Programování v jazyce C pro chemiky
(C2160)

11. Vizualizace sekvence proteinů

Práce s pamětí

- Každá proměnná musí mít přidělen (alokován) paměťový prostor odpovídající její velikosti
- Rozeznáváme dva základní typy alokace – statickou a dynamickou
- Statická alokace spočívá v přidělení paměti v okamžiku spuštění programu a používá se tehdy, kdy velikost požadované paměti je předem známa (již v okamžiku překladu programu)
- Paměť pro globální proměnné je vždy alokována staticky
- Paměť pro lokální proměnné je alokována až při vstupu do funkce a po opuštění funkce je opět uvolněna, jedná se tedy o automatickou dynamickou alokaci
- Paměť pro lokální proměnné je alokována v paměťové oblasti nazývané zásobník (angl. *stack*)
- Velikost zásobníku dostupná programům je často limitována (lze ji zobrazit např. příkazem `ulimit -s`), proto by paměťově náročné proměnné neměly být definovány jako lokální, ale jako globální nebo jim přidělíme dynamicky alokovanou paměť explicitně

Dynamická alokace paměti

- Paměť lze alokovat dynamicky na vyžádání kdykoliv za běhu programu pomocí knihovní funkce `malloc()`

```
#include <stdlib.h>
void *malloc(int size)
```

- Funkce `malloc()` přijímá parametr `size` specifikující velikost požadované paměti v bytech
- Návratovou hodnotou `malloc()` je ukazatel na začátek alokovaného paměťového prostoru
- Paměť je přidělena z paměťové oblasti nazývané halda (angl. *heap*), její velikost není zpravidla limitována (jen případně dostupnou fyzickou pamětí nebo pomocí `ulimit -d`), proto je tento způsob alokace vhodný pro paměťově náročné proměnné
- Pokud paměť nelze přidělit (např. při nedostatku paměti), je vrácena hodnota NULL
- K dynamicky alokované paměti přistupujeme pomocí ukazatelů
- Pro uvolnění paměti použijeme funkci `free()`

```
void free(void *ptr)
```

Dynamická alokace paměti – příklad

```
int mojefunkce(int n)
{
  int *numbers = NULL;        // Ukazatel na typ int

  numbers = malloc(n * sizeof(int)); //Alokujeme pamet pro n cisel typu int
  if (numbers == NULL)
  {
    printf("Neni k dispozici dostatek pameti\n");
    return 1;        // V pripade chyby vratime hodnotu 1
  }
  // Alokovanou pamet inicializujeme (napr. pomoci funkce memset)
  memset(numbers, 0, n * sizeof(int));


  // numbers je ukazatel a proto s nim muzeme pracovat i jako s polem

  for (int a = 0; a < n; a++)
    numbers[a] = a * 4;

  // Tady mohou byt nejake dalsi prikazy

  free(numbers);
  numbers = NULL;

  return 0;
}
```

Funkce s proměnným počtem parametrů

- V jazyce C lze vytvářet funkce jejichž počet parametrů není v jejich definici přesně specifikován, ale je určen až v okamžiku volání funkce
- Definice takové funkce obsahuje seznam povinných parametrů a za ním jsou uvedeny tři tečky reprezentující nespecifikovaný počet dalších parametrů

  Např: `int mojefunkce(int p1, int p2, ...)`
- Funkce s proměnným počtem parametrů jsou využívány hlavně v knihovních funkcích (`printf()`, `scanf()` atd.)
- Při běžném programování se jim pokud možno vyhýbáme, protože znepřehledňují program a mohou být zdrojem chyb (kompilátor nemůže kontrolovat počet a typy předaných argumentů)
- Pro získání argumentů uvnitř funkce se používá speciální mechanismus (podrobnosti viz. *man stdarg*)

Rekurzivní volání funkce

- Funkce může volat sama sebe – mluvíme o rekurzivním volání (zkráceně rekurzi)
- Rekurze se využívá především u problémů, které jsou ze své podstaty rekurzivní
- Implementace rekurze musí zajistit, že za žádných okolností nebude funkce volat sama sebe donekonečna
- Počet rekurzivních volání by neměl být příliš velký (rozumná hodnota je max. stovky volání)
- Použití rekurze není vhodné, pokud záleží na rychlosti programu, protože každé volání funkce je spojeno s určitými paměťovými a časovými nároky

Příklad na výpočet faktoriálu s využitím rekurze:
n! = n * (n-1) * (n-2) * … * 1 pro n > 0, n! = 1 pro n = 0

```
int faktorial(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return (n * faktorial(n - 1));     // n! = n * (n-1)!
  }
}
// Faktorial by bylo mozne spocitat efektivneji bez pouziti rekurze,
// u nekterych problemu je však rekurze nejefektivnejsim resenim.
```

Výčtový datový typ enum

- Výčtový datový typ enum se používá v případě, že existuje předem definovaná množina hodnot, kterých může proměnná nabývat
- Při definici výčtového typu je nutné všechny hodnoty uvést
- Proměnným daného výčtového typu lze přiřazovat pouze uvedené hodnoty
- Výčtové typy lze také využít pro specifikaci velikosti pole

```
typedef enum {
  PONDELI, UTERY, STREDA, CTVRTEK, PATEK, SOBOTA, NEDELE
} DNY;

DNY d;   // Definujeme vyctovou promennou

// Do vyctove promenne lze priradit pouze uvedene hodnoty
// PONDELI az NEDELE
d = UTERY;
if (d == UTERY)
  printf("utery");
```

```
// Ukazka vyuziti vyctoveho typu pro specifikaci velikosti pole
enum {MAX_ATOMS = 100000};

ATOM atoms[MAX_ATOMS];  // Definujeme pole atomu velikosti MAX_ATOMS
```

Chybové stavy

errno
char *strerror(int errnum);
void perror(const char *s);

- Pokud některá z funkcí pro práci se soubory (`fopen()`, `fclose()`) vrátí hodnotu indikující chybu, lze získat podrobnější informace o charakteru chyby
- Globální proměnná `errno` obsahuje číselný kód poslední chyby (vyžaduje hlavičkový soubor `#include <errno.h>`)
- Význam hodnot proměnné `errno` je popsán v manuálu (*man errno*)
- Funkce `strerror()` vrací text popisující chybu daného čísla (vyžaduje hlavičkový soubor `#include <string.h>`)
- Funkce `perror()` vypíše popis poslední chyby na standardní výstup (vyžaduje hlavičkový soubor `#include <stdio.h>`)
- Tyto funkce lze použít pro získání chyb z mnoha dalších funkcí standardní knihovny jazyka C, popis najdeme vždy v manuálu k příslušné funkci (*man jmeno_funkce*)

Chybové stavy při práci se soubory

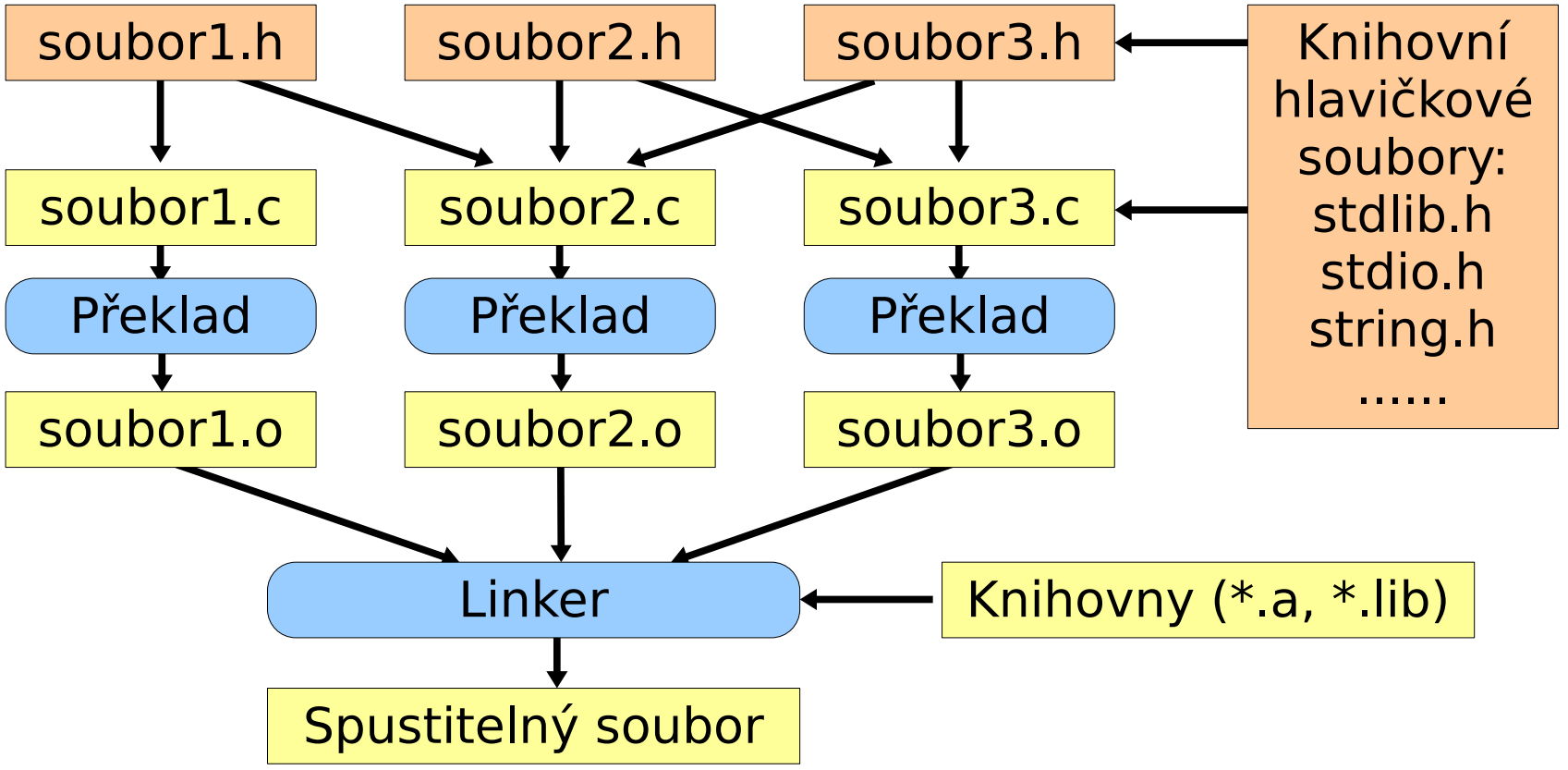
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

int read_pdb_file()
{
  FILE *f = NULL;
  f = fopen("crambin.pdb", "r");
  if (f == NULL) {
    printf("Cannot open input file %s \n", file_name);
    if (errno != 0) {
      printf("  Reason: %s\n", strerror(errno));
    }
    return 1;
  }
  // Zde bude nejaky kod zapisujici do souboru

  if (fclose(f) == EOF) {
    printf("Error closing the file %s\n", file_name);
    if (errno != 0) {
      printf("  Reason: %s\n", strerror(errno));
    }
    return 1;
  }
  return 0;
}
```

Překlad rozsáhlých programů

- U rozsáhlých programů nezapisujeme kód do jednoho souboru, ale rozdělíme jej do více souborů, každý soubor překládáme odděleně, překladač vždy vytvoří příslušný soubor s relativním kódem (*.o, *.obj)
- Soubory s relativním kódem použije linker pro vytvoření výsledného spustitelného souboru
- Výhodou tohoto odděleného překladu je časová úspora při vývoji programu – změna kódu v jednom souboru nevyžaduje kompilaci celého programu, ale jen příslušného souboru a následné linkování



Oddělený překlad programů

- Při děleném překladu překladačem *gcc* překládáme každý soubor zvlášť s použitím parametru -c, čímž vytvoříme příslušný soubor s relativním kódem (*.o), a tyto soubory nakonec slinkujeme:

```
gcc -c soubor1.c
gcc -c soubor2.c
gcc -o spustitelny_soubor soubor1.o soubor2.o
```

- Tyto činnosti zpravidla automatizujeme pomocí nástroje *make*

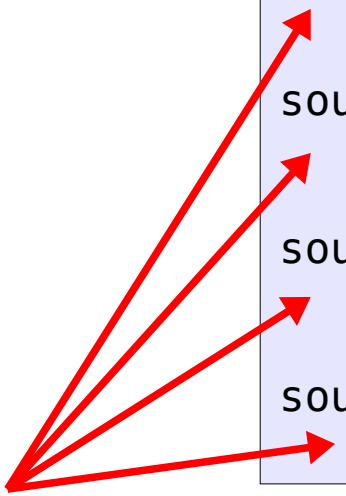
```
# Vytvori spustitelny program prog1, jehoz kod je zapsan
# v souborech soubor1.c, soubor2.c a soubor3.c a do nich jsou
# vlozeny hlavickve soubory soubor1.h, soubor2.h a soubor3.h

mujprogram : soubor1.o soubor2.o soubor3.o
	gcc -o mujprogram soubor1.o soubor2.o soubor3.o

soubor1.o : soubor1.c soubor1.h
	gcc -c soubor1.c

soubor2.o : soubor2.c soubor2.h
	gcc -c soubor2.c

soubor3.o : soubor3.c soubor3.h
	gcc -c soubor3.c
```



Zde musí být tabulátor,
ne mezery!

Učebnice jazyka C

- Herout, Pavel: **Učebnice jazyka C**.
  6. vyd., KOPP, 2009. 280 s. ISBN 978-80-7232-383-8. (cca 200 Kč)
- Herout, Pavel: **Učebnice jazyka C – 2. díl**.
  4. vyd., KOPP, 2008. 180 s. ISBN 978-80-7232-367-8. (cca 170 Kč)

- Václav Kadlec: **Učíme se programovat v jazyce C**.
  Computer Press, 2002. 294 s. ISBN 80-7226-715-9.
- Brian W. Kernighan, Dennis M. Ritchie: **Programovací jazyk C**.
  překlad angl. 2. vydání, Computer Press, 2006. 288 s.
  ISBN 80-251-0897-X. (cca 300 Kč)
- K. N. King: **C Programming: A Modern Approach**
  2 edition, W. W. Norton & Company, 2008. 832 p. ISBN 0393979504.
  (approx. 80 - 120 USD)

Barevné kódování aminokyselin

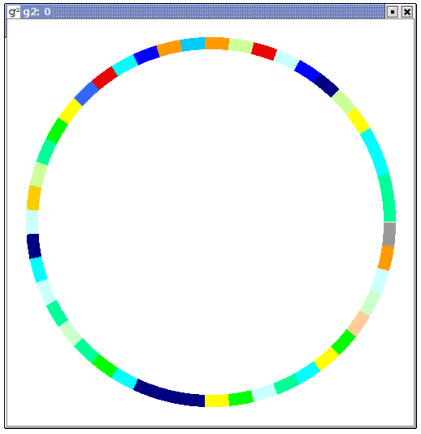
R - Arg - Arginine	[230, 6, 6]	[#E60606]
K - Lys - Lysine	[198, 66, 0]	[#C64200]
Q - Gln - Glutamine	[255, 102, 0]	[#FF6600]
N - Asn - Asparagine	[255, 153, 0]	[#FF9900]
E - Glu - Glutamic Acid	[255, 204, 0]	[#FFCC00]
D - Asp - Aspartic Acid	[255, 204, 153]	[#FFCC99]
H - His - Histidine	[255, 255, 153]	[#FFFF99]
P - Pro - Proline	[255, 255, 0]	[#FFFF00]
Y - Tyr - Tyrosine	[204, 255, 204]	[#CCFFCC]
W - Trp - Tryptophan	[204, 153, 255]	[#CC99FF]
S - Ser - Serine	[204, 255, 153]	[#CCFF99]
T - Thr - Threonine	[0, 255, 153]	[#00FF99]
G - Gly - Glycine	[0, 255, 0]	[#00FF00]
A - Ala - Alanine	[204, 255, 255]	[#CCFFFF]
M - Met - Methionine	[153, 204, 255]	[#99CCFF]
C - Cys - Cysteine	[0, 255, 255]	[#00FFFF]
F - Phe - Phenylalanine	[0, 204, 255]	[#00CCFF]
L - Leu - Leucine	[51, 102, 255]	[#3366FF]
V - Val - Valine	[0, 0, 255]	[#0000FF]
I - Ile - Isoleucine	[0, 0, 128]	[#000080]

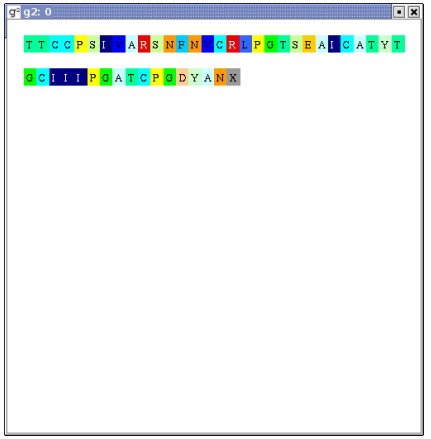
Reference: Ruiz, M. and Lefranc, M.-P., Currents in Computational Molecular Biology, Frontiers Science Series no.30, Universal Academy Press, Tokyo, pp 126-127 (2000)

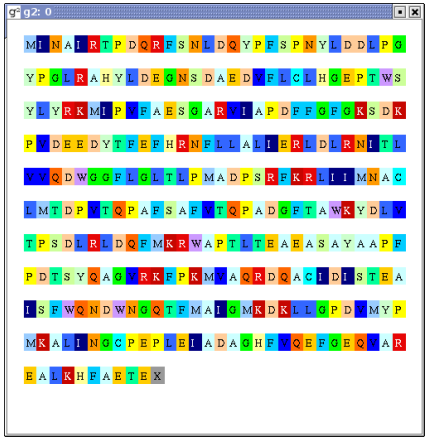
Úlohy

1. Do programu pro načítání a zápis PDB souboru (cv. 9, úloha 3) implementujte funkci, která pomocí knihovny g2 nakreslí kruhové schéma, kde každý barevný segment odpovídá jednomu residuu (viz první obrázek). Jméno PDB souboru bude specifikováno jako parametr na příkazovém řádku. Program otestujte se souborem *crambin_noal.pdb*.
   2 body

2. Do předchozího programu přidejte funkci, která zobrazí residua formou barevných segmentů, jak je zobrazeno na druhém obrázku. Segmenty budou popsány jednopísmennými zkratkami residuí. Na řádku bude max. 30 residuí. Dále v programu ošetřete chybové zprávy při otevírání a uzavírání souboru, tak aby došlo k přesnému nahlášení typu chyby. Otestujte se soubory *crambin_noal.pdb* a *2dhc_noH2O.pdb*. **2 body**







Úloha 1 – nápověda

Použijte následující strukturu pro získání jednopísmenných zkratek a barev RGB jednotlivých residuí (tento kód najdete také v souboru residue_types.c):

```
#define RESD_TYPES_COUNT 21

typedef struct
{
  char code3[4];        // three-letter code
  char code1;           // one-letter code
  double color_r;
  double color_g;
  double color_b;
} RESIDUE_TYPE;

RESIDUE_TYPE residue_types[RESD_TYPES_COUNT] =
{
  {"UNK", 'X', 153/255.0, 153/255.0, 153/255.0},
  {"ALA", 'A', 204/255.0, 255/255.0, 255/255.0},
  {"ARG", 'R', 230/255.0,   6/255.0,   6/255.0},
  {"ASN", 'N', 255/255.0, 153/255.0,   0/255.0},
  {"ASP", 'D', 255/255.0, 204/255.0, 153/255.0},
  {"CYS", 'C',   0/255.0, 255/255.0, 255/255.0},
  {"GLN", 'Q', 255/255.0, 102/255.0,   0/255.0},
  {"GLU", 'E', 255/255.0, 204/255.0,   0/255.0},
  {"GLY", 'G',   0/255.0, 255/255.0,   0/255.0},
  {"HIS", 'H', 255/255.0, 255/255.0, 153/255.0},
  {"ILE", 'I',   0/255.0,   0/255.0, 128/255.0},
  {"LEU", 'L',  51/255.0, 102/255.0, 255/255.0},
  {"LYS", 'K', 198/255.0,   6/255.0,   0/255.0},
  {"MET", 'M', 153/255.0, 204/255.0, 255/255.0},
  {"PHE", 'F',   0/255.0, 204/255.0, 255/255.0},
  {"PRO", 'P', 255/255.0, 255/255.0,   0/255.0},
  {"SER", 'S', 204/255.0, 255/255.0, 153/255.0},
  {"THR", 'T',   0/255.0, 255/255.0, 153/255.0},
  {"TRP", 'W', 204/255.0, 153/255.0, 255/255.0},
  {"TYR", 'Y', 204/255.0, 255/255.0, 204/255.0},
  {"VAL", 'V',   0/255.0,   0/255.0, 255/255.0}
};
```

Aby bylo možné snadno přistupovat k typům reziduí, přidejte do struktury RESIDUE proměnnou *residue_type* (typu int) a přiřaďte do ní odpovídající index v poli residue_types (tj. hodnotu 0 až 20, např. pro arginin 2, cystein 5 atd.). Toto proveďte např. na konci funkce sloužící k naplnění pole reziduí.