Contents

1 The C compilation process **1**
1.1 Translation units and object files 1
1.2 Producing direct object files 2
1.3 Linking object files to produce an executable 2

2 Linkage **2**
2.1 Using functions across different source files 3
2.1.1 Demonstrational code 3
2.1.2 The problem with our code and the solution 4
2.2 **TODO** Using header files 4
2.3 **TODO** Using global variables across different source files . . . 4
2.4 **TODO** Using structs across different source files 4
2.5 **TODO** The static keyword in linkage 4

3 Sources **4**

<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 test
```

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 TODO Using header files

2.3 TODO Using global variables across different source files

2.4 TODO Using structs across different source files

2.5 TODO The static keyword in linkage

3 Sources

- For Linkage
  - geeksforgeeks.org
  - microsoft.com