Skel

Shane Lopez

Skel is an experimental “pseudo-language” which I have designed for the purpose of program prototype, or “skeleton” generation. At present, it is still under development, and over the course of my tinkering with it, I have ascertained that it will not be entirely complete in my eyes for some time; the reasons for this are many, among them the fun I have programming it, and the stream of ideas that occur to me day to day – actually, I am looking into a few of these now.

Skel is simple, it doesn't require any special diligence beyond proper syntax, and it is portable to any machine with a C/C++ compiler. Skel is both a program and a language; one may think of the executable “skel” as a compiler of sorts, and the submitted file the code. By using simple shorthand syntax, Skel can generate a C file with a number of lines far greater than the number of Skel code lines.

In a previous incarnation, Skel was only an executable that required three files to function; these were clunky and annoying to modify repeatedly, so I came up with a few simple rules for a language of sorts which Skel would be able to interpret, cutting the number of files required down to one; a code file. I am hesitant to call Skel a language because it is so simple in it's present state, and nowhere near as large or featured as Java or C++; However, I hope to see it evolved in the future, as I really believe it's a neat project.

Actually, Skel began with a tweet from an old room mate of mine who has since gone on to greater things in the computer science world. I had tweeted excitedly about some small triumph of mine in programming, and his tweeted (twittered?) response was:

“Have you written any functions that can write functions yet?”

That got me thinking. Why on earth would I want to? Could I? And so I began with the inkling of an idea, trying to write a function that could write functions. I succeeded, and had an executable which would write hard-coded functions to a text file. Which was a neat trick, but then I thought, if I can do functions, how about a program?

That took some tinkering, after which I had an executable that could write a hard-coded program to a text file. This was cool the first few times I ran it, but then it got stale. It was, in a word, useless. If I wanted a program written for me, I essentially had to code it myself, and even then I still had to recompile my code, run it, and compile the generated text file to get it in executable form. And so, wanting more, I picked up a fork().

I did the typing in one forked child, the compiling in another, and finally the execution in the parent, which worked marvelously – apart from the fact that I was still doing all the work for my generated program. And so, after all of this post-tweet exploration, I decided to make it a little more useful, which led directly to the first true Skel, requiring an additional three files; one to specify what libraries to include, another the functions, and lastly a file specifying where the other two were.

History lessons aside, I would like to share with you the reader the very small set of rules which presently dictate Skel's behavior:

1. **Library Includes**

   Libraries can be standard or custom, and are designated by Skel syntax as such:

   ```
   #: standard_lib_name        !: custom_lib_name
   ```

   These may appear anywhere in the Skel code, no need to put them up top.

2. **Function Declarations**

   Function declarations can be extended or abbreviated, though the extended form is intended for use with features not yet implemented in Skel. The two forms follow:

   Extended

   ```
   @: function_name, function_type, ( function_parameters ):
   .
   ```

.

.

@!

Abbreviated

@: function_name, function_type, (function_parameters):;

3. **Variables and Return Types**

   When declaring the return type of a function, use one of the following values:

   %i for int

   %d for double

   %f for float

   %c for char

   nil for void

   When declaring parameters for a function, use one of the above types (excluding nil) followed by a variable name. Don't forget to separate function parameters with commas!

4. **Forget Main, Man**

   In the interest of time, and because Skel is only for skeleton generation, a standard main function will be written into your generated C file after your declared functions, with a friendly little printf() statement.

And voila, you know the rules for Skel code! As you can see, they are simple and easy to recall. Of course, these are only for the current version of Skel, and are subject to change in subsequent releases. Heck, you could change them all if you wanted to, just modify the source code until Skel is what you'd like it to be!

Overall, I would say that I have had a fun time developing Skel, and intend to work more on it in the future. I have also learned from the experience, and have a greater appreciation for the work of language developers everywhere. Furthermore, I am always open to suggestions, criticisms, witticisms, and general chit-chat regarding Skel, and I hope to get some user involvement in the project. All users are free at any time to modify Skel however they wish, as it is intended to be a tool of use, and different people sometimes need different tools.

The future of Skel will include rules for struct definition and simple operations such as looping and basic arithmetic, and may even leverage Perl for some parsing tasks. After that, anything or nothing may happen; it will be enough for me if at least one person finds Skel interesting or useful in some small way. If nothing else, it will have been an entertaining personal research project, and fodder for a soon to be sent belated reply tweet.