C Primer part 3

Science

Computer Science

CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Dynamic Memory Allocation

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` (size) -- allocate a chunk of memory
- `calloc` (size) -- zero allocated memory
- `realloc` (pointer, new size) -- get more/less space with copied contents
- `free` (pointer) -- releases memory

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 *)
```

assigning from/to `(void *)` to/from any other pointer *will never produce warnings*

… Hurrah! (but *dangerous*)

```
void *malloc(size_t size);

void *calloc(size_t size);

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

void  free(void *ptr);
```

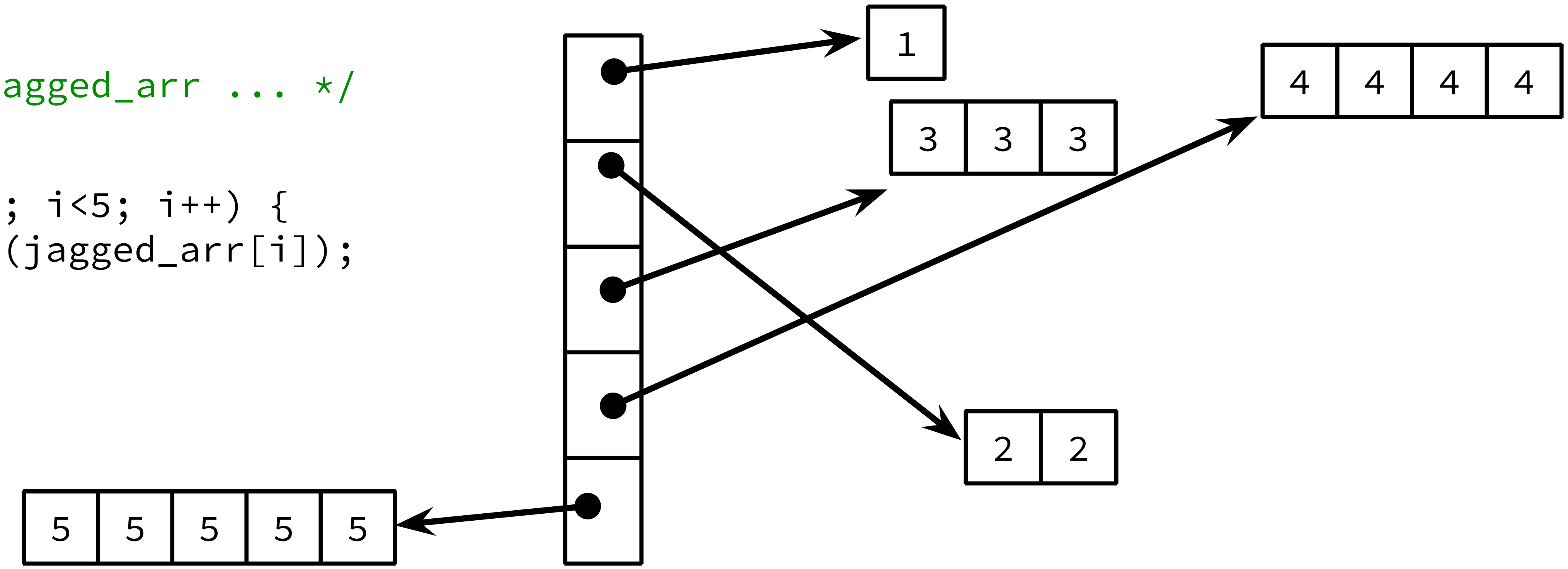
all **sizes** are in bytes

all **ptrs** are from previous malloc requests

```
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]);
}
```



Adjacency between arrays is NOT guaranteed!

```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;
}
```

```
(gdb) p jagged_arr
$1 = {0x1001000e0, 0x100103ad0, 0x100103ae0, 0x100103af0, 0x100103b00}
(gdb) p jagged_arr[0][0]
$2 = 1
(gdb) p *jagged_arr[0]
$3 = 1
(gdb) p *(int (*) [5])jagged_arr[4]
$4 = {5, 5, 5, 5, 5}
```

Composite Data Types

≈ objects in OOP

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

```
/* 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;
```

component access: dot ('.') operator

```
struct point {
    int x;
    int y;
};
struct point 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;` $\xcancel{\overset{?}{=}}$ `*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;

    ...
}
```

```
/* 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);
```

`sizeof` works with `struct`s, too, but with sometimes surprising results:

```
struct point {
    int  x;
    int  y;
};

struct foo {
    char name[10];
    int  id;
    char flag;
};
```

```
point size       = 8
point comps size = 8
foo size         = 20
foo comps size   = 15
```

(padding!)

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.

```
/* 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.
```

Function pointers

```c
int square(int x) {
    return x * x;
}

int cube(int x) {
    return x * x * x;
}


int main() {
    int (*f)(int) = square;
    printf("%d\n", (*f)(10));

    f = cube;
    printf("%d\n", (*f)(10));
    return 0;
}
```

Can be useful!
Also can be hard to read.
Use with care!

```
100
1000
```

typedef

declarations can get a little … wordy

- unsigned long int size;
- void (*fn)(int);
- struct llnode *lst;

`typedef` lets us create an *alias* for an existing type syntax:

```
typedef oldtype newtype;
```

- looks like a regular variable declaration to the right of the `typedef` keyword

```c
/* declare `int_t` as an alias for `int` */
typedef int int_t;

main() {
    int i;
    int_t j;
    i = j = 10;
    printf("%d, %d, %lu, %lu",
           i, j, sizeof(int), sizeof(int_t));
}
```

```
10, 10, 4, 4
```

```
/* declare `intp_t` as an alias for `int *` */
typedef int *intp_t;

main() {
    int i;
    intp_t p;
    p = &i;
}
```

```
/* define both preceding aliases */
typedef int int_t, *intp_t;

main() {
    int_t i;
    intp_t p;
    p = &i;
}
```

```
/* common integer aliases (see stdint.h) */

/* used to store "sizes" and "offsets" */
typedef unsigned long int size_t;
typedef long int          off_t;

/* for small numbers; 8 bits only */
typedef signed char       int8_t;
typedef unsigned char     uint8_t;

/* for large numbers; 64 bits */
typedef long int          int64_t;
typedef unsigned long int uint64_t;
```

```
/* fn pointer typedef */
typedef int (*handler_t)(int);

int kfn_menu(int duration) { /* ... */ }

main() {
    handler_t fp = kfn_menu;
    int ret = (*fp)(0);
    ...
}
```

```c
/* linked-list type aliases */
typedef struct ll_node node, *node_p, *list;

struct ll_node {
    void *val;
    node_p next;
};


main() {
    node n = { .val = NULL, .next = NULL };
    list l = &n;
}
```

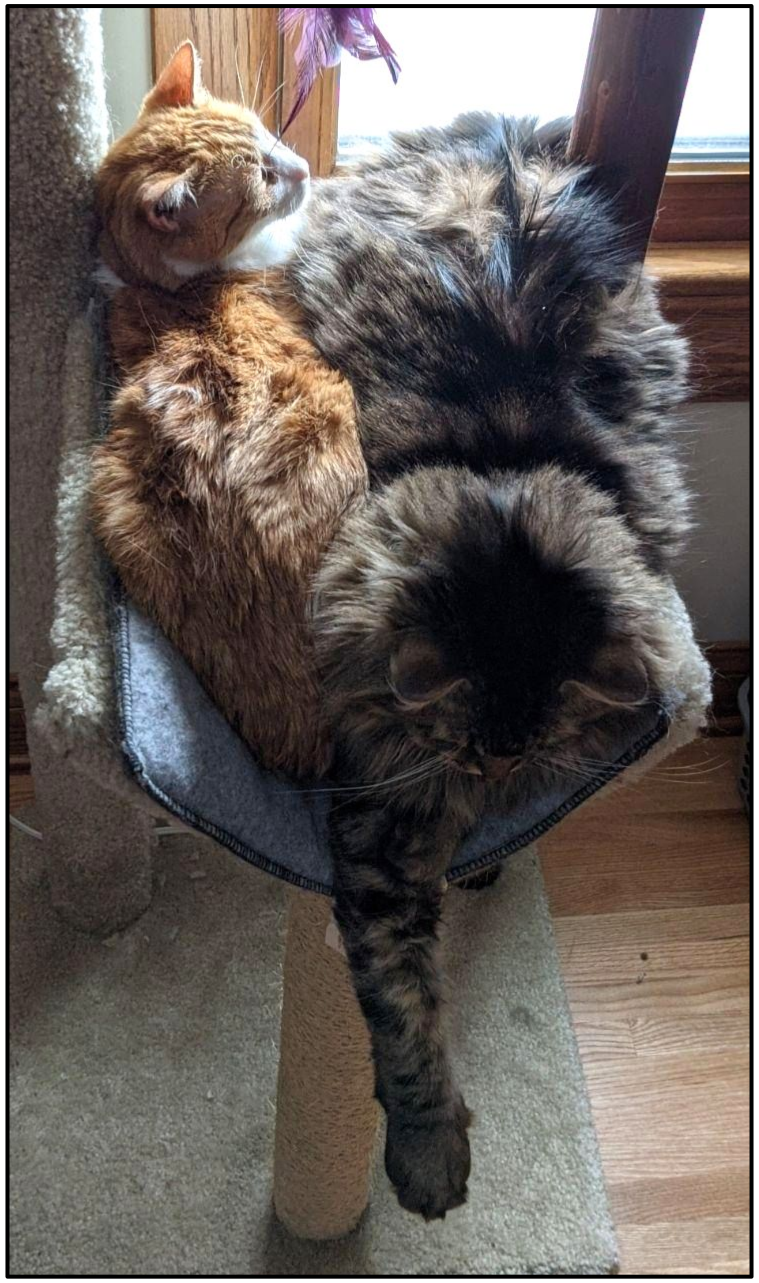
</C_Primer>

Next:

- Tiny quiz!
- The process and process management
- ECF (exceptional control flow)
- Re-posting Lec 03

Your tasks:

- **Read. every. word. of CH 8 in CS:APP**
- GitHub repo invite for Assignment 1 (discuss next wednesday)

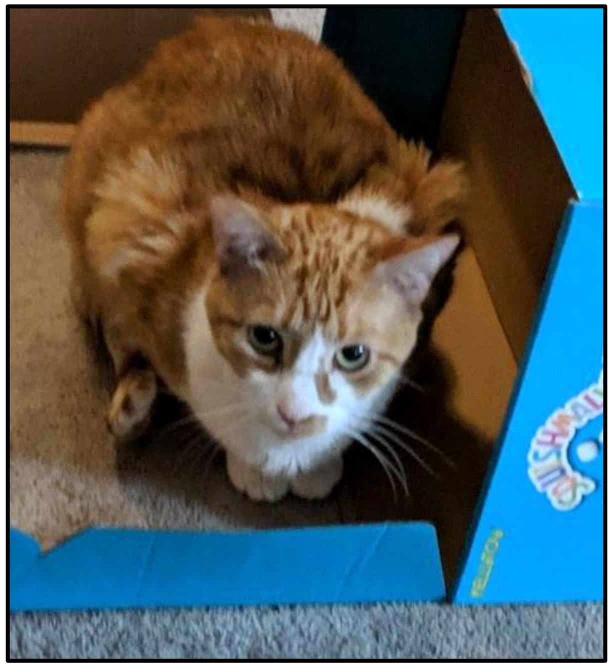


C Quiz: Kitties!

Because why not?

This is participation only
(effort = full credit)

I’m doing this to gauge your collective C understanding, so ***please work independently!***



Instructions:

1. Open BB -> CS351 -> Assessments
2. Start “C Quiz - Lec 04”