# Exam in EDAA25 C Programming

## October 18, 2011, 8-13

Inga hjälpmedel!

Examinator: Jonas Skeppstedt

## 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, judge youself on how serious error 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.
- See specific notes for each question below.

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.

1. (5p) Implement the function

   ```
   size_t strlen(const char *s);
   ```

   which returns the length of the argument string.

   *Answer: see book. Zero points if it computes the wrong answer.*

2. (5p) Implement the non-standard function

   ```
   char* strdup(const char *s);
   ```

   *Answer:*

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

char* strdup(const char* s)
{
        char*   t;

        t = malloc(strlen(s) + 1);
        if (t != NULL)
                strcpy(t, s);
        return t;
}
```

- *It's OK to use a loop instead of* `strcpy`*.*
- $-3$ *if* `strcat` *is used — why should any?*
- $-3$ *if the return value from* `malloc` *is not checked.*
- $-3$ *if too little memory is allocated.*
- $-3$ *if too much memory is allocated.*

which copies a string using memory from the heap, and returns the copy or a null pointer if the memory allocation failed.

3. (20p) Write a program which counts the number of times each unique word appears in the input read using `getchar`. A word is a sequence of characters for which `isalpha` returns a nonzero value. You cannot assume any maximum size of the input lines or the length of a word or the number of times a word appears in the input, except that these numbers can be represented by the type `size_t`.

   The function

   ```
   int strcmp(const char *s1, const char *s2);
   ```

   is probably useful and it returns zero if the two string parameters point to strings with the same contents. There are no requirements on efficiency except that you are not allowed to allocate memory from the heap for **each** input character read, and you are not allowed to leak memory. However, for simplicity, if a heap memory allocation fails, you are not required to deallocate any memory but rather can print an error message and terminate the program by calling `exit(1)`.

   When the program has reached end-of-file, it should print out the words and their counts — but they don't have to be sorted.

   You get **zero points** if you use any variable with static storage duration (such as a global variable) in this question.

With the input:

```
abc ab AB ab ?; ab
c +
```

the output should be for example:

```
1       abc
3       ab
1       AB
1       c
```

*Answer:*

- *It's OK to write everything in* `main`.
- *It's OK to call* `exit(1)` *if* `malloc` *or* `realloc` *return a null pointer.*
- *Start with 20 points and reduce each error as described above.*
- *Check that they print out a value with type* `size_t` *using* `%zu`.
- *Give* $-4$ *per forgotten* `free`.
- *They don't have to include any header files.*

```c
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INITIAL_SIZE    (4)             /* Size of buffer for storing a word. */

typedef struct list_t   list_t;

struct list_t {
        size_t          count;
        char*           word;
        list_t*         next;
};

void error(const char* s)
{
        fprintf(stderr, "error: %s\n", s);

        exit(EXIT_FAILURE);
}
```

```
void out_of_memory(void)
{
        error("out of memory");
}

static list_t* new_list(char* word)
{
        list_t*         p;

        p = malloc(sizeof(list_t));

        if (p == NULL)
                out_of_memory();

        p->count        = 0;
        p->word         = word;
        p->next         = NULL;

        return p;
}

static void insert_first(list_t* h, list_t* p)
{
        p->next = h->next;
        h->next = p;
}

static void free_list_and_data(list_t* h)
{
        list_t*         p;
        list_t*         q;

        p = h;

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

```
static list_t* lookup(list_t* h, char* word)
{
        list_t*         p;

        p = h->next;

        while (p != NULL)
                if (strcmp(p->word, word) == 0)
                        break;
                else
                        p = p->next;

        return p;
}

int main(void)
{
        list_t*         h;      /* list head. */
        list_t*         p;      /* list node with a word. */
        int             c;      /* input character. */
        char*           s;      /* start of word buffer. */
        char*           t;      /* current position in word buffer. */
        size_t          size;   /* size of word buffer. */
        size_t          n;      /* characters used in word buffer. */
        bool            inword; /* true if we are in a word. */

        h = new_list(NULL);

        if (h == NULL)
                error("out of memory");

        size = INITIAL_SIZE;
        t = s = malloc(size);
        n = 0;

        if (s == NULL)
                out_of_memory();

        inword = false;

        while ((c = getchar()) != EOF) {

                if (isalpha(c)) {
```

```
                inword = true;

                *t++ = c;
                n += 1;

                if (n+1 == size) {
                        char*   s0;

                        size *= 2;
                        s0 = realloc(s, size);
                        if (s0 == NULL)
                                out_of_memory();
                        s = s0;
                        t = s + n;
                }

        } else if (inword) {

                *t = 0;
                n += 1;

                p = lookup(h, s);

                if (p == NULL) {
                        t = malloc(n);
                        if (t == NULL)
                                out_of_memory();

                        memcpy(t, s, n);

                        p = new_list(t);
                        if (p == NULL)
                                out_of_memory();

                        insert_first(h, p);
                }

                p->count += 1;
                inword = false;
                n = 0;
                t = s;
        }
}
```

```
        for (p = h->next; p != NULL; p = p->next)
                printf("%10s %zu\n", p->word, p->count);

        free_list_and_data(h);

        free(s);

        return 0;
}
```

4. (5p) What is the difference between a `struct` and a `union`? Declare an example of a `struct` which contains a `union`.

   *Answer:*

   - *See book. 3 points for a correct explanation and 2 points for a correct declaration.*

5. (5p) What does the `volatile` type qualifier mean and when should it be used?

   *Answer: It means that the compiler cannot make any optimizations due to the data may change in ways that the compiler cannot see. All accesses must therefore be performed. It should be used for instance when data is modified by a signal handler. Note that it has a different meaning in Java.*

6. (10p) A recent post in the Usenet newsgroup comp.std.c suggested a change in C to make it possible to pass constant arrays as arguments to functions:

   *P.S. Would it be too much to be able to pass constant arrays to functions directly?*

   `C99:`

   ```
   int xs[] = {1, 2, 3};
   print_integers(xs);
   ```

   `C1X:`

   ```
   print_integers({1, 2, 3});
   ```

   The suggested syntax is invalid C but the requested feature already exists in C99. How can the call to `print_integers` be written without declaring the array identifier?

*Answer:*

*Either:*

```
print_integers((int []){ 1, 2, 3 });
```

*or:*

```
print_integers((int [3]){ 1, 2, 3 });
```

*Putting the array in a struct is not necessary and gives −2 points. Give zero points for instance for:*

```
print_integers((int*){ 1, 2, 3 });
```

7. (10p) Define a macro `STATIC_ASSERT(expr)` which takes an expression as parameter and generates a diagnostic message (i.e. ”error message”) if the expression is zero. It should be possible to use the macro multiple times in a translation unit, and the macro should with high probability somehow identify the source line number with the failed assumption. The C Standard does not *require* any particular output when a diagnostic message is printed but all reasonable compilers of course indicate which source file and line number contains an error when a diagnostic message is required.

   It should be possible to use the macro as follows:

```
#define STATIC_ASSERT(expr)  /* ... */

int main(void)
{
        STATIC_ASSERT(sizeof(char) < sizeof(short));
        STATIC_ASSERT(sizeof(void*) == sizeof(long long));

        /* ... */
}
```

   *Answer: see book. The idea is to use a conditional expression such as in:*

```
typedef int     array[ (expr) ? 1 : -1 ];
```

   - *A typedef should be used to avoid wasting memory by actually declaring an array.*
   - *Since it should be possible (according to the above example) to use* `STATIC_ASSERT` *multiple times, it’s necessary to use different names for the typedef-name, which can be accomplished by concatenating the current line number with the identifier, as in:*

```
#define PASTE(a,b)      a##b
#define EXPAND(a,b)     PASTE(a,b)
#define STATIC_ASSERT(expr) typedef char EXPAND     \
        (failed_static_assertion_in_line_,__LINE__) \
        [(expr) ? 1 : -1]
```

- *Give 2 points for a limited solution which uses the* `?:` *trick.*
- *Give 3 additional points if a typedef is used.*
- *Give 2 additional points if* `##` *is used but slightly wrongly.*
- *Give 10 points if the solution is completely correct.*