COMP2401B – Tutorial 8
Make utility and using function pointers

Learning Objectives

After this tutorial, you will be able to:
- Organize your program files for compiling
- Use the make program
- Have basic insight into the structure of makefiles
- Use function pointers

At the end of the tutorial submit a tar file – **t8.tar** - containing your work.

Overview

This tutorial is comprised of two sections. Section 1 explores the make utility – its benefits, usage and construction. Section 2 Function Pointers provide some insight into function pointers.

1 Make files

Purpose: Creating a makefile

Files: Download the tutorial tar file t8.tar and extract the files into the tutorial directory.

As programs get larger and include multiple files, the management of the files and the executable is getting harder. Often, it is hard to keep track of all the files that were changed. As a result, if changed files are not compiled and relinked then the changes are not included in the executable.

One option is to recompile and link all the files. However, this may be time consuming. For example, if a program includes 50 *.c files and each file compilation takes 3 seconds then one has to wait 150 seconds or about 3 minutes before one can test the program again.

Unix has a utility that assists in the management of the files – the make utility. The make utility is designed to help determine which files should be recompiled and relinked into the program.

The make utility reads an input file, which by default is named Makefile, to manage the code. The purpose of the Makefile is to maintain the dependencies between files. If FileA depends on FileB, then whenever a change occurs in FileB then FileA must also be recompiled. For example, file main_test.c contains a test program for a linked list and file linked_list.c contains the linked list functions. In this case if there are changes to the linked list code then the file main_test.c may need to be recompiled and re-linked in addition to linked_list.c.

The dependencies are listed in the makefile as follows:

File name: list of dependencies

In the above example the dependency will appear in the file as

FileA : FileB

FileA, which appear before the ':' is termed the target file and FileB is termed the dependency file list.

Next, one needs to determine the action that must be taken when a change has occurred in one of the files (e.g., FileB above)

The action is written below the list of dependencies. For example if FileB has changed then a new executable is created using gcc –o FileA FileB

This will appear in the file as follows:

FileA : FileB

<tab>gcc –o FileA FileB

The symbol <tab> represents a hard tab. This is a must because the make utility uses the <tab> to determine the command/rule to be invoked. Note that this allows one to create a sequence of commands. For example

FileA : FileB

<tab> command 1

<tab> command 2

Note for Vim users: if you find it converts the tabs to spaces then open the file .vimrc and comment out the line set expandtab. Commenting is done by insert a " as the first character.

FileA is termed the target file. The compiling instruction gcc… is termed the rule or command.

In this tutorial you have three .c files (person.c main.c and mystat.c) and two .h files (mystat.h and person.h).

1. What is the dependency of each of the .c files? Each file is dependent on the .h files that are included in the file. For example mystat.c depends on mystat.h

   1.1. Determine for each .c file which files it is dependent on

   1.2. Add the dependencies in a file called Makefile. One target per line and at least one line spacing two separate targets.

2. Here we will use as targets the object files of each of the c files. For example the file mystat.o will depend on mystat.c and mystat.h

1.1 Creating the rule for person.c

The file person.c depends on person.h. The reasons are: 1. person.h contains the data structure required by person.c. Thus, if the data structure was modified then the code should be aware of it; and 2. Person.h contains the prototypes of the functions in person.c. Thus, if the functions parameter were changed then they should be updated in the person.h.

2.1. Creating the rule for person.c

Add the following entries to your Makefile. This means that the target is person.o (we would like to create person.o) and that a change to any of the files person.c, person.h or mystat.h will trigger the rule gcc person.c

```
person.c: person.h mystat.h
    gcc person.c
```

Note the rule was indented using the tab.

2.2. Modify the file person.h in order to ensure that it is newer than person.c. Here you can either use the touch command as touch person.h or modify person.h and save it (e.g., by adding a space to person.h

2.3. Execute the make utility by typing make in the command line. The make utility will look for the default file name Makefile. If you are using another file (e.g., myMakefile) then you invoke the make tool as follows make -f myMakefile.

What was the outcome? Were there any errors? Why did the errors occur?

The errors occurred because the file person.c does not have all the needed code to compile and link it into an executable (e.g., it does not have a main() function). Therefore, the command gcc person.c was not correct. It should be modified to create an object file. Change the rule to

gcc –c person.c

2.4. Invoke the make utility again by typing make. What was the outcome? What file was created?

2.5. Invoke the make utility again. What happened? It seems that the file person.c was compiled again even though the file person.c and person.h were not changed. Why did it happen?

The file was compiled again because the target, which is person.c is not affected by the rule. That means that we used the wrong target for the file person.c. Change the target to person.o instead of person.c and update the dependency list to include person.c

```
person.o: person.h person.c mystat.h
    gcc -c person.c
```

2.6. Delete the file person.o and invoke the make utility again by typing make. What was the outcome? What file was created?

2.7. Invoke the make utility again. What happened? It seems that the target person.o is newer than person.c and person.h and therefore the file person.c was not compiled again. Thus, it is important when one creates a target and a rule in a makefile to ensure that the target is affected by the rule.

1.2 Creating the rule for the remaining files executable

1.1. Adding the rules for the remaining files

Add the target and the rules for the remaining files mystat.c and main.c so that an object file will be created for each file.

1.2. Test your Makefile by invoking the make program and ensuring that the object files were created.

Were all the object files generated?

No, they were not generated. This is because the make utility assumes, by default, that the first target in the file is the important one and that the dependency list consists of one or more other targets in the files.

Therefore, in order to explicitly invoke a target one can specify it as follows

make mystat.o

or if you use your own file myMakefile then

make -f myMakefile mystat.o

1.3 Creating the rule for the executable

Here we will create and executable with the name myExec. The program myExec will depend on the three object files that were generated. This means that the target is myExec.

1.1. Add the following to the Makefile:

```
myExec: main.o person.o mystat.o
        gcc -o myExec main.o person.o mystat.o
```

1.1. Invoke the make utility. Was the executable myExec generated? If it is was not generated then try it by invoking it with the target

make myExec

If it was generated then the problem is that it is not the first target in the file. Otherwise check your Makefile.

In order to make sure that it is run by default move the rules for the target myExec to be the first in the Makefile.

1.4 Delete all the *.o files

The main purpose of the make utility is to support maintenance of program executables. However, one can use the make utility to do other things (other than compiling and linking).

For example, one may want to delete all object file (*.o). To do this one can create a target and a command to do so. The "standard name" for a target to delete all object files is clean.

To do

1. Add a target ***clean*** to the file. This target will not have any dependencies.
2. Add the rule rm –f *.o This rule will remove all *.o files from the directories. Note that one needs to add the flag –f to the rm rule. The rule "rm *.o" will not work.

```
clean:
<tab>rm -f *.o
```

Invoke the command by calling ***make clean***

1.5 Additional Information

The program "make" offers of a large number of features. In fact, most of the SDKs are using one form of another of make to organize, compile and build your program.

One important feature of the make program is the ability to set up macros. Macros allow a program to set or change information in a Makefile in one place only. For example, one can set up the compile parameters to include the "-g" option when building each of the files. This is similar to the "#define " preprocessing directive in a c program.

See also http://www.tutorialspoint.com/makefile/

2 Function Pointers

Function pointers are variables that hold an address of a function. Function pointers can be variables, fields in structure or a parameter of a function.

Function pointers are usually used to create general-purpose functionality or framework – here the objective is to create a general-purpose framework or functionality that can be reused without writing everything again. At times, the functionality or framework is independent of data types. An example of a general-purpose framework that is data type independent is the function qsort(). Here, the designers wrote a fast sorting algorithm of arrays that can sort any array of any data type. They achieved it by requiring the user to provide a function that can compare two records of the array. To see an example read the course notes Chapter 3 pages 163-165.

2.1 Simple usage of a function pointer

In this part of the tutorial you will create a simple function that will allow the user to change the printing function of an array.

Step 1 – create the function that accepts a function pointer for processing the array records. Here, for processing you will use a printing function.

The function will accept three parameters: the array, the size of the array (number of records in the array), and a function pointer parameter that will be used on every record of the array.

The declaration of the function pointer is

int (*processFun)(PersonRec *rec)

Note:

1. The function pointer is declared as (*processFun). This tells the compiler that the name of the function pointer is processFun. The function processFun return data type is an "int" and it accepts one parameter, which is a pointer to a PersonRec structure.

2. Next write the prototype of the function that will process the array using a function pointer parameter. The function name is processArray. The prototype should be added to the .h file

The function prototype is

int processArray(PersonRec *arr, **int** size, **int** (*processFun)(PersonRec *rec));

Note that the function pointer is declared in a similar way to declaring a function prototype.

3. Write the function processArray in the file person.c
The pseudo code of the function is as follows:

// loop on each record of the array

// process each record using the function parameter

This is done by using a for loop

```
for (i = 0; i < size; i++) {            // loop on each record of the array
        processFun(arr[i]);             // process each record using the function pointer
}
```

Compile the file the person.c to create an object code. You should get an error. Fix the error by looking at the declaration of the function pointer and the actual parameter that is passed to it.

4. Using the function processArray
In the main function use the function processArray to print the array. Here you will use the function print() that is already supplied.

Add a one line of code in main as follow
processArray(people, NUM_PEOPLE, print);

Note that the function print() is passed in using only the function name.

2.2 Using a different printing function

Here you will create another print function that will print all the people whose salary is greater than 32000.

Steps

1. In the file person.c copy the function print() to a new function with the name printHighSalary()

2. Modify the printHighSalary() function by adding a condition to print only records with an average salary >= 32000.

Pseudo code

```
// Compute the average salary
average(p->salary, NUM_YEARS, &avg_best_five_years);
// if the average salary >= 32000 print the record
if (avg_best_five_years >= 32000) {
        // print the record
        printf("Person Name ….);
}
```

3. Add a prototype of the function printHighSalary() to the file person.h

4. Modify the main() function to invoke the function processArray() with the printHighSalary() as its function pointer.

```
processArray(people, NUM_PEOPLE, printHighSalary);
```

2.3 Using a variable salary

What if you wanted to print out the names of all the people with salary higher than 30000 instead of 32000?

How would you modify the code?

Take a few minutes to think about a solution.

Solution:

1. One option is to modify the number and then recompile the code. This is not a good solution because it is inconvenient and does not allow to use the function to be reused with different maximum salaries.

2. Add a parameter to the function pointer to indicate the salary. Modify the code of printHighSalary() by adding a parameter to its parameter list.

   Pseudo code

   int printHighSalary(PersonRec *p, float salary)

   {
   // Compute the average salary
   average(p->salary, NUM_YEARS, &avg_best_five_years);

   // if the average salary >= salary print the record
   if (avg_best_five_years >= salary) {
   // print the record
   printf("Person Name ….);
   }

3. Modify the processArray() function to include the new parameter

   int processArray(PersonRec *arr, **int** size, **int** (*processFun)(PersonRec *rec), int salary);

4. Modify to the invocation of the function processArray() in main() to include the salary.

At the end of the tutorial time submit a tar file – **t8.tar** - containing all the files you worked on.

End of tutorial