Guide To C Files And H Files
by Jacob “Bob” Egner

Introduction

This document explains the philosophy of C and H files, and what to put in each file type. At times, I'll stray from the main topic to talk about C compilation in general. The more you understand about C compilation, the less trouble you'll have getting your programs to compile and work.

Along with this document, you should also have the files heap.h, heap.c, and heap_test.c. This project can be found in the starter files section of the web site. heap.c and heap.h make up a “module” that implements a memory heap. heap_test.c uses the heap module. I wrote these files to be examples for this document, bring dynamic memory management to the 9S12, and mostly for fun. Please glance over these files before proceeding. The project can be found in the “Starter files” section of the course web site.

Why Do We Have H Files?

One key thing in understanding C and H files is that declaration and definition are two different things. A declaration tells the compiler that something exists and what kind of beast it is. A definition tells the compiler what it is. A function declaration tells the name of a function, what arguments it takes and what it returns. A function definition also has all that and the code that implements the function. A variable declaration tells the type of a variable. A variable definition tells the type and actually allocates space for the variable. Yes, **a definition is also a declaration. Variables and functions can have many declarations (as long as they agree) but they can only have one definition.**

H files are the accepted way to tell the compiler about functions, variables, and types that are outside a file. The compiler needs to be told about these functions, variables, and types outside of a file before you can use them inside the file. **H files have declarations of public functions, not definitions. C files have definitions.**

The `#include` directive might seem magical, but it is not. Look at heap_test.c; you will see that heap_test.c has a `#include “heap.h”` at the top. This `#include` directive makes the preprocessor dump the text of heap.h into heap_test.c. So, as far as the compiler is concerned, there's no difference between #include'ing heap.h and having the text of heap.h directly in heap_test.c. This behavior of `#include` is important.

In heap.c on line 26, the variable Heap is defined. Let's imagine you are writing a program that has many C files that use this heap module. If all of these C files had a `#include “heap.c”` in them, that would mean that the variable Heap would be allocated multiple times. This is not what we want! We don't want multiple Heap variables floating around and taking up space, we only want one. The problem here is that we'd have multiple definitions of the Heap variable. We'd also have multiple definitions of the heap functions. Thankfully the compiler would give many errors in such a situation.

Also, think of H files as the public interface of a module. heap.c is the domain of the developers of the heap module. heap.h is the domain of the users of the heap module. heap.h has all the information you need to use my heap module. If my heap module was rewritten as a Java class, the functions, constants, and variables in heap.h would be declared as public and the rest in heap.c would be declared as private. This issue of public versus private and hiding implementation details is critical for designing effective software..

Also, it is important that the C file in a module #include's the corresponding H file. That way, the C file can pick up all the declarations in the H file.

Functions

Never define a function in a H file. Function definitions belong only in C files. For functions that you want to be accessible from other files, also put a function declaration in the H file. Let's digress a bit and discuss the `static` keyword. The keyword `static` has more than one use in the C programming language:

1) When `static` is put in front of a function, it restricts the scope of the function to the file the function is in (making it private).
2) When `static` is put in front of a global variable (meaning the variable was defined outside of a function), that restricts the scope of the variable to the file it is in (making it private).
3) When `static` is put in front of a local variable (meaning the variable was defined inside of a function), that does something...tricky. The variable's scope is local to the function, but the variable is stored like a global variable and the variable's value persists through multiple invocations of the function.

So, the `static` keyword makes functions and global variables private. Please use the `static` keyword in declarations and definitions of private functions and global variables.

In heap.h, the functions like Heap_Malloc are declared, so any file that has a `#include heap.h` will be able to use those functions. heap.c has all the function definitions. You'll notice that at the top of heap.c, there are function declarations for the private functions of the heap module. This is because C compilers are usually one-pass compilers. They do a single pass over your source code, so if you call a function before that function has been declared or defined, the compiler won't know what that function is. Thankfully C++, Java, and other languages are not made to be parsed by one-pass compilers.

For example this will compile (using gcc):

```
void blah() {}
int main() {blah(); return 0;} // compiler has already seen blah, we're okay
```

But this will not:

```
int main() {blah(); return 0;} // blah? what's blah? error!
void blah(){}
```

But this will:

```
void blah(); // function declaration – definition comes later
void blah(); // doesn't conflict with any other declarations
void blah2(); // there is no blah2(), but okay as long as we don't actually use it
int main() {blah(); return 0;} // compiler knows blah from the declaration
void blah(){} // function definition
```

Variables

You should never define a variable in a H file. Variable definitions go in C files. Variable declarations go in H files. Currently, the Heap variable (near the beginning of heap.c) is private. Let's say we wanted the `Heap` variable to be public and accessible in other files (like heap_test.c). We do not want to put `int Heap[HEAP_SIZE_WORDS];` in heap.h. Doing so would create the same problem that was discussed in the “Why do we have...” section – multiple definitions of the Heap variable. The definition of the Heap variable should stay in heap.c.

So how do we declare a variable without defining it? We use the `extern` keyword. The `extern` keyword tells the compiler that a variable is being defined (and thus allocated) elsewhere.

You can also put the `extern` keyword in front of a function declaration, but is completely unnecessary – don't do it.

In general, using public variables is poor style. However if you really need to implement a public variable, we declare the Heap variable using the `extern` keyword in the header file. We *should not* put `extern int Heap[];` in files like heap_test.c. Instead, that declaration should go in heap.h. Just like it is the duty of a H file to advertise public functions, it is the duty of a H file to advertise public variables. Don't use the `extern` keyword in C files.

#define

The `#define` directive is often used for defining constants and macros. The rule of declarations in H files and definitions in C files does not apply here. Instead, **let the rule of public stuff in H files and private stuff in C files guide you**. For instance, in heap.h, there are constants like `HEAP_SIZE_BYTES` and macros like `HEAP_SIZE_WORDS`. Users of the heap module should be able to see (and set) the size of the heap, so `HEAP_SIZE_BYTES` and `HEAP_SIZE_WORDS` were put in heap.h. heap.c has the constant `HEAP_START` because I don't want the users of the heap module to know where the heap is located.

Types

A module often defines data structures to be used alone with the public functions. Types for these structures (like structs and unions) should be declared in H files. However, some modules define data structures only to be used internal to the module. Types for these private data structures should be defined in the C file. Notice that I declare the `heap_stats_t` type in heap.h. Any file that #include's heap.h can use the `heap_stats_t` type. For data structures that are public, put typedefs or struct type declarations in the H file. To be more precise, I actually *defined* the heap_stats_t type in heap.h, and that means I'll get an error if I try to define it again in the same file, even if the second definition agrees with the first.

As a side note, I used the `typedef` keyword to allow people to use my type without having to use the `struct` keyword every time.

#if, #ifdef, #ifndef, #endif

This section can help you avoid certain compilation pitfalls. Near the top of heap.h, there is `#ifndef HEAP_H` and at the very bottom, there is a `#endif`. Basically, this says to the preprocessor, “if HEAP_H is undefined, then let the compiler see all the code between here and the appropriate `#endif`, otherwise don't let the compiler see that code”. We also have a `#define HEAP_H` in that code to make sure that HEAP_H gets defined. This makes sure that when we `#include heap.h` from many files, the compiler only sees the code from heap.h once.

H files have declarations and multiple declarations are okay as long as they agree, so why do we care that the compiler only sees them once? Well, sometimes H files contain type definitions, and definitions can only appear once.

The `#if` and `#endif` directives can be used in a more general way. The conditional of the `#if` directive can be pretty much anything that can be evaluated before compilation. Try to learn about the C preprocessor directives. Definitely learn about all the C keywords.

Summary

Declarations in H files, definitions in C files. Public stuff in H files, private stuff in C files. Actual function code goes in C files. The `extern` keyword should only appear in H files. Data

structures used both externally and internally within the module are public, and the type definition should be declared in the H file. Data structures used only internally within the module are private, and the type definition should be declared in the C file. Public functions should be declared in H files. The C language allows multiple declarations (as long as they agree) but only one definition. Investigate language features you don't fully understand. The `#include` directive just dumps text from a file into the current file.