Exam in EDAA25 C Programming

October 22, 2013, 8-13

Inga hjälpmedel!

Do not write answers to multiple questions on the same paper.

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 yourself 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. (10p) Implement the standard function

   ```
   #include <string.h>

   void* memchr(const void* s, int c, size_t size);
   ```

   which first converts `c` to an `unsigned 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 in the first `size` characters.

   For instance, with the call:

   ```
   char*   s;

   s = memchr("abc", 'b', 3);
   ```

   `s` will point to the string `"bc"`

   Answer *See code on the next page. Note that since a* `char` *may be either signed or unsigned, that type is not sufficient, and that we cannot dereference a* `void`*-pointer.*

```
#include <stddef.h>

void* memchr(const void* s, int c, size_t size)
{
        unsigned char           ch = c;
        const unsigned char*    t = s;
        size_t                  i;

        for (i = 0; i < size; ++i)
                if (t[i] == ch)
                        return (void*)&t[i];

        return NULL;
}
```

2. (5p) Write a function `print` which prints an `int` in base 10. For instance, with the value 124 stored in an `int` the output should be the equivalent of:

   ```
   printf("%d\n", 124);
   ```

   You are not allowed to use `printf` or a similar function, only the function `putchar` which takes an `int` parameter with a character code, such as `'1'`, `'2'`, `'4'` and `'\n'`.

   Answer *See code on the next page. The basic idea is to print a positive number recursively one digit at a time — see the function* `rec_print_int`*. For negative numbers, we need to be more careful than just printing a minus sign followed by printing the negation of the number. Assume an integer is an N-bit number (the standard defines the integer types in ranges and not bits but this simplifies the explanation). If the compiler uses two's complement representation of signed integers, the range of values of the type* `int` *is from -2^{N-1} to $2^{N-1}-1$. For example, with $N = 32$, the range is from -2147483648 to 2147483647. Thus we cannot negate the smallest integer. If the compiler uses one of the other two valid representations of integers, according to ISO C, the smallest integer in the above example would be -2147483647.*

   The same situation applies to other signed integer types such as `signed char` *which usually has the range from -128 to 127.*

   So we need to treat the smallest integer in a special way for two's complement. First we need to detect that we have the smallest integer and that the compiler actually uses two's complement. This can be done by comparing the input with `INT_MIN` *and* $\texttt{INT\_MIN}+1$ *with* $-\texttt{INT\_MAX}$*.*

   Having determined that we want to print `INT_MIN` *and we have two's complement, we thus cannot negate the input. Instead we do as follows. We note that we want to print a minus sign followed by the mathematical value of* $\texttt{INT\_MAX}+1$*. Obviously, we cannot calculate that number since it would overflow (and it's not certain we can use for instance* `long` *or* `long long` *since the ISO C standard only guarantees minimal ranges but does not forbid all integer types having exactly the same ranges.*

   What we can do instead of calculating $\texttt{INT\_MAX}+1$ *is to first print* $\texttt{INT\_MAX}/10$ *and then print* $(\texttt{INT\_MAX} \mod 10)+1$*. The question then is if this last number to print can be 10 which would make our approach wrong. Since* $\texttt{INT\_MAX}+1$ *is a power of two (i.e. not divisible by 10), we have* $(\texttt{INT\_MAX} \mod 10)+1 \leq 9$*.*

   Note: to properly print all numbers except `INT_MIN` *gave 4 out of 5 points and to also handle* `INT_MIN` *gave the full 5 points.*

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

static void rec_print_int(int x)
{
        if (x > 9)
                rec_print_int(x / 10);
        putchar(x % 10 + '0');
}

void print_int(int x)
{
        if (x >= 0)
                rec_print_int(x);
        else {

                putchar('-');

                if (x == INT_MIN && INT_MIN + 1 == -INT_MAX) {

                        /* Two's complement representation.  */

                        rec_print_int(INT_MAX / 10);
                        rec_print_int(INT_MAX % 10 + 1);

                        /* The last call cannot cause a carry (by being 10)
                         * since (in ISO C) INT_MAX+1 == 2**N and therefore
                         * is not divisible by 10.
                         */


                } else {

                        /* For all other cases, including INT_MIN and either of
                         * the other two valid representations of negative
                         * integers, i.e. sign magnitude and one's complement,
                         * we can just print the negated value of x.
                         */

                        rec_print_int(-x);
                }
        }
}
```

3. (20p) Define a single linked list type called `list_t`. A list node should contain (in addition to any other pointer) a pointer declared as `void* data;`. An empty list should be represented by a null pointer. Implement the following functions:

   - (4p) Add data at the end of the list. Your function should take two parameters. The first somehow refering to an existing list (which may be empty) and the second a pointer to `void`.
   - (4p) Deallocate an entire list (but not what `data` points to).
   - (4p) Reverse a list. The reversed list should not be returned – instead the parameter should be changed. Thus it can be used as in:

```
    list_t*         list;

    /* ... */

    reverse(&list);
```

This function may not allocate memory for a new list and only use a constant amount of memory for local variables (i.e. don't use a variable length array to hold copies of the data pointers).

- (4p) Concatenate two lists (i.e. append the second list to the end of the first list). One or both lists may be empty.
- (4p) A function `apply` with the following behavior: for each data pointer in the list, call a function supplied as an argument to `apply` with that data pointer as the only argument.

  For instance, with a function `print` declared as below, the function `apply` can be used to print each object stored in the list (i.e. each object with a data pointer stored in the list).

```
    list_t*         list;
    void print(void*);

    /* ... */

    apply(list, print);
```

Answer *See book for all functions except* `apply` *which is shown here:*

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

        if (list == NULL)
                return;

        p = list;

        do {
                (*func)(p->data);
                p = p->succ;
        } while (p != list);
}
```

4. (10p) Explain briefly and give an example of each of the following:

- (2p) Bit-field. ***Answer*** *See book.*
- (2p) Compound literal. ***Answer*** *See book.*
- (2p) Flexible array member. ***Answer*** *See book.*
- (2p) Memory leak. ***Answer*** *See book.*
- (2p) `__func__` ***Answer*** *See book.*

5. (5p) Which, if any, of the following declarations in the function u are invalid? Explain!

   Answer for each of a-e either (1) Valid, or (2) Invalid because...

   (No minus points will be awarded.)

```
#include <stdlib.h>

void u(size_t n)
{
        static int      a[n];
        static int      (*b)[n];
        int             (*c)[n];
        int             d[n] = { 0, 1, 2, };
        int             e[] = { 0, 1, 2 };

        /* ... */

}
```

   Answer

   - a *is invalid since the size is unknown at the point of definition.*
   - b *is valid since the size of the pointer is not dependent of the parameter.*
   - c *is valid since it's a normal pointer with variably modified type.*
   - d *is invalid since a variable length array is not allowed to have an initializer.*
   - e *is valid since it's a normal array for which the compiler calculates the size from the list of initializers — it's not a flexible array member since it's not in a struct.*

6. (10p) Implement the standard function

```
#include <string.h>

void* memmove(void* dest, const void* src, size_t size);
```

   which copies `size` characters from `src` to `dest`. These memory areas may overlap, and if they do the behavior is as if the `size` characters first where copied to an additional memory area, and then copied to `dest`. The return value is `dest`.

   You are neither allowed to allocate memory from the heap nor to use a variable-length array, nor any symbol with static storage duration.

   Answer *See code on the next page. Note that we cannot use void pointers for the arithmetic.*

```
void* memmove(void* dest, const void* src, size_t size)
{
        char*           d = dest;
        const char*     s = src;
        size_t          i;

        if (s < d && s + size > d) {

                /* We copy from right to left, otherwise
                 * we will destroy the last part of src.
                 *
                 * src
                 * +------------+
                 * |            |
                 * |            |
                 * |       +----+-------+
                 * +-------|----+       |
                 *         |            |
                 *         |            |
                 *         +------------+
                 *         dest
                 *
                 */

                i = size;

                do {
                        --i;
                        d[i] = s[i];
                } while (i != 0);

        } else {

                /* Either the areas don't overlap or
                 * dest comes before src.
                 *
                 * dest
                 * +------------+
                 * |            |
                 * |            |
                 * |       +----+-------+
                 * +-------|----+       |
                 *         |            |
                 *         |            |
                 *         +------------+
                 *         src
                 */

                for (i = 0; i < size; ++i)
                        d[i] = s[i];
        }

        return dest;
}
```