Multiple Files

Motivation

Up to this point, all of our programs have been written within a single file. There is nothing wrong with this – a C program is just a bunch of functions, and it's fine to group those functions within a single file. However, as your programs get bigger, it's nice to physically separate functions into different files. This makes it easier to find certain pieces of your program.

Separating functions also promotes reuse. Right now, if we wanted to reuse a function we'd written in another program, we'd have to copy it from our old program to our new one. With multiple files, we can separate the functions we want reused and just link to that file when we want to use them (this is like separating the C library functions and including them when we want to use them).

Header Files

Our first step in writing a program with multiple files is to just divide related functions into different .c files. However, suppose we're in our main function and we call a function from a different file? This is just like calling a C library function without using any include statements. The compiler will not know where to find the outside function.

To solve this within the C libraries, function prototypes are placed in *header files* (.h files). The functions themselves are implemented in corresponding .c files. If I want to use a C library function, I include the appropriate header file so that the compiler knows about the function.

This is what we will do with our C programs. First, we will have a "main file" that contains the main function and any other function we do not want to reuse. This code will stay in a .c file as we've done before. Then, we will divide other functions among several .c files (functions that are related in some way are usually put in the same file). For each of these .c files, we will make a .h file with the same name that contains the prototypes of each function. Finally, we will include these header files from our main file.

One-File Program

To see how this works, consider the example below, which is in the file `prog.c`:

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

//break into stats.h
int min(int*, int);
int max(int*, int);
double avg(int*, int);

//keep main function here
int main() {
     int* nums;
```

```
	int i, length;

	printf("Enter size of array: ");
	scanf("%d", &length);

	nums = malloc(length*sizeof(int));
	for (i = 0; i < length; i++) {
		printf("Enter a number: ");
		scanf("%d", nums+i);
	}

	printf("The min is: %d\n", min(nums, length));
	printf("The max is: %d\n", max(nums, length));
	printf("The average is: %.2lf\n", avg(nums, length));

	return 0;
}

//break into stats.c
int min(int *vals, int size) {
	int min = vals[0];
	int i;
	for (i = 1; i < size; i++) {
		if (vals[i] < min) {
			min = vals[i];
		}
	}

	return min;
}

int max(int *vals, int size) {
	int max = vals[0];
	int i;
	for (i = 1; i < size; i++) {
		if (vals[i] > max) {
			max = vals[i];
		}
	}

	return max;

}

double avg(int *vals, int size) {
	int sum = 0;
	int i;
```

```
        for (i = 0; i < size; i++) {
                sum += vals[i];
        }

        return sum/(double)size;
}
```

Multiple Files Program

Suppose we want to be able to reuse our `min`, `max`, and `avg` functions. We'd create a corresponding file, `stats.h`, with the `min`, `max`, and `avg` prototypes:

```
// stats.h
#ifndef STATS_H
#define STATS_H

int min(int*, int);
int max(int*, int);
double avg(int*, int);

#endif
```

(Notice the ifndef statement, and defining the constant STATS_H. This is standard when writing header files to keep the same code from being included twice. Just surround your .h file with a similar statement – changing the STATS_H constant to match your header file's name.)

Then we'd create the file `stats.c` with the implementations of the `min`, `max`, and `avg` functions:

```
//stats.c

//We must include the corresponding header file
#include "stats.h"

int min(int *vals, int size) {
        int min = vals[0];
        int i;
        for (i = 1; i < size; i++) {
                if (vals[i] < min) {
                        min = vals[i];
                }
        }

        return min;
}

int max(int *vals, int size) {
```

```
        int max = vals[0];
        int i;
        for (i = 1; i < size; i++) {
            if (vals[i] > max) {
                max = vals[i];
            }
        }

        return max;

    }

    double avg(int *vals, int size) {
        int sum = 0;
        int i;
        for (i = 0; i < size; i++) {
            sum += vals[i];
        }

        return sum/(double)size;
    }
```

Finally, we can rewrite our main file:

```
    //prog.c
    #include <stdio.h>
    #include <stdlib.h>

    //include our header file
    //use "" since it is in the current directory
    #include "stats.h"

    int main() {
        int* nums;
        int i, length;

        printf("Enter size of array: ");
        scanf("%d", &length);

        nums = malloc(length*sizeof(int));
        for (i = 0; i < length; i++) {
            printf("Enter a number: ");
            scanf("%d", nums+i);
        }

        printf("The min is: %d\n", min(nums, length));
        printf("The max is: %d\n", max(nums, length));
```

```
        printf("The average is: %.2lf\n", avg(nums, length));

        return 0;
    }
```

Header files can also contain definitions of types needed for the corresponding .c file, like structs, unions, and enums.

Compiling with Multiple Files

The way we would compile the original statistics program (in the file `prog.c`) is:

```
gcc prog.c
```

Now that our program is in two files, we might try:

```
gcc prog.c stats.c
```

or

```
gcc stats.c
gcc prog.c
```

...but in fact neither of these statements will get our program to compile. The trouble is that the C compiler first compiles each .c file, and then tries to link each of the files together into a single executable. If we don't tell the compiler how the files are related, it won't be able to properly link them.

To compile our new program, we must first compile each file separately:

```
gcc –c stats.c
gcc –c main.c
```

The –c option forces the compiler to stop after compiling the file, before trying to link the files or create an executable. These two instructions will create the object files `stats.o` and `main.o`, respectively. Now, we need to link these two compiled files into an executable:

```
gcc stats.o main.o
```

This line will create the executable `a.out`, which we can now run as usual.

Makefiles

It now takes three lines to compile our program, which is a pain to have to type every time we make a change. We can simplify compilation by placing all of the compilation instructions in a single file called a *Makefile*.

Here is the format of a Makefile:

compiler declaration
object list declaration

compiling/linking instruction

cleaning instruction (removing output files)

Here is the Makefile for our statistics program:

```
CC = gcc
OBJECTS = prog.o stats.o

nums: $(OBJECTS)
      $(CC) $(OBJECTS) -o nums

clean:
      rm *.o nums
```

Now, let's go through the Makefile one line at a time:

```
CC = gcc
```

This line declares the variable `CC`, and says that we're using the `gcc` compiler. (When we move to C++ programs, we will start using the `g++` compiler.)

```
OBJECTS = prog.o stats.o
```

This declares the variable `OBJECTS`, which contains a list of all object files for the program. (An object file is a single file converted to machine code.) We have a .o file for every .c file in our program. So, since we have the files `prog.c` and `stats.c`, will we get the object files `prog.o` and `stats.o`.

```
nums: $(OBJECTS)
      $(CC) $(OBJECTS) -o nums
```

This line describes how to compile our program. The "`nums`" at the beginning is a name we're giving this compilation instruction – we could have called it anything. For our program to compile, all the .o files must be generated. The `$(OBJECTS)` on the top line gets the value out of the `OBJECTS` variable. This line says that to compile the program, all the object files must be made.

When we substitute the values of the variables, the next line becomes:

```
gcc prog.o stats.o –o nums
```

This line says to link together the two object files, and rename the executable to "`nums`". (That's the "`-o nums`" part).

Next, we have:

```
clean:
     rm *.o nums
```

This line says that to clean our program, we want to delete the object files (*.o) and the executable (`nums`).

Using a Makefile
Now that we have a Makefile, we want to use it to compile our program. To compile a program with a Makefile, type:

```
make
```

This will generate all the object files, link them together, and create the executable called "`nums`".

To run our program:

```
./nums
```

This is the same as what we've done before to run our program, but now the executable is named "`nums`" instead of "`a.out`".

To remove all the output files, we type:

```
make clean
```

This will remove all the .o files and the executable `nums`.

Makefile Rules
We could spend the rest of the semester talking about exactly how Makefiles work and different intricacies and still not cover everything. Makefiles are a very powerful tool – we've only scratched the surface to be able to compile our program in a specific way.

Here are some rules to follow as you write your own Makefiles:

- You should always define the variables `CC` and `OBJECTS`. The value of `OBJECTS` should include a .o file for every .c file in your program.

- You can rename your executable ("`-o nums`") to be whatever you want.

- There should be a **tab character** at the beginning of the second line for the `nums`: instruction and the clean: instruction. This MUST be a tab – spaces won't work.

- For the most part, your Makefiles will remain the same from program to program. You'll just change the `OBJECTS` list and change the name of the executable.

Extern Variables

Sometimes when our program is in multiple files, we still want to define variables that are visible to each file. Here's how to do this:

1) Declare the global variable in a .h file. This can either be done in a .h file with some of your function prototypes, or in a special file, `globals.h`, that contains the declarations of all global variables. If you do use a special globals header file, it does not need a corresponding .c file.

2) Include the .h file wherever you want to use the variable

3) When you want to use the variable, declare:

```
extern type name;
```

   where `type` is the type of the global variable, and `name` is its name. The "`extern`" keyword tells the compiler not to create a new variable, but instead to find the variable `name` declared in another file.

As an example, suppose we want to make the array size a global variable in our stats example from earlier in this document. Here's what we'd do:

```
// stats.h
#ifndef STATS_H
#define STATS_H

//define array size
int size;

//functions no longer need array lengths
int min(int*);
int max(int*);
double avg(int*);

#endif
```

****

```
//stats.c
```

```
//We must include the corresponding header file
#include "stats.h"

//say we're going to use the size variable
extern int size;

int min(int *vals) {
     int min = vals[0];
     int i;
     for (i = 1; i < size; i++) {
          if (vals[i] < min) {
               min = vals[i];
          }
     }

     return min;
}

int max(int *vals) {
     int max = vals[0];
     int i;
     for (i = 1; i < size; i++) {
          if (vals[i] > max) {
               max = vals[i];
          }
     }

     return max;

}

double avg(int *vals) {
     int sum = 0;
     int i;
     for (i = 0; i < size; i++) {
          sum += vals[i];
     }

     return sum/(double)size;
}
```

****

```
//prog.c
#include <stdio.h>
#include <stdlib.h>
```

```
//include our header file
//use "" since it is in the current directory
#include "stats.h"

//declare that we're using the size variable
extern int size;

int main() {
     int* nums;
     int i;

     printf("Enter size of array: ");
     scanf("%d", &size);

     nums = malloc(size*sizeof(int));
     for (i = 0; i < size; i++) {
          printf("Enter a number: ");
          scanf("%d", nums+i);
     }

     printf("The min is: %d\n", min(nums));
     printf("The max is: %d\n", max(nums));
     printf("The average is: %.2lf\n", avg(nums));

     return 0;
}
```

Notice that we declare the variable in a header file. When we want to use the `size` variable, we first include that header file. Then, we declare size as an extern variable.