Exam in EDAA25 C Programming

October 27, 2017, 8-13

The results will be announced by email on Friday November 17 at 15.00

Inga hjälpmedel!

You may answer in English, på svenska, auf Deutsch, или по-русски.

Examiner: Jonas Skeppstedt

30 out of 60p are needed to pass the exam.

Grading instructions

- In general: assess if a function or program works as intended while ignoring syntax errors. That gives full score. Then subtract each syntax error from that score and give at least zero points.
- If the code doesn't work, make a judgement on how serious errors it contains, keeping in mind the obvious that the intent is to give the proper grade.
- If a student makes the same mistake *n* times, the student's score will be reduced *n* times.
- In general −1 per trivial syntax error which can be regarded as a typo.
- In general −2 per serious syntax error which indicates the student needs more practising.

30 out of 60p are needed to pass the exam. You are not required to follow any particular coding style but your program should of course be readable. If you call a function which may fail, such as `malloc`, you must check whether the call succeeded or not. You do not have to include header files.

1. (10p) **Strings.**

   Implement the standard function

```
#include <string.h>
char* strchr(const char* s, int c);
```

   which first converts `c` to a `char` and then searches the string pointed to by `s` for an occurrence of this value and a pointer to the first such occurrence is returned or a null pointer if `c` was not found. The terminating null character is regarded as part of the string.

   Answer:

```
char* strchr(const char* s, int c)
{
        char    ch;
        char*   t;

        t = (char*)s;
        ch = c;

        for (;;) {
                if (*t == ch)
                        return t;
                else if (*t == 0)
                        return NULL;
                else
                        t += 1;
        }
}
```

   - *The* `int` *parameter must be converted to a* `char` *otherwise the function will fail when* `c` *is outside the range of* `char`*.*
   - *Note that* `c` *may be zero (taken care of in* `if`*).*
   - *The cast is needed but no points of removed from the solution if forgotten.*

2. (20p) Using the following type definitions:

```
typedef struct list_t list_t;

struct list_t {
        list_t* succ;
        list_t* pred;
        void*   data;
};
```

Implement a circular double linked list of void pointers, where an empty list is represented by `NULL` and with the following functions:

- (3p) `list_t* new_list(void* data);`
  It should create a list with one element.
- (3p) `void free_list(list_t* list);`
  It should deallocate all memory for the entire list but not for the data put in the list.
- (3p) `void append(list_t** list1, list_t* list2);`
  It should concatenate the two lists.
- (3p) `void add(list_t** list, void* data);`
  It should add the data to the end of the list (i.e. by creating a new list).
- (4p) `void reverse(list_t** list);`
  It should reverse the list and modify what the parameter points to so that it will point to the new start of the list. You are **not permitted** to allocate new memory from the heap (i.e. getting memory from `malloc/calloc/realloc`). Using a VLA (variable length array) is also forbidden in this question.
- (4p) `void print(list_t* list, void (*func)(void*));`
  It should print out each element stored in the list.

Your implementation should give an error message if the heap has no memory left.The following code shows an example of how a list can be used.

```
#include <stdio.h>
#include "list.h"

void print_string(void* s)
{
        printf("%s", s);
}

int main(int argc, char** argv)
{
        list_t* list;

        list = new_list("a");
        add(&list, "b");
        add(&list, "c");

        print(list, print_string);
        reverse(&list);
        print(list, print_string);
        free_list(list);

        return 0;
}
```

The output from the above program could for instance be:

```
a b c
c b a
```

Answer:

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

typedef struct list_t list_t;

struct list_t {
        list_t*         succ;
        list_t*         pred;
        void*           data;
};

void* xmalloc(size_t s)
{
        void*   p;

        p = malloc(s);

        if (p == NULL) {
                fprintf(stderr, "out of memory\n");
                exit(1);
        } else
                return p;
}

list_t* new_list(void* data)
{
        list_t*         p;

        p = xmalloc(sizeof(list_t));

        p->succ = p->pred = p;
        p->data = data;

        return p;
}

void free_list(list_t* list)
{
        list_t*         p;
        list_t*         q;

        if (list == NULL)
                return;

        p = list;

        p->pred->succ = NULL;
```

```
	while (p != NULL) {
		q = p->succ;
		free(p);
		p = q;
	}
}

void append(list_t** list1, list_t* list2)
{
	list_t*		p;

	if (*list1 == NULL)
		*list1 = list2;
	else if (list2 != NULL) {
		(*list1)->pred->succ = list2;
		list2->pred->succ = *list1;
		p = (*list1)->pred;
		(*list1)->pred = list2->pred;
		list2->pred = p;
	}
}

void add(list_t** list, void* data)
{
	append(list, new_list(data));
}

void reverse(list_t** list)
{
	list_t*		h;
	list_t*		p;
	list_t*		q;
	list_t*		r;

	h = *list;

	if (h == NULL)
		return;

	p = h->pred;

	*list = p;

	q = h;

	do {
		r = q->succ;
		q->succ = p;
		q->pred = r;
		p = q;
		q = r;
	} while (q != h);
}
```

```
void print(list_t* list, void (*func)(void*))
{
        list_t*         p;
        list_t*         h;

        p = h = list;

        if (h == NULL)
                return;

        do {
                (*func)(p->data);
                p = p->succ;
                if (p != h)
                        putchar(' ');
        } while (p != h);
        putchar('\n');
}
```

- *Although it is not permitted to use a VLA in* `reverse` *it is permitted to use recursion (despite being more memory consuming).*
- *There can be different goals with an implementation of* `reverse` *and the purpose of this one is to reduce the number of memory accesses to one read and two stores per loop iteration.*

3. (20p) Give brief descriptions of the following.

   (a) arithmetic shift right
   (b) bit-field
   (c) compound literal
   (d) corresponding type
   (e) function scope
   (f) flexible array member
   (g) implementation defined behaviour
   (h) undefined behaviour
   (i) unspecified behaviour
   (j) variable length array

4. (10p) **Integer arithmetic**

   What are the values of the following expressions?

   Hint: `1` has type `signed int`, `1U` has type `unsigned int`, and `1L` has type `signed long`.

   (a)-(f) each gives one point if answered correctly without motivation, and (g) and (h) each gives two points if answered correctly with a motivation.

   (a) `11 & 7`

   Answer: 3

   (b) `-5 / 3`

   Answer: -1

   (c) `5 % 3`

   Answer: 2

   (d) `-5 % 3`

   Answer: -2

   (e) `5 % -3`

   Answer: 2

   (f) `-5 % -3`

   Answer: -2

   (g) `-1 / 3U > 3`

   Answer: 1, note that -1 is converted to unsigned and becomes a large number.

   (h) `-1L / 3U > 3`

   Answer: implementation-defined. It depends on the relative sizes of the types `long` and `unsigned`. If `long` can represent all values of `unsigned` then the unsigned operand is converted to the type of the signed operand (and the value becomes zero of the expression), and if it cannot, the signed operand is converted to an unsigned type and the number becomes large (and the value of the expression becomes 1).