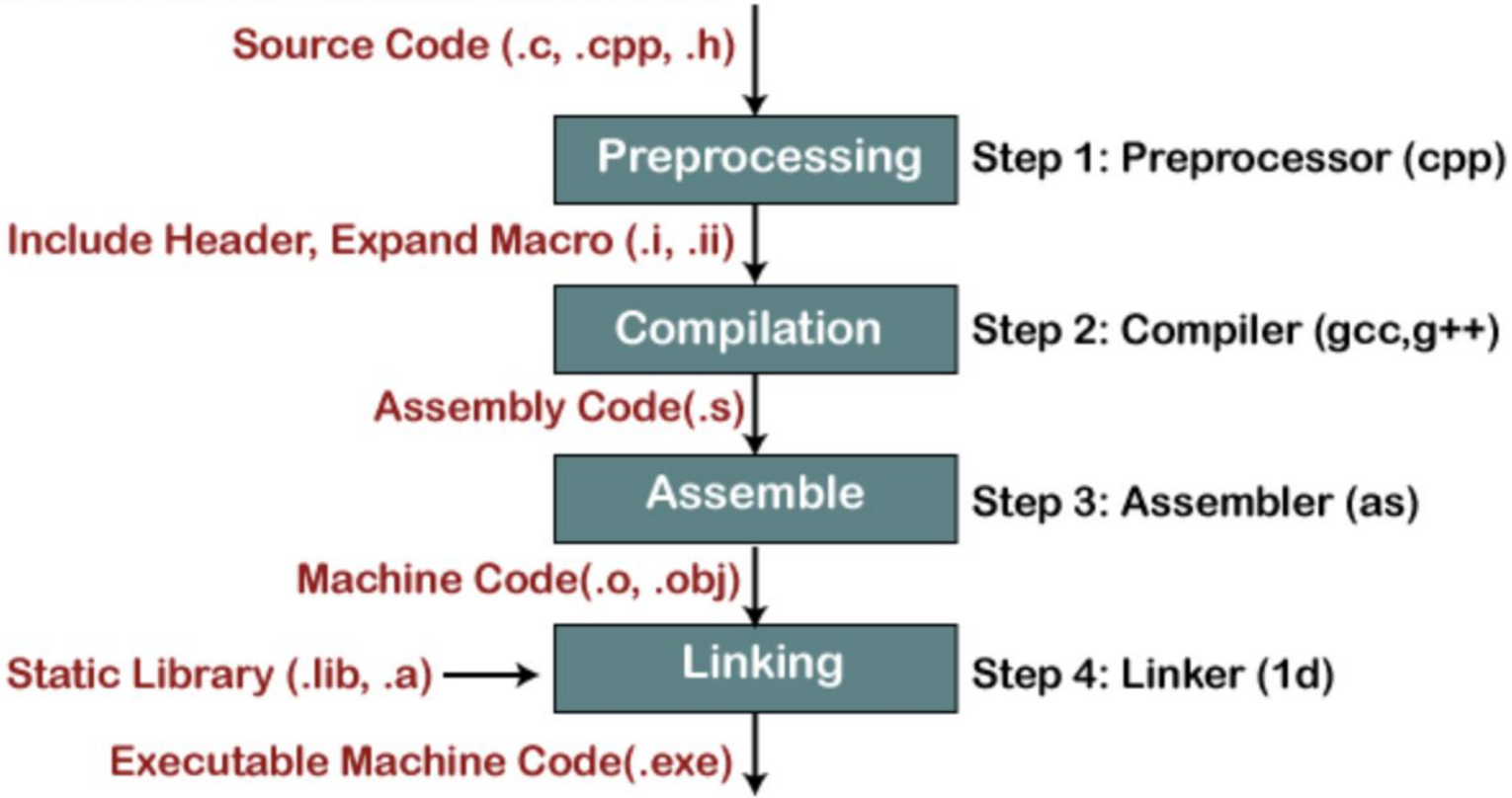
# Static Libraries Argc – Argv

# What are the steps involved in the C compilation process?

# Recap on the P.C.A.L process



# More Details

**Preprocessing:**
- The C preprocessor, often invoked automatically by the compiler, processes the source code before actual compilation.
- It performs tasks like including header files, macro expansion, and conditional compilation (e.g., `#include`, `#define`, `#ifdef`, `#ifndef`, etc.).
- The result is an expanded source code with all preprocessor directives replaced.

**Compilation:**
- The preprocessed source code is then passed to the C compiler, such as GCC or Clang.
- The compiler checks the syntax of your code, identifies any syntax errors, and generates an intermediate representation called assembly code or object code.
- No executable code is generated at this stage; it's still in a low-level, platform-specific form.

**Assembly (optional):**
- Some compilers proceed to an optional step of assembling the assembly code into machine code.
- This step is more relevant for low-level programming and embedded systems.

**Linking:**
- If your code consists of multiple source files or relies on external libraries, the linker comes into play.
- It combines the object code from the compilation step with other necessary object code files and libraries to create a single executable file.
- Symbol resolution and memory address assignment occur during this phase.

**Executable Output:**
- After linking, you obtain an executable file. This can be an executable binary on Unix-like systems (e.g., ELF format) or an executable file on Windows (e.g., EXE).
- The executable file contains machine code instructions that can be directly executed by the computer's CPU.

**Loading (Dynamic Linking):**
- For dynamically linked libraries, some linking might occur at runtime. The loader loads the necessary dynamic libraries when you run the program.
- This is common on many Unix-like systems (e.g., shared libraries).

**Execution:**
- Finally, you can execute the compiled program. The operating system loads it into memory, and the CPU executes the machine instructions in the program.

After step 3,  the resultant file is object code.

The compiler needs to know our functions and what they do. The compiler goes through our C code of our functions, transforms them into object code and links them to the object code of our main functions, resulting into an executable file ***main***

***=> obj code(main.c file) + obj code(C functions files) ⇒ executable file(****main****)***

# Recap on functions and Program Compilation

- Below is a small program that add and subtract two numbers.
- To illustrate how static libraries work better we need to implement each function in its own file.

add.c

```c
#include "main.h"
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}
```

subtract.c

```c
#include "main.h"
#include <stdio.h>

int subtract(int a, int b) {
    return a - b;
}
```

main.h

```
#ifndef MAIN_H
#define MAIN_H

int add(int a, int b);
int subtract(int a, int b);


#endif
```

# Explanation

- Each file contain a function that solves a particular problem
- If we compile the whole program, we get an executable file that when run is able to solve a bigger problem. For instance, addition, subtraction, multiplication, division etc...
- So in our main function, we just call the particular function to use it.
- But before that, we can compile our program using gcc compiler

# Code compilation

# What happens when our code base increases and we add more functionalities?

multiply.c

```c
#include "main.h"
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}
```

divide.c

```c
#include "main.h"
#include <stdio.h>

int divide(int a, int b) {
    return a / b;
}
```

Updated main.h

```
#ifndef MAIN_H
#define MAIN_H

int add(int a, int b);
int subtract(int a, int b);
int divide(int a, int b);
int multiply(int a, int b);

#endif
```

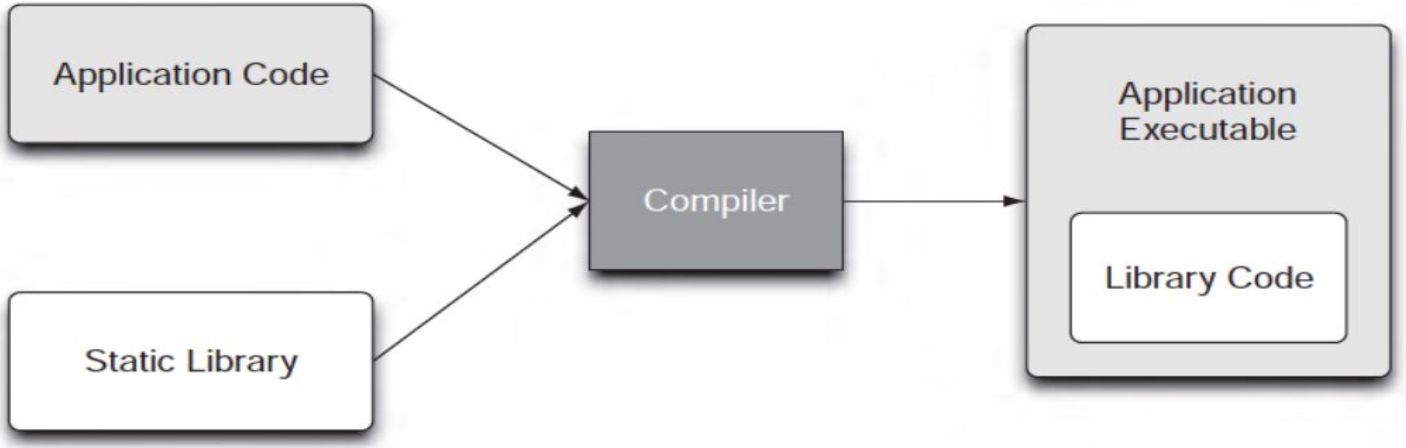
Here is how we compile it now:-

- Notice how when we add more function files to our program, the way we compile it becomes longer and longer?
- It was easier when we only had two function files, how about when we have a more complex program that requires up 100s of functions and files?
- How are we to compile it? Well, solutions lies on our using static library.

# Static Library

- In C programming language, a static library is a file containing compiled code that can be linked with a program at compile-time to provide additional functionality. A static library is called "`static`" because the library code is compiled into the final executable file, making it a part of the executable itself.
- In other words, a static library is a file containing several object files, that can be used as a single entity, and that are linked into the program during the linking phase of compilation

- So instead of compiling everything by including all the function files, we can just create a static library that we link to the application code at the linking step(step4 of PCAL).



# How do we create a static library?

**Step 1: Compile the functions files into their object files.** We use **-c** option to instruct the compiler not to link the files at the linking phase of compilation.

```
gcc -c *.c
```

**Step 2: Link our object files into an archive.** We use **ar** (archiver program). **-c** flag instructs the ar to create the library if it does not exist and **-r** replaces the older library objects files with the new ones.

- **lib**calculator**.a** will be the name of our library. 'calculator' is name and prefix '**lib**' and suffix '**.a**' are mandatory part of the name of the library.
- `*.o` are all object files created earlier.

  - `ar -rc libcalculator.a *.o`

- **Step 3: Indexing the library.** Every time a library is created or modified, we need to index it to speed up symbol lookup and to make sure the order of the symbol won't be a factor during compilation
- `ranlib libcalculator.a`

- How to list object files in the static library

  **`ar -t libcalculator.a`**
- How to list the symbols in the library

  **`nm libcalculator.a`**

# Using static library to compile out program

- We do this by adding the library to the list of files given to the linker.
- We use **`-l`** flag on the name to instruct the compiler that this is a static library file

```
gcc main.c -L. -lcalculator -o operations
```

- The resulting executable is ***`operations`***
- `-L` flags instructs the linker the location of the static library and "." here refers to the current working directory.

# Argc, ArgV

- In C programming language, `argc` and `argv` are variables that are used to handle command-line arguments.
- `argc` is an integer variable that represents the number of arguments passed to the program through the command line. It stands for "**argument count**". By convention, the first argument `argv[0]` is the name of the program itself. Thus, `argc` will always be at least 1.
- `argv` is an array of character pointers that contains the arguments passed to the program through the command line. It stands for "**argument vector**". The first element `argv[0]` is always the name of the program, and the following elements contain any additional arguments passed to the program.

- Listing all the arguments

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

# Sum all command line arguments

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

int main(int argc, char *argv[]) {
    int sum = 0;

    for (int i = 1; i < argc; i++) {
        sum += atoi(argv[i]);
    }

    printf("Sum: %d\n", sum);
    return 0;
}
```

# Resources

1. Comprehensive Lib Guide
2. Argc & argv
3. Commandline arguments

# See you at the next session!