Tasks

Phil Reinartz

May 30, 2025

1 Task 1 Stages of the compilation

First stage "Preprocessing"
The main function of the preprocessor is, that it adds the "full" c code into your .i data. For example, the stdio.h library is nothing other than some functions that someone declared, so that you don't have to write them yourself. These functions are loaded by the preprocessor in your program, because you wrote `#include <stdio.h>` inside your code. The same is also for macros that you have placed. For example: if you wrote `#define Maximalwert 12345678909876543212345678999999999`, the preprocessor takes the value of Maximalwert and puts it everywhere in the program where you wrote Maximalwert. So all in all, the preprocessor does nothing other, than complete your program by replacing predefined macros with the right value and adding and subtracting content you want to include, or your machine needs to work right. So that your program is ready to be compiled in the next step.

Second stage, the compilation to assembly
The next part of the compilation is the part where the most work gets done. In the first step, your preprocessor did necessary replacements for you, to make your .c script ready to be compiled. Now the compiler has to create a file, that your machine is able to process. The first step of this process is, to create assembly instructions of the preprocessors revised c code. The assembly instructions are a "humanoid" version of the binary, that is in some context also text based, but on a very low level text based. The now created .s file is function wise the equivalent of your .i file written in assembly. For example: the assembly instruction `movl    \$0, \%eax` is the equivalent of setting the return value of your function by zero.

Third stage, creating the object file
The object file is an interesting part of the compilation that is a easy to understand process, but not an easy to understand file. The object file is basically the functional equivalent of your assembly instructions, but written in machine language(not the full binary, but also written with 0s and 1s). The .o file is hard to read, because you have to translate every line of code with the manual of your processor, because the binary language is architecture dependent. If you work with several different processors, this could get very complicated very quickly.

Final stage, linking your .o file.
The final step of your compilation is, that your script needs somehow to be executable. This is done at the linking Stage, which works kind of similar to the preprocessing staged (don't get me wrong, it's definitely not the same or similar thing, but the basic idea is similar). The linker is necessary, to define everything, that is still undefined, which could be for example some kind of machine specific function that needs to be declared. Otherwise, your machine probably wouldn't know what to do at the part where the function is used, because the machine has not the ability to process other things than basic commands. So the linker replaces "more complex" functions with the address to the basic commands that describe the used function. This turns your file into an executable, that has several connections to many other .o files, that do nothing other, than describing functions of your .o file into processor understandable commands.

2 Task 2 Regex Search & Replace in Code

1)

I don't know if it's necessary to explain the cp (copy tool), but it's definitely an easy to use powerful tool to copy data.

The grep program
Grep is a powerful tool, which was made for searching text based data. It's useful for searching specific words, strings or patterns in a file. In the example we did: `grep -En printf\s*\(" solutions/debug_sample.c` grep is the program, that we are using, which is made for finding specific finding regex-based data. -E stands for "extended regular expressions", that means that differences for basic regular expressions are summarized afterwards. The -n stands for "line number", which does prefix each line of output with it's line number from the file. `"printf\s*("` is the regex pattern, for that we are searching. `\s` is nothing other than a empty char " ", the * means that the preceding item will be matched 0 or more times, which defines, that grep would also find the printf if there is a tab behind. Last but not least is the `\(` which does declare the "(" as a character and not as an open expression. The Last part of the command is the place where we are doing the grep:`solutions/debug_sample.c`, that in our case is in the `debug_sample.c` file. Honestly I forgot about grep, but I think it's yet another powerful tool, that we could use from the consol to find certain text patterns, which I guess would help us in the future with the next tasks.

The sed program
Sed (stream editor for filtering and transforming text) is a stream editor, which often is used to perform basic text transformations on an input stream. If you know what you are doing, than sed is way more efficient and faster than a normal editor, by manipulating the file from your terminal. In the example we did: `sed -E -i.bak 's/printf\s*/debug_printf/g' solutions/debug_sample.c`. The first part of the command is the program that we are using, that in our case is sed. Next part is the `-E`, which as in grep stands for the extended regular expressions. The `-i` command is the definitions of editing files in place, which does changes to the file directly. The `.bak` part does declare that there is made a backup with the ending `.bak`. The major part of the command comes now, which is the "replace" command. The comment is done like this : s/regexp/replacement/.In our case it's this part: `'s/printf\s*/debug_printf/g'`, the `\s*` is a empty space and the g say that this process is done to all other strings, that fit the same definition.

The awk program
The awk program is quite easy and does in our case nearly the same thing as the grep command. The awk is often used to search the line of a certain string.

Vim interactive
Now comes my favorite part, I like the vim interactive, because you are always sure what you're doing and you see the changes directly. The search command is very easy, by entering /'yoursearchedstring'. Honestly I didn't know the direct change command from vims command line, but I also don't like it that much. It's build similar to the other commands from this PP.

Vim Cli Working with this is basically the same as working with the vim consol but from the terminal. The vim -c does open the consol of whim, where we just tipped the command from before.

All in all, I think I like the grep command and the interactive vim most. Grep is a useful tool for finding things and vim is easy and save to use. Although the other tools are also very powerful, but you have to know what you're doing, otherwise you could get problems very quickly.

3 Task 3 Modular Linking with extern

On this task we had to define 2 function in separate files (one of them is main) and link them to one executable. The first important step to make this work, is that you have to declare the function that you want to use from another place on your main script. We did this by writing `extern int add(int,int);` this does announce, that we want to use the function for our script, it works similar to the `"#include<...>"` function, but only with one self written function. This process of cutting scripts/programs could speed up builds enormously, exspecially when you work on large projects with several people, or if you want to test out several different functions for the same purpose. If you want to include the function, you just have to declare it in the "main" program, and link the together, and than there is the function included.

I think the main purpose at all of doing manual linking instead of letting gcc do it all at once is that, you have way more flexibility at large projects. I guess for small project doing all at once is faster, but if you have a project with like 100 functions inside, it's faster, if you separate the functions from the main script. The hole project gets also clearer by doing this.