# ← Lab 6: Dynamic loading

**Due by midnight, Wednesday 10/31** 🎃

> Yeah, it's due Halloween... so get it done Tuesday ;)

In this lab, you'll be writing a **simple file compression/decompression utility.** No, you're not writing a compression algorithm!! Instead, you'll dynamically load **zlib** to do the compression and decompression for you.

This lab is a little more hands-off. I'm giving you a goal and some tools, and I wanna see how well you can put them together into a functioning program.

> **Refer to the official zlib documentation for the three functions you'll be using.**

---

## Getting started

First get a little test file from me:

```
$ cp /afs/pitt.edu/home/j/f/jfb42/public/html/img1.bmp .
```

Then, here's how your program should work:

```
$ ./lab6 -c img1.bmp > compressedimg1
$ ./lab6 -d compressedimg1 > img2.bmp
```

The first command compresses `img1.bmp` into the `compressedimg1` file. The second decompresses that file into `img2.bmp`.

After those two commands, `img1.bmp` and `img2.bmp` should be identical - in contents and length.

> *You can use `ls -l` to see the size of the files in bytes. The original and final image files should be 1179702 bytes. The compressed file should be 995320 bytes.*

---

## Your program

Here is a description of how your program will work. **Remember to start writing your code from the top down.** Stub out some functions for doing these things and call them from main.

- if `argc` < 3, complain and exit.
- load the `zlib` library.
- extract the 3 functions you need.
- open file `argv[2]` for **binary reading**.
- if `argv[1]` is "-c"
  - read the entire file into an **input buffer** that you `malloc`
  - `malloc` a **output buffer** using `compressBound` to figure out what size it should be
  - use `compress()` to compress the input buffer into the output buffer.
  - `fwrite` three things to `stdout`:
    - the **uncompressed size** (as an `unsigned long`)
    - the **actual compressed size** (as an `unsigned long`)
    - the **output buffer**
- else if `argv[1]` is "-d"
  - `fread` two things:
    - the **uncompressed size**
    - the **compressed size**
  - `malloc` an **input buffer** big enough to hold the compressed data
  - `fread` the rest of the data into that buffer (using the compressed size)
  - `malloc` an **output buffer** big enough to hold the uncompressed data (using the uncompressed size)
  - use `uncompress()` to decompress the input buffer into the output buffer
  - `fwrite` the output buffer to `stdout`
- else, complain and exit.

Your program should be fairly robust. It should **give an error message and then exit** in the following situations:

- too few program arguments
- invalid `argv[1]` (neither "-c" nor "-d")
- couldn't open the input file
- couldn't open `libz.so`
- couldn't get one or more of the symbols from zlib
- `compress` or `uncompress` failed (returned a negative number)

---

# How to do dynamic loading on UNIX

`#include <dlfcn.h>` in your program.

When you compile, give gcc the `-ldl` (that's lowercase LDL) flag, like `gcc -o lab6 -ldl abc123_lab6.c`.

To dynamically load a library:

```
void* lib = dlopen(library_file_name, RTLD_NOW);

if(lib == NULL)
{
    // couldn't load the library!
    // give an error and exit.
}
```

Then, to extract symbols from it, use `dlsym`:

```
void (*brand_new_function)() = dlsym(lib, "brand_new_function"

if(brand_new_function == NULL)
{
    // couldn't load the symbol!
    // give an error and exit.
}
```

Be sure to check the return values of `dlopen/dlsym` as shown above. Otherwise you'll start getting segfaults and not know why.

---

# Loading `zlib` and the needed functions

On thoth, `zlib` is already installed. It's named `"libz.so"`, so use that as the first argument to `dlopen`.

The three functions you need to extract are the following:

*You can make these global variables in your program. This **is** actually a legitimate use for globals!*

```
unsigned long (*compressBound)(unsigned lon
int (*compress)(void *dest, unsigned long*
                const void* source, unsigne
int (*uncompress)(void *dest, unsigned long
                const void* source, unsigne
```

For example, to load `compressBound`,

```
compressBound = dlsym(lib, "compressBound");

if(compressBound == NULL)
{
    // uh oh...
}
```

---

# How big is a file?

If you've opened a file, and you want to see how many bytes it is, it's simple:

- `fseek` to the end of the file
- use `ftell` to get the current position into an `unsigned long` variable
  - **this is the size!**
- `fseek` back to the beginning of the file

---

# Using the zlib `compress` and `uncompress` functions

Both functions have the same sort of prototype. Let's look at `compress` for now:

```
int (*compress)(void *dest, unsigned long* destLen, const void
```

- `dest` is the destination buffer, where the compressed data will go.
- `destLen` is the length of the destination buffer, but notice, **it's a pointer.**
  - when you call `compress`, give it the address of the length of your buffer.
  - that will tell `compress` how big the destination buffer is.

    - then, `compress` will **change the value of your buffer length variable.**
        - why does it do this?
        - cause it doesn't know exactly how big the compressed data will be!
    - so after `compress` returns, your variable now contains the "correct" compressed size.
        - you can now write it out.
- `source` is the uncompressed buffer.
- `sourceLen` is the size of the uncompressed buffer.

`uncompress` works virtually identically, except swap the words "compressed" and "uncompressed." :P

---

## Using `fread/fwrite` with single variables

You can think of a single variable as an array of length 1. So...

```
unsigned long myvar = ...;
fwrite(&myvar, sizeof(myvar), 1, myfile);
```

---

## Submission

Make sure you implemented error checking as detailed above!

**Then submit as usual.**