Přednáška 1

úterý 13. září 2022 13:13

počítač „počítá"

- výpočet realizovaný ALU
- číselné hodnoty uloženy v paměti počítače

```
:set number
:set relativenumber
:set autoindent
:set tabstop=4
:set shiftwidth=4
:set smarttab
:set softtabstop=4
:set mouse=a

call plug#begin()

Plug 'http://github.com/tpope/vim-surround' " Surrounding ysw)
Plug 'https://github.com/preservim/nerdtree' " NerdTree
Plug 'https://github.com/tpope/vim-commentary' " For Commenting gcc & gc
Plug 'https://github.com/vim-airline/vim-airline' " Status bar
Plug 'https://github.com/lifepillar/pgsql.vim' " PSQL Pluging needs :SQLSetType pgsql.vim
Plug 'https://github.com/ap/vim-css-color' " CSS Color Preview
Plug 'https://github.com/rafi/awesome-vim-colorschemes' " Retro Scheme
Plug 'https://github.com/neoclide/coc.nvim'  " Auto Completion
Plug 'https://github.com/ryanoasis/vim-devicons' " Developer Icons
Plug 'https://github.com/tc50cal/vim-terminal' " Vim Terminal
Plug 'https://github.com/preservim/tagbar' " Tagbar for code navigation
Plug 'https://github.com/terryma/vim-multiple-cursors' " CTRL + N for multiple cursors
Plug 'https://github.com/sheerun/vim-polyglot'
Plug 'https://github.com/lervag/vimtex'
Plug 'https://github.com/drewtempelmeyer/palenight.vim'
Plug 'https://github.com/rakr/vim-one'
Plug 'https://github.com/dkarter/bullets.vim'
Plug 'github/copilot.vim'
Plug 'https://github.com/glepnir/dashboard-nvim'
Plug 'https://github.com/junegunn/fzf.vim'
Plug 'https://github.com/junegunn/fzf'
Plug 'https://github.com/Mofiqul/dracula.nvim'
Plug 'https://github.com/navarasu/onedark.nvim'
Plug 'https://github.com/RRethy/nvim-base16'
```

```
Plug 'https://github.com/simnalamburt/vim-mundo'

let g:dashboard_default_executive = 'fzf'

set encoding=UTF-8

call plug#end()

nnoremap <C-f> :NERDTreeFocus<CR>
nnoremap <C-n> :NERDTree<CR>
nnoremap <C-t> :NERDTreeToggle<CR>
nnoremap <C-l> :call CocActionAsync('jumpDefinition')<CR>

nmap <F8> :TagbarToggle<CR>

:set completeopt-=preview " For No Previews

":colorscheme jellybeans

let g:NERDTreeDirArrowExpandable="+"
let g:NERDTreeDirArrowCollapsible="~"

" --- Just Some Notes ---
" :PlugClean :PlugInstall :UpdateRemotePlugins
"
" :CocInstall coc-python
" :CocInstall coc-clangd
" :CocInstall coc-snippets
" :CocCommand snippets.edit... FOR EACH FILE TYPE

" air-line
let g:airline_powerline_fonts = 1

if !exists('g:airline_symbols')
   let g:airline_symbols = {}
endif

let g:bullets_enabled_file_types = [
   \ 'markdown',
   \ 'text'
   \]

" airline symbols
let g:airline_left_sep = '
let g:airline_left_alt_sep = '
let g:airline_right_sep = '
let g:airline_right_alt_sep = '
let g:airline_symbols.branch = '
let g:airline_symbols.readonly = '
let g:airline_symbols.linenr = '

inoremap <expr> <Tab> pumvisible() ? coc#_select_confirm() : "<Tab>"

au BufRead,BufNewFile *.asm set filetype=nasm

:colorscheme base16-onedark
let g:airline_theme='onedark'
```

```
filetype plugin indent on

syntax enable

let g:vimtex_view_method = 'zathura'

let g:vimtex_view_general_viewer = 'okular'
let g:vimtex_view_general_options = '--unique file:@pdf\#src:@line@tex'
let g:vimtex_view_general_options_latexmk = '--unique'

let maplocalleader = ','
```

Přednáška 2

středa 5. října 2022 16:16

C

nízkoúrovňový programovací jazyk (systémový)
nechává vše na uživateli
má blízko k využití hardwarových zdrojů
klíčové je nakládání s pamětí

Je vhodné mít přehled plně pod kontrolou

i program co je syntakt. i sémant. správně
nemusí dělat co chceme

zdrojové soubory .c
hlavičkové soubory .h

zdrojové soubory překládány do binární podoby
překladačem (.o)

zdrojový kód se skládá z celých
a) klíčových slov
které pracují s proměnnými a S
b) čísly nebo znaky (literály)

b) číselnými hodnotami (literály)

Statická sémantika / plná sémantika

escape sekvence '\'

\o osmičková číslice

\x hex. číslice

Identifikátory

první znak není číslice

case sensitive

každý spustitelný program musí mít void main()

main(int argc, char *argv)

pozn. Faigl chce prototypy funkce

preprocesor → překladač → linker

little endian - big endian

literály — dekadický
šestnáctkový
osmičkový
unsigned
long
unsigned long

6 typů literálů:
celočíselné
znakové
řetězcové
výčtové

Výrazy:
může obsahovat:
literály
proměnné
konstanty
unární a binární operátory
volání operátory

Závorky

Binární operátory
- aritmetické
- relační
- logické
- operátor přiřazení

Unární
- + a —
- logické doplněk !
- ⋮

Ternární operátor

HW04

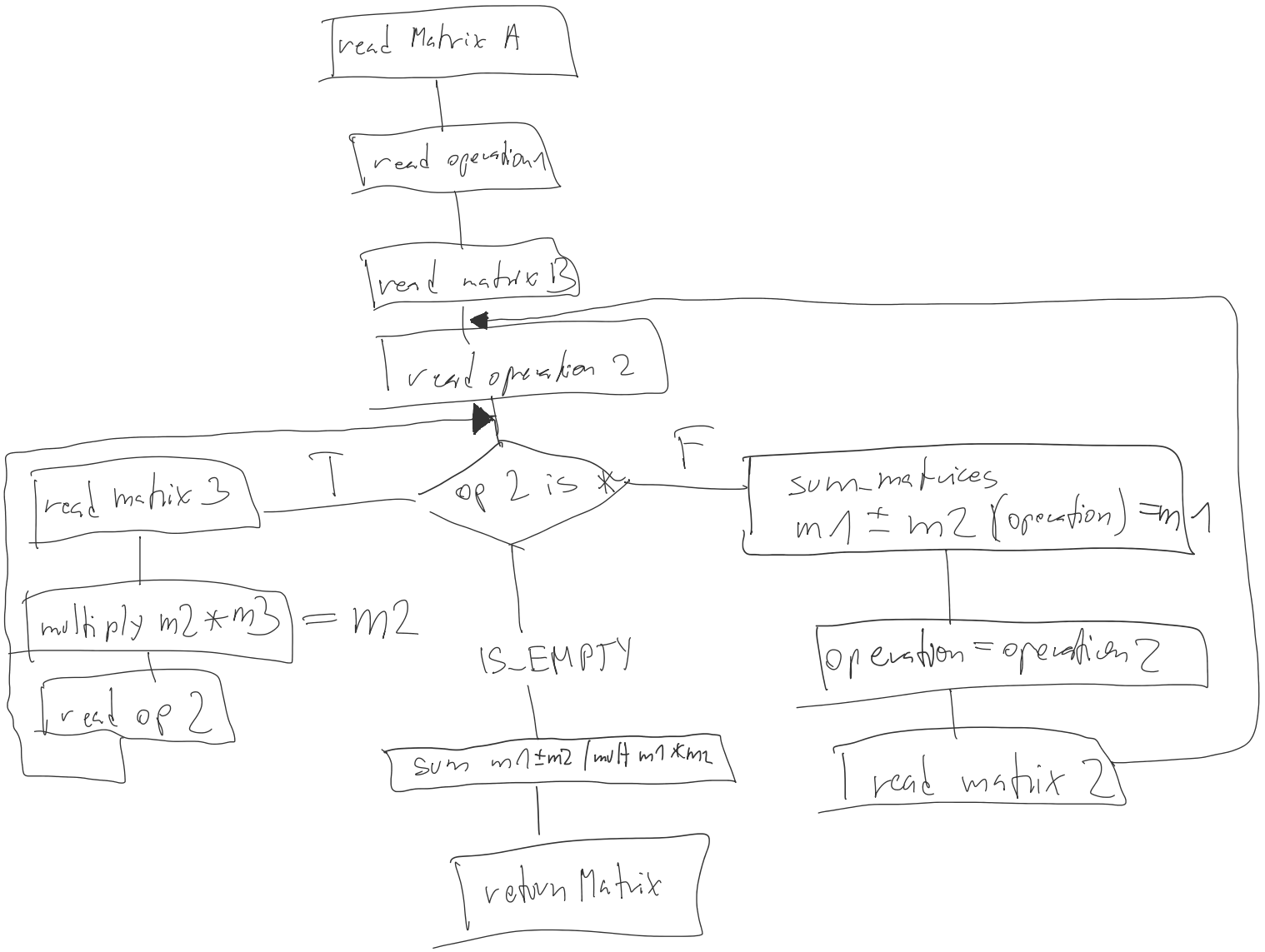
neděle 23. října 2022 22:55

Prime factorization

Eratosthenes sieve arr [1M] — size 1MB

HW06

úterý 22. listopadu 2022 9:44



Otázky PRP

pátek 13. ledna 2023 15:07

ZPRACOVANÉ OTÁZKY NA PRP

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA ELEKTROTECHNICKÁ

Poslední aktualizace: 8. 1. 2021

Úvod

Tento dokument vznikl jako učební pomůcka v rámci přípravy na zkoušku z předmětu PRP (Procedurální programování). Důvodem vzniku bylo velké množství informací z následujících zdrojů, které jsem si potřeboval centralizovat v jednom dokumentu, který by byl také snadno tisknutelný. Veškeré informace a obrázky zde uvedené pocházejí z následujících zdrojů:

- PRP Ones & Zeroes. Ones & Zeroes [online]. [cit. 2020-12-23]. Dostupné z: http://onesandzeroes.cz/prp.html?fbclid=IwAR2Hg4ub2rOrCXIbqsU538fGJXU9EUooSE3v3OnTi44UxmopO6p0p__HBLW0
- Zpracované otázky od zlatíček z KYRu [online]. [cit. 2020-12-23]. Dostupné z: https://www.docdroid.net/hhwG5rq/zpracovane-otazky-od-zlaticek-z-kyru-1-pdf#page=2
- Prezentace předmětu PRP jejichž autorem je prof. Ing. Jan Faigl, Ph.D. [online]. [cit. 2020-12-23]. Dostupné z: https://cw.fel.cvut.cz/b201/courses/b0b36prp/lectures/start

Je možné, že dokument obsahuje faktické či pravopisné chyby. Tyto chyby mohly vzniknout již v původních dokumentech a je možné, že jsem si jich nevšiml. V takovém případě se čtenáři omlouvám a doufám, že přínos textu převáží nad případnými nedostatky.

Uvedené webové stránky také obsahují pravděpodobně užitečné informace ohledně implementační zkoušky, které jsem sem nezahrnul.

Rezervoár otázek pokrývající znalost jazyka C

1) **c#001: Proč se programy v C rozdělují do hlavičkových souborů (.h) a zdrojových souborů (.c)?**
   - kvůli přehlednosti - nemusím v každém souboru .c definovat jednu funkci stále znovu, stačí její deklaraci uvést v includovaném souboru .h
   - kvůli kompilátoru (je třeba dopředu znát deklarace funkcí a jejich návratovou hodnotu)
   - "zapouzdření" - ostatní programátoři nemusí vidět přímo moje definice funkcí, stačí jim jen použít správné deklarace, tedy linkovat příslušný soubor .h

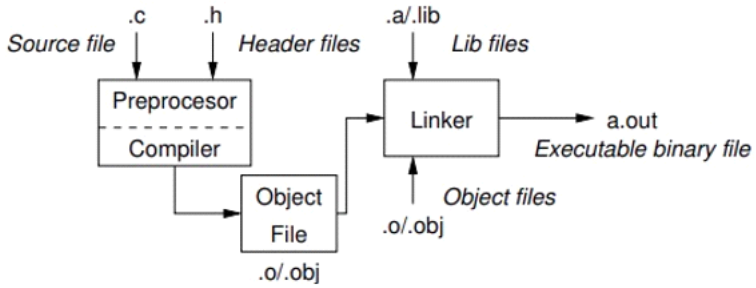
2) **c#002: Jaký význam má hlavičkový soubor zdrojových souborů programu v C?**
   - Jsou v něm funkce, jejich popisy a proměnné, které se sdílejí napříč soubory a není tak potřeba je všude kopírovat.
   - hlavičkový soubor je includovaný v main (stačí znát pouze použití, ne jejich přímou implementaci)

3) **c#003: Jak probíhá překlad a linkování (sestavení) programu v C?**
   - Preprocesor (dosazuje kusy kódu za includy, # makra, vyhazuje komentáře, spojuje řádky dohromady když je tam „/"). **Umožňuje definovat makra a tím přizpůsobit překlad aplikace kompilačnímu prostředí. Výstupem je textový („zdrojový") soubor**
   - Kompilace (překlad) (kompiluje – převede programátorem napsaný zdrojový kód do „.o" (binární a spustitelné) podoby. To jsou soubory, které již obsahují binární kód. Tento kód ale není výsledný program, protože nemá vyřešené závislosti na jiné části programu (například volání funkce, která je v jiném .c souboru). Je třeba znát deklarace, vkládá jen relativní adresy. **Překládá zdrojový (textový) soubor do strojově čitelné (a spustitelné) podoby.**
   - Linkování (vkládají se absolutní hodnoty adres – proměnné a funkce, výsledkem je spustitelný program). Při kompilování do objektových souborů kompilátor neví, zda volaná funkce existuje a kde je. Použití správných adres v paměti řeší linker. **Sestavuje program z objektových souborů do podoby výsledné aplikace.**

4) **c#004: Vysvětlete rozdíl mezi překladem zdrojových souborů a linkováním programu?**
   - Při překladu není vytvořený spustitelný soubor. To zajistí až linker při linkování.
   - Překlad vkládá jen relativní adresy, kdežto linkování vkládá již absolutní adresy, viz 3.

5) **c#005: Co je to preprocesor a jaká je jeho funkce při překladu zdrojového souboru v jazyce C?**
   - Preprocesor interpretuje jednoduché direktivy pro vložení zdrojového kódu z jiného souboru (#include), definice maker (#define) a podmíněné vložení kódu (#if).
   - Při překladu spojí řádky do jednoho, provede tokenizaci, tedy zalomí výsledek do posloupnosti tzv. tokenů preprocesoru a bílých znaků, pak nahradí komentáře bílým znakem, přepíše makra a provede direktivy.

6) c#006: Popište proces vytvoření spustitelného programu ze zdrojových souborů jazyka C.



- 1. Preprocessing - preprocesor:
  - Textové předzpracování preprocesorem, který má vlastní makro jazyk (příkazy uvozeny znakem #)
  - odstraní komentáře
  - inkluduje kódy hlavičkových souborů
  - interpetuje makra
  - výstupem je textový/zdrojový soubor
- 2. Kompilace (překlad) - kompilátor (překladač):
  - Překládá zdrojový (textový) soubor do strojově čitelné (a spustitelné) podoby (objektový kód).
  - vytvoří několik .o objektových souborů
- 3. Linking - linker:
  - Sestavuje program z objektových souborů do podoby výsledné aplikace.
  - připojí k němu použité knihovny
  - vytvoří .out spustitelný soubor

7) c#007: Jaké znáte překladače jazyka C?

- gcc, clang, msvc

8) c#008: Jak zajistíme, že se hlavičkový soubor programu v C nevloží při překladu vícekrát?

- pomocí tzv. Include (header) guard (hlavičkový strážce)

```
#ifndef MYCLASS_H_
#define MYCLASS_H_
//kod
#endif
```

- Když preprocesor skenuje hlavičkový soubor, narazí na #ifndef a pokračuje pouze pokud není MYCLASS_H_ už definovaná. Pokud ano, skočí na #endif.

9) **c#009: Jak zajistíte možnost ovlivnit výslednou podobu programu při překladu? Např. Velikost bufferu definovanou symbolickou konstantou?**

- Přepínačem -Dbufsize = 420 a přidáním include guard (vloží výchozí hodnotu, pokud ji nedostane z přepínače):

  ```
  #ifndef bufsize
  #define bufsize 420
  #endif
  ```

10) **c#010: Jak při překladu programu kompilátorem GCC nebo Clang rozšíříme seznam prohledávaných adresářů s hlavičkovými soubory?**

- Přepínačem -Idir [options] [source files] [object files] [-o output file].

11) **c#011: Záleží u kompilace programu kompilátorem GCC nebo Clang při specifikaci adresářů s hlavičkovými soubory na jejich pořadí?**

- Ano. (Pokud bychom na příklad linkovali více knihoven)

12) **c#012: Co způsobí definování makra preprocesoru NDEBUG v souvislosti s používáním funkce assert?**

- Dojde k tomu, že si přestane všímat funkce assert, tudíž program nespadne v případě nesplnění podmínky ve výrazu assert(condition). (po zadefinování NDEBUG assert() nefunguje)

13) **c#013: Jaký tvar má hlavní funkce programu v C, která se spustí při spuštění programu v prostředí s operačním systémem?**

- `int main(){//viz dále}` nebo
- `int main(int argc, char *argv[]){//viz dále}` nebo
-
  ```
  main (int argc, char **argv){ // code
  return 0; // Indicates that everything went well.
  }
  ```

14) **c#014: Jakou návratovou hodnotou programu v C indikujete úspěšné vykonání a ukončení programu? Proč zvolíte právě tuto hodnotu?**

- 0. Jedná se o konvenci (nula jen jedna, 0 chyb). (Případně EXIT_SUCCESS z stdlib.h.)

15) **c#015: Jak předáváme parametry programu implementovanému v jazyce C?**

- Pokud je main ve tvaru `int main(int argc, char *argv[]) { /* ... */ }`,
- tak přes konzoli ./program argument1 argument2, kde
- argv[0] bude ./program
- argv[1] bude argument1
- argv[2] bude argument2
- a argc bude počet argumentů, tedy 3.

16) **c#016: Existuje nějaká jiná možnost jak předat uživatelské parametry programu jinak než jako argument programu?**

- Ano, můžeme použít například standardní vstup (pomocí scanf a podobných z stdio.h), config file, nebo systémové proměnné.

17) **c#017: Jaký je rozdíl mezi staticky a dynamicky linkovaným programem implementovaným v jazyce C?**

- Při statickém linkování linker kopíruje všechny použité funkce z knihoven do spustitelného souboru, vlivem čehož zabírá soubor více místa.
- Při dynamickém linkování nejsou knihovny přímo v programu, ale načítají se až když jsou potřeba a to většinou ze systémové složky lib. Šetří se tak místo.
- Staticky slinkovaný program je lépe přenosný, stačí přenést jeden soubor. U dynamicky slinkovaného programu je nutno zajistit, aby na počítači byly nainstalovány správné verze požadovaných dynamických knihoven nebo při přenosu přidat správné verze jako další soubory. Toto pravidlo platí rekurzivně (dynamické knihovny mohou vyžadovat další dynamické knihovny).

18) **c#018: Linkují ve výchozím nastavení překladače GCC nebo Clang statické nebo dynamické binární spustitelné soubory?**

- Standardní knihovny se linkují dynamicky a mé vlastní knihovny staticky.

19) **c#019: Jak vkládáme do zdrojového souboru programu v C hlavičkové soubory jiných modulů nebo knihoven?**

- Preprocesorovým příkazem #include <soubor.h> pro systémové .h soubory, #include "soubor.h" pro naše. Pokud se soubor nenachází ve složce s main.c je potřeba v přepínači -I uvést adresu.

20) **c#020: Jaký rozdíl mezi použitím #include <soubor.h> a #include "soubor.h"?**

- U #include <soubor.h> preprocesor hledá hlavičkový soubor ve standardních systémových složkách.
- U #include "soubor.h" preprocesor hledá hlavičkový soubor ve složce, ve které je uložený zdrojový soubor (a také v systémových složkách).

21) **c#021: Popište rozdíl mezi deklarací a definicí funkce v jazyce C?**

- Při deklaraci říkáme, že danou s danými argumenty a návratovou hodnotou použijeme. Při deklaraci se nealokuje paměť. Např. int foo(int a, int b);.
- Při definici už píšeme celou funkci a alokuje se pro ni paměť. Např.:

```
int foo(int a, int b) {
    return a + b;
}
```

22) **c#022: Jak jsou předávány parametry funkci v jazyce C?**

- a) Hodnotou (pass by value)- hodnota proměnné se zkopíruje na zásobník. Jakékoliv změny tohoto lokálního parametru nemají vliv na hodnotu proměnné ve volající funkci.
- b) Odkazem (pass by reference) - hodnota ukazatele, tedy adresa, se zkopíruje na zásobník. Funkce může měnit hodnotu na dané adrese. Technicky se nejedná o pravé předání odkazem, protože je C jazyk předávání hodnotou (pass-by-value language). Pointer je proměnná typu pointer (integer s adresou).

23) c#023: Co je to literál a co tímto pojmem označujeme?

- Konstanta použitá uvnitř programu. Např. int a = 10; (literál 10)

24) c#024: Jak lze v jazyce C realizovat předání parametru funkci odkazem?

- Musíme funkci předat adresu, ne hodnotu:

```
#include <stdio.h>
int squareInt(int *num) {
   return (*num) * (*num);
}
int main() {
   int x = 4;
   printf("%d\n", squareInt(&x));
}
```

- Jde to, ale nejedná se o předání odkazem v pravém slova smyslu, protože C je striktně pass by value. Pouze se to „emuluje“ pomocí pointerů.

25) c#025: Jak lze v jazyce C omezit viditelnost funkce pouze v rámci jednoho modulu (souboru .c)?

- Klíčovým slovem static. (např. static void function(){})

26) c#026: Je možné v jazyce C volat funkci ze sebe sama (rekurze)?

- Ano. Např.:

```
unsigned int factorial(unsigned int n) {
   if (n == 0)
      return 1;
   return n * factorial(n - 1);
}
```

27) c#027: Při volání funkce v jazyce C jsou předávány argumenty funkce, které se stanou lokálními proměnnými. V jaké části paměti jsou tyto lokální proměnné při běhu programu uloženy?

- Na zásobníku (stacku) tj. část paměti pro staticky alokovanou paměť viz dále

28) c#028: Jaké znáte kategorie proměnných z hlediska jejich umístění v paměti?

- Lokální proměnné, argumenty funkcí a návratové hodnoty: na zásobníku (stack).

- Dynamicky alokované proměnné: na haldě (heap).
- Globální a statické proměnné: na statických datech (static data).
- Literály: v literálech.
- Strojové instrukce: v instrukcích (programu).

29) c#029: Je součástí jazyka C přímá podpora (tj. klíčové/á slovo jazyka) dynamická alokace paměti?

- Ne přímo v jazyce C, ale v knihovně stdlib.h, která je součástí standardních knihoven jazyka C.

30) c#030: Jak v jazyce C dynamicky alokovat paměť za běhu programu?

- Pomocí funkce malloc() nebo calloc().
- Například:

  ptr = (int *)malloc(100 * sizeof(float));

  nebo ptr = (float *)calloc(25, sizeof(float));

31) c#031: Je v jazyce C nutné uvolňovat dynamicky alokovanou paměť? Pokud ano, jak to uděláte?

- U moderních operačních systémů teoreticky ne. U starších by však mohlo docházet ke znatelným paměťovým ztrátám. Provádí se takto free(pointer);.

32) c#032: Jak zjistíme velikost reprezentace datových typů v jazyce C?

- Funkcí sizeof():

  printf("%lu\n", sizeof(int));.

33) c#033: Je vždy v jazyce C velikost proměnné typu int 32 bitů?

- Ne, záleží na kompilátoru a architektuře počítače.

34) c#034: Kdy je velikost ukazatele v jazyce C 32-bitů a kdy 64-bitů?

- To závisí na architektuře počítače. 32-bitové systémy mají 32-bitové ukazatele, 64-bitové 64-bitové.

35) c#035: Jak funguje modifikátor static při použití v definici lokální proměnné funkce v jazyce C?

- I po ukončení funkce zůstane proměnná na poslední hodnotě uložená na static data a je viděna pouze v souboru, ve kterém je deklarována.

36) c#036: Jak funguje modifikátor extern při definici globální proměnné v jazyce C?

- Extern proměnná je vidět ve všech modulech a při napsání se nedefinuje, ale pouze deklaruje.

37) c#037: Jaké znáte základní znaménkové celočíselné typy v jazyce C?

- Například: int, long, long long, short, signed char. (obsahují informaci o znaménku)

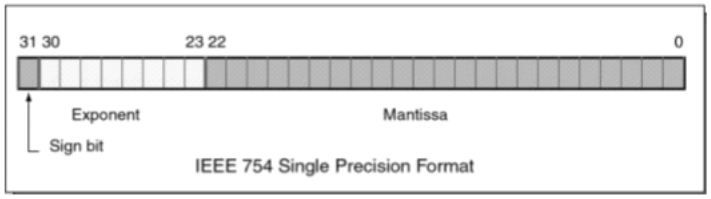
38) c#038: Rozlišuje jazyk C znaménkové a neznaménkové celočíselné typy? Pokud ano, co z toho plyne?

- Ano, rozlišuje (unsigned vs signed). Vyplývá z toho, že mají neznaménkové typy dvakrát větší rozsah v kladných číslech.

39) c#039: Jaké znáte neceločíselné typy v jazyce C? Jaká je jejich vnitřní reprezentace (velikost)?

- float (32 bit), double (64 bit)
- Jsou representovány jako mantisa * základ^exponent, kde 0,1 ≤ mantisa < 1.

$$x = \text{mantisa} \cdot \text{základ}^{\text{exponent}}$$



40) c#040: Jsou v jazyce C definovány rozsahy neceločíselných typů?

- Nejsou, jediná podmínka je, že double musí být 2* větší, než float. Nicméně na většině systémů je podle IEEE 754 float 32 bitů a double 64 bitů.

41) c#041: Jsou v jazyce C definovány rozsahy celočíselných typů?

- Ano i ne, liší se podle architektury. Obecně jsou dána pouze určitá pravidla: short <= int, int <= long atd.

42) c#042: Jak v C zajistíte proměnné celočíselného typu s konkrétním požadovaným rozsahem (tj. velikostí datové reprezentace)?

- Použitím proměnných z knihovny <stdint.h>, jako je uint16_t nebo uint32_t.

43) c#043: Je součástí jazyka C typ logické hodnoty „true/false"? Pokud ano, jak se používá? Pokud ne, jak jej definujete?

- Není. Je potřeba includovat <stdbool.h>. Používá se standardně bool x = true.
- Od c99 lze použít „_Bool“ i bez includování <stdbool.h> (_Bool a bool mají po includování stejný význam)

44) c#044: Jak v C definujete ukazatel na proměnnou, např. typu int?

- Ukazatel ptr na proměnnou x se definuje jako int *ptr = &x. (kde např. int x = 5)

45) c#045: Jaké jsou v C omezení pro názvy proměnných a funkcí?

- 1. Název nesmí začínat číslem.
- 2. Název může obsahovat pouze velká a malá písmena anglické abecedy, čísla 0 - 9 a podtržítko.
- 3. Název se nesmí shodovat s klíčovým slovem.
- 4. Omezená délka

46) c#046: Jaké znáte escape sekvence používané v C pro řídicí znaky?

- \n (new line, nový řádek)
- \r (carriage return)
- \t (horizontal tab)
- \v (vertical tab)
- \b (backspace)
- \f (form feed)
- \a (alert)
- \0 (NULL)

47) c#047: Jak jsou v C reprezentovány textové řetězce?

- Jako pole charů, které má jako poslední znak \0. (Pole s řetězcem tedy musí být vždy o 1 delší, než vkládaný text!)

48) c#048: Jak v C zapisujeme identifikátory (jména funkcí a proměnných)?

- Vše je case sensitive!
- 1. Používáme pouze velká a mála písmena anglické abecedy, čísla 0 - 9 a podtržítka.
- 2. Konstanty a makra v preprocesorových direktivách píšeme velkými písmeny: MAX_BUFFER_SIZE.
- 3. Držíme se pouze jednoho stylu zápisu. Např. camel case: int isPrime(int num) {...}, nebo snake case: int is_prime(int num) {...}.
- 4. Používáme pokud možno sebepopisné názvy, ze kterých je jasný účel funkce jako na příklad void fill_array_randomly(int *array) {...}). Případně dovysvětlíme účel funkce doprovodným komentářem.
- 5. Struktury a typedef píšeme v CamelCase

49) c#049: Jakými dvěma způsoby lze v C vytvářet konstanty?

- a) Definovat je jako makro (#define MAX_BUFFER_SIZE 100).
- b) Použitím klíčového slova const (const int MAX_BUFFER_SIZE = 100;).

50) c#050: Popište výčtový typ jazyka C. Uveďte vlastnosti, které považujete za důležité?

- Enum. Používá se na příklad následovně:

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
int main() {
    enum week today;
    today = Wednesday;
```

```
    printf("Today is the %d-th day of the week", today + 1);
    // Vypise: Today is the 4-th day of the week.
    return 0;
}
```

- Důležité vlastnosti enum:
  - 1. Konstanty výčtového typu jsou vždy celočíselné.
  - 2. Pokud konstantě nepřiřadíme hodnotu, získá ji automaticky na základě pořadí zápisu (indexováno od nuly).
  - 3. Enum slouží k ulehčení práce a větší přehlednosti. Lze jej použít při definici kalendáře, dní v týdnu atd.

51) c#051: Jaké znáte logické operátory jazyka C? Jak se zapisují?

- && - logický AND
- || - logický OR
- ! logická negace

52) c#052: Jaké znáte bitové operátory jazyka C? Jak se zapisují?

- x >> n je bitový posuv o n posic doprava.
- x << n je bitový posuv o n posic doleva.
- x & y s každým párem bitů provede logickou operaci AND.
- x | y s každým párem bitů provede logickou operaci OR.
- x ^ y s každým párem bitů provede logickou operaci XOR.
- ~x s každým bitem provede operaci NOT.

53) c#053: Jak v C realizujete dělení a násobení dvěma s využitím operátorů bitového posunu?

- Násobení bitovým posunem doleva o jednu posici: x << 1 = x * 2.
- Dělení bitovým posunem doprava o jednu posici: x >> 1 = x / 2.

54) c#054: Jak v C definujete složený typ (struct)?

```
struct my_struct_s {
    // code
};
```

55) c#055: Jak zavedeme nový typ struktury, např. pojmenovaný my__struct__s.

```
typedef struct {
    // code
} my_struct_s;
```

56) c#056: Kdy nemusíme psát před identifikátor určující jméno/typ struktury klíčové slovo struct?

- Když definujeme struct pomocí typedef.

57) c#057: Co je v jazyce C pointerová (ukazatelová) aritmetika a jak se používá?

- Pointerová aritmetika je soubor operací s ukazateli.
- 1. Součet či rozdíl ukazatele a celého nezáporného čísla: p + m posune adresu, na kterou ukazuje p o m * sizeof(typ p). Lze také použít „++“ a „--“ (pokud je např. Pointer na int, tak ptr++ = ptr+4). Například:

```
#include  <stdio.h>

int main() {
   int arr[] = {100, 110, 230, 440};
   int *ptr = &arr[0];
   printf("%d\n", *(ptr + 1)); // 110
   return 0;
}
```

- 2. Rozdíl dvou ukazatelů: při rozdílu ukazatelů p1 a p2 získáme počet prvků ležících mezi nimi. Například:

```
#include <stdio.h>

int main() {
   int arr[] = {100, 110, 230, 440};
   int *ptr1 = &arr[0];
   int *ptr2 = &arr[2];
   printf("%d\n", (ptr2 - ptr1)); // 2
   return 0;
}
```

- 3. Porovnávání ukazatelů: jedná se o klasické porovnávání, ne však hodnot ale adres. Například:

```
#include <stdio.h>

int main() {
   int arr[] = {100, 110, 230, 440};
   int *ptr1 = &arr[0];
```

```
    int *ptr2 = &arr[2];
    printf("%d\n", (ptr1 > ptr2)); // 0
    printf("%d\n", (ptr2 > ptr1)); // 1
    return 0;
}
```

58) c#058: Jak se v C liší proměnná typu ukazatel a typu pole[] (VLA - pole variabilní délky)?

- sizeof(ptr) vrací velikost adresy v paměti, zatímco sizeof(array) vrací velikost v paměti celého pole.
- Pole je ukazatel na první položku pole.

59) c#059: Jak v C přistupujeme k datovým položkám složeného typu (struct)?

- Přímým přístupem tečkou:

```
struct st {
    int i;
};
int main () {
    struct st my_struct = {.i = 6};
    printf("%d\n", my_struct.i); // 6
    return 0;
}
```

- nebo ukazatelem na struct šipkou ->

```
struct st {
    int i;
};
int main() {
    struct st *my_struct = malloc(sizeof(*my_struct));
    my_struct->i = 6;
    printf("%d\n", my_struct->i); // 6
}
```

60) c#060: Uveďte příklad přístupu k položkám proměnné složeného typu (struct) a proměnné typu ukazatel na složený typ.

- Poznámka: struct můžeme definovat jako proměnnou typu struct: struct st my_struct nebo jako ukazatel na struct: struct st *my_struct = malloc(sizeof *my_struct);
- Přístup k položkám proměnné my_struct typu struct: my_struct.promenna

- Přístup k položkám proměnné my_struct typu ukazatel na struct: my_struct->promenna

61) c#061: Jaký v C rozdíl mezi typy struct a union?

- 1. U structu je alokována paměť pro každý člen zvlášť (velikost struct je tedy >= součtu velikostí všech členů), zatímco u unionu je alokována paměť pro jeho největší člen (velikost je tedy rovna velikosti největšího členu.
- 2. U structu změna jedné hodnoty neovlivní ostatní, zatímco u unionu ano.
- U structu může být několik členů inicializováno najednou, zatímco u unionu jen první člen.

62) c#062: Stručně popište typ union používaný v jazyce C.

- Union je datový typ, který umožňuje ukládat různé datové typy na stejném místě v paměti. Proto když změníme hodnotu jednoho členu, změní se hodnota všech ostatních.
- Uniony poskytují účinný způsob použití stejné paměťové lokace pro víceúčelové aplikace.
- Kompilátor alokuje paměť dle člena s největší velikostí, tudíž je velikost unionu rovna velikosti největší členu, a proto šetří paměť.

```
union Data {
   int i;
   float f;
   char str[20];
};
```

63) c#063: Jak se v jazyce C používá operátor přetypování?

- Před proměnnou či výraz napíšeme do závorky typ, na který ji / jej chceme přetypovat. Například: (int)(x + y).
- Nelze však přetypovat cokoliv (např. řetězec na číslo).

64) c#064: Co v C reprezentuje typ void?

- a) Návratový datový typ - void funkce nic nevrací.
- b) Jako parametr - int fce(void) {...} indikuje, že funkce žádný parametr nepřijímá.

65) c#065: Co v C reprezentuje typ void*?

- void* deklaruje ukazatel bez předem definovaného typu - void *ptr = &a; může mít a jakýkoliv datový typ.
- Ukazatel void* lze poté přetypovat na jiný (umožňuje psát obecnější funkce).

66) c#066: Jak v C realizujete opuštění dvou nebo více vnořených cyklů z nejvnitřnějšího cyklu?

- Použitím kontrolní proměnné.
- Použitím *goto* např.:

```
if (i == 1 && j == 3) {
    goto exit_label;
```

```
}
exit_label:
```

- Použitím kombinací více breaků, které se postupně zavolají po splnění nějaké podmínky.
- Případně použijeme return pro úplné opuštění celé funkce.

67) c#067: Co v C reprezentuje identifikátor NULL?

- Ukazatelovou konstantu, která udává, že pointer na nic neukazuje. Je u ní zaručeno, že neukazuje na žádný reálný objekt.

68) c#068: Jak nastavíte proměnnou typu ukazatel na prokazatelně neplatnou hodnotu?

- Necháme ho ukazovat na NULL.

69) c#069: Napište základní tvar hlavní funkce main, která se používá v C programech? Uveďte další možné tvary.

- Tvar s nespecifikovaným počtem argumentů: int main() {...}.
- Tvar bez argumentů: int main(void) {...}.
- Rozšířený tvar o argumenty: int main(int argc, char *argv[]) {...}, nebo int main(int argc, char **argv) {...}.

70) c#070: Jaký je rozdíl mezi ukazatelem na konstantní hodnotu a konstantním ukazatelem?

- Ukazatel na konstantní proměnnou - můžeme pointer přesměrovat, aby ukazoval jinam, ale proměnná zůstává neměnná - const int *ptr = &variable.
- Konstantní ukazatel na proměnnou - můžeme měnit hodnotu proměnné, ale ukazatel nesmíme přesměrovat, aby ukazoval někam jinam - int *const ptr = &variable.

71) c#071: Jak v C zapíšete konstantní ukazatel na konstantní hodnotu, např., typu double?

- const double *const ptr;

72) c#072: Co je v C ukazatel na funkci? K čemu slouží a jak definujete proměnou typu ukazatel na funkci?

- Ukazatel na funkci je proměnná, která drží adresu funkce.
- Slouží k tomu, abychom přes něj volali funkce, ale dají na rozdíl od běžných volání dát do pole, načež lze provádět různé smyčky volání apod.
- Definice ukazatele functionPtr na funkci void printInteger(int num) je:

  void (*functionPtr)(int) = &printInteger;.
- Volání pak vypadá takto (*functionPtr)(nejaky_integer);.

73) c#073: Je v C rozdíl definovat složený typ pouze jako struct a prostřednictvím typedef struct? Pokud ano, tak jaký?

- Je rozdíl v deklaraci struktury:

- Pro:

```
typedef struct triangle_s {
   int a,b,c;
} triangle_alias;
```

  - píšeme triangle_alias triangle_a;
- Pro:

```
struct triangle_s {
   int a,b,c;
};
```

  - píšeme struct triangle_s triangle_a;.
- Jinými slovy: při použití typedef můžeme používat získaný alias pro jméno structu a nemusíme psát při deklaraci klíčové slovo struct.

74) c#074: Můžeme v C při definici proměnné typu pole, proměnnou přímo inicializovat? Pokud ano, jak?

- Ano, a to:
- a) inicializací všech prvků: int arr[] = {1, 2, 3};
- b) inicializací některých prvků: int arr[3] = {[0] = 1, [1] = 2};, kde třetí prvek zůstává neinicializovaný.
- Také je možné všechny prvky nastavit na 0 např. int arr[10] = {0}.

75) c#075: Můžeme v C při definici proměnné typu struct inicializovat pouze určitou položku?

- Ne, při definici nic inicializovat nesmíme. Až při inicializaci structu můžeme napsat třeba struct st my_struct = {.i = 6};.

76) c#076: Jakou funkcí v C vytisknete na obrazovku formátovaný znakový výstup? V jaké standardní knihovně je funkce definována?

- Použijeme funkci printf(); (případně fprintf(stdout, „něco“)), která je definována v knihovně stdio.h.

77) c#077: Jak v C načtete hodnotu textového řetězce a celého čísla od uživatele?

- scanf("%s %d", str, &num);

78) c#078: Jak v C vytisknete textový řetězec na standardních výstup a standardní chybový výstup? Jakou funkci k tomu použijete?

- Na standardní výstup buď printf(text); nebo fprintf(stdout, text); .
- Na standardní chybový výstup fprintf(stderr, text);.

79) c#079: V jakém kontextu se používá klíčové slovo break?

- break slouží k opuštění těla cyklu nebo switche.

80) c#080: V jakém kontextu se používá klíčové slovo case?

- case použijeme jako jednu z ověřovaných podmínek ve switchi:

81) c#081: V jakém kontextu se používá klíčové slovo continue

- Program skočí na začátek cyklu a zvýší se iterace (tj. pokračuje se dál v cyklu).

82) c#082: V jakém kontextu se používá klíčové slovo default?

- default použijeme ve switch pro případ, že se nesplní ani jedna podmínka. Je však nepovinné jej použít.

83) c#083: V jakém kontextu se používá klíčové slovo do?

- do použijeme u while smyčky, pokud chceme nejdříve vykonat první iteraci a až potom kontrolovat splnění podmínek. Například:

```
#include <stdio.h>


int main() {
   int x = 11;
   /*
    * Even though x doesn't meet the condition, the loop will start and end right after
    * the condition is checked in while().
    */
   do {
      x++;
   }
   while (x < 10);
   printf("%d\n", x); // Output: 12;
   return 0;
}
```

84) c#084: V jakém kontextu se používá klíčové slovo while?

- while použijeme jako smyčku ve tvaru while(podmínka) {...}, kde se už pro vykonání první iterace musí splnit podmínka, která se následně ověřuje pro vstup do každé další iterace.
- Lze také vytvořit nekonečný cyklus (while(„always true condition“){}).
- while smyčku nejčastěji zvolíme, pokud neznáme přesný počet iterací dopředu.

85) c#085: V jakém kontextu se používá klíčové slovo for?

- for smyčka se používá jako smyčka ve tvaru for(inicializace; podmínka; akce) {...}, kde se při vstupu evaluuje inicializace, poté se ověří podmínka a vstoupí se do první iterace, na jejímž konci se provede akce a proces pokračuje mimo evaluaci inicializace obdobně dál.

- for smyčku nejčastěji zvolíme, pokud známe přesný počet iterací dopředu.

86) c#086: Jaké bloky pro řízení cyklu definuje for cyklus?

- Bloky jsou následující: for(inicializace, podmínka, akce) {...}, kde se při vstupu evaluuje inicializace, poté se ověří podmínka a vstoupí se do první iterace, na jejímž konci se provede akce a proces pokračuje mimo evaluaci inicializace obdobně dál.

87) c#087: Jaký význam má uvedení specifikátoru register?

- Klíčové slovo register říká kompilátoru, aby proměnnou uložil do registru. Kompilátor se sám rozhodně, zda ji uloží do registru, nebo ne.
- Pozn. - Registr je paměť s rychlejším přístupem.

88) c#088: Kdy lze použít příkaz skoku goto?

- Nejčastější zdůvodnění pro použití je pro vyskočení z několika vnořených cyklů, protože break vyskočí jenom z aktuálního.
- Jedná se o nepodmíněný skok, který dělá obvykle (ne vždy) program méně čitelným. Doporučuje se nepoužívat ho. Obecně se goto používá takto:

```
#include <stdio.h>
int main() {
   goto label;
   return 0;
   label:
      return 1; // Kdyz takto zapneme program, vrati jednicku.
}
```

89) c#089: Co vrací operátor sizeof?

- sizeof vrací velikost datového typu/proměnné v bajtech.

90) c#090: Lze použít proměnnou jako argument operátoru sizeof?

- Ano, lze.

91) c#091: Jak můžeme zjistit konkrétní velikost určitého datového typu? Např. celočíselný int nebo short.

- Funkcí sizeof: sizeof(int); a sizeof(short);.

92) c#092: Jak zajistíme, že lokální proměnná ve funkci si zachová hodnotu i při opuštění funkce?

- Použitím specifikátoru paměťové třídy static.

93) c#093: Co reprezentuje klíčové slovo void?

- a) Návratový datový typ - void funkce nic nevrací.
- b) Jako parametr - int fce(void) {...} indikuje, že funkce žádný parametr nepřijímá.

- c) Jako pointer na předem nedefinovaný typ - u void *ptr = &a; může mít „a“ jakýkoliv datový typ.

94) c#094: Jak definujete konstantní hodnotu typu float?

- const float x;

95) c#095: Jak rozlišíte literál typu float a double?

- Písmenem F/f na konci: #define FLOAT_NUM 0.42F, nebo velikostí: jak vyplývá z názvu - nezávisle na architektuře má double (většinou 64 bitů) dvojnásobnou preciznost než má float (většinou 32 bitů).

96) c#096: Jak rozlišíte literál typu int a long?

- Písmenem L/l na konci: #define NUM 35 je integer, #define NUM 35L je long, nebo velikostí - int bývá zpravidla menší (většinou 32 bitů) než long (většinou 64 bitů).

97) c#097: Jak vytisknete hodnotu ukazatele int *p na standardní výstup? Jaký formátovací příkaz v printf použijete?

- Pro vytisknutí adresy nějakého pointeru ptr se doporučuje používat printf("%p", ptr);.

98) c#098: Jak vytisknete hodnotu proměnné typu int, na kterou odkazuje ukazatel deklarovaný jako int *p;?

- printf("%d\n", *p);

99) c#099: Jak získáte ukazatel na proměnnou definovanou jako double d = 12.3;

- double *ptr = &d;

100) c#100: Jak přistoupit na položku number proměnné data typu struktura?

- data.number;

101) c#101: Jak přistoupit na položku number proměnné data typu ukazatel na strukturu?

- data->number;

102) c#102: K čemu slouží modifikátor const?

- Řekneme překladači, aby hlídal, že proměnná zůstane konstantní a nedovolil nám ji měnit.

103) c#103: Jak se v C předává pole funkcím?

- Jako ukazatel na první položku pole.
- Např. pokud máme funkci int foo(int *arr) {...} a pole int arr[] = {1, 2, 3, 4};, voláme foo(arr);.

104) c#104: Je velikost paměťové reprezentace typu struct vždy součet velikostí typů jednotlivých položek?

- Ne, protože nemusí být nutně každá položka v paměti řádně zarovnána a můžou tak vzniknout hluchá místa.

- Můžeme říct kompilátoru, aby všechno nezarovnával mod čtyřmi (32 bit) nebo osmi (64 bit), protože existují třeba chary a menší věci:

  ```
  struct __attribute__((__packed__)) mystruct_A {..};
  ```

105) c#105: Podporuje jazyk C přetěžování jmen funkcí? Pokud ano, od jaké verze?

- Ne, nepodporuje.

106) c#106: Jak probíhá proces spuštění programu implementovaného v jazyce C?

- Systém namapuje do paměti zdrojový kód, statické hodnoty, předalokuje stack a začne vykonávat funkci main.

107) c#001: Popište jak v C probíhá volání funkce int doit(int r)? Jaká data jsou předávána do/z funkce a kam jsou hodnoty ukládány?

- Na zásobník se ukládá funkce, lokální proměnná a návratová adresa, kam se řízení programu po skončení funkce vrátí. Když funkce skončí, tak se funkce a lokální proměnné ze zásobníku odstraní, do zásobníku se vloží vypočtená návratová hodnota z funkce a řízení programu se vrátí na místo podle návratové adresy v zásobníku.

108) c#002: Vysvětlete rozdíl mezi proměnnou a ukazatelem na proměnnou v jazyce C?

- Proměnná uchovává hodnotu proměnné, ukazatel uchovává adresu proměnné, na kterou ukazuje. (Ukazatel jen jen adresa, neobsahuje proměnnou, která může být velká.)

109) c#003: Jaký je v C rozdíl mezi ukazatelem na konstantní proměnnou a konstantním ukazatelem? Jak definice těchto ukazatelů zapisujeme?

- Ukazatel na konstantní proměnnou - můžeme pointer přesměrovat, aby ukazoval jinam, ale proměnná zůstává neměnná - const int *ptr = &variable.
- Konstantní ukazatel na proměnnou - můžeme měnit hodnotu proměnné, ale ukazatel nesmíme přesměrovat, aby ukazoval někam jinam - int *const ptr = &variable.

110) c#004: Jak v C dynamicky alokujete paměť pro uložení posloupnosti 20 hodnot typu data_t ? Jak následně takové dynamické pole zvětšíte pro uložení dalších 10 položek?

```
int size = 20;
data_t *data = (data_t *)malloc(size * sizeof(data_t));
size += 10;
data_t *tmp= (data_t *)realloc(data, size * sizeof(data_t));
if (tmp != NULL) {  //včetně ověření správného provedení
data = tmp;
}
```

111) c#005: Jak v C zajistíte načtení textového řetězce ze souboru aniž byste překročili alokovanou paměť určenou pro uložení řetězce?

- Načítáme nějakou smyčkou (na příklad pomocí getchar()) a kontrolujeme, jestli se index, na který zapisujeme nerovná velikosti pole. Pokud ano, zvětšíme proměnnou určující velikost buď násobkem nebo přičtením nějaké konstanty a použijeme realloc.

112) c#006: Vysvětlete jaký je v C rozdíl mezi proměnnou typu ukazatel, proměnnou a polem z hlediska uložení hodnoty v paměti?

- V ukazateli se nachází adresa nějaké proměnné, v proměnné hodnota a u pole se po deklaraci zarezervují za sebou jdoucí místa v paměti o velikosti datového typu pole, do kterých lze následně zapisovat.

113) c#007: Vyjmenujte základní paměťové třídy, ve kterých mohou být uloženy hodnoty proměnných.

- register, auto (implicitní paměťová třída pro lokální proměnné), static, extern

114) c#008: Jaký je v C rozdíl mezi ukazatelem na hodnotu int a polem hodnot int, tj. int *p a int p[]?

- sizeof(p) vrací velikost adresy (typicky 8), zatímco sizeof(arr) vrací velikost v paměti celého pole.

115) c#009: Jaký význam má klíčové slovo static v závislosti na kontextu?

- a) Static proměnná ve funkci - hodnota proměnné zůstává mezi voláními.
- b) Static globální proměnná či statická funkce - je vidět jen v souboru, ve kterém je deklarována/definována.

116) c#010: Definujte pole variabilní délky o velikosti n, kterou načtete ze standarního vstupu.

```
int n;
scanf("%d", &n);
int arr[n];
```

117) c#011: Definujte diagonální (jednotkovou) matici 3×3 jako 2D pole typu int.

```
int matrix[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
```

118) c#012: Garantuje uvedení const u definice proměnné, že není žádná možnost jak příslušnou hodnotu proměnné změnit?

- Ne, můžeme adresu proměnné uložit do ukazatele a při dereferenci přepsat původní proměnnou.

```
#include <stdio.h>
int main() {
  const int x = 4;
  int *ptr;
  ptr = &x;
  *ptr = 10;
```

```
    printf("%d\n", x); // Output: 10
    return 0;
}
```

119) c#001: Je v C možné použít příkaz nepodmíněného skoku goto ke skoku z jedné funkce do jiné? Pokud ne, proč?

- Ne, labely jsou v C pouze lokální.

120) c#002: Popište k čemu slouží příkaz dlouhého skoku (longjmp/setjmp) v C. Jak se používá?

- Ke skoku mezi funkcemi.
- setjmp(jmp_buf buf) se používá k zapamatování aktuální posice a nastaví na ni buf.
- longjmp(jmp_buf buf, i) se vrátí na místo, kam ukazuje buf a vrací i.
- Používají se na příklad takto:

```
#include <stdio.h>
#include <setjmp.h> // Don't forget to include the setjmp.h library.
jmp_buf buf; // Set buf.
int main() {
   /* Print number from 0 do 'n' and repeat the process.*/
   int n;
   int x;
   setjmp(buf);
   x = 0;
   printf("Enter a number: ");
   scanf("%d", &n);
   while(1) {
      printf("%d ", x);
      if (x == n) {
         printf("\n");
         longjmp(buf, 1);
      }
      x++;
   }
   /*****************
   Output:
   Enter a number: 4
```

```
0 1 2 3 4
Enter a number: 6
0 1 2 3 4 5 6
Enter a number: 7
0 1 2 3 4 5 6 7
Enter a number:
    .
    .
    .
******************/
return 0;
}
```

121) c#003: Vyjmenujte základní rozdělení paměti přidělné spuštěnému programu z hlediska kódu, proměnných a literálů?

- Zásobník (Stack) - lokální proměnné, argumenty funkcí, návratová hodnota funkce.
- Halda (Heap) - dynamicky alokované proměnné.
- Statická data (Static Data) - globální nebo static proměnné.
- Literály (Literals) - hodnoty zapsané ve zdrojovém kódu.
- Instrukce (Instructions) (program) - strojové instrukce.

122) c#004: Vyjmenujte (čtyři) specifikátory paměťové třídy (Storage Class Specifiers – SCS).

- auto, static, register, extern

123) c#005: Jaké typy paměti dle způsobu alokace rozlišujeme v jazyce C?

- Zásobník (Stack) pro klasickou (asi se dá říct statickou) alokaci a haldu (Heap) pro dynamickou.

124) c#006: Definujte nový typ, který umožní sdílet paměť pro proměnnou typu double, nebo proměnnou typu int.

```
typedef union {
    double x;
    int y;
} name;
```

125) c#007: Co znamená klíčové slovo volatile?

- volatile říká kompilátoru, že může proměnná být změněna i něčím jiným z venku než programem, ve kterém se nachází.

126) c#008: Jaký význam má klíčové slovo extern dle kontextu?

- a) Pro funkce - je zahrnut i když ho nepíšeme.
- b) Pro proměnné - slouží k deklaraci proměnné bez vyhrazení paměti a zviditelňuje ji pro celý program.

127) cstlib#001: V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány funkce pro vstup a výstup?

- stdio.h

128) cstlib#002: V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány nejběžnější funkce std. Knihovny?

- stdlib.h

129) cstlib#003: V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány funkce pro práci s textovými řetězci?

- string.h

130) cstlib#004: Co je errno a v jakém hlavičkovém souboru standardní knihovny C je deklarováno?

- Nachází se v errno.h. Jedná se o proměnnou typu int (zkratka pro error number), která je standardně nastavená na 0 a při chybě se změní na nenulové číslo. V případě bezchybného vykonání programu se hodnota nezmění.

131) cstlib#005: Jakým způsobem jsou předávány nebo jinak ukládány chybové stavy ve většině funkcí standardní knihovny C?

- Například návratovou hodnotou -1, někdy postačí jakákoliv nenulová hodnota, někdy NULL.

132) cstlib#006: K čemu slouží makro assert a v jakém je hlavičkovém souboru standardní knihovny C?

- Nachází se v assert.h. Pokud se jeho argument ukáže jako nepravdivý, ukončuje program s nějakou chybovou hláškou.
- Obvykle se používá při ladění a debuggování, kdy usnadňuje hledání chyb.

133) cstlib#007: Ve kterém hlavičkovém souboru standardní knihovny C jsou definovány matematické funkce?

- math.h (je potřeba linkovat přepínačem -lm).

134) cstlib#008: Ve kterém hlavičkovém souboru standardní knihovny C byste hledali rozsahy základní číselných typů?

- stdint.h

135) cstlib#009: Jakým způsobem otevřete soubor pro čtení? Napište krátký (1-3 řádkový) kód?

- FILE *f = fopen(fileName, "r")
- Případně ještě kontrola úspěšného otevření (if(f == NULL){..})

136) cstlib#010: Jakým způsobem otevřete soubor pro zápis? Napište krátký (1-3 řádkový) kód?

- FILE *f = fopen(fileName, "w")
- Případně ještě kontrola úspěšného otevření (if(f == NULL){..})

137) cstlib#011: Proč je vhodné explicitně zavírat otevřený soubor? Jakou funkci standardní knihovny C k tomu použijete?

- Aby mohly soubor na příklad používat další procesy a abychom třeba nevyplýtvali maximální povolené množství otevřených souborů. Použijeme funkci fclose(file_pointer).

138) cstlib#012: Jak zjistíte, že jste při čtení souboru dosáhli konce souborů? Jakou funkci standardní knihovny C k tomu můžete použít?

- Kontrolou návratové hodnoty funkce feof(FILE *file_pointer). V případě, že soubor je soubor u konce, vrací nenulové číslo.

139) cstlib#013: Jak zjistíte podrobnosti o selhání čtení/zápisu z/do souboru s využitím funkcí standardní knihovny C?

- Podle návratové hodnoty použité funkce. Například fwrite vrací počet prvků, které zapsal. Pokud se tato hodnota liší od nmemb, nastala chyba. (Obdobně u fread())
- Případně také pomocí globální hodnoty errno.

140) cstlib#014: Jak rozlišíte chybu a dosažení konce souboru při neúspěchu čtení ze souboru, např. funkcí fscanf()?

- Při dosažení konce souboru je vráceno EOF a při neúspěchu čtení je vrácena nula.
- Nebo také využitím feof().

141) cstlib#015: Jaké znáte funkci/e standardní knihovny C pro náhodný přístup k souborům?

- int fseek(FILE *stream, long int offset, int position)
- offset: number of bytes to offset from position, position: position from where offset is added.
- Returns: zero if successful, or else it returns a non-zero value.
- Náhodný přístup nám umožňuje umístit kurzor.

142) cstlib#016: Jaké znáte funkce standardní knihovny C pro blokové čtení a zápis?

- size_t fread(void* ptr, size_t si*e_t nmemb, FILE *stream)
- size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
- ptr je to co čteme/zapisujeme, size je velikost čtených bloků, nmemb je počet čtených bloků a stream je většinou soubor do kterého zapisujeme / ze kterého čteme.

143) prg#001: Vysvětlete princip rekurze např. na výpočtu faktoriálů. Charakterizujte hlavní rozdíly mezi rekurzivním a iterativním výpočtem.

- Rekursivní faktoriál:

```
int factorial(int x) {
    if (x == 0)
        return 1;
    return x * factorial(x - 1);
}
```

- Iterativní faktoriál:

```
int factorial(int x) {
    int ret_val = 1;
    for (int i = 1; i <= x; i++)
        ret_val *= i;
    return ret_val;
}
```

- Rozdíly:
  - Rekurzivní algoritmus předepisuje výpočet „shora dolů“, zatímco iterativní „zdola nahoru“.
  - Nevýhodou rekurzivních algoritmů může být časová náročnost způsobená např. zbytečným opakováním výpočtu.
  - Rekurse většinou zabírá méně řádků. Výhodou rekurzivních funkcí je jednoduchost a přehlednost
  - Rekurzivní algoritmus lze v řadě případů nahradit iteračním, ale ne vždy.

144) prg#002: Proč je rekurzivním výpočet hodnoty prvku Fibonacciho posloupnosti výpočetně náročnější než iterativní výpočet? Na co je nutné brát ohled při rekurzivním výpočtu?

- Protože při každém zavolání je na zásobníku vytvořena nová sekce pro stávající volání funkce. Navíc je u rekurse více operací než u iterace. Složitost rekurze roste exponenciálně s $n \sim 2^n$.
- Je nutné brát ohled na správně nastavenou exit-podmínku, jestli máme dostatek paměti na zásobníku a také jestli rekurzivní algoritmus nebude mít velkou asymptotickou složitost.

145) prg#003: Charakterizujte rozdíl mezi polem a spojovým seznamem.

- 1. Pole obsahuje pouze stejné datové typy (třeba samé integery nebo chary), zatímco spojový seznam může obsahovat různé datové typy.
- 2. Každému prvku v poli náleží nějaký index, zatímco ve spojovém seznamu je potřeba začít od hlavy a k danému prvku se dostat přes jednotlivé odkazy na další prvky. Přístup k prvku v poli je tím pádem mnohem rychlejší, protože ve spojovém seznamu se jedná o lineární rychlost.
- 4. Přidání prvku někam doprostřed pole je na druhou stranu oproti spojovému seznamu mnohem pomalejší. V poli se musí všechno přeindexovat, kdežto v seznamu stačí, aby

předchozí prvek odkazoval na ten, který vkládáme a vkládaný prvek bude ukazovat na ten, na který odkazoval předchozí.

146) prg#004: Navrhněte datovou strukturu(y) pro vytvoření spojového seznamu

```
typedef struct entry {
  struct entry *next;
  void *value;
} entry_t;
typedef struct {
   entry_t *head;
   entry_t *tail; // Optional.
   int counter; // Optional.
} linked_list_t;
```

147) prg#005: Jak bude vypadat datová struktura pro realizaci spojového seznamu, u kterého chceme udržovat počet prvků v seznamu?

```
typedef struct entry {
  struct entry *next;
  void *value;
} entry_t;
typedef struct {
   entry_t *head;
   entry_t *tail; // Optional.
   int counter;
} linked_list_t;
```

148) prg#006: Jak bude vypadat datová struktura pro realizaci spojového seznamu s rychlým (O(1)) vkládáním na začátek i konec seznamu?

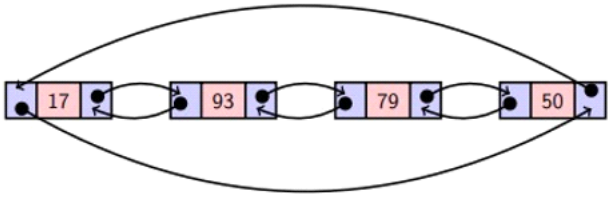
```
typedef struct entry {
  struct entry *next;
  void *value;
} entry_t;
typedef struct {
   entry_t *head;
   entry_t *tail;
   int counter; // Optional
} linked_list_t;
```

149) prg#007: Jak bude vypadat datová struktura pro realizaci spojového seznamu s rychlým (O(1)) vkládáním i odebíráním prvků na/z začátek i konec seznamu?

```
typedef struct entry {
  struct entry *next;
  struct entry *prev;
  void *value;
} entry_t;
typedef struct {
   entry_t *head;
   entry_t *tail;
   int counter; // Optional.
} linked_list_t;
```

150) prg#008: Jak se liší kruhový obousměrný spojový seznam od jednosměrného? Jaký způsobem vypíšete všechny prvky?

- V kruhovém obousměrném spojovém seznamu položka next posledního prvku odkazuje na první prvek a položka prev prvního prvku odkazuje na poslední prvek. V jednosměrném kruhovém chybí informace o prev prvku.



- Vypíšu např. takto:

```
void print(struct Node* start)
{
   struct Node *temp = start;

   //forward direction
   while (temp->next != start)
   {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("%d ", temp->data);
```

```
    //reverse direction
    Node *last = start->prev;
     temp = last;
     while (temp->prev != last)
     {
        printf("%d ", temp->data);
        temp = temp->prev;
     }
     printf("%d ", temp->data);
}
```

151) prg#009: Co je to hloubka stromu?

- Hloubka stromu je délka nejdelší cesty od kořene k listu, přičemž prázdný strom má definovánu hloubku jako -1.

152) prg#010: Jaký je analytický vztah pro výpočet hloubky binárního stromu o n vrcholech?

- floor(log(n))

153) prg#011: Co je to list stromu?

- Prvek, který nemá žádného potomka.

154) prg#012: Jak je možné reprezentovat binární strom?

- Polem, nebo spojovou datovou strukturou.

155) prg#013: Kdy budete reprezentovat binární strom polem?

- Ve chvíli kdy vím, že budu pracovat s vyváženým stromem (na příklad halda).

156) prg#014: Co musí platit pro binární strom, abychom jej mohli efektivně reprezentovat polem a přistupovat k následníkům a předkům pořadím prvku v poli?

- Že je vyvážený. To znamená:
  - 1. Každý uzel má žádného nebo 2 potomky (jednoho jen v nejspodnějším patře).
  - 2. Strom se zaplňuje z levé strany, tak aby bylo zachováno pole[2i+1] je levý potomek a pole[2i+2] je pravý potomek.

157) prg#015: Co je to plný binární strom?

- Strom, ve kterém každý vnitřní uzel má dva potomky a všechny uzly jsou co nejvíce vlevo.

158) prg#016: Kdy je strom dokonale vyvážený?

- 1. Výšky levého a pravého podstromu se liší nejvýše o 1.
- 2. & Levý podstrom je vyvážený.

- 3. & Pravý podstrom je vyvážený.

159) prg#017: Jaké znáte pořadí navštívení uzlů binárního stromu?

- a) Pre-order - levý podstrom -> kořen -> pravý podstrom.
- b) In-order - kořen -> levý podstrom -> pravý podstrom
- c) Post-order - levý podstrom -> pravý podstrom -> kořen

160) prg#018: Co je to binární vyhledávací strom?

- Datová struktura založená na binárním stromu, v němž jsou jednotlivé prvky (uzly) uspořádány tak, aby v tomto stromu bylo možné rychle vyhledávat danou hodnotu. To zajišťují tyto vlastnosti:
- 1. Jedná se o binární strom, každý uzel tedy má nanejvýš dva potomky − levého a pravého.
- 2. Každému uzlu je přiřazen určitý klíč. Podle hodnot těchto klíčů jsou uzly uspořádány.
- 3. Levý podstrom uzlu obsahuje pouze klíče menší než je klíč tohoto uzlu.
- 4. Pravý podstrom uzlu obsahuje pouze klíče větší než je klíč tohoto uzlu.

161) prg#019: Navrhněte složený datový typ (struct) pro reprezentaci n-arního stromu, tj. graf, který je stromem a vrcholy mohou mít více než dva následníky.

```
typedef struct node {
   /* We have an 'n' of children within each node, so we need an array. */
    struct node **arr_of_children;
    int arr_size;
    void *value; // Data carried by the node.
 } node_t;
typedef struct {
   /* We need to store the root, so we have a place to start searching from. */
   node_t *root;
   int arr_size; // Optional.
} tree_t;
```

162) prg#020: Charakterizujte abstraktní datový typ. Co se pod tímto pojmem myslí?

- Abstraktní datový typ je množina druhů dat a operací, které jsou specifikovány nezávisle na konkrétní implementaci.

163) prg#021: Charakterizujte základní rozdíly mezi zásobníkem a frontou?

- Zásobník je typu LIFO (last in, first out) - naposledy vložená hodnota se odebere jako první. Fronta je typu FIFO (first in, first out) – prvky se odebírají v pořadí, v jakém byly vloženy.
- U zásobníku používáme pro přístup jediný ukazatel, který ukazuje na poslední prvek. U fronty máme ukazatele dva - jeden ukazuje na první prvek ve frontě, druhý na poslední vložený prvek.

- U zásobníku se vložení a odebírání provádí jen z vrcholu zásobníku. U fronty se vkládá buď na začátek a odebírá z konce, nebo vkládá na konec a odebírá ze začátku.

164) prg#022: Charakterizujte rozdíly v implementaci zásobníku polem a spojovým seznamem?

- Na každou pozici v poli můžu přistoupit v konstantním čase, zatímco v SS potřebuji projít všechny předchozí.
- V poli jsou prvky uloženy v jednom bloku paměti, zatímco v SS jsou rozházené.
- V poli buď musíme definovat chování při zaplnění, nebo provádět realokaci, zatímco u SS to není potřeba.

165) prg#023: Charakterizujte rozdíly v implementaci fronty a kruhové fronty a to s využitím pole? Nápověda: Je dobré se zamyslet nad tím co je fronta a co je kruhová fronta.

- U lineární fronty je přidání prvku velmi jednoduché, ale odebírání je značně neefektivní, protože musíme vyjmout 1. prvek z čela a poté všechny prvky o jednu pozici posunout.
- U kruhové fronty naopak k žádnému posouvání při odebírání nedochází. Alokované pole si můžeme představit jako „zacyklené a neměníme tedy pozici jednotlivých prvků, ale pouze pohybujeme s ukazateli na začátek a konec oblasti s daty. Kruhová fronta je proto efektivnější.

166) prg#024: Jaké využití má kruhová fronta? Má smysl pevně definovat maximální počet položek v kruhové frontě? Nápověda: Kruhová fronta se vyznačuje charakteristickými vlastnostmi, které ji předurčují pro konkrétní použití, v podstatě je to její typické použití.

- V plánování procesů a správě paměť, například jako vyrovnávací paměť.
- Ano, protože zvětšení fronty je náročné.

167) prg#025: Jak nastavíte (zvolíte) vhodnou velikost kruhové fronty implementované polem definované velikosti? Nápověda: Podobně jako velikost statického pole, tak i u fronty je nutné nějak zvolit délku. Může to být 10 nebo třeba 100. Otázka je, kolik je ta správná konkrétní hodnota a na čem záleží.

- To samozřejmě záleží na rychlosti zápisu a čtení dat do fronty, na max. počtu znaků, které ti může např. modem vrátit jako odpověď na tvůj příkaz a na tom, jak ta data z fronty čteš - v cyklu nebo pomocí přerušení.[1]
- Určitě je potřeba zajistit dostatečnou velikost, aby nedocházelo k potřebě zvětšit frontu. Zároveň se ale také nesmí vytvořit zbytečně velká.

168) prg#026: Charakterizujte prioritní frontu. V čem se implementace liší od běžné fronty?

- U prioritní fronty obsahují prvky navíc hodnotu určující jejich prioritu.
- Některé prvky jsou při vyjmutí z fronty preferovány. Operace pop() odebírá z prioritní fronty prvek s nejvyšší prioritou, zatímco u běžné fronty odebírá první vložený prvek.

1 Odpověď na dotaz na stránce: https://www.itnetwork.cz/cplusplus/diskuzni-forum-c-visual-studio/jak-nastavit-vhodnou-velikost-kruhove-fronty-implementovane-polem-definovane-velikosti--5e1081c344b33

- Rozhraní mohou být stejná. Prioritní frontu můžeme implementovat různě složitě a s různými výpočetními nároky. Například buď polem/spojovým seznamem s modifikací push(), pop(), peek(), nebo využitím haldy.

169) prg#027: Jak nejjednodušeji realizujete prioritní frontu z implementace kruhové fronty polem o definované velikosti? Nápověda: Cílem je minimalizovat složitost implementace.

```
typedef struct{
    int priority;
    void *data;
} entry;
typedef struct {
    int capacity;
    int tail;
    int head;
    int real_size;
    entry_t **queue;
} queue_t;
```

- Dále je třeba upravit funkci pop(), aby vybrala prvek s nejvyšší prioritou.

170) prg#028: Jakou strukturu použijete pro efektivní implementaci prioritní fronty? V čem spočívá rychlost vkládání/odebírání prvků z fronty?

- Nejefektivnější je implementace haldou.
- U vkládání vložíme prvek na konec a necháme ho probublat kam patří.
- U odebírání prvek přesuneme do kořene probubláme dolů - rekursivně měníme uzel s menším potomkem, dokud je porušená vlastnost haldy.

171) prg#029: Definujte co je to halda a jak se používá pro implementaci prioritní fronty?

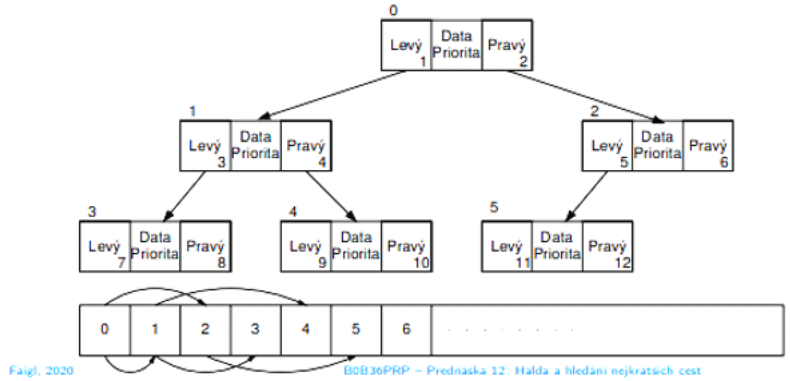
- Halda je dynamická datová struktura, která má „tvar“ binárního stromu a uspořádání prioritní fronty.
- Každý prvek haldy obsahuje hodnotu a dva potomky, podobně jako binární strom.
- Vlastnosti haldy – „Heap property“:
  - Hodnota každého prvku je menší než hodnota libovolného potomka.
  - Každá úroveň binárního stromu haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava. (Binární plný strom)
  - Prvky mohou být odebrány pouze přes kořenový uzel.
- Vlastnost haldy zajišťuje, že kořen je vždy prvek s nejnižším/nejvyšším ohodnocením.

172) prg#030: V čem se liší binární vyhledávací strom (Binary Search Tree - BST) a halda?

- BST – může obsahovat prázdná místa, hloubka stromu se může měnit, zajistit vyvážený strom je implementačně náročnější než implementace haldy
- Halda – binární plný strom, kořen stromu je vždy prvek s nejnižší(nejvyšší) hodnotou, každý podstrom splňuje vlastnost haldy (heap property).
- U haldy lze indexy rodičů a potomků spočítat, takže se vyplatí ji implementovat v poli. U BST to tak není.

173) prg#031: Jak implementujete binární strom v poli? Co musí pro takový binární strom platit?

- Pro definovaný maximální počet prvků v binárním plném stromu, si předalokujeme pole o daném počtu prvků. Binární plný strom má všechny vrcholy na úrovni rovné hloubce stromu co nejvíce vlevo.
- Kořen stromu je první prvek s indexem 0, následníky prvku na pozici i lze v poli určit jako prvky s indexy (podobně lze odvodit vztah pro předchůdce):
  - levý následník: $i_{levý} = 2i + 1$
  - pravý následník: $i_{pravý} = 2i + 2$



- (Kód z přednášky č. 12)

174) prg#032: Naznačte jak byste implementovali funkce push() a pop() pro prioritní frontu realizovanou haldou a polem?

- Polem:
  - push() je až na uložení priority identická s versí bez priorit:

```
int queue_push(void *value, int priority, queue_t *queue) {
   int ret = QUEUE_OK; // by default we assume push will be OK
   if (queue->count < MAX_QUEUE_SIZE) {
      queue->queue[queue->tail] = value;
      // store priority of the new value entry
```

```
        queue->priorities[queue->tail] = priority;
        queue->tail = (queue->tail + 1) % MAX_QUEUE_SIZE;
        queue->count += 1;
    }
    else {
        ret = QUEUE_MEMFAIL;
    }
return ret;
}
```

- U pop() musíme zajistit zaplnění místa, pokud je vyjmut prvek z prostředka fronty (pole):

```
void* queue_pop(queue_t *queue) // Případnou mezeru zaplníme prvkem ze startu {
    void *ret = NULL;
    int bestEntry = getEntry(queue); // getEntry() vybere prvek s nejvyšší prioritou
    if (bestEntry >= 0) { // entry has been found
        ret = queue->queue[bestEntry];
        if (bestEntry != queue->head) { //replace the bestEntry by head
            queue->queue[bestEntry] = queue->queue[queue->head];
            queue->priorities[bestEntry] = queue->priorities[queue->head];
        }
        queue->head = (queue->head + 1) % MAX_QUEUE_SIZE;
        queue->count -= 1;
    }
    return ret;
}
```

- Haldou:
  - Funkce push() přidá prvek jako další prvek v poli a následně propaguje prvek směrem nahoru až je splněna vlastnost haldy:

```
#define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2
_Bool pq_push(pq_heap_s *pq, int label, int cost) {
    _Bool ret = false;
    if (pq && pq->len < pq->size && label >= 0 && label < pq->size) {
        pq->cost[pq->len] = cost; //add the cost to the next free slot
        pq->label[pq->len] = label; //add label of new entry
        int cur = pq->len; // index of the entry added to the heap
```

```
        int parent = GET_PARENT(cur);
        while (cur >= 1 && pq->cost[parent] > pq->cost[cur]) {
          pq_swap(pq, parent, cur); // swap parent<->cur
          cur = parent;
          parent = GET_PARENT(cur);
        }
        pq->len += 1;
        ret = true;
      }
// assert(pq_is_heap(pq, 0)); // testing the implementation
    return ret;
}
```

- Při odebrání prvku funkcí pop() je odebrán kořen. Poté je poslední prvek v poli umístěn na začátek pole (tj. kořen stromu) a propagován směrem dolů až je splněna vlastnost haldy:

```
_Bool pq_pop(void *_pq, int *oLabel){
  _Bool ret = false;
  pq_heap_s *pq = (pq_heap_s *)_pq;
  if (pq && pq->len > 0){
    *oLabel = pq->label[0];
    pq->len -= 1;
    pq->label[0] = pq->label[pq->len];
    pq->cost[0] = pq->cost[pq->len];
    pq_down(pq);
    ret = true;
  }
  return ret;
}
static void pq_down(pq_heap_s *pq){
  int cur;
  int hl, hr;
  int best;
  cur = 0;
  hl = GET_LEFT(cur);
  while (hl < pq->len){
```

```
    hr = hl + 1;
    if (pq->cost[cur] > pq->cost[hl]){
      best = hl; // left is the candite
    }
    else{
      best = cur;
    }
    if (hr < pq->len && pq->cost[best] > pq->cost[hr]){
      best = hr; // right is the choice
    }
    if (best != cur){ // lower value found
      pq_swap(pq, cur, best);
      cur = best;
      hl = GET_LEFT(cur);
    }
    else{
      break;
    }
  }
  //check_heap(0, heap, nodes);
}
```

175) prg#033: Jak při implementaci binárního vyhledávacího stromu (BST) nebo prioritní fronty zajistíte porovnání prvků o neznámém typu (neznámém v době implementace funkcí pro práci s BST nebo prioritní frontou)? Nápověda: Představte si, že BST nebo prioritní frontu implementujete obecně, pak implementaci použijete. Inspirací může být například man 3 bsearch ze standardní knihovny.

- Při implementování stromu/fronty vytvořím obecnou porovnávací funkci, která bude jako parametr brát ukazatel na konkrétní porovnávací funkci, kterou při používání této implementace už budu znát, tak jako je to u funkce bsearch:
- (void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar) (const void *, const void *));)