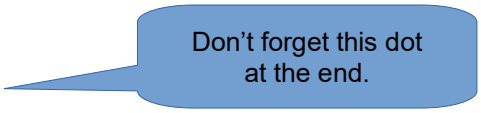
Exercise 11
Dynamic Linking

In exercise 09, you used shared, dynamically linked libraries that were automatically loaded at program start-up. In this exercise, you're going to use the Linux dynamic linking API directly, under program control. This will let you load in and execute code as your program runs, even code your program didn't know about when you wrote it. Techniques like this are used to provide a plug-in or some other extension interface for applications. An extendable application can be written to search for, load and execute new code modules that extend its functionality, all without needing to re-compile the application.

You'll be using a few shred objects I've already created for you, so you'll need to complete this exercise on an EOS Linux machine to make sure these objects are compatible. Log in on an EOS Linux machine, make a directory where you want to work on this exercise, then copy the starter files I've prepared. You should be able to use the following curl commands:

```
cp /mnt/coe/workspace/csc/CSC246/001/ex11/* .
```



You just copied a (mostly empty) source file named polyglot.c and three shared objects named hello-0.so, hello-1.so and hello-2.so. Each of these objects contains machine code for a function that looks like the following. Like in exercise 09, this function dynamically allocates an array of characters, fills it with a string (containing a message like "Hello world.") and returns a pointer to the string.

```
char *getMessage() {
  ...
}
```

You're going to complete the polyglot.c program so it is able to load the code for any number of shared objects like this. The list of shared objects will be given on the command line. It will load each shared object, find a pointer to its getMessage() function, call the getMessage() function to get a string, print out the string, then clean up by freeing the dynamically allocated string and closing the shared object.

By giving the program different command-line arguments, we can get it to load any of the shared objects. For example, running the program as follows will tell it to use just hello-0.so. The getMessage() function in this object returns the usual, "Hello world." message.

```
./polyglot hello-0.so
Hello world.
```

By listing more than one shared object on the command line, we can get the program to use the getMessage() function from multiple objects. The hello-1.so and hello-2.so shared objects return a hello world message in different languages.

```
./polyglot hello-1.so hello-2.so
Bonjour le monde.
Hola mundo.
```

We can even use the shell's pattern matching syntax to get the program to use all the shared objects in the current directory.

```
./polyglot *.so
Hello world.
Bonjour le monde.
Hola mundo.
```

Loading Shared Objects

You should already know how to iterate over the command-line arguments. We just need to add code to load and use the shared object named by each argument. The three functions you need are defined in the libdl library in Linux. You can read the online documentation for these functions using the "man" command (e.g., "man dlopen").

- **dlopen()**
  The dlopen() function opens a shared object and gives you back a void pointer as a handle for subsequently using the object. It expects two parameters. The first is the pathname for the shared object you want it to open, and the second gives additional flags. For the second parameter, I had success using either just RTLD_LAZY or RTLD_NOW. For the first parameter, I had to put a "./" at the start of the name, so the function would interpret it as a relative path (the documentation says a little bit about this). For example, if "hello-0.so" was one of the command-line arguments, I passed the string "./hello-0.so" to the function as the name of the object to open.
- **dlsym()**
  After successfully opening one of the shared objects, use dlsym() to look up the address of the getMessage() symbol. You'll need to pass in the handle you got back from dlopen() and the name of the symbol you want from the object, "getMessage". This returns a void pointer, but you want to assign it to a more specific pointer type. The getMessage symbol should be a pointer to a function that takes no parameters and returns a char pointer. Assign the return value from dlsym() to this kind of pointer type, then you can use the function pointer to call the getMessage() function, get the string it returns, then print (and free) the string.

- **dlclose()**

  Once you're done with the library, use dlclose() to release resources associated with the shared object and tell the operating system that it can be unmapped from your address space.

To use these library functions, you'll need to include the dlfcn.h header (already included for you in the starter). You'll also need to link with the dl library. The following should work to compile your program:

```
gcc -Wall -std=c99 -g polyglot.c -o polyglot -ldl
```

Once you have your polyglot program working, it should be able to load and use any shared objects that provide a getMessage() function like the one described above. You should be able to run it as shown in the examples above, using the three shared objects provided with the exercise. It should even work with other shared objects, ones you didn't know about when you wrote the program (when we test your program, we'll try this to see if it works).

When you're done, submit the source code for your completed polyglot.c program to the Gradescope assignment named EX11.