

Appendix D

Structuring a Software Project

D.1 Overview

This laboratory introduces good programming practice for structuring software projects. As your programs grow you will find that it is useful to split the functions you write across multiple source files. This laboratory teaches you the best way to do this. Although the concepts that are introduced in this lab will not have an effect on how your programs look once they are built, well structured source code is important. By using multiple files you can make your programs easier to understand and debug. You can also make it easier to reuse code you create for one program when you are writing another. Something which could save you a lot of effort!

Finally, a few tips on good programming practice are covered. Focussing on how to use indentation to make your programs more readable, and how to use the IDE to do this easily.

D.2 Splitting Your Program into Multiple Files

In laboratory 4 you learned how, and why, to split your programs into functions. One of the most important effects of splitting your code into functions is to wrap it up (or *encapsulate* it) into small chunks which can be *reused* easily. To be able to reuse your function in another program, you have had to copy it and paste it into another source file.

It is usual for more complicated C programs to be made up of more than one source file. Each source file typically contains multiple functions. The reasons for doing this are very similar to the reasons for using functions in the first place:

- using multiple files adds structure to your program and makes it easier to understand. For example, you could collect all the functions that produce output on the screen into one file, making it easier to find functions of a particular type.
- putting a set functions into a file allows you to use *the same file* in multiple projects. If there are bugs in the functions in that file, you only have to fix them once, rather than having to change the code in every project it is used in.

In this lab you will learn how to split your source code into multiple files, by placing different functions in each file.

It is assumed that you have created independent projects for your option work. If you have been doing the graphics option it is assumed that the name of his project is “graphics1” and if you have been doing the music option, then it is assumed that the name of the project is “music1”. If you do not know how to create your own project then look back at lab 1.

Begin by opening the project that you have been using for your option work in the IDE. These programs are getting quite complicated by now, and probably contain a number of functions. You are going to separate these functions into two files depending on their role in your program.



Exercise D.1: Deciding on a File Structure


Before you actually create any new files, or change your program in any way, you should decide how, and why, you will structure the files.

Decide how many files you want to use. This should be more than one and no more than about four at the moment. This decision should be based on what each of the functions you have written do. Some things to think about:


- You could collect together into one file the functions which interact with the user.
- If the `main` function is very complicated, you might want to leave it in a file on its own.
- You might want to collect the graphics/music functions together into one file.

Don't make things more complicated than necessary; try to create a structure which is logical and would help someone else understand your program. Decide on names for your files which reflect what they do. To keep things simple, it is best to avoid file names that have spaces in.


Make some notes about your file structure to help you remember your decisions.



You can now create a new file in which to place some of your code. Select the “New” item from the “File” menu. Select “Empty file”. You will see a message about whether you want to add the new file to your active project. Say “Yes”. You will then be able to edit your new file.



A blank file will appear in the editor ready for you to put some source code in it. Copy (or cut) and paste at least one function into this file. **You will need to #include any header files which the function needs.** Save the file with the extension “.c”. The best place to put the file is in the “src” directory (where all the source files are located).



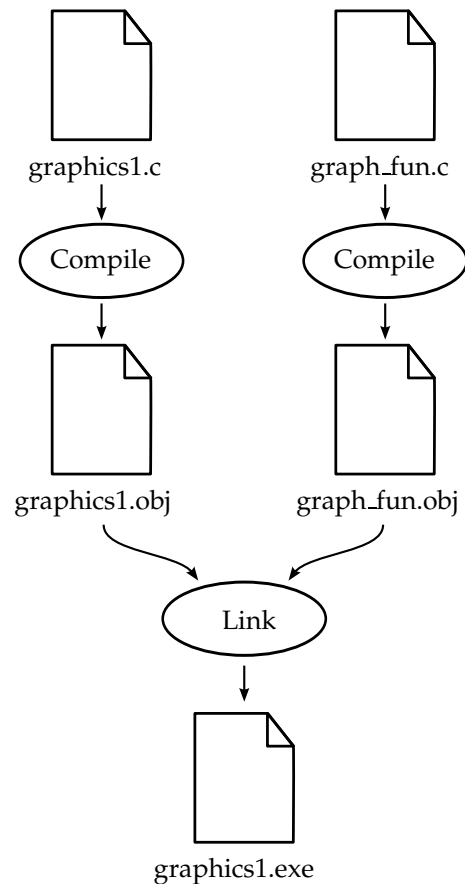
Repeat the process until you have the file structure that you decided on. If you want to remove a file from your project, select the file in the source file tree and press the “Delete” key. This **will not** actually remove the source file from the folder. But it will remove it from the project, so it is no longer part of your program.

You should now have a set of source files, which are all part of your program, and are all shown in the source file tree on the left-hand side of the IDE window. Each function in your program should only be in **one** of these source files.

Unfortunately, you are not yet ready to build your program. At the moment there is no way for a function in one file to know about a function in another file. To understand this, we must look at how projects are compiled and linked. You should remember from earlier labs that programs are built in two stages:

- **compilation** which produces an object file for each source file;
- **linking** which joins all of the object files together to make one executable.

The important part of this process is that the compiler only ever works on one source file at a time. This is shown in the diagram below:

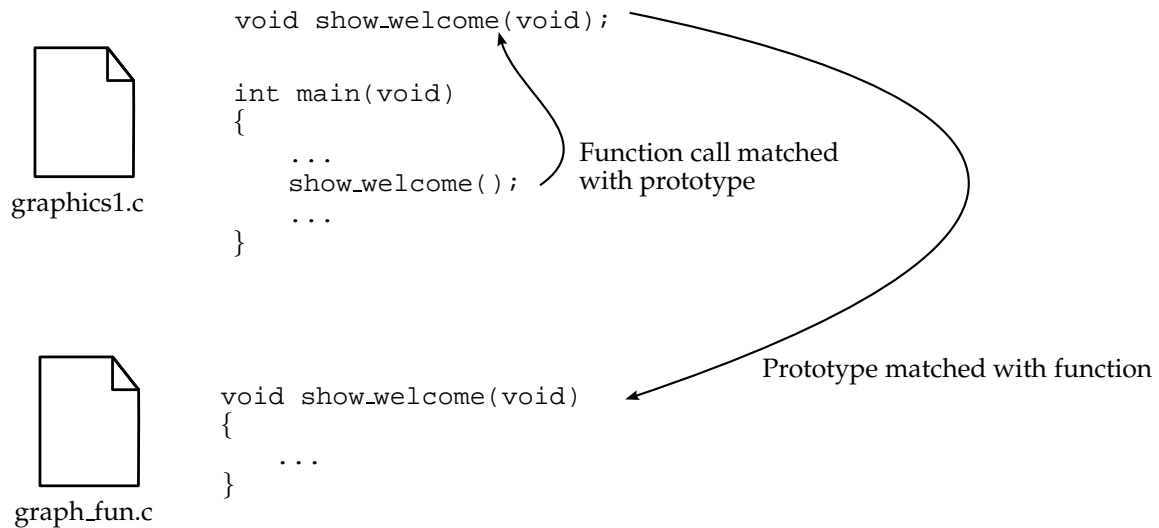


In this example, the “graphics1” project is being built. There are two source files: “graphics1.c” and “graph_fun.c”. The compiler first works on the “graphics1.c” file and produces the intermediate file “graphics1.obj”. It then works on the file “graph_fun.c” and produces the intermediate file “graph_fun.obj”. The compiler has now finished its work. The linker now runs. It takes the intermediate files “graphics1.obj” and “graph_fun.obj” and combines them together to make the executable file “graphics.exe”. The files in your project will probably have different names, but the principle is the same.

To make sure that you don’t get compiler errors or warnings, you must make sure that for each function that is used in one file, and declared in another, you must specify a prototype for that function. So, for example, in the example above:

- say “graph_fun.c” defines a function called `show_welcome`, which is called by the `main` function in “graphics1.c”;
- for the compiler to know that the function exists in another file, there must be a prototype for `show_welcome` in “graphics1.c”.

This situation is shown in the diagram below:



Exercise D.2: Declaring the Correct Prototypes

You should now have created the files that you decided on, and moved the parts of your source code to the relevant files.

Make sure that each source file you have created contains the prototypes of each of the functions it needs that are in other files. It is standard practice to put a comment at the top of each file documenting what the purpose of the functions in the file are. Make sure you add a comment like this to the top of each file that you create.

Your program should now build and execute without warnings or errors.

D.3 Creating Your Own Header Files

To decide what prototypes are necessary you have to check your source file to find out what functions (if any) you use from other files. When your source files are large it is easy to forget which functions are called. Including prototypes which are not called will not generate an error or warning, but it is misleading for anyone reading your source code. The need to include prototypes for each function called in another file is inconvenient when you want to use one source file in more than one project.

The easiest way to solve these problems is to place the prototypes for all the functions in particular source file into a header file with the same name. For example, assume that a source file called "graph_fun.c" contains the following functions:

- show_hello
- draw_stick_person
- draw_throw

To accompany this source file a header file called "graph_fun.h" should be created. A header file is just like a source file, except that the code that it contains shouldn't be code that actually executes. For example, the header file "graph_fun.h" will only contain function prototypes. The function prototypes will be used by the compiler and linker to match up the functions in "graph_fun.c" to other files in which they are called. The function prototypes themselves are not converted into executable code.

In our example, the header file "graph_fun.h" should contain the prototypes for the three functions in "graph_fun.c". For example, the contents of "graph_fun.h" might be as follows:

```
/*
 * Header file for graph_fun.c
 * Function prototypes only
 */

void show_hello(void);
void draw_stick_person(int x, int y);
void draw_throw(int x, int y, double velocity);
```

As you can see, header files can be very short!

In the file “graphics1.c”, which uses the functions `show_hello`, `draw_stick_person` and `draw_throw`, the header file must be `#included` with the line

```
#include "graph_fun.h"
```

There is now **no need** to put any prototypes for the functions `show_hello`, `draw_stick_person` and `draw_throw` in “graphics1.c”. If we wished to use any of these functions in another file, we would simply put the `#include` directive at the top of the file. You will remember from laboratory B that the `#include` directive just includes the source from the file that is named into the current file. In this case it is used to make sure that prototypes for the functions in “graph_fun.c” are included wherever they might be used.

Creating a new header file is very similar to creating a new source file. Choose “New” from the “File” menu: However, when you save it you should make sure that your file name ends in “.h”.



Exercise D.3: Creating Header Files

You should now create a header file for each one of your source files which contains functions that are used by another file. For example, if the file “music1.c” contains functions that are called **only** by other functions inside the “music1.c” file, there is **no need** to create a header file for it.

Use appropriate `#include` lines to allow you to remove function prototypes from your source files. Your program should still build and execute correctly.

D.4 Protecting Your Header Files

You have just created header files which allow you to encapsulate a set of functions in a source file in a way that makes them easy to use in other source files. Sometimes header files will have `#include` directives in them which include other header files. These header files may also have `#include` directives and so on. The process can become very complicated. In these situations two problems can occur:

- a *circular reference* can occur. This is where one header file includes another header file which includes the original header file. The compiler would then travel round this loop forever (actually it would run out of memory and crash, which is probably worse).
- a header file can be included into the same source file twice. If the header file only contains prototypes this is not a problem because you can declare the same prototype as many times as you want (as long as the definitions are the same). Later in the course you will put more complicated things in header files which can only be seen by the compiler once for each source file it processes.

Both of these problems can be solved if we make sure that the contents of a header file are seen **only once** by the compiler for each source file. We can do this by using the `#define`, `#ifndef` and `#endif` preprocessor directives you met in laboratory B.

Let us take the example of a header file called “graph_fun.h”. At the top of the header file there should be a line which checks to see whether a label has been defined. The label **must be unique** to the header

file. To try and ensure that the label is unique the name of the file is often used in some way. It is most common to use a label which is made of two underscores (“_”), the name of the file (except for the “.h” extension) another underscore, the letter ‘H’ and finally another two underscores. As with all preprocessor labels, it is good programming practice to use only UPPERCASE letters for these labels. For example, for the header file “graph_fun.h” we would use the label `_GRAPH_FUN_H_`. We would check whether it is defined using the preprocessor directive `#ifndef` like this:

```
#ifndef __GRAPH_FUN_H__
```

If the label has not been defined then the preprocessor will continue reading the contents of the file. To make sure that the file does not get processed again, the first thing we do is define the label using the `#define` directive like this:

```
#define __GRAPH_FUN_H__
```

At the end of the source file we must include a `#endif` directive to match the `#ifndef` directive, like this:

```
#endif
```

Now that the compiler has seen the contents of the header file once the label `_GRAPH_FUN_H_` has been defined. If the header file is included again, the compiler will reach the line:

```
#ifndef __GRAPH_FUN_H__
```

where it will find that the label `_GRAPH_FUN_H_` has already been defined. It will therefore skip to the `#endif` directive at the end of the source file. So the contents of the header are only processed **once** no matter how many times the file is included.

The example header file that we looked at earlier, “graph_fun.h” should look like this:

```
/*
 * Header file for graph_fun.c
 * Function prototypes only
 */

#ifndef __GRAPH_FUN_H__
#define __GRAPH_FUN_H__

void show_hello(void);
void draw_stick_person(int x, int y);
void draw_throw(int x, int y, double velocity);

#endif
```

This system of protection, made up of a `#define` directive, a `#ifndef` directive and a `#endif` directive is called a *sentry*.

Exercise D.4: Adding Sentries to Your Header Files

Make sure you understand how sentries work, and why they are useful. If you need any help, ask one of the demonstrators.

Include the correct preprocessor directives to make a sentry in each of your header files. Your program should still build and execute correctly.

D.5 Good Programming Practice: Some More Tips

This course has stressed, many times, the importance of creating source code that is easy to read and understand. The best way to achieve this is:

- structure your code logically into functions which allow for reusability;
- structure your functions logically into different source files;
- use header files so that your source files are easily reused and are not cluttered with prototypes;
- document your source code with plenty of comments that are aimed at someone who understands C but does not necessarily understand how your program works.

You can make your easy to read through careful use of *indentation*. Indentation is about changing where a line of code in your source file has its left-hand edge. It is very useful for making the structure of code inside functions clear. You should control where the left-hand edge of your lines of source code are using the “Tab” key. Whenever you start a new structure, like an `if` statement or a `while` loop, you should indicate which lines are part of this structure by moving the lines to the right by inserting a tab character at the start of each line.

For example, an `if` statement should always be structured like this:

```
if (number_entered < 10)
    printf("That's less than 10\n");
```

Notice how the `printf` line is indented by one tab character relative to the `if` clause. The next line after this one is not part of the `if` statement and so should start in the same vertical position as the `if` clause.

The editor in the IDE will try to carry out the correct indentation for you automatically. Sometimes, however, it may get it wrong. You can move a line to the right by placing the text cursor at the beginning of the line and pressing “Tab”, which inserts a tab character. You can move the line left by holding down the “Shift” key and pressing “Tab”.

In the IDE editor, this also works with blocks of code. If you highlight a block of code, like this:

```
scanf("%d", &number);
/* Get access to the file by opening it */
file_handle = fopen("number.txt", "w");
/* Write a single number to the file */
fprintf(file_handle, "%d", number);
/* We're done with the file, close it */
fclose(file_handle);
```

and press the “Tab” key, the editor inserts a tab character at the beginning of every line in the block. The block of code therefore moves to the right:

```
scanf("%d", &number);
/* Get access to the file by opening it */
file_handle = fopen("number.txt", "w");
/* Write a single number to the file */
fprintf(file_handle, "%d", number);
/* We're done with the file, close it */
fclose(file_handle);
```

You can move a block of code to the left by highlighting it and holding down the “Shift” key when you press “Tab”.

Exercise D.5: Cleaning Up Your Code

To make sure your code is easy to understand you should:

- make sure you have commented your code thoroughly. If you are unsure, ask a friend to have a look at your code and see if they can understand what it does and how it works.
- make sure your indentation is correct and consistent. If you are unsure about this, check with one of the demonstrators.

There are no hard and fast rules for how to lay out and comment your source code in C, although there is some generally accepted good practice. With things like indentation it is most important that you are **consistent** across all of the source code in your project.

D.6 Summary

Now that you have finished this laboratory you should have either a music or graphics project that is split into multiple files in a logical way. It should use header files, protected by sentries, to allow easy use and reuse of the code you have written. You should understand the reasons for wanting to structure your code into multiple files, and how to apply these techniques to other projects.

You should also have spent some time making sure that your code is well-formatted, well-commented and easy to read.

You are now ready to tackle larger projects.