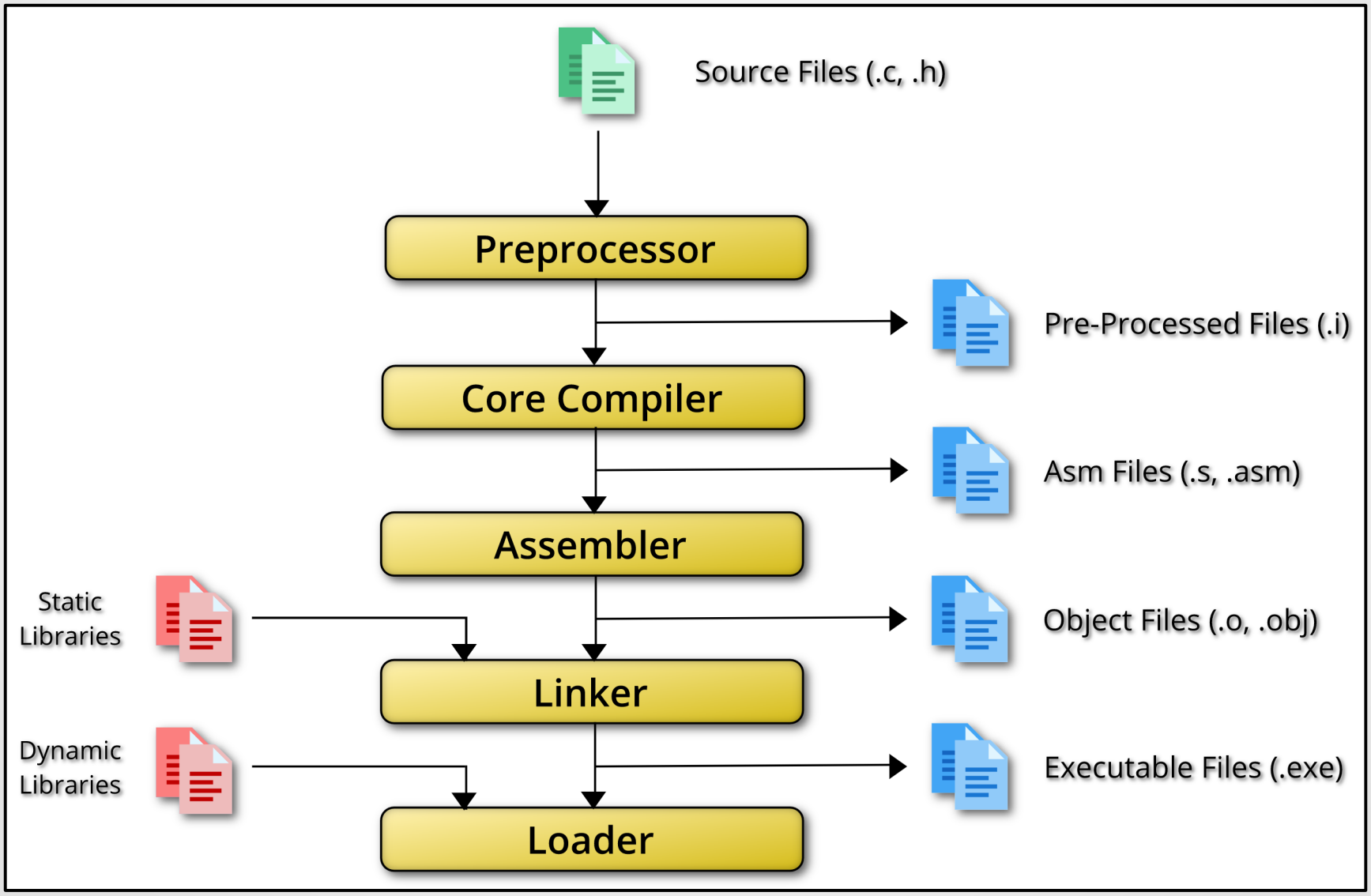
Practical Programming

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.

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 (3)

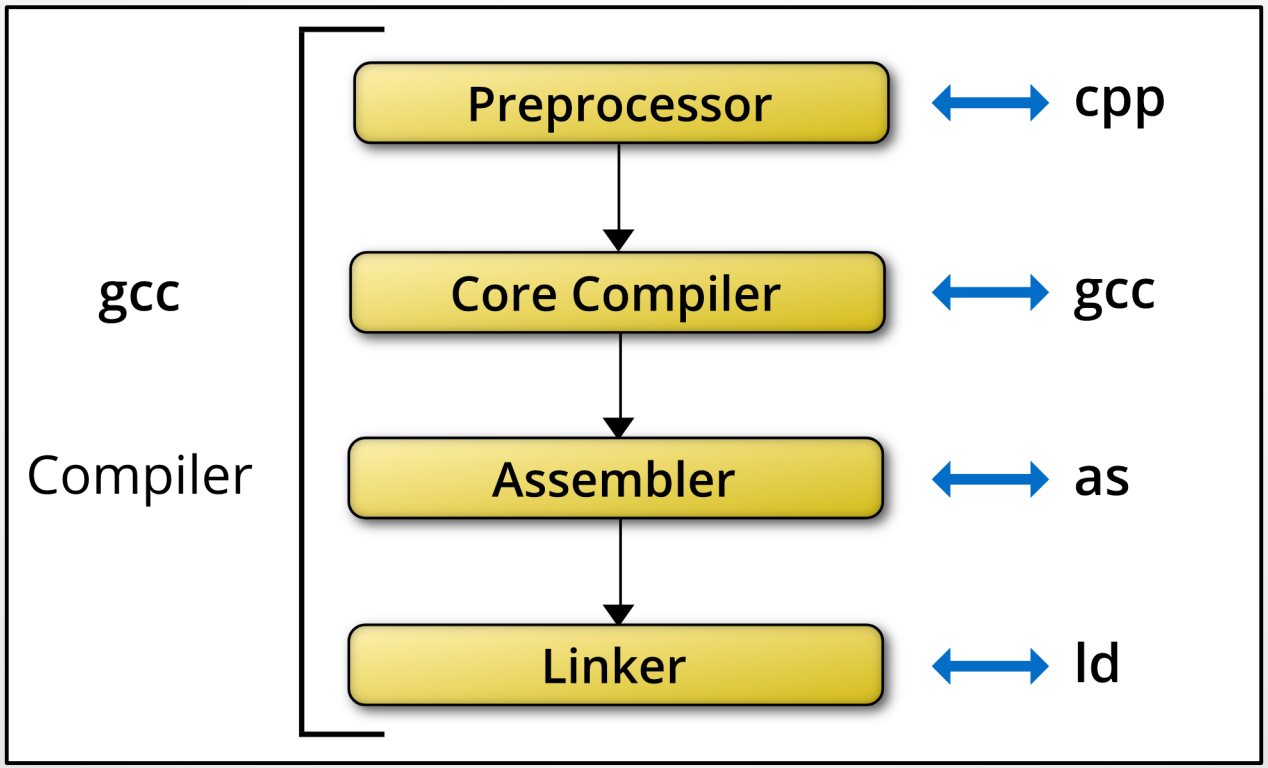
- Preprocessor: Allows inclusion of files, macro expansions and conditional compilation. (Preprocessor instructions are prefixed with #.)
- Core compiler: Translates C language into assembly language.
- Assembler: Translates assembly language into native machine code (object files).
- Linker: Links different 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.



Your First Program (1)

hello_world.c

```c
#include <stdio.h>

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

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

→ "a.out" is the default filename for the executable file.

Your First Program (2)

Some options can be used:

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

- 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 (1)

hello.c

```c
#include <stdio.h>
#include "blank.h"
#include "world.h"

void print_hello()
{
    printf("Hello");
}

int main()
{
    print_hello();
    space();

    print_world();
    new_line();

    // print_good_bye(); Not defined!

    return 0;
}
```

blank.c

```c
#include <stdio.h>

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

void space()
{
    printf(" ");
}
```

world.c

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

void print_good_bye();

void print_world()
{
    printf("World!");
    new_line();
    print_good_bye();
}

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

blank.h

```c
#ifndef BLANK_H
#define BLANK_H

void new_line();
void space();

#endif
```

world.h

```c
#ifndef WORLD_H
#define WORLD_H

void print_world();

#endif
```

Header Files
.h = .header

Multiple Files (2)

Generate the object files (preprocessor, core compiler, assembler)

```
$ gcc -c blank.c     # Generate blank.o
$ gcc -c world.c     # Generate world.o
$ gcc -c hello.c     # Generate hello.o
```

Link the object files and generate the executable file (linker)

```
$ gcc hello.o blank.o world.o
```

Execute the executable file (loader)

```
$ ./a.out
Hello World!
Good Bye!
```

Multiple Files (3)

Let us modify “world.c”

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

void print_good_bye();

void print_world()
{
    printf("World!");
    new_line();
    print_good_bye();
}

void print_good_bye()
{
    printf("Ciao!");    // This line has changed.
}
```

Multiple Files (4)

Generate the object files (preprocessor, core compiler, assembler)

```
$ gcc -c world.c     # Update world.o
```

We do not have to update “hello.o” and “blank.o” because “hello.c” and “blank.c” have not been modified.

Link the object files and generate the executable file (linker)

```
$ gcc hello.o blank.o world.o
```

Execute the executable file (loader)

```
$ ./a.out
Hello World!
Ciao!
```

GNU Make – First Makefile (1)

Makefile

```
# Define the default target.

all: hello

# Define dependencies and compile information.

hello: hello.o blank.o world.o
    gcc hello.o blank.o world.o -o hello

hello.o: hello.c blank.h world.h
    gcc -c hello.c

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

world.o: world.c blank.h
    gcc -c world.c
```

```
$ make
gcc -c hello.c
gcc -c blank.c
gcc -c world.c
gcc hello.o blank.o world.o -o hello
$ ./hello
Hello World!
Ciao!
```

GNU Make – First Makefile (2)

Let us modify “world.c”

```
#include <stdio.h>
#include "blank.h"

void print_good_bye();

void print_world()
{
    printf("World!");
    new_line();
    print_good_bye();
}

void print_good_bye()
{
    printf("Arrivederci!");    // This line has changed.
}
```

```
$ make
gcc -c world.c
gcc hello.o blank.o world.o -o hello
$ ./hello
Hello World!
Arrivederci!
$ make
Make: Nothing to be done for 'all'.
```

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

Some Terminology

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

void print_good_bye();

void print_world()
{
    printf("World!");
    new_line();
    print_good_bye();
}

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

Declare a function

Define a function

Define a function

To **declare** a function, we use its **prototype**. For instance:

```c
void print_good_bye();
```

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

Header Files

- They can be included only once.
- They contain declarations.

```
#ifndef BLANK_H
#define BLANK_H

void new_line();
void space();

#endif
```

Conditional compilation.
Prevent from being included several times.

Function declarations.

Note that:

- Compiling (generating object files) needs declarations.
- Linking (generating executable files) needs definitions.