3.4 Linking & Loading

- *Static Linking & Loading*
- Shared Libraries

Remember This?

```
main:
  pushl %ebp
  movl %esp, %ebp
  pushl %esi
  pushl %edi
  subl $8, %esp
  ...
  pushl $1
  movl -12(%ebp), %eax
  pushl %eax
  call sub
  addl $8, %esp
  movl %eax, -16(%ebp)
  ...
  addl $8, %esp
  movl $0, %eax
  popl %edi
  popl %esi
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

push args

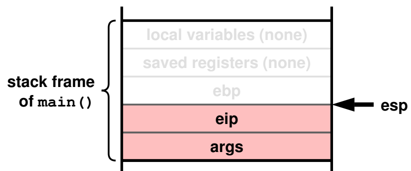
pop args; get result

set return value and restore frame


```
int main() {
  int i;
  int a;
  ...
  i = sub(a, 1);
  ...
  return(0);
}
```


Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```




```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

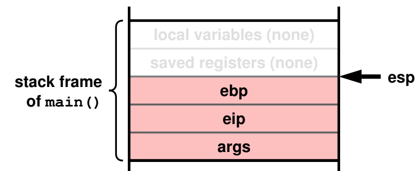
set up stack frame

set return value and restore frame

- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```




```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

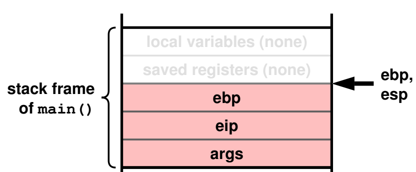
set up stack frame

set return value and restore frame

- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```




```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

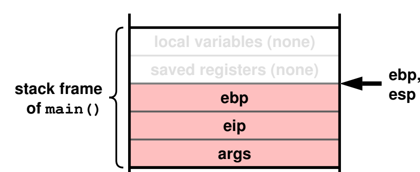
set up stack frame

set return value and restore frame

- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```




```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

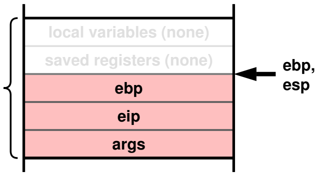
set return value and restore frame

- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}

main:
  pushl %ebp            } set up
  movl %esp, %ebp       } stack frame
  movl 8(%ebp), %eax
  movl %ebp, %esp       } set return
  popl %ebp             } value and
  ret                   } restore frame
```

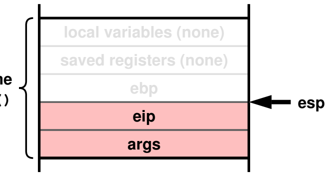


- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}

main:
  pushl %ebp            } set up
  movl %esp, %ebp       } stack frame
  movl 8(%ebp), %eax
  movl %ebp, %esp       } set return
  popl %ebp             } value and
  ret                   } restore frame
```

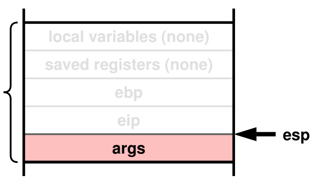


- Does location matter?

Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}

main:
  pushl %ebp            } set up
  movl %esp, %ebp       } stack frame
  movl 8(%ebp), %eax
  movl %ebp, %esp       } set return
  popl %ebp             } value and
  ret                   } restore frame
```



- Does location matter?
  - if everything can be accessed relative to the *frame pointer*, then you don't need to know the actual address of an object
    - just use *relative-addresses* to *access variables*
    - the *code* is also *location-independent*

Location Matters ...

```
int X = 6;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = X;
  subr(y);
  return(0);
}
```

```
void subr(int i) {
  printf("i = %d\n", i);
}
```

- Why does it matter here?
  - need to put the *address* of `X` into `aX`
    - *what* is the address of `X`?
      - *when* do you know?
    - remember, both `X` and `aX` are in the *data segment*
    - *who* would put the actual value into `aX`?
  - also need to put the *address* of `subr` into `main()`

Coping

- *Relocation*
  - modify internal references in memory depending on where module is expected to be *loaded*
    - one of the *exec* system calls loads a program into memory
      - everything is laid out carefully in memory
  - modules requiring relocation are said to be *relocatable*
  - the act of modifying such a module to *resolve these references* is called *relocation*
  - the program that performs relocation is called a *linker*

textbook
is wrong

- Two main functions of a *linker*
  1) *relocation*
  2) *symbol resolution*
- A *loader* loads a program into memory
  - a "relocating loader" may perform additional relocation

A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

- `main.c` is compiled into `main.o` } relocatable modules
- `subr.c` is compiled into `subr.o` }
- `ld` is then invoked to combine them into `prog`
  - `ld` knows where to find `printf()`
  - `prog` can be loaded into memory through one of the *exec* system calls

A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

- how does `ld` decides what needs to be done?
- `main.c` contains undefined references to `X` and `subr()`
  - instructions for doing this are provided in `main.o`
- later on, when the actual locations for these are determined, `ld` will modify them when `main.o` is ***copied*** into `prog`

A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

- `main.o` must contains a list of external symbols, along with their types, and instructions for updating this code

main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; what follows is
              ; text (read-only code)
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

What follows goes into the *data* segment

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

What follows goes into the *text* segment

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

offset got restarted because ***segments are relocatable***

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

`.global` directive means that the symbol mentioned is defined *here* and is *exported*
- i.e., can be referenced by other modules
- `aX` and `main` are global

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

`aX` is 4 bytes long and put the value of `X` here
- `X` will remain *unresolved*

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

these 3 places require *relocation*

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0:      .globl aX ; aX is global: it may be used
                  ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y on
19:     call  subr
24:     addl  $4,%esp ; remove y f
27:     movl  $0,%eax ; set return
31:     movl  %ebp, %esp ; restore
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

this call is a PC-relative call
- what's stored at offset 20 is not the absolute address of `subr`, but a relative address
- this is just how x86 works

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


subr.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS is required
                   ; for global X
4:
0:      .text ; offset restarts; what follows is
              ; text (read-only code)
0:      .globl subr
0: subr:
0:      pushl %ebp   ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp   ; pop frame pointer
23:     ret
```

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


subr.s

```
Offset  Op     Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp   ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp   ; pop frame pointer
23:     ret
```

this is how you create a string constant
- this one is 8 bytes long
- and local to this module (since it's not global)

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.s

```
Offset  Op      Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp  ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp  ; pop frame pointer
23:     ret
```

this is how you create a string constant
- this one is 8 bytes long
- and local to this module (since it's not global)
- it is used here

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


subr.s

```
Offset  Op      Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp  ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp  ; pop frame pointer
23:     ret
```

4 bytes is required in the *bss* segment for this global variable
- `.comm` directive also means that the symbol mentioned is defined *here* and is *exported*

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


subr.s

```
Offset  Op      Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp  ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp  ; pop frame pointer
23:     ret
```

`subr` is a global symbol exported from here

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


subr.s

```
Offset  Op      Arg
0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp  ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i on
6:      pushl $printfarg ; push ad
                         ; onto st
11:     call  printf
16:     addl  $8, %esp ; pop argum
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp  ; pop frame pointer
23:     ret
```

relocation is required for `printf` and `printfarg`

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


Object Files

- An object file describes what's in the data, bss, and text segments in separate sections
- Along with each section is a list of:
  - global symbols
  - undefined symbols
  - instructions for relocation
    - these instructions indicate
      - which locations within the section must be modified
      - which symbol's value is used to modify the location
    - a symbol's value is the address that is ultimately determined for it
    - typically, this address is added to the location being modified
- To inspect an object file on Unix
  - `nm` - list symbols from object files
  - `objdump` - display information from object files

subr.s

```
Offset  Op      Arg
0:      .data ; what follows is in
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X,4  ; 4 bytes in BSS
                   ; for global X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp  ; save the fra
1:      movl  %esp, %ebp ; point t
3:      pushl 8(%ebp)   ; push i onto stack
6:      pushl $printfarg ; push address of string
                         ; onto stack
11:     call  printf
16:     addl  $8, %esp ; pop arguments from stack
19:     movl  %ebp, %esp ; restore stack pointer
21:     popl  %ebp  ; pop frame pointer
23:     ret
```

relocation is required for:
- `printfarg` at offset 7
- `printf` at offset 12

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.o

```
Data:
  Size:  8
  Contents: "i = %d\n"

bss:
  Size:  4
  Global: X, offset 0

Text:
  Size:  24
  Global: subr, offset 0
  Undefined: printf
  Relocation:
    offset 7, size 4, value: addr of printfarg
    offset 12, size 4, value: PC-relative addr of
                              printf
  Contents: [machine instructions]
```

relocation is required for:
- `printfarg` at offset 7
- `printf` at offset 12

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


subr.o

```
Data:
  Size:  8
  Contents: "i = %d\n"

bss:
  Size:  4
  Global: X, offset 0

Text:
  Size:  24
  Global: subr, offset 0
  Undefined: printf
  Relocation:
    offset 7, size 4, value: addr of printfarg
    offset 12, size 4, value: PC-relative addr of
                              printf
  Contents: [machine instructions]
```

`X` and `subr` are *exported*
- needed in `main.o`

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


main.s

```
Offset  Op     Arg
0:      .data ; what follows is in
0:      .globl aX ; aX is global:
              ; by others
0: aX:
0:      .long  X
4:
0:      .text ; offset restarts; w
              ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the fram
1:      movl  %esp,%ebp ; point to
3:      subl  $4,%esp ; make space
6:      movl  aX,%eax ; put conten
11:     movl  (%eax),%eax ; put *X
13:     movl  %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y onto stack
19:     call  subr
24:     addl  $4,%esp ; remove y from stack
27:     movl  $0,%eax ; set return value to 0
31:     movl  %ebp, %esp ; restore stack pointer
33:     popl  %ebp  ; pop frame pointer
35:     ret
```

these 2 places remained *unresolved*

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


main.o

```
Data:
  Size:  4
  Global: aX, offset 0
  Undefined: X
  Relocation: offset 0, size 4, v
  Contents: 0x00000000

bss:
  Size:  0

Text:
  Size:  36
  Global: main, offset 0
  Undefined: subr
  Relocation:
    offset 7, size 4, value: addr of aX
    offset 20, size 4, value: PC-relative
                              addr of subr
  Contents: [machine instructions]
```

these 2 places remained *unresolved*
- they are noted in `main.o`

```c
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```


subr.o

```
Data:
  Size:  8
  Contents: "i = %d\n"

bss:
  Size:  4
  Global: X, offset 0

Text:
  Size:  24
  Global: subr, offset 0
  Undefined: printf
  Relocation:
    offset 7, size 4, value: addr of printfarg
    offset 12, size 4, value: PC-relative addr of
                              printf
  Contents: [machine instructions]
```

`printf` remained *unresolved*

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```


printf.o

```
Data:
  Size:  1024
  Global: StandardFiles
  Contents: ...

bss:
  Size:  256

Text:
  Size:  12000
  Global: printf, offset 100
  ...
  Undefined: write
  Relocation:
    offset 211, value: addr of StandardFiles
    offset 723, value: PC-relative addr of write
  Contents: [machine instructions]
```

assume that `printf.o` looks like this
- `write` is *unresolved*

write.o

```
Data:
  Size:  0

bss:
  Size:  4
  Global: errno, offset 0

Text:
  Size:  16
  Contents: [machine
             instructions]
```

and `write.o` looks like this

startup function

```
Data:
  Size:  0

bss:
  Size:  0

Text:
  Size:  36
  Undefined: main
  Relocation:
    offset 21, value: main
  Contents: [machine
             instructions]
```

every C program contains a *startup* routine that is called first

- it calls `main()`
- if `main()` returns, it calls `exit()`
- our example is incomplete

prog

```
Text
  main            4096
  subr            4132
  printf          4156
  write          16156
  startup        16172

Data
  aX             16384
  printfargs     16388
  StandardFiles  16396

BSS
  X              17420
  errno          17424
```

- this is how `ld` might set things up
  - `ld` *lays out* the *address space*
  - `ld` allocates memory in *pages* (typically 4KB each)
- `main` does not start at location 0
  - first "page" is typically made inaccessible so that references to null pointers will fail (get SIGSEG)

prog

```
Text
  main            4096
  subr            4132
  printf          4156
  write          16156
  startup        16172

Data
  aX             16384
  printfargs     16388
  StandardFiles  16396

BSS
  X              17420
  errno          17424
```

- due to the use of *"pages"*, the data segment needs to start at a page boundary (i.e., multiple of page size)
  - this way, the *text segment* can be made *read-only* while the *data* and *bss segments* made *read-write*
  - here we assume 4KB pages (therefore, pages start at 4096, 8192, 12288, 16384, etc.)

Virtual Memory Basics

- A process has, say, a 32-bit address space
  - that's 4GB of memory
- Our `prog` process, when it starts, only needs about 16KB for text+data+bss
  - plus more for stack
- Allocating 4GB of physical memory will be a *huge waste*
- Solution: *page table* to *map virtual to physical addresses*
  - OS allocate *pages* of *physical memory*
  - a page is 4KB in many systems
  - a page corresponds to physical memory that can be located *(or "mapped")* anywhere in virtual memory
  - one level of *indirection* to get to the physical memory
    - the hardware makes this transparent
- We will spend a lot of time talking about virtual memory (Ch 7)

Virtual Memory Basics

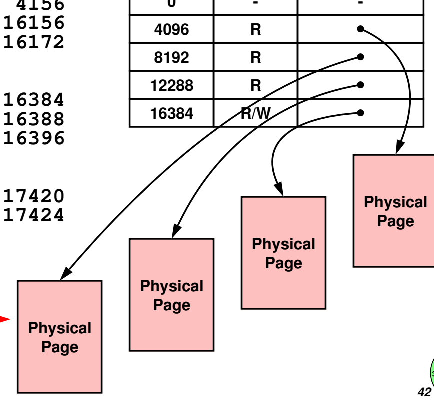
```
Text
  main            4096
  subr            4132
  printf          4156
  write          16156
  startup        16172

Data
  aX             16384
  printfargs     16388
  StandardFiles  16396

BSS
  X              17420
  errno          17424
```

Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	
8192	R	
12288	R	
16384	R/W	



ask buddy system to allocate these pages