gcc

- Randal Bryant, et.al., Computer Systems: A Programmer's Perspective, 3rd, Pearson, 2015
- Using the GNU Compiler Collection https://gcc.gnu.org/onlinedocs/gcc-8.5.0/gcc/

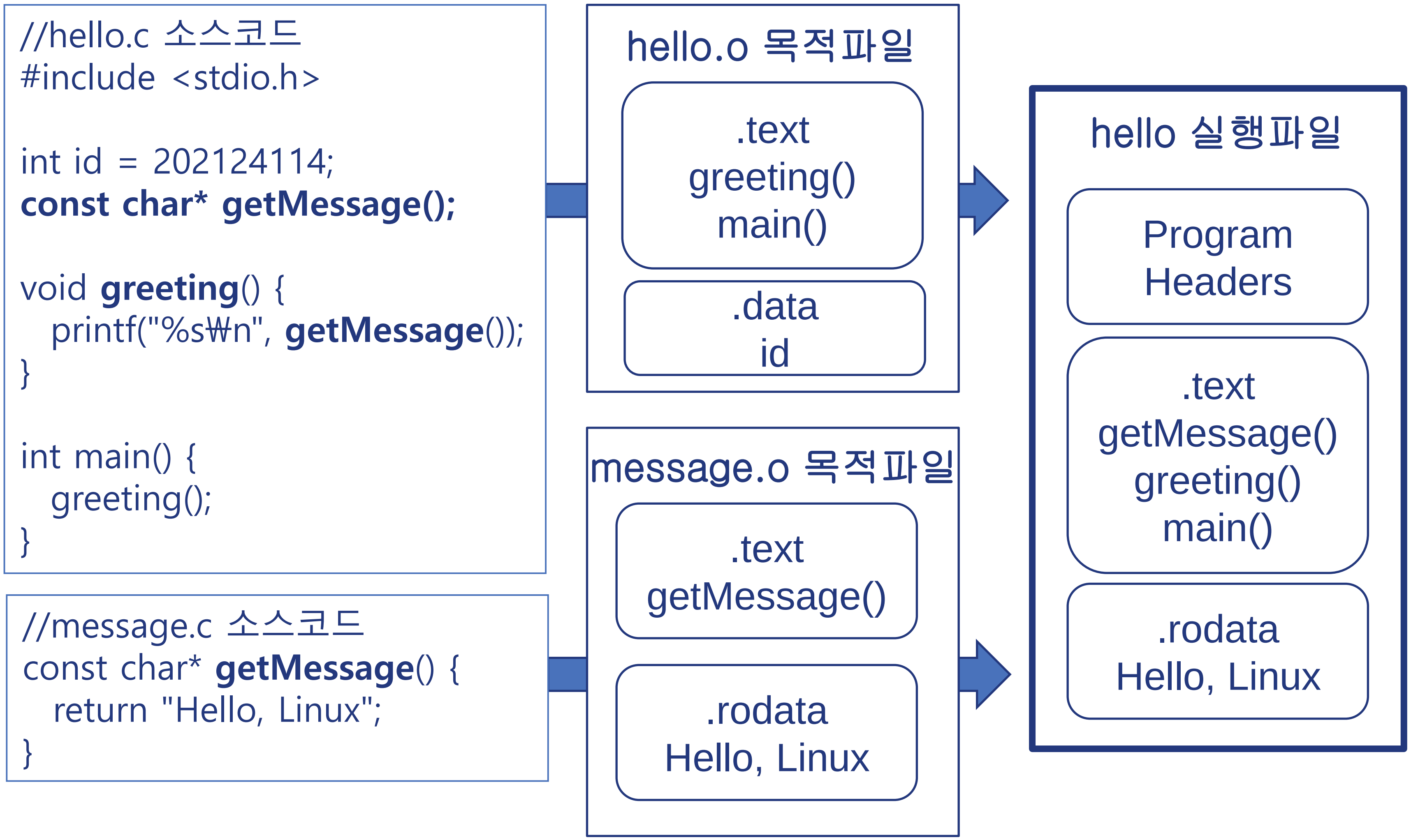
gcc

- GCC stands for “GNU Compiler Collection”
- GCC is an integrated distribution of compilers for several major programming languages.
  - C, C++, Objective-C, Fortran, Ada, Go, etc.
- The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990.
  - This standard was ratified as an ISO standard (ISO/IEC 9899:1990) later in 1990
  - A fourth version of the C standard, known as C11, was published in 2011 as ISO/IEC 9899:2011.
  - GCC has substantially complete support for this standard, enabled with ‘-std=c11’
  - The default, if no C language dialect options are given, is ‘-std=gnu11’.

Compiler

- 컴파일러는 소스 언어의 프로그램을 읽어서 의미가 같은 타켓 언어의 프로그램으로 변역하는 프로그램
  - C 언어로 작성된 소스 코드를 읽어서 실행 가능한 기계어 (machine language) 프로그램으로 변역함
- 컴파일러는 전단부에서 소스 프로그램을 어휘 분석(토큰), 구문 분석(파싱, 구문 트리), 의미 분석(타입 체크, 강제 형변환) 단계 등을 거쳐 중간 표현 코드를 생성함
- 컴파일러는 후단부에서 중간 표현 코드를 이용하여 코드 최적화 (빠르게 실행, 크기가 작음) 를 수행 후 특정 목표 기계를 위한 코드를 생성함
  - C 소스 코드 하나 당 하나의 목적 파일을 생성함(e.g., add.c, sub.c, calc.c 별로 add.o, sub.o, calc.o 각각 생성함)
  - gcc -O2 (최적화 옵션), gcc -c add.c (add.o 파일 생성)

Hello Linux C Program



```
//hello.c 소스코드
#include <stdio.h>

int id = 202124114;
const char* getMessage();

void greeting() {
   printf("%s₩n", getMessage());
}

int main() {
   greeting();
}
```

```
//message.c 소스코드
const char* getMessage() {
   return "Hello, Linux";
}
```

Matt Godbolt, "The Bits Between the Bits: How We Get to main()", CppCon, 2018 4

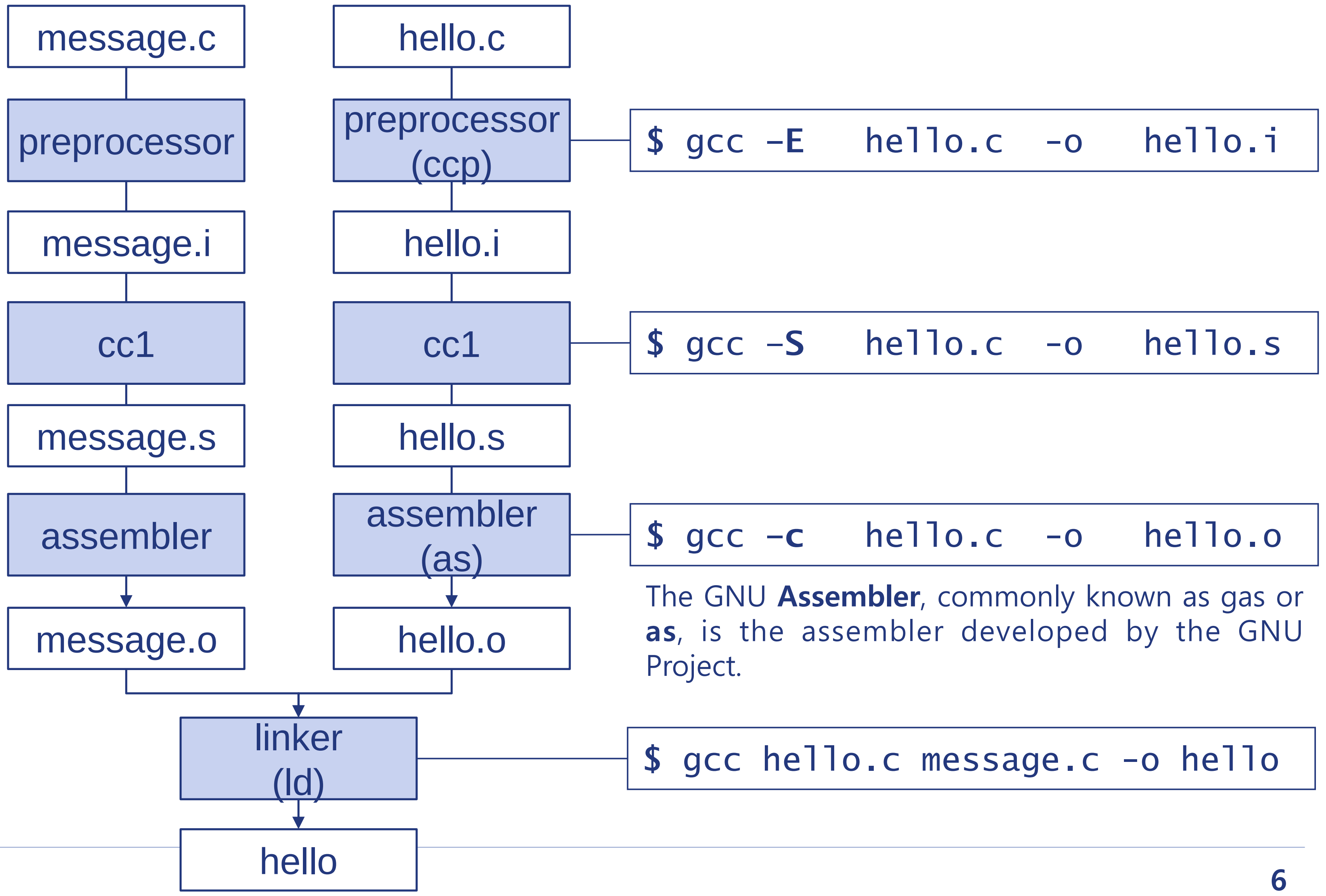
Declaration vs Definition in C

- A declaration can be done any number of times but **definition only once (one definition rule, ODR).**
- The ***extern*** keyword is used to extend the visibility of variables/functions.
- **Functions** are visible throughout the program **by default**.
- When ***extern*** is used with a variable, it's only declared, not defined **except for with initialization.**

선언 (일반적으로 .h 파일)	정의 (일반적으로 cpp 파일)

extern int max ;
struct Foo { int seq; } ;
int add(int a, int b) ;
	int len;
	int sum = 0;
	int add(int a, int b) {
	    return a + b;
	}
	struct Foo foo;

Building Process for the C Program



```
$ gcc -E   hello.c  -o   hello.i
```

```
$ gcc -S   hello.c  -o   hello.s
```

```
$ gcc -c   hello.c  -o   hello.o
```

The GNU **Assembler**, commonly known as gas or **as**, is the assembler developed by the GNU Project.

```
$ gcc hello.c message.c -o hello
```

GCC Command Options

- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking.
- The "overall options" allow you to stop this process at an intermediate stage.
  - e.g., the '-c' option says not to run the linker. The output consists of object files output by the assembler
- Many options have multi-letter names; therefore multiple single-letter options may not be grouped:
  - '-dv' is very different from '-d -v'.
- You can mix options and other arguments.
- For the most part, the order you use doesn't matter.

filename	Description
file.h	C, C++ header file to be turned into a precompiled header
file.i	C source code that should not be preprocessed
file.c	C source code that must be preprocessed
file.s	Assembler code
file.o	An object file to be fed straight into linking

GCC Command Options (con't)

- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking.

option	Description
-E	**Stop after the preprocessing stage**; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to t he standard output.
-S	**Stop after the stage of compilation proper**; do not assemble. By default, the assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'.
-c	**Compile or assemble the source files, but do not link.** By default, the object file name for a source file is made by replacing the suffix '.c', '.i', '.s', etc., with '.o'.
-o file	**Place output in file file. This applies to whatever sort of output is being produced**, whether it be an executable file, an object file, an ass embler file or preprocessed C code. If '-o' is not specified, the default is to put an executable file in 'a.out'
-v	Print (on standard error output) the commands executed to run the stag es of compilation.

Preprocessor

- The C preprocessor, often known as cpp, is a macro processor that is used automatically by the C compiler to transform your program before compilation.

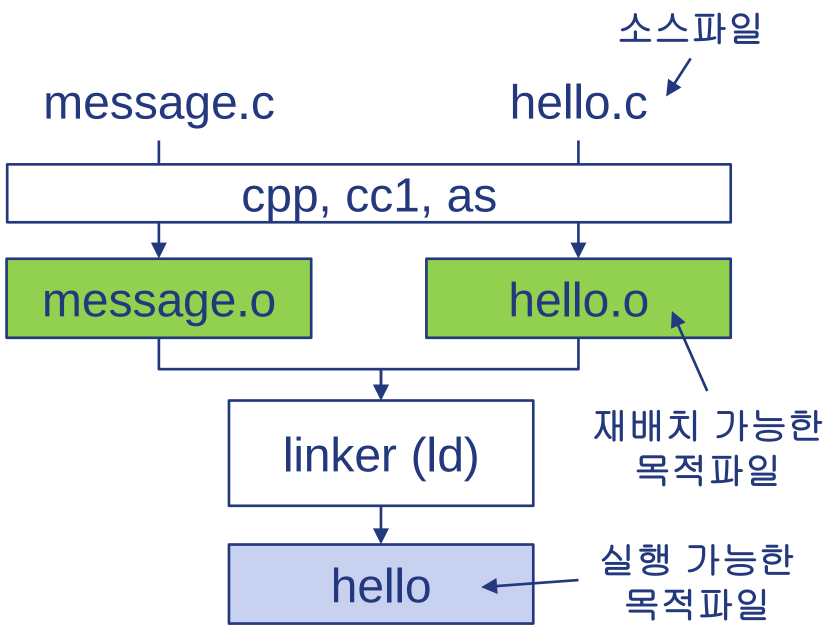
- There are 3 main types of preprocessor directives:
  - **Macros**: The compiler replaces the name with the actual piece of code. The **#define** directive is used to define a macro. Conversely, The **#undef** is used to undefined it.
  - **File Inclusion**: The compiler includes files (e.g., stdio, "your header") in the source code program. The **#include** directive is used to include files.
  - **Conditional Compilation**: The compiler compiles a specific portion of the program or to skip compilation. The **#ifdef** and **#endif** is used.

Linker

- Linking is the process of collecting and **combining various pieces of code and data into a single file** that can be loaded (copied) into memory and executed.
- Linking can be performed at:
  - **compile time**: when the source code is translated into machine code
  - **load time**: when the program is loaded into memory and executed by the loader
  - **even at run time**: by application programs
- On modern systems, linking is performed **automatically** by programs called **linkers**.
- Instead of organizing a large application as one monolithic source file, **we can decompose it into smaller, more manageable modules that can be modified and compiled separately**.

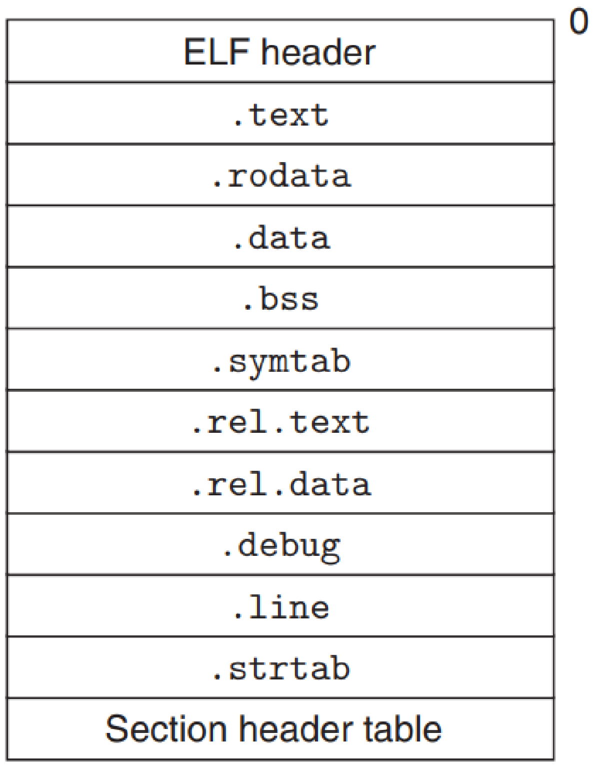
Static Linking

- Static linkers such as the Linux ***ld*** program take as input a collection of **relocatable object files** and command-line arguments and generate as output a fully linked **executable object file** that can be loaded and run.
- To build the executable, the linker must perform two main tasks:
  - Step 1. **Symbol resolution:** Object files define and reference symbols, where each symbol corresponds to a function, a global variable, or a static variable.**(ORD)**
  - Step 2. **Relocation**: Compilers and assemblers generate code and data sections that start at address 0. The linker relocates these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.



Object Files

- Object files are organized according to specific object file formats, which vary from system to system.
- Modern x86-64 Linux and Unix systems use **Executable and Linkable Format (ELF).**
- A typical ELF relocatable object file contains the following sections:
  - **.text** The machine code of the compiled program.
  - **.rodata** Read-only data such as the format strings in printf statements, and jump tables for switch statements.
  - **.data** Initialized global and static C variables.
  - **.bss** Uninitialized global and static C variables, along with any global or static variables that are **initialized to zero**.
  - **.symtab** A symbol table with information about functions and global variables that are defined and referenced in the program.



GNU Binutils

- The GNU Binutils are a collection of binary tools.
- ***readelf*** - Displays information from any ELF format object file.

```
$ gcc -g message.c hello.c -o hello
$ readelf -s hello
$ gcc -g -c message.c hello.c
$ readelf -x .data hello.o                # 522b0c0c
( little-endian in x86 )
```

- ***objdump*** - Displays information from object files.

```
$ gcc -g -c message.c hello.c
$ objdump -d hello.o
```

- ***objcopy*** - Copy and translate object files

```
$ gcc -g message.c hello.c -o hello
$ objcopy --only-keep-debug hello hello.debug
```

Symbol Resolution & Relocation

Step 1. Resolution

```
//hello.c 소스코드
#include <stdio.h>

int id = 202124114;
const char* getMessage();

void greeting() {
   printf("%s₩n", getMessage());
}

int main() {
   greeting();
}
```

global variable

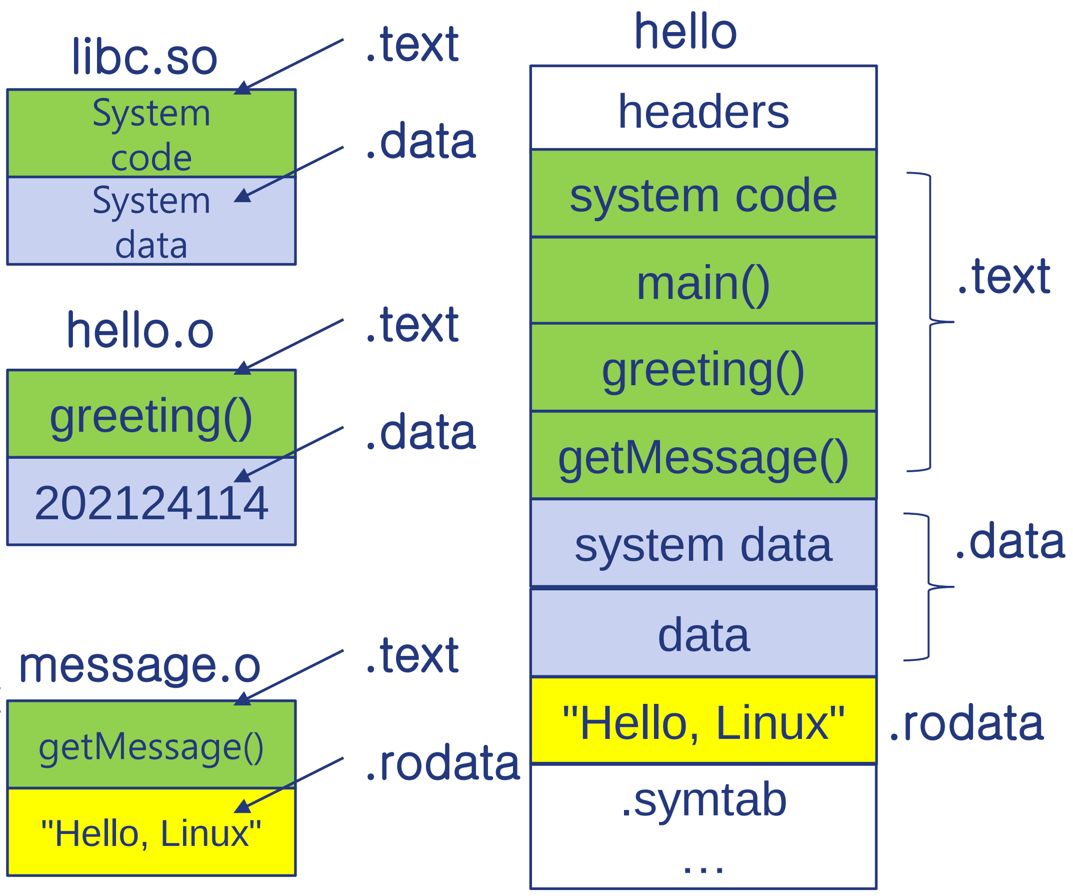
declaration

implicit extern

implicit extern

```
//message.c 소스코드
const char* getMessage() {
   return "Hello, Linux";
}
```

Step 2. Relocation

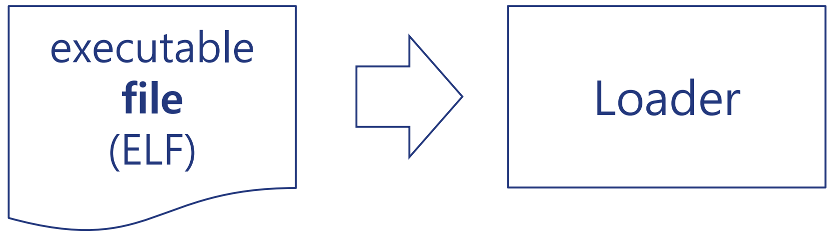


Relocatable Object Files

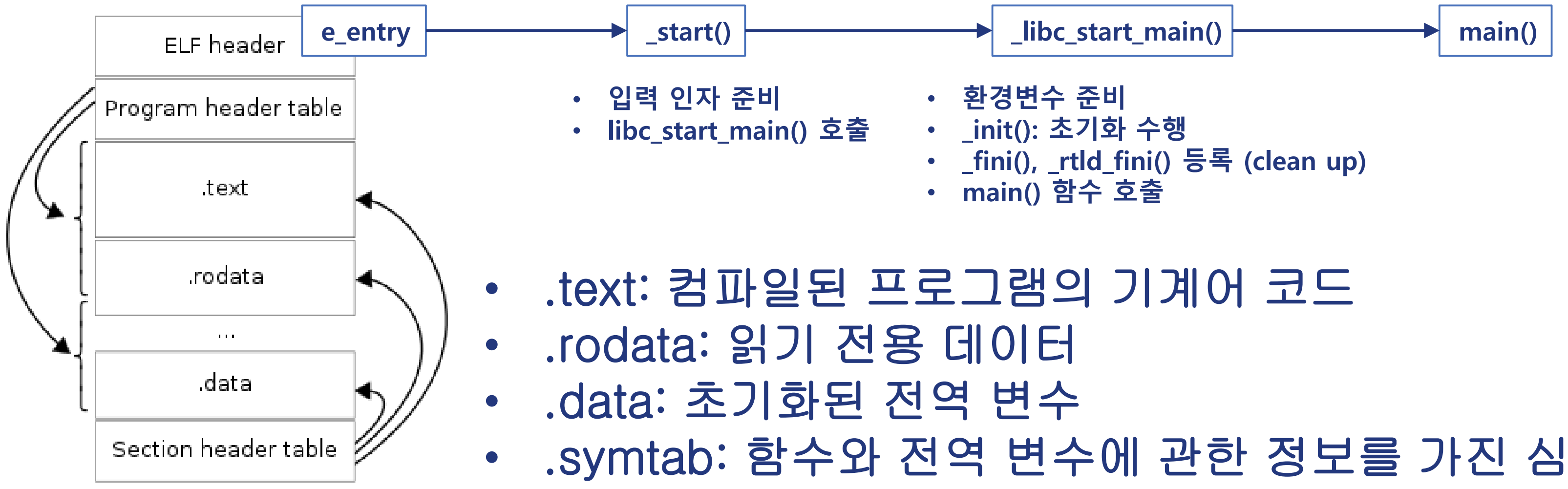
Executable Object File

Executing main() in C (Linux)

❖ Execution Process



- 실행파일에 있는 코드와 데이터를 메모리에 복사
- 제어를 프로그램의 시작으로 옮김



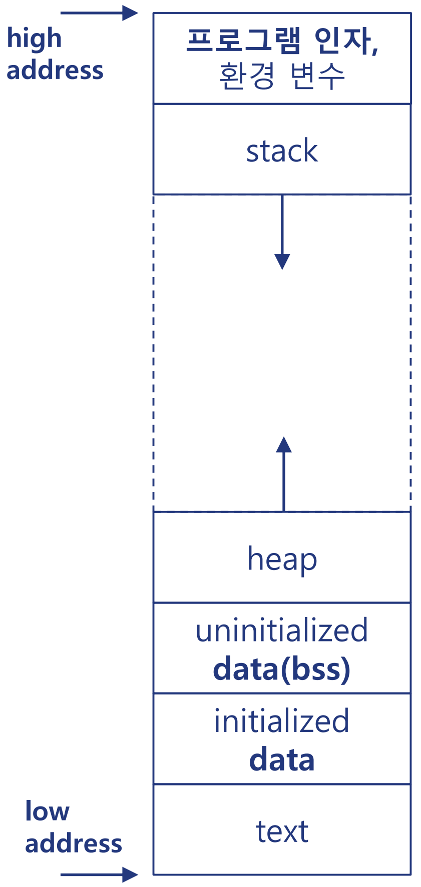
- 입력 인자 준비
- libc_start_main() 호출

- 환경변수 준비
- _init(): 초기화 수행
- _fini(), _rtld_fini() 등록 (clean up)
- main() 함수 호출

- .text: 컴파일된 프로그램의 기계어 코드
- .rodata: 읽기 전용 데이터
- .data: 초기화된 전역 변수
- .symtab: 함수와 전역 변수에 관한 정보를 가진 심볼 테이블

https://www.geeksforgeeks.org/executing-main-in-c-behind-the-scene/
https://www.linuxjournal.com/article/6463
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Memory Layout



- arg.exe I am Kim → int main(argc, argv[])
- PATH=C:₩Clion₩bin
- stack: 함수 호출 시 함수 반환 주소, 인자 값, 로컬 변수 등이 추가됨
- heap: new, delete 연산자로 관리하는 메모리 공간
- uninitialized data : 프로그래머가 초기화 하지 않은 전역변수 및 static 변수가 위치함 (실행 시 0으로 초기화)
- initialized data : 프로그래머가 초기화한 global, static, const, extern 변수가 위치함 (읽기전용과 읽기쓰기 영역으로 나눠짐)
- text (read-only) : 실행 가능한 명령어

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Packaging Commonly Used Functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.
- given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer
- Commonly Used Libraries
  - `libc.a` (the C standard library): 4.6 MB archive of 1496 object files.
  - `libm.a` (the C math library): 2 MB archive of 444 object files.

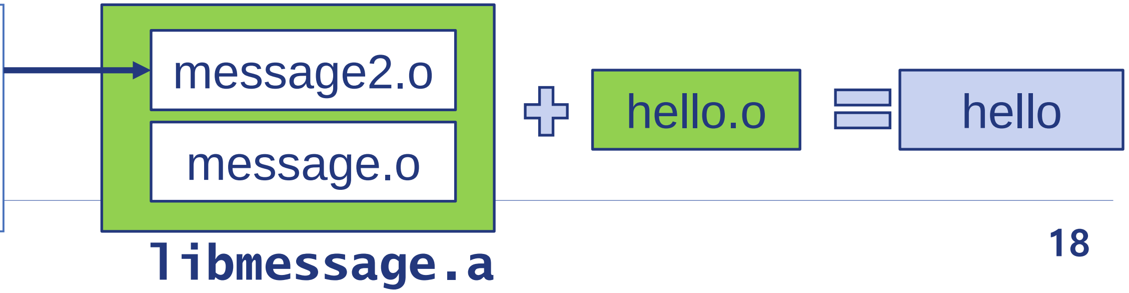
```
$ ar -t /usr/lib32/libc.a | sort
$ ar -t /usr/lib32/libm.a | sort
```

Old-fashioned Solution: Static Libraries

- Static libraries (.a archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

```
$ gcc -c message.c
$ gcc -c message2.c
$ ar rcs libmessage.a message.o message2.o
$ gcc -static hello.o ./libmessage.a -o hello
```

```
//message2.c 소스코드
const char* getMessage2() {
   return "Goodbye";
}
```



Modern Solution: Shared Libraries

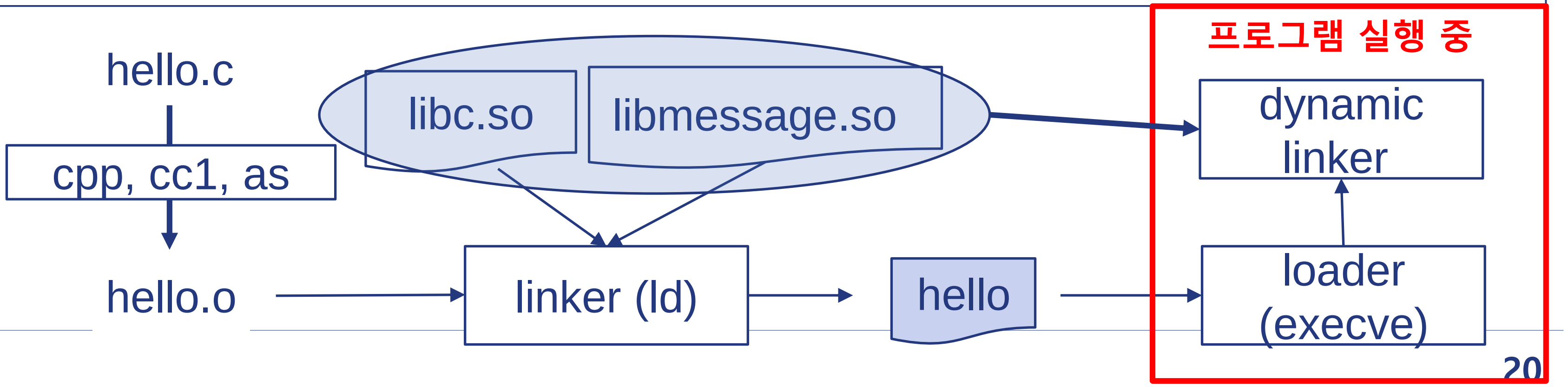
- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- Modern solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application ***dynamically***, at either ***load-time*** or ***run-time***
  - Also called: dynamic link libraries, DLLs, **`.so` files**
- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (**`ld-linux.so`**).
  - Standard C library (**`libc.so`**) usually dynamically linked.

Shared Libraries (con't)

- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.

- Shared library routines can be shared by multiple processes.

```
$ gcc -shared -fpic message.c message2.c -o libmessage.so
$ gcc hello.c -o hello ./libmessage.so
```



Frequently Used GCC Command Options

option	Description
-ansi	In C mode, this is equivalent to '-std=c90'
-std=	Determine the language standard. (e.g., c11, gnu11, or c++17)
-Wall	enables all compiler's warning messages.
-l	Link with shared libraries
-fPIC	Create position independent code
-V	Print all the executed commands
-D	Use compile time macros
-g	Generate debug symbols
-O	Improve the performance and/or code size (Optimization option)