Lab 9: Project Management with Make Files

This week's introduces you the following features of C/Linux programming:

- Header files (See Chapter 15 of your King book)
- Make files (also discussed in Chapter 15, King)

Up until now the entire text of your program resides in a single source file (some of you may have already done this for your Assignment 5, but we will walk through it again for practice). To begin, the file:

~dkumar/cs223/a5/a5.c

contains the source code for the program you wrote in Assignment#5 (Lab 8). Make a new directory called, **Lab9** and copy the above file into it. Also, copy the file **code.txt** into the same directory.

Before we begin, let us first compile and run this program:

```
[Xena@CodeWarrior]$ gcc –o a5 a5.c
[Xena@CodeWarrior]$ ./a5 code.txt
```

Run the program and interact with it. Search for a couple of airports to remind yourself of what it does. Next, open the source in an editor to examine its contents. It has the following structure:

```
// FILE: a5.c
…
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
…

// Define the Airport data structure

typedef struct {
   …
} Airport;

// Functions relating to Airport type
void show(Airport a);               // prints out a given airport object
Airport parseAirport(char * str);   // Parses str into an Airport object

// Functions for handling Airport database load and search
void readData(char *file, Airport list[], int *n);
int search(Airport list[], int n, char *code);
int more();

int main(int argc, char *argv[]) {

   …

} // main()

int search(Airport list[], int n, char *code) {

   …
```

```
} // search()


int more() {
   …
} // more()
void readData(char *file, Airport list[], int *n) {
   …
} // readData()


void show(Airport a) {  // print Airport object
   …
} // show()


Airport parseAirport(char * str) { // Parse srt into an Airport object
 …
} // parseAirport()
```

Please, read and study the program carefully before proceeding. If you have any questions about anything on this program, this would be a good time to call your instructor and ask.

Task#1: Breaking it up

Clearly, as you know from your earlier experience with Java/Python, we are defining a new data type called, **`Airport`** in the program. At least the functions `show()` and `parseAirport()` are directly associated with the `Airport` type. As you did in Java/Python, C also allows you to define a data type in a separate file. In fact, in C, you will use two files to define a data type: **A header file** (ending with the extension ".h"), and a **source file** (ending with the extension ".c"). Let's go ahead and break up the file **`a5.c`** into three pieces:

- A header file: **`Airport.h`** with the following contents
    - The definition of the `Airport` type
    - Headers of functions associated with the `Airport` type: `show()`, and `parseAirport()`
- A source file: **`Airport.c`** containing the actual definitions of the `Airport` functions: `show()` and `parseAirport()`
- A main program file: **`a5.c`** (we will describe its contents below)

Here are the contents of the header file: `Airport.h`

```
// FILE: Airport.h
// Define the Airport data structure
```

```
typedef struct {
   char code[5];  // code
   char name[50]; // name of airport
   char city[30]; // city
   char state[5]; // state
   char country[20];    // duh
} Airport;

// Functions relating to Airport type
void show(Airport a);   // prints out a given airport object
Airport parseAirport(char * str);   // Parses str into an Airport object
```

And here are the contents of the file: `Airport.c`

```
// FILE: Airport.c
#include <stdio.h>
#include <string.h>

#include "Airport.h"

void show(Airport a) {  // print Airport object
   printf("%s - %s, %s, %s (%s).\n",a.code,a.name,a.city,a.state,a.country);
} // show()

Airport parseAirport(char * str) { // Parse srt into an Airport object
   …
} // parseAirport()
```

Notice the highlighted `#include` command above. Also, note that the file name `Airport.h` is enclosed in double-quotes and not <…>. This command essentially says that at that point, the compiler will insert the contents of the `Airport.h` file (as if it were just one continuous file).

Next, edit the file `a5.c` to have the following:

```
// FILE: a5.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Airport.h"

// Functions for handling Airport database load and search

void readData(char *file, Airport list[], int *n);
int search(Airport list[], int n, char *code);
int more();

int main(int argc, char *argv[]) {
   …
} // main()

int search(Airport list[], int n, char *code) {
   …
} // search()
```

```
int more() {
   …
} // more()

void readData(char *file, Airport list[], int *n) {
   …
} // readData()
```

Once again, notice that we have the `#include` command (highlighted). The header file, `Airport.h` has all the information that is needed for the compiler to compile the file `a6.c`

Separate Compilation in C

One advantage of having a program split into multiple files is that you only need to focus on and edit one file at a time. The GCC compiler allows you to compile the individual file without compiling all the other files. Let us learn how to compile files individually and then putting together (or, building) the executable program.

To compile a single file, you use the following command:

```
[Xena@CodeWarrior]$ gcc –c Airport.c
```

This produces a file named, Airport.o (called an **object file**). The `–c` option tells the compiler to just compile the file specified without attempting to build the entire program. Similarly, you can then compile the main program file:

```
[Xena@CodeWarrior]$ gcc –c a6.c
```

Now, you will have two object files: `a5.o` and `Airport.o`

You never need to compile a header file. Why?

To build the executable program using the two object files, you now have to give the command:

```
[Xena@CodeWarrior]$ gcc –o a5 a5.o Airport.o
```

This command tells the compiler to produce a runnable program, `a6` by combining the two object files that are needed to build the complete the program. Once again, you should test to see if the program is running properly:

```
[Xena@CodeWarrior]$ ./a5 code.txt
```

From now on, all programs you write in C will involve one or more data structures. You should design the program so that each data structure is defined in a separate file (along with its header file). In the next few weeks, we will learn further details about how to plan and code the programs that span multiple files.

Task#2: Make Files

When programs get larger, they are typically split across several files (dozens!). It would be quite tedious then to try and keep track of all the files that need to be compiled, or recompiled, and the list of files needed to build a complete runnable program. Linux/GCC has a very useful tool called, **`make`**, that makes this process very easy. Since we now have a program with three files (`Airport.h`, `Airport.c`, `a5.c`), it is a perfect situation to learn how to set up and use `make`.

Essentially, all the details about compiling and building a program can be defined in a file called, **`Makefile`**. In it, you will provide information (rules) on what files are needed to compile a given source file, how to create a runnable program using all the needed object files, etc. Furthermore, Linux keeps track of what files you may have been editing and then the rules in a make file will take care of the needed compilation process.

Create a new file called, **`Makefile`** in the same directory as your program from Task#1. In it, place the following lines:

```
a5: a5.o Airport.o
      gcc –c a5 a5.o Airport.o


a6.o: Airport.h a5.c
      gcc –c a5.c


Airport.o: Airport.h Airport.c
      gcc –c Airport.c
```

The `Makefile` (as shown above) contains three rules. Each rule has the form:

```
<target>: <files needed to build target>
<TAB><command needed to build target>
```

The second line of the rule should always use the **TAB** character, and NOT spaces. This is a requirement for specifying make files.

As you can see, the `Makefile` has three rules:

1. To build `a5` (complete program), you need to have current versions of `a5.o` and `Airport.o` Then, to produce `a5` use the command provided on the second line.
2. To build `a5.o`, you need to have current versions of `Airport.h` and `a5.c`
3. To build `Airport.o`, you need to have current versions of `Airport.h` and `Airport.c`

Now all you have to do is to use the `make` command as shown below:

```
make <target>
```

For example, for our `Makefile`:

```
[Xena@CodeWarrior]$ make a5
gcc –c a5.c
gcc –c Airport.c
gcc –o a5 a5.o Airport.o
```

Notice that the files were compiled in proper order. To see the "magic" of a make file, go ahead and edit the file `a5.c` (just add an extra line or something). Then give the `make` command again:

```
[Xena@CodeWarrior]$ make a5
gcc –c a6.c
gcc –o a6 a6.o Airport.o
```

Notice that only the file `a5.c` needed to be recompiled. If you make a change in `Airport.h` (go ahead and try it) and then give the make command:

```
[Xena@CodeWarrior]$ make a5
gcc –c a6.c
gcc –c Airport.c
gcc –o a6 a6.o Airport.o
```

All the files will need to be recompiled because they are dependent on the `Airport.h` file. One last time, go ahead and edit the `Airport.c` file and issue the make command:

```
[Xena@CodeWarrior]$ make a5
gcc –c Airport.c
gcc –o a5 a5.o Airport.o
```

This time, only the `Airport.c` file will be compiled.

The `make` command has many useful features. We will introduce some more in the coming weeks. For now, please review what you did today and ask your instructor if you have any questions.