CSE 2421

Software Design Principles in C

Communication with software users

- Give users precise and unambiguous prompts for input, especially with respect to the type of the input, and how it should be formatted (“Please enter one or more integers. The integers should be separated by spaces or tabs, with the first integer preceded optionally by spaces or tabs. Hit enter after the last integer:”).
- Print output with clear descriptions of what it represents (total, average, minimum, decoded message, etc.).
- Print error messages which give the user a reasonable description of the error which occurred (data set empty, value not found, memory not available, etc.).
- We will not be very strict about this in grading labs, but if your messages to the user are not reasonably clear, or if you have no user directions for input, or unlabeled output, we can, and usually will, deduct some points.

Dividing software into reusable components

- Each function in a C program should provide *a single piece of functionality,* defined as *a data processing activity which is reasonably likely to be useful in other programs*.
- For example, suppose we are to write a program which gets an unknown number of integer values from the user, and then sorts the values, and finally prints them out in sorted order.
- We could put all of this functionality in a single function, but this makes it much harder to reuse the code, because we may sometimes only want to have one or two pieces of this functionality without the other one(s).

Dividing software into reusable components (continued)

- If we write four separate functions, one to allocate space for the data, one to get the input and store it, one to sort the integers, and one to print them, it is easier to reuse each function.
- Then, we can write main() in our program to call each of these four other functions. That is, main is written as a driver which calls other functions to actually provide the functionality which the program requires.
- This not only promotes reusability, but also makes testing and debugging easier, because we can test and debug each of these four functions independently of the others.
- When software is written in this way, the individual functions are often short, with only a small number of lines of code (perhaps only one or two), but this is *very usual in C*.
- There are costs associated with function calls, which we will discuss further later, but the gains in reusability, testing, debugging and maintainability will easily outweigh them generally.

Advantage to One-Line Functions

- Detail hiding and flow of concentration
  - When writing one function, stay at that level of detail – don't stop and write a lower level of detail function until you finish the high level fucntion
    - It will break your flow to switch contexts
    - You will purge your "head picture" of the high level routine and then later need to get it back, effectively doing the same work twice
    - You are likely to need **time** to do that
    - You are likely to **forget** something important that you had in your head picture

Example

```
/* this will print, move, and pilot the plane */
        while(over_denver(airliner))
        {
            /* if this is my next line, I keep thinking about what belongs
             * in this loop */
            print_plane(airliner);
            /* if this is my next line, I'm thinking about how many decimal
             * places and tab stops and doubles and conversion characters
             * and names of structure members.  I might lose sight of the
             * details of the loop I'm writing */
            printf("%s is at FL%d heading %d degrees at %d knots (%7.3lf lat
%7.3lf long)\n", airliner->callsign, airliner->FL, airliner->heading,
airliner->knots, airliner->lattitude, airliner->longitude);
```

C functions in industry

- Again (it bears repeating), C functions in the outside world are often *quite short*, in order to facilitate testing, debugging, and reusability, and to make software maintenance easier.
- Again, even one or two statement functions are not uncommon (although not ubiquitous); this is *not a bad way* to write software. From a C point of view, it is *a good way*.
- We will be **fairly strict** about requiring you to separate functionality in grading labs, beginning with Lab 3. If you combine multiple reusable pieces of functionality in a single function, you will usually incur a penalty.

Communication between software components

- Software components should only communicate at their interfaces (that is, through input parameters to the function and return values).
- Also, the interface should limit interaction between the functions as much as possible, while still allowing for the required functionality.
- We will see examples soon of how to implement these principles.
- Even though C is not an object-oriented language, if C software complies with these principles, it has a number (though of course, not all) of the desirable properties of object-oriented software.

Global variables disallowed!

- “Global” variables (file scope variables in C) violate these principles, and make code *much* harder to debug and maintain. For these reasons, they should be avoided whenever possible, and only used in *very limited* types of situations.
- **You are not permitted to use global, i.e., file scope variables in C programs for this class!** Any of the programs which you are asked to write can be written without the use of “global” or file scope variables (though it can be harder to write the functions that way).

Function parameters and interfaces

- A function should only be given *as much access as needed* to data in the caller's environment by using the parameters passed to it. For example, if a function needs to be able to read a value, but not to change it in the calling environment, then the parameter should be passed in such a way that it can only be read by the function. This is the default case in C, that is, parameters are passed **by value** (the function actually only gets a copy of the value).
- If a C function needs to be able to modify the value of a parameter in the calling environment, then we must pass a pointer (using the & operator, or a pointer variable which holds the address of the value to be modified).
- Sometimes, we will need to pass a pointer, but we still do not wish for the function to be able to modify the value of the variable in the calling environment.

Formalizing it

- The S in SOLID is “single responsibility principle.” Whilst SOLID is directed at object-oriented design, we will apply it on a per-function basis in this class. Each function must have one job to do.
- Each C function in this class is limited to 10 lines of non-declarative code in nearly all cases. This requires multiple levels of code where the upper layer is about meaning and the lower level handles details.