()`, `isEmpty()`.
- Zásobník vs. fronta se liší chováním, tj. jaký prvek vrací při vyjmutí.
- Obě struktury můžeme implementovat polem nebo spojovým seznamem.
- Implementace polem (definované kapacity)
  - Zásobník inkrementujeme/dekrementujeme pouze index na volný prvek v poli.
  - Frontu implementujeme kruhovou frontou v poli, indexy na první a poslední prvek pouze inkrementujeme modulo kapacita pole.
- Postačí jednosměrný spojový seznam, implementace se liší kam přidáváme nové prvky.
  - Přidávání je snadné před první prvek (`head`) nebo za poslední prvek.
  - Odebrání je snadné pro první prvek (`head`).



## Kódovací příklad – Číslo karty

- Implementujte program, který nahradí prvních 12 cifer 16 ciferného čísla karty znakem \* a zobrazí pouze poslední čtyřčíslí.
- Čísla uvažujte jako hodnoty typu `long`.
- Implementujte s využitím pole.

- Implementujte s využitím dynamické alokace.

■ Program rozšířte o načítání čísel karet ze souboru.

■ Program dále rozšířte o kontrolní součet cifer čísla karty, např. s využitím Luhnova algoritmu.

<https://www.cssz.cz/kontrola-icpe-pomoci-luhnova-algoritmu>

```
$ clang card.c && ./a.out
9876543210987654 -> *****7654
1111222233334444 -> *****4444
5555444433332222 -> *****2222
1234567890123456 -> *****3456

char* hide_card_number(long int card);
int main(void)
{
 long nums[] = {9876543210987654l,
 1111222233334444l, 5555444433332222l,
 1234567890123456l };
 const int n = sizeof(nums) / sizeof(long);
 for (int i = 0; i < n; ++i)
 printf("%ld -> %s\n", nums[i],
 hide_card_number(nums[i]));
 return 0;
}
```

## Kódovací příklady

- Ve vstupním řetězci změňte velikost písmen tak, aby první písmeno ve slově bylo vždy velké a další malá. *Title Case*
- Ve vstupním řetězci změňte velikost písmen tak, aby se v každém slově střídala velká-malá písmena. *SpOnGeCaSe*
- Ve vstupním řetězci (větě) zaměňte pořadí slov. *Reverse words*
- Ve vstupním řetězci uspořádejte znaky abecedy vzestupně a na konec řetězce vložte hodnotu součet cifer ve vstupním řetězci. *Rearrange string*  
Např. "gzve534" → "egvz12".
- Převeďte vstupní řetězec o jednom až čtyřech slovech na čtyřpísmený ptačí kód. *4-Letter Birds Code*  
Např. "Arctic Tern" → "ARTE".
- Napište program, který k zadanému číslu najde nejbližší vyšší číslo použitím stejných číslic. *Next higher number*

Např. 231334113 → 231334131 nebo 897654321 → 912345678.

*Spíš algoritmizace než programování. Lze implementovat na cca 50 řádků.*

*Použijete pro návrh řešení poznámky v textovém souboru nebo spíše „tužku a papír“ (fixu a tabuly)?*