C Primer

CS 351: Systems Programming
Michael Lee <lee@iit.edu>

Science | Computer Science

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

1. Overview
2. Basic syntax & structure
3. Compilation
4. Visibility & Lifetime

Agenda

5. Pointers & Arrays
6. Dynamic memory allocation
7. Composite data types

Not a Language Course!

- Resources:
  - K&R (*The C Programming Language*)
  - comp.lang.C FAQ (`c-faq.com`)
  - UNIX man pages (`kernel.org/doc/man-pages/`)

>man strlen

```
NAME
     strlen - find length of string

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     size_t
     strlen(const char *s);

DESCRIPTION
     The strlen() function computes the length of the string s.

RETURN VALUES
     The strlen() function returns the number of characters that precede the
     terminating NUL character.

SEE ALSO
     string(3)
```

§Overview

C is …

- imperative
- statically typed
- weakly type checked
- procedural
- low level

C	Java
Procedural	Object-oriented
Source-level portability	Compiled-code portability
Manual memory management	Garbage collected
Pointers reference addresses	Opaque memory references
Manual error code checking	Exception handling
Manual namespace partitioning	Namespaces with packages
Small, low-level libraries	Vast, high-level class libraries

§Basic syntax & structure

Primitive Types

- `char`: one byte integer (e.g., for ASCII)
- `int`: integer, *at least* 16 bits
- `float`: single precision floating point
- `double`: double precision floating point

Integer type prefixes

- `signed` (default), `unsigned`
  - same storage size, but sign bit on/off
- `short`, `long`
  - sizeof (`short int`) ≥ 16 bits
  - sizeof (`long int`) ≥ 32 bits
  - sizeof (`long long int`) ≥ 64 bits

Recall C's weak type-checking…

```
/* types are implicitly "converted" */
char  c  = 0x41424344;
short s  = 0x10001000;
int   i  = 'A';
unsigned int u = -1;

printf("'%c', %d, %X, %X\n", c, s, i, u);
```

```
'D', 4096, 41, FFFFFFFF
```

Basic Operators

- Arithmetic: +, -, *, /, %, ++, --, &, |, ~
- Relational: <, >, <=, >=, ==, !=
- Logical: &&, ||, !
- Assignment: =, +=, *=, …
- Conditional: *bool* **?** *true_exp* **:** *false_exp*

True/False

- 0 = False
- **Everything else** = True
  - But *canonical* True = 1

Boolean Expressions

```
!(0)             → 1
0 || 2           → 1
3 && 0 && 6      → 0
!(1234)          → 0
!!(-1020)        → 1
```

Control Structures

- if-else
- switch-case
- while, for, do-while
  - continue, break

Variables

- Must declare before use
- Declaration implicitly **allocates** storage for underlying data
  - Note: not true in Java!

Functions

- C's *top-level* modules
- Procedural language vs. OO: no classes!

Declaration vs. *Definition*

- *Declaration* (aka *prototype*): arg & ret type
- *Definition*: function body
- A function can be *declared many times* but only *defined once*

Declarations reside in *header* (.h) files,
Definitions reside in *source* (.c) files

(Suggestions, not really requirements)

hashtable.h

```
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void  ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void  ht_del(hashtable_t *ht, char *key);
void  ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void  ht_rehash(hashtable_t *ht, unsigned long newsize);
int   ht_max_chain_length(hashtable_t *ht);
void  free_hashtable(hashtable_t *ht);
```

← "API"

hashtable.c

```
#include "hashtable.h"

unsigned long hash(char *str) {
  unsigned long hash = 5381;
  int c;
  while ((c = *str++))
    hash = ((hash << 5) + hash) + c;
  return hash;
}

hashtable_t *make_hashtable(unsigned long size) {
  hashtable_t *ht = malloc(sizeof(hashtable_t));
  ht->size = size;
  ht->buckets = calloc(sizeof(bucket_t *), size);
  return ht;
}

...
```

hashtable.h

```
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void  ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void  ht_del(hashtable_t *ht, char *key);
void  ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void  ht_rehash(hashtable_t *ht, unsigned long newsize);
int   ht_max_chain_length(hashtable_t *ht);
void  free_hashtable(hashtable_t *ht);
```

← "API"

main.c

```
#include "hashtable.h"

int main(int argc, char *argv[]) {
  hashtable_t *ht;
  ht = make_hashtable(atoi(argv[1]));
  ...
  free_hashtable(ht);
  return 0;
}
```

§Compilation

main.c

```c
#include <stdio.h>

int main () {
    printf("Hello world!\n");
    return 0;
}
```

```
$ gcc main.c -o prog
$ ./prog
Hello world!
```

greet.h

```c
void greet(char *);
```

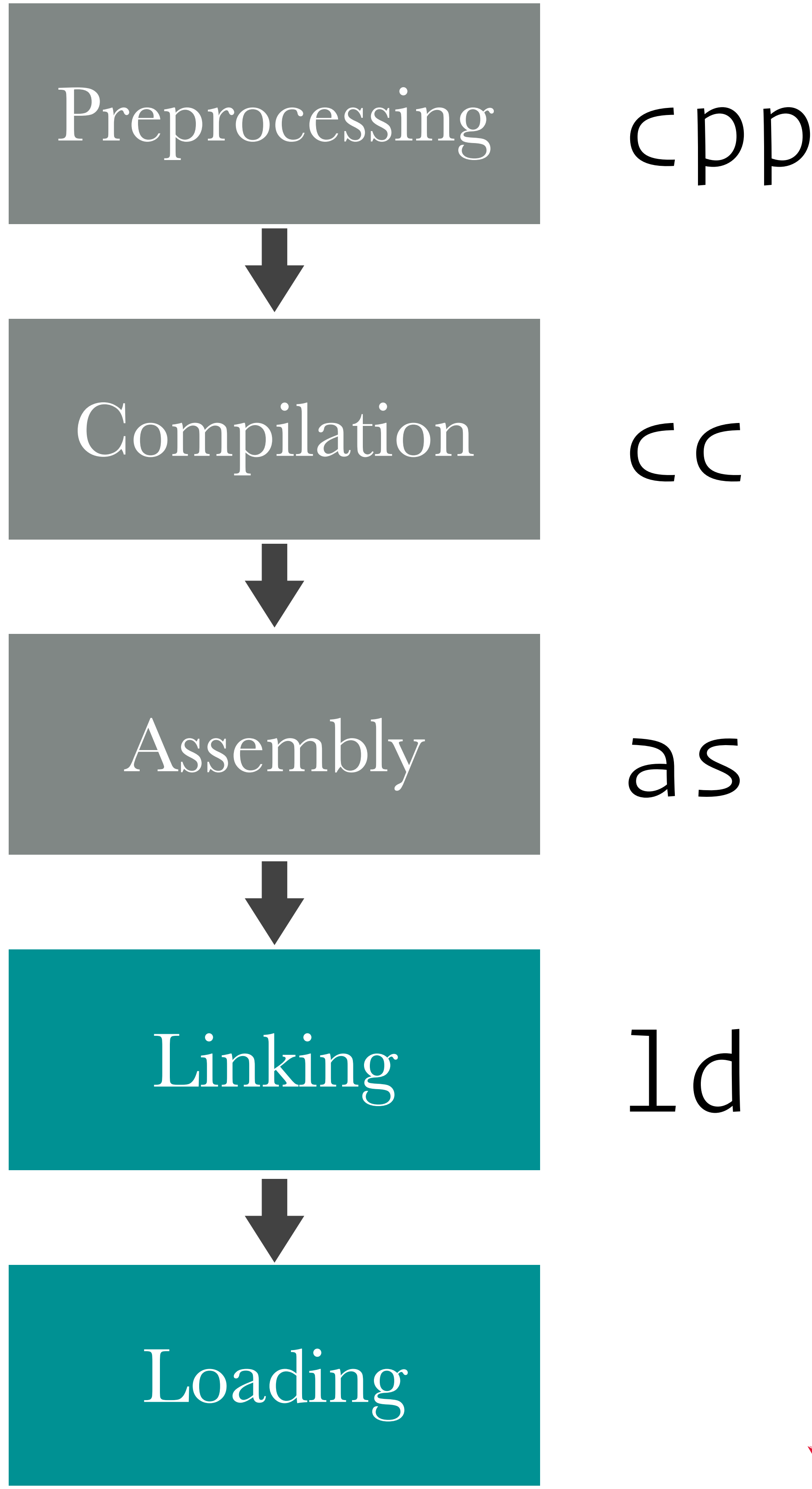
greet.c

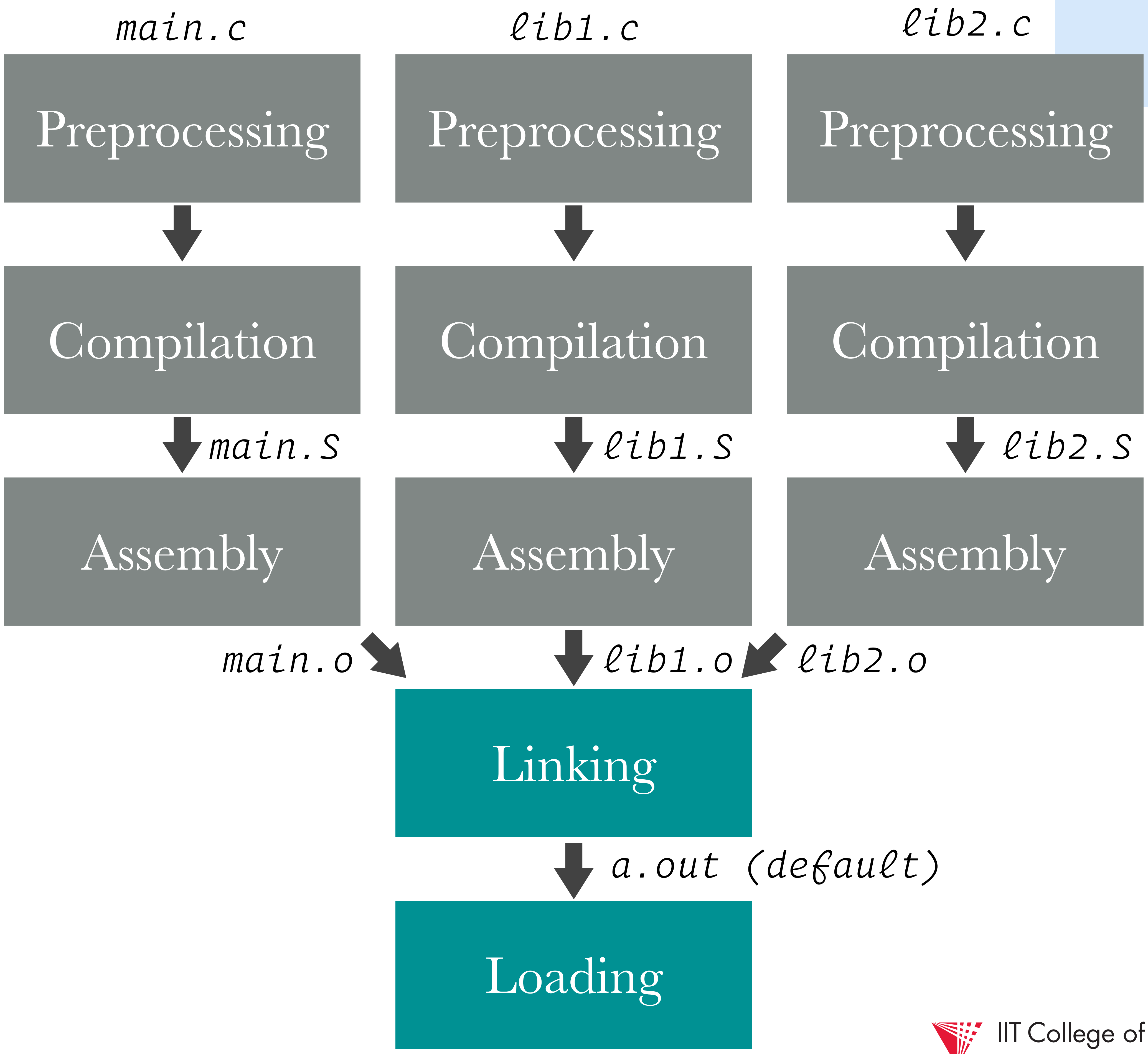
```c
#include <stdio.h>
#include "greet.h"

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

```c
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c      -o greet.o
$ gcc -c main.c       -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
```


Preprocessing
cpp
Compilation
cc
Assembly
as
Linking
ld
Loading


main.c
lib1.c
lib2.c
Preprocessing
Preprocessing
Preprocessing
Compilation
Compilation
Compilation
main.S
lib1.S
lib2.S
Assembly
Assembly
Assembly
main.o
lib1.o
lib2.o
Linking
a.out (default)
Loading

Preprocessing

↓

Compilation

↓

Assembly

↓

Linking

↓

Loading

“Preprocessing”

- preprocessor *directives* exist for:
  - text substitution
  - macros
  - conditional compilation
- directives start with ‘#’

greet.h

```c
void greet(char *);
```

greet.c

```c
#include "greet.h"

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

stop and show source after preprocessing stage

```
$ gcc -E greet.c

void greet(char *);

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

```c
#define msg "Hello world!\n"

int main () {
    printf(msg);
    return 0;
}
```

```
$ gcc -E hello.c

int main () {
    printf("Hello world!\n");
    return 0;
}
```

```c
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

```
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
#define PLUS1(x) x+1

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

same effect?

```
$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

```
$ gcc -E plus1b.c

int main () {
    int y;
    y = y * y+1;
    return 0;
}
```

no!

macros *blindly* manipulate *text*!

```c
int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
#ifdef VERBOSE
    printf("Debugging: %d\n", f0);
#endif
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

create preprocessor definition

```
$ gcc -E fib.c

int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

```
$ gcc -D VERBOSE -E fib.c

int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
    printf("Debugging: %d\n", f0);
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

“Linking”

- Resolving symbolic references (e.g., variables, functions) to their definitions
  - e.g., by placing final target addresses in `jump/call` instructions
- Both *static* and *dynamic* linking are possible; the latter is performed at run-time

greet.h

```c
void greet(char *);
```

greet.c

```c
#include <stdio.h>
#include "greet.h"

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

```c
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
```

```
$ objdump -d main.o
0000000000000000 <main>:
   0:    55                           push   %rbp
   1:    48 89 e5                     mov    %rsp,%rbp
   4:    bf 00 00 00 00               mov    $0x0,%edi
   9:    e8 00 00 00 00               callq  e <main+0xe>
   e:    b8 00 00 00 00               mov    $0x0,%eax
  13:    5d                           pop    %rbp
  14:    c3                           retq
```

placeholder addresses

```
$ objdump -d greet.o
0000000000000000 <greet>:
   0:    55                           push   %rbp
   1:    48 89 e5                     mov    %rsp,%rbp
   4:    48 83 ec 10                  sub    $0x10,%rsp
   8:    48 89 7d f8                  mov    %rdi,-0x8(%rbp)
   c:    48 8b 45 f8                  mov    -0x8(%rbp),%rax
  10:    48 89 c6                     mov    %rax,%rsi
  13:    bf 00 00 00 00               mov    $0x0,%edi
  18:    b8 00 00 00 00               mov    $0x0,%eax
  1d:    e8 00 00 00 00               callq  22 <greet+0x22>
  22:    90                           nop
  23:    c9                           leaveq
  24:    c3                           retq
```

greet.h

```
void greet(char *);
```

greet.c

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

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

```
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
```

```
$ objdump -d prog
00000000004003f0 <printf@plt-0x10>:
  4003f0:  ff 35 12 0c 20 00    pushq  0x200c12(%rip)  # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  4003f6:  ff 25 14 0c 20 00    jmpq   *0x200c14(%rip) # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  4003fc:  0f 1f 40 00          nopl   0x0(%rax)

0000000000400400 <printf@plt>:
  400400:  ff 25 12 0c 20 00    jmpq   *0x200c12(%rip) # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  400406:  68 00 00 00 00       pushq  $0x0
  40040b:  e9 e0 ff ff ff       jmpq   4003f0 <_init+0x28>

0000000000400526 <main>:
  400526:  55                   push   %rbp
  400527:  48 89 e5             mov    %rsp,%rbp
  40052a:  bf e4 05 40 00       mov    $0x4005e4,%edi
  40052f:  e8 07 00 00 00       callq  40053b <greet>
  400534:  b8 00 00 00 00       mov    $0x0,%eax
  400539:  5d                   pop    %rbp
  40053a:  c3                   retq

000000000040053b <greet>:
  40053b:  55                   push   %rbp
  40053c:  48 89 e5             mov    %rsp,%rbp
  40053f:  48 83 ec 10          sub    $0x10,%rsp
  400543:  48 89 7d f8          mov    %rdi,-0x8(%rbp)
  400547:  48 8b 45 f8          mov    -0x8(%rbp),%rax
  40054b:  48 89 c6             mov    %rax,%rsi
  40054e:  bf ec 05 40 00       mov    $0x4005ec,%edi
  400553:  b8 00 00 00 00       mov    $0x0,%eax
  400558:  e8 a3 fe ff ff       callq  400400 <printf@plt>
  40055d:  90                   nop
  40055e:  c9                   leaveq
  40055f:  c3                   retq
```

“Linking”

- I.e., the linker allows us to create large, multi-file programs with complex variable/function cross-referencing
- Pre-compiled libraries can be “linked in” (statically or dynamically) without rebuilding from source

"Linking"

- But, we don't always *want* to allow linking a call to a definition!
  - e.g., to hide implementations and build *selective* public APIs

§Visibility & Lifetime

Visibility: *where* can a symbol (var/fn) be seen from, and how do we refer to it?

Lifetime: *how long* does allocated storage space (e.g., for a var) remain useable?

sum.c

```c
int sumWithI(int x, int y) {
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
sum.c: In function `sumWithI':
sum.c:2: error: `I' undeclared (first use in this function)
main.c: In function `main':
main.c:6: warning: implicit declaration of function `sumWithI'
```

sum.c

```c
int sumWithI(int x, int y) {
    int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
-1073743741
```

problem: variable *declaration* & *definition* are implicitly tied together

note: definition = *storage allocation* + possible *initialization* `extern` keyword allows for declaration *sans definition*

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
```

… and now global variables are visible from *everywhere*.

Good/Bad?

`static` keyword lets us limit the *visibility* of things

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

static int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_I", referenced from:
      _sumWithI in ccmvi0RF.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

sum.c

```c
static int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_sumWithI", referenced from:
      _main in cc9LhUBP.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

`static` also forces the *lifetime* of variables to be equivalent to `global`

(i.e., stored in static memory vs. stack)

sum.c

```c
int sumWithI(int x, int y) {
    static int I = 10; // init once
    return x + y + I++;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int main() {
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
14
15
```

§Pointers

(don’t panic!)

a *pointer* is a variable declared
to store a *memory address*

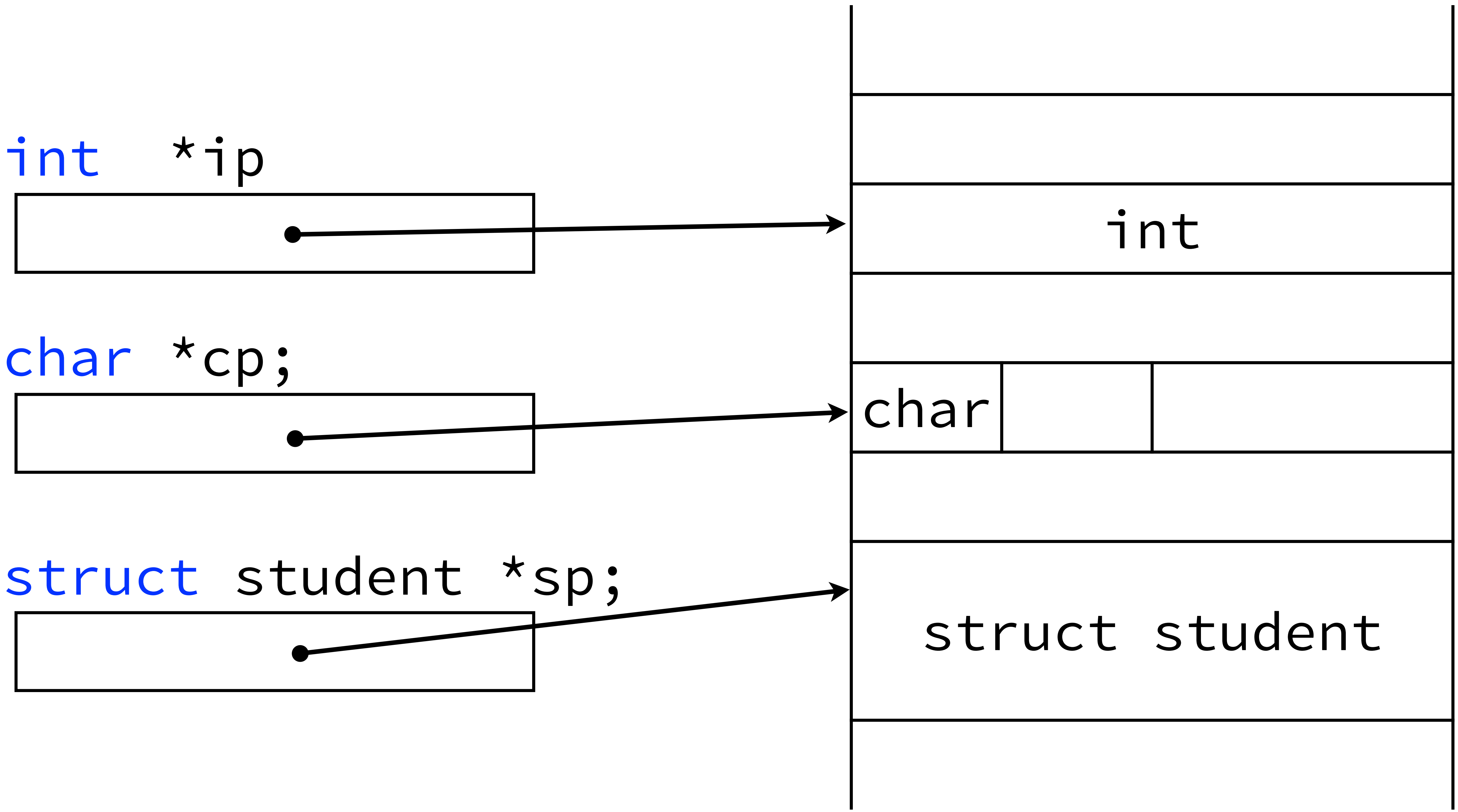
Q: by examining a variable's contents, can we tell if the variable is a pointer?

e.g., `0x0040B100`

No!

- a pointer is designated by its *static (declared) type*, not its contents

A pointer declaration also tells us the *type of data to which it should point* declaration syntax: `type *var_name`


int *ip
char *cp;
struct student *sp;
int
char
struct student

Important pointer-related operators:

& : address-of

* : dereference (*not the same as the * used for declarations!!!*)

```
int i = 5;   /* i is an int containing 5 */
int *p;      /* p is a pointer to an int */

p = &i;      /* store the address of i in p */

int j;       /* j is an uninitialized int */
j = *p;      /* store the value p points to into j*/
```

```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
    return 0;
}
```

```
$ gcc pointers.c
$ ./a.out
i=10, j=20, *p=20, *q=20
```

```
int i, j, *p, *q;
i = 10;
```

Address	Data	
1000	10	(i)
1004	?	(j)
1008	?	(p)
1012	?	(q)

```
p = &j;
```

Address	Data	
1000	10	(i)
1004	?	(j,*p)
1008	1004	(p)
1012	?	(q)

```
q = p;
```

Address	Data	
1000	10	(i)
1004	?	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```
*q = i;
```

Address	Data	
1000	10	(i)
1004	10	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```
*p = *q * 2;
```

Address	Data	
1000	10	(i)
1004	20	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    return 0;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $10, -4(%rbp)
        leaq    -28(%rbp), %rax
        movq    %rax, -16(%rbp)
        movq    -16(%rbp), %rax
        movq    %rax, -24(%rbp)
        movq    -24(%rbp), %rax
        movl    -4(%rbp), %edx
        movl    %edx, (%rax)
        movq    -24(%rbp), %rax
        movl    (%rax), %eax
        leal    (%rax,%rax), %edx
        movq    -16(%rbp), %rax
        movl    %edx, (%rax)
        movl    $0, %eax
        popq    %rbp
        ret
```

(via Compiler Explorer: https://godbolt.org)

why have pointers?

```
int main() {
    int a = 5, b = 10;
    swap(a, b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```

pointers enable *action at a distance*

```
void bar(int *p) {
    *p = ...; /* change some remote var! */
}

void bat(int *p) {
    bar(p);
}

void baz(int *p) {
    bat(p);
}

int main() {
    int i;
    baz(&i);
    return 0;
}
```

action at a distance is an *anti-pattern*

i.e., an oft used but typically crappy programming solution

back to swap

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}
```

… for swapping pointers?

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c, *d;
    c = &a;
    d = &b;

    swap(c, d);
    /* want c to point to b, d to a */
    ...
}
```

```c
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;

    swap(&c, &d);
    /* want c to point to b, d to a */
}
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:10: warning: passing argument 1 of ‘swap’ from
incompatible pointer type
pointers.c:10: warning: passing argument 2 of ‘swap’ from
incompatible pointer type
```

```
void swapp(int **p, int **q) {
    int *tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;

    swapp(&c, &d);
    /* want c to point to b, d to a */
}
```

`(int **)` declares a *pointer to a pointer* to an `int`

Uninitialized pointers

- are like all other uninitialized variables
  - i.e., contain **garbage**
- dereferencing garbage ...
  - if lucky → crash
  - if unlucky → ???

“Null” pointers

- never returned by `&` operator
- safe to use as sentinel value
- written as `0` in *pointer context*
  - for convenience, `#define`’d as `NULL`

“Null” pointers

```
int main() {
    int i  = 0;
    int *p = NULL;

    ...

    if (p) {
        /* (likely) safe to deref p */
    }
}
```

§Arrays

contiguous, indexed region of memory

Declaration: `type arr_name[size]`

- remember, declaration also allocates storage!

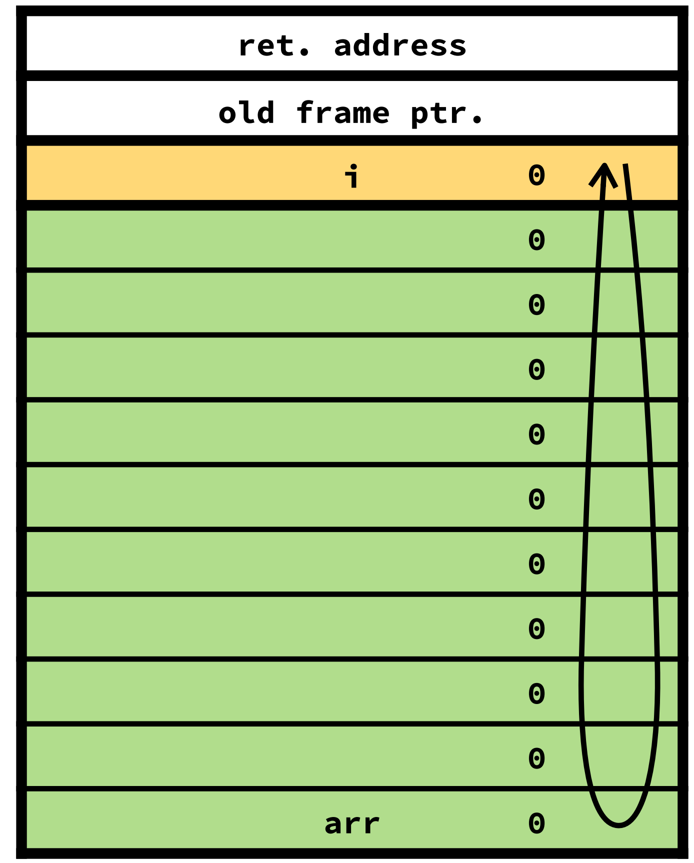
```c
int  i_arr[10];         /* array of 10 ints */
char c_arr[80];         /* array of 80 chars */
char td_arr[24][80];    /* 2-D array, 24 rows x 80 cols */
int  *ip_arr[10];       /* array of 10 pointers to ints */

/* dimension can be inferred if initialized when declaring */
short grades[] = { 75, 90, 85, 100 };

/* can only omit first dim, as partial initialization is ok */
int sparse[][10] = { { 5, 3, 2 },
                     { 8, 10 },
                     { 2 } };

/* if partially initialized, remaining components are 0 */
int zeros[1000] = { 0 };

/* can also use designated initializers for specific indices*/
int nifty[100] = { [0]  = 0,
                   [99] = 1000,
                   [49] = 250 };
```

In C, arrays contain *no metadata*

i.e., ***no*** *implicit size*, ***no*** *bounds checking*

```c
int main() {
    int i, arr[10];

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }
    printf("Done\n");

    return 0;
}
```

stack




```
$ gcc arr.c
$ ./a.out
                                    (runs forever ... no output)
```

```c
int main() {
    int arr[10], i;

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }
    printf("Done\n");

    return 0;
}
```


```
$ gcc arr.c
$ ./a.out
Done
[1]    10287 segmentation fault  ./a.out
$
```

this is the basis of *buffer overrun* attacks!

what can you do with stack manipulation?

- code injection
- return redirection
- et al

direct access to memory can be *dangerous*!

pointers ♥ arrays

- an array name is bound to the address of its first element
  - i.e., array name is a *const pointer*
- conversely, a pointer can be used as though it were an array name

```c
int *pa;
int arr[5];

pa = &arr[0];  /* <=> */  pa = arr;

arr[i];        /* <=> */  pa[i];

*arr;          /* <=> */  *pa;
```

```c
int i;

pa  = &i;     /* ok */

arr = &i;     /* not possible! */
```

§Pointer Arithmetic

follows naturally from allowing array subscript notation on pointers

```c
int arr[100];

int *pa = arr;

pa[10] = 0;       /* set tenth element */

/* so it follows ... */

*(pa + 10) = 0;  /* set tenth element */

/* surprising! "adding" to a pointer
   accounts for element size -- does not
   blindly increment address */
```

```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

pa[10] = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBEEF
```

```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof (int);

*(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBE00
```

`sizeof`: an operator to get the size *in bytes*

- can be applied to a datum or type

```
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof (int);

*(int *)(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
0
```

takeaway:

- pointer arithmetic makes use of pointee data types to compute byte offsets

strings are just `0` terminated char arrays

```c
char str[]      = "hello!";
char *p         = "hi";
char tarr[][5] = {"max", "of", "four"};
char *sarr[]    = {"variable", "length", "strings"};
```

```c
/* printing a string (painfully) */

int i;
char *str = "hello world!";
for (i = 0; str[i] != 0; i++) {
    printf("%c", str[i]);
}


/* or just */

printf("%s", str);
```

```c
/* Beware: */

int main() {
    char *str = "hello world!";
    str[12] = 10;
    printf("%s", str);
    return 0;
}
```

```
$ ./a.out
[1]    22432 segmentation fault (core dumped)  ./a.out
```

```c
/* the fleshed out "main" with command-line args */

int main(int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++) {
        printf("%s", argv[i]);
        printf("%s", ((i < argc-1)? ", " : "\n") );
    }
    return 0;
}
```

```
$ ./a.out testing one two three
./a.out, testing, one, two, three
```

§Dynamic Memory Allocation

dynamic vs. *static* (lifetime = forever)

vs. *local* (lifetime = LIFO)

C requires *explicit* memory management

- must request & free memory manually

- if forget to free → memory **leak**

vs., e.g., Java, which has *implicit* memory management via *garbage collection*

- allocate (via `new`) & forget!

basic C "malloc" API (in stdlib.h):

- `malloc`
- `realloc`
- `free`

malloc lib is *type agnostic*

i.e., it doesn't care what data types we store in requested memory need a “generic” / type-less pointer:

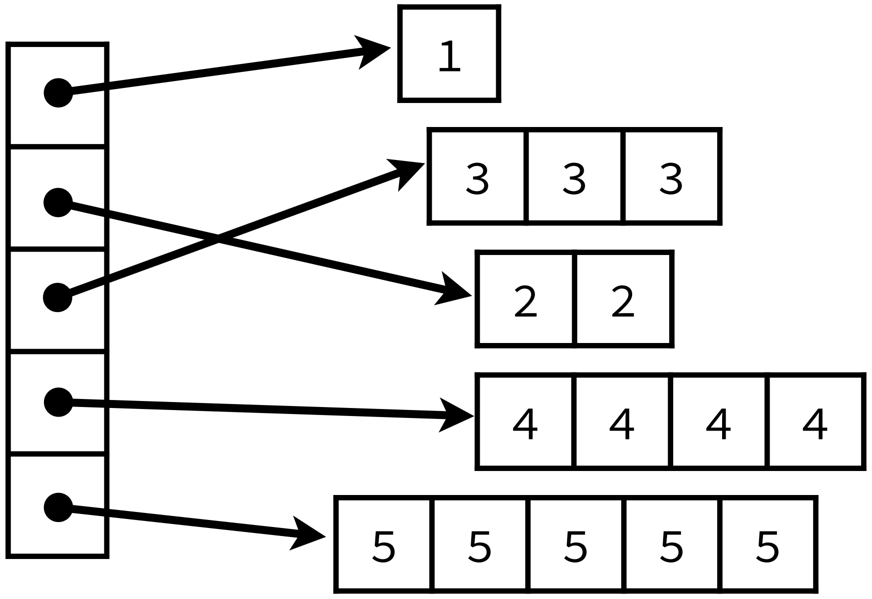
```
(void *)
```

```c
void *malloc(size_t size);

void *realloc(void *ptr, size_t size);

void  free(void *ptr);
```

all `sizes` are in bytes;

all `ptrs` are from previous malloc requests

```c
int i, j, k=1;
int *jagged_arr[5]; /* array of 5 pointers to int */
for (i=0; i<5; i++) {
    jagged_arr[i] = malloc(sizeof(int) * k);
    for (j=0; j<k; j++) {
        jagged_arr[i][j] = k;
    }
    k += 1;
}

/* use jagged_arr ... */

for (i=0; i<5; i++) {
    free(jagged_arr[i]);
}
```



§Composite Data Types

$\approx$ objects in OOP

C `struct`s create user defined types, based on primitives (and/or other UDTs)

```c
/* type definition */
struct point {
    int x;
    int y;
}; /* the end ';' is required */

/* point declaration (& alloc!) */
struct point pt;

/* pointer to a point */
struct point *pp;
```

```c
/* combined definition & decls */
struct point {
    int x;
    int y;
} pt, *pp;
```

component access: dot ('.') operator

```
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pt.x = 10;
    pt.y = -5;

    struct point pt2 = { .x = 8, .y = 13 }; /* decl & init */

    pp = &pt;

    (*pp).x = 351; /* comp. access via pointer */

    ...
}
```

```
(*pp).x = 351;  ≟  *pp.x = 351;
```

‘.’ has higher precedence than ‘*’

```
$ gcc point.c
... error: request for member ‘x’ in  something not a
           structure or union
```

But `(*pp).x` is painful

So we have the ‘**->**’ operator

- component access via pointer

```
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pp = &pt;
    pp->x = 10;
    pp->y = -5;

    ...
}
```

```c
/* Dynamically allocating structs: */

struct point *parr1 = malloc(N * sizeof(struct point));
for (i=0; i<N; i++) {
    parr1[i].x = parr1[i].y = 0;
}

/* or, equivalently, with calloc (which zero-inits) */
struct point *parr2 = calloc(N, sizeof(struct point));


/* do stuff with parr1, parr2 ... */

free(parr1);
free(parr2);
```

In C *all* args are *pass-by-value*!

```
void foo(struct point pt) {
    pt.x = pt.y = 10;
}

int main() {
    struct point mypt = { .x = 5, .y = 15 };
    foo(mypt);
    printf("(%d, %d)\n", mypt.x, mypt.y);
    return 0;
}
```

```
(5, 15)
```

```c
/* self referential struct */
struct ll_node {
  char *data;
  struct ll_node next;
};
```

```
$ gcc ll.c
ll.c:4: error: field ‘next’ has incomplete type
```

problem: compiler can’t compute size of `next` — depends on size of `ll_node`, which depends on size of `next`, etc.

```c
/* self referential struct */
struct ll_node {
  char *data;
  struct ll_node *next; /* need a pointer! */
};

struct ll_node *prepend(char *data, struct ll_node *next) {
  struct ll_node *n = malloc(sizeof(struct ll_node));
  n->data = data;
  n->next = next;
  return n;
}

void free_llist(struct ll_node *head) {
  struct ll_node *p=head, *q;
  while (p) {
      q = p->next;
      free(p);
      p = q;
  }
}
```

```
main() {
  struct ll_node *head = 0;

  head = prepend("reverse.", head);
  head = prepend("in", head);
  head = prepend("display", head);
  head = prepend("will", head);
  head = prepend("These", head);

  struct ll_node *p;
  for (p=head; p; p=p->next) {
    printf("%s ", p->data);
  }
  printf("\n");

  free_llist(head);
}
```

```
These will display in reverse.
```

very handy tool for detecting/debugging memory leaks: **valgrind**

```
main() {
  struct ll_node *head = 0;

  head = prepend("reverse.", head);
  ...

  // free_llist(head);
}
```

```
# valgrind --leak-check=full ./12c-dma
==308== HEAP SUMMARY:
==308==     in use at exit: 80 bytes in 5 blocks
==308==   total heap usage: 6 allocs, 1 frees, 1,104 bytes allocated
==308==
==308== 80 (16 direct, 64 indirect) bytes in 1 blocks are definitely lost
==308==    at 0x483B7F3: malloc
==308==    by 0x1091C6: prepend (12c-dma.c:20)
==308==    by 0x1092AF: main (12c-dma.c:42)
==308==
==308== LEAK SUMMARY:
==308==    definitely lost: 16 bytes in 1 blocks
==308==    indirectly lost: 64 bytes in 4 blocks
```

```
void free_llist(struct ll_node *head) {
  struct ll_node *p=head, *q;
  while (p) {
      //q = p->next;
      free(p);
      p = p->next;
  }
}
```

```
main() {
  struct ll_node *head = 0;

  head = prepend("reverse.", head);
  ...

  free_llist(head);
}
```

```
# valgrind --leak-check=full ./12c-dma
==322== Invalid read of size 8
==322==    at 0x109212: free_llist (12c-dma.c:31)
==322==  Address 0x4a47188 is 8 bytes inside a block of size 16 free'd
==322==    by 0x10920D: free_llist (12c-dma.c:30)
==322==  Block was alloc'd at
==322==    by 0x1091C6: prepend (12c-dma.c:20)
==322==
==322== HEAP SUMMARY:
==322==     in use at exit: 0 bytes in 0 blocks
==322==   total heap usage: 6 allocs, 6 frees, 1,104 bytes allocated
==322==
==322== All heap blocks were freed -- no leaks are possible
```

</C_Primer>