CSE 3100 Systems Programming
Lab #5

Welcome to Lab 5! At the start of each lab, you will receive a document like this detailing the tasks you are to complete. Read it carefully. Try your best to finish labs during the lab session. However, labs are not due right after each lab period.

1 Union

A *union* is a variable that may hold (at different times) objects of different types and sizes (though it holds *one* at a time). For example, we have provided you with a union called `Values`:

```
typedef union Values {
   char ch;
   unsigned int uint;
   int integer;
} Values;
```

We have also provided a `struct` called `Variant`, which holds a `Values`, and an `int` called the *tag*, which stores what kind of data is being stored in `Values`:

```
typedef struct Variant {
   int tag;
   Values val;
} Variant;
```

Note the `define` at the top of `union.c`

```
#define CH 0
#define UINT 1
#define INT 2
```

These correspond to the values that the `tag` field should take, based on if `val` is holding a `char`, `unsigned int`, or an `int`, respectively.

There are three incomplete functions `setUnion`, `getUnion` and `printUnion` that should be completed such that they can be used to respectively set, return, and print the value stored within the union.

1.1 `int setUnion(Variant* tagged_union, int dType, void* value)`

This should set the `tag` and `val` fields of `tagged_union`. The code we've provided you handles the `tag`, and the case that `dType` is 0. You need to handle the case of `unsigned int` or `int`.

1.2 `getUnion` and `printUnion`

For both of these functions, you need to handle the case that `tagged_union` holds an `unsigned int` or `int`. These are the function signatures:

```
void* getUnion(Variant* tagged_union);
void printUnion(Variant* tagged_union);
```

2 QuickSort

The `qsort` function provides an implementation of QuickSort. You can (should) read more about it in the `man` pages, but here is the function signature:

```
void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

The last argument, `compar`, is a function that compares two objects. We've provided you with an (incomplete) function called `compUnion`, which takes two arguments: `tagged_union1` and `tagged_union2`. This function should operate according to the following rule: return a negative integer if union 1 is smaller than union 2, zero if they're equal, and a positive integer if union 2 is smaller than union 1. This `compUnion` function has the following signature:

```
int compUnion(const void * tagged_union1, const void * tagged_union2);
```

You should finish the `compUnion` function. Be sure that `compUnion` compares integers by their *magnitude*; so, -7 is greater than 5, for example. You should also add the call to `qsort` in the `sortUnion` function.

2.1 Function Pointers

Function pointers are regular pointers but they point to functions so these functions can be used as arguments to other functions. In fact, if you are doing this lab in order you already used function pointers! The `compareUnion` function was passed as an argument to the `qsort` function such that it could be used to compare two elements in the array.

In this section we will be exploring how an array of function pointers can also be used to replace a switch. However, we'll be using the `qsort_r` function, because it can pass a third argument to the comparison function. The relevant comparison functions are `compareUnionAscending` and `compareUnionDescending`; for now, we'll focus just on `compareUnionAscending`, since `compareUnionDescending` simply multiplies the return value of `compareUnionAscending` by -1. Here's the function signature:

```
int compareUnionAscending(const void * tagged_union1, const void * tagged_union2, void * fun_ptrs);
```

The last argument is an array of functions. Notice that in the `main` function we construct an array called `fun_ptrs`, and place (pointers to) the `compCH`, `compUINT` and `compINT` functions in it. This is passed as the third argument to the `compareUnionAscending` function. The function used to actually compare should be determined by the `tag` field of the `tagged_union1` argument.

Notice the following at the top of `union.c`:

```
typedef int(*VarFun)(Variant*, Variant*);
```

This declares a type called `VarFun`, that points to a function that takes in two `Variant*`, and returns `int`. Thus, in `compareUnionAscending`, you should cast `fun_ptrs` to type `VarFun*`. Then, extract the correct function (again, based on `tag`), call than function, and return the result of the function.