Jazyk C – Část II

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 9

A0B36PR2 – Programování 2

Část 1 – Příklad – Pracujeme s ukazateli

Část 2 – Jazyk C - struktury a uniony

Část 3 – Jazyk C - základní knihovny, dynamická alokace paměti, soubory

Část I

Část 1 – Příklad – Pracujeme s ukazately

Zadání

- Implementuje program, který bude simulovat jednoduchý 1D binární celulární automat
- Automat je reprezentovaný řádkem o `SIZE` znacích
- Každá buňka nabývá hodnoty `'-'` nebo `'*'`
- Hodnota buňky v dalším stavu je určena na základě 1 okolí buňky a definovanými pravidly
- Uvažujte řádek „uzavřený" do kruhu, tj.
  - Předchozí buňka první buňky je buňka poslední
  - Následující buňka poslední buňky je buňka první

lec09/lec09-demo.c

Pravidla

1. – – – → *
2. * – – → –
3. – *– → –
4. * * – → *
5. – – * → –
6. * – * → *
7. – * * → –
8. * * * → *

lec09/lec09-demo.c

Definice pravidel v programu

```
char *rules_definition[] =
                { "*", "-", "-", "*", "-", "*", "-", "*" };
void generate_patterns(char *patterns, int n, int s) {
   char *cur = patterns;
   int v[s], b[s], wb[s];
   for(int i = 0; i < s; ++i) {
      v[i] = 1<<i; b[i] = 0; wb[i] = 1;
   }
   for(int i = 0; i < n; ++i) {
      for(int j = 0; j < s-1; ++j) {
         *(cur++) = b[j] ? ACTIVE : EMPTY;
         if (wb[j] >= 0 && (wb[j] % v[j]) == 0) {
            b[j] = !b[j];
         }
         wb[j]++;
      }
      *(cur++) = '\0';
} }

char patterns[8*4];
generate_patterns(patterns, 8, 4);
```

lec09/lec09-demo.c

Pomocné funkce

```
#define SIZE 100
#define ACTIVE '*'
#define EMPTY  '-'

void random_line(char *line, int size) {
   for(int i = 0; i < size; ++i) {
      line[i] = (random() % 50 < 25) ? ACTIVE : EMPTY;
   }
}

void print_line(char *line, int size) {
   for(int i = 0; i < size; ++i) {
      putchar(*(line++));
   }
   putchar('\n');
}

void swap(char **s1, char **s2) {
   char *s = *s1;
   *s1 = *s2;
   *s2 = s;
}
```

lec09/lec09-demo.c

Nalezení pravidla

```
int find_pattern(char *start, char *patterns, int
    patterns_size) {
   int pattern_idx = -1;
   for(int i = 0; i < patterns_size; ++i) {
      int ok = 1;
      char *cur_pattern = patterns + i * 4; /
      char *c = start;
      while(*cur_pattern != '\0') {
         if (*cur_pattern != *c) {
            ok = 0;
            break;
         }
         cur_pattern++;
         c++;
      }
      if (ok) {
         pattern_idx = i;
         break;
      }
   }
   return pattern_idx;
}
```

lec09/lec09-demo.c

Změna stavu

```
void evolve_line(char *src, char *dst, int line_size, char *
    patterns, char *rules, int rule_size)  {

   /* replicate last character to prev of the 1st one */
   *src = *(src + line_size);

   /* replicate 1st character as the next of the last one */
   *(src + line_size + 1) = *(src + 1);
   for(int i = 0; i < line_size; ++i) {
      int r = find_pattern(src++, patterns, rule_size);
      if (r != -1) {
         *(++dst) = *(rules + r * 2);
      } else {
         fprintf(stderr, "Pattern does not match!\n");
      }
   }
}
```

lec09/lec09-demo.c

Inicializace pravidel a výpisy

- Velikost pole `rules_definition` je větší než $8 * 2$

Viz dále

```
int main(int argc, char *argv[]) {
   printf("rules size %lu\n", sizeof(rules_definition));
   char *cu = rules_definition[0];
   for(int i = 0; i < 8; ++i) {
      char c = *(cu++);
      printf("rul def[%d] - %s\n", i, rules_definition[i]);
   }
   char rules[8*2]; /* because of padding */
   char *rules_p = rules;
   for(int i = 0; i < 8; ++i) {
      for(int j = 0; j < 2; ++j) {
         *(rules_p++) = rules_definition[i][j];
      }
   }
   for(int i = 0; i < 8; ++i) {
      printf("rule[%d] - %s\n", i, rules + i * 2);
   }
```

lec09/lec09-demo.c

Inicializace vzorů

- Vzory (levé strany pravidel) vygenerujeme

```
/* +2 for prev and next of the first and last character */
char lines[2][SIZE + 2];
for(int i = 0; i < 2; ++i) {
 lines[i][SIZE + 1] = '\0'; /* null termination for print */
}
random_line(lines[0] + 1, SIZE); /* +1 -- 1st is for prev */
printf("Random line '%s'\n", lines[0]+1);
print_line(lines[0]+1, SIZE);
putchar('\n');

char patterns[8*4];
generate_patterns(patterns, 8, 4);
char *cur = patterns;
for(int i = 0; i < 8; ++i) {
   printf("Patterns[%d] = '%s'\n", i, cur);
   cur += 4;
}
```

lec09/lec09-demo.c

Alternace řádků `lines[][]`

```
char *lines1 = lines[0];
char *lines2 = lines[1];

printf("Evolve lines\n");
while(1) {
   print_line(lines1 + 1, SIZE);
   evolve_line(lines1, lines2, SIZE, patterns, rules, 8);
   print_line(lines2 + 1, SIZE);
   swap(&lines1, &lines2);
}
```

lec09/lec09-demo.c

Část II

Část 2 – Jazyk C - struktury a uniony

Modifikátor typu `const`

- Uvedením klíčového slova **const** můžeme označit proměnnou jako konstantu

Překladač kontroluje přiřazení

- Můžeme použít pro definici konstant, např.

```
const float pi = 3.14159265;
```

- Na rozdíl od symbolické konstanty

```
#define PI 3.14159265
```

- mají konstantní proměnné typ a překladač tak může provádět **typovou kontrolu**

Ukazatele na konstantní proměnné a konstantní ukazatele

- Klíčové slovo **const** můžeme zapsat před jméno typu nebo před jméno proměnné
- Dostáváme 3 možnosti jak definovat ukazatel s `const`
  (a) `const int *ptr;` – ukazatel na konstantní proměnnou
    - Nemůžeme použít pointer pro změnu hodnoty proměnné
  (b) `int* const ptr;` – konstantní ukazatel
    - Pointer nemůžeme nastavit na jinou adresu než tu při inicializaci
  (c) `const int* const ptr;` – konstantní ukazatel na konstantní hodnotu
    - Kombinuje předchozí dva případy

lec09/const_pointers.c

Příklad – Ukazatel na konstantní proměnnou

- Prostřednictvím ukazatele na konstantní proměnnou tuto proměnnou měnit nemůžeme

```
1  int v = 10;
2  int v2 = 20;
3
4  const int *ptr = &v;
5  printf("*ptr: %d\n", *ptr);
6
7  *ptr = 11; /* THIS IS NOT ALLOWED! */
8
9  v = 11;  /* We can modify the original variable */
10 printf("*ptr: %d\n", *ptr);
11
12 ptr = &v2; /* We can assign new address to ptr */
13 printf("*ptr: %d\n", *ptr);
```

lec09/const_pointers.c

Příklad – Konstantní ukazatel

- Hodnotu konstantního ukazatele nelze po inicializaci měnit
- Zápis `int *const ptr;` můžeme číst zprava doleva
  - `ptr` – proměnná, která je
  - `*const` – konstantním ukazatelem
  - `int` – na proměnnou typu `int`

```
1 int v = 10;
2 int v2 = 20;
3 int *const ptr = &v;
4 printf("v: %d *ptr: %d\n", v, *ptr);
5
6 *ptr = 11; /* We can modify addressed value */
7 printf("v: %d\n", v);
8
9 ptr = &v2; /* THIS IS NOT ALLOWED! */
```

lec09/const_pointers.c

Příklad – Konstantní ukazatel na konstantní proměnnou

- Hodnotu konstantního ukazatele na konstantního proměnnou nelze po inicializaci měnit a ani nelze prostřednictvím takového ukazatele měnit hodnotu adresované proměnné
- Zápis `const int *const ptr;` můžeme číst zprava doleva
  - `ptr` – proměnná, která je
  - `*const` – konstantním ukazatelem
  - `const int` – na proměnnou typu `const int`

```
1 int v = 10;
2 int v2 = 20;
3 const int *const ptr = &v;
4
5 printf("v: %d *ptr: %d\n", v, *ptr);
6
7 ptr = &v2; /* THIS IS NOT ALLOWED! */
8 *ptr = 11; /* THIS IS NOT ALLOWED! */
```

lec09/const_pointers.c

Ukazatel na funkci

- Implementace funkce je umístěna někde v paměti a podobně jako na proměnnou v paměti může ukazatel odkazovat na paměťové místo s definicí funkce
- Můžeme definovat **ukazatel na funkci** a dynamicky volat funkci dle aktuální hodnoty ukazatele
- Součástí volání funkce jsou předávané parametry, které jsou též součástí typu ukazatele na funkci
- Funkce (a volání funkce) je identifikátor funkce a **()**, tj.
  `typ_návratové_hodnoty funkce(parametry funkce);`
- Ukazatel na funkci definujeme jako

```
typ_návratové_hodnoty (*ukazatel)(parametry funkce);
```


Příklad – Ukazatel na funkci 1/2

- Používáme dereferenční operátor * podobně jako u proměnných

```
double do_nothing(int v); /* function prototype */

double (*function_p)(int v); /* pointer to function */

function_p = do_nothing; /* assign the pointer */

(*function_p)(10); /* call the function */
```

- Závorky `(*function_p)` „*pomáhají*" číst definici ukazatele
  Můžeme si představit, že závorky reprezentují jméno funkce. Definice proměnné ukazatel na funkci se tak v zásadě neliší od prototypu funkce.
- Podobně je volání funkce přes ukazatel na funkci identické běžnému volání funkce, kde místo jména funkce vystupuje v závorce jméno ukazatele na funkci

Příklad – Ukazatel na funkci 2/2

- V případě funkce vracející ukazatel postupujeme identicky

```
double* compute(int v);

double* (*function_p)(int v);
        ^^^^^^^^^^^^^---- substitute a function name

function_p = compute;
```

- Příklad použití ukazatele na funkci – `lec09/pointer_fnc.c`
- Ukazatele na funkce umožňují realizovat dynamickou vazbu volání funkce identifikované za běhu programu
  V objektově orientovaném programování je dynamická vazba klíčem k realizaci polymorfismu.

Definice typu – typedef

- Operátor `typedef` umožňuje definovat nový datový typ
- Slouží k pojmenování typů, např. ukazatele, struktury a uniony
- Například typ pro ukazatele na `double` a nové jméno pro `int`:

```
1 typedef double* double_p;
2 typedef int integer;
3 double_p x, y;
4 integer i, j;
```

- je totožné s použitím původních typů

```
1 double *x, *y;
2 int i, j;
```

- Zavedením typů operátorem `typedef`, např. v hlavičkovém souboru, umožňuje systematické používání nových jmen typů v celém programu *Viz např.* `inttypes.h`
- Výhoda zavedení nových typů je především u složitějších typů jako jsou ukazatele na funkce nebo struktury

Struktura – struct

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům struktury **přistupujeme tečkovou notací**
- K prvkům můžeme přistupovat přes ukazatel operátorem ->
- Pro struktury stejného typu je definována operace přiřazení

  `struct1 = struct2;`

  Pro proměnné typu pole není přímé přiřazení definováno, pouze po prvcích.
- Struktury (jako celek) **nelze** porovnávat relačním operátorem ==
- Struktura může být funkci předávána hodnotou i odkazem
- Struktura může být návratovou hodnotou funkce

Příklad struct – Definice

- Bez zavedení nového typu (typedef) je nutné před identifikátor jména struktury uvádět klíčové slovo struct

```
struct record {
   int number;
   double value;
};
```

```
typedef struct {
   int n;
   double v;
} item;
```

```
record r; /* THIS IS NOT ALLOWED!*/
          /* Type record is not known */

struct record r; /* Keyword struct is required */
item i;     /* type item defined using typedef */
```

- Zavedením nového typu typedef můžeme používat typ struktury již bez uvádění klíčového slova struct

lec09/struct.c

Příklad struct – Inicializace

- Struktury:

```
struct record {
   int number;
   double value;
};
```

```
typedef struct {
   int n;
   double v;
} item;
```

- Proměnné typu struktura můžeme inicializovat prvek po prvku

```
struct record r;
r.value = 21.4;
r.number = 7;
```

- nebo podobně jako pole lze inicializovat přímo

```
item i = { 1, 2.3 };
```

lec09/struct.c

Příklad struct jako parametr funkce

- Struktury můžeme předávat jako parametry funkcí hodnotou

```
void print_record(struct record rec) {
   printf("record: number(%d), value(%lf)\n",
   rec.number,  rec.value);
}
```

- Nebo odkazem

```
void print_item(item *v) {
   printf("item: n(%d), v(%lf)\n", v->n, v->v);
}
```

- Při předávání parametru
  - **hodnotou** se vytváří nová proměnná a původní obsah předávané struktury se kopíruje na zásobník
  - **odkazem** se kopíruje pouze hodnota ukazatele (adresa) a pracujeme s původní strukturou

lec09/struct.c

Příklad struct – Přiřazení

- Hodnoty proměnné stejného typu struktury můžeme přiřadit operátorem =

```
struct record {
   int number;
   double value;
};
```

```
typedef struct {
   int n;
   double v;
} item;
```

```
struct record rec1 = { 10, 7.12 };
struct record rec2 = { 5, 13.1 };
item i;
print_record(rec1); /* number(10), value(7.120000) */
print_record(rec2); /* number(5), value(13.100000) */
rec1 = rec2;
i = rec1; /* THIS IS NOT ALLOWED! */
print_record(rec1); /* number(5), value(13.100000) */
```

lec09/struct.c

Příklad struct – Přímá kopie paměti

- Jsou-li dvě struktury stejně veliké, můžeme přímo kopírovat obsah příslušné paměťové oblasti

Například funkcí `memcpy()` *z knihovny* `string.h`

```
struct record r = { 7, 21.4};
item i = { 1, 2.3 };
print_record(r); /* number(7), value(21.400000) */
print_item(&i); /* n(1), v(2.300000) */
if (sizeof(i) == sizeof(r)) {
   printf("i and r are of the same size\n");
   memcpy(&i, &r, sizeof(i));
   print_item(&i); /* n(7), v(21.400000) */
}
```

- V tomto případě je interpretace hodnot v obou strukturách identická, obecně tomu však být nemusí

lec09/struct.c

Struktura struct a velikost

- Vnitřní reprezentace struktury nutně nemusí odpovídat součtu velikostí jednotlivých prvků

```
struct record {
   int number;
   double value;
};
```

```
typedef struct {
   int n;
   double v;
} item;
```

```
printf("Size of int: %ld size of double: %ld\n", sizeof
    (int), sizeof(double));
printf("Size of record: %ld\n", sizeof(struct record));
printf("Size of item: %ld\n", sizeof(item));

Size of int: 4 size of double: 8
Size of record: 16
Size of item: 16
```

lec09/struct.c

Struktura struct a velikost 1/2

- Při kompilaci zpravidla dochází k zarovnání prvků na velikost slova příslušné architektury

Např. 8 bytů v případě 64-bitové architektury.

- Můžeme explicitně předepsat kompaktní paměťovou reprezentaci, např. direktivou `__attribute__((packed))` pro překladače `clang` a `gcc`

```
struct record_packed {
   int n;
   double v;
} __attribute__((packed));
```

lec09/struct.c

Struktura struct a velikost 2/2

- Nebo

```
typedef struct __attribute__((packed)) {
   int n;
   double v;
} item_packed;
```

- Příklad výstupu:

```
printf("Size of int: %ld size of double: %ld\n", sizeof(int),
     sizeof(double));
printf("record_packed: %ld\n", sizeof(struct record_packed));
printf("item_packed: %ld\n", sizeof(item_packed));
```

```
Size of int: 4 size of double: 8
Size of record_packed: 12
Size of item_packed: 12
```

lec09/struct.c

- Zarovnání zpravidla přináší rychlejší přístup do paměti, ale zvyšuje paměťové nároky

http://www.catb.org/esr/structure-packing

Proměnné se sdílenou pamětí – union

- **Union** je množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionu sdílejí společně stejná paměťová místa

Překrývají se

- Velikost unionu je dána velikostí největšího z jeho prvků
- Skladba unionu je definována uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům unionu se přistupuje tečkovou notací
- Pokud nedefinujeme nový typ je nutné k identifikátoru proměnné unionu uvádět klíčové slovo union

```
1 union Nums {
2    char c;
3    int i;
4 };
5 Nums nums; /* THIS IS NOT ALLOWED! Type Nums is not known! */
6 union Nums nums;
```


Příklad union 1/2

- Union složený z proměnných typu: char, int a double

```
1  int main(int argc, char *argv[]) {
2     union Numbers {
3        char c;
4        int i;
5        double d;
6     };
7     printf("size of char %ld\n", sizeof(char));
8     printf("size of int %ld\n", sizeof(int ));
9     printf("size of double %ld\n", sizeof(double));
10    printf("size of Numbers %ld\n", sizeof(union Numbers));
11
12    union Numbers numbers;
13
14    printf("Numbers c: %d i: %d d: %lf\n", numbers.c,
       numbers.i, numbers.d);
```

- Příklad výstupu:

```
size of char 1
size of int 4
size of double 8
size of Numbers 8
Numbers c: 48 i: 740313136 d: 0.000000
```

lec09/union.c

Příklad union 2/2

- Proměnné sdílejí paměťový prostor

```
1  numbers.c = 'a';
2  printf("\nSet the numbers.c to 'a'\n");
3  printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i,
       numbers.d);
4
5  numbers.i = 5;
6  printf("\nSet the numbers.i to 5\n");
7  printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i,
       numbers.d);
8
9  numbers.d = 3.14;
10 printf("\nSet the numbers.d to 3.14\n");
11 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i,
       numbers.d);
```

- Příklad výstupu:

```
Set the numbers.c to 'a'
Numbers c: 97 i: 1374389601 d: 3.140000

Set the numbers.i to 5
Numbers c: 5 i: 5 d: 3.139999

Set the numbers.d to 3.14
Numbers c: 31 i: 1374389535 d: 3.140000
```

lec09/union.c

Standardní knihovny

- Jazyk C sám osobě neobsahuje prostředky pro vstup/výstup dat, složitější matematické operace ani:
  - práci z textovými řetězci
  - správu paměti pro dynamické přidělování
  - vyhodnocení běhových chyb (run-time errors)
- Tyto a další funkce jsou obsaženy ve standardních knihovnách, které jsou součástí překladače jazyka C
  - Knihovny – přeložený kód se připojuje k programu, např `libc.so`
  - Hlavičkové soubory – obsahují prototypy funkcí, definici typů, makra a konstanty a vkládají se do zdrojových souborů příkazem preprocesoru `#include <jmeno_knihovny.h>`

Např. `#include<stdio.h>`

Standardní knihovny

- `stdio.h` – Vstup a výstup (formátovaný i neformátovaný)
- `stdlib.h` – Matematické funkce, alokace paměti, převod řetězců na čísla, řazení (`qsort`), vyhledávání (`bsearch`), generování náhodných čísel (`rand`)
- `limits.h` – Rozsahy číselných typů
- `math.h` – Matematické funkce
- `errno.h` – Definice chybových hodnot
- `assert.h` – Zpracování běhových chyb

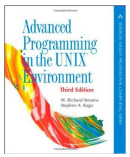
- `ctype.h` – Klasifikace znaků (`char`)
- `string.h` – Řetězce, blokové přenosy dat v paměti (`memcpy`)
- `locale.h` – Internacionalizace
- `time.h` – Datum a čas

Standardní knihovny (POSIX)

Komunikace s operačním systémem (OS)

- `stdlib.h` – Funkce využívají prostředků OS
- `signal.h` – Asynchronní události,vlákna
- `unistd.h` – Procesy, čtení/zápis souborů, ...
- `pthread.h` – Vlákna (POSIX Threads)
- `threads.h` – Standardní knihovna pro práci s vlákny (C11)

Advanced Programming in the UNIX Environment, 3rd edition, *W. Richard Stevens, Stephen A. Rago* Addison-Wesley, 2013, ISBN 978-0-321-63773-4



Část III

Část 3 – Jazyk C - základní knihovny, dynamická alokace paměti, soubory

Dynamická alokace paměti

- `void* malloc(size);` – přidělení bloku paměti z haldy (heap)
  - Velikost alokované paměti je uložena ve správci paměti
  - **Není však součástí ukazatele**
  - Je plně na uživateli (programátorovi), jak bude s pamětí zacházet
- Příklad alokace pole 10 proměnných typu `int`

```
1 int *int_array;
2 int_array = (int*)malloc(10 * sizeof(int));
```

- Práce s polem je identická jako se statickým polem
  - Používáme pointerovou aritmetiku

Pro statické pole používáme `int_array[i]`

- `void* free(pointer);` – Uvolnění paměti zpět do haldy
  - Správce paměti uvolní paměť asociovanou k ukazateli
  - Hodnotu ukazatele však nemění

Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutně povést
- Pro vyplnění adresy alokované paměti předáváme proměnnou ukazatele odkazem

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void* allocate_memory(int size, void **ptr) {
5     *ptr = malloc(size);
6     if (*ptr == NULL) {
7        fprintf(stderr, "Error: allocation fail");
8        exit(-1); /* exit program if allocation fail */
9     }
10    return *ptr;
11 }
```

lec09/malloc_demo.c

Příklad alokace dynamické paměti 2/3

- Pro vyplnění hodnot pole alokovaného dynamicky nám postačuje předávat hodnotu adresy paměti pole

```
1 void fill_array(int* array, int size) {
2    for(int i = 0; i < size; ++i) {
3       *(array++) = random();
4    }
5 }
```

- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto můžeme explicitně nulovat

```
1 void deallocate_memory(void **ptr) {
2    if (ptr != NULL && *ptr != NULL) {
3       free(*ptr);
4       *ptr = NULL;
5    }
6 }
```

lec09/malloc_demo.c

Příklad alokace dynamické paměti 3/3

- Příklad použití

```
1  int main(int argc, char *argv[]) {
2     int *int_array;
3     const int size = 4;
4
5     allocate_memory(sizeof(int) * size, (void**)&
       int_array);
6     fill_array(int_array, size);
7     int *cur = int_array;
8     for(int i = 0; i < size; ++i, cur++) {
9        printf("Array[%d]=%d\n", i, *cur);
10    }
11    deallocate_memory((void**)&int_array);
12    return 0;
13 }
```

lec09/malloc_demo.c

Základní práce se soubory – otevření souboru

- Knihovna `stdio.h`
- Přístup k souboru `FILE *f;`
- Otevření souboru

  `FILE *fopen(char *filename, char *mode);`
- Práce s binárními i textovými soubory
- Soubory jsou čteny/zapisovány sekvenčně
  - Se soubory se pracuje jako s proudem dat
  - Aktuální „pozici" v souboru si můžeme představit jako kurzor
  - Při otevření souboru se kurzor nastavuje na začátek souboru
- Režim práce se souborem je dán hodnotou proměnné `mode`
  - `"r"` – režim čtení,
  - `"w"` – režim zápisu

    Vytvoří soubor, pokud neexistuje, jinak smaže obsah souboru
  - `"a"` – režim přidávání do souboru

    Kurzor je nastaven na konec souboru.
  - Můžeme otevřít s příznakem `+`, např. `"+r"` pro otevření souboru pro čtení i zápis

    viz `man fopen`

Testování – otevření/zavření souboru

- Testování otevření souboru

```
1 char * fname = "file.txt";
2
3 if ((f = fopen(fname, "r")) == NULL) {
4    fprintf(stderr, "Error: open file '%s'\n",
      fname);
5 }
```

- Zavření souboru – `int fclose(FILE *file);`

```
1 if (fclose(f) == EOF) {
2    fprintf(stderr, "Error: close file '%s'\n",
      fname);
3 }
```

- Dosažení konce souboru – `int feof(FILE *file);`

Příklad – čtení souboru znak po znaku

- Čtení znaku: `int getc(FILE *file);`
- Hodnota znaku (`unsigned char`) je vrácena jako `int`

```
1 int count = 0;
2 while ((c = getc(f)) != EOF) {
3    printf("Read character %d is '%c'\n", count,
      c);
4    count++;
5 }
```

lec09/read_file.c

- Pokud nastane chyba nebo konec souboru vrací funkce `getc` hodnotu `EOF`
- Pro rozlišení chyby a konce souboru lze využít funkce `feof()` a `ferror()`

Formátované čtení ze souboru

- `int fscanf(FILE *file, const char *format, ...);`
- Analogie formátovanému výstupu – hodnoty jsou předávány odkazem
- Vrací počet přečtených položek, například pro vstup

```
record 1 13.4
```

- příkaz: `int r = fscanf(f, "%s %d %lf\n", str, &i, &d);`
- bude hodnota proměnné

  `r == 3`
- Při čtení textového řetězce je nutné zajistit dostatečný paměťový prostor pro načítaný textový řetězec, např. omezením velikosti řetězce

```
char str[10];
int r = fscanf(f, "%9s %d %lf\n", str, &i, &d);
```

lec09/file_scanf.c

Zápis do souboru

- Po znaku – `int putc(int c, FILE *file);`
- Formátovaný výstup

  `int fprintf(FILE *file, const *format, ...);`

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

lec09/file_printf.c

- Identicky lze použít `stdin`, `stdout`, `stderr`

Náhodný přístup k souborům – `fseek()`

- Nastavení pozice kurzoru v souboru relativně vůči `whence` v bajtech
- `int fseek(FILE *stream, long offset, int whence);`, kde `whence`
  - `SEEK_SET` – nastavení pozice od začátku souboru
  - `SEEK_CUR` – relativní hodnota vůči současné pozici v souboru
  - `SEEK_END` – nastavení pozice od konce souboru
- `fseek()` vrací 0 v případě úspěšného nastavení pozice

- Nastavení pozice v souboru na začátek

  `void rewind(FILE *stream);`

Binární čtení/zápis z/do souboru

- Pro čtení a zápis bloku dat můžeme využít funkce `fread` a `fwrite` z knihovny `stdio.h`
- Načtení `nmemb` prvků, každý o velikosti `size` bajtů

  `size_t fread(void* ptr, size_t size, size_t nmemb, FILE *stream);`
- Zápis `nmemb` prvků, každý o velikosti `size` bajtů

  `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
- Funkce vrací počet přečtených/zapsaných bajtů
- Pokud došlo k chybě nebo detekci konce souboru funkce vrací menší než očekávaný počet bajtů

Práce s textovými řetězci

- V C jsou řetězce pole znaků zakončené znakem `'\0'`
- Základní operace jsou definovány v knihovně `string.h`, například pro kopírování nebo porovnání řetězců
  - `char* strcpy(char *dst, char *src);`
  - `int strcmp(const char *s1, const char *s2);`
  - Funkce předpokládají dostatečný rozsah alokovaných polí
  - Funkce s explicitním limitem na maximální délku řetězců:

    `char* strncpy(char *dst, char *src, size_t len); int strncmp(const char *s1, const char *s2, size_t len);`
- Převod řetězce na číslo – `stdlib.h`
  - ~~`atoi()`~~, ~~`atof()`~~ – převod celého a necelého čísla
  - `long strtol(const char *nptr, char **endptr, int base);`
  - `double strtod(const char *nptr, char **restrict endptr);`

`man strcpy`, `man strncmp`, `strtol`, `strtod`

Zpracování chyb

- Základní chybové kódy jsou definovány v `errno.h`
- Tyto kódy jsou ve standardních C knihovnách používány jako příznaky nastavené v případě selhání volání funkce v globální proměnné `errno`
- Například otevření souboru `fopen()` vrací hodnotu `NULL`, pokud se soubor nepodařilo otevřít
- Z této hodnoty, ale nepoznáme proč volání selhalo
- Pro funkce, které nastavují `errno`, můžeme podle hodnoty identifikovat důvod chyby
- Textový popis číselných kódů pro standardní knihovnu C je definován v `string.h`
- Řetězec můžeme získat voláním funkce

```
char* strerror(int errnum);
```


Příklad použití errno

- Otevření souboru

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6     FILE *f = fopen("soubor.txt", "r");
7     if (f == NULL) {
8        int r = errno;
9        printf("Open file failed errno value %d\n", errno);
10       printf("String error '%s'\n", strerror(r));
11    }
12    return 0;
13 }
```

lec09/errno.c

- Výstup při neexistujícím souboru

```
Open file failed errno value 2
String error 'No such file or directory'
```

- Výstup při pokusu otevřít soubor bez práv přístupu k souboru

```
Open file failed errno value 13
String error 'Permission denied'
```


Testovací makro assert

- Do kódu můžeme přidat podmínky na hodnoty proměnných, které jsou nutné pro další běh programu
- Při nesplnění podmínky program vypíše jméno souboru a řádek, kde došlo chybě
- Takový test je definován jako makro `assert` v knihovně `assert.h`
- Makro vloží příslušný kód do programu
- Vložení makra lze zabránit kompilací s definováním makra `NDEBUG`

man assert

- Příklad

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

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

lec09/assert.c

Příklad použití makra assert

- Kompilace s makrem a spuštění programu bez/s argumentem

```
clang assert.c -o assert
./assert
Assertion failed: (argc > 1), function main, file assert.c
    , line 5.
zsh: abort      ./assert

./assert 2
start argc: 2
```

- Kompilace bez makra a spuštění programu bez/s argumentem

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

lec09/assert.c

Matematické funkce

- `math.h` – základní funkce pro práci s „reálnými" čísly
  - Výpočet odmocniny necelého čísla `x`
    `double sqrt(double x);`, `float sqrtf(float x);`
    V C funkce nepřetěžujeme, proto jsou jména odlišena
  - `double pow(double x, double y);` – výpočet obecné mocniny
  - `double atan2(double y, double x);` – výpočet *arctan* y/x s určením kvadrantu
  - Symbolické konstanty – `M_PI`, `M_PI_2`, `M_PI_4`, atd.
    - `#define M_PI 3.14159265358979323846`
    - `#define M_PI_2 1.57079632679489661923`
    - `#define M_PI_4 0.78539816339744830962`
  - `isfinite()`, `isnan()`, `isless`, `...` – makra pro porovnání reálných čísel.
- `complex.h` – funkce pro počítání s komplexními čísly
  ISO C99
- `fenv.h` – funkce pro řízení zaokrouhlování a reprezentaci dle IEEE 754.

Shrnutí přednášky

Diskutovaná témata

- Jazyk C – Modifikátor `const`
- Jazyk C – Ukazatel na konstantní proměnnou
- Jazyk C – Konstantní ukazatel
- Jazyk C – Ukazatel na funkci
- Jazyk C – Definice typu `typedef`
- Jazyk C – Struktury a uniony
- Jazyk C – Dynamická alokace paměti
- Jazyk C – Práce se soubory
- Jazyk C – Funkce standardní knihovny C

- Příště: příklad programu v C