Stages of Compilation

Before jumping straight to compiler optimizations, which is what most of this chapter is about, let's briefly recap the "big picture" first. Skipping the boring parts, there are 4 stages of turning C programs into executables:

1. **Preprocessing** expands macros, pulls included source from header files, and strips off comments from source code: `gcc -E source.c` (outputs preprocessed source to stdout)
2. **Compiling** parses the source, checks for syntax errors, converts it into an intermediate representation, performs optimizations, and finally translates it into assembly language: `gcc -S file.c` (emits an `.s` file)
3. **Assembly** turns assembly language into machine code, except that any external function calls like `printf` are substituted with placeholders: `gcc -c file.c` (emits an `.o` file, called *object file*)
4. **Linking** finally resolves the function calls by plugging in their actual addresses, and produces an executable binary: `gcc -o binary file.c`

There are possibilities to improve program performance in each of these stages.

Interprocedural Optimization

We have the last stage, linking, because it is is both easier and faster to compile programs on a file-by-file basis and then link those files together — this way you can do this in parallel and also cache intermediate results.

It also gives the ability to distribute code as *libraries*, which can be either *static* or *shared*:

- *Static* libraries are simply collections of precompiled object files that are merged with other sources by the compiler to produce a single executable, just as it normally would.
- *Dynamic* or *shared* libraries are precompiled executables that have additional meta-information about where their callables are, references to which are resolved during runtime. As the name suggests, this allows *sharing* the compiled binaries between multiple programs.

The main advantage of using static libraries is that you can perform various *interprocedural optimizations* that require more context than just the signatures of library functions, such as function inlining or dead code elimination. To force the linker to look for and only accept static libraries, you can pass the `-static` option.

This process is called *link-time optimization (LTO)*, and it is possible because modern compilers also store some form of *intermediate representation* in object files, which allows them to perform certain lightweight optimizations on the program as a whole. This also allows using different compiled languages in the same program, which can even be optimized across language barriers if their compilers use the same intermediate representation.

LTO is a relatively recent feature (it appeared in GCC only around 2014), and it is still far from perfect. In C and C++, the way to make sure no performance is lost due to separate compilation is to create a *header-only library*. As the name suggests, they are just header files that contain full definitions of all functions, and so by simply including them, the compiler gets access to all optimizations possible. Although you do have to recompile the library code from scratch each time, this approach retains full control and makes sure that no performance is lost.

Inspecting the Output

Examining output from each of these stages can yield useful insights into what's happening in your program.

You can get assembly from the source by passing the `-S` flag to the compiler, which will then generate a human-readable `*.s` file. If you pass `-fverbose-asm`, this file will also contain compiler comments about source code line numbers and some info about variables being used. If it is just a little snippet and you are feeling lazy, you can use Compiler Explorer, which is a very handy online tool that converts source code to assembly, highlights logical asm blocks by color, includes a small x86 instruction set reference, and also has a large selection of other compilers, targets, and languages.

Apart from the assembly, the other most helpful level of abstraction is the intermediate representation on which compilers perform optimizations. The IR defines the flow of computation itself and is much less dependent on architecture features like the number of registers or a particular instruction set. It is often useful to inspect these to get insight into how the compiler *sees* your program, but this is a bit out of the scope of this book.

We will mainly use GCC in this chapter, but also try to duplicate examples for Clang when necessary. The two compilers are largely compatible with each other, for the most part only differing in some optimization flags and minor syntax details.