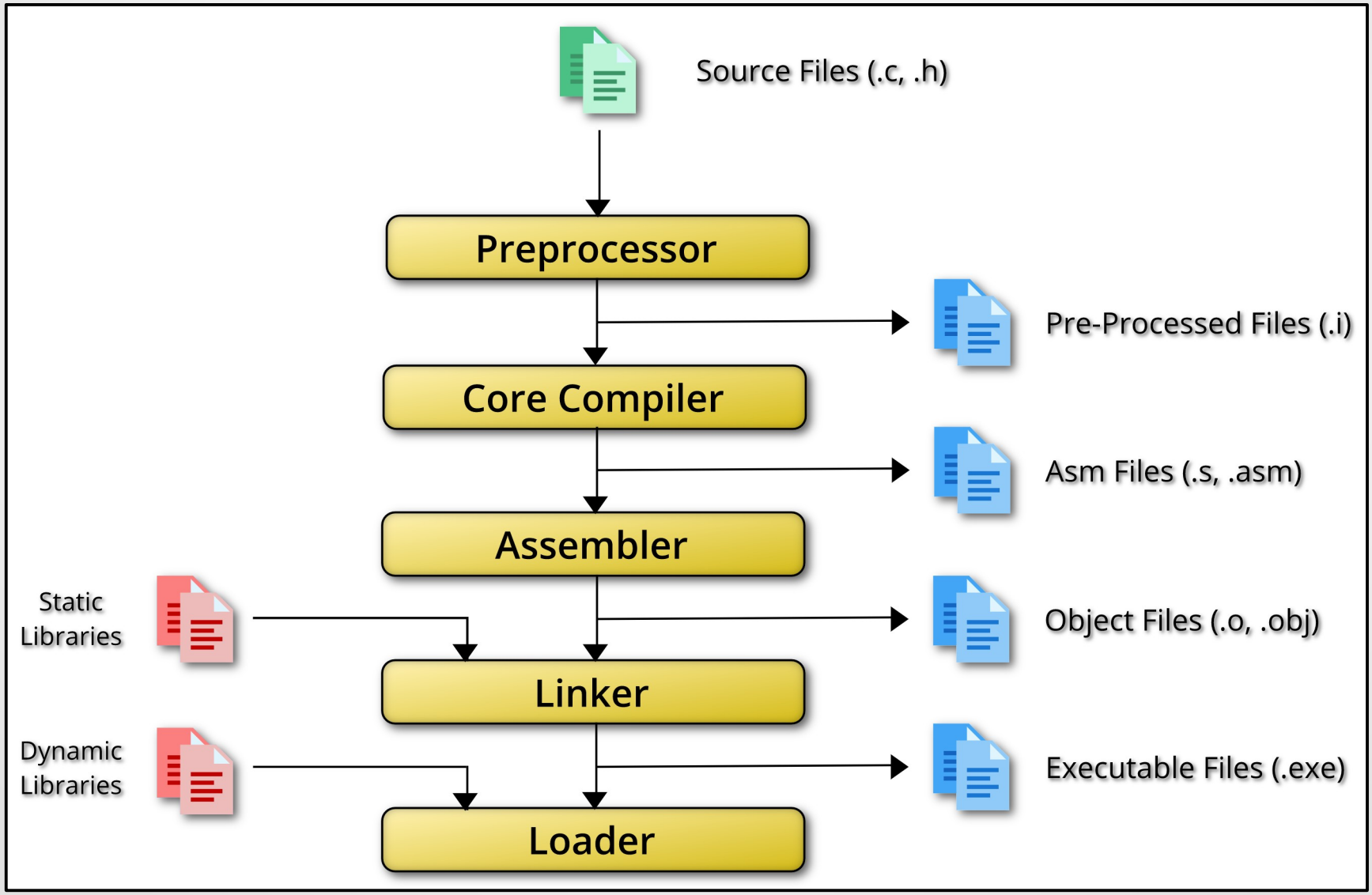
Practical Programming

The C Language : Compiling and Running

David Bouchet

david.bouchet.epita@gmail.com

Compiling and Running Processes (1)



Compiling and Running Processes (2)

Static Libraries

- Set of routines that are included in the executable file.
- The executable file is larger.
- The user does not have to own the required libraries.
- If several executable files use the same library, each file contains its own version of the library, which cannot be shared.

Compiling and Running Processes (3)

Dynamic Libraries

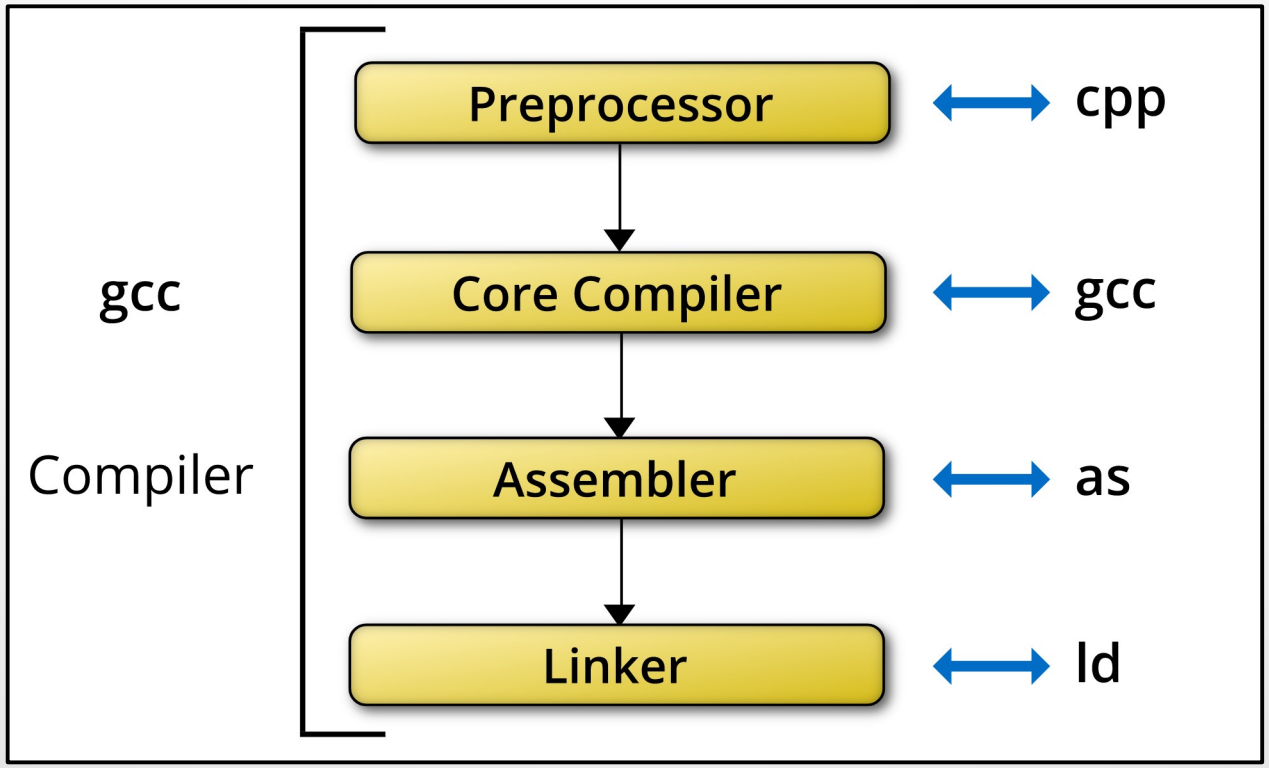
- Set of routines that are not included in the executable file.
- The executable file is smaller.
- The user has to own the required libraries.
- If several executable files use the same library, they all share the same version of the library.

Compiling and Running Processes (4)

- Preprocessor: Allows inclusion of files, macro expansions and conditional compilation. (Preprocessor instructions are prefixed with #.)
- Core compiler: Translates C language into assembly language.
- Assembler: Translates assembly language into native machine code (object files).
- Linker: Links different object files and libraries (if required) and generates an executable file.
- Loader: Loads an executable file into memory, links it to the dynamic libraries (if required) and executes it.

Compiling Process

Each stage of the compiling process can be done separately.
Each tool can be invoked one at a time.
But usually, they are invoked implicitly by the compiler.



Function Definitions and Declarations

A function must be **defined** or **declared** before being called.

Do not confuse **definitions** and **declarations**.

Function Definitions

Definition of *sum()*

Definition of *square_sum()*

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

int square_sum(int x, int y)
{
    return sum(x * x, y * y);
}
```

Call to *sum()*

It works because *sum()* is **defined before** the call.

Function Definitions

Definition of *square_sum()*

Definition of *sum()*

```
int square_sum(int x, int y)
{
    return sum(x * x, y * y);
}

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

Call to *sum()*

It does not work because *sum()* is **defined after** the call.

Prototypes

To declare a function, we use its **prototype**.

For example:

```
int sum(int a, int b);
```

is the prototype of the *sum()* function.

Function Declarations

```
int sum(int a, int b);

int square_sum(int x, int y)
{
    return sum(x * x, y * y);
}

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

Declaration of *sum()*

Call to *sum()*

It works because *sum()* is **declared before** the call.

Header Files

Header files are used to declare functions.

However, they may sometimes contain the definitions of small functions.

sum.h

```
int sum(int a, int b);
```

Header Files

```
#include “sum.h”

int square_sum(int x, int y)
{
    return sum(x * x, y * y);
}

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

It includes the declaration of *sum()*

Multiple Inclusions

sum.h

```
int sum(int a, int b);
```

sum_square.h

```
#include “sum.h”
int square_sum(int x, int y);
```

example.c

```
#include “sum.h”
#include “square_sum.h”

// … some code …
```

It includes the declaration of *sum()*

It includes the declarations of *sum()* and *sum_square()*

Problem: *sum()* is declared twice.

Include Guards

Include guards prevent multiple inclusions.

sum.h

```
#ifndef SUM_H
#define SUM_H

int sum(int a, int b);

#endif
```

sum_square.h

```
#ifndef SUM_SQUARE_H
#define SUM_SQUARE_H

#include “sum.h”
int square_sum(int x, int y);

#endif
```

#pragma once

The “#pragma once” directive is:

- Equivalent to include guards.
- Shorter than include guards.
- Supported by almost all compilers.
- Not in the standard.

#pragma once

sum.h

```
#pragma once

int sum(int a, int b);
```

sum_square.h

```
#pragma once

#include “sum.h”
int square_sum(int x, int y);
```

Your First Program

hello.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

<stdio.h>:

- Header file in the standard library.
- It contains the declaration of *printf()*.

Your First Program

```
$ ls
hello.c
$ gcc hello.c
$ ls
a.out hello.c
$ ./a.out
Hello World!
```

→ “a.out” is the default filename for the executable file.

Compiler Options - Examples

Some options can be used:

```
$ ls
hello.c
$ gcc -Wall -Wextra -Werror -O3 -o hello hello.c
$ ls
hello hello.c
$ ./hello
Hello World!
```

Compiler Options - Examples

- Wall: Enables all warnings.
- Wextra: Enables extra warnings.
- Werror: Makes all warnings into error.
- 03: Enables all optimizations.
- o: Specifies the output filename.

Multiple Files

main.c

```c
#include "greet.h"
#include "blank.h"

int main()
{
    hello();
    new_line();
    bye();
    new_line();

    return 0;
}
```

greet.c

```c
#include <stdio.h>

void hello()
{
    printf("Hello!");
}

void bye()
{
    printf("Good Bye!");
}
```

blank.c

```c
#include <stdio.h>

void new_line()
{
    printf("\n");
}
```

blank.h

```c
#pragma once

void new_line();
```

greet.h

```c
#pragma once

void hello();
void bye();
```

Multiple Files – Compilation – **Method 1**

Method 1

We can execute a single command line

Multiple Files – Compilation – **Method 1**

```
$ gcc blank.c greet.c main.c
$ ./a.out
Hello!
Good Bye!
```

Problem: All files are always compiled.

If we modify one file only, such a command recompiles all of the files.

Multiple Files – Compilation – **Method 2**

Method 2

We can separate the compilation process and the linking process

Multiple Files – Compilation – **Method 2**

Generating object files (preprocessor, core compiler, assembler)

```
$ gcc -c blank.c    # Generate blank.o
$ gcc -c greet.c    # Generate greet.o
$ gcc -c main.c     # Generate main.o
```

-c option: compilation only (no linking).
No executable file is generated.

Multiple Files – Compilation – **Method 2**

Linking object files and generating executable file (linker)

```
$ gcc blank.o greet.o main.o
$ ./a.out
Hello!
Good Bye!
```

Multiple Files – Compilation – **Method 2**

Let us modify greet.c

```
#include <stdio.h>
void hello()
{
    printf("Hello!");
}
void bye()
{
    printf("Ciao!");
}
```

Multiple Files – Compilation – **Method 2**

We can recompile greet.c only…

```
$ gcc -c greet.c  # Regenerates greet.o
```

… and regenerate the executable file.

```
$ gcc blank.o greet.o main.o
$ ./a.out
Hello!
Ciao!
```

Multiple Files – Compilation – **Method 2**

Problem

During the development process, we cannot memorize all of the files we modify.

So, we do not know which files have to be recompiled.

Multiple Files – Compilation – **Method 3**

Method 3

We can use GNU Make

Multiple Files – Compilation – **Method 3**

Makefile

```
a.out: blank.o greet.o main.o
   gcc blank.o greet.o main.o

blank.o: blank.c
   gcc -c blank.c

greet.o: greet.c
   gcc -c greet.c

main.o: main.c blank.h greet.h
   gcc -c main.c
```

Multiple Files – Compilation – **Method 3**

```
$ ls
blank.c  blank.h  greet.c  greet.h
main.c  Makefile
$ make
gcc -c blank.c
gcc -c greet.c
gcc -c main.c
gcc blank.o greet.o main.o
$ ./a.out
Hello!
Ciao!
```

Multiple Files – Compilation – **Method 3**

Let us modify greet.c

```c
#include <stdio.h>
void hello()
{
    printf("Hello!");
}
void bye()
{
    printf("Arrivederci!");
}
```

Multiple Files – Compilation – **Method 3**

```
$ make
gcc -c greet.c
gcc blank.o greet.o main.o
$ ./a.out
Hello!
Arrivederci!
$ make
make: 'a.out' is up to date.
```

GNU Make – First Makefile (3)

Makefiles can be much smarter than that.

To know more about Makefiles, read the following page:

https://slashvar.github.io/2017/02/13/using-gnu-make.html