

Learn Programming Basics (C Language)

LESSON #003

Stages of Compiling

Purpose

- able to understand when, where and what errors are occurs at which stage in programming

Objective

- learn the stages of how your code is being compiled

- Create a text file and rename it to “**a.c**” (change its extension as well)
- Copy the snippet of code exactly

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

Explanation

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("peko peko");
6     return 0;
7 }
8
```

<- include <stdio.h> library
.h means header file

Explanation

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

<- main point of entry of a program

Explanation

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

- the function “printf” is comes from the library <stdio.h>

Explanation

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

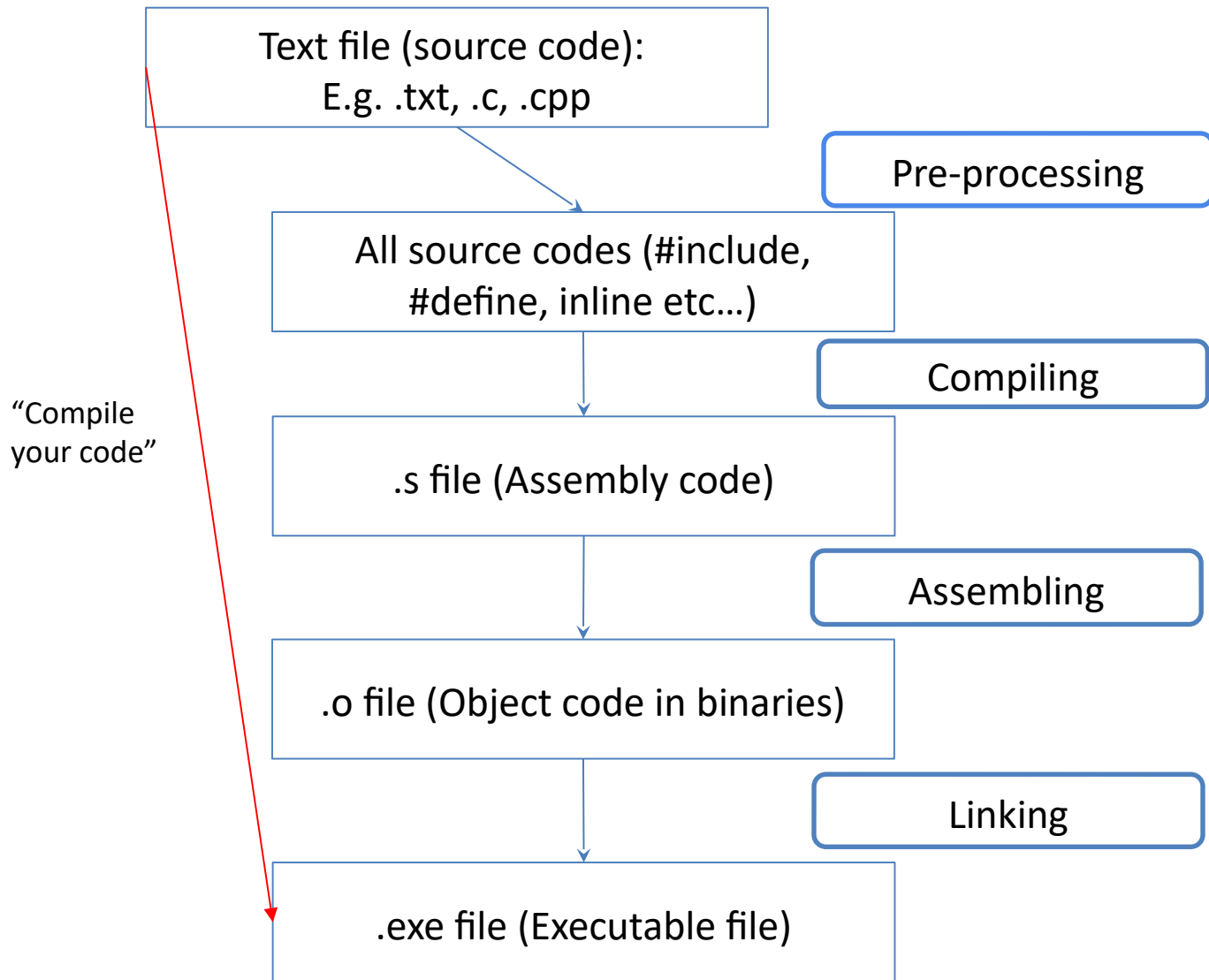
- return 0;
- return output/resource from the function.
- learn more about function in the future.

Explanation

```
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

- the function “printf” came from the library <stdio.h>.
- If the line #include<stdio.h> is removed, it is a **linker error** during the **linking stage**.
- The error described that it could not find the function’s definition, or function’s body, more details in next future lessons.

Demonstration (Compiling a program)



Example 1: printing out a “string literal”

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("peko peko");
6      return 0;
7  }
8
```

<- prints the string “peko peko”

- Compile with this command line, follow by the file “a.c”:
- gcc -Werror -Wall -Wextra -ansi -pedantic a.c
- Run the program command line: ./a.exe OR ./a

```
$ gcc -Werror -Wall -Wextra -ansi -pedantic a.c
```

```
$ ./a
peko peko
```

Compile and run, it should print
“peko peko”

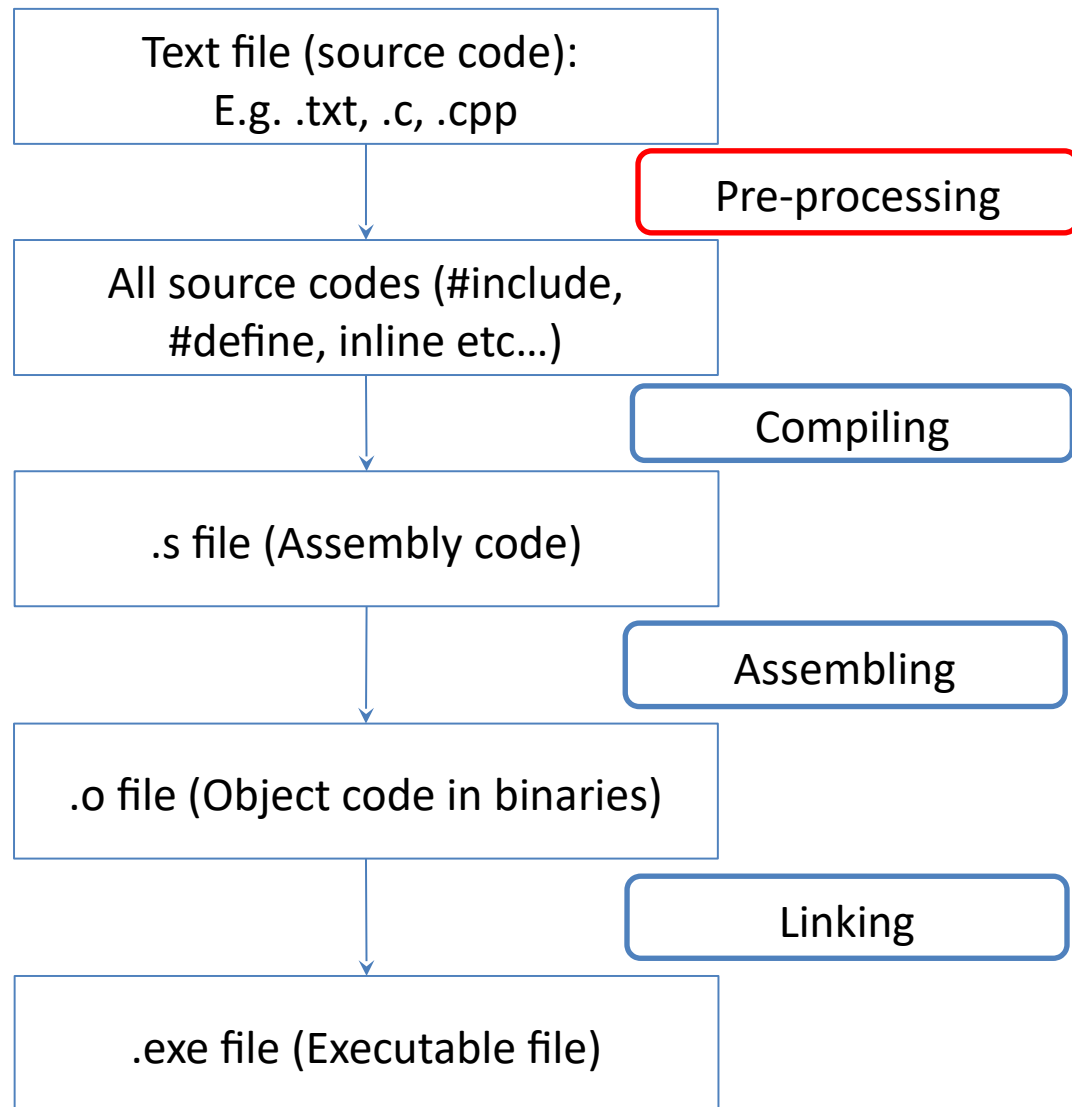
What does this mean?

```
$ gcc -Werror -Wall -Wextra -ansi -pedantic a.c
```

- gcc (run the C compiler program)
- -Werror (flag, make all warnings into errors)
- -Wall (flag, enables all warnings flag)
- -Wextra (flag, enables more extra warnings flag)
- -ansi (flag, standardize flag enable)
- -pedantic (flag, enable warning flag)
- a.c (filename)

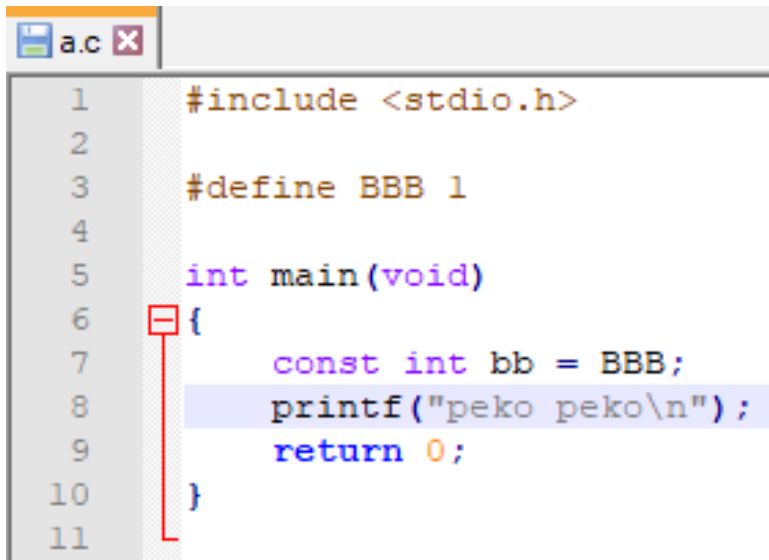
for the purpose of learning, strict rules to follows during compiling to learn better, compiling arguments may change accordingly

Pre-processing stage



Pre-Processing Stage

- Read Libraries, etc
- Remove comments from all compiling files



```
1 #include <stdio.h>
2
3 #define BBB 1
4
5 int main(void)
6 {
7     const int bb = BBB;
8     printf("peko peko\n");
9     return 0;
10 }
11
```

- Copy this code into a.c

Try this command: **gcc -E a.c**

What do you observe?



```
$ gcc -E a.c
```


You found a very long chunk of code at the top

```
$ gcc -E a.c
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "a.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 29 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/_ansi.h" 1 3 4
# 10 "/usr/include/_ansi.h" 3 4
# 1 "/usr/include/newlib.h" 1 3 4
# 14 "/usr/include/newlib.h" 3 4
# 1 "/usr/include/_newlib_version.h" 1 3 4
# 15 "/usr/include/newlib.h" 2 3 4
# 11 "/usr/include/_ansi.h" 2 3 4
# 1 "/usr/include/sys/config.h" 1 3 4

# 1 "/usr/include/machine/ieeefp.h" 1 3 4
```

And the very bottom you should see this:

```
# 5 "a.c"
int main(void)
{
    const int bb = 1;
    printf("peko peko\n");
    return 0;
}
```

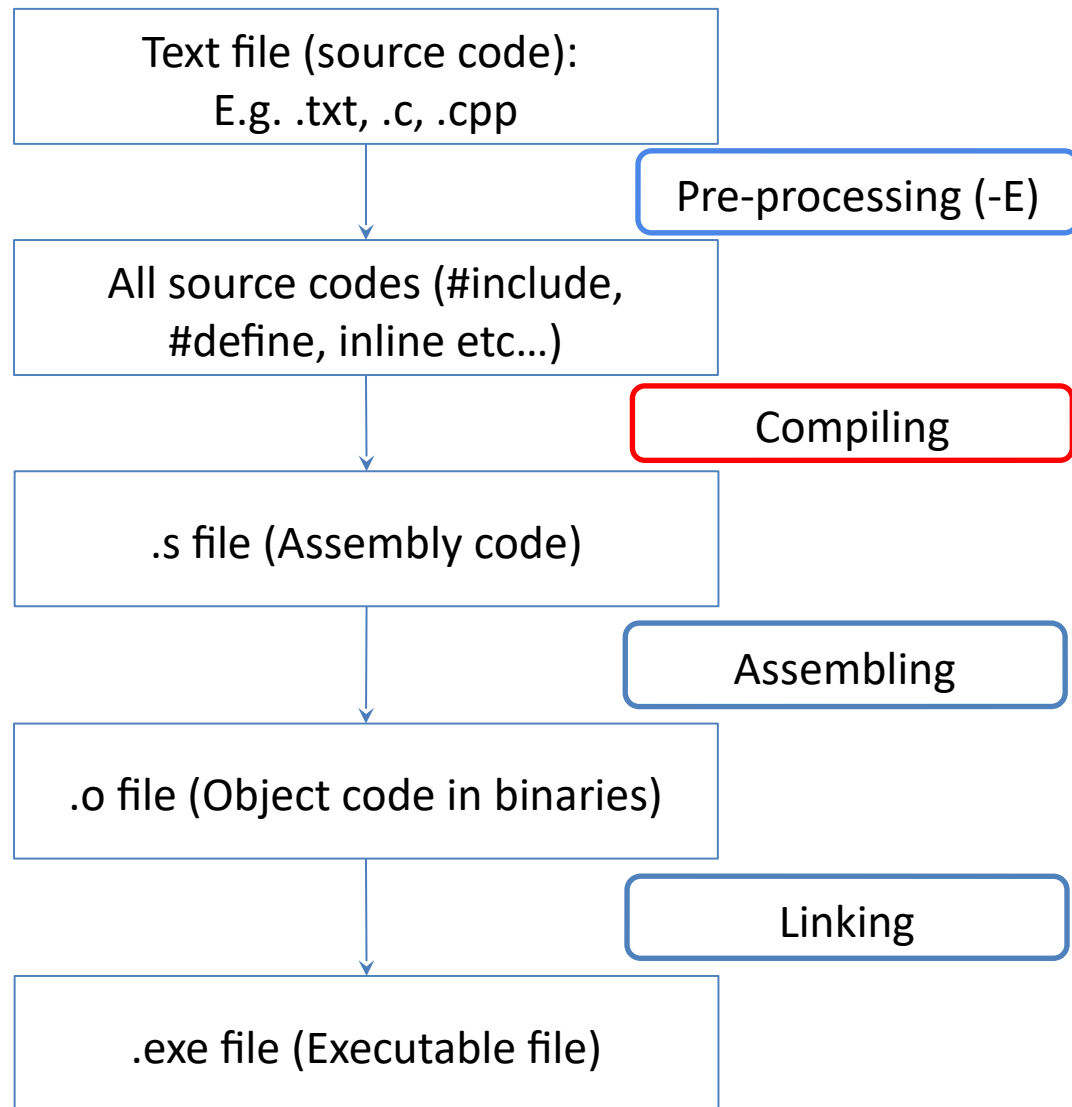
Sees something familiar?

```
$ gcc -E a.c
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "a.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 29 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/_ansi.h" 1 3 4
# 10 "/usr/include/_ansi.h" 3 4
# 1 "/usr/include/newlib.h" 1 3 4
# 14 "/usr/include/newlib.h" 3 4
# 1 "/usr/include/_newlib_version.h" 1 3 4
# 15 "/usr/include/newlib.h" 2 3 4
# 11 "/usr/include/_ansi.h" 2 3 4
# 1 "/usr/include/sys/config.h" 1 3 4

# 1 "/usr/include/machine/ieeefp.h" 1 3 4
```

```
# 5 "a.c"
int main(void)
{
    const int bb = 1;
    printf("peko peko\n");
    return 0;
}
```

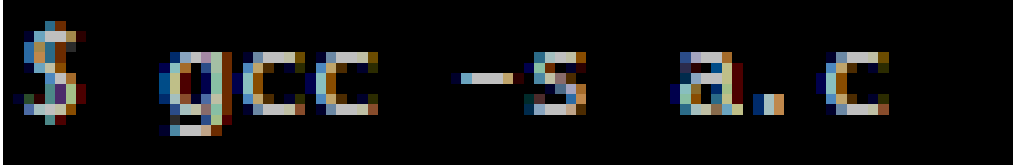
Compile stage



Compiling Stage

- Translation of C code into Assembly code

Try this command: **gcc -s a.c**

A screenshot of a terminal window with a black background. The text '\$ gcc -s a.c' is displayed in a light blue, monospaced font. The dollar sign is at the start of the line, followed by a space, then 'gcc', another space, '-s', a third space, 'a.', and finally '.c'.

A “a.s” file should be generate in your pwd, open the file with notepad++ or notepad

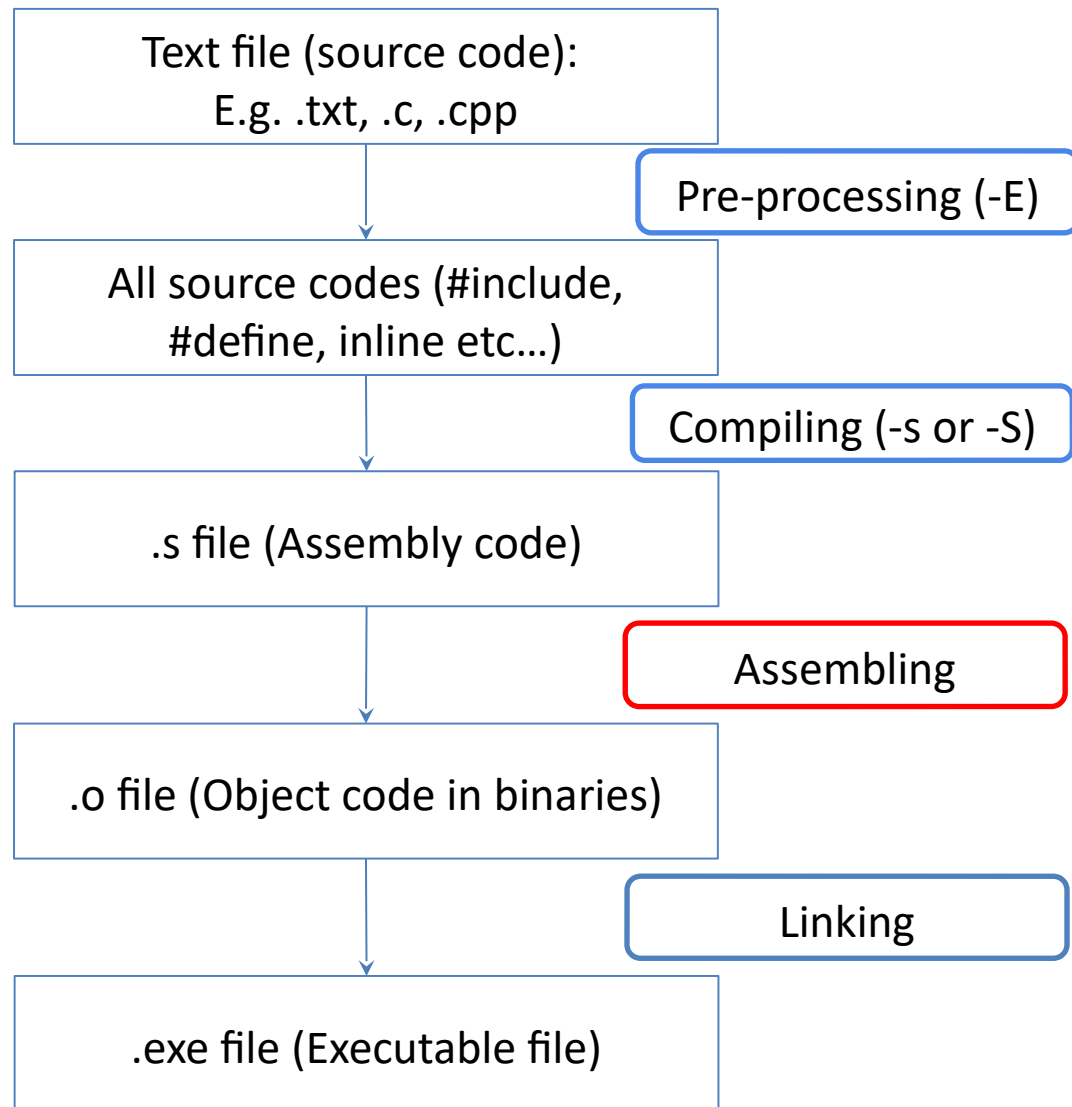
You should see a bunch of **instructions** with **arguments** a.k.a low-level language (still human readable codes)

```
.file "a.c"
.text
.def __main; .scl 2; .type 32; .endef
.section .rdata,"dr"
.LC0:
.ascii "peko peko\0"
.text
.globl main
.def main; .scl 2; .type 32; .endef
.seh_proc main
main:
pushq %rbp
.seh_pushreg %rbp
movq %rsp, %rbp
.seh_setframe %rbp, 0
subq $32, %rsp
.seh_stackalloc 32
.seh_endprologue
call __main
leaq .LC0(%rip), %rcx
call printf
movl $0, %eax
addq $32, %rsp
popq %rbp
ret
.seh_endproc
.ident "GCC: (GNU) 10.2.0"
.def printf; .scl 2; .type 32; .endef
```

In this stage, High-level language is converted into assembly code,

which is to be prepared to be converted into opcode and operands.

Assembling stage



Assembling Stage

- Translation of into Assembly code into Incomplete machine code

Try this command: **gcc -c a.c**

```
$ gcc -c a.c
```

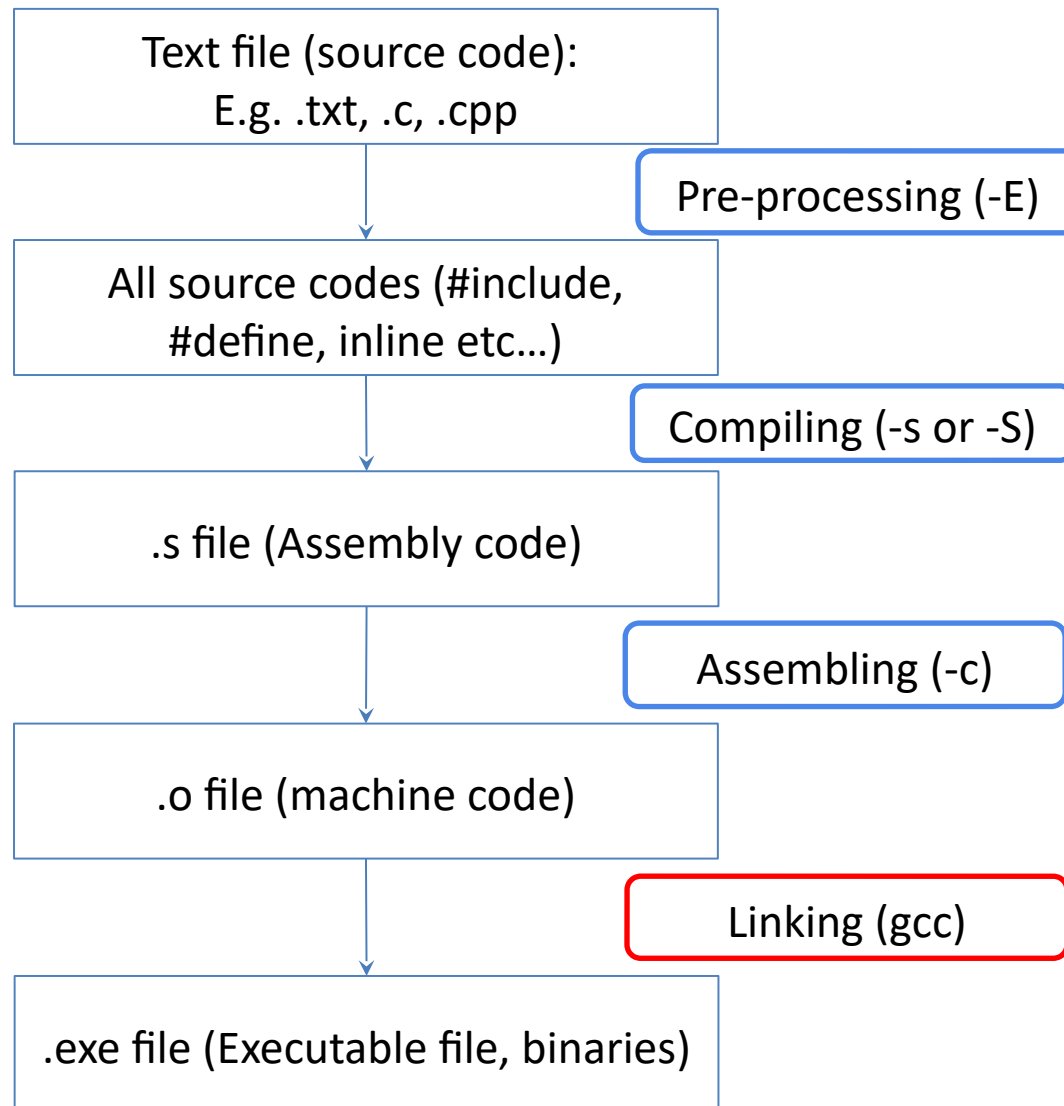
An object file, “**a.o**” should be generated, what you see are incomplete machine code

```

1 d†BELNULNULNULNULNULSO STX NULNULSYNNULNULNULNULEOTNUL.t
2 NULNULNULNULèNULNULNULNULèNULNULNULNUL:ÀH
3 NULNULNULèNULNULNULNUL,NULNULNULNULHfÄ ]Äpeko pekoNUL%d
4 NULNULNULSOHBS ETXENQBS2EOTETXSOHPNULNULNULNULNUL7NULNU
5 NULNULNULEOTNULNAKNULNULNULDC3NULNULNULEOTNULSUBNULNULNUL
6 NULNULNULEOTNUL(NULNULNULNAKNULNULNULEOTNULNULNULNULEO

```

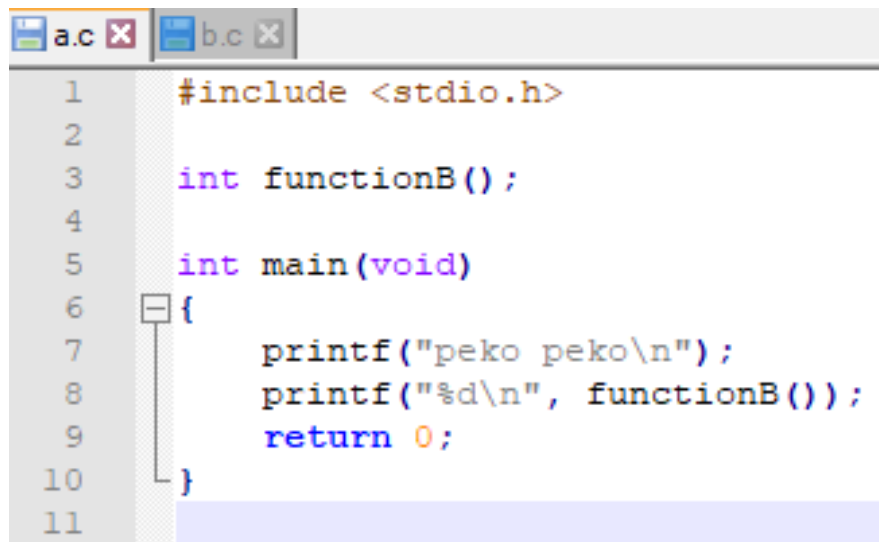
Linking stage



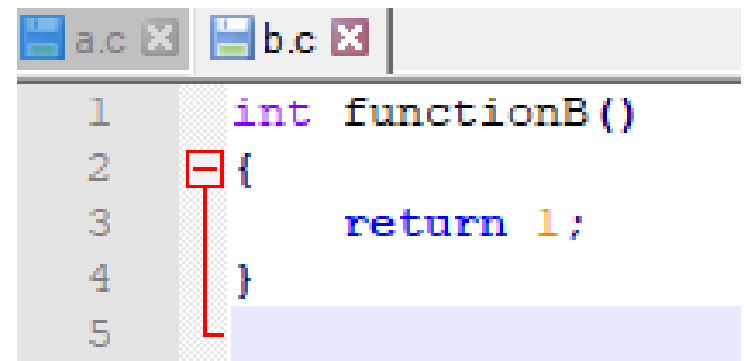
Linking Stage

- Combination of all incomplete machine code into one complete file
- Linking error happens here
 - it means that a called “component” is missing

Copy these code into a.c and b.c respectively

A screenshot of a code editor window with two tabs: 'a.c' and 'b.c'. The 'a.c' tab is active. The code in 'a.c' is as follows:

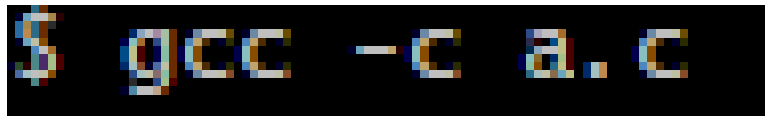
```
1  #include <stdio.h>
2
3  int functionB();
4
5  int main(void)
6  {
7      printf("peko peko\n");
8      printf("%d\n", functionB());
9      return 0;
10 }
11
```

Line 6 has a small square icon, and a vertical line connects it to the opening curly brace of the main function. Line 11 is highlighted with a light blue background.A screenshot of a code editor window with two tabs: 'a.c' and 'b.c'. The 'b.c' tab is active. The code in 'b.c' is as follows:

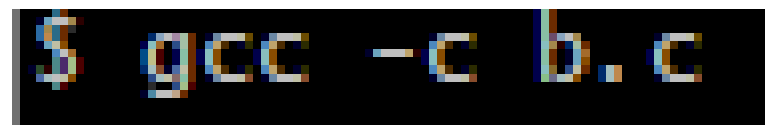
```
1  int functionB()
2  {
3      return 1;
4  }
5
```

Line 2 has a small square icon, and a vertical line connects it to the opening curly brace of the functionB function. Line 5 is highlighted with a light blue background.

Try this command: **gcc -c a.c**

A screenshot of a terminal window showing the command `$ gcc -c a.c` being executed. The output is not visible.

Try this command: **gcc -c b.c**

A screenshot of a terminal window showing the command `$ gcc -c b.c` being executed. The output is not visible.

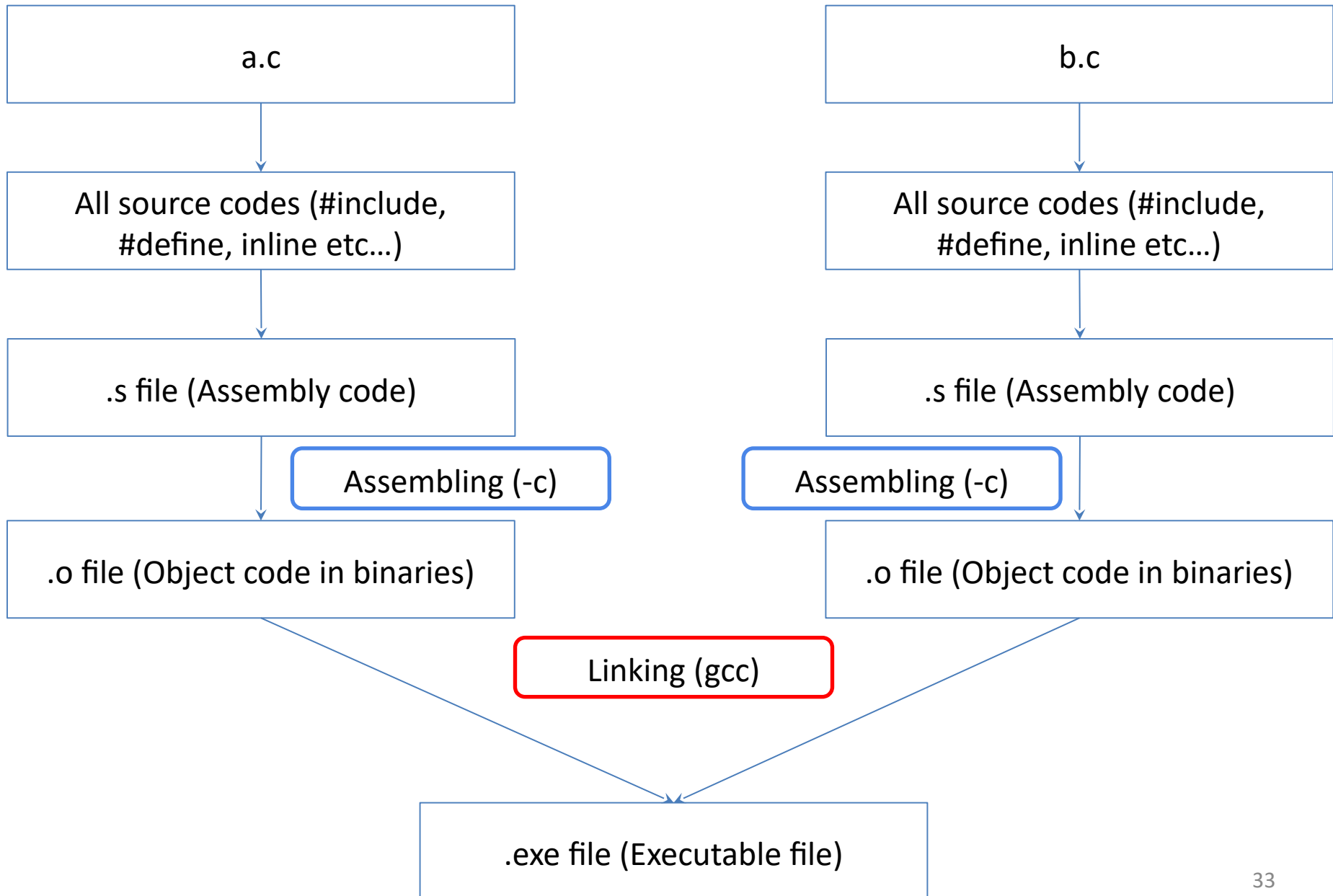
- Two object file should appear, **a.o** and **b.o**
- Next, type the command: **gcc a.o b.o**

```
$ gcc a.o b.o
```

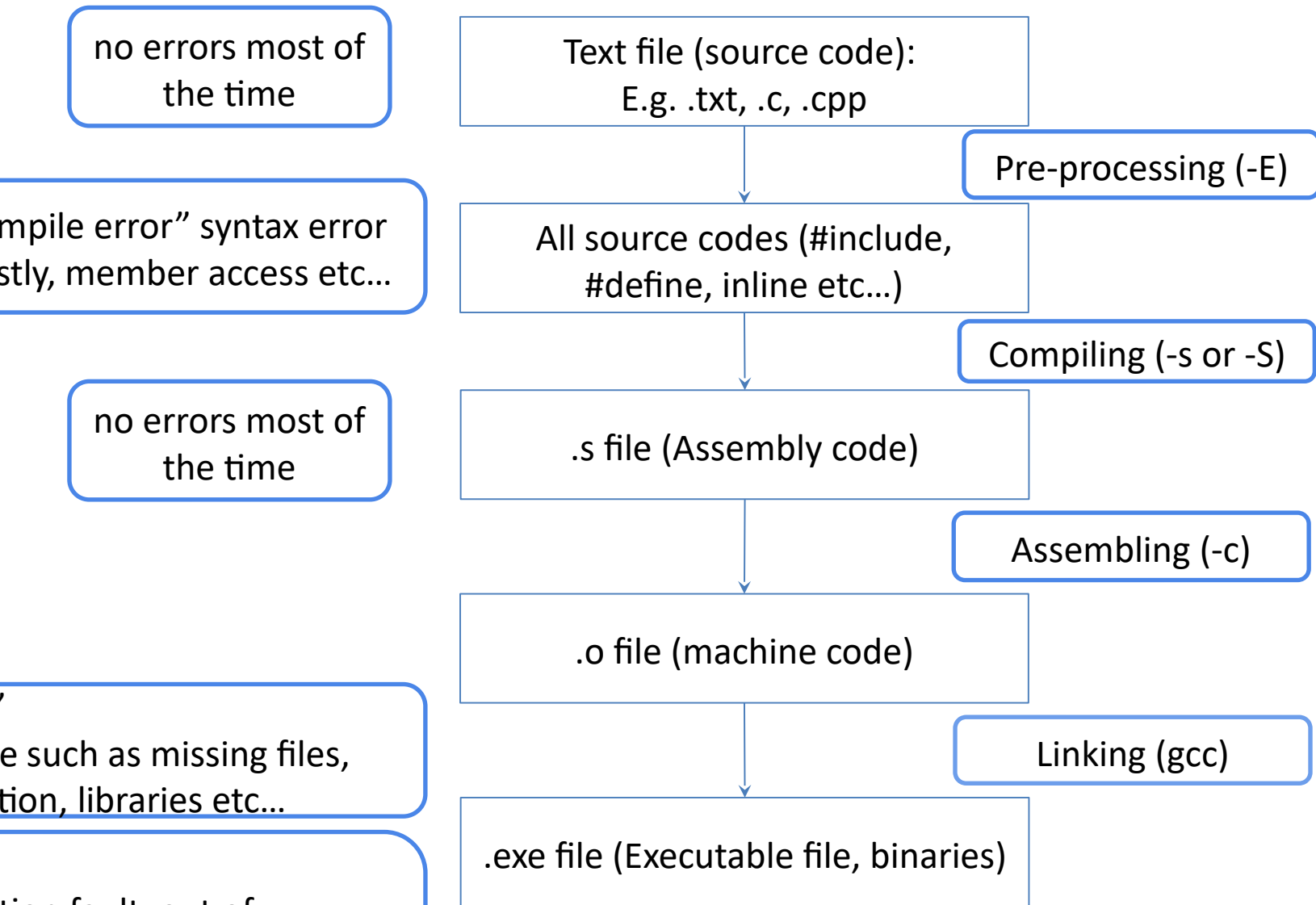
- Run the program: **./a**

```
$ ./a  
peko peko  
1
```


- What's really happening:



Errors Occuring



Compile Error

- Happens when the program is compiling
- syntax error mostly, member access etc...

Linker Error

- Happens after the program compiled successfully and right before generating .exe file
- missing linkage such as missing files, missing definition, libraries etc...

Runtime Error

- Happens when the program is running;
- memory out of range/segmentation fault;
- out of memory(RAM);
- dereferencing null pointer (accessing staff that doesn't exist);
- explicitly coded to quit the application when error occurs
- etc....

Purpose

- Be able to understand when, where and what errors occurs at which stage.
- Eventually you will be able to identify the different types of errors on which stage and where etc....; the more you code.

Objective

- learn the stages of how your code is being compiled.