Chapter 29

Compilers and libraries

29.1 What is executed when you run a program?

Purpose. In this section you will be introduced to the different types of files that you encounter while programming.

29.1.1 Source versus program

There are two types of programming languages:

1. In an *interpreted language* you write human-readable source code and you execute it directly: the computer translates your source line by line as it encounters it.
2. In a *compiled language* your code whole source is first compiled to a program, which you then execute.

Examples of interpreted languages are *Python*, *Matlab*, *Basic*, *Lisp*. Interpreted languages have some advantages: often you can write them in an interactive environment that allows you to test code very quickly. They also allow you to construct code dynamically, during runtime. However, all this flexibility comes at a price: if a source line is executed twice, it is translated twice. In the context of this book, then, we will focus on compiled languages, using *C* and *Fortran* as prototypical examples.

So now you have a distinction between the readable source code, and the unreadable, but executable, program code. In this tutorial you will learn about the translation process from the one to the other. The program doing this translation is known as a *compiler*. This tutorial will be a 'user manual' for compilers, as it were; what goes on inside a compiler is a different branch of computer science.

29.1.2 Binary files

Programs can get very large, so you don't want to recompile everything for every change you make: typically you divide your source in multiple files, and compile them separately.

First of all, a source file can be compiled to an *object file*, which is a bit like a piece of an executable: by itself it does nothing, but it can be combined with other object files to form an executable.

If you have a collection of object files that you need for more than one program, it is usually a good idea to make a *library*: a bundle of object files that can be used to form an executable. Often, libraries are written by an expert and contain code for specialized purposes such as linear algebra manipulations. Libraries are important enough that they can be commercial, to be bought if you need expert code for a certain purpose.

You will now learn how these types of files are created and used.

29.2 Simple compilation

Purpose. In this section you will learn about executables and object files.

29.2.1 Compilers

Your main tool for turning source into a program is the *compiler*. Compilers are specifically to a language: you use a different compiler for C than for Fortran. You can also have two compilers for the same language, but from different 'vendors'. For instance, while many people use the open source *gcc* or *clang* compiler families, companies like *Intel* and *IBM* offer compilers that may give more efficient code on their processors.

29.2.2 Compile a single file

Let's start with a simple program that has the whole source in one file.

One file: `hello.c` Compile this program with your favourite compiler; we will use `gcc` in this tutorial, but substitute your own as desired. As a result of the compilation, a file `a.out` is created, which is the executable.

```
%% gcc hello.c
%% ./a.out
hello world
```

You can get a more sensible program name with the `-o` option:

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

29.2.3 Multiple files: compile and link

Now we move on to a program that is in more than one source file.

Main program: `fooprog.c`

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

extern void bar(char*);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `fooprog.c`

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

void bar(char *s) {
  printf("%s",s);
  return;
}
```

As before, you can make the program with one command.

Output
[compile/c] makeoneprogram:

```
clang -o oneprogram fooprog.c foosub.c
./oneprogram
hello world
```

However, you can also do it in steps, compiling each file separately and then linking them together.

Output
[compile/c] makeseparatecompile:

```
clang -o oneprogram fooprog.o foosub.o
./oneprogram
hello world
```

The `-c` option tells the compiler to compile the source file, giving an *object file*. The third command than acts as the *linker*, tieing together the object files into an executable. (With programs that are spread over several files there is always the danger of editing a subroutine definition and then forgetting to update all the places it is used. See the 'make' tutorial, section 30, for a way of dealing with this.)

Each object file defines some routine names, and uses some others that are undefined in it, but that will be defined in other object files or system libraries. Use the *nm* command to display this:

```
[c:264] nm foosub.o
0000000000000000 T _bar
                 U _printf
```

Lines with `T` indicate routines that are defined; lines with `U` indicate routines that are used but not define in this file. In this case, `printf` is a system routine that will be supplied in the linker stage.

29.3 Libraries

Purpose. In this section you will learn about libraries.

If you have written some subprograms, and you want to share them with other people (perhaps by selling them), then handing over individual object files is inconvenient. Instead, the solution is to combine them into a library. First we look at *static libraries*, for which the *archive utility* `ar` is used. A static library is linked into your executable, becoming part of it. This may lead to large executables; you will learn about shared libraries next, which do not suffer from this problem.

Create a directory to contain your library (depending on what your library is for this can be a system directory such as `/usr/bin`), and create the library file there.

Output
[compile/c] makestaticlib:

```
Making static library

for o in foosub.o ; do \
  ar cr libs/libfoo.a ${o} ; \
done
```

The *nm* command tells you what's in the library, just like it did with object files, but now it also tells you what object files are in the library:

```
%% nm ../lib/libfoo.a

../lib/libfoo.a(foosub.o):
00000000 T _bar
         U _printf
```

The library can be linked into your executable by explicitly giving its name, or by specifying a library path:

Output
[compile/c] staticprogram:

```
for o in foosub.o ; do \
  ar cr libs/libfoo.a ${o} ; \
done

Linking to static library

clang  -o staticprogram fooprog.o -Llibs -lfoo

.. note the size of the program

-rwxr-xr-x  1 eijkhout  _avahi  8464 Sep 16 14:33 staticprogram

.. running:

hello world
```

Although they are somewhat more complicated to use, *shared libraries* have several advantages. For instance, since they are not linked into the executable but only loaded at runtime, they lead to (much) smaller executables. They are not created with `ar`, but through the compiler. For instance:

Output
[compile/c] makedynamiclib:

```
Making shared library

clang -o libs/libfoo.so -shared foosub.o
```

You can again use *nm*:

```
%% nm ../lib/libfoo.so

../lib/libfoo.so(single module):
00000fc4 t __dyld_func_lookup
00000000 t __mh_dylib_header
00000fd2 T _bar
         U _printf
00001000 d dyld__mach_header
00000fb0 t dyld_stub_binding_helper
```

Shared libraries are not actually linked into the executable; instead, the executable needs the information where the library is to be found at execution time. One way to do this is with *LD_LIBRARY_PATH*:

Output
[compile/c] dynamicprogram:

```
Linking to shared library

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo

.. note the size of the program

-rwxr-xr-x  1 eijkhout  _avahi  8432 Sep 16 14:33 dynamicprogram

.. note unresolved link to a library

make[3]: ldd: No such file or directory
make[3]: [run_dynamicprogram] Error 1 (ignored)

.. running by itself:

hello world

.. running with updated library path:

LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs ./dynamicprogram
hello world
```

Another solution is to have the path be included in the executable:

```
%% gcc -o foo fooprog.o -L../lib -Wl,-rpath,`pwd`/../lib -lfoo
%% ./foo
hello world
```

The link line now contains the library path twice:

1. once with the `-L` directive so that the linker can resolve all references, and
2. once with the linker directive `-Wl,-rpath,'pwd'/../lib` which stores the path into the executable so that it can be found at runtime.

Use the command *`ldd`* to get information about what shared libraries your executable uses. (On Mac OS X, use `otool -L` instead.)