AS-0.1103 Basic course in C-programming
Aalto University School of Electrical Engineering
Exam of 05.01.2012, Raimo Nikkilä

Instructions

All of the programming questions are to be answered in the C programming language and abiding to a good programming style. The exam is graded on a scale of 0 to 25 points with each individual question worth 0 to 5 points. Answer any **five** questions of your choice. During the exam you are allowed to use a scientific non-programmable calculator and the provided C99 mini reference. All of the questions in the exam can be solved with the functions listed in the mini reference but you are allowed to use all functions of the C99 standard library.

In every question a byte (`char`) has exactly 8 bits.

Please write at least your student number in legible hand.

Question 1

What does the following program print? The output itself, without any additional reasoning, is sufficient as an answer.

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

size_t function(int* arr, size_t n)
{
  size_t w = 0;
  for (size_t r = 0; r < n; r++)
    if (arr[r] % 2)
      arr[w++] = arr[r];
  return w;
}

int main(void)
{
  int arr1[6] = {1,2,3,4,5,6};
  int arr2[6] = {-1,-2,-3,-4,-5,-6};

  int **arr = malloc(4 * sizeof(int *));
  arr[0] = arr1;
  arr[1] = arr2;
  arr[2] = arr1 + 3;
  arr[3] = arr2 + 3;

  printf("A:");
  for (size_t i = 0; i < 3; i++)
    printf(" %d", arr[0][i] + arr[2][i]);
  printf("\n");

  printf("B: %ld %d\n", arr[1] - arr[3], arr[3][0]);

  printf("C: %d %d\n", **(arr + 1), *(arr[2] + 2));

  printf("D:");
  size_t n = function(arr[0], 6);
  for (size_t i = 0; i < n; i++)
    printf(" %d", arr[0][i]);
  printf("\n");

  printf("E:");
  for (size_t i = 0; i < 6; i++)
    printf(" %d", arr1[i]);
  printf("\n");

  free(arr);
}
```

Question 2

Morse code is a simple way to transmit information using only two characters, the dot and the dash.

Implement the function `morseEncode()`, which performs morse coding with bits so that each character of the morse code is represented using two bits.

```
1 unsigned char* morseEncode(const char* str);
```

This function takes the morse code as a character string containing the characters '-', '.' and ' ', and produces the bit encoding of this string.

Normally, morse code would use breaks to indicate spaces, however, with a bit encoding the space requires a distinct code. Likewise, the end of the message is given a distinct code.

Implement the morse code using the following encoding:

Character	Encoding bits
Dash	10
Dot	11
Space	01
End of message	00

- You can assume that calls to functions `malloc()`, `calloc()` and `realloc()` always succeed.
- You can assume that the parameter string contains no additional characters.
- Any leftover bits at the end of the encoding are set to zero.

An example of the encoding:

```
morseEncode("-- --- .-. ... ."); // MORSE
Tuottaa koodauksen: {0xA6, 0xA7, 0xB7, 0xF7, 0x00}
```

A	.-	M	--	Y	-.--	6	-....
B	-...	N	-.	Z	--..	7	--...
C	-.-.	O	---	Ä	.-.-	8	---..
D	-..	P	.--.	Ö	---.	9	----.
E	.	Q	--.-	Ü	..--	.	.-.-.-
F	..-.	R	.-.	Ch	----	,	--..--
G	--.	S	...	0	-----	?	..--..
H	....	T	-	1	.----	!	..--.
I	..	U	..-	2	..---	:	---...
J	.---	V	...-	3	...--	"	.-..-.
K	-.-	W	.--	4	....-	'	.----.
L	.-..	X	-..-	5	.....	=	-...-

The international Morse alphabet

Question 3

Explain briefly and using at most two sentences what each of the three functions listed below generally does.

Do not describe the internal details or the structure of the function, but rather, what functionality the function provides or what value it calculates from the parameters.

```
1  include <string.h>
2
3  void function1(char *str, size_t n)
4  {
5    if (n < strlen(str))
6      str[n] = '\0';
7  }
8
9  void function2(char *str, size_t n)
10 {
11   size_t s = strlen(str);
12   if (n >= s)
13   {
14     *str = '\0';
15     return;
16   }
17   for (size_t i = 0;i + n <= s;i++)
18     str[i] = str[i + n];
19 }
20
21 size_t function3(char **arr, const char *str)
22 {
23   size_t c = 0;
24   for (char** p1 = arr;*p1;p1++)
25     for (const char *p2 = *p1;*p2;p2++)
26     {
27       int f = 0;
28       for (const char *s = str;*s;s++)
29         if (*p2 == *s)
30         {
31           f = 1;
32           break;
33         }
34       c += f;
35     }
36   return c;
37 }
```

Question 4

Implement the function `readMatrix()`, which reads a floating point matrix from a file.

The function takes the name of the file containing the matrix as a parameter.

The function returns a dynamically (`malloc()`) allocated `struct Matrix` structure of the matrix read from the file.

```
1  #include <stddef.h>
2
3  struct Matrix
4  {
5    double **values;
6    size_t nrows; // Rows
7    size_t ncols; // Columns
8  };
9
10 struct Matrix* readMatrix(const char* fname);
```

- You can assume that calls to functions `malloc()`, `calloc()` and `realloc()` always succeed.
- In case the file can not be opened, a `NULL` value is returned.
- First two values of the file are integers which indicate the number of rows and columns in the matrix.
- You can assume that the file is not malformed.

An example of an input file for a matrix of size 2x3:

```
2 3
1.0 -2.5 7.0
0.5 12.0 2.0
```

Question 5

Following is a (somewhat poor) program that handles simple strings of text. **Briefly** answer the following questions based on the given program.

A) What are the other two ways to indicate the necessary size of the allocation on line 12?

B) Why is no `sizeof(char)` factor required on line 13?

C) What obvious flaw or functional restriction is there in the `findText()` function starting from line 22?

D) What about the `removeText()` function starting from line 32?

E) What memory should be freed and when in the `removeText()` function?

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  typedef struct text_s
5  {
6    char *content;
7    struct text_s *next;
8  } Text;
9
10 void addText(Text *txt, const char *s)
11 {
12   Text *new = malloc(sizeof(*new));
13   new->content = malloc(strlen(s) + 1);
14   new->next = NULL;
15   strcpy(new->content, s);
16
17   while (txt && txt->next)
18     txt = txt->next;
19   txt->next = new;
20 }
21
22 const char* findText(Text *txt, const char *s)
23 {
24   while (txt && txt->next)
25     if (!strcmp(txt->content, s))
26       return s;
27     else
28       txt = txt->next;
29   return NULL;
30 }
31
32 void removeText(Text *txt, const char *s)
33 {
34   const char* p = findText(txt, s);
35   if (!p)
36     return;
37   for (;txt->next->content != s;txt = txt->next);
38   Text *tmp = txt->next->next;
39   txt->next = tmp;
40 }
```

Question 6

Let us handle simple floating point functions which take one floating point value as parameter and return a floating point value.

Our purpose is to collect such functions in an array inside a `Functions` data type so that we can call all functions in sequence for some given value.

For this purpose, define the data type `Functions`, which contains at least the function pointer array for the functions described above. Also, implement the functions `addFunction()` and `callFunctions()`.

Function `addFunction()` takes two parameters, an address to a `Functions` data type and one function pointer. This function pointer is added as the last member of the array contained in the data type.

Function `callFunctions()` also takes two parameters, an address to a `Functions` data type and one floating point value. The function calls, in sequence, all functions contained in the array of the data type for this floating point value. Functions are called so that the result of the previous function is given as the parameter to the next function. E.g. for three functions: $f_1(f_2(f_3(parametri)))$; The function returns this result.

- You can assume that calls to functions `malloc()`, `calloc()` and `realloc()` always succeed.
- You may indicate the size of the array in the `Functions` data type as you see fit.

C99 minireference

Data types required in the exam and their corresponding sizes

Type	Size in bytes	Range	I/O format specifier
`char`	1	-127 ... 127	%c
`unsigned char`	1	0 ... 255	%c
`int`	4	-2,147,483,647 ... 2,147,483,647	%d
`unsigned int`	4	0 ... 4,294,967,295	%u %x %X
`long`	8	Sufficiently large $(-2^{63}...2^{63}-1)$	%ld
`size_t`	8	Sufficiently large $(0...2^{64}-1)$	%zu
`double`	8	Sufficiently large	%lf
`void*`	8	All (object) pointers	%p

Operator precedence from highest to lowest and operator associativity

`() [] . → ++ --`	left to right	postfix ++ and --
`++ -- + - ! ~ (type) * & sizeof`	right to left	prefix ++ ja --, * and & as memory operators
`* / %`	left to right	* as an arithmetic operator
`+ -`	left to right	
`<< >>`	left to right	
`< <= > >=`	left to right	
`== != & ^ | && ||`	left to right	& as a logical operator
`?:`	right to left	
`= op=`	right to left	all assignment operators
`,`	left to right	comma as an operator

C99 reserved words

`auto`	`break`	`case`	`char`	`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`	`float`	`for`	`goto`	`if`
`inline`	`int`	`long`	`register`	`restrict`	`return`	`short`	`signed`
`sizeof`	`static`	`struct`	`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`	`_Bool`	`_Complex`	`_Imaginary`			

Functions of the standard library required in the exam

ctype.h

```
int isalnum(int ch);   int isalpha(int ch);   int isdigit(int ch);   int islower(int ch);
int ispunct(int ch);   int isspace(int ch);   int isupper(int ch);   int tolower(int ch);
int toupper(int ch);
```

math.h

```
double pow(double base, double exponent);   double sqrt(double value);
```

stdio.h

```
int printf(const char* format, ...);
int sprintf(char* str, const char* format, ...);
int fscanf(FILE* stream, const char* format, ...);
FILE* fopen(const char* path, const char* mode);
int getchar(void);
int putchar(int ch);
int puts(const char* str);
char* fgets(char* str, int size, FILE* stream);
int fprintf(FILE *stream, const char* format, ...);
int scanf(const char* format, ...);
int sscanf(const char* str, const char* format, ...);
int fclose(FILE* fp);
int fgetc(FILE *stream);
int fputc(int ch, FILE* stream);
int fputs(const char* str, FILE* stream);
```

stdlib.h

```
void* calloc(size_t nmemb, size_t size);   void free(void* ptr);   void abort(void);   int abs(int);
void* realloc(void* ptr, size_t size);   void* malloc(size_t size);   void exit(int);
void qsort(void* base, size_t nmemb, size_t size, int (*cmp)(const void*, const void*));
```

string.h

```
char* strcat(char* dest, const char* src);
int strcmp(const char* str1, const char* str2);
char *strstr(const char* haystack, const char* needle);
size_t strxfrm(char* dest, const char* src, size_t n);
char* strchr(const char* str, int ch);
char* strcpy(char* dest, const char* src);
size_t strlen(const char *str);
void* memset(void* s, int c, size_t n);
```