1. **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 inkludovaném 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. **Jaký význam má hlavičkový soubor zdrojových souborů programu v C?**
   - jsou v něm popisy funkcí, funkce samotné a také proměnné, které se sdílejí napříč soubory
   - hlavičkový soubor je includovaný v mainu (je třeba znát pouze použití, ne jejich přímou implementaci)
3. **Jak probíhá překlad a linkování (sestavení) programu v C?**
   - preprocesor (dosazuje kusy kódu za includy, # makra, vyhazuje komentáře, spojuje řádky dohromady když tam je /). Pokud chceme spustit pouze fázi preprocessingu, přidáme přepínač -E do kompilace.
   - kompilace (kompiluje – převede programátorem napsaný zdrojový kód do spustitelné .o (binární) podoby. To jsou soubory, které již obsahují binární kód. Tento kód ale není spustitelný, 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.
   - 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.
4. **Vysvětlete rozdíl mezi překladem zdrojových souborů a linkováním programu?**
   - překlad vkládá jen relativní adresy, kdežto linkování vkládá již absolutní adresy, viz 3.
5. **Co je to preprocesor a jaká je jeho funkce při překladu zdrojového souboru v jazyce C?**
   - počítačový program, který zpracovává vstupní data
   - dosazuje kusy kódu a hlídá, jestli nebyl nějaký kus dosazen dvakrát (to nelze) viz ot. 3.
6. **Popište proces vytvoření spustitelného programu ze zdrojových souborů jazyka C.**
   - viz 3.
7. **Jaké znáte překladače jazyka C?**
   - gcc, clang, tcc, icc

   compile: gcc -c main.c -o main.o

   link: gcc main.o -o main
8. **Jak zajistíme, že se hlavičkový soubor programu v C nevloží při překladu vícekrát?**
   - pomocí tzv. head guard (hlavičkového strážce)

```
#ifndef PRG
#define PRG
…kod
#endif
```

9. **Jak zajistíte možnost ovlivnit výslednou podobu programu při překladu? Např. Velikost bufferu definovanou symbolickou konstantou?**
   - přidat přepínač -Dbufsize=1234
     (obecne : -D name=definition) + v kódu mít něco jako
     #ifndef bufsize
     #define bufsize 4321 //vychozi hodnota
     #endif
     //aby překladač neřval, že se znovu definuje bufsize
     nebo jen definice -Dněco, případně s mezerou -D něco, -D něco=4321
10. **Jak při překladu programu kompilátorem GCC nebo Clang rozšíříme seznam prohledávaných adresářů s hlavičkovými soubory?**
    - -I
    - gcc -Idir [options] [source files] [object files] [-o output file]
    - gcc -Iproj/src myfile.c -o myfile
11. **Záleží u kompilace programu kompilátorem GCC nebo Clang při specifikaci adresářů s hlavičkovými soubory na jejich pořadí?**
    - ano, např. kdybychom linkovali více knihoven.
12. **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 pripade nesplneni podminky ve výrazu assert(condition)
      Definice makra assert závisí na jiném makru, NDEBUG, které není definováno v knihovně <assert.h>. Pokud je NDEBUG definováno jako makro v programu v němž se inkluduje <assert.h>, pak funkce assert nic neudělá.
13. **Jaký tvar má hlavní funkce programu v C, která se spustí při spuštění programu prostředí s operačním systémem?**
    - int main(){} nebo int main(int argc, **char *argv[]**){}

```
main ( int argc, char **argv ) {
    // code
    return 0; // Indicates that everything went well.
}
```

14. **Jakou návratovou hodnotou programu v C indikujete úspěšné vykonání a ukončení programu? Proč zvolíte právě tuto hodnotu?**
    - return 0;
    - nula je jen jedna a také protože to značí 0 chyb
    - Je taky možné použít EXIT_SUCCESS a EXIT_FAILURE ktere jsou definované v stdlib.h
15. **Jak předáváme parametry programu implementovanému v jazyce C?**
    - pomocí příkazové řádky (zvýší se hodnota argc a zapíše se to do argv pole)

16. **Existuje nějaká jiná možnost jak předat uživatelské parametry programu jinak než jako argument programu?**
    - ano, například načíst ze standardního vstupu pomocí příkazu scanf (nebo podobných)
17. **Jaký je rozdíl mezi staticky a dynamicky linkovaným programem implementovaným v jazyce C?**
    - staticky – každá část se zkompiluje samostatně a pak se spojí do jednoho
    - dynamicky – operační systém si je zavolá až v momentě, kde je bude potřebovat v tom souboru, nejsou přímo zakompilované v binárce
    - Statická knihovna se spojuje se spustitelným souborem v době linkování programu, zatímco dynamická knihovna se spojuje se spustitelným souborem v době spuštění programu nebo až za jeho běhu.
    - 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. **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. **Jak vkládáme do zdrojového souboru programu v C hlavičkové soubory jiných modulů nebo knihoven?**
    - #include <… >(pro standardní knihovny) nebo #include "…" (nejčastěji se používá pro vlastni knihovny)
20. **Jaký rozdíl mezi použitím #include <soubor.h> a #include "soubor.h"?**
    a) #include <hledá jen ve standardních knihovnách linuxu>
    b) #include "hledá jak ve standardních knihovnách, tak u mě ve složce "
21. **Popište rozdíl mezi deklarací a definicí funkce v jazyce C?**
    - deklarací pouze řekneme, že zde tato funkce bude (zadáme spolu s parametry) ; Musíme určit jméno funkce, vstupní parametry a návratovou hodnotu.
    - definice už je přímo celá funkce – {}
22. **Jak jsou předávány parametry funkci v jazyce C?**
    - hodnota či reference se zkopíruje na stack (reference – hodnota pointeru)
    - 2 možnosti : Pass by Value a Pass by Reference.

    - Pass by Value znamená, že kopie dat je vytvořena a uložena do proměnné stejného typu s názvem parametru funkce. Jakékoli změny tohoto lokálního parametru nemají vliv na hodnotu proměnné ve volající funkci
    - Pass by Reference "odkazuje" na původní data -> používáme pointer.

    pass by reference v ccku neexistuje, vsetko je pass by value, lebo pointer je proměnná typu pointer a.k.a integer s adresou
23. **Co je to literál a co tímto pojmem označujeme?**

- konkrétní konstanta v textu (int a = 5 –> a je literál)

- podla wiki by v tomto pripade mala byt a premenna a 5 literal
  https://en.wikipedia.org/wiki/Literal_(computer_programming)
  https://www.webopedia.com/TERM/L/literal.html

24. **Jak lze v jazyce C realizovat předání parametru funkci odkazem?**
    - ccko je striktně pass by value, takže nijak
    - C nemá pass by reference, může být pouze emulováno pomocí ukazatelů (ale stejně předáme pouze *hodnotu* ukazatele a potom dereferencí získáte hodnotu literálu na který odkazuje):

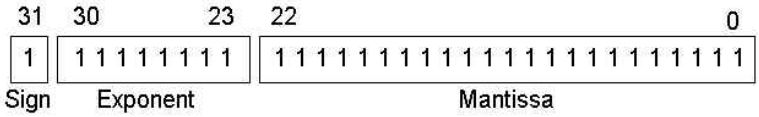
      void func(int *pointer){...}

      voláme: int a; func(&a);
    - Obecně: Ukazatele jsou proměnné, které uchovávají adresu ukazující do paměti počítače, kde je uložená konkretní proměnná.
25. **Jak lze v jazyce C omezit viditelnost funkce pouze v rámci jednoho modulu (souboru .c)?**
    - static názevfunkce
26. **Je možné v jazyce C volat funkci ze sebe sama (rekurze)?**
    - ano
27. **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 stacku

      (Stack se používá pro staticky allokovanou paměť -> lokální proměnné, pote co je funkce dokončená, tak je místo vyhrazené pro lokální proměnné uvolněno. Heap pro dynamicky allokovanou paměť ->malloc/calloc. Oba dva se nachází v RAM.)
28. **Jaké znáte kategorie proměnných z hlediska jejich umístění v paměti?**
    - lokální proměnné, argumenty funkce a návratové hodnoty jsou na stacku;
    - dynamicky alokované proměnné na heapu;
    - globální a statické proměnné na static data (pak případně ještě literály mají svou paměť)
29. **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 standardního C
    - dynamická paměť je přidělována až za běhu programu
30. **Jak v jazyce C dynamicky alokovat paměť za běhu programu?**
    - pomocí příkazu malloc() - případně calloc() a realloc() - a free()

      int *p = malloc(4*sizeof(int)); // array of 4 int
    - je možné i přes pole variabilní délky, které je sice alokované na stacku, ale dynamicky
31. **Je v jazyce C nutné uvolňovat dynamicky alokovanou paměť? Pokud ano, jak to uděláte?**
    - ano
    - pomocí příkazu free()

32. **Jak zjistíme velikost reprezentace datových typů v jazyce C?**
    - pomocí příkazu sizeof()
33. **Je vždy v jazyce C velikost proměnné typu int 32 bitů?**
    - ne, závisí na kompilátoru a na počítači
34. **Kdy je velikost ukazatele v jazyce C 32-bitů a kdy 64-bitů?**
    - závisí na architektuře počítače
    - 64-bitový systém by měl používat 64-bitů, 32-bitový 32 bitu apod.
35. **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
36. **Jak funguje modifikátor extern při definici globální proměnné v jazyce C?**
    tato proměnná bude vidět ve všech modulech
    - global variable != extern variable
37. **Jaké znáte základní znaménkové celočíselné typy v jazyce C?**
    - int, long, long long, short
    - signed char (-128 az 127)
38. **Rozlišuje jazyk C znaménkové a neznaménkové celočíselné typy? Pokud ano, co z toho plyne?**
    - ano (unsigned a signed)
    - při aritmetice se počítá jen s kladnými čísly
39. **Jaké znáte neceločíselné typy v jazyce C? Jaká je jejich vnitřní reprezentace (velikost)?**
    - float (32 bit), double (64 bit)
    - mantisa a exponent (2^exponent * mantisa)



40. **Jsou v jazyce C definovány rozsahy neceločíselných typů?**
    - nejsou
    - jediná podmínka – double musí být 2x větší než float
    - float : “Actual properties unspecified (except minimum limits), however on most systems this is the IEEE 754 single-precision binary floating-point format (32 bits).”
    - double: podle IEEE 754 je to 64 bitu
41. **Jsou v jazyce C definovány rozsahy celočíselných typů?**
    - lisia sa od architektury, preto ot. 42.
    - ano i ne - jsou daná určitá pravidla - short menší nebo rovno int, long větší nebo rovno int atd.

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

– pomocí uint16_t a dalších – kompilátor dosadí to, co má 16 bitů (vůbec se nemusí jednat o int)
– musí být includovaná knihovna <stdint.h>

da sa aj int lolo : 5; v strukture napr, nastavi ho na 5 bit

43. Je součástí jazyka C typ logické hodnoty „true/false"? Pokud ano, jak se používá? Pokud ne, jak jej definujete?

– nejsou, je potřeba includovat knihovnu <stdbool.h>, definujeme pomocí bool názevproměnné
– vlastně trochu jsou, od c99 můžeme použít "_Bool" bez includování <stdbool.h>, pokud ji ale includujeme, nic nezkazíme, _Bool a bool budou mít stejný význam

44. Jak v C definujete ukazatel na proměnnou, např. typu int?

```
int a = 5;
int *b = &a;
```

45. Jaké jsou v C omezení pro názvy proměnných a funkcí?

– nesmí začínat číslem; obsahuje pouze pismena, cisla a _, nesmí se shodovat s klíčovým slovem
- omezená délka

46. Jaké znáte escape sekvence používané v C pro řídicí znaky?

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

47. Jak jsou v C reprezentovány textové řetězce?

– pole charů, které má na konci \0. Pole s řetězcem tedy musí být vždy o 1 delší, než je délka textu, který do něj vkládáme

48. Jak v C zapisujeme identifikátory (jména funkcí a proměnných)?

– identifikátory jsou názvy C entit, jako jsou proměnné, funkce, struktury apod.
– Můžeme mít pouze alfanumerické znaky (a-z, A-Z, 0-9) a podtržítko (_). První znak identifikátoru může obsahovat pouze (a-z, A-Z) nebo podtržítko (_). Identifikátory jsou case sensitive.
– Name Convention:

1) All macros and constants in caps: MAX_BUFFER_SIZE
2) Struct names and typedef's in camelcase: GtkWidget, TrackingOrder.
3) Functions that operate on structs: gtk_widget_show()
4) Jmena, ktera naznacuji k cemu ty objekty potrebujeme

49. Jakými dvěma způsoby lze v C vytvářet konstanty?

– definovat je jako macro
– pomocí const

50. Popište výčtový typ jazyka C. Uveďte vlastnosti, které považujete za důležité?

– enum
– Konstanty výčtového typu jsou vždy celočíselné. Pokud konstantě nepřiřadíte hodnotu, pak má hodnotu o jednotku vyšší, než konstanta předešlá.

typedef enum {

vlevo, vpravo, stred, center = stred
} zarovnani;

- přehlednost, ulehčení práce (např. při definici kalendáře, stací napsat leden = 1, zbytek se očísluje a tak…)

51. **Jaké znáte logické operátory jazyka C? Jak se zapisují?**
    - && (and) || (or) ! (negation)
52. **Jaké znáte bitové operátory jazyka C? Jak se zapisují?**
    - >> (logical right shift) << & (logical and) | (logical or) ^ (exclusive or) (XOR) ~ (inverse)(NOT)
53. **Jak v C realizujete dělení a násobení dvěma s využitím operátorů bitového posunu?**
    - násobení – napr. posunu bitově o dva doleva i<<1
    - dělení – napr. posunu bitově o dva doprava i>>1

54. **Jak v C definujete složený typ (struct)?**

```
struct my_struct_s{
    int a;
}
```

- musím volat pomocí:

```
struct my_struct_s A; //declare structure A
```

- Pole může obsahovat několik datových položek stejného druhu. Podobně struktura umožňuje kombinovat datové položky různých druhů.

55. **Jak zavedeme nový typ struktury, např. pojmenovaný my_struct_s.**

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

volání

```
my_struct_s A;
```

56. **Kdy nemusíme psát před identifikátor určující jméno/typ struktury klíčové slovo struct?**
    - když použijeme typedef
57. **Co je v jazyce C pointerová (ukazatelová) aritmetika a jak se používá?**
    - jedná se o aritmetiku ukazatelů (tedy proměnných, které ukazují na adresu jiné proměnné)
    - *"There are four arithmetic operators that can be used on pointers: ++(pokud je pointer na int, tak ptr++=ptr+4), --(pokud ukazatel na int, tak minus 4), +, and -. Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared. Otherwise it doesn't have a point"*

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

- pole je ukazatel na první prvek "pole": int pole[50]
  pole je pak ukazatelem na: &pole[0]
- pole[1] = *(pole+1)

59. Jak v C přistupujeme k datovým položkám složeného typu (struct)?

- pomocí tečky (.)

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

- msg.data
- msg->data

61. Jaký v C rozdíl mezi typy struct a union?

- struct
  - kompilátor alokuje pamět pro každého člena, tudíž velikost structu je větší nebo rovna součtu velikostí všech členů
  - změna hodnoty člena neovlivní ostatní
  - několik členů může mít inicializováno najednou
- union
  - kompilátor alokuje paměť dle člena s největší velikostí, tudíž velikost unionu je rovna velikosti největší členu –> šetří paměť
  - změna hodnoty člena změní hodnoty ostatních členů
  - přístup pouze k jednomu členu v daný čas
  - pouze první člen může být inicializován

62. Stručně popište typ union používaný v jazyce C.

- syntaxe je stejná jako u struktury, až na to, že místo klíčového slova struct se použije klíčové slovo union. Umožňuje ukládat různé typy dat do stejné paměti. Můžete definovat union s mnoha členy, ale pouze jeden člen může obsahovat hodnotu v daném okamžiku. Uniony poskytují účinný způsob použití stejné paměťové lokace pro víceúčelové aplikace.
- kompilátor alokuje paměť dle člena s největší velikostí, tudíž velikost unionu je rovna velikosti největší členu –> šetří paměť

63. Jak se v jazyce C používá operátor přetypování?

- napíšeme před výraz do závorky typ, na který chceme přetypovat
  (int)promenna
- anglicky: Type Casting
- nelze přetypovat cokoliv (např. řetězec na číslo)

64. Co v C reprezentuje typ void?

- speciální datový typ, kdy se při návratu neposkytuje do volajícího výrazu žádná návratová hodnota
- void je považováno za datový typ (pro organizační účely), ale je to v podstatě klíčové slovo, které používáme jako zástupný symbol, na jehož místě bychom napsali jméno datového typu. Tedy void v podstatě reprezentuje "žádná data".

65. Co v C reprezentuje typ void*?

- void * deklaruje ukazatel bez typu (void *g;)
- Deklaruje ukazatel, ale bez zadání typu dat, na který ukazuje. (takže můžeme použít přetypovani jako např. u (int*)malloc(...), jelikož malloc taky vrací void*)

66. **Jak v C realizujete opuštění dvou nebo více vnořených cyklů z nejvnitřnějšího cyklu?**
    - nastavením iteratora na horní mez anebo goto
    - můžeme taky přerušit cyklus pomoci kombinací breaku který se volají v každém ze vnořených cyklu pokud je splněna nějaká podmínka. T.j. nejdřív vystoupíme ze vnitřního cyklu a pak postupně ze všech ostatních
    - Pokud chceme ukončit program/funkci úplně použijeme return
    - Nebo dáme vnořené cykly do funkci a pak jenom zavolame return z funkce ze vnitrniho cyklu :D
    - přidáním kontrolní proměnné - for (i = 0; i < 10 && !quit; i++)
67. **Co v C reprezentuje identifikátor NULL?**
    - jedná se o konstantu, která udává, že pointer je prázdný a že zrovna na nic neukazuje
    - můžeme myslet na NULL jako na nulový ukazatel. Některé z nejběžnějších případů použití pro NULL jsou:
      a) inicializovat ukazatel proměnné, když mu zatím není přiřazena žádná platná adresa.
      b) Můžeme provádět zpracování chyb v kódu souvisejícím s ukazatelem, např. dereference ukazatel ukazatel pouze pokud to není NULL.
      c) Když nechcete předat žádnou platnou adresu paměti.
68. **Jak nastavíte proměnnou typu ukazatel na prokazatelně neplatnou hodnotu?**
    - = 0 anebo = NULL
69. **Napište základní tvar hlavní funkce main, která se používá v C programech? Uveďte další možné tvary.**
    - int main(){} nebo int main(int argc, **char *argv[]**){}

```
main ( int argc, char **argv ) {
    // code
    return 0; // Indicates that everything went well.
}
```

70. **Jak v C zapíšete konstantní ukazatel na konstantní hodnotu, např., typu double?**
    - const double * const ptr;
71. **Co je v C ukazatel na funkci? K čemu slouží a jak definujete proměnou typu ukazatel na funkci?**
    - First thing, let's define a pointer to a function which receives 2 ints and returns an int:

```
int (*functionPtr)(int,int);
```

Now we can safely point to our function: *functionPtr = &addInt;*
Now that we have a pointer to the function, let's use it: *int sum = (*functionPtr)(2, 3); // sum == 5*

najkrasnejsie vyuzitie je pseudo objekt ako napr v hw07opt - do struktury ulozene pointre na funkcie zavisle na type fronty

- void (*fun_ptr)(int) = &fun; //kde fun je funkce:
  `void fun(int a){return 0;}`

72. **Je v C rozdíl definovat složený typ pouze jako struct a prostřednictvím typedef struct? Pokud ano, tak jaký?**
    - viz 54 a 55. Je rozdíl ve volání struktury. Když definujeme pomoci typedef, pak nemusíme psát klíčové slovo struct v definici.
73. **Můžeme v C při definici proměnné typu pole, proměnnou přímo inicializovat? Pokud ano, jak?**
    - ano, např. int pole[] = { 0, 0, 0, 0}; int myArray[10] = { 0 }; // all elements 0

    Nebo: char array[255]=”Hello”;

    Nebo inicializace pouze některých prvku: int n[5] = {[4]=5,[0]=1,2,3,4} // holds 1,2,3,4,5
74. **Můžeme v C při definici proměnné typu struct inicializovat pouze určitou položku?**
    - ne, při definici nemůžeme inicializovat nic. Pouze při inicializaci! https://stackoverflow.com/questions/13716913/default-value-for-struct-member-in-c
    - (my_struct a = {.x = 1};
75. **Jakou funkcí v C vytisknete na obrazovku formátovaný znakový výstup? V jaké standardní knihovně je funkce definována?**
    - printf(); (taky fprintf(stdout,”smth”))
    - knihovna <stdio.h>
76. **Jak v C načtete hodnotu textového řetězce a celého čísla od uživatele?**
    - int a; char pole[255];

    scanf(“%i”, &a); scanf(“%s”, pole); -> to nacte jenom prvni slovo Napr: scanf ("%[^\n]%*c", name);

    Nebo: fgets (pole, 255, stdin);
77. **Jak v C vytisknete textový řetězec na standardních výstup a standardní chybový výstup? Jakou funkci k tomu použijete?**
    - fprintf(stderr, “….”);
    - fprintf(stdout, “….”) - stejný výsledek jako printf(“....”);
78. **V jakém kontextu se používá klíčové slovo break?**
    - slouží k opuštění těla cyklu nebo příkazu switch
79. **V jakém kontextu se používá klíčové slovo case?**
    - označuje jednu “možnost” v příkazu switch
80. **V jakém kontextu se používá klíčové slovo continue**
    - Program skočí na začátek cyklu a zvýší se iterace (tj. pokracuje se dal v cyklu (pokud není porušena podmínka průběhu cyklu))
81. **V jakém kontextu se používá klíčové slovo default?**
    - při použití switch, abychom ošetřili situaci, kdy se nesplní ani jeden case
82. **V jakém kontextu se používá klíčové slovo do?**

- pro použití do…while smyčky
- do { něco } while(podmínka)
- protože se podmínka nachází až na konci, cyklus proběhne vždy alespoň jednou

83. V jakém kontextu se používá klíčové slovo while?

- while (podmínka) {}
- pro vytvoření cyklu, lze i nekonečný cyklus (while(1){})
- na rozdíl od do..while se podmínka vyhodnotí hned na začátku, takže cyklus nemusí proběhnout ani jednou

84. V jakém kontextu se používá klíčové slovo for?

- for cyklus, nejčastěji když známe přesný počet opakování
- for(int i = 0; i < hranice; ++i) {}

85. Jaké bloky pro řízení cyklu definuje for cyklus?

- for(expr1; expr2; expr3){}
- expr1 - inicializace kontrolní proměnné (obecne cokoliv, co se vykona na zacatku for_cyclu)
- expr2 – testování podmínky, řídícího výrazu
- epxr3 – aktualizace kontrolní proměnné (přičítání, odečítání atd.), vykoná se až na konci smyčky (obecne cokoliv, co se ma udelat pri kazde iteraci)

86. Jaký význam má uvedení specifikátoru register?

- klíčové slovo registr říká překladači (kompilátorů), že daná proměnná se může uložit do registru – rychlejší přístup než do paměti. Kompilátor sám zvolí zda ji uloží do registru nebo ne.
- nelze použít se „static“, lze použít pro pointery atd. Nelze na něj odkazovat pointrem.

87. Kdy lze použít příkaz skoku goto?

- třeba když chci vyskočit z několika vnořených cyklů (ale dá se tomu vyhnout přidáním kontrolní proměnné - for (i = 0; i < 10 && !quit; i++)
- nepodmíněný skok, dělá program méně čitelným a doporučuje se ho nepoužívat
- label: ……. goto: label; -> to make a loop

88. Co vrací operátor sizeof?

- velikost datového typu v bajtech

89. Lze použít proměnnou jako argument operátoru sizeof?

- ano

90. Jak můžeme zjistit konkrétní velikost určitého datového typu? Např. celočíselný int nebo short.

- sizeof(int), sizeof(short)

91. Jak zajistíme, že lokální proměnná ve funkci si zachová hodnotu i při opuštění funkce?

- pomoci keyword static

92. **Co reprezentuje klíčové slovo void?**
    - void funkce nemá návratovou hodnotu, void* ukazuje na libovolný datový typ
93. **Jak definujete konstantní hodnotu typu float**
    const float a = 1.5; (jedna se defakto o definici spojenou s inicializaci, rozdelit tyto procesy by nedavalo smysl)
94. **Jak rozlišíte literál typu float a double?**
    - podle velikosti. float 32 má 32 bitu, double 64 (obecne je double vzdy 2x vetsi nez float, 32 bitu neni zaruceno, zalezi na standardech, C velikosti datovych typu implicitne nedefinuje)
    - float f = 0.154f;
    - double d = 0.154;
95. **Jak rozlišíte literál typu int a long?**
    - literál pro long zápisu long a = 156l;
96. **Jak vytisknete hodnotu ukazatele int *p na standardní výstup? Jaký formátovací příkaz v printf použijete?**
    - %p, printf("%p", p);
97. **Jak vytisknete hodnotu proměnné typu int, na kterou odkazuje ukazatel deklarovaný jako int *p;?**
    - printf("%d", *p);
98. **Jak získáte ukazatel na proměnnou definovanou jako double d = 12.3?**
    - double *dp = &d;
99. **Jak přistoupit na položku number proměnné data typu struktura?**
    - data.number;
100. **Jak přistoupit na položku number proměnné data typu ukazatel na strukturu?**
    data->number;
101. **K čemu slouží modifikátor const?**
    - určíme tím, že daná proměnná je neměnná, označíme ji za konstantu, překladač hlídá abychom do nich nepřiřazovali novou hodnotu -> nedovolí ji změnit
    - např. const int a = 5;
102. **Jak se v C předává pole funkcím?**
    - jako ukazatel na první položku pole
103. **Je velikost paměťové reprezentace typu struct vždy součet velikostí typů jednotlivých položek?**
    - Ne, každá položka musí být v paměti řádně zarovnána a tak mohou vzniknout "hluchá" místa -> padding

    Kompilátoru lze ovšem říct, aby padding neprováděl:

```
struct __attribute__((__packed__)) mystruct_A {
    char a;
    int b;
    char c;
};
```

104. **Podporuje jazyk C přetěžování jmen funkcí? Pokud ano, od jaké verze?**

– Ne. Podporuje pouze C++ za předpokladu odlišení funkcí na základě odlišnosti v počtu nebo typech parametrů

"Jde o deklarování funkcí se stejným jménem, které by se chovaly jinak v závislosti na typech vstupních parametrů. Nic takového nebylo možné v jazyce C, kde mohla být funkce daného jména maximálně jedna. V jazyce C++ je možné vytvořit libovolné množství funkcí stejného jména za předpokladu, že budou rozlišitelné."

105. Jak probíhá proces spuštění programu implementovaného v jazyce C?

– ./a.out v prislusnem adresari

Program se nachází v sekundární paměti počítače(hard disk). Po spuštění programu je celý zkopírován do paměti RAM. Potom procesor načte několik instrukcí (záleží na velikosti sběrnice) najednou, umístí je do registrů a provede je. Computer program používá dva druhy paměti: stack a heap, které jsou také součástí primární paměti počítače. Stack se používá pro nedynamickou paměť a heap pro dynamickou paměť

106. Popište jak v C probíhá volání funkce int doit(int r)? Jaká data jsou předávána do/z funkce a kam jsou hodnoty ukládány?

– do funkce vstupuje: int, z funkce: int, ukladaji se na stack

volani napriklad: int value = doit(5);

107. Vysvětlete rozdíl mezi proměnnou a ukazatelem na proměnnou v jazyce C?

– proměnná je uložená na určité adrese a má nějakou hodnotu, ukazatel na proměnnou odkazuje na adresu, na které je uložená

pointer je premenna typu pointer, je to len 32/64bit cislo s adresou kterou tam dame

108. Jaký je v C rozdíl mezi ukazatelem na konstantní proměnnou a konstantním ukazatelem? Jak definice těchto ukazatelů zapisujeme?

– ukazatel na konstantní proměnnou const char* myPtr = &a; , *myPtr nelze již změnit např. *myPtr =b; (mějme char a = …, char b =…)

– char *const myPtr, nemůžeme změnit kam tento pointer ukazuje

109. 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?

```
data_t *data = (data_t *)malloc(20 * sizeof(data_t));
data_t *tmp = (data_t*)realloc(data, 30 * sizeof(data_t));
if (tmp != NULL) {  //včetně ověření dostatku paměti
      data = tmp;
}
```

110. Jak v C zajistíte načtení textového řetězce ze souboru aniž byste překročili alokovanou paměť určenou pro uložení řetězce?

- postupnou realokací/dynamickou alokací

111. Vysvětlete jaký je v C rozdíl mezi proměnnou typu ukazatel, proměnnou a polem z hlediska uložení hodnoty v paměti?

– Pole je ukazatel na první prvek pole, ukazatel je adresa na které je uložena proměnná na kterou odkazuje.Jakmile v céčku deklarujeme nějakou proměnnou ve

zdrojovém kódu a aplikaci spustíme, Céčko si řekne operačnímu systému o tolik paměti, kolik je pro tuto proměnnou třeba. Od systému získá přidělenou adresu do paměti, na kterou může hodnotu proměnné uložit (zjednodušeně řečeno). Tento proces nazýváme alokace paměti.

– ukazatele jsou (obvykle) uloženy do stacku. Paměť, na kterou směřují (obvykle se přiděluje přes malloc / calloc) je (obvykle) na heapu.

Dynamicky allokovane promenne (pomocí malloc, calloc) --------> heap
Proměnné deklarované v mainu-----> stack
globální proměnné ------->heap ??nejsou na static??
lokalni promenne------->stack
pole je ukazatel, takze asi na stacku, pokud neni mallocovano

You got some of these right, but whoever wrote the questions tricked you on at least one question:

- global variables -------> data (correct)
- static variables -------> data (correct)
- constant data types -----> code and/or data. Consider string literals for a situation when a constant itself would be stored in the data segment, and references to it would be embedded in the code
- local variables(declared and defined in functions) --------> stack (correct)
- variables declared and defined in `main` function -----> ~~heap~~ also stack (the teacher was trying to trick you)
- pointers(ex: `char *arr` , `int *arr`) -------> ~~heap~~ data or stack, depending on the context. C lets you declare a global or a `static` pointer, in which case the pointer itself would end up in the data segment.
- dynamically allocated space(using `malloc` , `calloc` , `realloc`) --------> ~~stack~~ heap

It is worth mentioning that "stack" is officially called "automatic storage class".

112. Vyjmenujte základní paměťové třídy, ve kterých mohou být uloženy hodnoty proměnných.

- **auto** - Jedná se o implicitní paměťovou třídu pro lokální proměnné (stack)
- **static** - Proměnné této třídy jsou stejně jako proměnné globální uloženy v datové oblasti paměti a tedy existují po celou dobu provádění programu. Tuto třídu využívají především lokální proměnné, které si ponechávají svojí hodnotu mezi jednotlivými voláními definiční funkce.
- **extern** - pro globální proměnné. Tyto proměnné jsou uloženy v datové oblasti. Klíčové slovo extern použijeme v případě kdy potřebujeme aby jedna proměnná byla viditelná ve více souborech.V jednom souboru je třeba takovou globální proměnnou definovat (bez extern) v dalších souborech, v nichž má být viditelná ji pouze tzv. deklarujeme právě pomocí klíčového slova extern.
- **register** - Proměnné této třídy mohou být (uzná-li to překladač za vhodné) umístěny přímo v některém z registrů. Výhoda takto umístěných proměnných tkví v mnohem rychlejším přístupu k nim ve srovnání s proměnnými uloženými v paměti což poněkud urychlí program.

113. Jaký je v C rozdíl mezi ukazatelem na hodnotu int a polem hodnot int, tj. int *p a int p[]?

- sizeof (array) vrátí množství paměti použité všemi prvky v poli
- sizeof (ukazatel) pouze vrací množství paměti použité samotnym ukazatelem
- & (array) je stejne jako &pole[0] a vrátí adresu prvního prvku v poli
- & (ukazatel) vrátí adresu ukazatele
- char array [] = "abc" nastavuje první čtyři prvky v poli na 'a', 'b', 'c' a '\ 0'
  char * pointer = "abc" nastaví ukazatel na adresu řetězce "abc" (který může být uložen v paměti jen pro čtení a tedy neměnný)
- Ukazateli může být přiřazen hodnota, poli ne.
  int a [10]; int * p;
- p = a; /*správne*/ a = p; /*ilegální*/

apod….

114. Jaký význam má klíčové slovo static v závislosti na kontextu?

– static fce a static proměnná - bude mít vnitřní linkování a nebude viditelné z ostatních modulů

Statická proměnná uvnitř funkce nechává svoji hodnotu po skončení funkce.

Statická globální proměnná nebo funkce je "vidět" pouze v souboru, v němž je deklarována

– static v rámci funkce – viz 35

115. Definujte pole variabilní délky o velikosti n, kterou načtete ze standardního vstupu.

- int n = 0; scanf("%d", &n); int vla[n];

116. Definujte diagonální (jednotkovou) matici 3×3 jako 2D pole typu int.

- int Array[3][3]={{1,0,0},{0,1,0},{0,0,1}};

117. Garantuje uvedení const u definice proměnné, že není žádná možnost jak příslušnou hodnotu proměnné změnit?

- ne, lze změnit hodnotu pomoci pointeru

118. Je v C možné použít příkaz nepodmíněného skoku goto ke skoku z jedné funkce do jiné? Pokud ne, proč?

- Nemůžete v Standardnim C; labely jsou pouze lokální (pro jednu funkci). Nejbližším standardním ekvivalentem jsou funkce setjmp () a longjmp ().

119. Popište k čemu slouží příkaz dlouhého skoku (longjmp/setjmp) v C. Jak se používá?

- setjmp si zapamata momentalnu poziciu a nastavi tam jmp_buf
- longjmp skoci na jmp_buf
- macro int setjmp, uloží aktuální environment do promenne specialniho typu pro pozdější použití funkcí longjmp (). Pokud se toto makro nevola z longjump(), vrátí hodnotu 0, ale pokud se vrátí z volání funkce longjmp (), vrátí hodnotu předanou na longjmp jako druhý argument.
- https://www.tutorialspoint.com/c_standard_library/c_macro_setjmp.htm

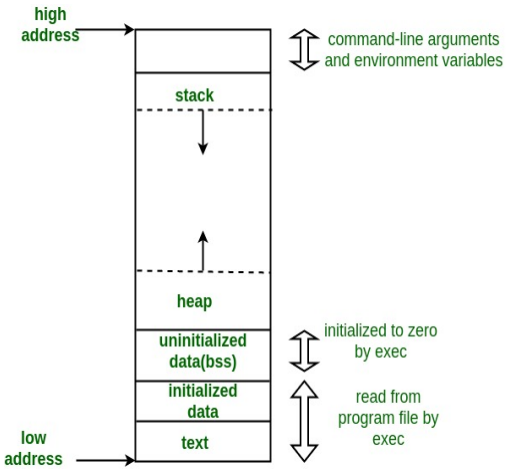
– slouží ke skoku mezi funkcemi

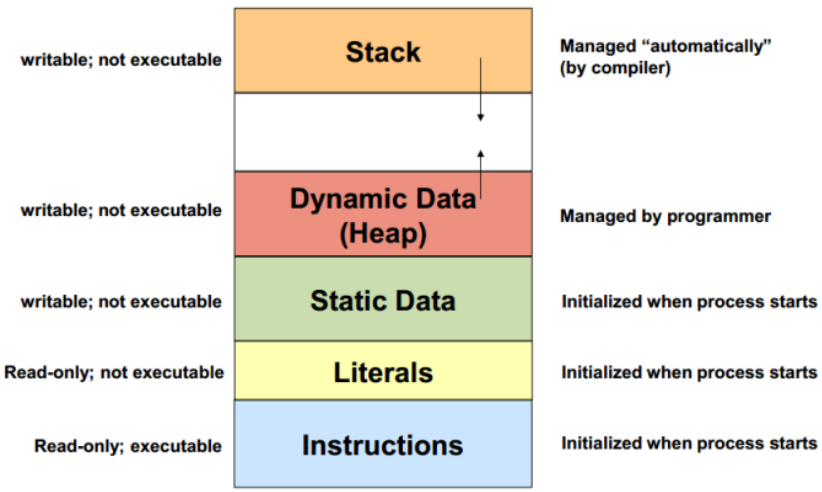
– setjump(jmp_buf buf) – používá buf k zapamatování si aktuální pozice (vrací 0)
longjmp(jmp_buf buf, i) – vrátí se na místo, kam ukazuje buf (vrací i)

- příklad: https://www.geeksforgeeks.org/g-fact22-concept-of-setjump-and-longjump/

120. Vyjmenujte základní rozdělení paměti přidělené spuštěnému programu z hlediska kódu, proměnných a literálů?

- kód (najnižšie adresy - text segment/code segment), literály, global variables a static variables (initialized segment, uninitialized segment - tie ktore sú nedefinované, resp = 0), variables - heap (dynamicky alokované), stack
-





- https://www.geeksforgeeks.org/memory-layout-of-c-program/

121. Vyjmenujte (čtyři) specifikátory paměťové třídy (Storage Class Specifiers).

- auto,static,register,extern

122. Jaké typy paměti dle způsobu alokace rozlišujeme v jazyce C?

- Staticky(stack) a dynamicky(heap) alokovaná.

123. Definujte nový typ, který umožní sdílet paměť pro proměnnou typu double, nebo proměnnou typu int.

- typedef union {
  double doub;

```
        int integer;
    } sharing_int_double;
```

124. Co znamená klíčové slovo volatile?

- Volatile určuje, že proměnná bude vždy na stejném místě v paměti (například nebude se ukládat do cache apod.)
- k obsahu proměnné může přistupovat ještě jiný proces, je viditelná i v ostatních modulech
- volatile odpovida za to ze se dana promenna neuklada do cache. Hodnota dané volatilní proměnné bude tedy převzata z hlavní paměti vždy, když se s ní setká. Tento mechanismus se používá, protože kdykoliv může být hodnota upravena OS nebo libovolným přerušením a pokud ji vezmeme s cache tak se o tom vubec nedozvime

125. Jaký význam má klíčové slovo extern dle kontextu?

- pro funkce – extern je zahrnut i když jej nepíšeme
- pro proměnné - pouze deklarujeme nebude vyhrazená paměť pro ně, zviditelňuje proměnnou pro celý program – např. deklarujeme mimo main, definice proměnné bude v nějakém hlavičkovém souboru – pak můžeme proměnnou v mainu změnit

viz 36

126. V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány funkce pro vstup a výstup?

- stdio.h

127. V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány nejběžnější funkce std. knihovny?

- stdlib.h

128. V jakém hlavičkovém souboru standardní knihovny C jsou deklarovány funkce pro práci s textovými řetězci?

- string.h

129. Co je errno a v jakém hlavičkovém souboru standardní knihovny C je deklarováno?

- <errno.h> extern int errno;
- ze zacatku je errno=0, v případě úspěšného bezchybného vykonání programu se nemění
- error number – najde chybu a vypíše číslo chyby

130. Jakým způsobem jsou předávány nebo jinak ukládány chybové stavy ve většině funkcí standardní knihovny C?

- funkce vraci hodnotu -1 nebo NULL nebo EOF
- EXIT_SUCCESS a EXIT_FAILURE Defined in header <cstdlib.h>

131. K čemu slouží makro assert a v jakém je hlavičkovém souboru standardní knihovny C?

- <assert.h>
- umožňuje zápis diagnostických informací na standardní chybový výstup, tedy zjednodusuje debuggovani

assert ukončí program (se zprávou), pokud se jeho argument ukáže jako nepravdivý. Obvykle se používá při ladění, pokud dojde k neočekávanému stavu.

132. Ve kterém hlavičkovém souboru standardní knihovny C jsou definovány matematické funkce?

- math.h linkujeme pomoci -lm

133. Ve kterém hlavičkovém souboru standardní knihovny C byste hledali rozsahy základní číselných typů?

- stdint.h // (napriklad uint8_t) a taky muzem pouzit `inttypes.h`.

134. Jakým způsobem otevřete soubor pro čtení? Napište krátký (1-3 řádkový) kód?

– FILE* soubor= fopen(”text.txt“, “r”); popř. místo text.txt třeba argv[1]

135. Jakým způsobem otevřete soubor pro zápis? Napište krátký (1-3 řádkový) kód?

– FILE* soubor = fopen(”text.txt“, “w”); popř. místo text.txt třeba argv[1]

136. Proč je vhodné explicitně zavírat otevřený soubor? Jakou funkci standardní knihovny C k tomu použijete?

– fclose(), Aby došlo k uvolnění paměti, kterou zabírá soubor.

137. Jak zjistíte, že jste při čtení souboru dosáhli konce souborů? Jakou funkci standardní knihovny C k tomu můžete použít?

- př. funkcí (int feof(FILE *stream)), která vrátí na konci souboru nenulovou hodnotu
- stdio.h
- while ((c=getchar()) != EOF) {pole[i++]=c;} NE!
- getc () vrátí EOF po dosažení konce souboru a také vrátí EOF, když selže. Takže pouze porovnání hodnoty vrácené metodou getc () s EOF není dostatečné pro kontrolu skutečného konce souboru. Chcete-li vyřešit tento problém, C poskytuje feof (), který vrátí nenulovou hodnotu pouze v případě, že konec souboru dosáhl, jinak vrací 0.

138. Jak zjistíte podrobnosti o selhání čtení/zápisu z/do souboru s využitím funkcí standardní knihovny C?

- podľa return value použitej funkcie
- príklad:
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
- príklad: fwrite vracia pocet prvkov ktory zapisal. Ak sa tato hodnota lisi od nmemb, nastal chyba. (podobne pre fread, kde bud nastala chyba alebo EOF)

139. Jak rozlišíte chybu a dosažení konce souboru při neúspěchu čtení ze souboru, např. funkcí fscanf()?

- pomocou feof()

140. Jaké znáte funkci/e standardní knihovny C pro náhodný přístup k souborům?

- Random access means you can move to any part of a file and read or write data from it without having to read through the entire file.
- fseek(`FILE *stream, long int offset, int whence`);

141. Jaké znáte funkce standardní knihovny C pro blokové čtení a zápis?

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
- kde do ptr pojde to co vyjde z funkcie a stream to co ide do funkcie, nmemb je pocet elementov o velkosti size

142. Co je to proces v terminologii operačního systému?

- proces = instance programu, který je právě spuštěn operačním systémem ve vyhrazeném prostoru paměti
- skládá se z kódu programu a jeho současného stavu spuštění (Executing - právě běžící na procesoru; Blocked - čekající na periferie; Waiting - čekající na procesor)
- může se skládat z jednoho nebo více vláken
- procesy jsou využívány operačním systémem pro oddělení různých běžících aplikací
  - proces je tvořen paměťovým prostorem a jedním nebo více vlákny, přičemž tento paměťový prostor jednotlivá vlákna sdílejí
  - v rámci operačního systému pak může běžet více procesů, ovšem každý proces již má svůj vlastní paměťový prostor
- Proces je identifikován v systému identifikačním číslem PID.
- Plánovač procesů řídí efektivní přidělování procesoru procesům na základně jejich vnitřního stavu.

143. Budete se snažit svůj program paralelizovat i když máte pouze jeden procesor? Svou odpověď zdůvodněte.

- Ano. Pre simultanne spracovanie viacerych poziadaviek
- i na jednom procesoru může běžet simultánně více vláken (pseudoparalelizmus) - praktický příklad: zpracování soketů na webových serverech - každému klientovi je přiřazeno jedno vlákno, které se stará o komunikaci s ním
- tato paralelizace nepřinese výhody v oblasti výpočetního výkonu, ale v organizaci programu; smysl má především v objektově orientovaném programování

144. Jaké základní operace související s paralelním programováním (více procesové/vláknové) řeší programovací jazyky s explicitní podporou paralelismu?

- jazyky s explicitní podporou paralelizmu jsou takové, které nabízejí výrazové prostředky pro vznik nového procesu/vlákna (bez podpory paralelizmu je nutné využívat např. služby OS pro paralelizaci, nebo ponechat vše na kompilátoru a OS)
- takový jazyk musí poskytnout:
  1) Prostředky pro tvorbu a rušení procesů
  2) Prostředky pro správu více procesorů a procesů, rozvrhování procesů na procesory.
  3) Systém sdílené paměti s mechanismem řízení.
  4) Mechanismy mezi-procesní komunikace.
  5) Mechanismy synchronizace procesů.

- granularita procesů je rozdělení od paralelismu na úrovni instrukcí až po paralelismus na úrovni programů.

145. Jaké entity (operačního systému) slouží k řízení přístupu ke sdíleným zdrojům?

146. Jak lze standardní vstup a výstup využít pro komunikaci mezi procesy?

147. Co je to vlákno (thread)?

Vlákno je samostatně prováděný výpočetní tok.
Vlákna běží v rámci procesu.
Vlákna jednoho procesu běží v rámci stejného prostoru paměti.
Každé vlákno má vyhrazený prostor pro specifické proměnné (runtime prostředí).

„Vlákna jsou lehčí variantou procesů, navíc sdílejí paměťový prostor."

1)Efektivnější využití zdrojů.
Čeká-li proces na přístup ke zdroji, předává řízení jinému procesu.
Čeká-li vlákno procesu na přístup ke zdroji, může jiné vlákno téhož procesu využít časového kvanta přidělené procesu.

2)Reakce na asynchronní události.
Během čekání na externí událost (v blokovaném režimu), může proces využít CPU v jiném vlákně.

3)Vstupně výstupní operace.
Vstupně výstupní operace mohou trvat relativně dlouhou dobu, která většinou znamená nějaký druh čekání. Během komunikace, lze využít přidělený procesor na výpočetně náročné operace.

4)Interakce grafického rozhraní.
Grafické rozhraní vyžaduje okamžité reakce pro příjemnou interakci uživatele s naší aplikací. Interakce generují událostí, které ovlivňují běh aplikace. Výpočetně náročné úlohy, nesmí způsobit snížení interakce rozhraní s uživatelem.

148. Jaký je rozdíl mezi vláknem a procesem?

Procesy

Výpočetní tok.
Běží ve vlastním paměťovém prostoru.
Entita OS.
Synchronizace entitami OS (IPC).
Přidělení CPU, rozvrhovačem OS.
- Časová náročnost vytvoření procesu.

Běží ve společném paměťovém prostoru.
Uživatelská nebo OS entita.
Synchronizace exkluzivním přístupem k proměnným.
Přidělení CPU, v rámci časového kvanta procesu.
+ Vytvoření vlákna je méně časově náročné.

Vlákna procesu

Výpočetní tok.

149. Co musí úloha splňovat, aby mělo smysl uvažovat o vícevláknové architektuře aplikace (obecně, ne konkrétní typ aplikací)?

- práce s větším kvantem dat, náročnější početní výkony atd.
- vícero na sobě nezávislých podúloh
- může být na určitou dobu zablokována
- musí reagovat na asynchronní události
- podúlohy mají odlišnou prioritu
- hlavní početní úloha může být zrychlena paralelním algoritmem s použitím více-jádrových procesorů

150. Jaké výhody má vícevláknová aplikace oproti víceprocesové aplikaci?

Vícevláknová aplikace má oproti více procesové aplikaci výhody:

Aplikace je mnohem interaktivnější.

Snadnější a rychlejší komunikace mezi vlákny (stejný paměťový prostor).

I na jednoprocesorových systémech vícevláknové aplikace lépe využívají CPU.

Nevýhody:

Distribuce výpočetních vláken na různé výpočetní systémy(počítače).

151. Je aplikace s interaktivním rozhraním vhodným kandidátem pro vícevláknovou aplikaci a proč?

- ano
- výpočetně náročné procesy tím nesníží interaktivitu aplikace
- okamžitá odpověď aplikace zlepšuje uživatelovu práci s aplikací
- uživatelova práce vytváří 'event' jenž má efekt na aplikaci

152. Kdy nemá smysl použití více vláken pro aplikaci s uživatelským rozhraním?

-

153. Má smysl vyvíjet vícevláknové aplikace pro systémy s jediným CPU a proč?

- ano, stále je možné, že lépe využijí výpočetní výkon (multi-threading)
- využití paralelního vykonávání instrukcí může zrychlit aplikaci, lepší využití CPU

154. Kde se mohou nacházet vlákna z hlediska řízení přidělování procesoru?

155. Jak jsou rozvrhována vlákna řešená uživatelskou knihovnou a co to znamená z hlediska priority vláken?

156. Jaké modely vícevláknových aplikací znáte?

- boss/worker
- pipeline
- peer
- producer/consumer

157. Co je to Thread Pool a k čemu je dobrý?

- zásobárna předpřipravených vláken, která čekají na úkoly od hlavního vlákna. Nemusí se díky němu neustále inicializovat nová vlákna.

158. Jak snížíte nároky opakovaného vytváření vláken?

– využitím Thread Pool

159. Jaké vlastnosti sledujeme při návrhu struktury Thread Pool?

– počet předpřipravených vláken

– maximální počet požadavků ve frontě požadavků

– co se má stát, když je fronta plná a žádné z vláken není k dispozici

160. Jakou architekturu vícevláknové aplikace použijete v případě zpracování proudu dat?

- pipeline

161. Jaké vlastnosti musí splňovat proudové zpracování dat, aby bylo výhodné použít více vláken?

- předpoklady: proud dat, určitá vlákna pracují paralelně na různých částech ‘proudu’ dat

162. Jak předáváme data mezi vlákny v úloze producent/konzument?

- pomocí memory bufferu nebo jen pomocí bufferu referencí (pointerů) na konkrétní datové jednotky

163. Jaké je základní primitivum synchronizace více vláken?

- pthread_mutex_lock, pthread_mutex_unlock

164. Jaké znáte primitiva pro synchronizaci více vláken?

- pthread_cond_signal, pthread_cond_wait, pthread_cond_broadcast

165. Kdy říkáme, že je funkce reentrantní?

- když můžeme být funkce v průběhu přerušena a pak bezpečně znova zavolána, nepíše do static data a nepracuje s globálními daty
- bezpečné pro paralelní volání

166. Co je to thread-safe funkce?

- thread-safe funkce: jedno či více vláken může vykonat stejnou část kódu bez toho aby způsobily synchronizační problémy

167. Jak dosáhneme reentrantní funkce?

– tak, že tato funkce nebude zapisovat do static data a nepracuje s globálními proměnnými

168. Jak dosáhneme thread-safe funkce?

– tak, že tato funkce bude mít přímý přístup ke globálním datům za použití synchronizačních primitiv

169. Jaké hlavní synchronizační problémy se objevují u vícevláknových aplikací?

- deadlock, race condition

170. Co je to problém uváznutí (deadlock)?

- když pro dokončení první operace je potřeba dokončit operaci druhou a naopak → zacyklení se, čekají jedna na druhou

```
mutex2.lock();
printf(“%d”, 5);
mutex1.lock();
mutex2.unlock();
mutex1.unlock();
```

171. Co je to problém souběhu (race conditions) u vícevláknové aplikace?

- jedná se o přístup více vláken ke sdílenému zdroji, kdy alespoň jedno z nich nepoužívá synchronizační mechanismus

vlákno čte hodnotu, zatímco jiné vlákno zapisuje hodnotu, v momentě, kdy operace zápisu a čtení nejsou atomické (jsou vytvořeny s rizikem, že mezi zápisem a čtením dojde k přepsání)

172. Jak se lze vyhnout problému uváznutí (“dead-lock”) u vícevláknové aplikace?

- zamykat proměnné vždy ve stejném pořadí (spraví většinu problémů), jinak žádné 100% pravidlo neexistuje

```
/*
 * File name: ukazkovy_test.c
 * Date:      1620/04/20 16:20
 * Author:    Jan Faigl
```

- Jak zjistíme velikost datové reprezentace základních celočíselných typů v jazyce C? *- sizeof(int), sizeof(long)...*
- Jak rozlišíte literál typu int a long? *- pomoci sizeof() to nepujde. int a long int maji bud stejny pocet bajtu - obvykle 4, nebo long je delsi - 8 bajtu. Jedinou podminkou je ze long nikdy nebude mensi jak int. Takze nekdy je muzeme rozlisit pomoci sizeof()*
- Jak vkládáme do zdrojového souboru programu v C hlavičkové soubory jiných modulů nebo knihoven? *- #include<lib.h> nebo #include”lib.h”*
- Jaké znáte znaky používané v C pro řízení výstupu?

  d, i *Celé číslo se znaménkem (Zde není mezi d a i rozdíl. Rozdíl viz scanf() níže).*
  u *Celé číslo bez znaménka.*
  o *Číslo v osmičkové soustavě.*
  x, X *Číslo v šestnáctkové soustavě. Písmena ABCDEF se budou tisknout jako malá při použití malého x, nebo velká při použití velkého X.*
  p *Ukazatel (pointer)*
  f *Racionální číslo (float, double) bez exponentu.*
  e, E *Racionální číslo s exponentem, implicitně jedna pozice před desetinnou tečkou a šest za ní. Exponent uvozuje malé nebo velké E.*
  g, G *Racionální číslo s exponentem nebo bez něj (podle absolutní hodnoty čísla). Neobsahuje desetinnou tečku, pokud nemá desetinnou část.*
  c *Jeden znak.*
  s *Řetězec.*
- Deklarujte pole variabilní délky s velikosti n, kterou načtete ze standarního vstupu. *-variable length array. scanf(“%d”,&n); int s=n; int pole[s];*
- Jak v C definujete ukazatel na proměnné, např. typu int? *- A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.*

- Je v jazyce C nutné uvolňovat dynamicky alokovanou paměť? Pokud ano, jak to uděláte? - *ano.Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. free(ptr);*

Součástí písemného testu jsou také implementační otázky např.

Opravte implementaci spojového seznamu.(linked list)

```
struct Node
{
        void  *data;
        struct Node *next;
};
```

```
/* Function to add a node at the beginning of Linked List*/
void push(struct Node** head_ref, void *new_data, size_t data_size)
{
   // Allocate memory for node
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   new_node->data  = malloc(data_size);
   new_node->next = (*head_ref);
    // Copy contents of new_data to newly allocated memory.
   for (int i=0; i<data_size; i++)
      *(char *)(new_node->data + i) = *(char *)(new_data + i);
    // Change head pointer as new node is added at the beginning
   (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list.  */
void printList(struct Node *node, void (*fptr)(void *))
{
   while (node != NULL)
   {
      (*fptr)(node->data);  //funkce fptr je funkce kterou naapsiseme pro vypisovani
      node = node->next;
   }
}

/* Driver program to test above function */
int main()
{
```

```
   struct Node *start = NULL;
   // Create and print an int linked list
   unsigned int_size = sizeof(int);
   int arr[] = {10, 20, 30, 40, 50}, i;
   for (i=4; i>=0; i--)
     push(&start, &arr[i], int_size);
   printf("Created integer linked list is \n");
   printList(start, printInt);

   // Create and print a float linked list
   unsigned float_size = sizeof(float);
   start = NULL;
   float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
   for (i=4; i>=0; i--)
     push(&start, &arr2[i], float_size);
   printf("\n\nCreated float linked list is \n");
   printList(start, printFloat);

   return 0;}
```

Napište program, který vygeneruje uvedený obrazec.

Napište program pro realizaci zásobníků celých čísel (`int`) jako třídu `Stack` s metodami `push()` a `pop()`.

```
struct Stack
{
   int top;
   unsigned capacity;
   int* array;
};

// function to create a stack of given capacity. It initializes size of stack as 0
struct Stack* createStack(unsigned capacity)
{
   struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
   stack->capacity = capacity;
   stack->top = -1;
   stack->array = (int*) malloc(stack->capacity * sizeof(int));
   return stack;
}
```

```
// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1;  }

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

int main()
{
    struct Stack* stack = createStack(100);
     push(stack, 10);
    push(stack, 20);
    push(stack, 30);
    printf("%d popped from stack\n", pop(stack));
     return 0;
}
```

Uveďte příklad implementace třídy pro komunikaci po sériovém portu dle RAII (Resource acquisition is initialization). HELP

Kdyz chceme ziskat nejaky objekt?soubor? tak ho musime inicializovat, kdyz ho chceme vyprazdnit, tak ho musime smazat.

```
#include <cstdio>
#include <stdexcept>
  class file {
public:
   file( const char* filename ) : m_file_handle(std::fopen(filename, "w+"))
   {
      if( !m_file_handle )
         throw std::runtime_error("file open failure") ;
   }
   ~file()
   {
      if( std::fclose(m_file_handle) != 0 )
       {
              }
   }

   void write( const char* str )
   {
      if( std::fputs(str, m_file_handle) == EOF )
         throw std::runtime_error("file write failure") ;
   }

private:
   std::FILE* m_file_handle ;
   file( const file & ) ;
   file & operator=( const file & ) ;
};

void example_usage() {
     file logfile("logfile.txt") ;
   logfile.write("hello logfile!") ;
}
```

///----------Poznamky----------

1) Interpretovany jazyk - u nehoz je pro spusteni potreba zdrojovy kod a specialni program zvany interpret. Cecko NENI Interprenovany jazyk. Je ho potreba nejdriv prelozit do strjoveho kodu -> to je spustitelny soubor, ktery muze primo provadet processor pocitacu.
2) **Linked List (spojovy seznam):** It is a collection of NODES, where node is a memory block which is divided into two parts:INFORMATION field and ADDRESS field. Information field holds the data part and Address part holds the address of the next node.

Stack (zasobnik): It is a linear List of elements, elements can be inserted or deleted only at one end called the TOP of the stack. It uses the concept of LIFO (Last In First Out).
Queues (fronta): It is the list of elements, element can be inserted only at one end called the REAR end and element can be deleted only at other end called the FRONT end. It uses the concept of FIFO (First In First Out)

Priority queue: is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.
Circular Queue: is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.
Trees: It is a non linear Data Structure,which is hierarchical in nature and must maintain a hierarchy.
Graphs: It is a non linear data structure in which it is not necessary that the relationship between the data items is in hierarchy.
*/

Q: *Co udělá* `realloc()` *při zmenšení velikost nebo nastavení velikosti na 0?*

-Zmenší alokované místo a v případě 0, např. `realloc(ptr, 0);` uvolní paměť a odpovídá tak volání `free(ptr);`.

Q: *Je nutné nebo vhodné explicitně typovat ukazatel návratové hodnoty z volání funkce* `malloc`*()?*

-Vyloženě nutné to v současných verzích Cčka není, přestože pro některé kompilátory (zvláště pak před standarem) to nutné bylo. V současné době je typ `void*` chápan jako generický ukazatel, jehož přetypování na konktrétní typ ukazatel na proměnné příslušné typu je zřejmé dle typu proměnné a není tak nutné explicitní přetypování uvádět.

Řízení vykonávání jednotlivých instrukcí.
SIMD (single-instruction, multiple-data) - stejné instrukce jsou vykonávány na více datech. Procesory jsou identické a pracují synchronně.
MIMD (multiple-instruction, multiple-data) - procesory pracujínezávisle a asynchronně.
Řízení přístupu k paměti.
Systémy se sdílenou pamětí - společná centrální paměť.
Systémy s distribuovanou pamětí - každý procesor má svou paměť.

Základní pojmy
Paralelismus - Vytváření souběžnosti, zdánlivý nebo skutečný paralelní běh více procesů zároveň.
Skutečný (paralelismus) - Každý proces má svůj vlastní procesor, v praxi téměř nemožné.

Zdánlivý (pseudoparalelismus) - Několik procesů se navzájem dělí o časové kvantum na jednom CPU, takto funguje drtivá většina moderních systémů.
Multitasking - Schopnost operačního systému provádět (přinejmenším zdánlivě) několik procesů současně.
Program - Realizace algoritmu v programovacím jazyce, pro interpretované jazyky je vyskytuje ve formě spustitelného skriptu a pro kompilované jazyky jako spustitelný binární soubor.
Proces - Spuštěný program, který OS zavedl do operační paměti a přidělil mu určitá práva a vyhrazenou pamět.
Vlákno - Při vytvoření nového procesu je automaticky vždy vytvořeno primární vlákno. Vlákno je objektem OS, ve kterém běží samotný programový kód. Všechny vlákna v rámci procesu sdílí tentýž virtuální adresní prostor.
V rámci jednoho procesu může běžet několik vláken, ty mají sdílenou pamět a při jejich střídání nemusí docházet ke změně kontextu. Díky tomu je mezi vláknová komunikace jednoduší a střídání vláken je rychlejší. Použití oddělených procesu se používá například z bezpečnostních důvodů k omezení přístupu do sdílené paměti.
/* end of ukazkovy_test.c */