HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2019
COMPILING A C PROGRAM: THE C BUILD PROCESS

1. The build process

The process of transforming a written program into an executable in Linux/Unix takes 4 distinct steps, see Figure 1.1.

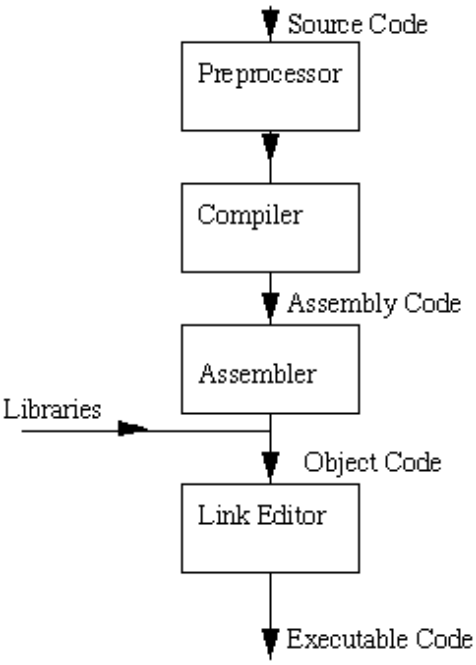

FIGURE 1.1. An overview of the build process. The "Link Editor" that converts object code to executable code is often simply called a "Linker"

1.1. **Preprocessing.** The preprocessor takes your C code as input and generates another C code as output. Preprocessing is used to combine different source files together, introduce code from an external source (e.g. a library), and to customise a code so it suits system or user requirements.

A list of common preprocessor directives:

Date: January 22, 2019.

- `#include` inserts code into a file. Typically, a C program code is separated into two sets of files. Header files (ending in .h) contain function prototypes, and source files (ending in .c) contain the function definitions (more on function prototypes and definitions in Section 1.2). In first.c, the include directive is used to include the function definitions from the standard library for I/O (input/output).
- `#define` defines a macro. Macros can be used to store predefined constants like "`#define PI 3.14`" or short code snippets like "`#define SQR(x) ((x)*(x))`". The preprocessor will then look for PI or SQR(...) and replace those tokens with 3.14 or ((...)*(...)), respectively.
- `#if, #ifdef, #ifndef, #endif` are used to add or remove code depending either on a boolean condition or whether a keyword is defined.

Task 1: Cd into the Task-1 directory, type `gcc -E -o preprocess.i preprocess.c` in the terminal and compare the output file with the .c and .h files.

Useful predefined macros, handy for debug prints or to annotate output:

- `__LINE__` Line of the current file
- `__FILE__` Name of the current file
- `__DATE__` Current date
- `__TIME__` Current time

1.2. **Compilation.** The compiler takes C code as input and generates assembly code as output.

If we want, we can examine the assembly code to see what the program is *actually* doing. For example, this can be useful if we suspect that the compiler has a bug in it. Later in the course, being able to look at the assembly code will be useful when we want to check if and how the compiler has optimized the code.

You can ask gcc to generate assembly code with `gcc -S *.c`, and it will output a .s file for every .c file.

The compiler does the bulk of the work when building a program. The compiler therefore gives you a wealth of options to tweak its behaviour, a few of which are listed below. Read the `man` page or the option summary online at gcc.gnu.org/onlinedocs/gcc/Option-Summary.html for more details.

1.2.1. *Useful basic compiler options, warnings, etc.*

- `-Imyincludedir` Add myincludedir to search path for include files.
- `-Wall` Turn on all warnings. Make this one a habit!
- `-g` Produce debugging information that a debugger (like `gdb`) can use.
- `-O` Optimisation settings.
  - `-O0` Turn off all optimisation (default). Use this when debugging!

- `-O` or `-O1` Turn on optimisations to reduce code size and execution time that don't take much compilation time.
- `-O2` Optimize even more. Perform nearly all optimisations that don't involve a space-speed tradeoff.
- `-O3` Turn on maximum optimisation.
- `-Os` Optimise for size.

1.3. **Assembly.** The assembler takes assembly code as input and generates object files as output. Object files are binary files that can either be linked with start-up code or wrapped into a library. Each source file will typically result in a .o file.

You can tell gcc to stop at the assembly step with the `-c` flag. This is commonly done in projects when you want explicit control over the linking step.

Task 2:

Cd into the Task-2 directory and compile all the .c files with the command `gcc -c *.c`. Also build the `singleFile_calculator` executable.

The `nm` program can parse an object file, library, or executable and report on its contents. This is useful if you need to know what symbols are in a library or if you're having trouble linking object files produced by different compilers. The `nm` output contains one line for each symbol found in the object file.

Use `nm` to examine the contents of the `singleFile_calculator` executable and compare it to the contents of the .o files. You should be able to find the names of symbols and whether they are defined. Undefined symbols cannot be used unless a definition is linked in via a library or object file!

1.4. **Linking.** The linker combines object files and external libraries and generates an executable or library as output. There are two types of libraries: *static* and *dynamic*.

The static library is just an archive of .o files. You use it by linking it into your executable in the same way as your own .o files. The linker only loads the functions of the library that your code references. This moderately increases the size of the executable, but the running process will be small.

The dynamic library is prepared differently. When you link to a dynamic library, the linker just puts references in the executable, which results in a small executable file. When the running executable encounters function that is linked in from a dynamic library, the function loads dynamically (hence the name). The dynamic linker tends to load a large portion of the library, even if only a few functions are used, which increases the memory footprint of the running process.

We are going to create libraries for the add and subtract functions of the calculator program.

Create a static library from the .o files by writing

```
ar crs libstaticfunction.a add.o subtract.o
```

Now generate a new executable file using the static library and the calculator.o file:

```
gcc -o static_calculator calculator.o -L. -lstaticfunction
```

The `-L` flag tells the linker in which directory to look for libraries. The `-l`*name* flag tells the linker to link with static libraries with the name `libname.a` or dynamic libraries with the name `libname.so`.

Create a dynamic library with these commands:

```
gcc -o libdynfunction.so -shared add.o subtract.o
```

And link a corresponding executable:

```
gcc -o dyn_calculator calculator.o -L. -ldynfunction
```

If you get an error that the system cannot find libdynfunction.so when running `dyn_calculator`, you have to update the search path for dynamic libraries:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Tip: Use `ldd dyn_calculator` to ask the executable which dynamic libraries it is linked to.

(note) – The intrepid student can read about performance issues of dynamic libraries in Section 14.11 in *Optimizing software in C++*.

Use `nm` to examine the structure of your executables and libraries.