Chapter 37: Compilation

The C language is traditionally a compiled language (as opposed to interpreted). The C Standard defines ***translation phases***, and the product of applying them is a program image (or compiled program). In c11, the phases are listed in §5.1.1.2.

Section 37.1: The Compiler

After the C pre-processor has included all the header files and expanded all macros, the compiler can compile the program. It does this by turning the C source code into an object code file, which is a file ending in `.o` which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were `#include`d into the file (this is not the same as including the declarations, which is what `#include` does). This is the job of the linker.

In general, the exact sequence how to invoke a C compiler depends much on the system that you are using. Here we are using the GCC compiler, though it should be noted that many more compilers exist:

```
% gcc -Wall -c foo.c
```

`%` is the OS' command prompt. This tells the compiler to run the pre-processor on the file `foo.c` and then compile it into the object code file `foo.o`. The `-c` option means to compile the source code file into an object file but not to invoke the linker. This option `-c` is available on POSIX systems, such as Linux or macOS; other systems may use different syntax.

If your entire program is in one source code file, you can instead do this:

```
% gcc -Wall foo.c -o foo
```

This tells the compiler to run the pre-processor on `foo.c`, compile it and then link it to create an executable called `foo`. The `-o` option states that the next word on the line is the name of the binary executable file (program). If you don't specify the `-o`, (if you just type `gcc foo.c`), the executable will be named `a.out` for historical reasons.

In general the compiler takes four steps when converting a `.c` file into an executable:

1. **pre-processing** - textually expands `#include` directives and `#define` macros in your `.c` file
2. **compilation** - converts the program into assembly (you can stop the compiler at this step by adding the `-S` option)
3. **assembly** - converts the assembly into machine code
4. **linkage** - links the object code to external libraries to create an executable

Note also that the name of the compiler we are using is GCC, which stands for both "GNU C compiler" and "GNU compiler collection", depending on context. Other C compilers exist. For Unix-like operating systems, many of them have the name `cc`, for "C compiler", which is often a symbolic link to some other compiler. On Linux systems, `cc` is often an alias for GCC. On macOS or OS-X, it points to clang.

The POSIX standards currently mandates `c99` as the name of a C compiler — it supports the C99 standard by default. Earlier versions of POSIX mandated `c89` as the compiler. POSIX also mandates that this compiler understands the options `-c` and `-o` that we used above.

Note: The `-Wall` option present in both `gcc` examples tells the compiler to print warnings about questionable constructions, which is strongly recommended. It is a also good idea to add other warning options, e.g. `-Wextra`.

Section 37.2: File Types

Compiling C programs requires you to work with five kinds of files:

1. **Source files**: These files contain function definitions, and have names which end in `.c` by convention. Note: `.cc` and `.cpp` are C++ files; *not* C files.
   e.g., `foo.c`
2. **Header files**: These files contain function prototypes and various pre-processor statements (see below). They are used to allow source code files to access externally-defined functions. Header files end in `.h` by convention.
   e.g., `foo.h`
3. **Object files**: These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in `.o` by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in `.obj`.
   e.g., `foo.o` `foo.obj`
4. **Binary executables**: These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in `.exe` on Windows.
   e.g., `foo` `foo.exe`
5. **Libraries**: A library is a compiled binary but is not in itself an an executable (i.e., there is no `main()` function in a library). A library contains functions that may be used by more than one program. A library should ship with header files which contain prototypes for all functions in the library; these header files should be referenced (e.g; `#include <library.h>`) in any source file that uses the library. The linker then needs to be referred to the library so the program can successfully compiled. There are two types of libraries: static and dynamic.

   - **Static library**: A static library (`.a` files for POSIX systems and `.lib` files for Windows — not to be confused with DLL import library files, which also use the `.lib` extension) is statically built into the program . Static libraries have the advantage that the program knows exactly which version of a library is used. On the other hand, the sizes of executables are bigger as all used library functions are included.
     e.g., `libfoo.a` `foo.lib`
   - **Dynamic library**: A dynamic library (`.so` files for most POSIX systems, `.dylib` for OSX and `.dll` files for Windows) is dynamically linked at runtime by the program. These are also sometimes referred to as shared libraries because one library image can be shared by many programs. Dynamic libraries have the advantage of taking up less disk space if more than one application is using the library. Also, they allow library updates (bug fixes) without having to rebuild executables.
     e.g., `foo.so` `foo.dylib` `foo.dll`

Section 37.3: The Linker

The job of the linker is to link together a bunch of object files (`.o` files) into a binary executable. The process of *linking* mainly involves *resolving symbolic addresses to numerical addresses*. The result of the link process is normally an executable program.

During the link process, the linker will pick up all the object modules specified on the command line, add some system-specific *startup code* in front and try to resolve all *external* references in the object module with *external definitions* in other object files (object files can be specified directly on the command line or may implicitly be added through libraries). It will then assign *load addresses* for the object files, that is, it specifies where the code and data

will end up in the address space of the finished program. Once it's got the load addresses, it can replace all the symbolic addresses in the object code with "real", numerical addresses in the target's address space. The program is ready to be executed now.

This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into library files. These files have names which end in `.a` or `.so`, and you normally don't need to know about them, as the linker knows where most of them are located and will link them in automatically as needed.

Implicit invocation of the linker

Like the pre-processor, the linker is a separate program, often called `ld` (but Linux uses `collect2`, for example). Also like the pre-processor, the linker is invoked automatically for you when you use the compiler. Thus, the normal way of using the linker is as follows:

```
% gcc foo.o bar.o baz.o -o myprog
```

This line tells the compiler to link together three object files (`foo.o`, `bar.o`, and `baz.o`) into a binary executable file named `myprog`. Now you have a file called `myprog` that you can run and which will hopefully do something cool and/or useful.

Explicit invocation of the linker

It is possible to invoke the linker directly, but this is seldom advisable, and is typically very platform-specific. That is, options that work on Linux won't necessarily work on Solaris, AIX, macOS, Windows, and similarly for any other platform. If you work with GCC, you can use `gcc -v` to see what is executed on your behalf.

Options for the linker

The linker also takes some arguments to modify it's behavior. The following command would tell gcc to link `foo.o` and `bar.o`, but also include the `ncurses` library.

```
% gcc foo.o bar.o -o foo -lncurses
```

This is actually (more or less) equivalent to

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(although `libncurses.so` could be `libncurses.a`, which is just an archive created with `ar`). Note that you should list the libraries (either by pathname or via `-lname` options) after the object files. With static libraries, the order that they are specified matters; often, with shared libraries, the order doesn't matter.

Note that on many systems, if you are using mathematical functions (from **`<math.h>`**), you need to specify `-lm` to load the mathematics library — but Mac OS X and macOS Sierra do not require this. There are other libraries that are separate libraries on Linux and other Unix systems, but not on macOS — POSIX threads, and POSIX realtime, and networking libraries are examples. Consequently, the linking process varies between platforms.

Other compilation options

This is all you need to know to begin compiling your own C programs. Generally, we also recommend that you use the `-Wall` command-line option:

```
% gcc -Wall -c foo.cc
```

The `-Wall` option causes the compiler to warn you about legal but dubious code constructs, and will help you catch a lot of bugs very early.

If you want the compiler to throw more warnings at you (including variables that are declared but not used, forgetting to return a value etc.), you can use this set of options, as `-Wall`, despite the name, doesn't turn *all of the possible warnings* on:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow ...
```

Note that `clang` has an option `-Weverything` which really does turn on all warnings in `clang`.

Section 37.4: The Preprocessor

Before the C compiler starts compiling a source code file, the file is processed in a preprocessing phase. This phase can be done by a separate program or be completely integrated in one executable. In any case, it is invoked automatically by the compiler before compilation proper begins. The preprocessing phase converts your source code into another source code or translation unit by applying textual replacements. You can think of it as a "modified" or "expanded" source code. That expanded source may exist as a real file in the file system, or it may only be stored in memory for a short time before being processed further.

Preprocessor commands start with the pound sign ("#"). There are several preprocessor commands; two of the most important are:

1. **Defines**:

   `#define` is mainly used to define constants. For instance,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   becomes

   ```
   int a = 1000000;
   ```

   `#define` is used in this way so as to avoid having to explicitly write out some constant value in many different places in a source code file. This is important in case you need to change the constant value later on; it's much less bug-prone to change it once, in the `#define`, than to have to change it in multiple places scattered all over the code.

   Because `#define` just does advanced search and replace, you can also declare macros. For instance:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   becomes:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

At first approximation, this effect is roughly the same as with inline functions, but the preprocessor doesn't provide type checking for `#define` macros. This is well known to be error-prone and their use necessitates great caution.

Also note here, that the preprocessor would also replace comments with a blanks as explained below.

2. **Includes**:

   `#include` is used to access function definitions defined outside of a source code file. For instance:

   ```
   #include <stdio.h>
   ```

   causes the preprocessor to paste the contents of **`<stdio.h>`** into the source code file at the location of the `#include` statement before it gets compiled. `#include` is almost always used to include header files, which are files which mainly contain function declarations and `#define` statements. In this case, we use `#include` in order to be able to use functions such as `printf` and `scanf`, whose declarations are located in the file `stdio.h`. C compilers do not allow you to use a function unless it has previously been declared or defined in that file; `#include` statements are thus the way to re-use previously-written code in your C programs.

3. **Logic operations**:

   ```
   #if defined A || defined B
   variable = another_variable + 1;
   #else
   variable = another_variable * 2;
   #endif
   ```

   will be changed to:

   ```
   variable = another_variable + 1;
   ```

   if A or B were defined somewhere in the project before. If this is not the case, of course the preprocessor will do this:

   ```
   variable = another_variable * 2;
   ```

   This is often used for code, that runs on different systems or compiles on different compilers. Since there are global defines, that are compiler/system specific you can test on those defines and always let the compiler just use the code he will compile for sure.

4. **Comments**

   The Preprocessor replaces all comments in the source file by single spaces. Comments are indicated by `//` up to the end of the line, or a combination of opening `/*` and closing `*/` comment brackets.

Section 37.5: The Translation Phases

As of the C 2011 Standard, listed in §*5.1.1.2 Translation Phases*, the translation of source code to program image (e.g., the executable) are listed to occur in 8 ordered steps.

1. The source file input is mapped to the source character set (if necessary). Trigraphs are replaced in this step.
2. Continuation lines (lines that end with `\`) are spliced with the next line.
3. The source code is parsed into whitespace and preprocessing tokens.
4. The preprocessor is applied, which executes directives, expands macros, and applies pragmas. Each source file pulled in by `#include` undergoes translation phases 1 through 4 (recursively if necessary). All preprocessor related directives are then deleted.
5. Source character set values in character constants and string literals are mapped to the execution character set.
6. String literals adjacent to each other are concatenated.
7. The source code is parsed into tokens, which comprise the translation unit.
8. External references are resolved, and the program image is formed.

An implementation of a C compiler may combine several steps together, but the resulting image must still behave as if the above steps had occurred separately in the order listed above.