# CSci 2021 Fall 2018 Lab 14 – Linking

This lab will guide you through some practical application of the material on linking from lecture and Chapter 7 in the text.

Before you begin, recall that the Linux linker deals with global symbols as *strong* and *weak*. That designation is given by the compiler to each global symbol and passed along to the linker. A *strong* symbol is a function definition *or* initialized global variable; a *weak* symbol is an *un*initialized global variable. This classification helps the linker to resolve symbols duplicated between linked modules (files). The linker will need to resolve duplicated symbol names, and the strong/weak designation for global symbols is helpful in this process. Note that the linker does *not* deal with local (automatic) variables at all (they are handled by the compiler and system stack). But, the linker *does* deal with static (non-global) variables whether they are *local* to a method or declared within a module.

The task of the linker is largely overlooked—or ignored—since it is often bundled with the compilation process. (gcc, for example, invokes the linker by default). But, a knowledge of what the linker does (and how it goes about doing it) is helpful when building (especially) larger systems. The linker can and does detect and report errors, but sometimes what the linker does *not* tell you can be more important that what it *does* tell you. With the goal of helping you to be a more astute developer, we will explore a couple of situations that will help you to be more aware of situations where “what the linker doesn’t tell you” can be a problem.

**Step 1. A Few Basics.**

Consider the following two code modules:

```
/* linknew.c */

#include <stdio.h>

int if_compute(int);

int i = 1;  /* Answer 1a: weak, strong or neither? */
int j = 10;
int k;  /* Answer 1b: weak, strong or neither? */
        /* Answer 1c: Does it matter if k is weak or strong? */
static int a = 7;  /* Answer 1d: weak, strong, or neither? */

int main() {
    static int x = 1;  /* Answer 1e: weak, strong or neither? */
    int y;  /* Answer 1f: weak, strong or neither? */

    printf("\nif_compute(i) is: %d", if_compute(i));
    printf("\nif_compute(j) is: %d", if_compute(j));
    printf("\nif_compute(k) is: %d", if_compute(k));
    printf("\nvalue of x is: %d", x);
    printf("\nThe value of a is: %d", a);
    printf("\n");
}
```

```
/* linknewif.c */

int k = 5;  /* Answer 1g: weak, strong or neither? */
static int i = 456;  /* Answer 1h: weak, strong or neither? */

int if_compute(int x) {  /* Answer 1i: weak, strong or neither? */

    int result;

    if (x > 1)
      result = x;
    else result = -100;

    return result;
}
```

For each Answer 1*x* above, answer the question within the comment and be ready to justify your response.

1a: _______________

1b: _______________

1c: _______________

1d: _______________

1e: _______________

1f: ________________

1g: ________________

1h: ________________

1i: ________________

**Step 2. Basic Symbol Resolution.**

When compiling the code from Step 1 with gcc, the linker will need to resolve symbol references, but for *which symbols*? List all the symbols from each module that that linker will need to resolve in one column, and in a second column, list the symbols from each module that that linker will *not* need to resolve. One example of a linker-resolved symbol and one example of a non-linker-resolved symbol is shown below. (Note: Symbol resolution is necessary both within a module and between modules, but it does *not* apply to automatic variables--including formal parameters.)

| Linker Resolved Symbols | Symbols the Linker Does Not Resolve |
|---|---|
| int x in main() | int x in if_compute() |

Now, compile the code from Step 1 with the following command line:

```
gcc linknew.c linknewif.c
```

and run it with: `./a.out`

Do the results agree with what you would have thought prior to running the code? Be ready to explain.

**Step 3. What to do?**

Make one simple change to the code used in Steps 1 and 2 above. In `linknewif.c`, change the declaration of `k` from:

```
int k = 5;
```

to:

```
float k = 3.14;
```

by commenting out the old declaration and adding the new one, changing the type of `k` to `float`.

Now recompile and run:

```
gcc linknew.c linknewif.c
./a.out
```

Surprised? Or not surprised? Can you explain what is happening?

**Step 4. What to do, what to do?**

Let's try something similar, but this time replace module `linknewif.c` with module `linknewif2.c` which is shown below:

```
/* linknewif2.c */

/* Effectively, the value of k (both here and in main()) is set by
   the last call to if_compute(), so the value of k will vary as
   the program is run.  Try it! */

int k;  // changed to weak symbol, maybe static was intended?

int if_compute(int x) {

    int result;
```

```
      k = x;

      if (x > 1)
        result = x;
      else result = -100;

      return result;
  }
```

Recompile and run as follows:

```
gcc linknew.c linknewif2.c
./a.out
```

Do these results surprise you? Perhaps not, but if this were part of a *very large project*, it could be that the author (not you) of `linknewif2.c` had meant `k` to be *private* to his module. The compiler nor runtime system give any hint of what is going on with the likely unintentional connection between the two declarations of k. Would you be able to figure it out?

There is help available, you can compile with the `-fno-common` option as shown:

```
gcc linknew.c linknewif.c -fno-common
```

which will tell you about duplicate variable definitions between modules.

Try compiling again with the added linker option.

What do you see?

For larger projects, especially those written by multiple programmers, this can be a helpful linker option.

**Step 5. But, is the linker doing the best it can?**

This time, we will use a *third* version of the second module called: `linknewif3.c` and make a slight change to the original `linknew.c`

```
/* linknewif3.c */

double k;

int if_compute(int x) {

     int result;
     k = 3.14;

     if (x > 1)
       result = x;
     else result = -100;
```

```
    return result;
}
```

Now, *change the declaration* of `int k` in `linknew.c` to:

```
int k = 5;
```

Then, compile:

```
gcc linknew.c linknewif3.c
```

This time, the compiler helps out! You should see an error from the linker that looks something like this:

```
/usr/bin/ld: Warning: alignment 4 of symbol `k' in /tmp/ccKBVcx8.o is
smaller than 8 in /tmp/ccxjqz2m.o
```

When resolving `k` from `linknewif3.c` (weakly declared as `double`) with k from `linknew.c` (strongly declared as `int k = 5;`), the linker has determined that there is a problem, and it lets us know. This is good. But, exactly what has the linker determined is wrong? Compare what you think the problem is from a C perspective with what the linker gives as an error.

To help check if you have this right, try yet one more option:

In `linknewif3.c`, change the weak declaration of `k` to `float` instead of `double`.

Compile and run:

```
gcc linknew.c linknewif3.c
./a.out
```

This time, it compiles, loads and runs—although with a strange result.

What does this tell you about what the linker has to work with?

Would you say the linker is doing the best it can?
Keep in mind, when resolving the symbol `k` between modules, the linker allows the mis-match of `int/float` to pass without error (although a strange runtime result is produced similar to what happened in Step 3), but the linker generates an error for the mismatch of `int/double`.

That's it for now. Next week is your review for the last Midterm.

Have a great week!