Standardní knihovny C. Kódovací příklady.

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 07

BAB36PRGA – Programování v C

Přehled témat

Část I

Část 1 – Standardní knihovny, čtení/zápis ze souboru

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`.

    Viz např. `ldd a.out`.
  - 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

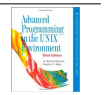
Standardní knihovny (POSIX)

Komunikace s operačním systémem (OS).

POSIX – Portable Operating System Interface

- `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



Základní práce se soubory – otevření souboru

- Knihovna `stdio.h`.
- Přístup k souboru je prostřednictvím ukazatele `FILE*`.
- Otevření souboru `FILE *fopen(char *filename, char *mode);`.
- Práce s textovými a binárními (*modifikátor* `"b"`) soubory.
- Soubory jsou čteny/zapisovány sekvenčně.
  - Se soubory se pracuje jako s proudem dat — postupné načítání/zápis.
  - *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;

    `"r"` – čtení textového souboru, `"rb"` – čtení binárního souboru
  - `"w"` – režim zápisu;

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

    Kurzor je nastaven na konec souboru.
  - Můžeme kombinovat s dalšími režimy otevření soboru, např. `"r+"` pro otevření souboru pro čtení i zápis.

    viz `man fopen`

Testování – otevření/zavření souboru

- Testování otevření souboru.

```
1 char *fname = "file.txt";

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 }
```

lec07/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 z textového souboru

- `int fscanf(FILE *file, const char *format, ...);`
- Analogie formátovanému vstupu.

Pro vyplnění hodnot proměnných předáváme ukazatel.

- Vrací počet přečtených položek, například pro vstup

```
record 1 13.4
```

- příkaz: `int r = fscanf(f, "%s %d %lf\n", str, &i, &d);`
- vrátí v případě úspěšného čtení hodnotu 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);
```

lec07/file_scanf.c

Zápis do textového souboru

- Zápis po znaku – `int putc(int c, FILE *file);`
- Formátovaný výstup `int fprintf(FILE *file, const *format, ...);`

```
1  int main(int argc, char *argv[])
2  {
3     char *fname = argc > 1 ? argv[1] : "out.txt";
4     FILE *f;
5     if ((f = fopen(fname, "w")) == NULL) {
6        fprintf(stderr, "Error: Open file '%s'\n", fname);
7        return -1;
8     }
9     fprintf(f, "Program arguments argc: %d\n", argc);
10    for (int i = 0; i < argc; ++i) {
11       fprintf(f, "argv[%d]='%s'\n", i, argv[i]);
12    }
13    if (fclose(f) == EOF) {
14       fprintf(stderr, "Error: Close file '%s'\n", fname);
15       return -1;
16    }
17    return 0;
18 }
```

lec07/file_printf.c

- Identicky k `stderr` lze použít `stdout` a `stdin` pro čtení.

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

- Otevření souboru s příznakem `"b"`.

Vliv na řetězce, řídicí znaky např. `"\0"`, `\n` nebo `EOF` či `EOT` – Ctrl+Z.

- 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 prvků o velikosti `size`.
- Pokud došlo k chybě nebo detekci konce souboru, funkce vrací menší než očekávaný počet bajtů.

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` (chyba při otevření souboru)

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <string.h>

5  int main(int argc, char *argv[])
6  {
7     FILE *f = fopen("soubor.txt", "r");
8     if (f == NULL) {
9        int r = errno;
10       printf("Open file failed errno value %d\n", errno);
11       printf("String error '%s'\n", strerror(r));
12    }
13    return 0;
14 }
```

lec07/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 lze přidat podmínky na nutné hodnoty proměnných.
- Testovat můžeme makrem `assert(expr)` z knihovny `<assert.h>`.
- Pokud není `expr true`, program se ukončí a vypíše jméno zdrojového souboru a číslo řádku.
- Makro vloží příslušný kód do programu.

Relativně jednoduchý způsob indikace chyby, např. nevhodným argumentem funkce, posloupností volání, ale jako strukturální chyba programu, nikoliv hodnot definovaných za běhu.

- Vložení makra lze zabránit kompilací s definováním makra `NDEBUG`. `man assert`
- Makro `assert` má význam zejména při ladění program.

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

4  int main(int argc, char *argv[])
5  {
6     assert(argc > 1);
7     printf("program argc: %d\n", argc);
8     return 0;
9  }
```

lec07/assert.c

- Uvedený příklad pouze demonstruje použití `assert()`.
- Není vhodné testovat proměnné definované za běhu programu.
- Test `assert()` nebude do programu vložen při kompilaci s `NDEBUG`.

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
```

lec07/assert.c

Test alokace paměti a předčasné ukončení programu

- Dynamické přidělení paměti (`malloc`) je vhodné vždy kontrolovat.
- Pragmaticky můžeme očekávat typický průběh program bezchybný, včetně dynamické alokace paměti.
- Pak může být vhodné přidělení paměti kontrolovat, ale předčasně program ukončit v případě chyby.
- Nicméně stále je vhodné dát uživateli možnost dozvědět se, proč se program předčasně ukončil.
- Můžeme si tak napsat vlastní funkci (makro) `my_assert`, které ovšem nelze vyřadit kompilaci s `-NDEBUG`.

```
1  #ifndef __MY_ASSERT_H__
2  #define __MY_ASSERT_H__

4  #include <stdio.h>  //because of fprintf()
5  #include <stdlib.h> //because of exit() and malloc

7  #define my_assert(x, line, file) \
8     if (!(x)) {\
9        fprintf(stderr, "my_assert fail, line: %d,
       file %s\n", line, file);\
10       exit(-1);\
11    }
12 #endif
```

lec07/my_assert.h

```
1  #include "my_assert.h"
2  ...
3  int *a = malloc(SIZE * sizeof(*a));
4  my_assert(a, __LINE__, __FILE__);
5  ...
```

lec07/demo-my_assert.c

```
$ clang demo-my_assert.c -o demo-my_assert
$ ulimit -v 10240
$ ./demo-my_assert
my_assert fail, line: 14, file demo-my_assert.c
```

- Při chybě dokážeme ve zdrojovém souboru dohledat místo a důvod chyb.

Příkazy dlouhého skoku

- Příkaz `goto` je možné použít pouze v rámci jedné funkce.
- Knihovna `<setjmp.h>` definuje funkce `setjmp()` a `longjmp()` pro skoky mezi funkcemi.
- `setjmp()` uloží aktuální stav registrů procesoru a pokud funkce vrátí hodnotu různou od 0, došlo k volání `longjmp()`.
- Při volání `longjmp()` jsou hodnoty registrů procesoru obnoveny a program pokračuje od místa volání `setjmp()`.

Kombinaci `setjmp()` a `longjmp()` lze využít pro implementace ošetření výjimečných stavu podobně jako `try-catch` v jiných programovacích jazycích.

```c
#include <setjmp.h>
jmp_buf jb;
int compute(int x, int y);
void error_handler(void);
if (setjmp(jb) == 0) {
    r = compute(x, y);
    return 0;
} else {
    error_handler();
    return -1;
}
```

```c
int compute(int x, int y) {
    if (y == 0) {
        longjmp(jb, 1);
    } else {
        x = (x + y * 2);
        return (x / y);
    }
}
void error_handler(void)  {
    printf("Error\n");
}
```


Část II

Část 2 – Kódovací příklady

Kódovací příklad – Ne úplně čitelné použití goto

- Použití `goto` souvisí zejména s čitelností kódu, můžeje jej využít reakci na návratové hodnoty.

```c
#include <stdio.h>

int main(void)
{
    int c = getchar();
    if (c >= '0' && c <= '9') {
        goto print_digit;
    }
    goto print_error;
print_digit:
    printf("User input '%c' that has ASCII code value %d\n", c, c);
    goto leave;
print_error:
    fprintf(stderr, "ERROR: Expected input value is '0'--'9'\n");
leave:
    return 0;
}
```

Low Level Learning:why is it illegal to use "goto"?

- Uvedený příklad je funkční, ale načitelnosti úplně nepřidává.

V našem případě se plně obejdeme bez `goto`, obecně ale může být jeho použití užitečné, např. pokud testujeme postupně volání funkcí.

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.

```c
#define USERNAME_SIZE 8
#define NUMBER_DIGITS 4

struct record {
    char username[USERNAME_SIZE + 1];
    char number[NUMBER_DIGITS];
};

int main(void) {
    struct record records[] = {
        { .username = "user01" },
        { .username = "user02" },
        { .username = "admin" },
        { .username = "root" },
        { .username = '\0' } // null terminating array
    };

    fprintf(stderr, "DEBUG: size of struct %lu\n",
     sizeof(struct record));
    fprintf(stderr, "DEBUG: size of records %lu\n",
     sizeof(records));
```

- Položka `username` je o jeden znak delší, uložení `'\0'`.
- Položka `number` je zápis čísla o maximální hodnotě 9999 (počet řádků 4) v textové podobě (0000–9999).
- Velikost složeného typu je dána velikostí jednotlivých položek.
- Pole záznamů incializujeme se zarážkou (poslední prvek obsahuje prázdný řetězec v položce `username`).

```
$ clang struct.c && ./a.out
DEBUG: size of struct 13
DEBUG: 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 2/3

```c
#ifndef __RECORD_H__
#define __RECORD_H__

#define USERNAME_SIZE 8
#define NUMBER_DIGITS 4

struct record {
    char username[USERNAME_SIZE + 1];
    char number[NUMBER_DIGITS];
};

void print_record(const struct record * record);
void print(const struct record * const records);

unsigned int fill_numbers(struct record * const records);

#endif
```

record.h

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h> // strukturální testy

#include "record.h"
```

record.c

```c
void print_record(const struct record * record)
{
    if (record) {
        printf("Record\n");
        printf("|- username: \"%s\"\n", record->username);
        printf("|- number: ");
        print_chars(NUMBER_DIGITS, record->number);
        printf("\n\n");
        // printf("|- number:  %s\n\n"); // Vyzkoušejte!
    }
}

void print(const struct record * const records)
{
    struct record const *cur = records;
    while (cur && cur->username[0]) {
        print_record(cur);
        cur += 1; // pointer arithmetic
    }
}
```

record.c

- 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 3/3

```c
static unsigned short get_max_number(unsigned int digits)
{
    unsigned int number = 1;
    assert(sizeof(short) < sizeof(int));// 16 vs.  32?
    for (unsigned int i = 0; i < digits; ++i) {
        number *= 10;
    }
    assert(number <= USHRT_MAX); // short
    return number - 1;
}

static void fill_record_number(unsigned short n, int
    digits, char *number)
{ //number needs to be at least digits large
  for (int i = digits - 1; i >= 0; --i) {
      number[i] = (n % 10) + '0';
      n = n / 10;
  }
}
```

record.c

```c
static void print_chars(size_t digits, const char *number)
{ // number je ukazatel na konstantní hodnotu
    for (size_t i = 0; i < digits; ++i, ++number) {
        putchar(*number);
    }
}
```

record.c

```c
unsigned int fill_numbers(struct record * const records)
{
    struct record *cur = records;
    unsigned short n = 0;
    const short max_number = get_max_number(NUMBER_DIGITS);

    while (cur && cur->username[0]) {
        assert(n <= max_number); // Vyplňujeme programově
        fill_record_number(n, NUMBER_DIGITS, cur->number);
        n += 1;
        cur += 1;
    }
    return n;
}
```

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

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

#include "record.h"

int main(void) {
    int ret = EXIT_SUCCESS;

    struct record records[] = {
        { .username = "user01" },
        { .username = "user02" },
        { .username = "admin" },
        { .username = "root" },
        { .username = "jf" },
        { .username = '\0' } // null terminating array
    };

    fprintf(stderr, "DEBUG: size of struct %lu\n",
     sizeof(struct record));
    fprintf(stderr, "DEBUG: size of records %lu\n",
     sizeof(records));

    unsigned int n = fill_numbers(records); // number!
    print(records);
```

```c
    const char *fname = "records.dat";
    FILE *fd = fopen("records.dat", "w"); // uložení records
    if (fd) {
        size_t saved = fwrite(records, sizeof(struct record), n, fd);
        size_t size = n * sizeof(struct record);
        printf("DEBUG: saved bytes %lu out of %lu\n", saved * sizeof(struct
       record), size);
    } else {
        fprintf(stderr, "ERROR: Cannot open \"%s\" for writting\n", fname);
        ret = EXIT_FAILURE;
    }
    return ret;
};
```

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 2/4

- Obsah uloženého souboru můžeme zkusit přímo 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 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 30 33 6a 66 00 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 writting
1
```


Kódovací příklad – Načítání a ukládání struct 3/4

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

#include "record.h"

#define NUM_RECORDS 2

int main(void) {
    const char *fname = "records.dat";

    FILE *fd = fopen(fname, "r");
    if (!fd) {
        perror("Error open file");
        goto error;
    }
    struct record *records = malloc(NUM_RECORDS
     * sizeof(struct record));
    if (!records) {
        perror("Error malloc");
        fclose(fd); // Korektně soubor zavíráme.
        goto error;
    }
    ssize_t loaded;
```

load_struct.c

```c
    while ((loaded = fread(records, sizeof(struct record), NUM_RECORDS, fd))) {
        fprintf(stderr, "DEBUG: loaded records %lu\n", loaded);
        for (size_t i = 0; i < loaded; ++i) {
            print_record(&records[i]);
        }
    }
    free(records);
    fclose(fd);
    goto leave;
error:
    fprintf(stderr, "ERROR: during reading from the file \"%s\"\n", fname);
    return 1;
leave:
    return 0;
}
```

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
```

- Načtení bloku dat funkcí `fread()`.

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

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

#include "record.h"

#define NUM_RECORDS 2

enum { ERROR_FILE = 101, ERROR_MEM = 102};

void print_error(int error);

int main(void) {
   int ret = EXIT_SUCCESS;
   const char *fname = "records.dat";

   FILE *fd = fopen(fname, "r");
   if (!fd && (ret = ERROR_FILE)) { // ret!
      goto leave;
   }
   struct record *records = malloc(
     NUM_RECORDS * sizeof(struct record));
   if (!records && (ret = ERROR_MEM)) { // ret!
      goto leave;
   }
   ssize_t loaded;
   while ((loaded = fread(records, sizeof(struct record), NUM_RECORDS, fd))) {
      fprintf(stderr, "DEBUG: loaded records %lu\n", loaded);
      for (size_t i = 0; i < loaded; ++i) {
         print_record(&records[i]);
      }
   }
   free(records);

leave:
   if (fd) {
      fclose(fd);
   }
   print_error(ret);
   return ret;
}

void print_error(int error)
{
   switch (error) {
      case ERROR_FILE:
         fprintf(stderr, "ERROR: open file\n");
         break;
      case ERROR_MEM:
         fprintf(stderr, "ERROR: mem allocation\n");
         break;
   }
}
```


Pravidla překladu v gmake / make

- Pro řízení překladu použijeme pravidlový předpis programu GNU make. make nebo gmake
- Pravidla se zapisují do souboru Makefile.

  http://www.gnu.org/software/make/make.html
- Pravidla jsou deklarativní ve tvaru definice cíle, závislostí cíle a akce, která se má provést.

  cíl : **závislosti** *dvojtečka*

  akce *tabulátor*
- Cíl (podobně jako závislosti) může být například symbolické jméno nebo jméno souboru.

  tload.o : **tload.c**

  clang -c tload.c -o tload.o
- Předpis může být napsán velmi jednoduše.

  Například jako v uvedené ukázce.

  Flexibilita použití však spočívám především v použití zavedených proměnných, vnitřních proměnných a využití vzorů, neboť většina zdrojových souborů se překládá identicky.

Příklad – Makefile

- Definujeme pravidlo pro vytvoření souborů **.o** z **.c** z aktuálních souborů v pracovním adresáři s koncovkou .c.

```
CC:=ccache $(CC)
CFLAGS+=-O2

OBJS=$(patsubst %.c,%.o,$(wildcard *.c))

TARGET=tload

bin: $(TARGET)

$(OBJS): %.o: %.c
  $(CC) -c $< $(CFLAGS) $(CPPFLAGS) -o $@

$(TARGET): $(OBJS)
  $(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
  $(RM) $(OBJS) $(TARGET)
```

ccache

CC=clang make vs CC=gcc make

- **Při linkování záleží na pořadí souborů (knihoven)!**
- Jednou z výhod dobrých pravidel je možnost paralelního překladu nezávislých cílů . `make -j 4`

Část III

Část 3 – Zadání 7. domácího úkolu (HW7)

Zadání 7. domácího úkolu HW7

Téma: **Kruhová fronta v poli**

Povinné zadání: **3b**; Volitelné zadání: **4b**; Bonusové zadání: není

- **Motivace**: Práce s pamětí a datovými strukturami.
- **Cíl**: Prohloubit si znalost paměťové reprezentace a dynamické alokace paměti s uvolňováním.
- **Zadání**: https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw7
  - Implementace kruhové fronty s využitím předalokovaného pole pro vkládané prvky.
  - Volitelné zadání rozšiřuje úlohu o dynamické zvětšování a zmenšování kapacity fronty podle aktuálních požadavků na počet vkládaných/odebíraných prvků.
- Termín odevzdání: 27.04.2024, 23:59:59 PDT.

Shrnutí přednášky

Diskutovaná témata

- Standardní knihovny C
- Čtení a ukládání z/do souboru
- Ošetření chybových stavů - assert(), errno, setjmp(), longjmp()
- Kódovací příklady

- Příště: Spojové struktury.