Contents

<Sun 12.24.23> Note: so far this file only details what was covered in meetings. I will add much more content to it later.

1 The C compilation process

If you are writing your program entirely in one file, compiling your program with this shell command will do:

```
cc file.c -o executable_name
```

If you split your code across *multiple* **source files** (.c files) and **header files** (.h files), however, you may want to understand the compilation process at a deeper level. That is what this document will detail.

1.1 Translation units and object files

A **translation unit** or **compilation unit** is a collection of source code that the compiler reads to output an object file. A translation unit is made up of a source file and all of the header files it may include.

An **object file** (.o file) contains the machine code of a translation unit. One or more object files are fed to the **linker**, a program which will link data,

such as function calls between object files, to output the final executable of a program.

1.2 Producing direct object files

When you run this command:

```
cc file.c -o executable_name
```

The compiler will produce object files and link them automatically for you. To only produce object files and skip linking, use the `-c` flag:

```
cc -c file.c
```

This will produce the object file `file.o`. If you want to explicitly choose the object file name, use the `-o` flag:

```
cc -c file.c -o file.o
```

1.3 Linking object files to produce an executable

To produce an executable using all of your object files, run this command:

```
cc file.o -o program
```

This command will produce the executable file `program`. You may insert as many object file names before `-o` as you like.

2 Linkage

Linkage is a property of variables and functions that determines if they can be accessed in different source files that are to be compiled and linked into one program.

There are two types of linkage: internal and external.

Variables and functions with **internal linkage** can only be accessed in the translation unit they are defined in. See here to learn how to declare variables and functions with this linkage.

Variables and functions with **external linkage** can be accessed in every translation unit.

2.1 Using functions across different source files

2.1.1 Demonstrational code

To use *any* function or variable across different source files (which will be included in different translation units), the function or variable needs to have external linkage.

By default, all functions have external linkage.

Here is some code to illustrate this:

This is my file `sum.c`, which contains the definition for the function `sum`:

```
// Returns the sum of two integers
int sum(int x, int y)
{
        return x + y;
}
```

This is my file `main.c`:

```
#include <stdio.h>

// Returns the sum of two integers
int sum(int x, int y);

int main(void)
{
        printf("The sum of 2 and 3 is %d\n", sum(2, 3));
        return 0;
}
```

I will use both of these source files in the same program, which I will compile with the following shell commands:

```
gcc -c sum.c
gcc -c main.c
gcc sum.o main.o -o program
```

This should work perfectly fine. Functions have external linkage by default, therefore `sum` has external linkage so I am able to call it from `main.c` even if it is defined in `sum.c`. The program links successfully with the execution of the last command because the linker was able to find the definition of `sum` in `sum.o`.

If I left `sum.o` out of the final compile command, I would see a linker error such as this:

```
/usr/lib/gcc/x86_64-pc-linux-gnu/13/../../../../x86_64-pc-linux-gnu/bin/ld: main.o: in
main.c:(.text+0xf): undefined reference to ‘sum’
collect2: error: ld returned 1 exit status
```

It is the **linker**'s job to handle linkage of variables and functions, not the compiler's job. That is why we get a linker error. You can typically tell a linker error apart from a compiler error because linker errors contain some reference to the program `ld`, which is the linker on UNIX.

2.1.2 The problem with our code and the solution

While our example code *does* work, it has some flaws that become apparent when we begin adding more to our program.

What if we wanted to include more source files that used the `sum` function? We would need to copy and paste the function prototype of `sum` in each of these files. That would be a violation of the DRY principle: don't repeat yourself. Violating DRY means your program will be harder to modify. If we wanted to change the arguments of `sum`, we would need to find every occurence of `sum`'s prototypes and change them as well!

The solution is to store the function prototype for `sum` in a header file, which we will cover in the next section.

2.2 Header files

Header files (ending in .h), or **headers**, are files that contain commonly re-used code to be `#include`'d at the top of other header or source files in our program, hence the name.

As we learn more about different C concepts, we will need to know whether the lines of code they involve should go in header or source files. Generally, code statements (instructions that cause the computer to actually execute something) go in source files, and information for the compiler to use, such as function prototypes, goes in header files.

2.2.1 Using function prototypes in header files

Function prototypes for functions with external linkage are best placed in header files.

1. Demonstrational code Building on our previous code for demonstrating use of functions with external linkage, I will create a new header file `sum.h`:

   ```
   // Returns the sum of two integers
   int sum(int x, int y);
   ```

   Now, instead of writing the raw function prototype into `main.c`, I can simply `#include sum.h` in `main.c`:

   ```
   #include <stdio.h>

   #include "sum.h" // For sum()

   int main(void)
   {
           printf("The sum of 2 and 3 is %d\n", sum(2, 3));
           return 0;
   }
   ```

   It is a convention to name your header files after the source file they are related to. I named the header file `sum.h` because it contains the function prototype for `sum`, whose function definition is in `sum.c`.

2. #include paths Notice how I used double quotes around `sum.h` in the `#include` statement. These indicate that `sum.h` is a relative path. Any path enclosed in double quotes indicates a path relative to the directory that the file containing the `#include` statement is in.

   Any path enclosed in angle brackets `<>`, such as `stdio.h`, is relative to the include directories available to the compiler. The compiler will search for the named file and error if it isn't found in any of the include directories.

   On UNIX, `/usr/include` is one of these directories. You can add to this list of directories by running the compiler with the `-I` option. For example, passing `-I.` will add the working directory to the list of include directories.

   See this document for an explanation of relative vs absolute paths.

3. #include comments I added the comment "For sum()" after `#include`'ing `sum.h` to make my intentions clearer. I recommend that you do the same in commenting what you need from each header file.

   The more code we add to `sum.h`, the harder it is to tell which functions need to be used by which source files. Also, if we want to move a function prototype from `sum.h` to some other header file, our comments will let us know which files need their `#include` statements altered accordingly.

4. The benefits of header files Header files allow us to follow the DRY (don't repeat yourself) principle. With them, we can avoid rewriting the same function prototypes where they are needed. We can also update our function prototypes everywhere they are used by just changing them in one file.

2.2.2 Header guards

A **header guard** or **include guard** is a mechanism that prevents a header file from being included more than once in one translation unit. Not using one can cause errors.

So far, we only know to add function prototypes to our header files. No errors occur if these are repeated, however, code that we will later learn to place in header files *can* cause errors when repeated.

1. Traditional #ifndef ... #endif guards The standard way to implement header guards is to use the `#ifndef`, `#define` and `#endif` preprocessor statements like so:

   `sum.h`:

   ```
   #ifndef SUM_H
   #define SUM_H

   // Returns the sum of two integers
   int sum(int x, int y);

   #endif
   ```

   The `#ifndef` statement is short for "if not defined." It checks if the symbol it is given (in this case `SUM_H`) is not defined. If it isn't, the

section of code spanning from just after `#ifndef` to the next `#endif` is kept in the file. If the symbol *is* defined, this section of code is deleted.

When `sum.h` is included for the first time in a translation unit, `SUM_H` shouldn't be defined, so the code between `#ifndef` and `#endif` is kept. The next time `sum.h` is included, `SUM_H` should already be defined, so the rest of the code in the header file is ignored.

You can name the symbol that `#ifndef` checks whatever you want, as long as you're sure it won't be defined before the header file is included for the first time. A common convention is to name it the header file's filename in all uppercase, replacing `.h` with `_H`.

Be careful not to reuse the same symbol for different header files. If you have two header files with the same filename in your project, but they are stored in different directories, you may want to include the directory name in the symbol in the header guard. For example, a file `a.h` in the directory `b` could use the symbol `B_A_H`.

FYI, there is an `#ifdef` preprocessor statement you can use, which does the same thing as `#ifndef` but keeps the next code section if the symbol *is* defined.

2. #pragma once You can use the preprocessor statement `#pragma once` to implement a header guard:

   `sum.h`:

   ```
   #pragma once

   // Returns the sum of two integers
   int sum(int x, int y);
   ```

   This is easier to type than a traditional header guard and lets you skip having to name a symbol to be used for the `#ifndef` statement. A downside to this approach, however, is that the `#pragma` statement is not standardized in the C language, so you need to verify that the compiler you use supports it.

2.3 Using global variables across different source files

All global variables have external linkage by default. To access a global variable from a source file where it is not declared, you need an extern declaration.

An **extern declaration** tells the compiler about the potential existence of a global variable that could be in another translation unit of the program. It is not an actual declaration of a variable. All an extern declaration does is stop the compiler from erroring when it sees use of a variable that wasn't declared in the current translation unit.

To write an extern declaration, simply write a declaration statement for your global variable and prepend it with the `extern` keyword.

```
// Extern declaration
extern int g_x;

// Declaration of global variable g_x
int g_x;
```

Declaration of global variables should go in source files. Extern declarations should go in header files.

2.3.1 Demonstrational code

This program will demonstrate accessing a global variable from a source file where it isn't declared. It will contain three files: `x.h`, `x.c` and `main.c`.

`x.h`:

```
#ifndef X_H
#define X_H

// This variable is used to test external linkage of global variables
extern int g_x;

#endif
```

`x.c`:

```
#include "x.h"

// This variable is used to test external linkage of global variables
int g_x;
```

`main.c`:

```
#include <stdio.h>

#include "x.h" // For g_x

int main(void)
{
        printf("The value of g_x is %d\n", g_x);
        return 0;
}
```

You can compile this program with the following shell commands:

```
gcc -c x.c
gcc -c main.c
gcc x.o main.o -o extern_test
```

2.4 Using structs across different source files

Struct definitions should be placed in header files. To use global variables with struct types across different source files, create extern declarations like you would with other global variables.

See struct.c for general notes on structs.

2.5 The static keyword in linkage

The `static` keyword has different meanings in different contexts.

- When used on functions and global variables, `static` affects linkage
- When used on local variables, `static` affects variable lifetime

In this section, I will cover how `static` affects linkage. See static.c for its use on local variables.

`static` makes functions and global variables have **internal linkage**. To use `static` with a global variable, add `static` before the type in the declaration or initialization statement. To use `static` with a function, add `static` before the return type like so:

```
static int sum(int x, int y);
static int g_x;
```

You may want to use `static` for functions and global variables that you only need to use in one file.

Since `static` symbols don't have external linkage, you shouldn't put them in header files.

3 Sources

- For Linkage
  - geeksforgeeks.org
  - microsoft.com