Managing multi-file C programs

As we start to write more complex programs in C, we will find that we need to distribute our source code among multiple files. In particular, we will often have our `main` function in a separate file from the other functions that we define. For example, if we were tasked with writing a linear search implementation, we might write a function `linsearch` in its own file (maybe called `linsearch.c`), which does not have a `main` function. This would allow us to write different programs (i.e., `main` functions), which use `linsearch` in different ways. Two programs that we will commonly use in assignments are:

- A *driver* program, which is typically an interactive program that allows the user to enter data and then uses the code you wrote to produce results. In this example, a driver might ask the user to enter a series of numbers, save them in an array, then ask the user to enter a number to search for, then call `linsearch` and print the results.
- A *test* program, which is usually a non-interactive program that calls the code that you wrote with many different arguments and prints out the results. In this example, a test program might construct many different arrays and search keys, then call `linsearch` on each pair, printing out both what `linsearch` returns and what it should have returned.

Driver programs tend to be good for interactive testing of code, but are not so great for doing a lot of tests, whereas test programs are just the opposite.

Since every program must have exactly one `main` function, and these two different programs have rather different `main` implementations, `linsearch.c` had better not have *any* implementation of `main`. Instead, we should write different files (e.g., `driver.c` and `tests.c`), each of which defines `main`, and then compile our program using one of those files along with `linsearch.c`.

1. Multi-file programs: first pass

As far as the command-line is concerned, compiling multiple files into a single program is not really different from compiling a single file: just list all the files you want to compile. For example, suppose we had written the files as indicated in Figure 1 (we will explain line 4 in each momentarily). If we wanted to create a program from `linsearch.c` and `driver.c`, we could compile with `gcc` as follows:

```
$ gcc --std=c99 -o driver linsearch.c driver.c
```

Note that the name of the program (specified by `-o driver`) need not have any connection with any of the files being compiled. If instead we wanted to create a program from `linsearch.c` and `tests.c`, we would compile with:

```
$ gcc --std=c99 -o tests linsearch.c tests.c
```

The one novelty in the source code itself is lines 4 in `driver.c` and 4 in `tests.c`. The compiler always compiles each file as though it knows nothing about any of the other files. In particular, when `gcc` compiles `driver.c`, it uses no information about the contents of `linsearch.c`. But that means the compiler knows nothing about the function `linsearch`. But the compiler will not let you write code in which you call functions that you haven't told the compiler about. So far we have gotten around this by defining functions before we use them, because the compiler reads each file in order. But in this multi-file setting, somewhere in `driver.c` we have to tell the compiler something about the function `linsearch`. The only information the compiler needs in order to compile `driver.c` or `tests.c` is the types of the parameters and the return type of `linsearch`.

```
1  // file:  linsearch.c
2  int linsearch(int xs[], int n, int x) {
3      // Implemenation of linear search.
4  }
```

```
1  // file driver.c
2
3  // Function declaration.
4  int linsearch(int[], int) ;
5
6  int main() {
7      // Code to read in numbers
8      // and a search key, call
9      // linsearch, and print
10     // the results.
11     ...
12     int r ;
13     int xs[20] ;
14     int x ;
15     ...
16     // Read xs, x from terminal.
17     ...
18     r = linsearch(xs, 20, x) ;
19     printf("%d\n", r) ;
20     ...
21 }
```

```
1  // file tests.c
2
3  // Function declaration.
4  int linsearch(int[], int) ;
5
6  int main() {
7      // Code to construct many
8      // arrays and search keys
9      // and call linsearch on
10     // all of them.
11     ...
12     int xs[20] ;
13     int x ;
14
15     for (int i=0; i<20; ++i) {
16       xs[i] = 2*i ;
17     }
18
19     for (int i=0; i<40; ++i) {
20       printf("%d\n",
21              linsearch(xs, 20, i)) ;
22     }
23 }
```

Figure 1: Library code (`linsearch.c`) and two programs that use it (`driver.c` and `tests.c`).

That is what line 4 does in `driver.c`: it tells the compiler that `linsearch` is a function; it takes two parameters, the first of which is of type `int[ ]` and the second of which is of type `int`, and it returns an `int`. That is all the compiler needs in order to compile `main`; it needs this information to ensure that when `linsearch` is used, it is used correctly, which is to say that it is called with the right number of arguments, that the arguments have the right type, and that the result is used as an expression of the right type. The information consisting of the function name, the types of its parameters, and its return type is called the *signature* of the function. Line 4 is an example of a *function declaration* or *function prototype*: that is, an instruction that identifies the signature of some function.

Though it may seem surprising, the compiler does not need the definition of `linsearch` in order to compile `main`. It does need the definition in order to make an executable program out of `main`. But the compiler takes care of this by using the definition in `linsearch.c`.

```c
// file:  linsearch.c

#include "linsearch.h"

int linsearch(int xs[], int n, int x) {
    // Implemenation of linear search.
}
```

```c
// file:  linsearch.h

int linsearch(int[], int, int) ;
```

```c
// file driver.c

#include "linsearch.h"

int main() {
    // Code to read in numbers
    // and a search key, call
    // linsearch, and print
    // the results.
    ...
    int r ;
    int xs[20] ;
    int x ;
    ...
    // Read xs, x from terminal.
    ...
    r = linsearch(xs, 20, x) ;
    printf("%d\n", r) ;
    ...
}
```

```c
// file tests.c

#include "linsearch.h"

int main() {
    // Code to construct many
    // arrays and search keys
    // and call linsearch on
    // all of them.
    ...
    int xs[20] ;
    int x ;

    for (int i=0; i<20; ++i) {
      xs[i] = 2*i ;
    }

    for (int i=0; i<40; ++i) {
      printf("%d\n",
             linsearch(xs, 20, i)) ;
    }
}
```

Figure 2: Library code (`linsearch.c`) and two programs that use it (`driver.c` and `tests.c`), this time using an include file for the function declarations.

2. Multi-file programs: second pass

Writing function declarations as we did in Figure 1 is legal C, but it raises two concerns with respect to writing correct code:

(1) You have to write the function declaration every time you use the function in every file. This is an example of repeated code, which we know leads to mistakes and should be avoided.
(2) There is no way to ensure that the function declaration agrees with the definition. That is, the signature in the declaration might not be the same as the signature in the definition. Because the compiler compiles each file separately, the files `driver.c` and `tests.c` would compile just fine, even if `linsearch.c` defined `linsearch` to return a `bool` instead of an `int`. This would eventually show up as an error, but a confusing one.

We solve both problems in one stroke with *include* files.

An *include* file is just a file with function declarations (for now; later will will put other things in include files). The name comes from the fact that we then "include" the file in the other files that mention the functions that are declared in it. When we write a file of functions `f.c`, the corresponding include file is usually named `f.h`; thus the include file corresponding to `linsearch.c` is `linsearch.h`. The typical contents of `linsearch.h` and the modifications to the three files of Figure 1 are shown in Figure 2.

In `driver.c` and `tests.c` we have replaced the function declaration with an `#include` directive. Roughly speaking, the compiler replaces this line with the contents of the specified file (it "includes" the specified file at this point). Since we always include the same file, every file gets the exact same function declaration. This resolves the first problem we mentioned. Notice that we also include the file in `linsearch.c`. It is legal to have both a declaration and definition of the same function in the same file, as long as the declaration precedes the definition. When you do so, the compiler will require that the signature of the function in its definition matches the signature of the function in its declaration. This solves the second problem we mentioned, because now when the compiler compiles `linsearch.c`, it will complain if the declaration in `linsearch.h` does not match the definition in `linsearch.c`. But that means that the declarations in `driver.c` and `tests.c` must also match the definition in `linsearch.c`, because they are the same as the declaration that `linsearch.c` uses.