Linking

2017 – 2018

Florent Gluck – Florent.Gluck@hesge.ch

Version 0.3

h e p i a
Haute école du paysage, d'ingénierie et d'architecture de Genève

Hes·so GENÈVE
Haute Ecole Spécialisée de Suisse occidentale

Example

main.c

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main() {
    int val = sum(array, 2);
    return val;
}
```

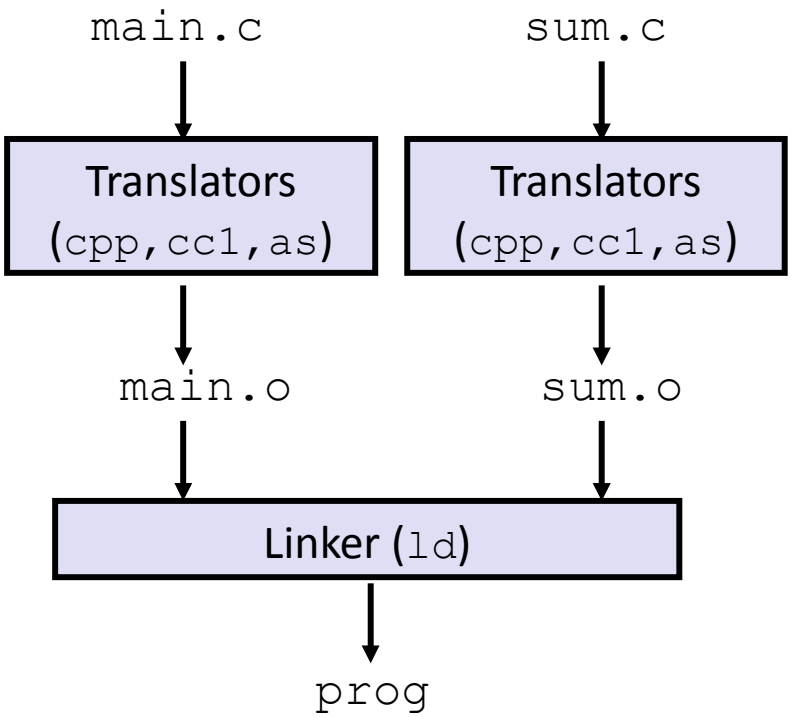
sum.c

```
int sum(int *a, int n) {
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

Linking

Programs are translated and linked using a toolchain:

```
gcc -o prog main.c sum.c
./prog
```



Sources files

Relocatable object files

Fully linked **executable** object file (contains code and data for all functions defined in main.c and sum.c)

Why linkers? (1)

Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions:
  - e.g., math library, standard C library

Why linkers? (2)

Reason 2 : Efficiency

- **Time: Separate compilation**
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- **Space: libraries**
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

What do linkers do? (1)

Step 1: symbol resolution

- Programs define and reference symbols (global variables and functions):

```
int sum(int *a, int n) {…} // define symbol sum
sum(...);                  // reference symbol sum
int *xp = &x;              // define symbol xp, reference x
```

- Symbol definitions are stored in the object file (by assembler) in the **symbol table**.
- Each entry includes name, size, and location of symbol.
- During **symbol resolution** step, the linker associates each symbol reference with exactly one symbol definition.

Three kinds of object files

- **Relocatable object file (.o file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form an executable object file.
  - Each .o file is produced from exactly one source (.c) file
- **Executable object file (a.out file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (.so file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

Three kinds of object files

compiler and assembler

- **Relocatable object file (.o file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form an executable object file.
  - Each .o file is produced from exactly one source (.c) file

- **Executable object file (a.out file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.

- **Shared object file (.so file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

Three kinds of object files

- **Relocatable object file (.o file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form an executable object file.
  - Each .o file is produced from exactly one source (.c) file

linker (ld)

- **Executable object file (a.out file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (.so file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

Executable and Linkable Format (ELF)

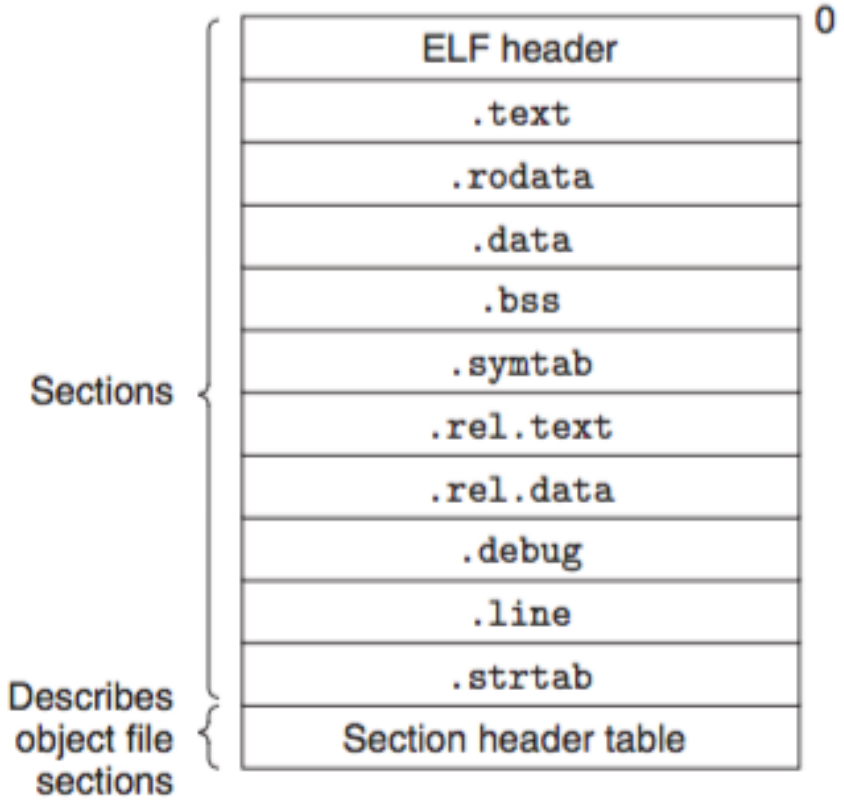
- ELF is an unified, architecture agnostic, binary file format designed to store object files content.
- Quick adoption by numerous operating systems and architectures.
- Derived from UNIX AT&T System V and chosen in 1999 to be the standard binary file format for all UNIX systems.
- One **unified** format for
  - **Relocatable** object files (.o),
  - **Executable** object files (a.out)
  - **Shared** object files (.so)
- Generic name: ELF binaries

Relocatable object files (1)

- A **relocatable** object file is basically a collection of sections.
- Each section contains a single type of information, such as: program code, read only data, symbols, etc.
- Each symbol's address is defined relatively to its section
  - A function's entry point is relative to the section where the function's code resides.

Relocatable object files (2)

Example of a **relocatable** object file:

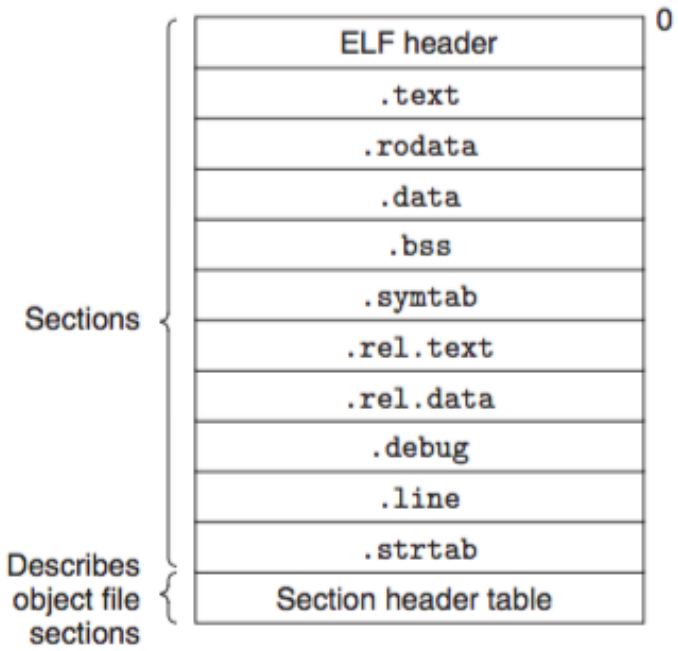


Executable object files

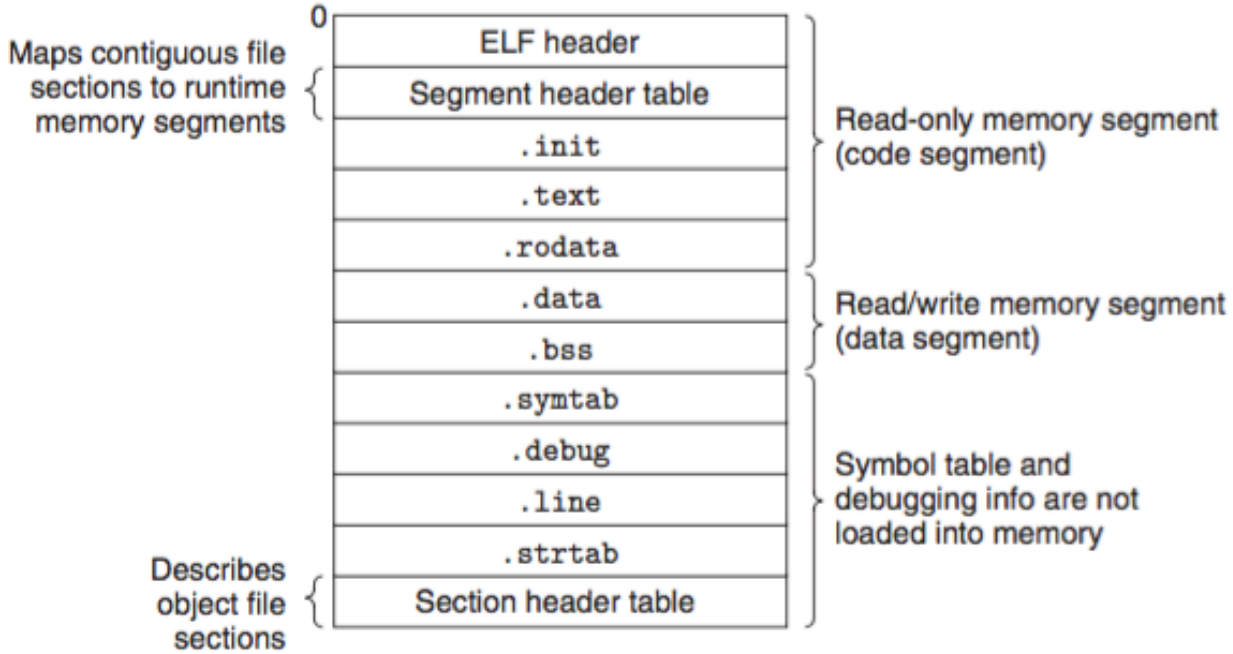
- **Very similar** to **relocatable** object files, except :
  - Program's entrypoint stored in the ELF header.
  - .text, .rodata, and .data sections **similar** to **relocatable** object files, except:
    - These sections' addresses are relocated at runtime.
- ELF format designed to easily load into RAM:
  - Contiguous chunks of files mapped to contiguous memory segments.
- The process of copying a program into RAM to be executed is called "loading".
  - Performed by the "loader" of the operating system.

Relocatable vs executable files

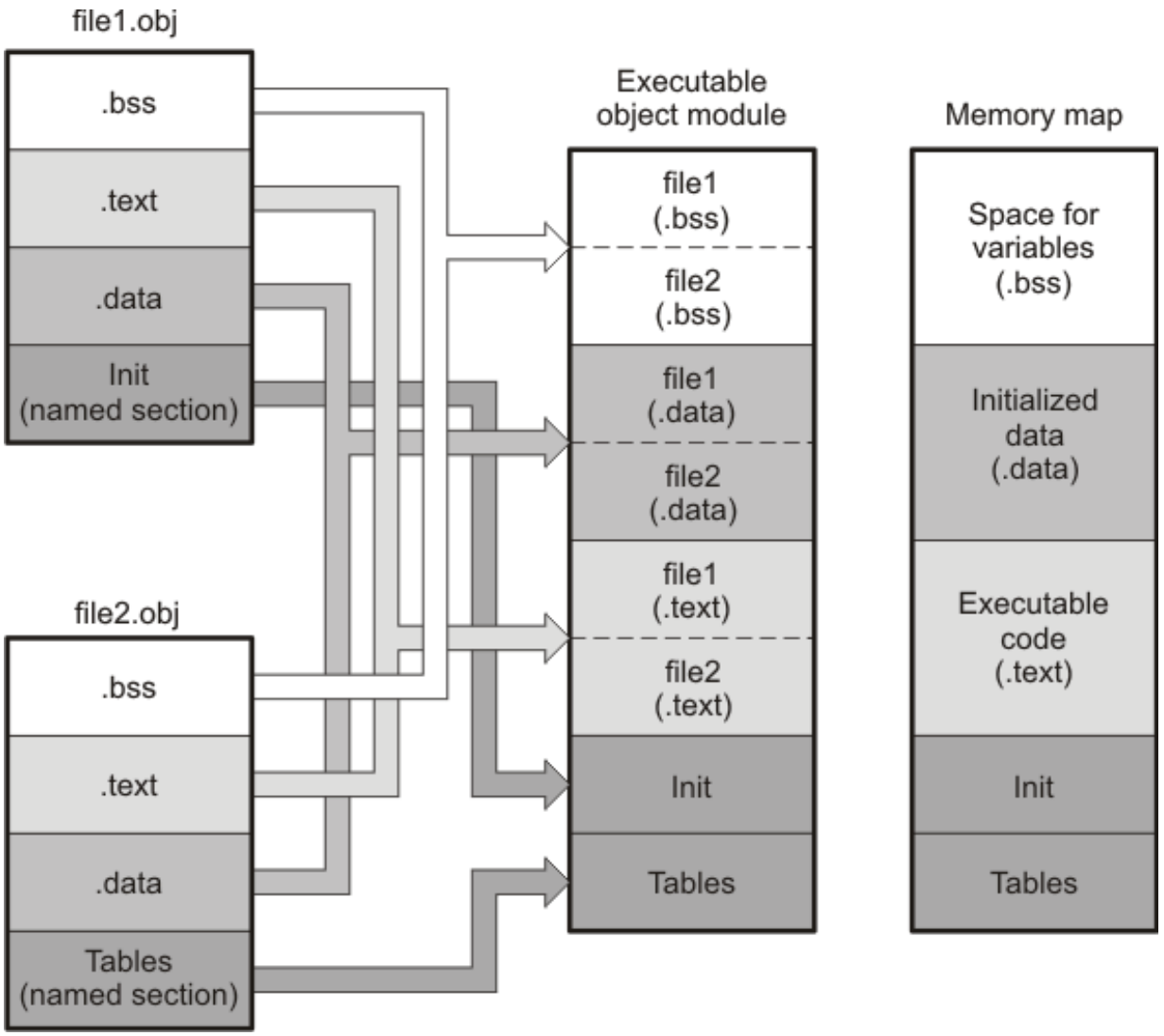
Relocatable object file:



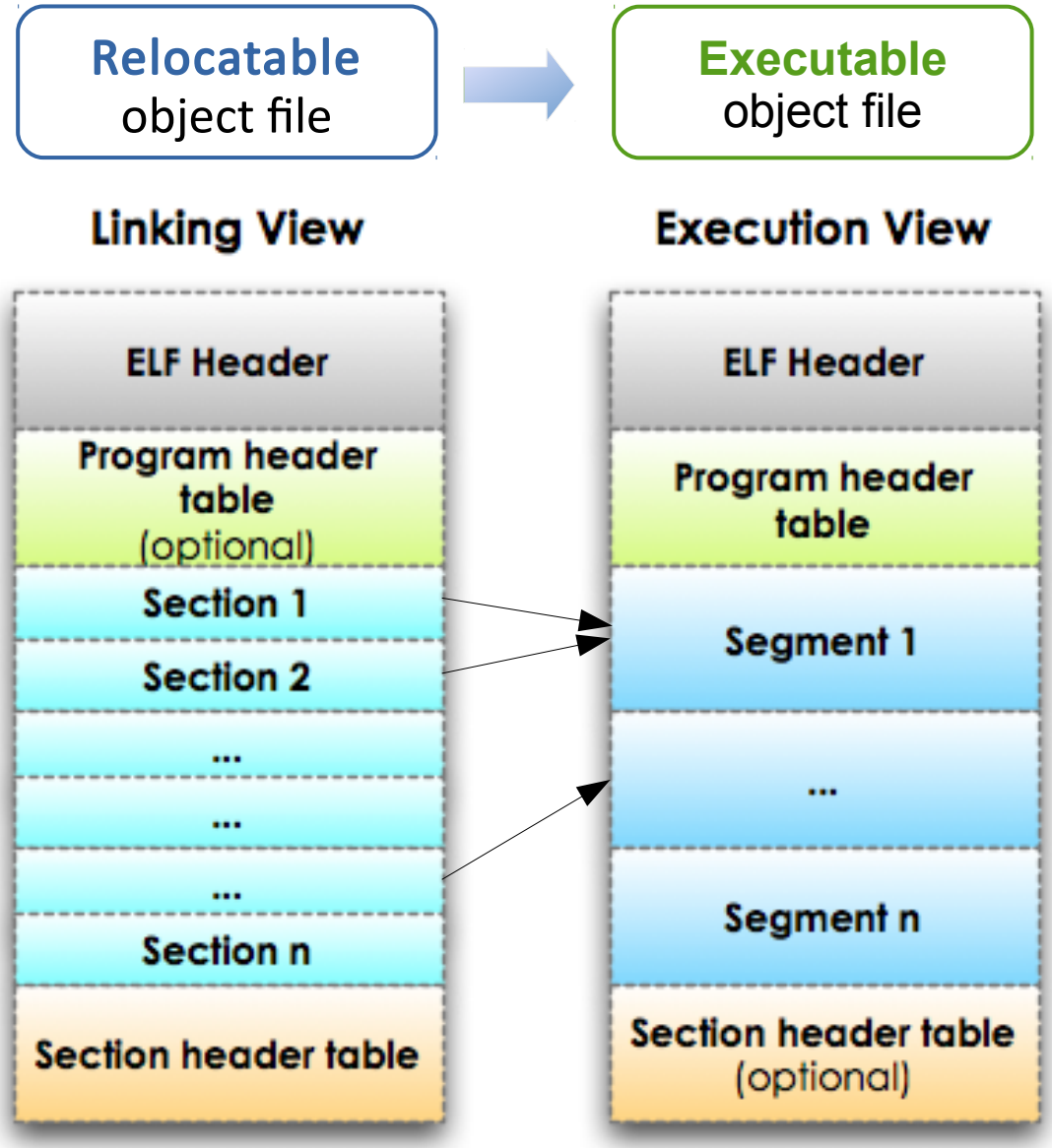
Executable object file:



From relocatable to executable (1)



From relocatable to executable (2)



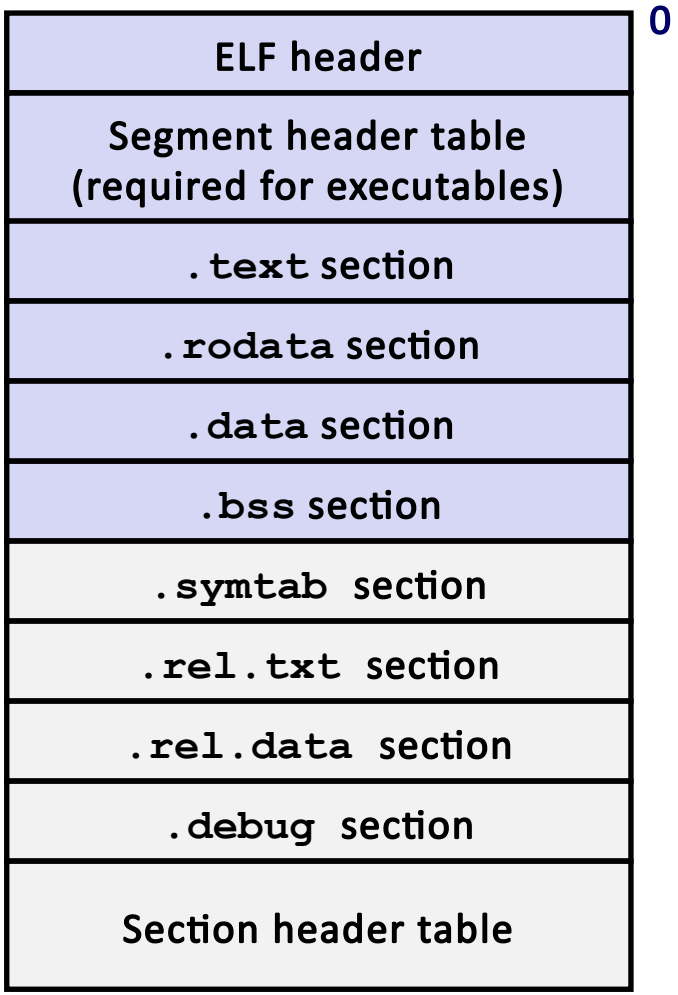
Structure of an ELF file

An ELF file has two personalities:

- **A link view**: compilers, assembler and linkers treat an ELF file as a set of logical **sections** defined by a section header table.
- **An execution view**: the loader treats an ELF file as a set of **segments** defined by a program header table.

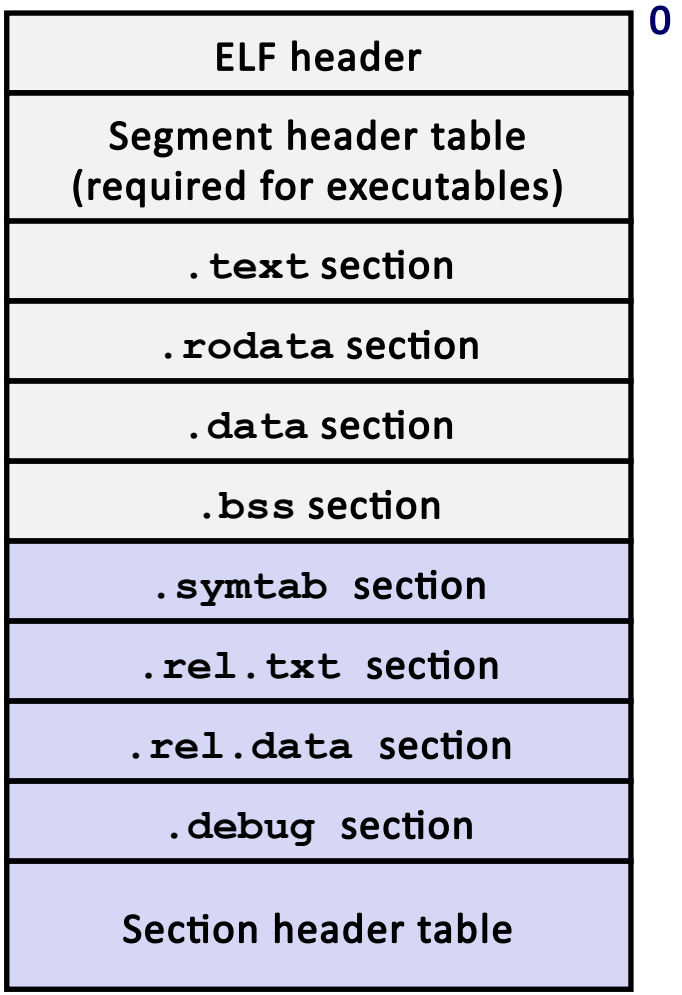
ELF file format (1)

- **Elf header**
  - word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - page size, virtual addresses memory segments (sections), segment sizes
- **.text section**
  - code
- **.rodata section**
  - read only data
- **.data section**
  - initialized global variables
- **.bss section**
  - uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - has section header but occupies no space



ELF file format (2)

- **.symtab section**
  - symbol table
  - procedure and static variable names
  - section names and locations
- **.rel.text section**
  - relocation info for .text section
  - addresses of instructions that will need to be modified in the executable
  - instructions for modifying.
- **.rel.data section**
  - relocation info for .data section
  - addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
  - info for symbolic debugging (gcc -g)
- **Section header table**
  - offsets and sizes of each section



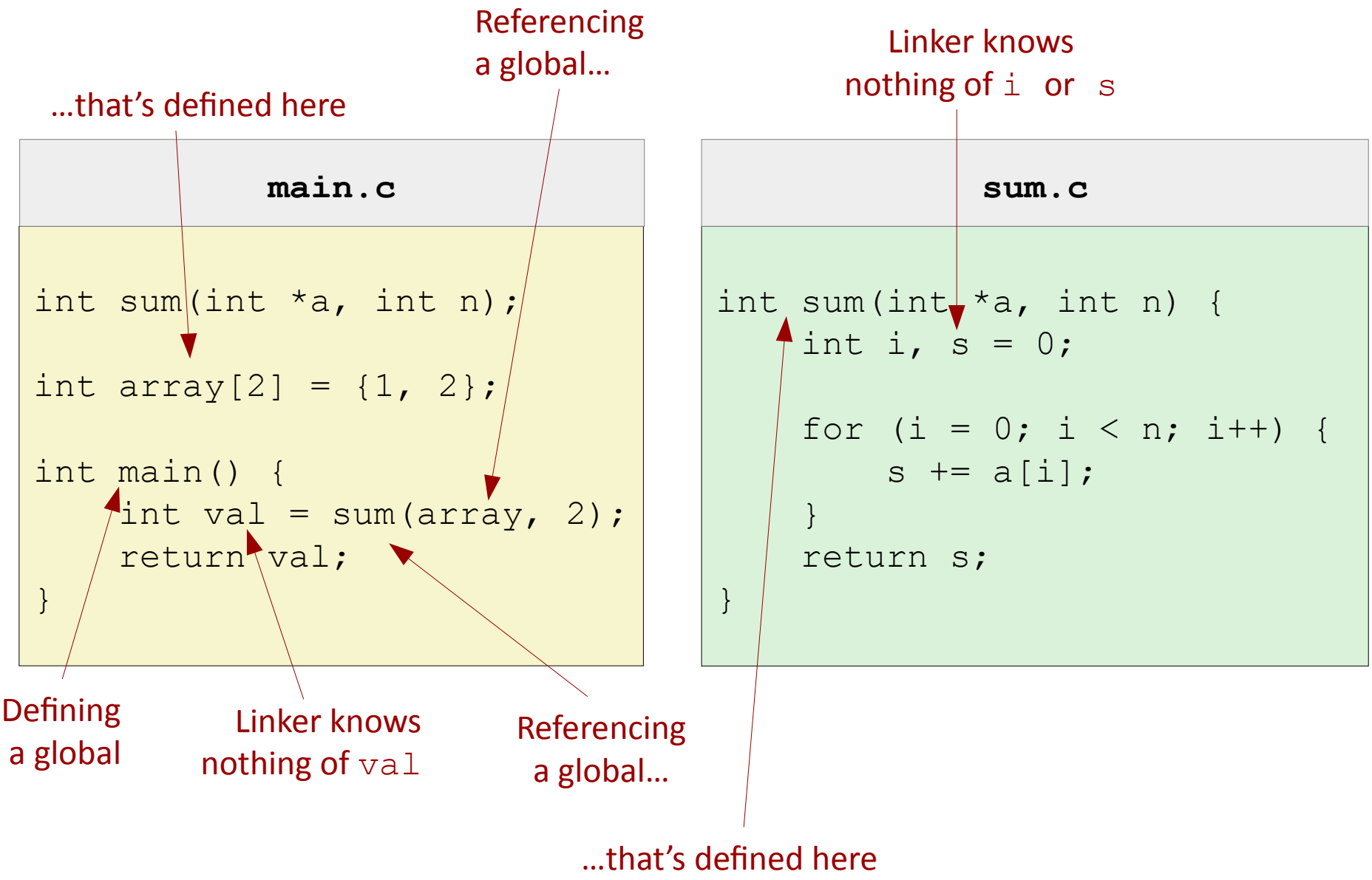
Linker symbols

- **Global symbols**
  - Symbols defined by module **m** that can be referenced by other modules.
  - E.g.: non-`static` C functions and non-`static` global variables.
- **External symbols**
  - Global symbols that are referenced by module **m** but defined by some other module.
- **Local symbols**
  - Symbols that are defined and referenced exclusively by module **m**.
  - E.g.: C functions and global variables defined with the `static` attribute.
  - Local linker symbols are not local program variables

Executable binary generation and execution

- Two steps are performed when generating an executable binary file:
  - Step 1: Symbol resolution
  - Step 2: Relocation
- When the binary file is executed, the **loader** loads it in memory (RAM)

Step 1: symbol resolution



Local symbols

- Local non-static C variables vs. local static C variables
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either .bss, or .data

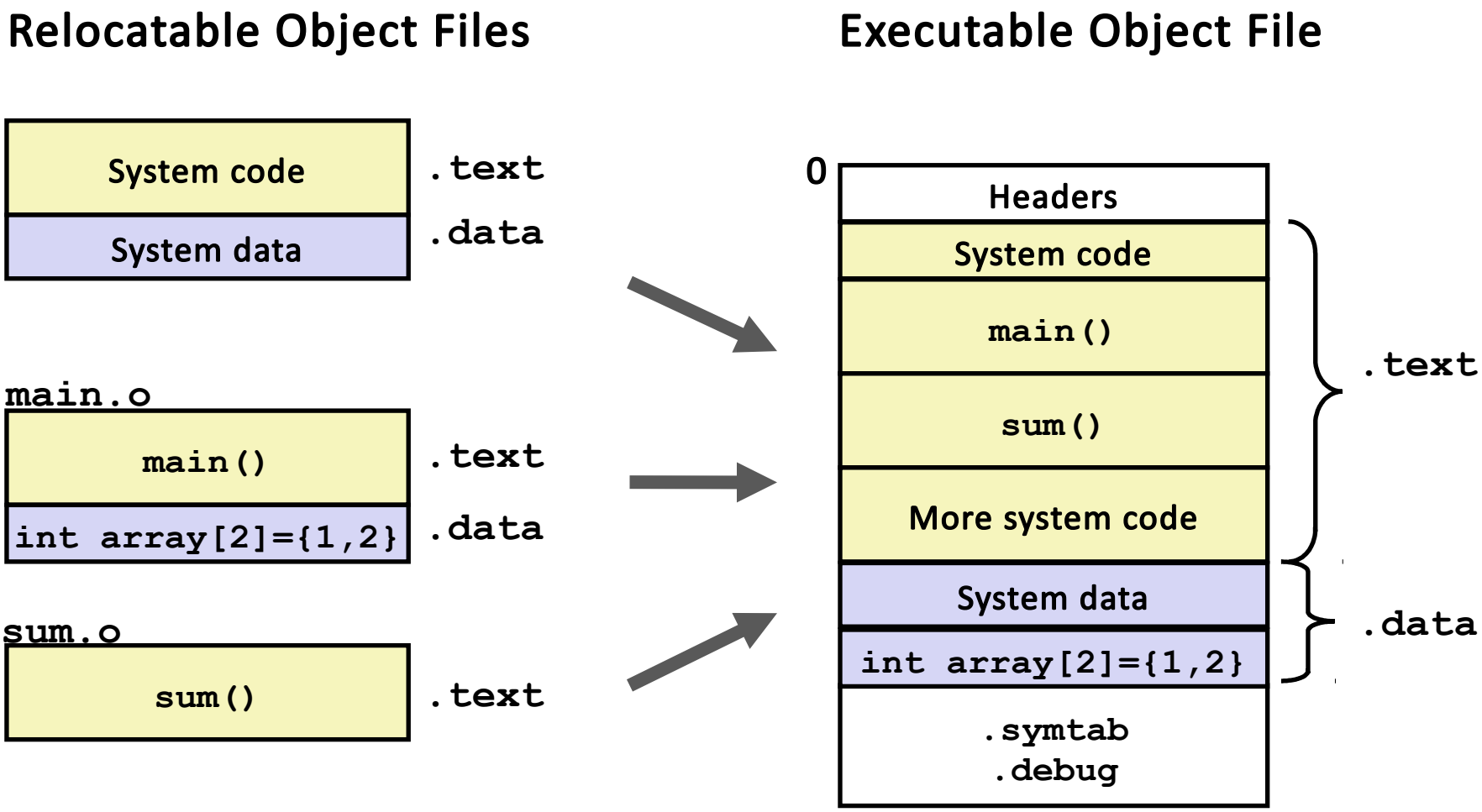
```
int f() {
    static int x = 0;
    int y = 3;
    return x+y;
}

int g() {
    static int x = 1;
    return x;
}
```

- Compiler allocates space in .data for each definition of x
- Creates local symbols in the symbol table with unique names, e.g., x.1 and x.2.

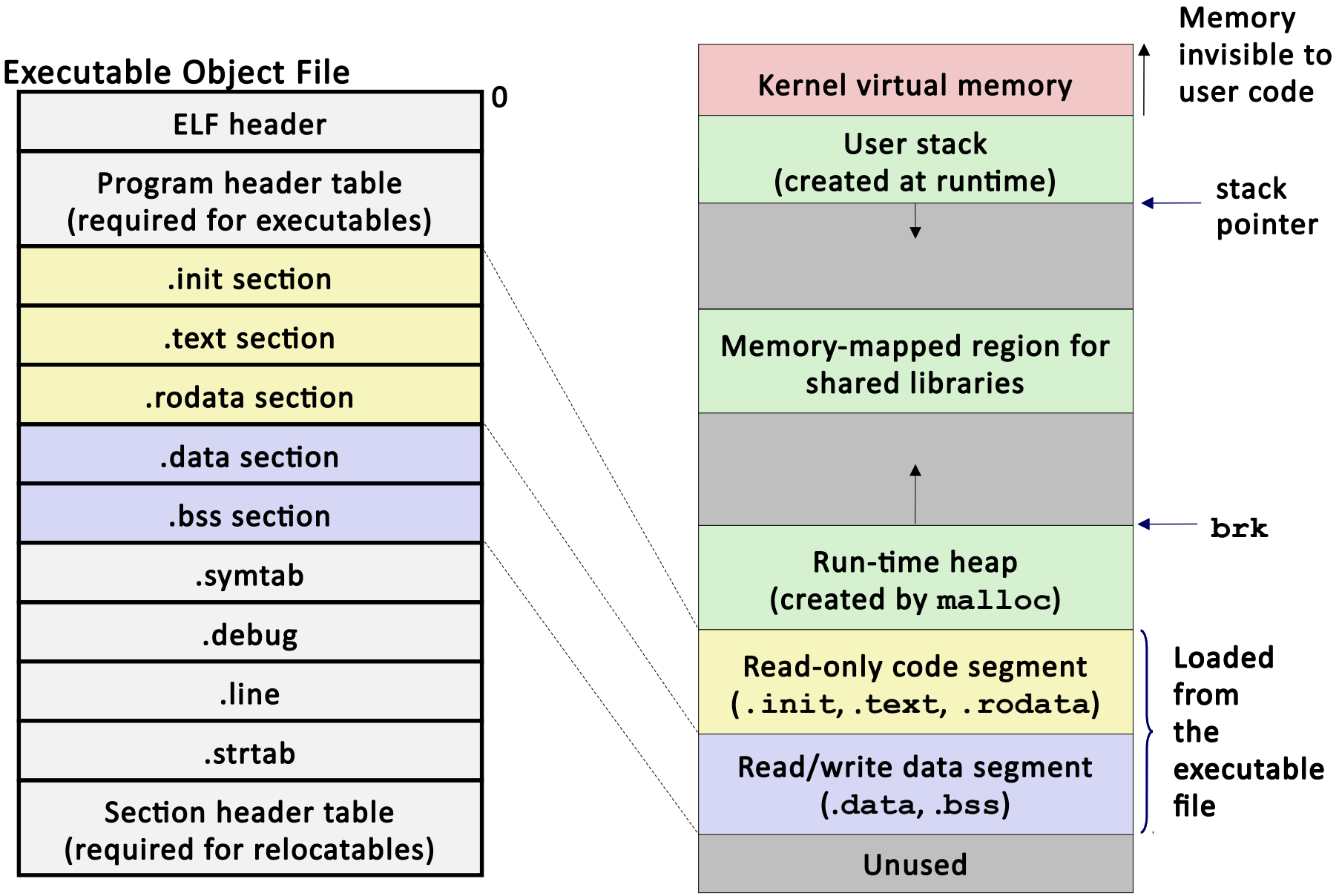
Global variables

- **Avoid** if you can!
- Otherwise
  - Use `static` if you can
  - Initialize if you define a global variable
  - Use `extern` if you reference an external global variable

Step 2: relocation



Loading executable object files



`readelf` tool

- To inspect the content of ELF files, use the `readelf` tool.
- Exemple with the `test.c` file below:

```
int A;
int B = 10;
static int C = 4;

int pipo() { return 42; }

void main() { printf("blahblah\n"); }
```

ELF header – relocatable file

```
$ gcc -c test.c
$ readelf -h test.o
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          848 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

ELF header – executable file

```
$ gcc test.c -o test
$ readelf -h test
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400430
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6728 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

Symbole table

```
$ readelf -s test.o
```

```
Symbol table '.symtab' contains 15 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS test.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 C
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    10: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM A
    11: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 B
    12: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 pipo
    13: 000000000000000b    17 FUNC    GLOBAL DEFAULT    1 main
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

- `ABS` specifies non relocatable symbols.
- `UND` specifies non resolved symbols (i.e. defined elsewhere).
- `COM` specified a non allocated object.

Tools for inspecting ELF files

- `nm` to list symbols from object files
- `readelf` to displays information about ELF files
- `objdump` to display information from object files

- To display the usage:
  - `nm --help`
  - `readelf --help`
  - `objdump --help`

Static vs dynamic linking

- **Static** linking is the result of the linker copying all library routines used in the program into the executable image.
  - Require more disk space and memory than dynamic linking
  - Faster and more portable, since it does not require the presence of the library on the system where it is run.

- **Dynamic** linking is accomplished by placing the name of a sharable library in the executable image.
  - Actual linking with the library routines occurs when the image is run when executable and library are placed in memory.
  - An advantage of dynamic linking is that multiple programs can share a single copy of the library.

Static and dynamic linking with gcc

- By default executables are dynamically linked:

```
gcc app.c -o app
```

- To link statically, specify the `-static` agument to `gcc`:

```
gcc -static app.c -o app
```

- The `file` command is useful to determine the type of linking:

```
$ file app1
test: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, ...

$ File app2
app2: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, ...
```

References

- Executable and Linkable Format
  - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- Linker scripts
  - http://wiki.osdev.org/Linker_Scripts
  - https://www.math.utah.edu/docs/info/ld_3.html
- Linux ELF Object File Format Basics
  - http://www.thegeekstuff.com/2012/07/elf-object-file-format
- The ELF Object File Format by Dissection
  - http://www.linuxjournal.com/article/1060