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 Build systems

Compiling and linking an entire program is known as **building**.

A **build system** is a system that handles the execution of compiler commands needed to build a program, usually to automate the process.

In this section, we will explore various ways to set up build systems. As of <*Sat 2.3.24*>, this section only details build systems that work on UNIX-like OSes such as macOS and GNU/Linux. You can use these build systems on Windows if you run them through MSYS2.

3.1 Shell scripts

In my opinion, a shell script is the easiest-to-create and most basic build system. Depending on how you write them, they can make for fairly rudimentary build systems, but if your project is small in scope and its number of source files, you may get by with one.

On UNIX, a **shell script** is a file in which each line of text is a shell command that gets run. Like C, execution runs from top to bottom.

3.1.1 Writing a shell script

To write a shell script, open a new text file and name it whatever you want. Shell script filenames typically end in `.sh`.

On the first line of your script, we must write a **shebang**, which specifies what interpreter will execute the rest of the contents of the file. An **interpreter** is a program that reads code in text form and executes it line by line. To write the shebang, write `#!` followed by the absolute path to the interpreter program:

```
#!/bin/bash
```

Since we are writing a shell script, I picked `bash`, the Bourne-Again SHell, as our interpreter. It is available on most Linux distros.

Now, we can start writing the commands we want the script to run, line by line:

```
#!/bin/bash

# Test some stuff
echo line 1
echo line 2
echo line 3
echo Hello World!
```

Lines that start with `#`, besides the shebang, are comments.

3.1.2 Running a shell script

To run our shell script, we can type the path to it on the shell, starting with `./`, just like we would when running any other program.

You will probably get this error, though, if your shell script was newly created:

```
bash: ./script.sh: Permission denied
```

In this case, we need to give executable permissions to our file so we are able to execute it. You can do that with the `chmod` command:

```
chmod +x script.sh
```

Replace `script.sh` with the file you want to give executable permissions to.

Now you should be able to run your script.

3.1.3 Writing a shell script to build a program

We can make our shell script into a build system by populating it with compiler commands.

If we wanted to build our external linkage demo program, for example, we could use this script:

```
#!/bin/bash
gcc -c sum.c
gcc -c main.c
gcc sum.o main.o -o program
```

Since the ultimate goal of the script is to build the program and we don't need the individual object files for anything, we could just compile and link everything in one command like so:

```
#!/bin/bash
gcc sum.c main.c -o program
```

3.1.4 Problems with this setup

The biggest issue with this shell script build system is that it re-compiles every source file in the program each time it is run. This can be problematic if we just made a small change to one source file and wanted to re-link the program. All that we need to do to re-link the program is recompile the one source file we changed and re-link, however, this script compiles all source files and links in one command, so we would have to run the compile commands on our own to get what we want. This issue will only be exacerbated as we add more source files to our program.

Still, though, if your program is fairly small, using a shell script like this as a build system is fine.

There also may be some way to use shell scripts to create more elaborate build systems than this, but I don't know enough about shell scripting to do that.

3.2 Makefiles

I have learned most of what I know about makefiles through this tutorial website: Makefile Tutorial By Example. It is very well written, possibly more so than my own writing, so check it out if you want. If you like the way I explain things, read on.

3.2.1 A basic intro to makefiles

A **makefile** is a text file that gets read by the UNIX utility `make`, which is used to run commands used to maintain groups of files that depend on each other. Most people use makefiles as build systems, but it's possible to use them for other purposes as well.

When you run `make` with no additional arguments, it will automatically try to find a makefile in the working directory named `makefile` or `Makefile`. If you use GNU make, it will also search for a makefile named `GNUmakefile`. It is recommended that you only name your makefile this if it uses a feature of GNU make that isn't present in other versions of `make`.

The goal of using makefiles as build systems is to create rule

1. Syntax In a makefile, lines starting with `#` are comments.

2. Rules

   (a) Syntax of a rule A **rule** is a section of a makefile that specifies what file(s) a file depends on to be created and what shell commands must be run to create the file. Rules follow this syntax:

   ```
   target: prerequisites ...
           recipe command 1
           recipe command 2
           recipe command 3
           ...
   ```

   The **target** is the file that needs to be created.

   The **prerequisites** are one or more files, separated by a space character, that must exist for the target to be created. If any of the prerequisites are modified since the last time the target was created, the target will be re-created.

   The **recipe** is the sequence of shell commands that are meant to create the target. Each command exists on its own line, tabbed up once from the target. The recipe ends when the tabulation stops.

   (b) How rules are processed The first rule that gets processed depends on how `make` is run. If you run `make` with no additional arguments, the first rule that gets processed is the first rule that appears in the file when its read from top to bottom. If you run `make` with arguments that don't start with `-`, `make` will attempt to process the rules whose targets are the names of the arguments.

   Before the recipe for a rule is run, `make` will ensure that the prerequisites have been created by processing the rules whose targets are the prerequisites.

   (c) Demo Here is a demonstrational makefile to build our external linkage demo program:

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

   main.o: main.c sum.h
   ```

```
        gcc -c main.c

sum.o: sum.c sum.h
        gcc -c sum.c
```

Running `make`, I get this output:

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

`make` will print every recipe command it runs to standard output. To suppress this, you can add `@` to the start of the recipe commands you don't want to be echoed.

i. Pros of this build system
   - Unlike our shell script, it will only re-compile the source files whose dependencies changed since the last build
   - We can directly specify what files we want to create (e.g. running `make main.o` will run the recipe for `main.o`)

ii. Cons of this build system
   - We have to write rules to create object files manually
   - We have to keep track of which object files depend on which headers

iii. My personal thoughts on this build system Despite the cons, I personally think that this build system is better than our shell script. We can remedy having to manually write rules for object files by writing a makefile that automatically finds the source files used in our program and writes rules for them. I will later explain how to write one here, but don't skip ahead to that without reading the rest of the content in this section, as it builds on such content.
   Even without a makefile that automatically generates rules for source files, you can get by just fine. For example, see the makefile for dwm, a somewhat popular minimal window manager for X11.

3. Variables We can use variables in makefiles to store and reuse text. Common variables to create include:

   `CC` the C compiler to use

`CFLAGS` flags to pass to the C compiler when compiling source files

`LDFLAGS` flags to pass to the C compiler when linking object files

I personally like creating a variable for the program name named `BIN`.

To create a variable, write its name, then ":=" and its value. To access a variable, use `$(VARIABLE_NAME)` or `${VARIABLE_NAME}`.

Just like in C, variables make our code easier to modify and help us follow the DRY principle.

(a) Demo Implementing variables into our first makefile demo, our new makefile may look like this:

```
CC := gcc

# The -g flag includes debugging information in object files
CFLAGS := -g

# This stores no value
LDFLAGS :=

# Binary to build
BIN := program

$(BIN): main.o sum.o
        $(CC) main.o sum.o -o $(BIN) $(LDFLAGS)

main.o: main.c sum.h
        $(CC) $(CFLAGS) -c main.c

sum.o: sum.c sum.h
        $(CC) $(CFLAGS) -c sum.c
```

4. Phony targets It is common practice to create rules that exist to easily run recipes rather than generate target files.

   An example of this is creating a rule for the target `clean`, which will be used to store a recipe that removes the files generated during building:

```
clean:
        # The asterisk (*) will match anything, so "*.o" will expand to all files
        # Be very careful when using * to remove files, as you wouldn't want to re
        rm *.o
```

Since `clean`'s recipe doesn't actually generate a file named `clean`, `clean` is a **phony target**.

The problem with this is that, if a file named `clean` did exist, the recipe wouldn't run. To mark `clean` as a phony target so that it will run regardless, we can create a rule with the special traget `.PHONY` and add `clean` as a prerequisite:

```
.PHONY: clean
```

(a) Demo Adding a `clean` phony target to our second makefile demo, we get this:

```
CC := gcc

# The -g flag includes debugging information in object files
CFLAGS := -g

# This stores no value
LDFLAGS :=

# Binary to build
BIN := program

$(BIN): main.o sum.o
        $(CC) main.o sum.o -o $(BIN) $(LDFLAGS)

main.o: main.c sum.h
        $(CC) $(CFLAGS) -c main.c

sum.o: sum.c sum.h
        $(CC) $(CFLAGS) -c sum.c

.PHONY: clean

clean:
        rm *.o
        rm $(BIN)
```

3.2.2 TODO Crazy automated makefile

In this section, I will explain how to create a makefile that automatically finds the source files in a given directory and generates rules for them. The makefile we create will be similar to the one I use to build my game SoupDL 06.

4 Sources for these notes

- For Linkage
  - geeksforgeeks.org
  - microsoft.com
- For Makefiles
  - Makefile Tutorial By Example
  - GNU Make Manual