INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

System Software
CSN-252
Linkers

R. E. Bryant Chap. On Linkers

What Do Linkers Do?

- Step 1: Symbol resolution
  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* refer symbol swap */`
    - `int *xp = &x;      /* define symbol xp, refer x */`
  - Symbol definitions are stored in object file (by assembler) in *symbol table* (linux)
    - Each entry includes name, size, and location of symbol
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

What Do Linkers Do?

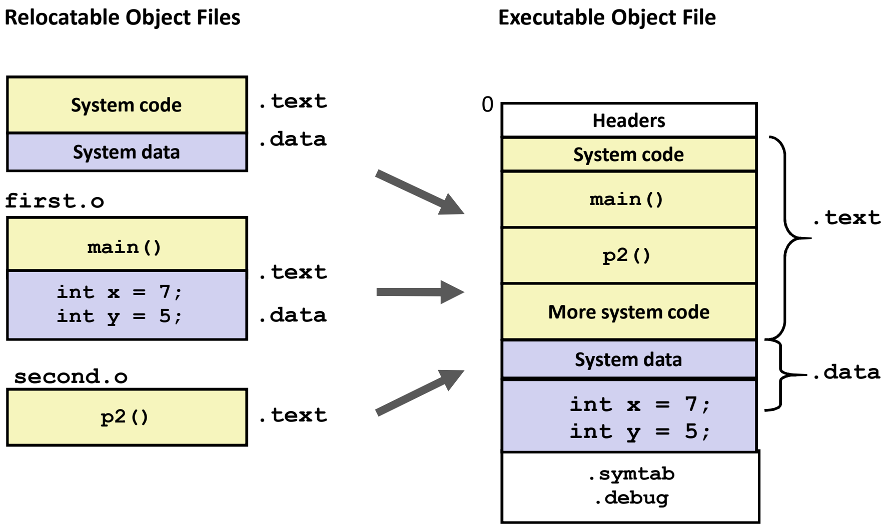
Step 2: Relocation

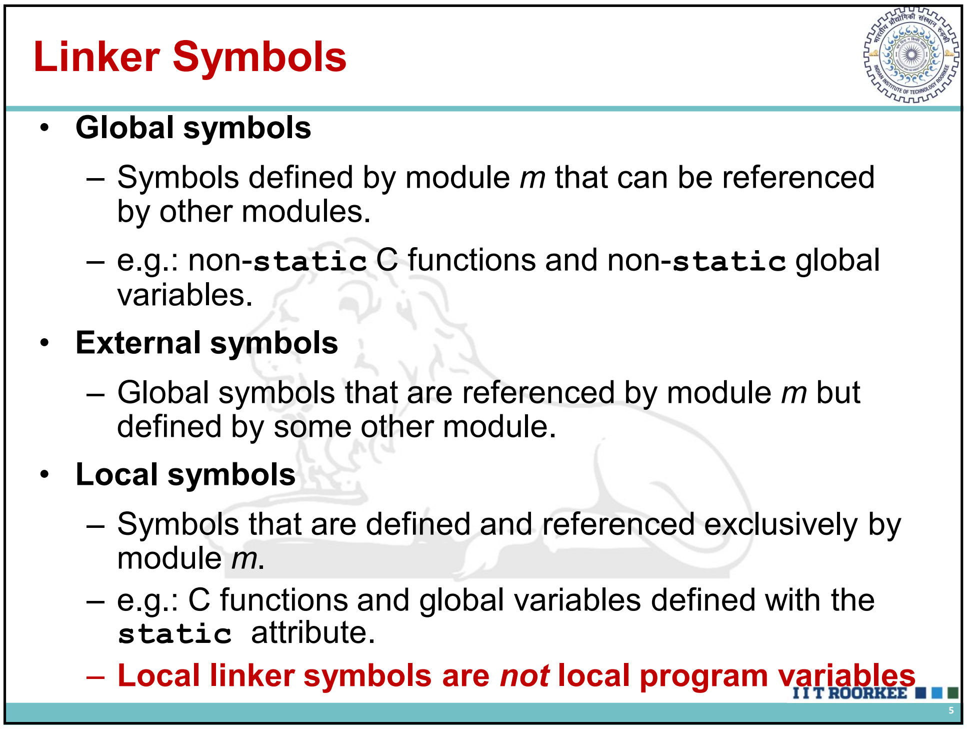
Merges separate code and data sections into single sections

Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.

Updates all references to these symbols to reflect their new positions.

Step 2: Relocation



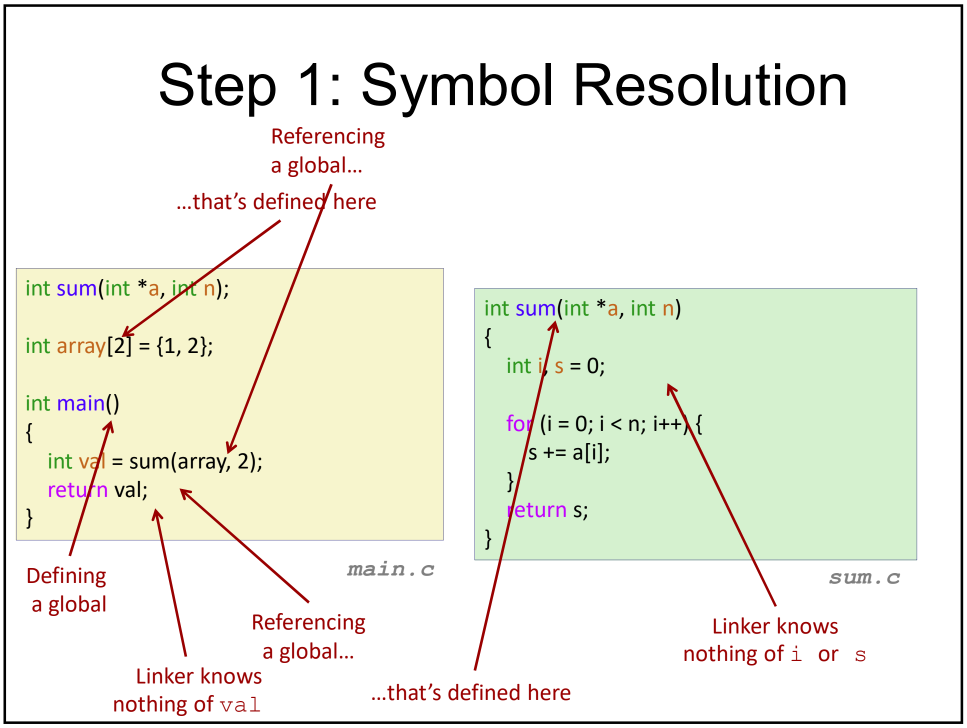


Linker Symbols

- **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**

IIT ROORKEE

5



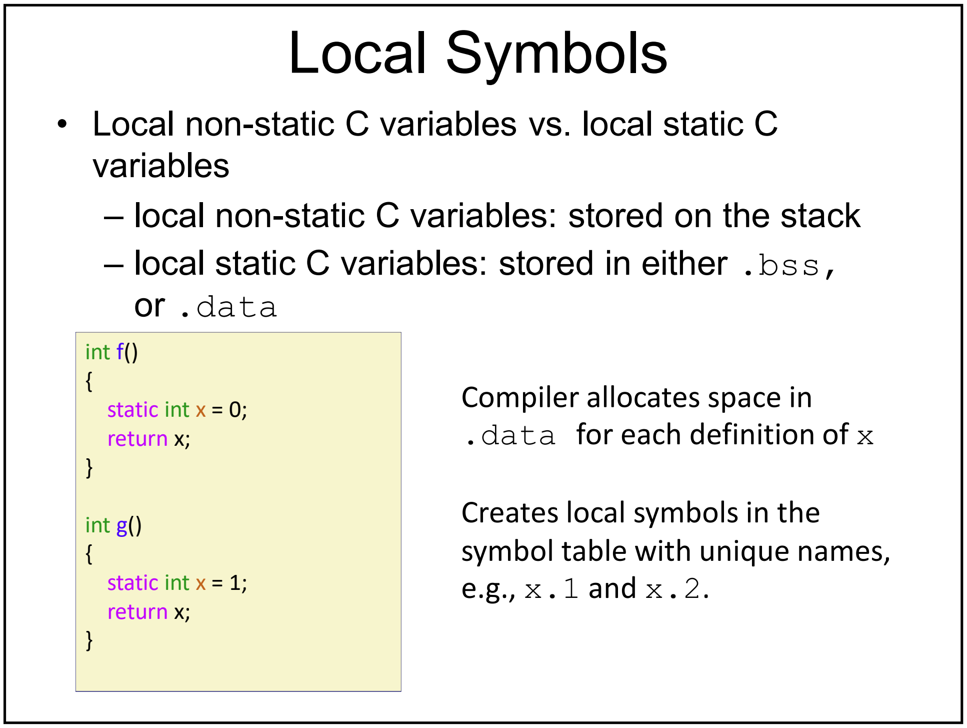
Step 1: Symbol Resolution

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

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

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

main.c

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

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

sum.c



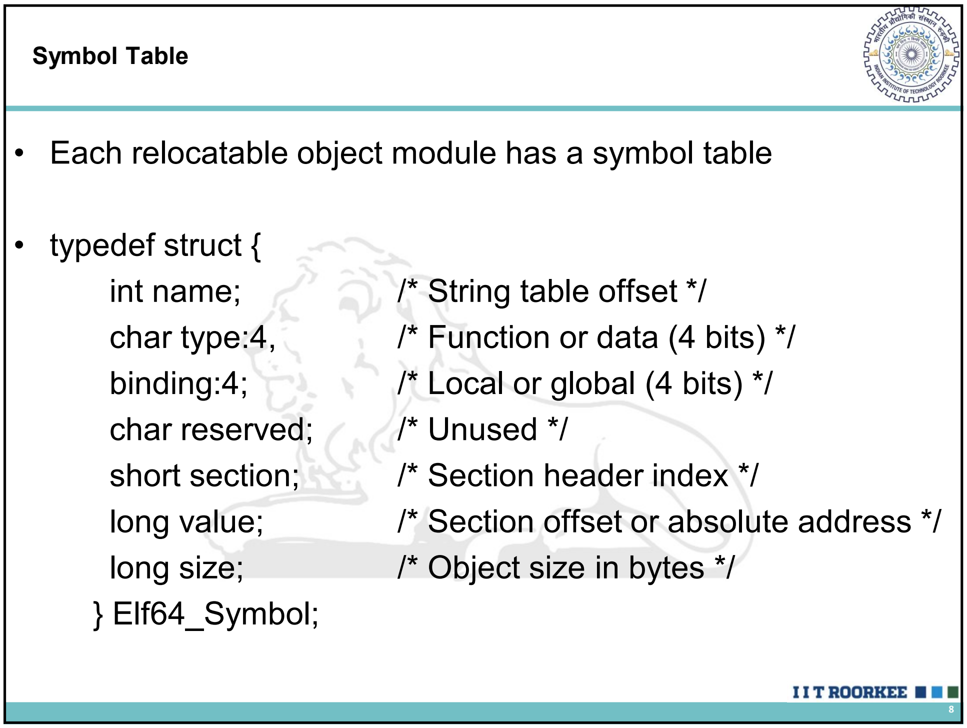
Local Symbols

- 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;
   return x;
}

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`.

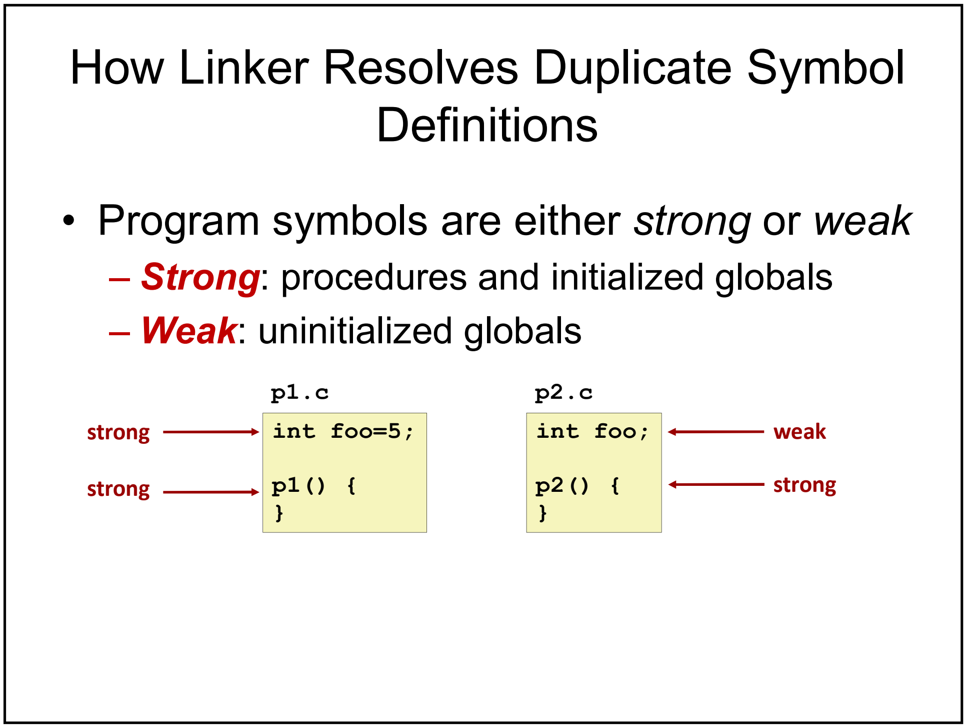


Symbol Table

- Each relocatable object module has a symbol table

- typedef struct {

```
        int name;             /* String table offset */
        char type:4,          /* Function or data (4 bits) */
        binding:4;            /* Local or global (4 bits) */
        char reserved;        /* Unused */
        short section;        /* Section header index */
        long value;           /* Section offset or absolute address */
        long size;            /* Object size in bytes */
     } Elf64_Symbol;
```



How Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals

p1.c

```
int foo=5;

p1() {
}
```

strong

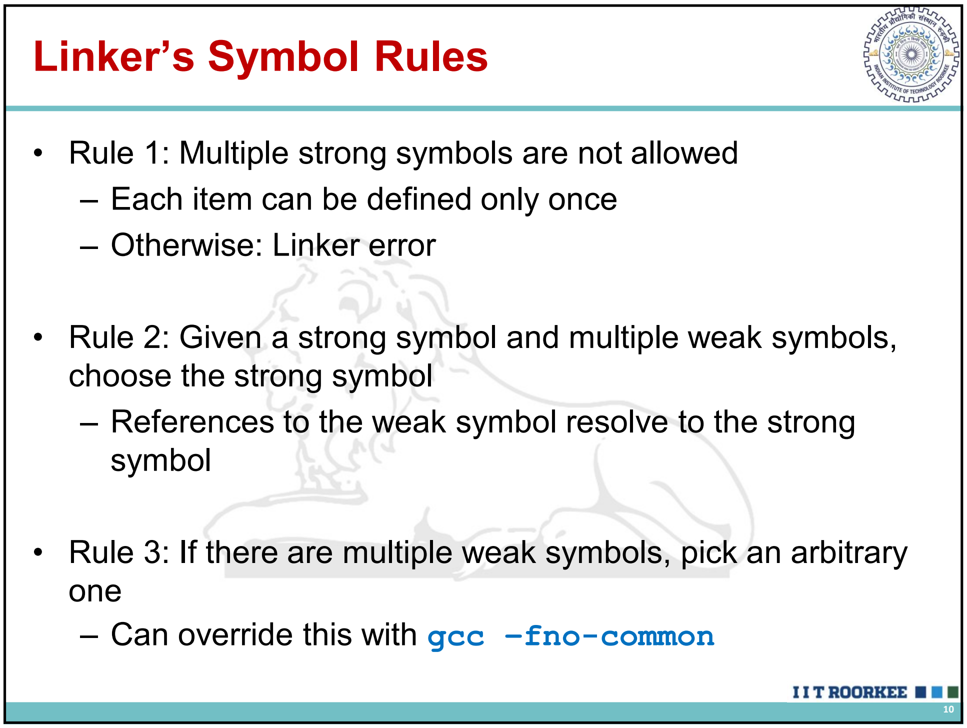
strong

p2.c

```
int foo;

p2() {
}
```

weak

strong



Linker's Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`

IIT ROORKEE

10

Linker Puzzles

`int x;`
`p1() {}`
	`p1() {}`	Link time error: two strong symbols (`p1`)

`int x;`
`p1() {}`
	`int x;`
	`p2() {}`
		References to `x` will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` might overwrite `y`!

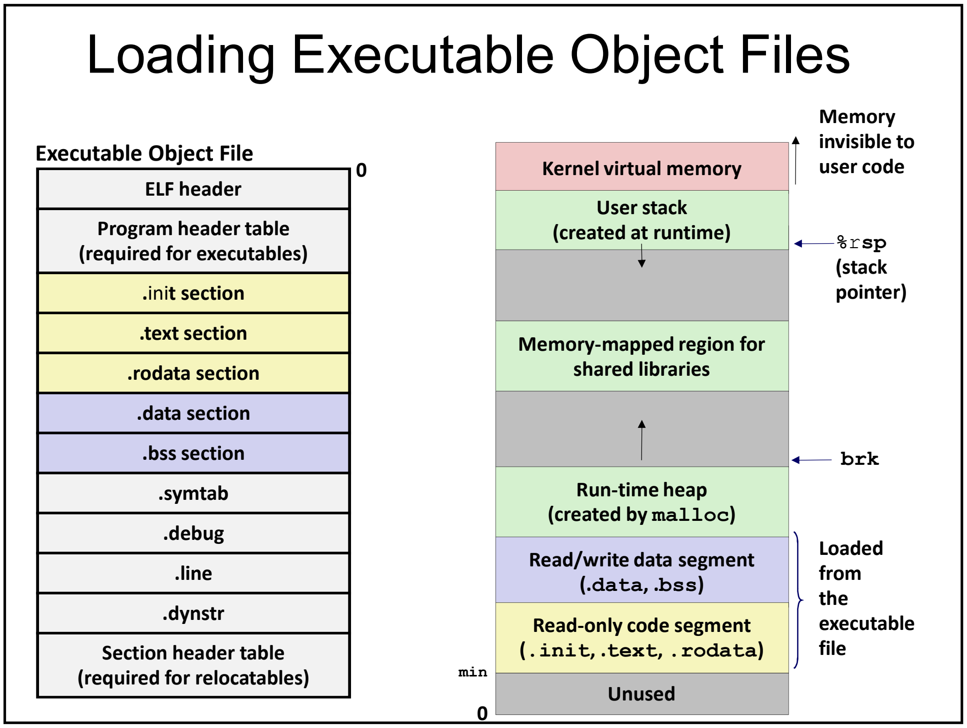
`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` will overwrite `y`!

`int x=7;`
`p1() {}`
	`int x;`
	`p2() {}`
		References to `x` will refer to the same initialized variable.

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



Loading Executable Object Files

Executable Object File

0

- ELF header
- Program header table (required for executables)
- .init section
- .text section
- .rodata section
- .data section
- .bss section
- .symtab
- .debug
- .line
- .dynstr
- Section header table (required for relocatables)

- Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write data segment (`.data`, `.bss`)
- Read-only code segment (`.init`, `.text`, `.rodata`)
- Unused

Memory invisible to user code

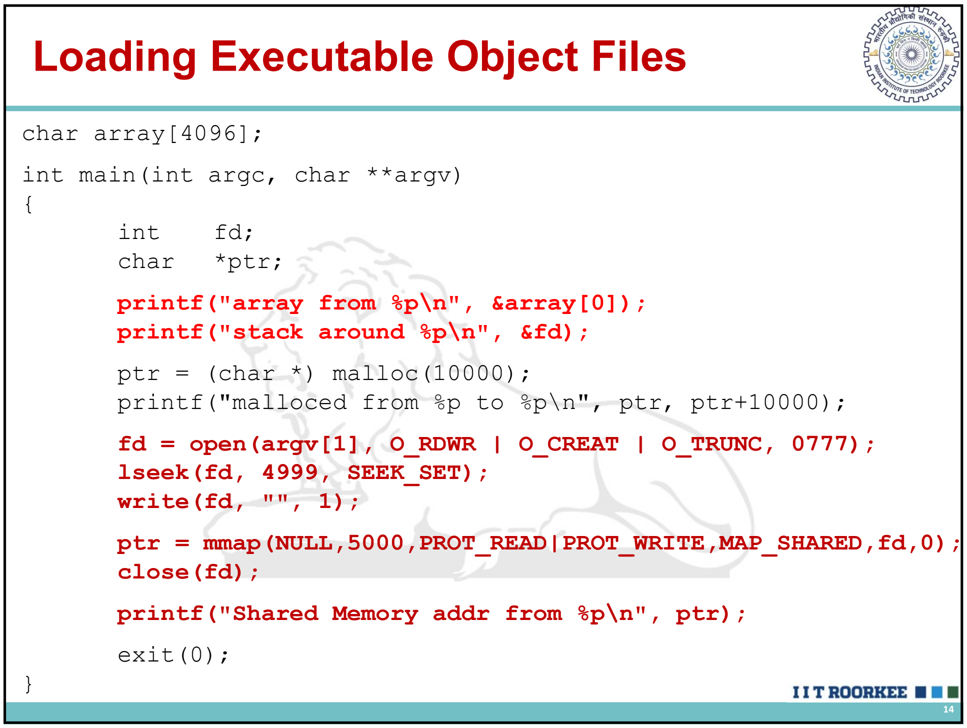
`%rsp` (stack pointer)

`brk`

Loaded from the executable file

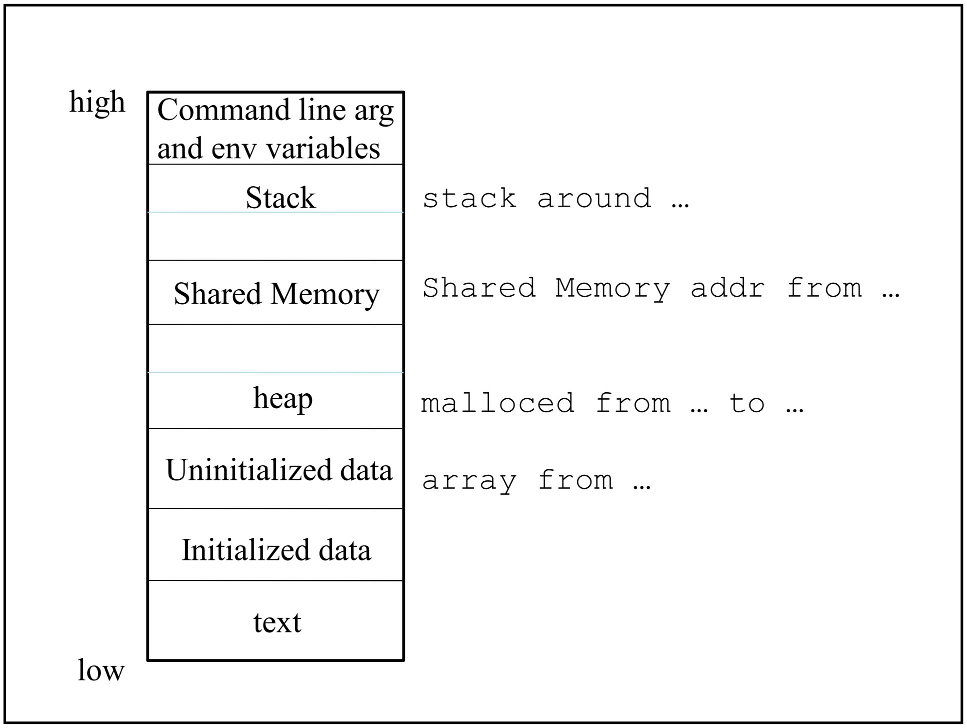
`min`

0

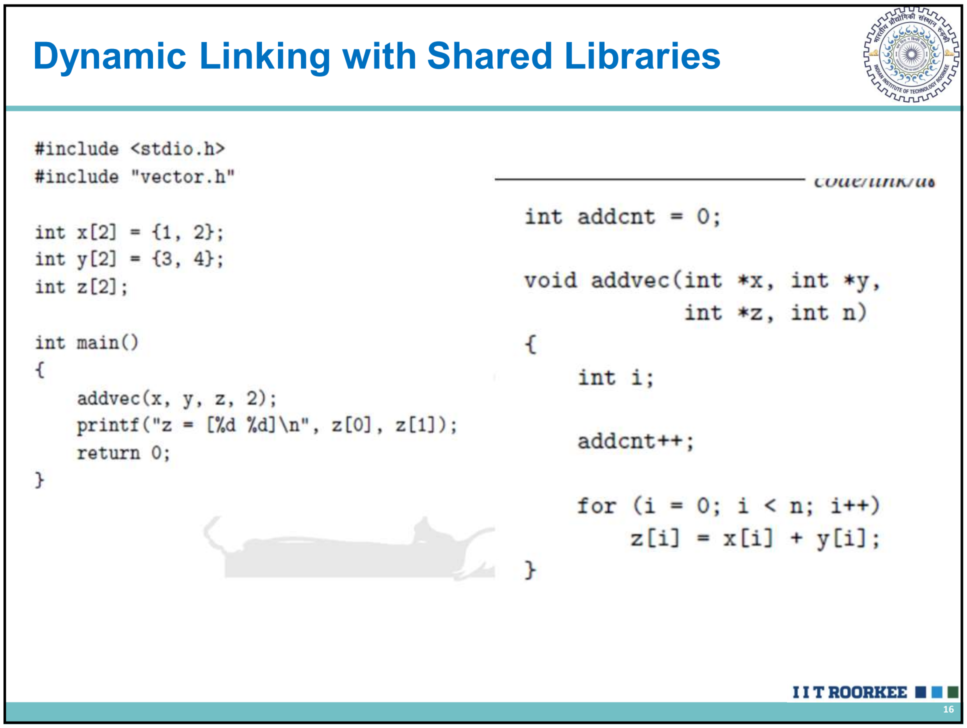


Loading Executable Object Files

```
char array[4096];

int main(int argc, char **argv)
{
        int     fd;
        char    *ptr;

        printf("array from %p\n", &array[0]);
        printf("stack around %p\n", &fd);

        ptr = (char *) malloc(10000);
        printf("malloced from %p to %p\n", ptr, ptr+10000);

        fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0777);
        lseek(fd, 4999, SEEK_SET);
        write(fd, "", 1);

        ptr = mmap(NULL,5000,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
        close(fd);

        printf("Shared Memory addr from %p\n", ptr);

        exit(0);
}
```



```
stack around …

Shared Memory addr from …

malloced from … to …

array from …
```



Dynamic Linking with Shared Libraries

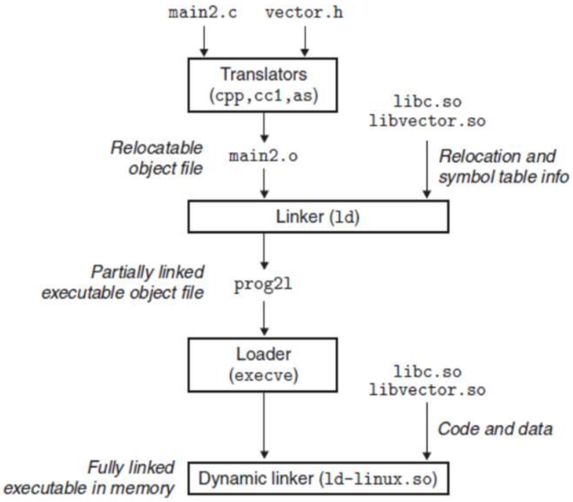
```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);
    return 0;
}
```

code/link/ad

```
int addcnt = 0;

void addvec(int *x, int *y,
            int *z, int n)
{
    int i;

    addcnt++;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

IIT ROORKEE

Dynamic linking process



do some of the linking statically when the executable file is created

then complete the linking process dynamically when the program is loaded.

Dynamic linking

Approach 1: Dynamic linker may load and link shared libraries when an application is loaded, just before it executes

Approach 2: Application requests the dynamic linker to load and link arbitrary shared libraries while the application is running

- void *dlopen(const char *filename, int flag);
- void *dlsym(void *handle, char *symbol);
- int dlclose (void *handle);