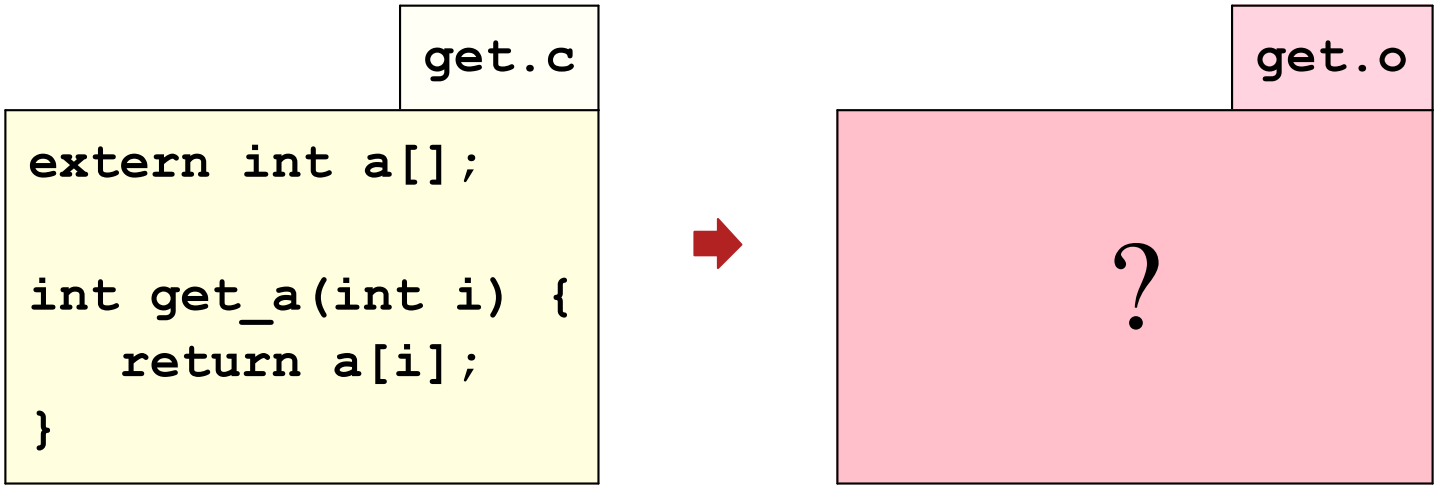
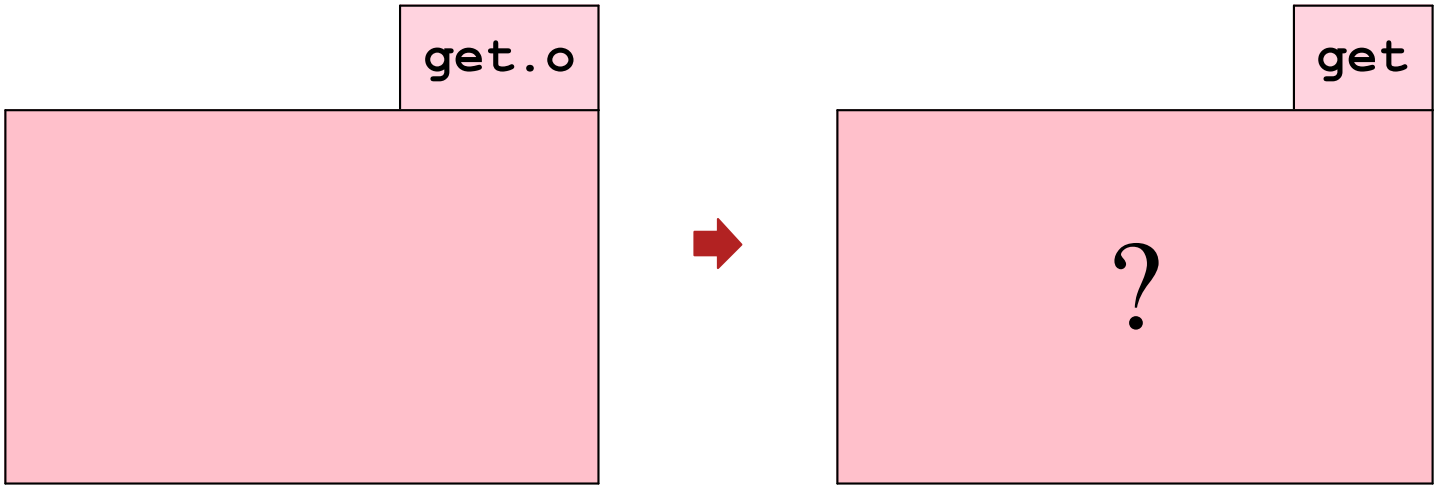
# Object File Content




- machine code
- global variable initialization
- defined and used symbols
- debugging information
- backtrace information

... and more!

# Executable Content



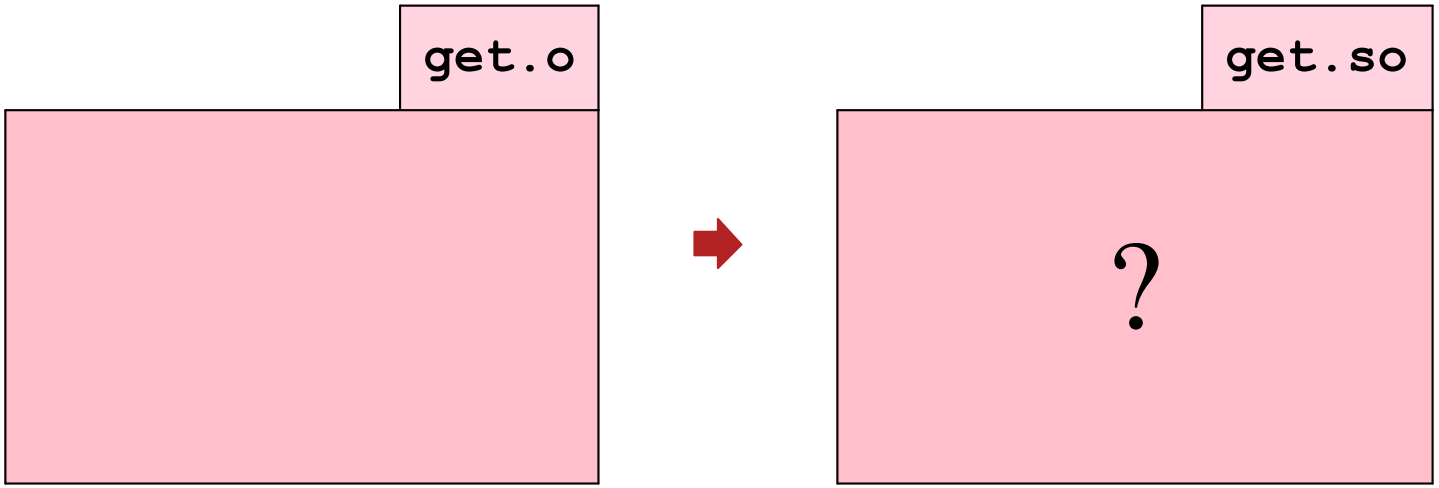

- machine code
- global variable initialization
- defined and used symbols
- debugging information
- backtrace information

... and more!

- machine code
- global variable initialization
- defined and used symbols
- debugging information
- backtrace information

... and more!

# Shared Object Content




- machine code
- global variable initialization
- defined and used symbols
- debugging information
- backtrace information

... and more!

- machine code
- global variable initialization
- defined and used symbols
- debugging information
- backtrace information

... and more!

# ELF

***ELF***: Executable and Linkable Format

On Linux and most other variants of Unix:

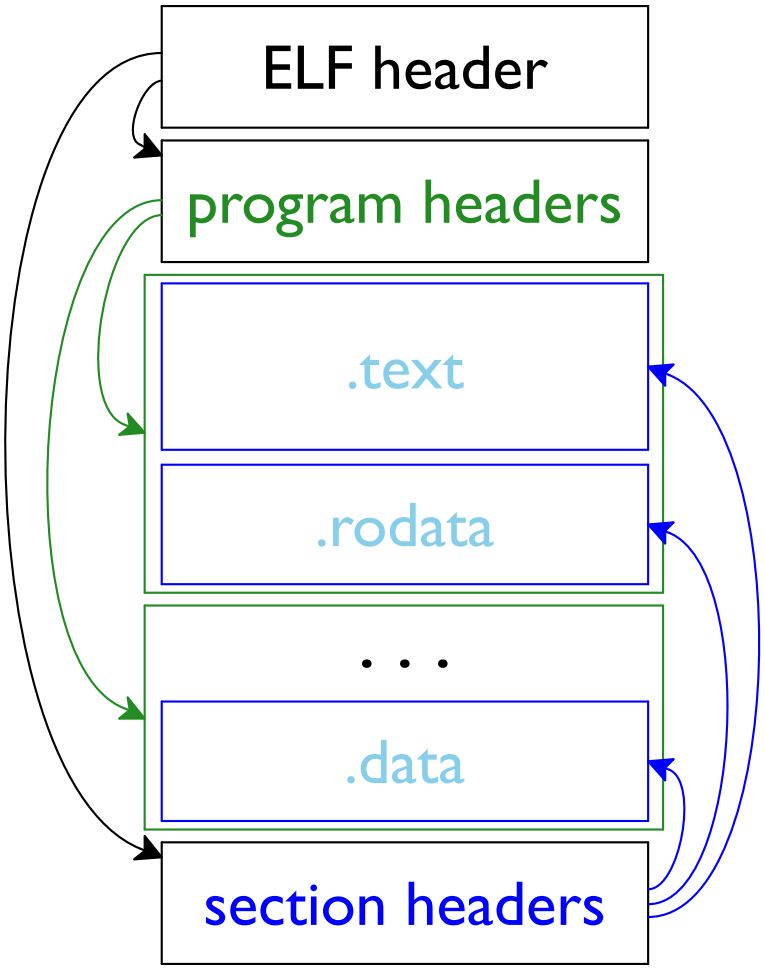
- Object files
- Executables
- Shared libraries

(Not used for static libraries)

Generic container, but tuned for fast loading

# ELF

***ELF***: Executable and Linkable Format



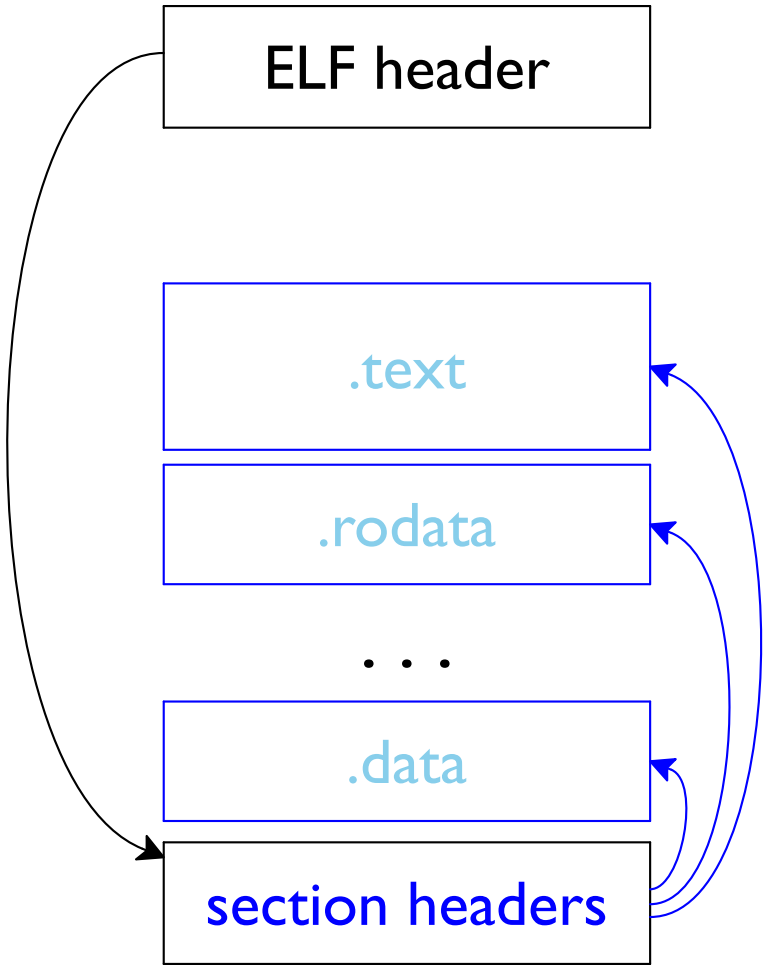

***segments*** = runtime view

***sections*** = linking view

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# ELF

***ELF***: Executable and Linkable Format

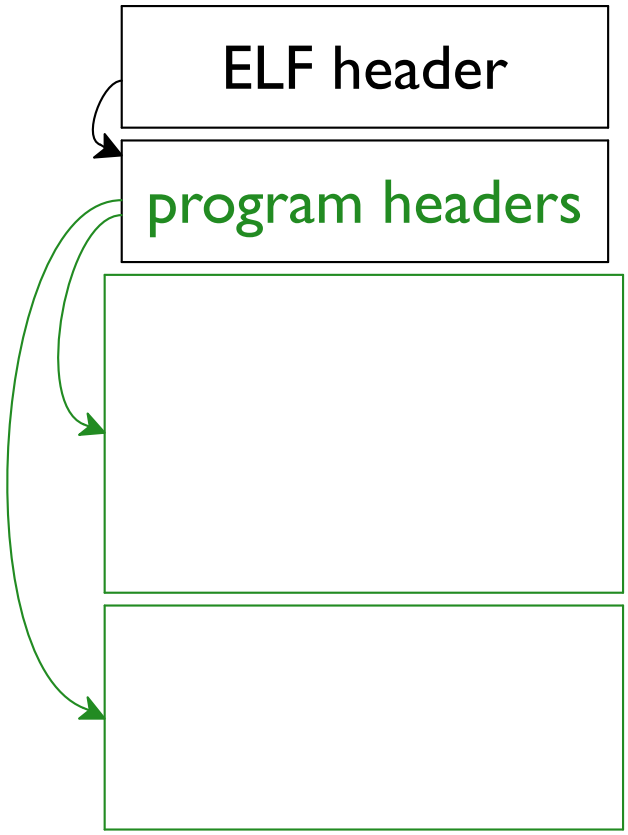



Object file: sections only

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# ELF

***ELF***: Executable and Linkable Format

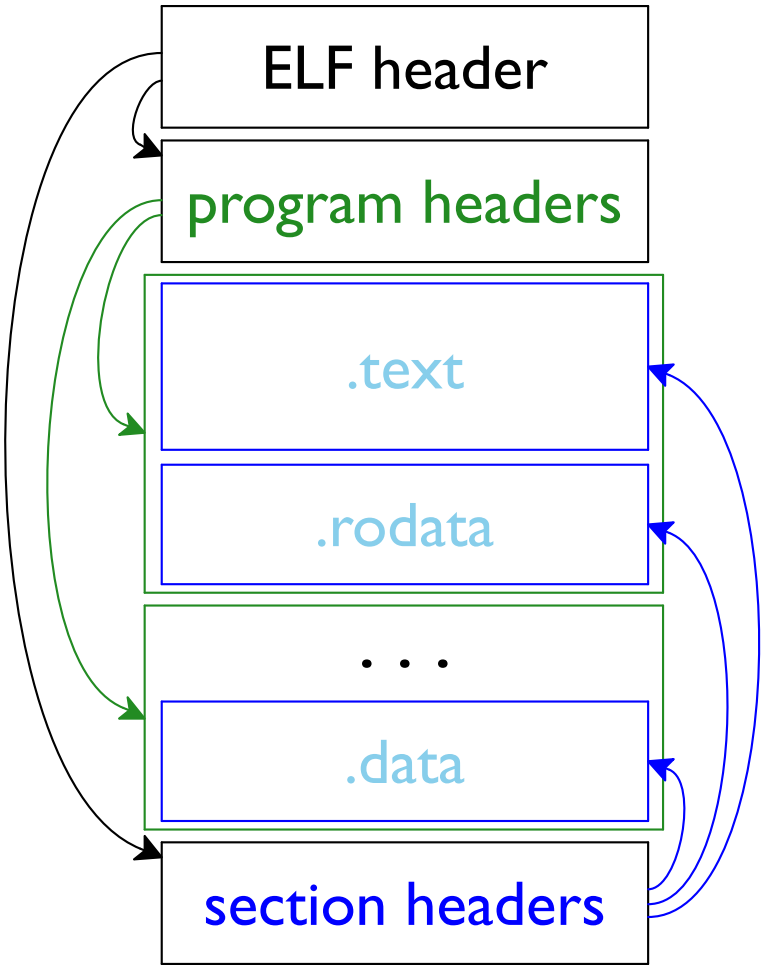



Executable: maybe segments only

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# ELF

***ELF***: Executable and Linkable Format




***segments*** = runtime view

***sections*** = linking view

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# Segments

Segments have ***types*** assigned by program headers

Meaning of a type is defined by the OS:

- **`PT_LOAD`** (= 1) — load into memory
- **`PT_DYNAMIC`** (= 2) — dynamic-linking info
- **`PT_INTERP`** (= 3) — interpreter of dyn-linking info
- **`PT_TLS`** (= 7) — thread-local storage

These constants are from **`elf.h`**

# Sections

Sections have ***names*** assigned by section headers

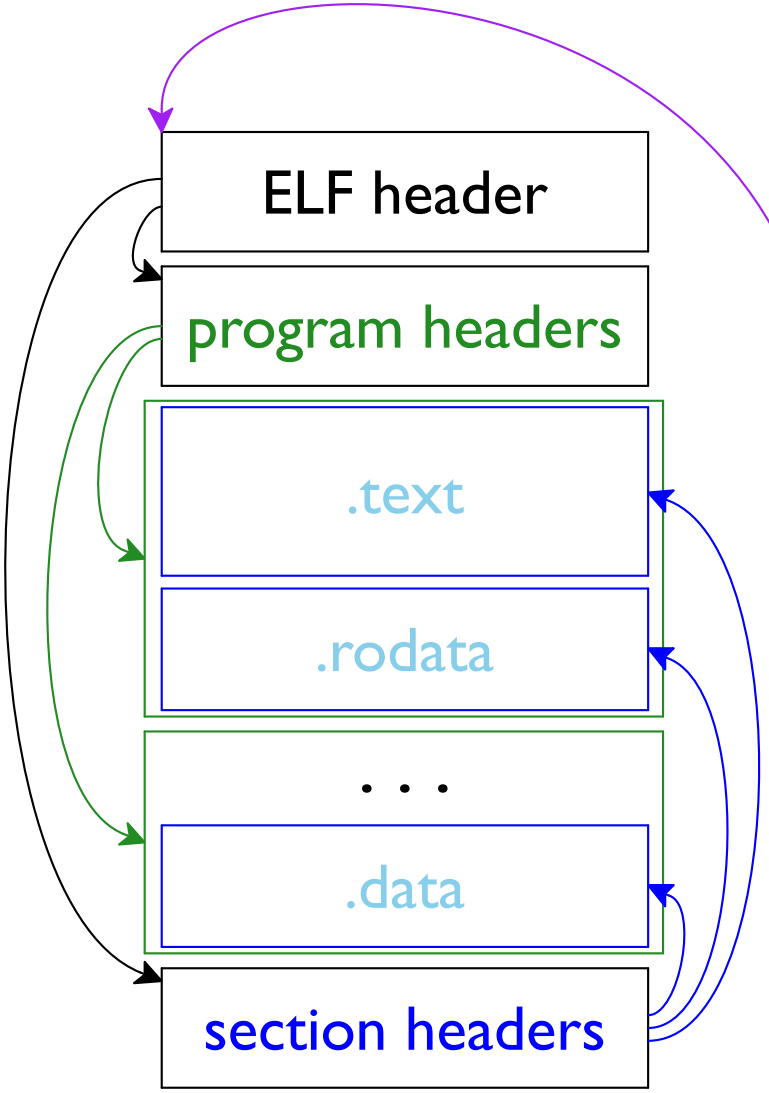
Some are standard, others are allowed:

- `.text` — machine code
- `.bss` — uninitialized global variables
- `.data` — initialized global variables
- `.rodata` — read-only global variables
- `.symtab` — defined/used functions and variables
- `.strtab` — strings (referenced by symbols)
- `.shstrtab` — strings (referenced by section table)

# Using `readelf`

```
$ readelf -a demo.o
```

# File Format Details

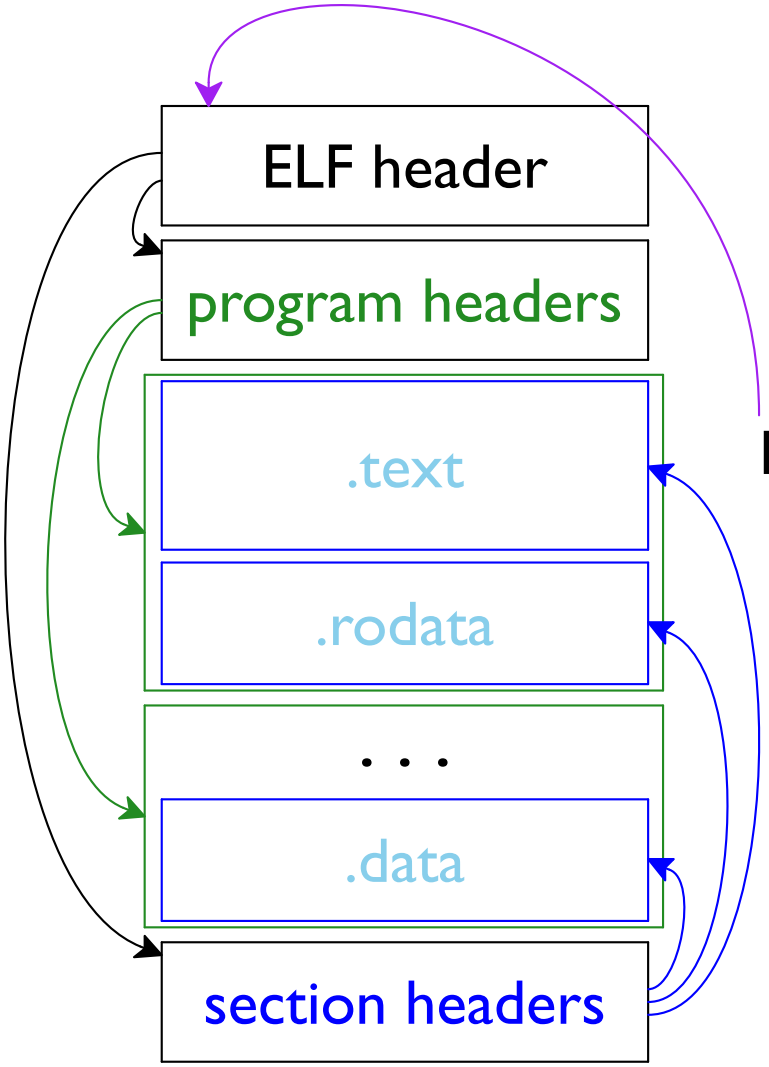



Initial bytes:

```
0x7F  'E'  'L'  'F'
```

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# File Format Details




Bytes 16-17 specify file type:

- **`ET_REL`** (= 1) = object file
- **`ET_EXEC`** (= 2) = executable
- **`ET_DYN`** (= 3) = shared library

Figure inspired by **`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`**

# ELF Header

/usr/include/elf.h

```
typedef struct {
  unsigned char  e_ident[EI_NIDENT]; /* 16 bytes */
  Elf64_Half  e_type;  /* File type */
  ....
  Elf64_Addr  e_entry; /* Entry point virtual address */
  Elf64_Off   e_phoff; /* Prog headers file offset */
  Elf64_Off   e_shoff; /* Sec headers file offset */
  ....
  Elf64_Half e_phentsize; /* Prog headers entry size */
  Elf64_Half e_phnum;     /* Prog headers entry count */
  Elf64_Half e_shentsize; /* Sec headers entry size */
  Elf64_Half e_shnum;     /* Sec headers entry count */
  Elf64_Half e_shstrndx;  /* Sec string table index */
} Elf64_Ehdr;
```

Copy file into memory, cast pointer to **`Elf64_Ehdr*`**

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <elf.h>

int main(int argc, char **argv) {
  /* Open the file and get its size: */
  int fd = open(argv[1], O_RDONLY);
  size_t len = lseek(fd, 0, SEEK_END);

  /* Map the whole file into memory: */
  void *p = mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);

  Elf64_Ehdr *ehdr = (Elf64_Ehdr *)p;

  if ((ehdr->e_ident[0] != 0x7F) || (ehdr->e_ident[1] != 'E')
      || (ehdr->e_ident[2] != 'L') || (ehdr->e_ident[3] != 'F'))
    printf("not an ELF file!\n");

  switch (ehdr->e_type) {
  case ET_REL: printf("object\n"); break;
  case ET_EXEC: printf("executable\n"); break;
  case ET_DYN: printf("shared library\n"); break;
  }

  return 0;
}
```

Copy

# ELF References

Two ways to refer to ELF content:

- **`Elf64_Off`** — offset within the file

  When we copy a file into memory, equals bytes to add to the copy's address

- **`Elf64_Addr`** — address when loaded for running

  Completely different than the address of a plain copy into memory

program headers (for segments) and
section headers provide a consistent mapping

**`Elf64_Off`** → **`Elf64_Addr`**

# Run-time Addresses

```
#include <stdio.h>

int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8};
const int b[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int c[8];

int main() {
  printf("%p = a\n", a);
  printf("%p = b\n", b);
  printf("%p = c\n", c);
  printf("%p = main\n", main);
  return 0;
}
```

Copy

**a** resides in the **.data** section

# Run-time Addresses

```
#include <stdio.h>

int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8};
const int b[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int c[8];

int main() {
  printf("%p = a\n", a);
  printf("%p = b\n", b);
  printf("%p = c\n", c);
  printf("%p = main\n", main);
  return 0;
}
```

Copy

**b** resides in the **.rodata** section

# Run-time Addresses

```
#include <stdio.h>

int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8};
const int b[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int c[8];

int main() {
  printf("%p = a\n", a);
  printf("%p = b\n", b);
  printf("%p = c\n", c);
  printf("%p = main\n", main);
  return 0;
}
```

Copy

`c` resides in the `.bss` section

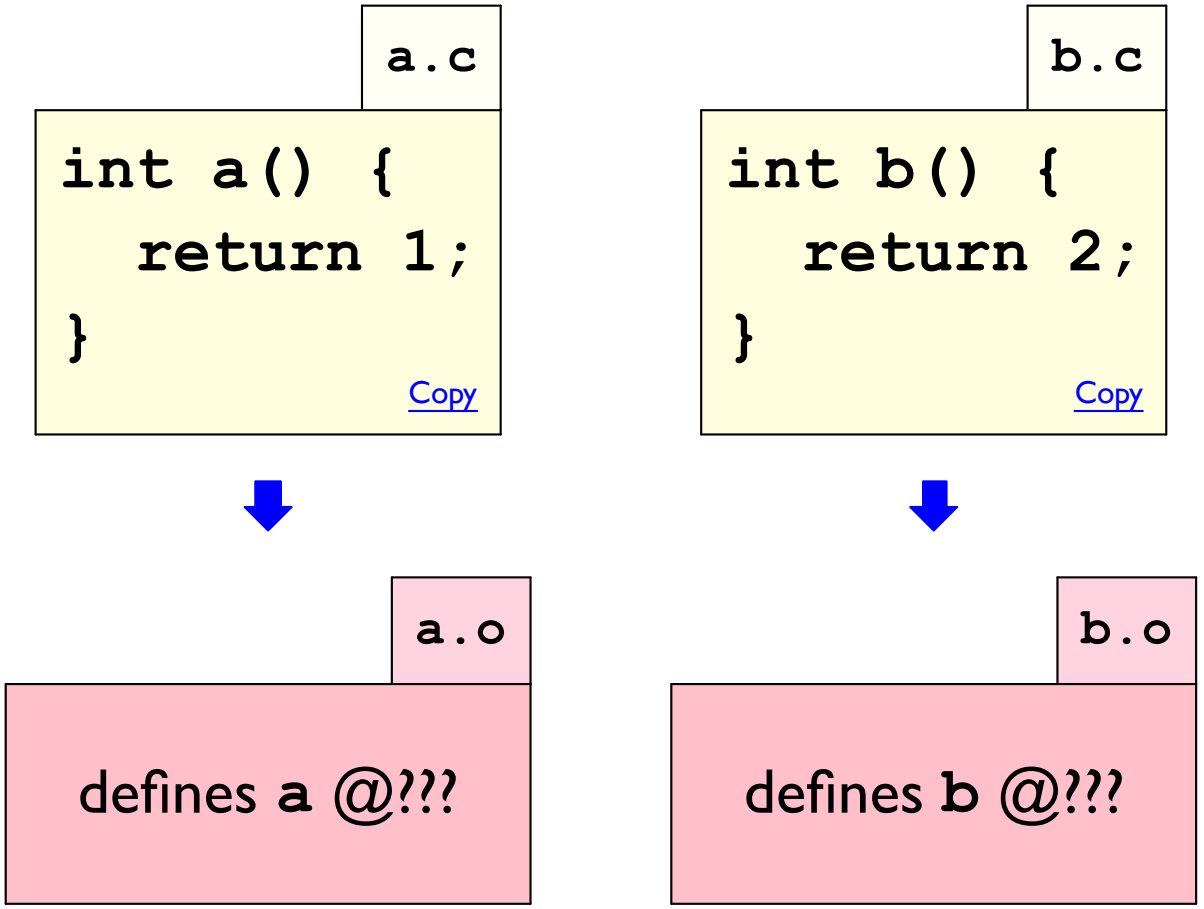
# Run-time Addresses

```
#include <stdio.h>

int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8};
const int b[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int c[8];

int main() {
  printf("%p = a\n", a);
  printf("%p = b\n", b);
  printf("%p = c\n", c);
  printf("%p = main\n", main);
  return 0;
}
```

Copy

**main** resides in the **.text** section

# Run-time Addresses in Object Files




How does a compiler avoid using the same address for different functions?

# Run-time Addresses in Object Files

main.c

```
int helper() {
  return 1;
}

int main() {
  return helper();
}
```

Copy

```
$ gcc -c main.c
$ objdump -d main.o
...
$ gcc main.o
$ objdump -d a.out
```

# Run-time Addresses in Object Files

In an object file:

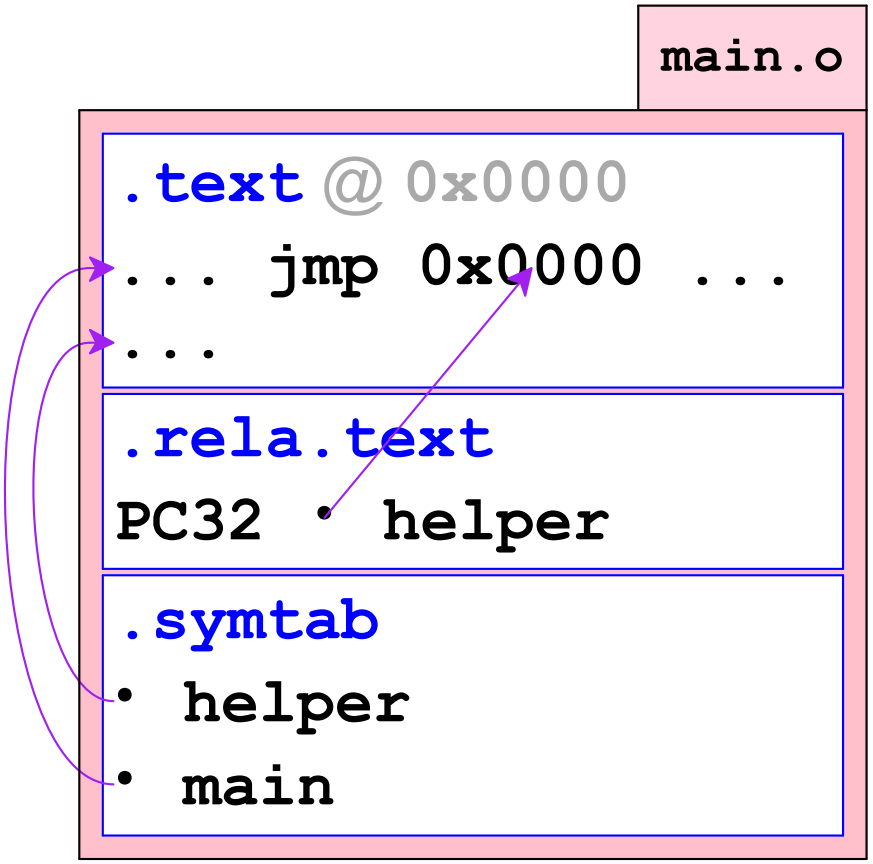
`.text` — machine code with zeroed addresses

`.rela.text` — details on how to fix addresses

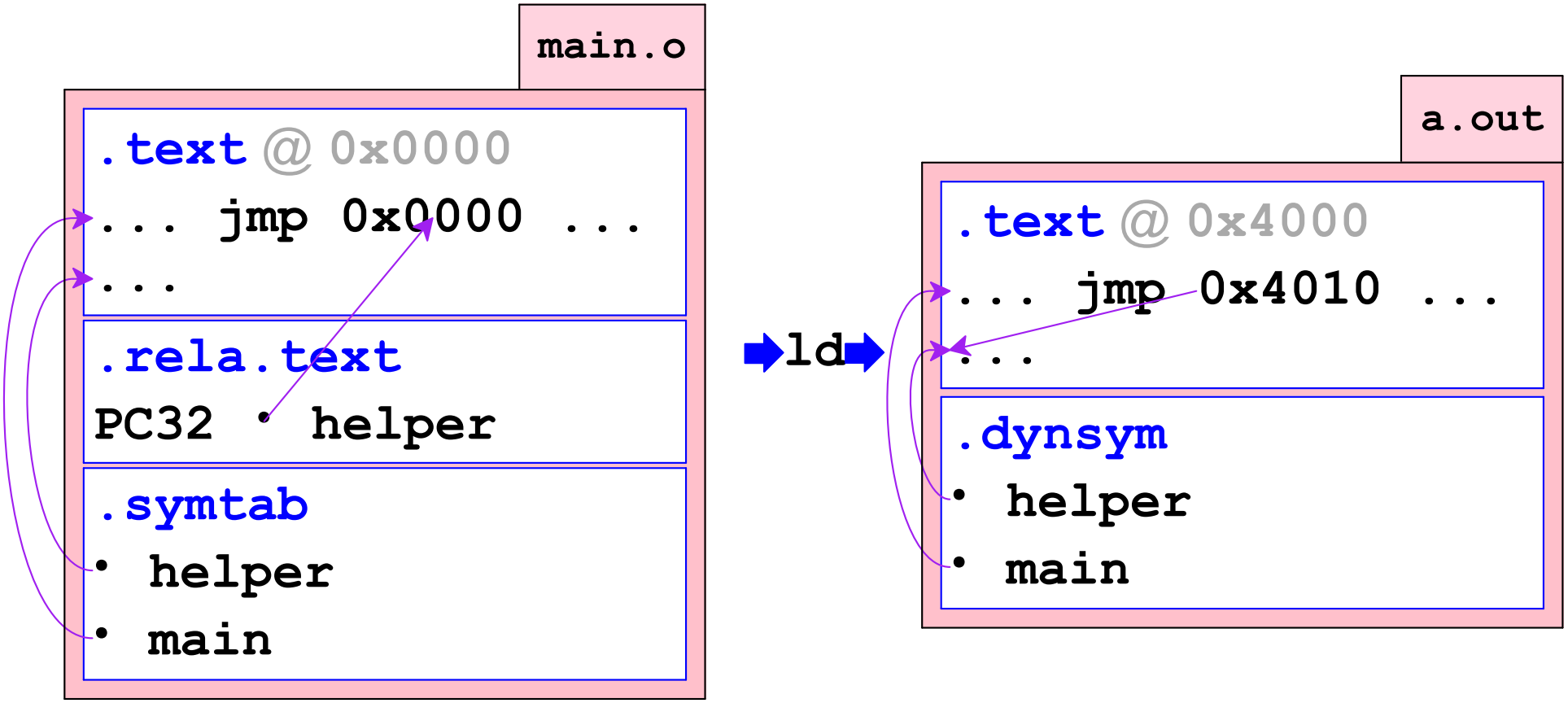
- location in `.text` to repair
- symbol whose final address to use

`.symtab` — maps names to section-relative offsets

# Objects and Linking



# Objects and Linking



# Objects and Linking

a.c

```
int a() {
  return 1;
}
```

Copy

main.c

```
int main() {
  return a() + b();
}
```

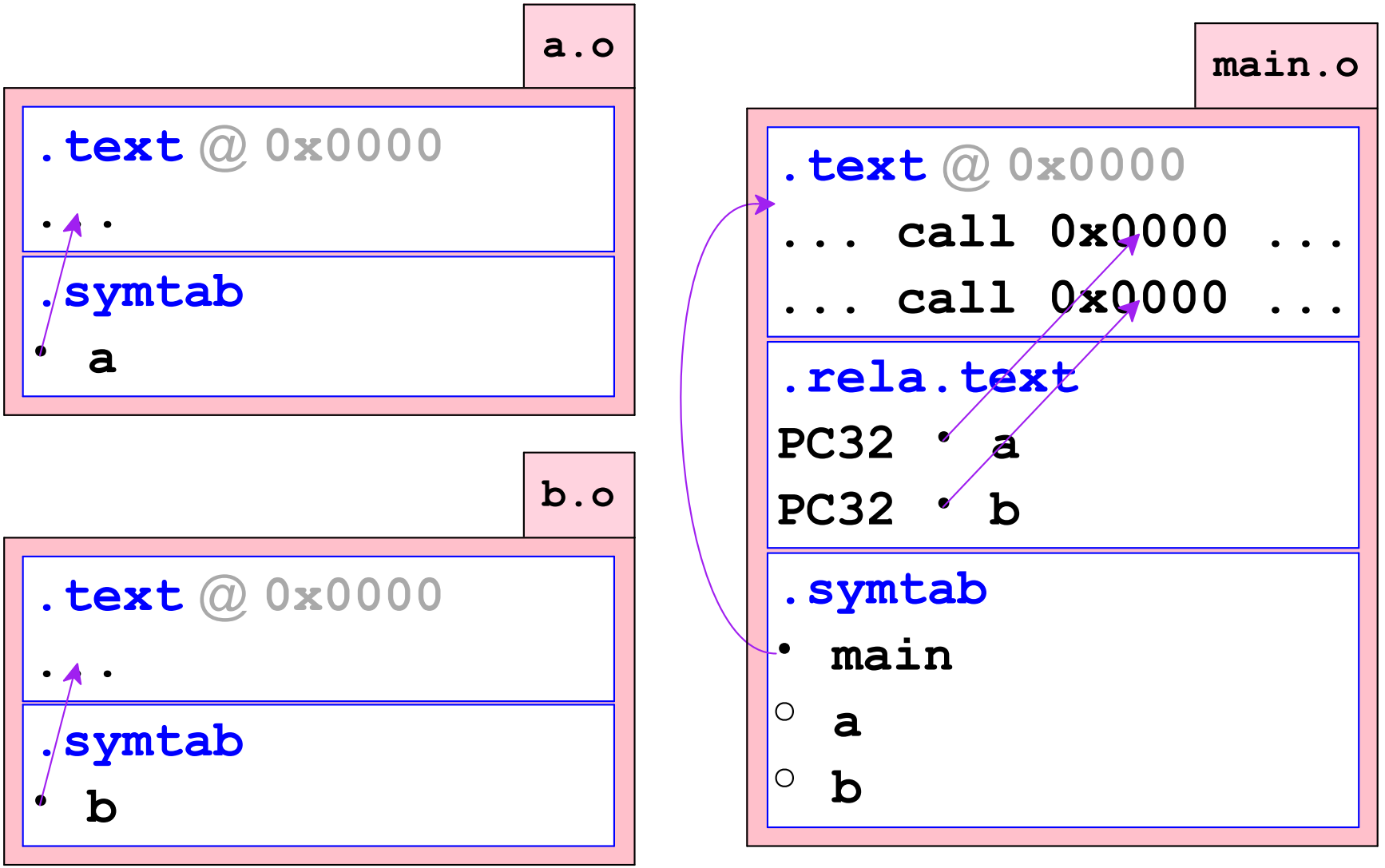
Copy

b.c

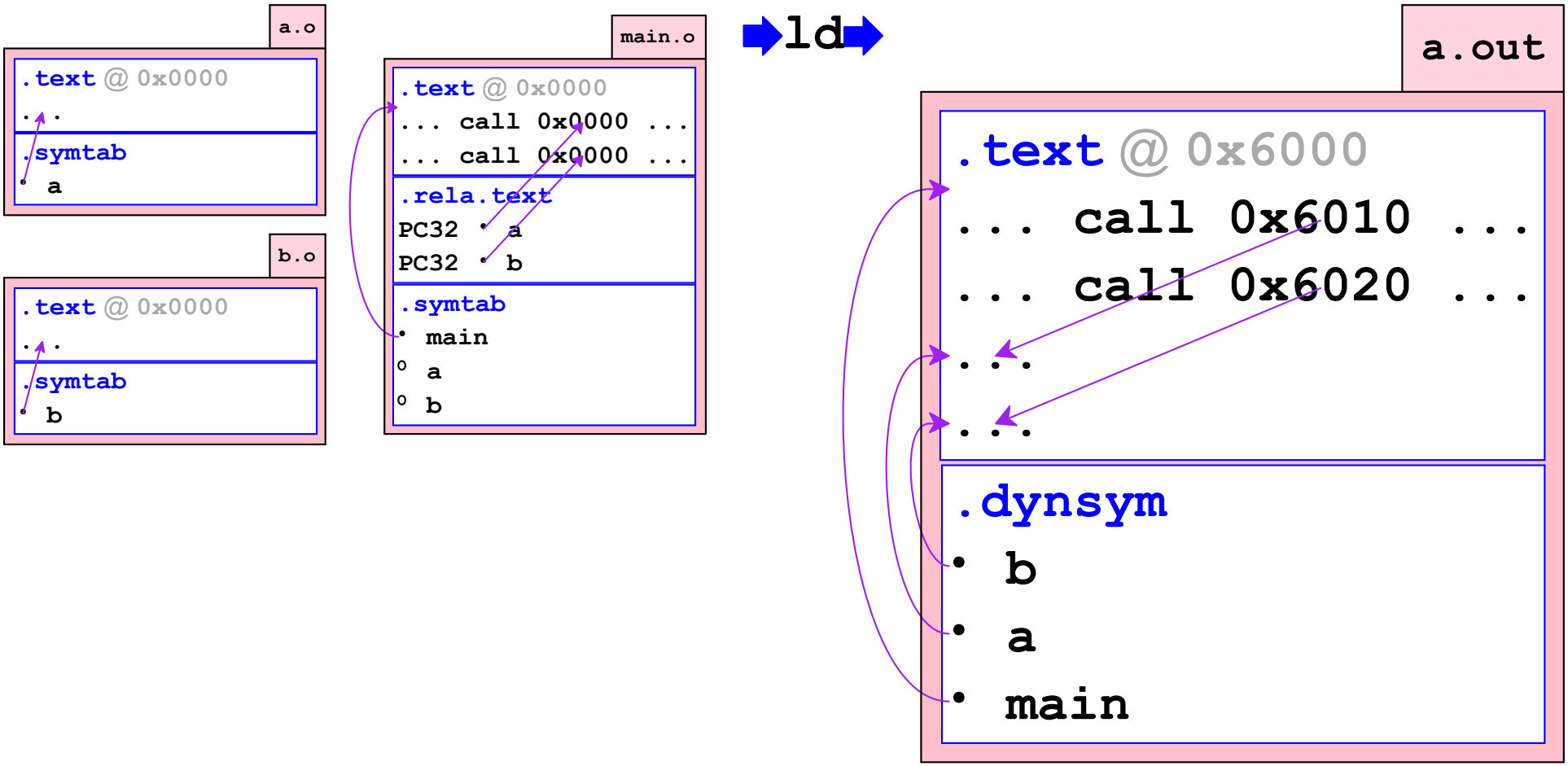
```
int b() {
  return 2;
}
```

Copy

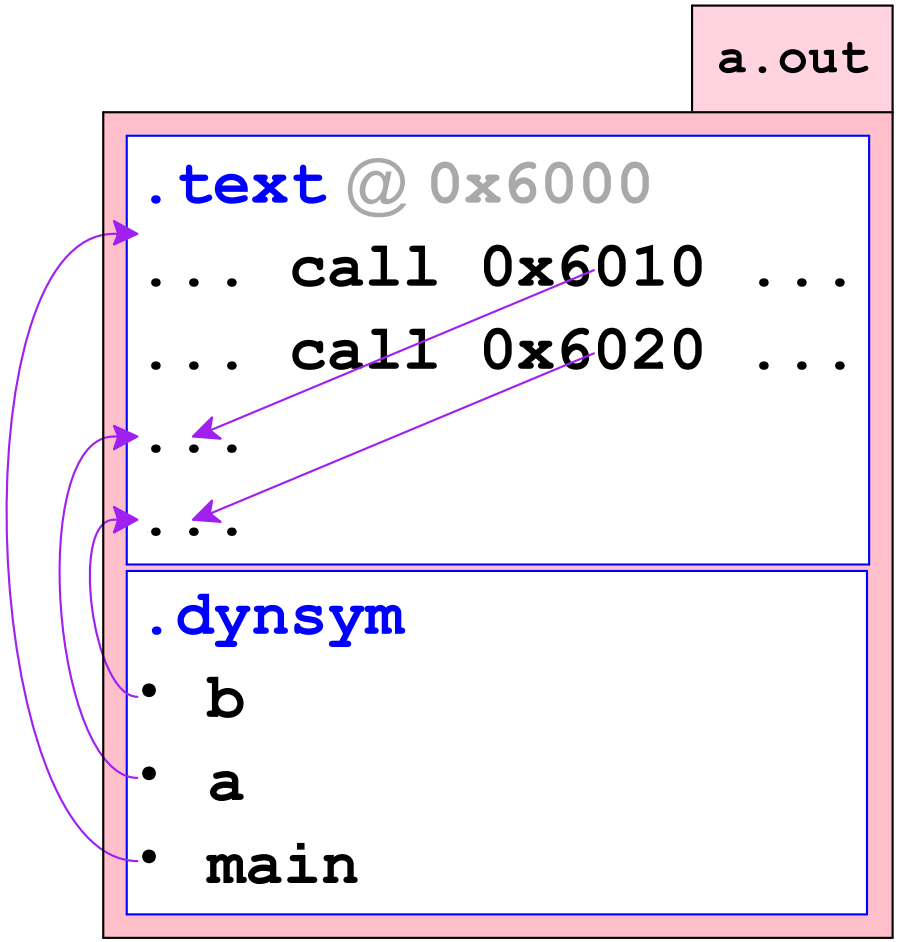
# Objects and Linking



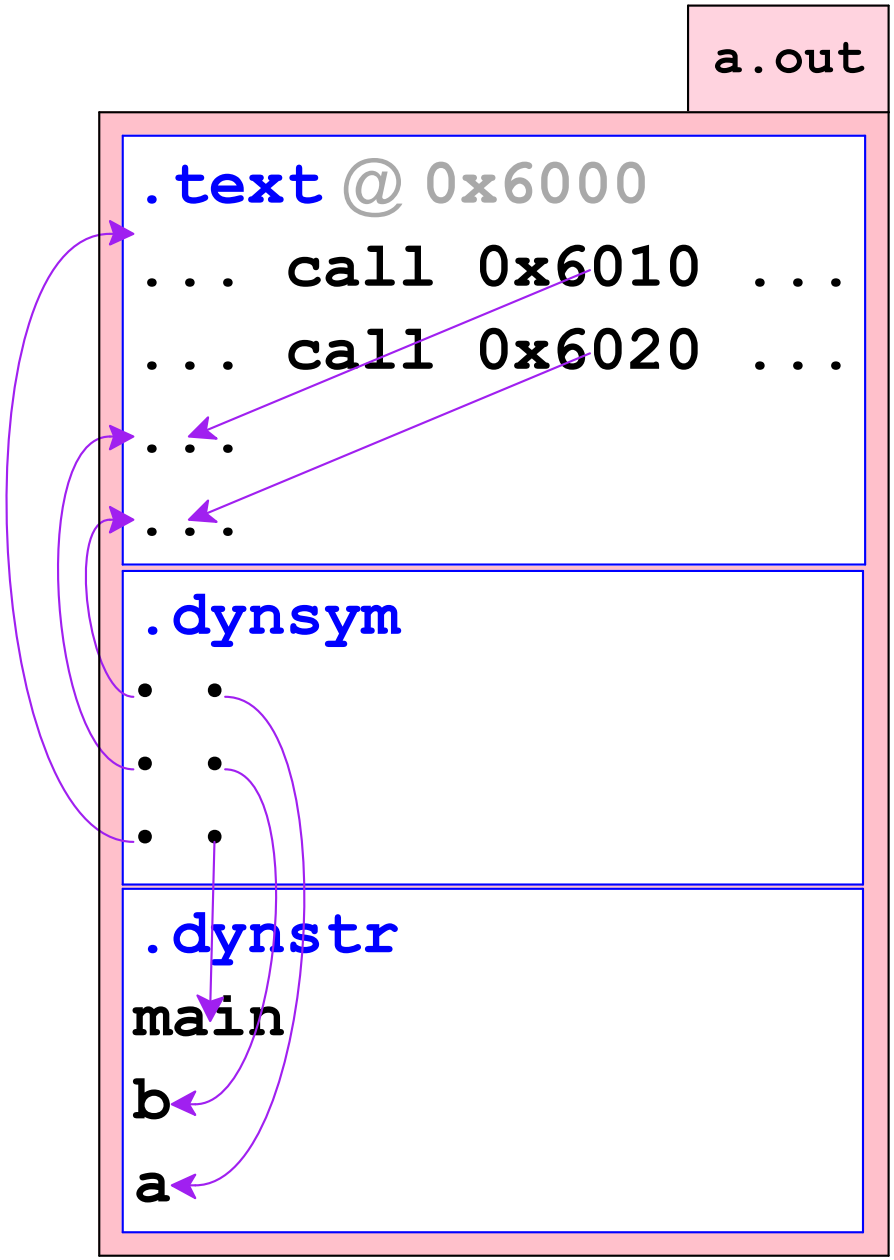
# Objects and Linking



# Symbol Names



# Symbol Names



# Example: Printing Section Names

```
typedef struct {
  ....
  Elf64_Off   e_shoff; /* Sec headers file offset */
  ....
  Elf64_Half e_shnum;     /* Sec headers entry count */
  Elf64_Half e_shstrndx;  /* Sec string table index */
} Elf64_Ehdr;

typedef struct {
  Elf64_Word sh_name;  /* offset in .shstrtab */
  ....
  Elf32_Addr sh_addr;  /* addr at execution */
  Elf64_Off sh_offset; /* file offset */
  Elf64_Xword sh_size; /* size in bytes */
} Elf64_Shdr;
```

```
Elf64_Shdr *shdrs = (void*)ehdr+ehdr->e_shoff;
```

# Example: Printing Section Names

```
typedef struct {
  ....
  Elf64_Off   e_shoff; /* Sec headers file offset */
  ....
  Elf64_Half e_shnum;     /* Sec headers entry count */
  Elf64_Half e_shstrndx;  /* Sec string table index */
} Elf64_Ehdr;

typedef struct {
  Elf64_Word sh_name;  /* offset in .shstrtab */
  ....
  Elf32_Addr sh_addr;  /* addr at execution */
  Elf64_Off sh_offset; /* file offset */
  Elf64_Xword sh_size; /* size in bytes */
} Elf64_Shdr;
```

```
Elf64_Shdr *shdrs = (void*)ehdr+ehdr->e_shoff;
for (i = 0; i < ehdr->e_shnum; i++) {

}
```

# Example: Printing Section Names

```
typedef struct {
  ....
  Elf64_Off   e_shoff; /* Sec headers file offset */
  ....
  Elf64_Half e_shnum;     /* Sec headers entry count */
  Elf64_Half e_shstrndx;  /* Sec string table index */
} Elf64_Ehdr;

typedef struct {
  Elf64_Word sh_name;  /* offset in .shstrtab */
  ....
  Elf32_Addr sh_addr;  /* addr at execution */
  Elf64_Off sh_offset; /* file offset */
  Elf64_Xword sh_size; /* size in bytes */
} Elf64_Shdr;
```

```
Elf64_Shdr *shdrs = (void*)ehdr+ehdr->e_shoff;
for (i = 0; i < ehdr->e_shnum; i++) {
  ... shdrs[i].sh_name ...
}
```

# Example: Printing Section Names

```
Elf64_Shdr *shdrs = (void*)ehdr+ehdr->e_shoff;
char *strs = (void*)ehdr+shdrs[ehdr->e_shstrndx].sh_offset;
int i;

for (i = 0; i < ehdr->e_shnum; i++) {
  printf("%s\n", strs + shdrs[i].sh_name);
}
```

Copy

# Shared Library Relocations

a.c

```
int a() {
  return 1;
}
```

Copy

c.c

```
int c() {
  return a() + b();
}
```

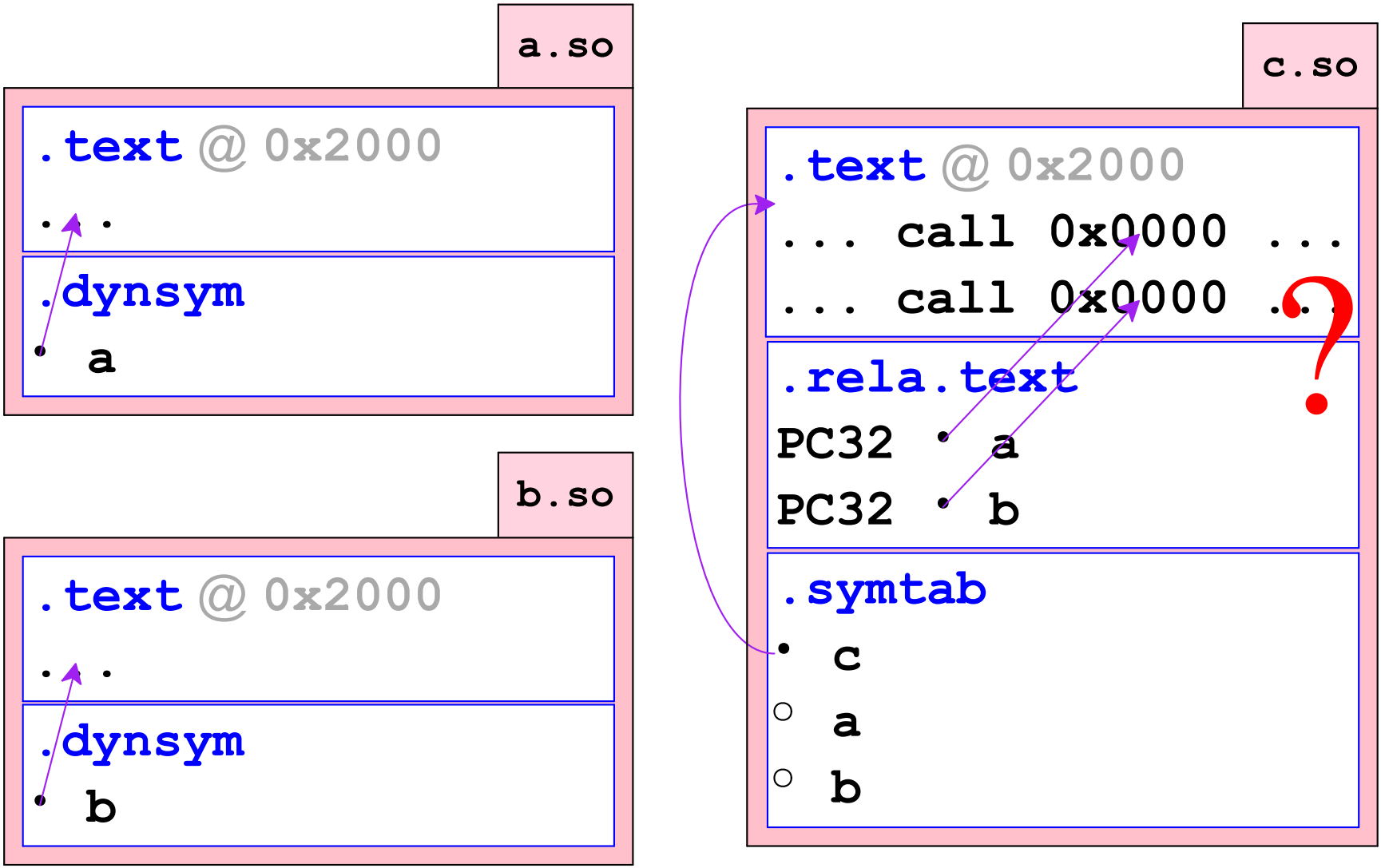
Copy

b.c

```
int b() {
  return 2;
}
```

Copy

# Shared Library Relocations




**Goal:** shared-library `.text` the same for every instance

# Position-Independent Code

Position-dependent

```
extern int v;

int a();
int b();

int f() {
  return v + a() + b();
}
```

# Position-Independent Code

Position-dependent

```
extern int v;

int a();
int b();

int f() {
  return v + a() + b();
}
```

Position-independent

```
int *vp;

int (*ap)();
int (*bp)();

int f() {
  return *vp + ap() + bp();
}
```

PI code does not depend on address of **v**, **a**, or **b**

***Dynamic linker*** must fill in **vp**, **ap**, and **bp**

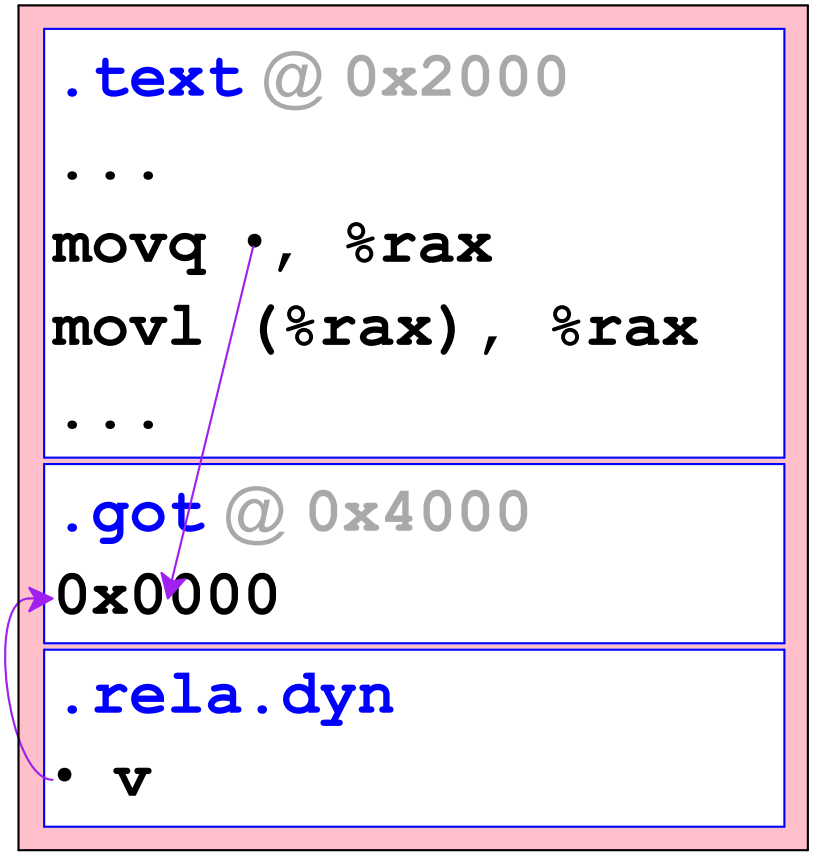
# Position-Independent Code: Variables

```
extern int v;

int f() {
  return v;
}
```

# Position-Independent Code: Variables

Using `gcc -fPIC -shared`:

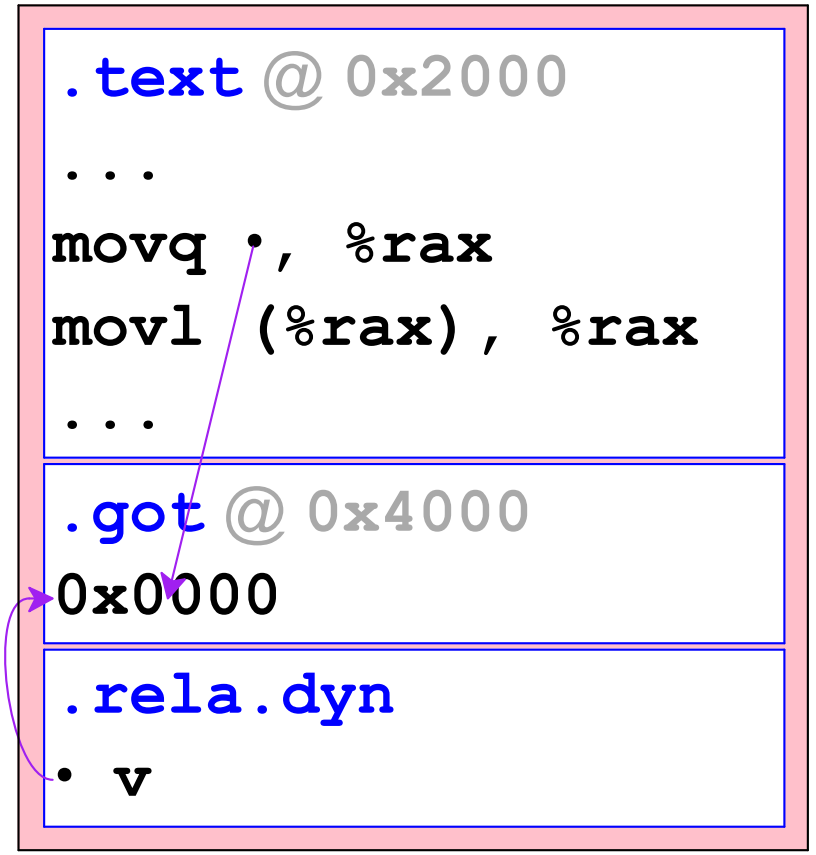
```
extern int v;

int f() {
  return v;
}
```



# Position-Independent Code: Variables

Using `gcc -fPIC -shared`:

```
extern int v;

int f() {
  return v;
}
```



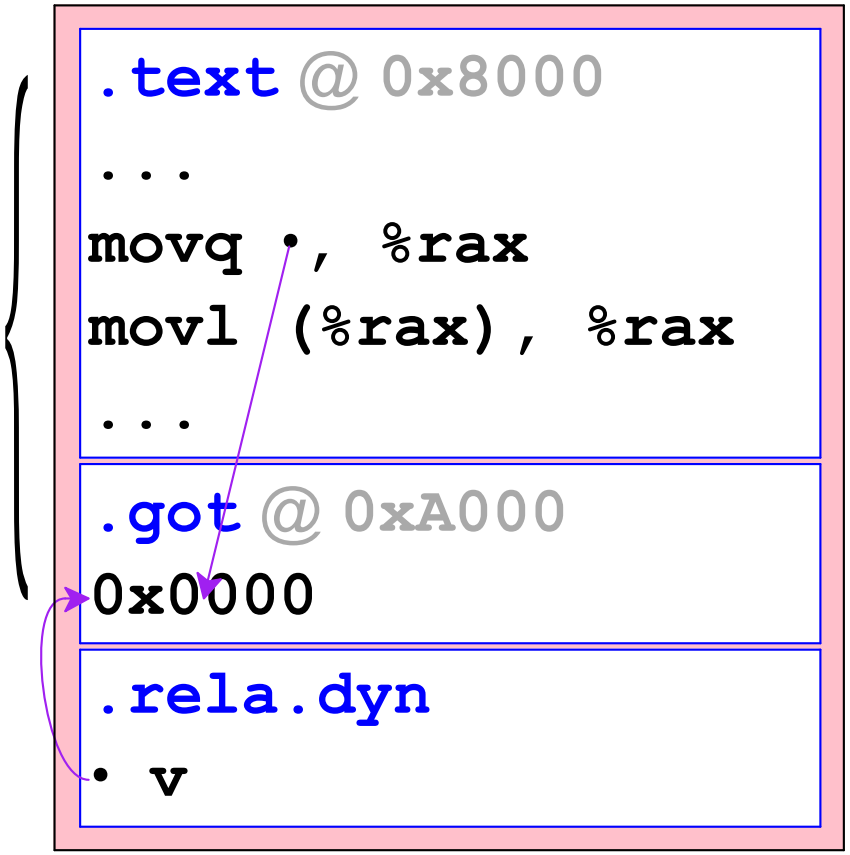

Reference from `.text` to `.got`:

```
mov 0x1FF0(%rip), %rax
```

# Position-Independent Code: Variables

Using `gcc -fPIC -shared`:

Dynamic linker can move, but must move both by the same amount




Reference from `.text` to `.got`:

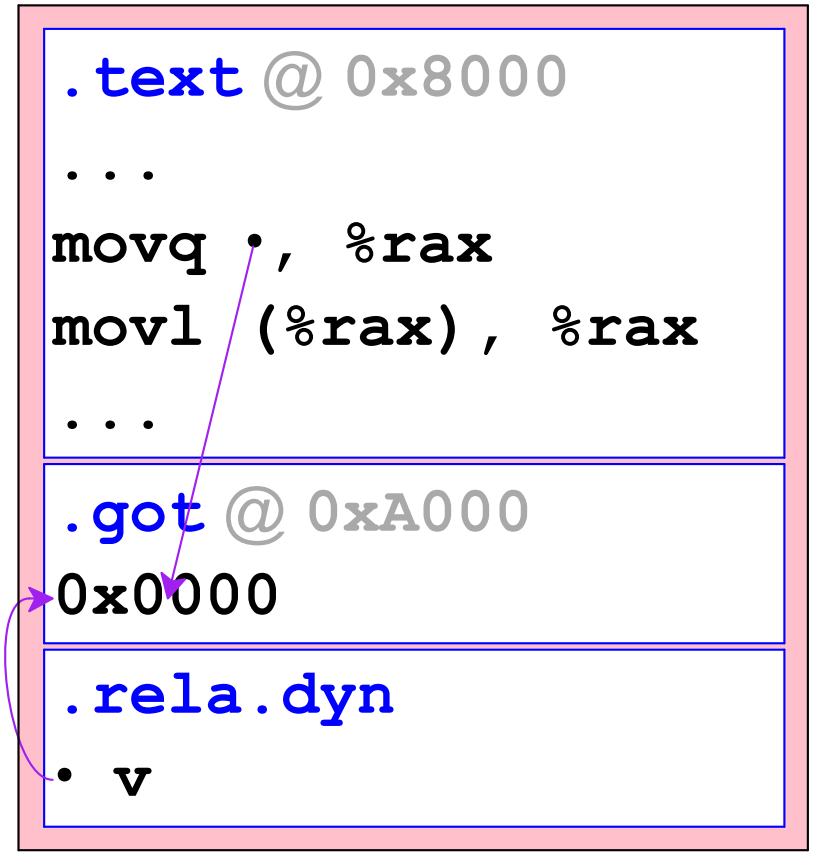
```
mov 0x1FF0(%rip), %rax
```

# Position-Independent Code: Variables

Using `gcc -fPIC -shared`:

Dynamic linker can move, but must move both by the same amount

Tells dynamic linker how to patch on startup




Reference from `.text` to `.got`:

```
mov 0x1FF0(%rip), %rax
```

# Position-Independent Code: Functions

PIC for function calls is different:

- Usually a lot more shared functions than shared data
- Only some of the function references actually happen

⇒ Lazy updating via interposition

# Position-Independent Code: Functions

Position-dependent

```
int a();

int f() {
  return a();
}
```

# Position-Independent Code: Functions

Position-dependent

```
int a();

int f() {
  return a();
}
```

Position-independent

```
void *real_a = <next in a@plt>;

int a@plt() {
  goto real_a;
  next:
   goto fixup(a@plt);
}

int f() {
  return a@plt();
}
```

# Position-Independent Code: Functions

Position-dependent

```
int a();

int f() {
  return a();
}
```

**`fixup`** adjusts the variable

Position-independent

```
void *real_a = <next in a@plt>;

int a@plt() {
  goto real_a;
  next:
   goto fixup(a@plt);
}

int f() {
  return a@plt();
}
```

# Position-Independent Code: Functions

Position-dependent

```
int a();

int f() {
  return a();
}
```

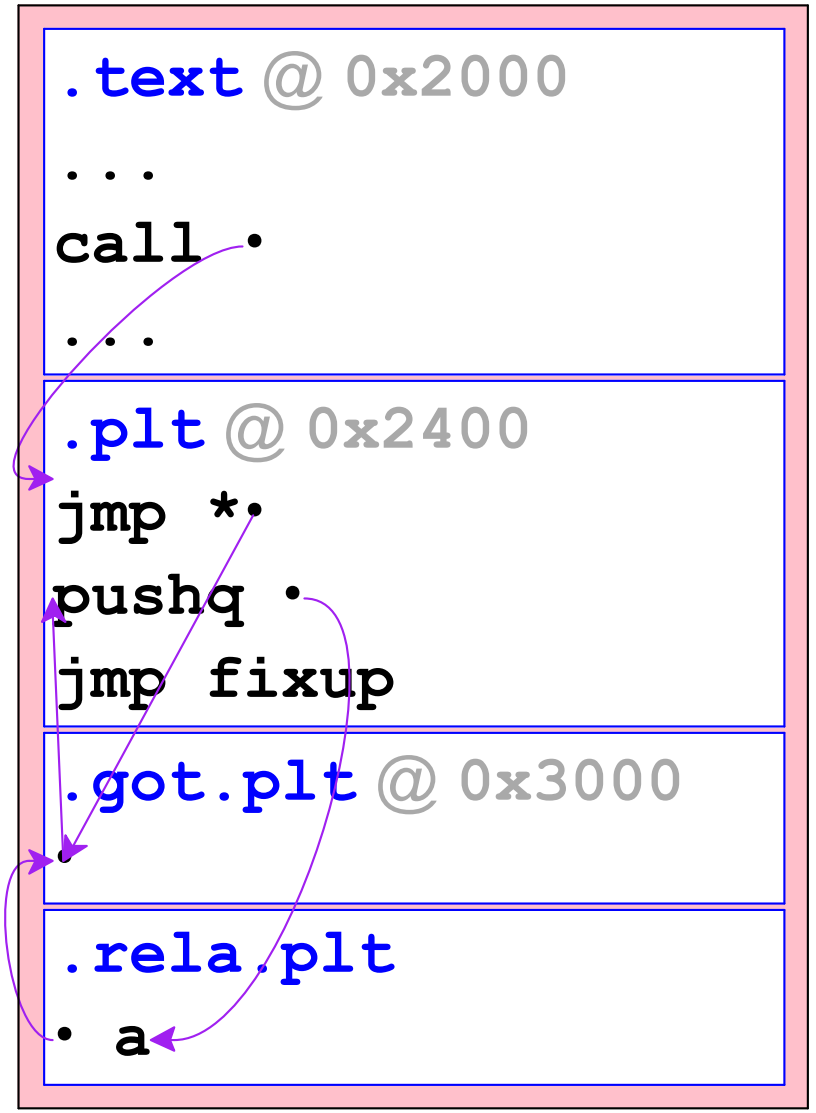
Position-independent

```
void *real_a = a;

int a@plt() {
  goto real_a;
  next:
   goto fixup(a@plt);
}

int f() {
  return a@plt();
}
```

# Position-Independent Code: Functions

Using **`gcc -fPIC -shared`**:

```
int a();

int f() {
  return a();
}
```



# ELF Helpers

```
/* Functions that you should be able to write: */
Elf64_Shdr *section_by_index(Elf64_Ehdr *ehdr, int idx);
Elf64_Shdr *section_by_name(Elf64_Ehdr *ehdr, char *name);

/* Helper to get pointer to section content: */
#define AT_SEC(ehdr, shdr) ((void *)(ehdr) + (shdr)->sh_offset)
```

# Symbol Tables

**.symtab** or **.dynsym** = array of **Elf64_Sym**

**elf.h**

```
typedef struct {
  Elf64_Word  st_name;   /* name (string index) */
  unsigned char st_info; /* type and binding */
  ....
  Elf64_Section st_shndx; /* section index */
  ....
  Elf64_Addr    st_value; /* location */
  Elf64_Xword   st_size;  /* size */
} Elf64_Sym;

#define ELF64_ST_TYPE(st_info) ((st_info) & 0xf)
```

```
Elf64_Shdr *dynsym_shdr = section_by_name(ehdr, ".dynsym");
Elf64_Sym *syms = AT_SEC(ehdr, dynsym_shdr);
char *strs = AT_SEC(ehdr, section_by_name(ehdr, ".dynstr"));
int i, count = dynsym_shdr->sh_size / sizeof(Elf64_Sym);
for (i = 0; i < count; i++) {
  printf("%s\n", strs + syms[i].st_name);
}
```

# Symbol Tables

**.symtab** or **.dynsym** = array of **Elf64_Sym**

**elf.h**

```
typedef struct {
  Elf64_Word  st_name;   /* name (string index) */
  unsigned char st_info; /* type and binding */
  ....
  Elf64_Section st_shndx; /* section index */
  ....
  Elf64_Addr    st_value; /* location */
  Elf64_Xword   st_size;  /* size */
} Elf64_Sym;

#define ELF64_ST_TYPE(st_info) ((st_info) & 0xf)
```

Checking for a function:

```
if (ELF64_ST_TYPE(syms[i].st_info) == STT_FUNC)
 ....
```

# Symbol Tables

**.symtab** or **.dynsym** = array of **Elf64_Sym**

**elf.h**

```
typedef struct {
  Elf64_Word  st_name;   /* name (string index) */
  unsigned char st_info; /* type and binding */
  ....
  Elf64_Section st_shndx; /* section index */
  ....
  Elf64_Addr    st_value; /* location */
  Elf64_Xword   st_size;  /* size */
} Elf64_Sym;

#define ELF64_ST_TYPE(st_info) ((st_info) & 0xf)
```

Inspecting function machine code:

```
Elf64_Shdr *shdr = section_by_index(ehdr, syms[i].st_hndx);

.... AT_SEC(ehdr,shdr) + (syms[i].st_value - shdr->sh_addr) ....
```

# Relocation Records

**.rela.dyn** or **.rela.plt** = array of **Elf64_Rela**

elf.h

```
typedef struct {
  Elf64_Addr    r_offset;
  Elf64_Xword   r_info; /* type and sym index */
  Elf64_Sxword  r_addend;
} Elf64_Rela;

#define ELF64_R_SYM(r_info)   ((r_info) >> 32)
```

```
Elf64_Shdr *rela_dyn_shdr = section_by_name(ehdr, ".rela.dyn");
Elf64_Rela *relas = AT_SEC(ehdr, rela_dyn_shdr);
int i, count = rela_dyn_shdr->sh_size / sizeof(Elf64_Rela);

for (i = 0; i < count; i++) {
  printf("%d\n", ELF64_R_SYM(relas[i].r_info));
}
```

# Machine Code

`.text` and `.plt` contain machine code

- Some symbols point to start of function machine code
- Machine code may contain PC-relative jumps or call to other machine code
- Jumps or calls may stay in section or go to `.plt`

# Machine Code

```
$ objdump -d f.so
...
0000000000006b8 <f>:
 6b8:   55                      push   %rbp
 6b9:   48 89 e5                mov    %rsp,%rbp
 6bc:   b8 00 00 00 00          mov    $0x0,%eax
 6c1:   e8 ea fe ff ff          callq  5b0 <a@plt>
 6c6:   5d                      pop    %rbp
 6c7:   c3                      retq
```

# Machine Code

```
$ objdump -d f.so
...
0000000000006b8 <f>:
 6b8:   55                      push   %rbp
 6b9:   48 89 e5                mov    %rsp,%rbp
 6bc:   b8 00 00 00 00          mov    $0x0,%eax
 6c1:   e8 ea fe ff ff          callq  5b0 <a@plt>
 6c6:   5d                      pop    %rbp
 6c7:   c3                      retq
```

Raw bytes are content from the section