C Programming, Autumn 2013, Exercises for the Fifth Week

1. Wite a function

   ```
   double maxi(double x, double y, double (*fp)(double z));
   ```

   which takes three parameters, two double values x and y, and a pointer to a function. The function has a double value as its parameter and it returns a double value. The function `maxi` returns the larger of the function values `(*fp)(x)` and `(*fp)(y)`. Test your function with various parameter functions.

2. Write a function

   ```
   int compareGen(const void *block1, const void *block2,
                  size_t elemSize, size_t block1Size, size_t block2Size,
                  int (*compareIt)(const void * const void *));
   ```

   which performs a lexicographical comparison of the two blocks. This function returns the value of `compareIt` when there is a differing element. Otherwise it return -1, 0 or 1, depending on blocksizes. The given blocksize parameters are given as bytes, as is element size. Test your program using various types of blocks.

3. Write a function

   ```
    void printGen(const void * block, size_t elemSize, size_t blockSize,
                  void (*printIt) (const void *));
   ```

   which prints all the elements in the block using the callback function `printIt`. Test your program with various types, for example a block of doubles and a block of pointers to doubles.

Intrusive Linked List

Because you guys like linked lists so much, we decided to give you another one to implement. :) Let's build an intrusive linked list. An intrusive linked list is an awesome data structure that does not contain data (also called payload) in its nodes. Nodes constist only of pointers to next and prev nodes.

```
typedef struct intrusiveListNode {
    struct intrusiveListNode *next;
    struct intrusiveListNode *prev;
} IntrusiveListNode;
```

You might wonder how to use this list for anything. Well, intrusiveness here means that the list nodes themselves are part of the data that are added to the list. This makes intrusive lists also generic, since the nodes do not really care to what kind of data they belong to. If one wanted to add integers to an intrusive list, this could be done via a wrapper struct such as this:

```
typedef struct wrapInt {
    int i;
    IntrusiveListNode node;
} WrapInt;
```

In addition to the actual data, we want a list type that contains the pointer to the first element. To make operations nice an easy, we use a headered cyclic doubly linked list. This means that we do not have to take into account any nasty special cases! In case you have forgotten, a headered linked list is a linked list where the first node is not a node with any real data, and a cyclic doubly linked list is a list where the last node's next pointer points to the first element (here the header), and the prev pointer of the first node (here the header) points to the last node. Here is the list struct used in this exercise:

```
typedef struct intrusiveList {
    IntrusiveListNode header;
} IntrusiveList;
```

4. Now implement function `create_intrusive_list()` which allocates memory for the list and returns a pointer to that list. Remember to intialize the header node as it should be in a cyclic doubly linked list.

5. Now let's implement

   ```
   add_node(IntrusiveList *list, IntrusiveListNode *node)
   ```

   which takes a node and adds it to the **front** of the list. Remember that the list is cyclic, and be careful not to do weird things to the header node (this node should be inserted to the right side of the header node).

6. Now to do things with the list we also need to implement

   ```
   remove_node(IntrusiveList *list, IntrusiveListNode *node)
   ```

   which removes a given node from the list. Here you should note that since our list is cyclic and headered, you don't need to worry about special cases when updating links. You can simply update next and prev pointers of the appropriate prev and next nodes.

7. Now that we have all basic list operations implemented, let's add actual integers. We'll do this by adding the `WrapInt` structures introduced before. Implement function `add_int(IntrusiveList *list, int i)` that takes as parameter and int value that is wished to add to the list. The function should allocate memory for a WrapInt structure, and add the int to it's appropriate field. This structure should then be added to the list. Adding this structure to the list is done by adding the node of the struct with the operation `add_node` we just implemented. This may seem weird at first, but you will understand how to access the data very soon.

   Now let's address the headache of how to access this data. This might seem like an awfully unclear or impossble operation. Well, C has a cool macro called offsetof, which gives the offset in memory (in bytes) of a field of struct with respect to the address of the struct itself. We do not need to use this macro, however. Instead a macro called `container_of` (stolen and modified from the linux kernel) is given, which uses offsetof:

   ```
   #define container_of(ptr, type, member) (type*)((char*)(ptr) - offsetof(type, member))
   ```

   This macro returns a pointer to the the wrapping struct (typedefed in this exercise, do not get confused). It is given a pointer to the node whose corresponding value you wish to access, the name of the struct (or in our case we can and should use the typdefed name) as a word, not a string and the name of the field containing the node again not as a string. In this case, that field is called node. Let's use these concepts that we just learned.

8. Now that you know how to access data from nodes, let's implement the function
   `remove_int(IntrusiveList *list, int r)` which seeks for a node in the list containing a value equal to r. If there are multiple occurrences, it is enough to remove the first of these. Hint: you know that you have searched the entire list, when the node you are currently inspecting points to the header as the next node. In case it is unclear how `container_of` should be used, here is an example: `WrapInt *wi = container_of(list->header.next, WrapInt, node)`. This call to `container_of` returns the pointer to the `WrapInt` struct of the first node in the list.

   Now to further demonstrate what is so cool about this list and also to introduce the important concept of union, let's add ints and doubles to the same list. We will wrap these types in Numeric, which is a union type. A union type is a type that reserves enough memory to fit the largest possible value it should hold. These values are defined inside the union block. In a union

   ```
   typedef union numeric {
       int i;
       double d;
   } Numeric;
   ```

a variable of that union type can be used as both integer and double. Here is an example of usage.

```
    Numeric num;
    num.i = 10;
    printf(%i", num.i); // This prints 10
    num.d = 10.1; //
    pritnf("%i", num.i); // This prints 858993459
    printf("%lf", num.d); // This prints 10.100000
```

So you see that by using the correspondingly typed field of the union, we treat the union variable as a variable of that type. Note that just like in the example if you assign to any field of a union type, values of all types will be affected, since they use the same block of memory. Now we can wrap this union type in a wrapper that also contains an extra field called type, which contains an identifier of the type of the variable that the union should be treated as. This frees the programmer from headaches when trying to figure out what was the type of this node's value. Possible types are defined in macros INT_TYPE and DOUBLE_TYPE in the header file. To save some memory, we'll put these in an `uint8_t` field. Here is the wrapping struct also found in the header file:

```
typedef struct wrapNumeric {
    uint8_t type;
    Numeric num;
    IntrusiveListNode node;
} WrapNumeric;
```

9. Let's implement a function `add_numeric(IntrusiveList *list, Numeric numi uint8_t type)` that allocates memory to a WrapNumeric strucutre and adds it to the list, much in the same way as with `add_int`. Remember to also set given type in the allocated wrapper.

   By checking the type from the type field, we can infer as which type (int or double) we should treat each element in the list. Let's practice the use of this, and the union type by implementing two functions: one for computing the sum of all int variables in the list, and another for computing the sum of all double variables in the list.

10. Implement functions `int_sum(IntrusiveList *list)` and `double_sum(IntrusiveList *list)`. These function should go through each element in the list. Again you can conclude that the whole list has been gone through once you are back to the header node. Now for each value you have to check what is the type of the value corresponding to each node. This can be done by checking the type field of the struct that you access via `container_of`. If the node is of the correct type, then you can include the value of the union type (as that type) in the sum.

    Now you should know something about intrusive lists, which are a good way to produce generic lists. Intrusive lists are useful because they keep memory allocation away from the actual list operations. Also it does not involve expensive copying operations, when adding new nodes. You should also know something about union types. If you are still confused about union types, have a look at for example http://www.tutorialspoint.com/cprogramming/c_unions.htm