Lab 10/28: Project Management with Make Files

Today's lab re-introduces you the following features of C/Linux programming:

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

Until now the entire text of your program has probably resided in a single source file. In this lab we will be working with code from the Airports homework. You can either use your own code, or mine. Mine might be easier as it follows the text below. However, you should feel free to use your own.

Make a new directory called for this lab and copy either your code from Airports or mine into it. (Mine is at `/home/gtowell/Public246/Lab1028/airport.c`.) Also, for ease of use make a soft link to the airport data set. (The data is in `/home/gtowell/Public246/HW05/code.txt`.)

If needed, please, read and study the program carefully before proceeding.

Task#1: Breaking it up

In the code we define a new data type called, **`Airport`**. At least the functions `show()` and `parseAirport()` are directly associated with the `Airport` type. As in Java (and Python), C allows you to define a data type in a separate file. In C, you 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"). Start by breaking up the file `airport.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: AMain**`.c`** (described below)

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

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

typedef struct {
   char code[5]; // iata 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);     // Parse to an Airport object
```

And 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
  …
} // show()

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

Notice 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).

Finally, edit the file AMain`.c` to have the following:

```
#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 an individual file without compiling all the other files.

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 AMain.c
```

Now, you will have two object files: AMain.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 arprt AMain.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]$ ./arprt 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 significant data structure is defined in a separate file (along with its header file).

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, AMain.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 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 have changed; the rules in a make file 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:

```
AMain: AMain.o Airport.o
      gcc –c arprt AMain.o Airport.o
```

```
AMain.o: Airport.h a6.c
      gcc –c a6.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 `a6` (complete program), you need to have current versions of `a6.o` and `Airport.o`
   Then, to produce `a6` use the command provided on the second line.
2. To build `a6.o`, you need to have current versions of `Airport.h` and `a6.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 AMain
gcc –c AMain.c
gcc –c Airport.c
gcc –o arprt AMain.o Airport.o
```

Notice that the files were compiled in proper order. To see the "magic" of a make file use the unix `touch` command on a6.c — `touch a6.c`. (What does touch do?) You could also make a change in the file. Then give the `make` command again:

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

Notice that only the file `a6.c` needed to be recompiled. If you touch `Airport.h` and then give the make command:

```
[Xena@CodeWarrior]$ make AMain
gcc –c AMain.c
gcc –c Airport.c
gcc –o arprt AMain.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 Airport
gcc —c Airport.c
gcc —o arprt AMain.o Airport.o
```

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

The `make` command has many useful features beyond those you worked with today (or we discussed in class several weeks ago).

Task 3:

Copy your Makefile to `/home/gtowell/submissions/fall2019/cmsc246/YOURLOGIN-makefile/lab1028`