How to Structure C Programs

Sophie Coudert
Télécom Paris

November 14, 2024

VERSION 0.

Prerequisite: You are supposed to have some notions of C types (e.g., `int`), to have written at least a few (even small) C functions, and be able to handle C variables. You are also supposed to know that human written C code is not directly executable by the machine: Compilation transforms the source C code into machine code, often referred to as "binary". Only these binary files can be directly executed. These binary executable files are not directly readable by humans.

Preliminary: `#include "filename"` or `#include <filename>` means that all the content of `filename` is inserted in your code at this place. The difference between `""` and `<>` is just where the compilation chain searches the file to insert. `"path"` denotes a path starting from the current directory. `<path>` denotes a path to a library provided by the operating system.

1 Function Declarations and Implementation

1.1 Function Profiles

The profile of a C function is as follows:

```
type function_name(type1 param1,...,typeN paramN)
```

- *The function name* `function_name` is used to call the function in the code.
- *The output type* `type` refers to the type of the value returned by the function when it has completed its call.
- *A (potentially empty) list of typed parameters* `type1 param1,...,typeN paramN` specifies the **values** (and their type) given as argument to the function when called.

From such a function, you can call this function in your C code as follows:

```
x = function_name(val1,...,valN);
```

where `x` is a variable of type `type`, `val1` denotes a value of type `type1`,... and `valN` denotes a value of type `typeN`.

The compiler uses function definition (providing profile) to figure out if a function call respects the arguments' type and the return type. Thus any function call in your code requires that the profile has been provided in some way, before calling a function.

Also, when defining a function, the code executed by this function does not need to be given immediately: it can be given later in the code (but it must be given at some point, otherwise the compilation fails). So, there are two types of function definitions: function declaration and function implementation.

1.2 Function Declaration *(also called "Prototype")*

A function declaration gives function profile only, i.e., its name, return type and list of arguments, but not its internal code. The precise syntax of function declaration is:

```
type function_name(type1 param1,...,typeN paramN);
```

with a ";" at the end. This also declares that an implementation of the function exists somewhere else, which must be true at final compilation time (detailed later).

Such declarations can be introduced in two ways. They can be directly written in the file or imported via an include directive:

yourcode.c
```
t0 foo(t1 p1,...,tN pN);
...code ...
...foo(v1,...,vN) ...
```

or

yourcode.c
```
#include<lib.h>          (or "lib.h")
...code ...
...foo(v1,...,vN) ...
```

lib.h
```
...
t0 foo(t1 p1,...,tN pN);
...
```

Usually, header files of libraries ('`xxx.h`' files) contain the declarations of the functions provided by the libraries, but not their implementations.

1.3 Function Implementations and System Libraries

Environment/system provided libraries are of course provided with implementations, and generally, manual pages describe the behaviours of these implementations.

For your user-defined functions, you must provide these implementations. An implemented function starts with the function prototype followed by the code of this function:

```
type function_name(type1 param1,...,typeN paramN) {
   ... body ...
}
```

where body is the code that implements the function, i.e., the algorithm associated to the function.

A function declaration is optional, while a function implementation is mandatory. Once a function has been declared or implemented in the code, it can be called afterwards by any other function.

The following example summarizes the use of both function declaration and implementation.

Let `systemlib.h` be a library that declares a function `type libfoo(...);`

yourcode.c:

```
1. #include <systemlib.h>
2. type myfoo(...); // declaration
3.  ... myfoo(...) ... // call of myfoo possible after declaration
4. ... libfoo(...) ... // call of libfoo possible after systemlib.h inclusion
5. type myfoo(...) {  // implementation of myfoo
   ... body ...
   }
6. ... myfoo(...) ... // call of myfoo possible without declaration in line 2
7. int main(...){... myfoo(...) ... libfoo(...) ... }
```

The executable binary `prog` produced by `gcc -o prog yourcode.c` automatically contains:

- The binary implementation of your code, obtained by compiling your source code in `yourcode.c`

- The binary implementation of the environment/system libraries you use in your code if compiled statically ; otherwise, it contains a reference to the library containing the implementation. These shared library have a `.so` extension in UNIX and a `.dll` extension in Windows.

With this approach, the system must know where is located the binary code of these libraries. This information can be customized by using environment variables or specific options on the `gcc` command line, which is out of the scope of this document. Generally, the default configuration of your environment/system knows where system libraries have been installed. This means that when you want to use a system library function, what you generally have to do is to read the man page of this function to know about the libraries to be included and the potential additional compilation options that must be used (e.g., `-lrt`).

System libraries are not installed as C source code. Indeed, it would be costly to recompile this code each time you invoke it in a `gcc` call, as this code does not change between these calls. They are provided as object code (among others, files `xxx.o`) which is a kind of unfinished (not yet executable) binary. It is compiled code (binary, machine code) where some information is still missing such as for example, the code of imported libraries.

You can also produce your own libraries to better structure large projects, as explained hereafter.

2 Developing Your Own Libraries

This section provides the basics for developing your own libraries.

2.1 Developping and Compiling

Suppose you develop a library `mylib`. Similarly to system libraries, a library must come with:

- a header file `mylib.h` which contains the declaration of the functions provided by your library. You generally have to manually write this file as it is part of the C source code.
- an object file `mylib.o` which contains the binary code that implements these functions. This `.o` file is produced from a `.c` source file by a C compiler. This C source files must give an implementation to all functions declared in the `.h` file(s) of the library.

Here is an example on how to write such `.h` and `.c` files:

mylib.h

```
...
int f1(...);
int f2(...);
...
```

and

mylib.c

```
#include "anotherlib.h"
...
int g(...){ g's body }
int f1(...){ f1's body }
int f2(...){ f2's body }
static int h(...){ h's body }
static int intvar;
...
```

Note that `mylib.c` may also contain internal features that are not exposed in `mylib.h`, such as the function `g` in our example. Also, even if these features are "hidden" to the user of the library, they are nonetheless present in the code: this can lead to name clashes if you were to define other functions elsewhere in your code with similar names. To make a feature (function, variable) local, you can use the `static` modifier. In the above example, the function `h` and the variable `intvar` are local and can't be accessed outside `mylib.c`.

Then, to produce `mylib.o` from `mylib.c`, you simply use the `-c` option of the `gcc` compiler:

`gcc -c -o mylib.o mylib.c` produces `mylib.o`.

The `-o mylib.o` option can be omited as `xxx.o` is the default name when compiling `xxx.c` with `gcc -c`. The `-o name.o` option makes it possible to choose another name. Among others, `mylib.o` contains:

- the binary code of `g`, `f1`, `f2`,...
- information about the other libraries on which it depends. For example, if the body of `g` contains a call to a function `h` that is provided by the library `anotherlib`, then the binary code of `h` is not in `mylib.o`. It is elsewhere, for example in a file `anotherlib.o` also produced following the process we are describing. Thus `h` belongs to the imports of `mylib.o`.
- dually, information about its exports, so that later, the compiler (in the linking step) will be able to combine several `.o` object files together by connecting their expected imports to the relevant offered exports of their neighbours. In our example, `g`, `f1` and `f2` are exported by `mylib.o`. Note that `g` is exported too, although it is not declared in `mylib.h`. Indeed, while producing the code of `mylib.o` with the command line above, `gcc` doesn't know anything about `mylib.h` (not provided on the command line) and thus cannot distinguish the functions declared in `mylib.h` from the ones that are only declared in `mylib.c`.

Following this process, you can develop your own libraries, with dependencies to your other libraries, as illustrated on the example where `mylib.c` includes `anotherlib.h`.

Note that the example uses `"anotherlib.h"` and not `<anotherlib.h>`. The `<path>` notation is dedicated to environment/system libraries. They are searched in default directories that depend on your system configuration. If `anotherlib` is some library you have developed and its associated files (`.h .o`) are in some of your directories, the compiler won't find them in the system's default directories. For such libraries, we use the `"path"` notation, where `path` is the path to the library header (for example `anotherlib.h`) starting from the current directory.

Generally, headers (`.h`) are stored in `include` subdirectory, C files (`.c`) in `src`, and object files(`.o` in `lib`.

2.2 Linking

To build an executable file referencing libraries, the C compiler uses a *linker*. This step combines components, connecting exported features to imported ones, and thus, resolving the dependencies between the components of the finally produced (compound) program. For experts, there is a dedicated program in the `gcc` toolchain, `ld`, which does this specific work. But most users generally don't use `ld` directly. They use the `gcc` command and `gcc` is in charge to call `ld`. However `ld` can appear in error messages returned when compiling with `gcc` fails. This happens in particular when the implementation of an expected import is not found in the exports of the provided libraries.

A C file that is meant to produce a library does not contain a `main` function while a C file used to produce an executable file has a `main` function. The prototype of `main` is:

```
int main(int argc, char *argv[]).
```

This function is the entry point of any C executable file, that is it is the first to be called when the program starts.

Usually, an executable file of a program is build from:

- a main file that has a `main` function. Let us call this file `main.c`. From `main.c`, you may have generated a `main.o` with the command `gcc -c main.c` but it is not required.
- a list of user-defined libraries `ulib1.o`,...,`ulibN.o` obtained, for example, by following the process described in previous section.
- a list of references to system provided libraries.

Such a an executable file can be built only if all dependencies are resolved, i.e., if all the functions defined in includes have an implementation in one of the two library lists.

When using gcc in the final step, i.e. to link the components and build the final executable,

- system libraries are generally found automatically, thus they do not have to be provided on the command line. However, note that some of these libraries require other options in the command line. Thus carefully read man pages... information are at the top of them.
- user libraries have to be provided on the command line

Then, the command line looks like (either `main.c` or `main.o` can be used):

```
gcc -o prog ulib1.o ...ulibN.o main.c
```

or

```
gcc -o prog ulib1.o ...ulibN.o main.o
```

This produce the final executable `prog`. Also, it is recommanded to use the `-Wall` option of `gcc` to get all compilation warnings. These warnings may help you identify bugs.

2.3 Beyond Functions

When compiling C programs, the first step is called *prepocessing*. Preprocessing performs textual transformations of the code before calling the compilation itself. This preprocessing identifies *directives* that are present in the code in the form of `#xxx ...` keywords (followed by parameters), where `xxx` is the name of the directive. These directives are put at the beginning of lines. The sequel illustrates some of them.

Previous sections focussed on functions (declaration, implementation), but other features may be put in a header file.

• Macros: yes

For example, macros "`#define CONSTNAME macro_definition`" can be used to define constants. Then, if the header is included in some file, all the occurrences of `CONSTNAME` in this file will be replaced by `macro_definition` before compiling (preprocessing, textual replacement). Macros may also have parameters, as for example "`#define MAX(i,j) (i>j ? i : j)`". As macro semantics is just text replacement, it is not recommanded to define big macros if they are widely used. Indeed, it could duplicates large text, which can usually be avoided by using functions.

There are many ways to efficiently use macros. In the scope of this document, we present on of them: definiting constant values. Thus, defining constants with macros in headers is a simple way to share constant values between files (that include this header) in a consistent way, and to be able to modify their values at a single place. For instance:

```
#define MAX_NUMBER_OF_CONNECTIONS 10
```

or

```
#define ERROR_MESSAGE "Please enter another value, previous value is invalid"
```

• Types: yes

Type declarations and definitions are welcome to centralize the declaration of a shared data type in a multicomponent application. Libraries are commonly used to define some data types and exporting the functions that allow to handle these types. For example a header may contain `struct {...} stud; typedef struct stud student;` which defines a type `student` for representing student data (name, birth,...) and manipulate this type without having to repeatedly write the `struct` keyword.

• Implementation Code: no

Except in unusual cases and if you exactly know what you are doing, it is not a good pratice to put implementation code in your headers. Indeed, if you were to put `int f(...){ body }` in a file `ulib.h` and then include `ulib.h` in two files `file1.c` and `file2.c`, as "include" corresponds to

a text insertion, the body code of `f` will be duplicated in `file1.c` and `file2.c`. Then, once you have compiled `file1.c` and `file2.c`, you have two identical binary implementations of `f` in your libraries with the same name, one in each object file (`.o`), which is probably not what you want. Moreover, this may cause name clashes when building an executable using these two libraries.

• **Variables: subtle**
Variables also belong to implementation (they exist in code and take place in memory) and they have the same problem as functions. Indeed, when the compiler finds a simple variable declaration `int x;` in code , it allocates a slot for `x` in memory providing `x` with an address and the required memory space to store its value. In some ways, a simple variable declaration is also a variable implementation as it is the place where the concrete representation of the variable is elaborated while compiling. Then, if `int x;` is in a file `ulib.h` and `file1.c` and `file2.c` include `ulib.h`, you will have two different variables with the same type and names in the two files. While compiling `file1.c` and `file2.c`, address and memory will be allocated twice (once in each file) and thus, the variable is duplicated, not shared. Once again, this is probably not what you want.

However, you may want to share variables between files, i.e., for example have a variable `x` that both `file1.c` and `file2.c` can read and modify. Declaring `x` in `ulib.h` and then including `ulib.h` in `file1.c` and `file2.c` is the good way to do this. But for this, we need a declaration that is not an implementation. This is the purpose of the `extern` modifier in C.

A declaration line `extern int x;` only postulate that `x` exists somewhere without implementing it, i.e., without giving it an address. As for functions, an implementation must actually be provided somewhere at compile time. This implementation is simply a simple variable declaration `int x;` in one of the source files (`.c`). Here is an example:

ulib.h
```
extern int shared_int;
```

file1.c
```
#include "ulib.h"
... shared_int = 5;...
... x = shared_int;...
```

file2.c
```
#include "ulib.h"
...  int shared_int;...
... shared_int = 8;...
... z = shared_int;...
```

`ulib.h` declares `shared_int` which can then be accessed by `file1.c`, as `file1.c` includes `ulib.h`. In the same way, `file2.c` can access to `shared_int`. The unique implementation of `shared_int` is provided within `file2.c`.

Note that obviously, if your library only defines macros and types (thus nothing that requires an implementation), you do not have to provide (and compile) a `.c` file associated to your header file: your library is there simply a `.h` file

2.4 Avoiding Header Duplication

Using the coding approach described in this document should help you to better structure your software. Yet, there is a common pitfall you will rapidly encounter in middle-size project with multiple libraries. Avoiding this pitfall is the purpose of this section.

Not only source files (`.c`) but also headers (`.h`) can use the include directive. For example, if a library `lib1.h` exports a function `type2 f(...);` and `type2` is a type defined in another library `lib2.h`, then `lib1.h` must include `lib2.h` to be able to use `type2`. Thus, the following situation can occur:

ulib.h
```
...
```

ulib1.h
```
#include "ulib.h"
...
```

ulib2.h
```
#include "ulib.h"
...
```

code.c
```
#include "ulib1.h"
#include "ulib2.h"
...
```

Then, as "include" is just text insertion, `ulib.h` appears two times in `code.c`: it is inserted both through `ulib1.h` and `ulib2.h`. This generally lead to errors at compilation time or bugs while executing.

To avoid this problem, when writing a library header, it is a good pratice to embed the content of header files in a "protecting" `#ifndef` directive which prevents the header content from being inserted multiple times.

ulib.h
```
... original content ...
```

$\Longrightarrow$

improved ulib.h
```
#ifndef ULIB_H
#define ULIB_H
... original content ...
#endif
```

The text between `#ifndef ULIB_H` and `#endif` is only taken into account if the macro `ULIB_H` is not yet defined. Moreover, this text defines the macro `ULIB_H`. Thus, after it has been included once, the macro is defined and further inclusions are removed by the preprocessing. In the previous example:

code.c
```
#include "ulib1.h"
#include "ulib2.h"
```

$\Rightarrow$

code.c — Preprocessing:
```
... // include through ulib1.h
#ifndef ULIB_H                          true
#define ULIB_H              kept, defines ULIB_H
... original content ...                kept
#endif
... // include through ulib2.h
#ifndef ULIB_H                          false
#define ULIB_H                          removed
... original content ...                removed
#endif
```

Notice that the name `ULIB_H` of the macro can be any name of your choice. Yet, it is usual to select a name that is closed to the library name. The single requirement is that this name does not clash with the name of other macros used elsewhere in the same project.

3 Automatic Build with `make` and `Makefile`

When developing a project, it is frequent to have to recompile the source code multiple times. Also, when dealing with librairies, the source code including librairies must also be recompiled

Of course you could write a bash script that contains the precise sequence of all the compilation commands to compile the whole project. But it would also recompile files that do not need to be recompiled because none of the source they depend on has changed. You could write a script that could handle these situations, but your script would be very complex both to write and then to handle when adding new source code.

Thus, it is preferable to use dedicated tools to automate compilation of multiple files in an optimized way. `make` is such a tool.

More generally, `make` is a tool to automate the production of files from source files, but one of its interesting usage is to execute compilation commands in a smart way. Basically, when called from a directory, it looks for a `Makefile` file in this directory. If there is no `Makefile`, then `make` fails and returns an error.

The format for `Makefile` files is very rich, this document only gives the basics on how to write a `Makefile` and use `make`.

In a `Makefile`, the files to produce are identified to "targets". A `Makefile` contains a list of targets together with the instructions on how and when to produce these targets. For instance, when `make tgt` is called, `make` searches for target `tgt` in the list of targets, and follows the instructions associated with `tgt`. When `make` is called without targets, `make` considers the first target of the list. When the argument target (e.g., `tgt`) is not found, `make` fails.

A `Makefile` is a text file that essentially contains a list of target sections. A target section is something like

```
target_name:  target dependencies
    List of commands to produce target_name
```

- in our simple case, `target_name` is the name of the file to produce.
- the `List of commands to produce target_name` implements the building of the file. IMPORTANT: indentation at beginning of lines must not use spaces but tabulations: tabulations are part of `Makefile` syntax.
- `target dependencies` is a list of files that decide if the commands must be executed or not when `make target_name` is called. Of course, if `target_name` does not exist, the file has to be produced and then, the commands are executed. But otherwise commands are only executed if the current version of the file is not up-to-date. To determine if the target is up-to-date, `make` relies on the `target dependency list` which should contain the files on which `target_name` depends. As soon as one of this dependency file has changed (its modification time is more recent than the one of `target_name`), the list of commands must be executed to rebuild an up-to-date version of `target_name`.

Note that if target `target_name` depends on file `lib.o`, it is not sufficient to compare the time information of `target_name` and `lib.o`, as `lib.o` may itself be obsolete. Thus `make` searches for target `lib.o` and if the target exists, recursively, `make` decides if the commands to rebuild `lib.o` must be executed or not by looking at its dependencies. In short, make does the job of maintaining up-to-date version of the handled files, provided that you give it the relevant dependencies in the `target dependencies` sections. Then, calling `make` on a target automatically updates (if necessary) the target itself, and all the other targets it depends on.

The following example illustrates some of the most usual other features of `make`...
Let `main.c` be the source code of an executable that uses a library `lib1`, which in turn uses a library `lib2`. In short:

- main.c: `#include "lib1.h" ...  int main(...){...}`
- lib1.c: `#include "lib2.h" ...`
- lib2.c: `only includes standard system libraries`

a `Makefile` to build the final executable from the sources could be:

```
.PHONY: clean

all: prog

lib1.o: lib1.c lib2.h
    gcc -c -o lib1.o lib1.c

lib2.o: lib2.c
    gcc -c -o lib2.o lib2.c

prog: lib1.o lib2.o main.c
    gcc -o prog lib1.o lib2.o main.c

clean:
    rm -f lib1.o lib2.o prog
```

The three targets `lib1.o`, `lib2.o` and `prog` exactly illustrate what have been explained above. For example, `lib1.o` must be recompiled from `lib1.c` as soon as `lib1.c` has been modified. It also depends on `lib2.h` as modifying `lib2.h` modifies `lib1.c` through inclusion.

The target `all` does not have any associated command. Thus it does not produce any files. But its dependencies are checked and updated if necessary. Thus the command line `make all` causes the update of all the files that are enumerated in the dependency list of the target `all`. It is the usual way to tell to `make` what software (targets) must be build when building the whole project. As this target is the first one in the file (except the `.PHONY` special one), it is used by default when calling `make` without parameter. Making `make` equiavalent to `make all` to build the whole project is a common pratice.

The target `clean` is another conventional one. As you can see it is (usually) used to clean up the project directory by removing all the files that have been added during the different invocations of the building processes (through this `Makefile`). As the `clean` target is not intended to produce files based on timing dependencies, its list of dependencies is empty.

Now, just assume if there were a file named `clean` in the directory of the Makefile: the `make clean` would not execute since `clean` has no dependencies. To avoid this situation, the `.PHONY:` at the beginning of the file gives a list of targets to be executed even if there exists a file with the name of the target.

There are many more features in `Makefile`s, e.g., to make them more concise, more powerful, more structured... such as using variables and/or wildcards, or recursively calling `make` in subdirectories,... You can find lot of documentation about this on the web, and ask us any question you could have!