Example Solution to Exam in EDA150 C Programming

Janurary 12, 2011, 14-19

Inga hjälpmedel!

Examinator: Jonas Skeppstedt, tel 0767 888 124

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

General Remarks

- A function which may crash gets -2 points while normal minor errors get -1 point. For each question you receive at least zero points.
- Memory allocations from the heap should be checked but errors made by programmers using your code should not be checked unless requested. For example, if a list has a special head node which should always be non-null, then your code should not check that it is.

1. (20p) Write code to allocate and deallocate two-dimensional arrays of double precision floating point numbers, setting and getting matrix elements, and to add two such matrices. The number of rows and columns should be specified when a matrix is allocated. The type of the function to add two matrices should be as below and you decide the types of the other functions and the type `matrix_t`

   ```
   matrix_t* add(matrix_t* a, matrix_t* b);
   ```

   Answer:

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

   typedef struct {
           size_t          rows;
           size_t          cols;
           double**        a;
   } matrix_t;

   void error(const char* msg)
   {
           fprintf(stderr, "error: %s\n", msg);
           exit(1);
   }
   ```

```
matrix_t* new_matrix(size_t rows, size_t cols)
{
        matrix_t*       a;
        size_t          i;

        a = malloc(sizeof(matrix_t));

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

        a->rows = rows;
        a->cols = cols;

        a->a = malloc(rows * sizeof(double*));
        if (a->a == NULL)
                error("out of memory");

        for (i = 0; i < rows; ++i) {
                a->a[i] = calloc(cols, sizeof(double));
                if (a->a[i] == NULL)
                        error("out of memory");
        }

        return a;
}

void free_matrix(matrix_t* a)
{
        size_t          i;

        for (i = 0; i < a->rows; ++i)
                free(a->a[i]);
        free(a->a);
        free(a);
}
```

```
matrix_t* add(matrix_t* a, matrix_t* b)
{
        matrix_t*       c;
        size_t          i;
        size_t          j;

        c = new_matrix(a->rows, a->cols);

        for (i = 0; i < c->rows; ++i)
                for (j = 0; j < c->rows; ++j)
                        c->a[i][j] = a->a[i][j] + b->a[i][j];

        return c;
}
```

2. (20p) Give brief descriptions of the following.

   (a) Bit-field
   Answer: see book

   (b) Compound literal
   Answer: see book

   (c) Static storage duration
   Answer: see book

   (d) Variable length arrays
   Answer: see book

   (e) `~` operator
   Answer: see book

   (f) `continue` statement
   Answer: see book

   (g) `inline` function specifier
   Answer: see book

   (h) `_Bool` type
   Answer: see book

   (i) `char` type
   Answer: it's either a signed or an unsigned integer type of size one byte, i.e. **at least** *8 bits wide (that is the definition of a byte in the ISO C Standard). Note that it's neither one of* `signed char` *nor* `unsigned char` *but a distinct type (which means pointer assignments mixing* `char*` *and for example* `signed char*` *are invalid).*

   (j) `a = *b++` expression
   Answer: Read the value pointed to by `b`*, copy it to* `a` *and then increment* `b`.

3. (5p) Suppose you wish to have an array allocated with `malloc` aligned on a 32-byte boundary. How can you achieve that in a portable way? Show code.

Answer: We cannot use pointer arithmetic to achieve alignment so we must cast the pointer to an integer type. With a too narrow type the function will not work correctly and with a too large type it will be slower than needed. The most suitable type is `uintptr_t`*, see book.*

To be able to deallocate the original pointer we must keep the value, hence the last parameter.

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

void* malloc_aligned(size_t n, size_t a, void** ptr)
{
        uintptr_t       b;
        void*           p;

        n += a-1;

        *ptr = p = malloc(n);
        b = (uintptr_t)p;
        b += a-1;
        b &= ~(a-1);
        p = (void*)b;

        return p;
}


int main(void)
{
        void*   p;
        void*   q;

        p = malloc_aligned(128, 32, &q);

        /* ... */

        free(q);

        return 0;
}
```

4. (5p) Array parameters are automatically converted to pointers in C. Why is it so and what can you do if you really want to pass an array to a function that should get its own copy of the array (just like an `int`-parameter is copied and modifiying the parameter does not affect the copied value).

   Answer:

Passing complete arrays can be inefficient and therefore only a pointer to the first array element is passed. To really pass an array, it can be put in a struct which is copied, or at least the compiler produces code which behaves as if it is.

5. (10p) What is a **flexible array member** and why can you not have an array of such structs? If you want to have such structs accessible from an array, how would you do?

   Answer: See the book for an explanation of flexible array members. Since their size is unknown they cannot be part of an array. The array should contain pointers to such structs instead.