Introduction

An Object File

Consider the following C file `factFib.c` that contains two functions.

```
int fact(int n){
    if(n==0) return 1;
    int fact=1;
    for(int i=1; i<=n; ++i)
        fact *= i;
    return fact;
}
```

```c
int fib(int n){ // factFib.c
    if(n==0) return 0;
    if(n==1) return 1;
    int f0=0, f1=1;
    for(int i=2; i<=n; ++i){
        int temp = f0;
        f0 = f1;
        f1 += temp;
    }
    return f1;
}
```

An Object File

- This is not a complete program as the function `main()` is missing.
- We compile it to object file `factFib.o` containing machine code of x86-64.

  `$ cc -Wall -c factFib.c`
- The file type is - `$ file factFib.o`

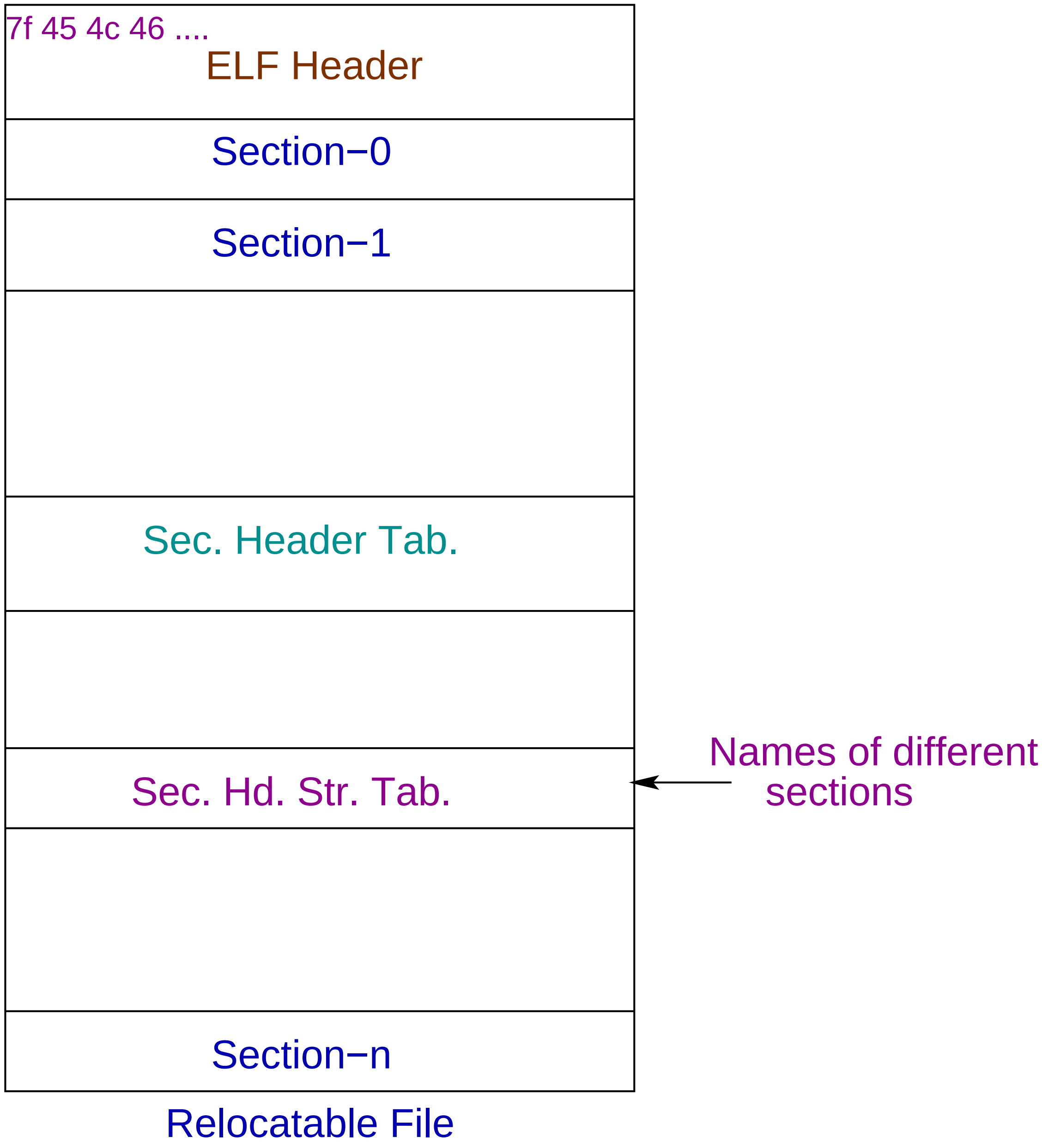
  ```
  factFib.o:  ELF 64-bit LSB
  relocatable, x86-64, ...
  ```

ELF

- ELF - executable and linking format
- There are different types of ELF files - executable files e.g. `a.out`, relocatable file e.g. `factFib.o`, core file, shared libraries.
- Here we discuss about the structure of a relocatable file.

ELF Data Structures

- The data structures of ELF file are available in `elf.h`. On my machine it is under the subdirectory `/usr/include/`.
- Every ELF file starts with a file header that gives the road-map of the file.
- A relocatable file is divided into sections and there is a section header table containing information about different sections in the file.


7f 45 4c 46 ....
ELF Header
Section−0
Section−1
Sec. Header Tab.
Sec. Hd. Str. Tab.
Names of different sections
Section−n
Relocatable File

ELF Data Structures

- An executable file is divided into segments and there is a program header table. It may or may not contain section header table.
- The program header table describes the segments of loadable code and data and other data structures e.g. that are required to link dynamically-linked library.

ELF Data Structures

- Different sections of a relocatable file contains code (text), data, and auxiliary data structures e.g. symbol table, relocation information, string tables for section names and symbol names, hash table etc.
- The ELF header is independent of hardware platform and OS. It specifies the positions of section header and/or program header tables within the file.

Reading ELF Header

- We open the relocatable file `factFib.o` and map it to the logical memory of a process so that we can read the file from memory locations.
- First let us read the ELF header and print different fields.
- The structure for the ELF header is available in `elf.h`.

ELF Header

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
                /* Magic number and other info */
  Elf64_Half    e_type;
                /* Object file type */
  Elf64_Half    e_machine;
                /* Architecture */
  Elf64_Word    e_version;
```

```
                  /* Object file version */
Elf64_Addr    e_entry;
              /* Entry point virtual address */
Elf64_Off     e_phoff;
              /* Program header table file offs
Elf64_Off     e_shoff;
              /* Section header table file offs
Elf64_Word    e_flags;
              /* Processor-specific flags */
Elf64_Half    e_ehsize;
              /* ELF header size in bytes */
```

```
    Elf64_Half      e_phentsize;
                    /* Program header table entry siz
    Elf64_Half      e_phnum;
                    /* Program header table entry cou
    Elf64_Half      e_shentsize;
                    /* Section header table entry siz
    Elf64_Half      e_shnum;
                    /* Section header table entry cou
    Elf64_Half      e_shstrndx;
                    /* Section header string table in
} Elf64_Ehdr;
```

ELF Header

```
/*
 printELFheader.c++
 $ ./a.out factFib.o
 $ ./a.out factFib   // executable file
*/
// Header files
void printIdent(unsigned char *cp){
     cout << "ELF identification: ";
     for(int i=0; i<EI_NIDENT; ++i)
```

```
            cout << hex << (int)cp[i] << "  ";
        cout << endl;
}


int main(int ac, char *av[]){
    int fd, size;
    Elf64_Ehdr *elfhP;

    if(ac < 2){
      cerr << "Object file name not specified" <<
      return 0;
```

```
    }

    fd = open(av[1], O_RDONLY);
    size = sysconf(_SC_PAGE_SIZE);
    elfhP = (Elf64_Ehdr *)mmap(0, size,
            PROT_READ, MAP_PRIVATE, fd, 0);

//  Printing ELF Identification
    printIdent(elfhP->e_ident);

//  Printing other fields
```

```
    cout << "File type: " << dec
         << elfhP->e_type << endl;

//  ... Printing other fields
    close(fd);
    return 0;
}
```

ELF Header of `factFib.o`

```
ELF identification: 7f  45  4c  46  2  1  1  0
                    0   0   0   0   0  0  0  0
File type: 1
Machine type: 62
VA Entry Point: 0x0
Program header file offset: 0
Section header file offset: 752
ELF Header size: 64
```

ELF Header of `factFib.o`

```
Program header entry size: 0
Program header entry count: 0
Section header entry size: 64
Section header entry count: 11
String table header index: 10
```

Different Fields of e_ident

- The e_ident is 16 byte long and identifies an ELF file.
  - First 4-bytes (0-3): 0x7fELF.
  - Byte 4: 2 is 64 bit objects.
  - Byte 5: 1 is LSB or little-endian encoding.
  - Byte 6: 1 is current version.
  - Byte 7: 0 System V ABI etc.

Other Header Fields in `factFib.o`

- `e_type`: object file type, 1 is relocatable file.
- `e_machine`: machine type, 62 is AMD x86-64 arch.
- `e_shoff`: file offset of the section header table.
- `e_ehsize`: size of the ELF header.
- `e_shentsize`: size of each entry of the section header table.

- `e_shnum`: number of entries in the section header table.
- `e_shstrndx`: index of the section header for the string table.
- The fields like `e_entry`, `e_phoff`, `e_phentsize`, `e_phnum` are not meaningful in a relocatable file.

Generation of Relocatable and Loadable Files

```
cc -Wall -c factFib.c ⇒ factFib.o
cc -Wall factFib.o mainFactFib.c -o factFib
⇒ factFib
```

ELF Header of `factFib`

```
ELF identification: 7f 45 4c 46 2 1 1 0
                    0  0  0  0  0 0 0 0
File type: 3
Machine type: 62
VA Entry Point: 0x610
Program header file offset: 64
Section header file offset: 6648
ELF Header size: 64
```

ELF Header of `factFib`

```
Program header entry size: 56
Program header entry count: 9
Section header entry size: 64
Section header entry count: 29
String table header index: 28
```

Other Header Fields in `factFib`

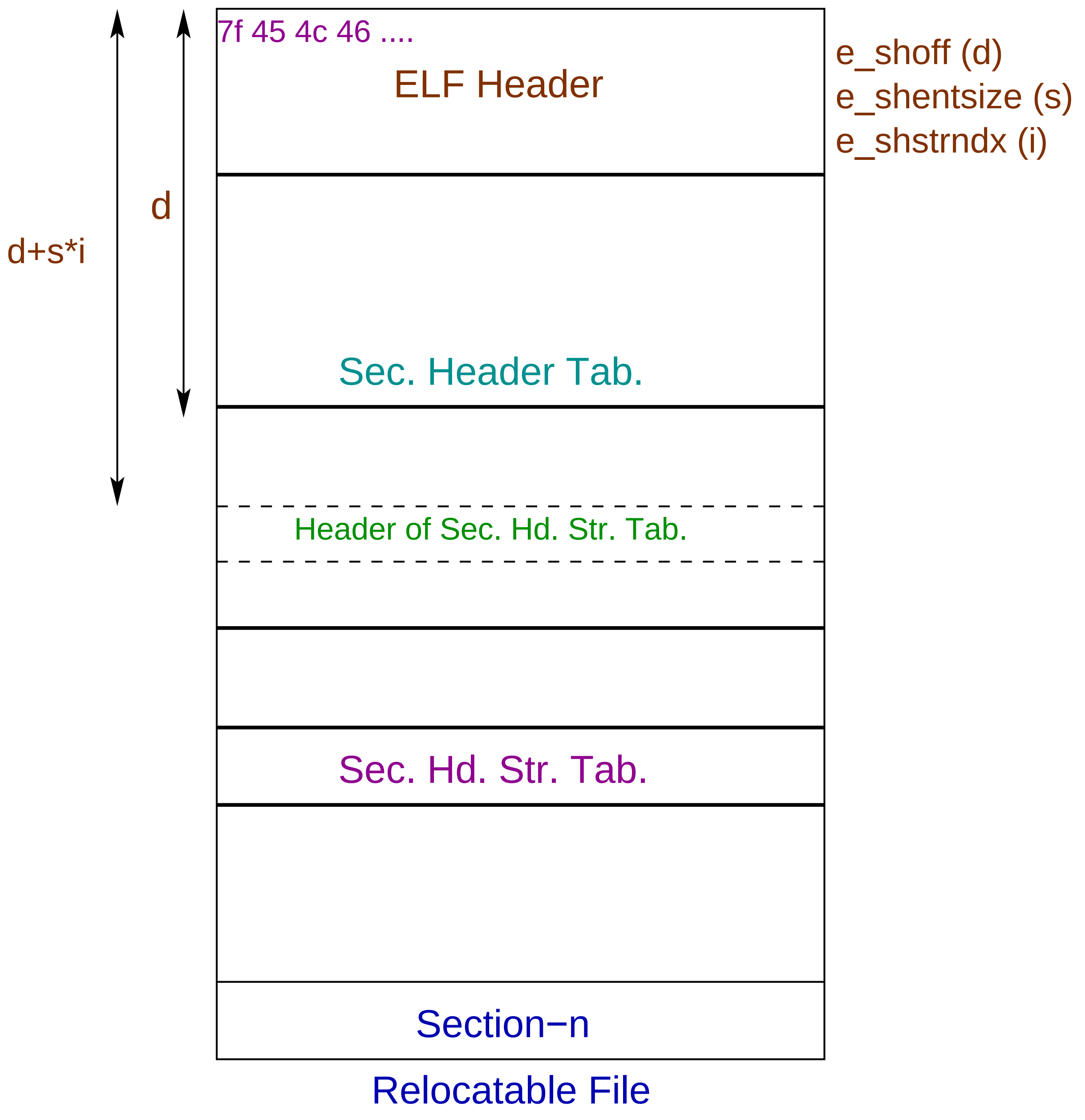
- `e_entry`: entry point (logical address) to the program. It is the address of `_start`: `0x610` (`$ objdump -d factFib | less`).
- `e_phoff`: offset of the program header table (PHT).
- `e_phentsize`: size of each entry of PHT.
- `e_phnum`: number of entries of PHT.

Section Header Table

- Information about different sections are stored in the section header table.
- Names of different sections are stored in a string table known as section header (name) string table.
- This string table itself is a section and it has a header in the section header table.

Section Name String Table: `.shstrtab`

- The ELF header contains three pieces of information; file offset of the section header table (`e_shoff`), size of each section header entry (`e_shentsize`), and the index of the section header corresponding to its string table (section names) (`e_shstrndx`).
- The file offset of the section header corresponding to the string table is $e\_shoff + e\_shstrndx \times e\_shentsize$.


7f 45 4c 46 ....
ELF Header
e_shoff (d)
e_shentsize (s)
e_shstrndx (i)
d
d+s*i
Sec. Header Tab.
Header of Sec. Hd. Str. Tab.
Sec. Hd. Str. Tab.
Section−n
Relocatable File

Structure of Section Header Table Entry

```
typedef struct
{
  Elf64_Word    sh_name;
                /* Section name (string tbl index
  Elf64_Word    sh_type;
                /* Section type */
  Elf64_Xword   sh_flags;
                /* Section flags */
  Elf64_Addr    sh_addr;
```

```
                 /* Section virtual addr at execut
Elf64_Off        sh_offset;
                 /* Section file offset */
Elf64_Xword      sh_size;
                 /* Section size in bytes */
Elf64_Word       sh_link;
                 /* Link to another section */
Elf64_Word       sh_info;
                 /* Additional section information
Elf64_Xword      sh_addralign;
                 /* Section alignment */
```

```
    Elf64_Xword   sh_entsize;
                  /* Entry size if section holds ta
} Elf64_Shdr;
```

A Few Fields of Section Header Table

- `sh_name` specifies the name of the section. The actual name is not stored here. All section names are stored in the section header string table. This field specifies the position (offset) in the string table where the name starts.
- `sh_offset` specifies the file offset of the section.

A Few Fields of Section Header Table

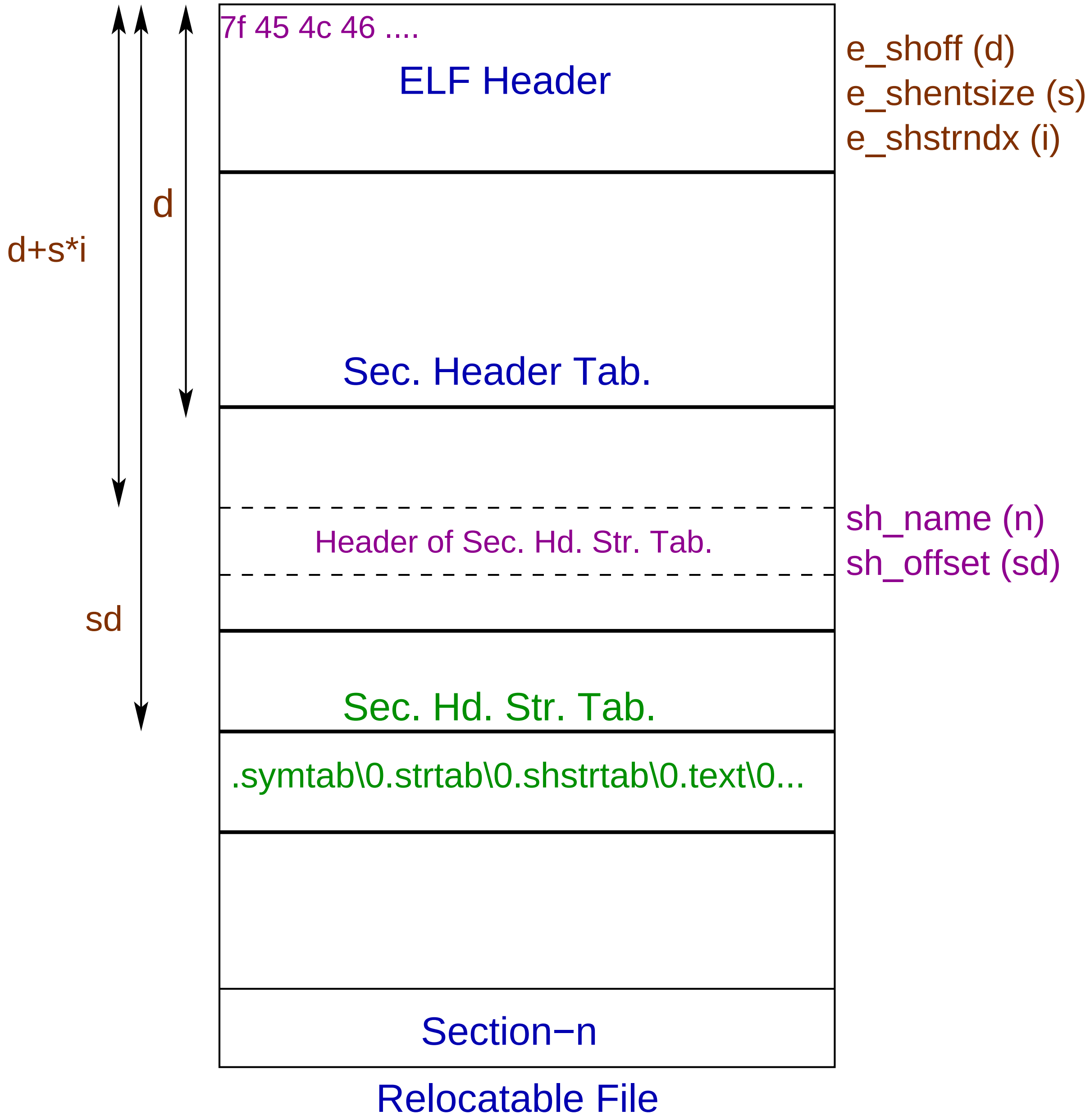
- `sh_type` specifies the type of the section e.g. program code or data, string table, relocation entries etc.
- `sh_size` stores the size of the section in bytes.
- `sh_link` section header index of a related section.

Section Header String Table: Section Header Data

We can read the section header entry of the sec head string table using the file offset formula e_shoff + e_shstrndx × e_shentsize and get the following data. Code: `printSecHdStrTab.c++`

```
Section Name: 17
Section Type: 3
File Offset of Section: 664
Section Size: 84
```

The section type 3 means a string table.


7f 45 4c 46 ....
ELF Header
e_shoff (d)
e_shentsize (s)
e_shstrndx (i)
d
d+s*i
Sec. Header Tab.
sh_name (n)
sh_offset (sd)
Header of Sec. Hd. Str. Tab.
sd
Sec. Hd. Str. Tab.
.symtab\0.strtab\0.shstrtab\0.text\0...
Section−n
Relocatable File

Section Names from `.shstrtab`

- Once we know the file offset and size of the section header (name) string table, we can print the section names.
- The section names are stored sequentially as strings in the string table.
- The offset of the first character of a section name from the beginning of the section (string table) is stored in the field `sh_name` of the section header.

Section Names from `.shstrtab`

We use the following function to print section names. Code: `printSecNames.c++`

```
void printSectionNames(char *sP, int size){
     for(int i=1; i<size; ++i){
         if(sP[i] == '\0') cout << endl;
         cout << sP[i];
     }
}
```

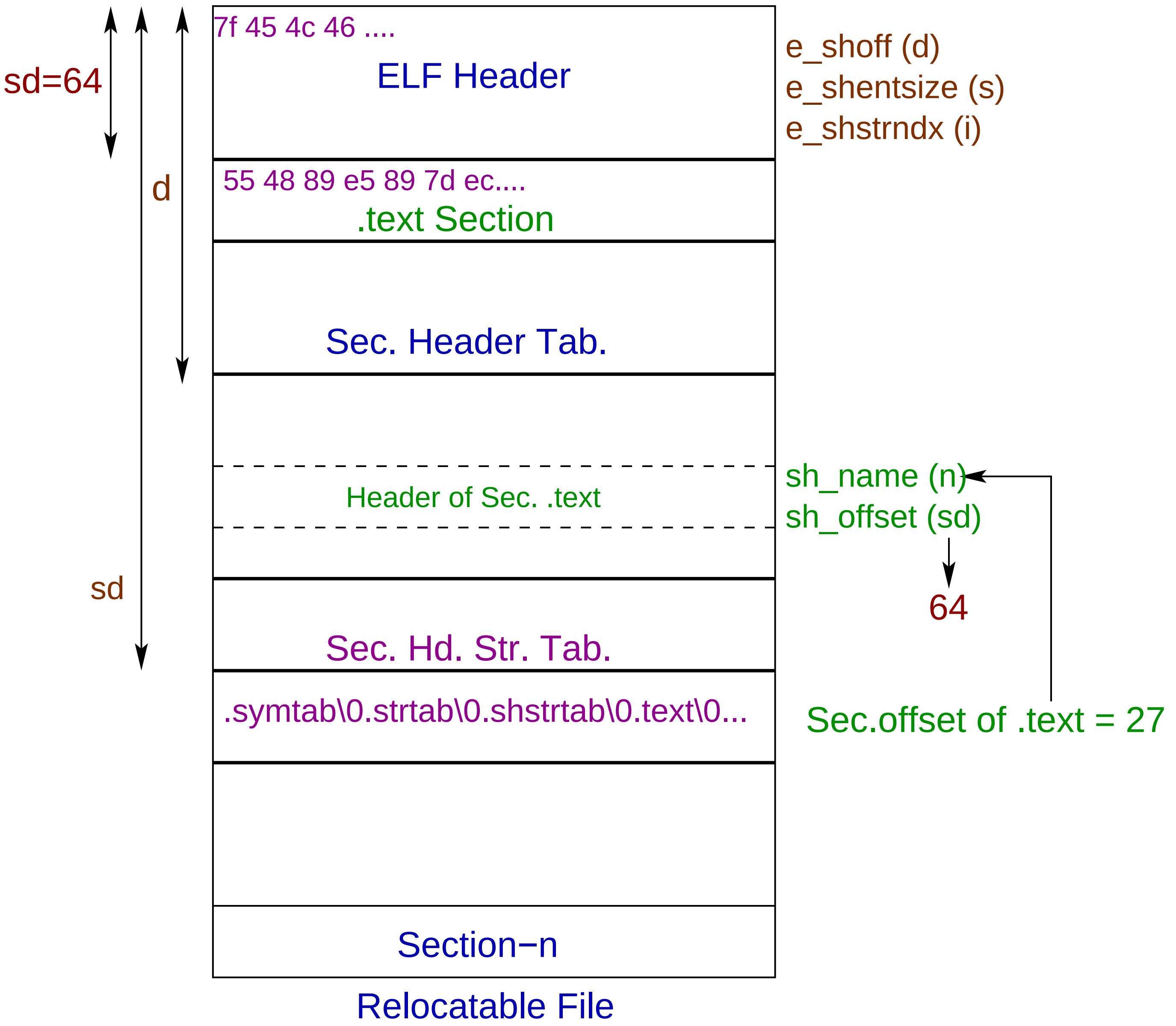
Section Names of `factFib.o`

```
$ a.out factFib.o
Section type: 3
Section names of the object file:
.symtab
.strtab
.shstrtab
.text
.data
.bss
```

```
.comment
.note.GNU-stack
.rela.eh_frame
```

Finding the Section Header of `.text` Section

- We can find out the offset of the section name `.text` from the section header (name) string table.
- This offset value is stored in the field `sh_name` of the section header entry corresponding to `.text`.
- We search for this entry and print important fields. Code: `printSecHdText.c++`


7f 45 4c 46 ....
ELF Header
e_shoff (d)
e_shentsize (s)
e_shstrndx (i)
sd=64
d
55 48 89 e5 89 7d ec....
.text Section
Sec. Header Tab.
Header of Sec. .text
sh_name (n)
sh_offset (sd)
64
sd
Sec. Hd. Str. Tab.
.symtab\0.strtab\0.shstrtab\0.text\0...
Sec.offset of .text = 27
Section−n
Relocatable File

Finding the Section Header of .text Section

```
$ a.out factFib.o
Offset of .text in string table: 27
Section Name of .text offset: 27
Section Type: 1
File Offset of Section: 64
Section Size: 154
```

Section type 1 means program data. The machine code or text starts from file offset 64.

An Experiment with `.text`

- The `.text` of `factFib.o` starts either with the function `int fact(int)` or with the function `int fib(int)`.
- If a function is called using a function pointer pointing to the logical address of `.text`, with parameters, the first function of `.text` will be invoked.

An Experiment with .text

```
int (*funP)(int);
// other code
cout << "Enter a +ve integer: " ;
cin >> n ;
funP = (int (*)(int))textP;  // Pointer to .text
fact = (*funP)(n);
cout << "fact(" << dec << n << ") = " << fact <<
```

An Experiment with `.text`

```
$ a.out factFib.o
Enter a +ve integer: 0
fact(0) = 1
```

Note

- It will not work for a function that calls other function due to address relocation problem.
- Replace the factorial function by a recursive one (`factFib1.c`).

```
int fact(int n){ // factFib1.c
    if(n==0) return 1;
    return n*fact(n-1);
}
```

Note: `$ a.out factFib1.o`

```
$ ./a.out factFib1.o
Enter a +ve integer: 5
fact(5) = 20
$ ./a.out factFib1.o
Enter a +ve integer: 6
fact(6) = 30
$ ./a.out factFib1.o
Enter a +ve integer: 7
fact(7) = 42
```

Note

- The function is returning wrong value, but it is doing it in a consistent way - returns $n(n-1)$ where n is the parameter.
- We look at the code of `int fact(int n)` using `$ objdump -d factFib1.o`.

Code of `fact()` in `factFib1.o`

Comments are mine -

```
0000000000000000 <fact>:
 0:   55                     push   %rbp
 1:   48 89 e5               mov    %rsp,%rbp
 4:   48 83 ec 10            sub    $0x10,%rsp
 8:   89 7d fc               mov    %edi,-0x4(%rbp)
 b:   83 7d fc 00            cmpl   $0x0,-0x4(%rbp)
 f:   75 07                  jne    18 <fact+0x18>
11:   b8 01 00 00 00         mov    $0x1,%eax
```

```
16:  eb 11                 jmp    29 <fact+0x29>
18:  8b 45 fc              mov    -0x4(%rbp),%eax
1b:  83 e8 01              sub    $0x1,%eax
1e:  89 c7                 mov    %eax,%edi
20:  e8 00 00 00 00        callq  25 <fact+0x25>
25:  0f af 45 fc           imul   -0x4(%rbp),%eax
29:  c9                    leaveq
2a:  c3                    retq
```

There is relocation requirement at 0x20, the recursive call.

Code of `fact()` in `a.out`

Comments are mine -

```
0000000000007b9 <fact>:
 7b9:   55                push   %rbp
 7ba:   48 89 e5          mov    %rsp,%rbp
 7bd:   48 83 ec 10       sub    $0x10,%rsp
 7c1:   89 7d fc          mov    %edi,-0x4(%rbp)
 7c4:   83 7d fc 00       cmpl   $0x0,-0x4(%rbp)
 7c8:   75 07             jne    7d1 <fact+0x18>
 7ca:   b8 01 00 00 00    mov    $0x1,%eax
```

```
7cf:  eb 11            jmp    7e2 <fact+0x29>
7d1:  8b 45 fc         mov    -0x4(%rbp),%eax
7d4:  83 e8 01         sub    $0x1,%eax
7d7:  89 c7            mov    %eax,%edi
7d9:  e8 db ff ff ff   callq  7b9 <fact>
7de:  0f af 45 fc      imul   -0x4(%rbp),%eax
7e2:  c9               leaveq
7e3:  c3               retq
```

Section Names of `factFib1.o`

There is a new section in `factFib1.o`. We may talk about that afterward.

```
$ a.out factFib1.o
Section type: 3
Section names of the object file:
.symtab
.strtab
.shstrtab
.rela.text
```

```
.data
.bss
.comment
.note.GNU-stack
.rela.eh_frame
```

Finding a Symbol

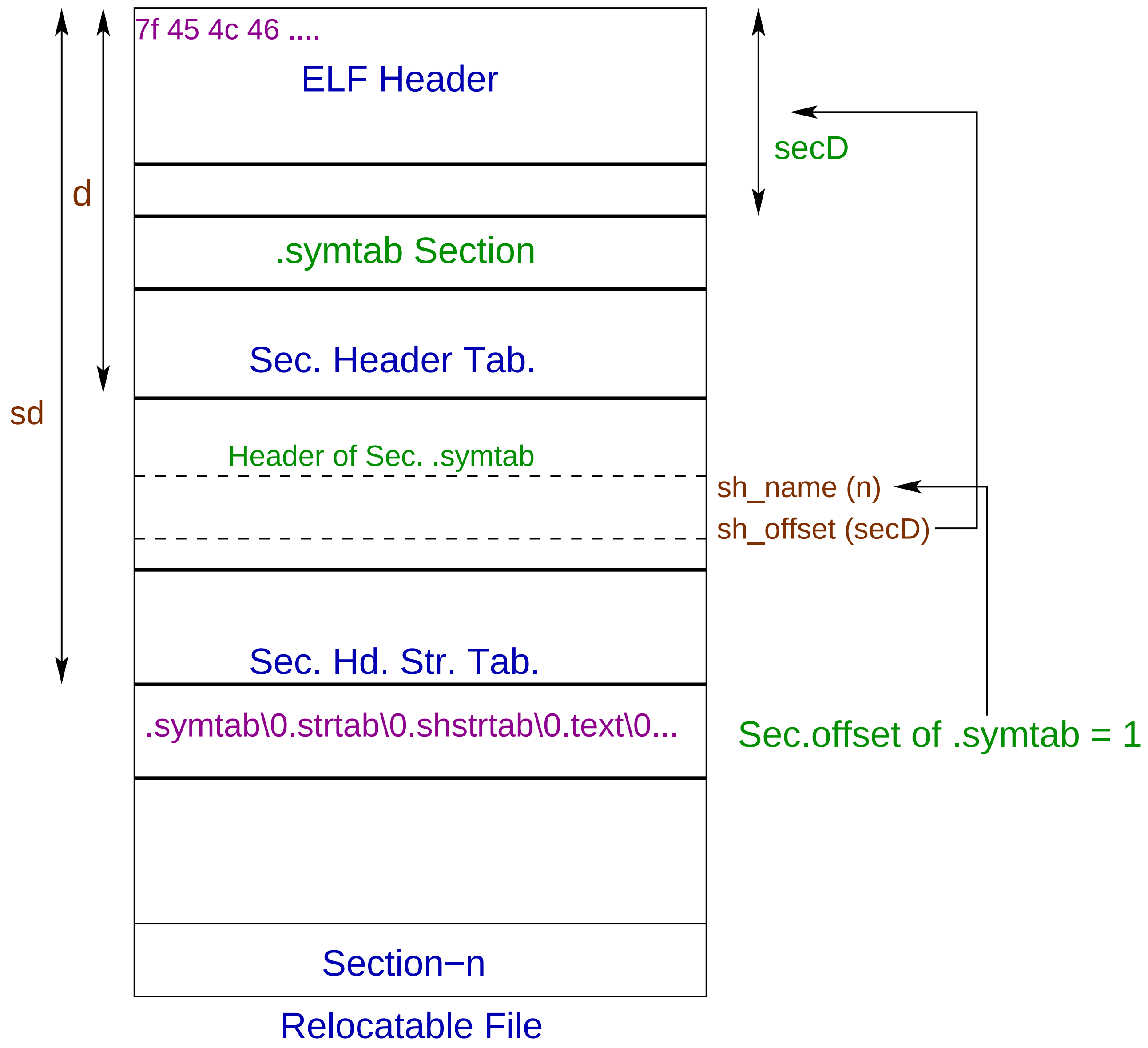
- The earlier method can invoke only the first function of `.text`. An obvious question is how can we invoke any other function present in the `.text` section of the mapped relocatable file.
- In other words, how to find the ELF file displacement of a global symbol present in the file.

Symbol Table

There is a symbol table (`.symtab`) section in an ELF file, where each entry is of following type.

```
typedef struct
{
  Elf64_Word    st_name;
     /* Symbol name (string tbl index) */
  unsigned char st_info;
     /* Symbol type and binding */
  unsigned char st_other;
```

```
    /* Symbol visibility */
  Elf64_Section st_shndx;
    /* Section index */
  Elf64_Addr    st_value;
    /* Symbol value */
  Elf64_Xword   st_size;
    /* Symbol size */
} Elf64_Sym;
```

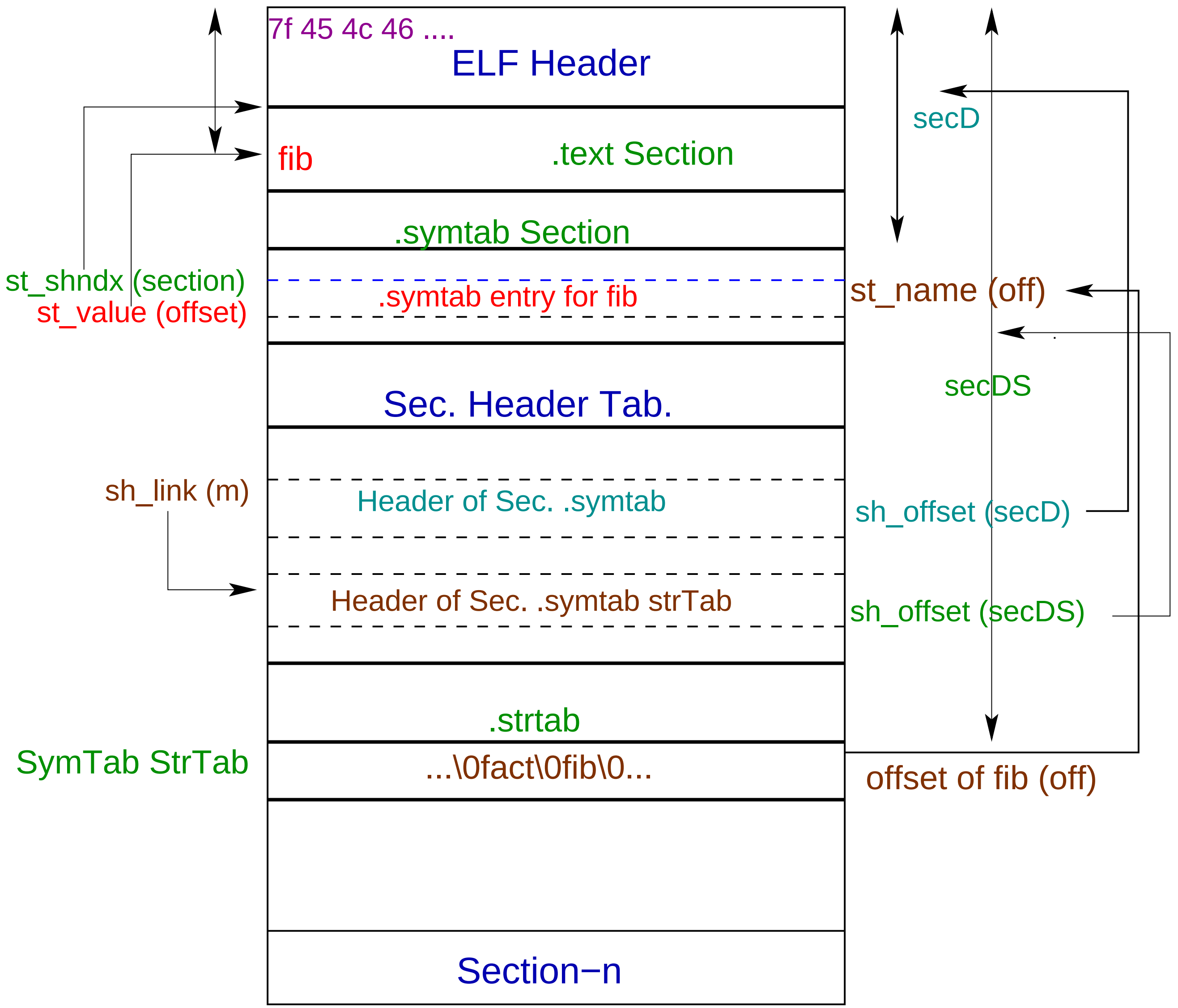

7f 45 4c 46 ....
ELF Header
.symtab Section
Sec. Header Tab.
Header of Sec. .symtab
Sec. Hd. Str. Tab.
.symtab\0.strtab\0.shstrtab\0.text\0...
Section−n
Relocatable File
d
sd
secD
sh_name (n)
sh_offset (secD)
Sec.offset of .symtab = 1

String Table of a Symbol Table

- Each symbol table has its string table (`.strtab`) containing the names of symbols present in the symbol table.
- We can find out the section header of the symbol table (`.symtab`) exactly the way we found out the section header of `.text`.

String Table of a Symbol Table

- The `sh_link` field of section header of `.symtab` gives the section header index of its string table.
- So the section header of this string table can be located, its file offset can be found and the desired string can be searched in it.



Relocatable File

String Table to Symbol Table

- We search for the symbol table entry corresponding to the symbol name (`st_name`) using the offset of the symbol in the string table.
- Once the symbol table entry is obtained, we get the section where it belongs to (`st_shndx`) and the offset of the symbol within the section (`st_value`).

Calling a Function by Name

- Using the information of file mapping address, offset of the section within the file and offset of the symbol within the section, we can calculate the logical address of the symbol after mapping.
- If it is a function it can be called. Note that the function name may not the one in your program in C++. That is one reason we use the object module of a C program.

Note

A function with relocatable entries will produce wrong result.

Bibliography

1. http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

2. https://docs.oracle.com/cd/E19620-01/805-4693/6j4emccrq/index.html