HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 9: Data Representation and Memory Management

Stephen Chong (Today: Ming Kawaguchi)

https://www.seas.harvard.edu/courses/cs153

Announcements

- project 1 feedback out
- Project 2 out
  - Due Thursday Oct 4 (2 days)
- Project 3 out
  - Due Tuesday Oct 9 (7 days)
- Project 4 out tonight!
  - Due Thursday Oct 25 (23 days)

Today

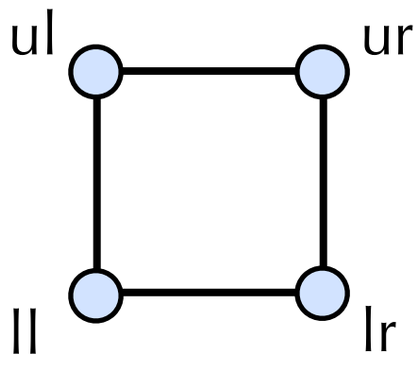
- Structs and memory
- Copy-in/Copy-out vs Call-by-reference
- Arrays and strings
- Allocation on stack vs heap
- Malloc/free

Structs (~Records) in C

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

struct Rect { struct Point ll,lr,ul,ur; };
```

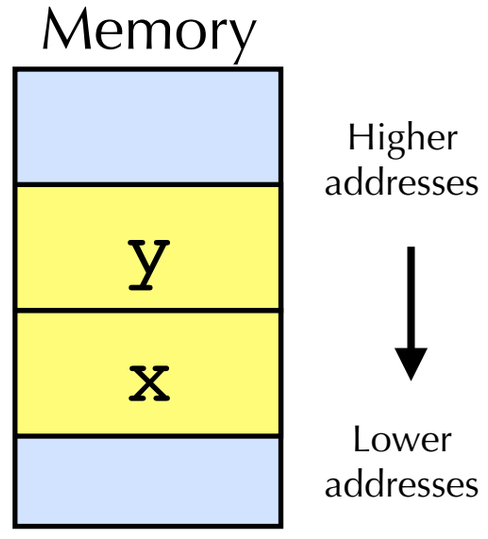
```
struct Rect mkSquare(struct Point ll, int len) {
  struct Rect res;
  res.lr = res.ul = res.ur = res.ll = ll;
  res.lr.x += len;
  res.ur.x += len;
  res.ur.y += len;
  res.ul.y += len;
}
```



Representation of Structs

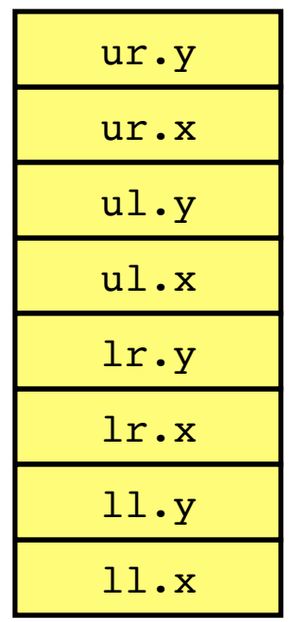
```
struct Point { int x; int y; };
```

- Two contiguous words
- Alternatively, use two registers?




```
struct Rect { struct Point ll,lr,ul,ur; };
```

- Eight contiguous words

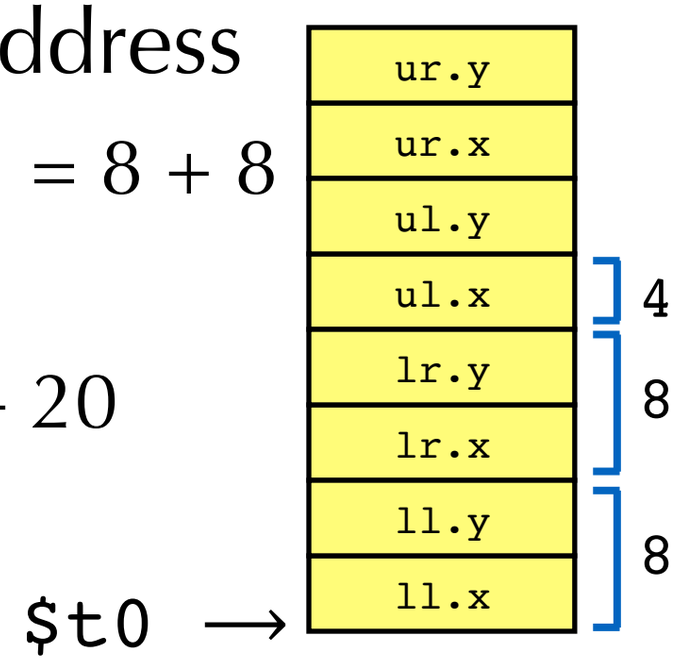


Accessing Struct Members

```
struct Point { int x; int y; };
struct Rect { struct Point ll,lr,ul,ur; };
struct Rect r = ...;
int i = r.ul.y;
```

- How do we access a member of a record? E.g., `r.ul.y`
- Assume that `$t0` holds the base address of `r`
- Calculate offsets of path relative to base address
  - `.ul` = `sizeof(Point)` + `sizeof(Point)` = 8 + 8
  - `.y` = `sizeof(int)` = 4
  - So `r.ul.y` can be accessed at address `$t0` + 20

```
lw $t2, 20($t0)
```



Copy-In/Copy-Out

- How do we handle assignment of records?

```
struct Rect mkSquare(struct Point ll, int len) {
  struct Rect res;
  res.lr = ll;
  ...
}
```

- Copy all elements out of source, and into target
  - Equivalent to doing word-level operations:

```
res.lr.x = ll.x;
res.lr.y = ll.y;
```

  - For large structs, could use something like memcpy to copy contiguous memory

Procedure Calls

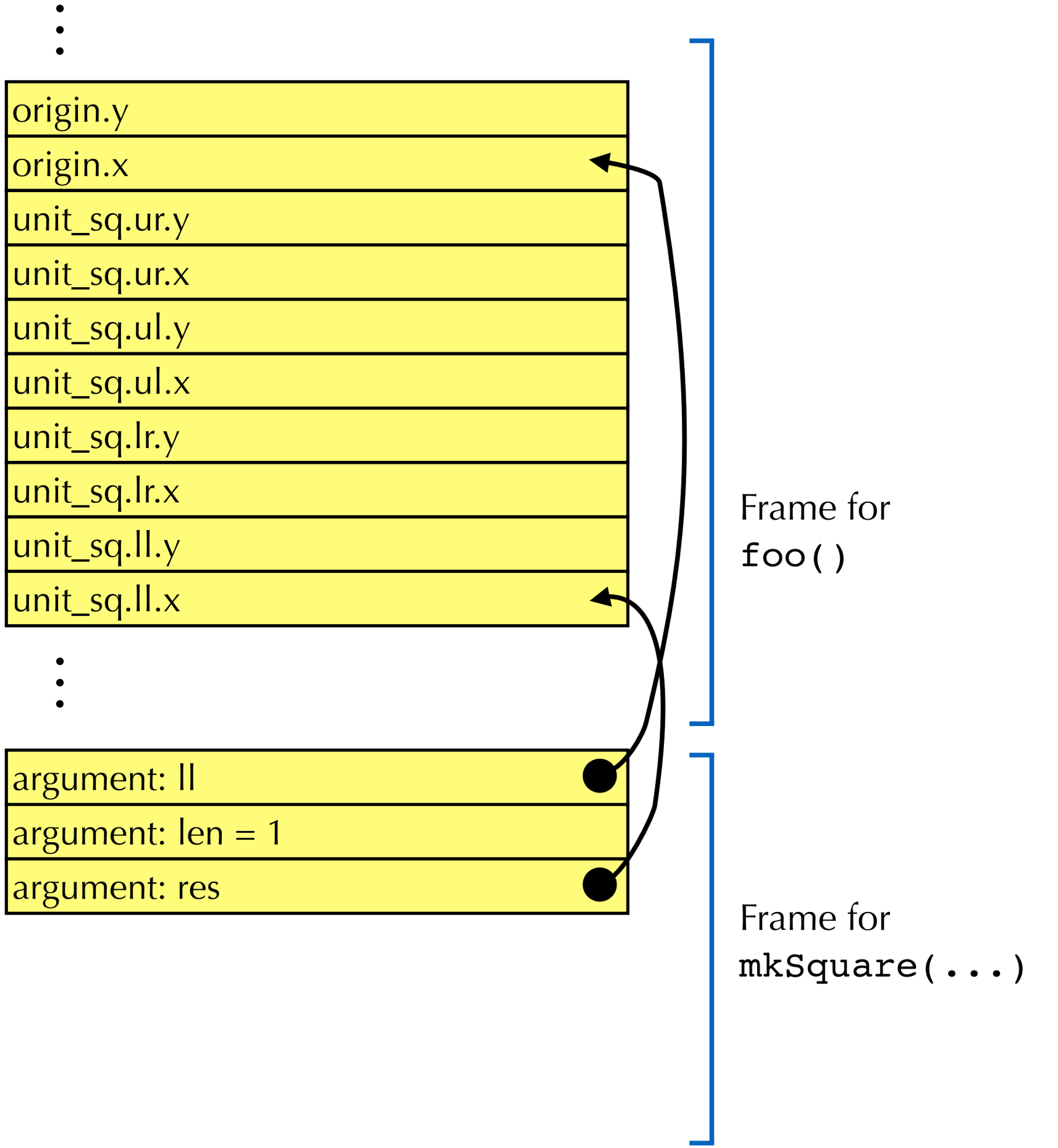
- Similarly, when we call a procedure we copy arguments in and result out
  - Caller sets aside additional space in frame for results that are bigger than 2 words
- Sometimes this is called “call-by-value” or “pass-by-value”
  - Bad terminology
  - Copy-in/copy-out is better
- Problem: expensive for large records

Pass-by-Reference

- Instead of copy-in/copy-out, caller can pass address of struct
- Called **pass-by-reference**

```
void mkSquare(struct Point *ll, int len,
              struct Rect *res) {
  res->lr = res->ul = res->ur = res->ll = *ll;
  res->lr.x += len;
  res->ur.x += len;
  res->ur.y += len;
  res->ul.y += len;
}

void foo() {
  struct Point origin = {0,0};
  struct Rect unit_sq;
  mkSquare(&origin, 1, &unit_sq);
}
```


origin.y
origin.x
unit_sq.ur.y
unit_sq.ur.x
unit_sq.ul.y
unit_sq.ul.x
unit_sq.lr.y
unit_sq.lr.x
unit_sq.ll.y
unit_sq.ll.x
argument: ll
argument: len = 1
argument: res
Frame for
foo()
Frame for
mkSquare(...)

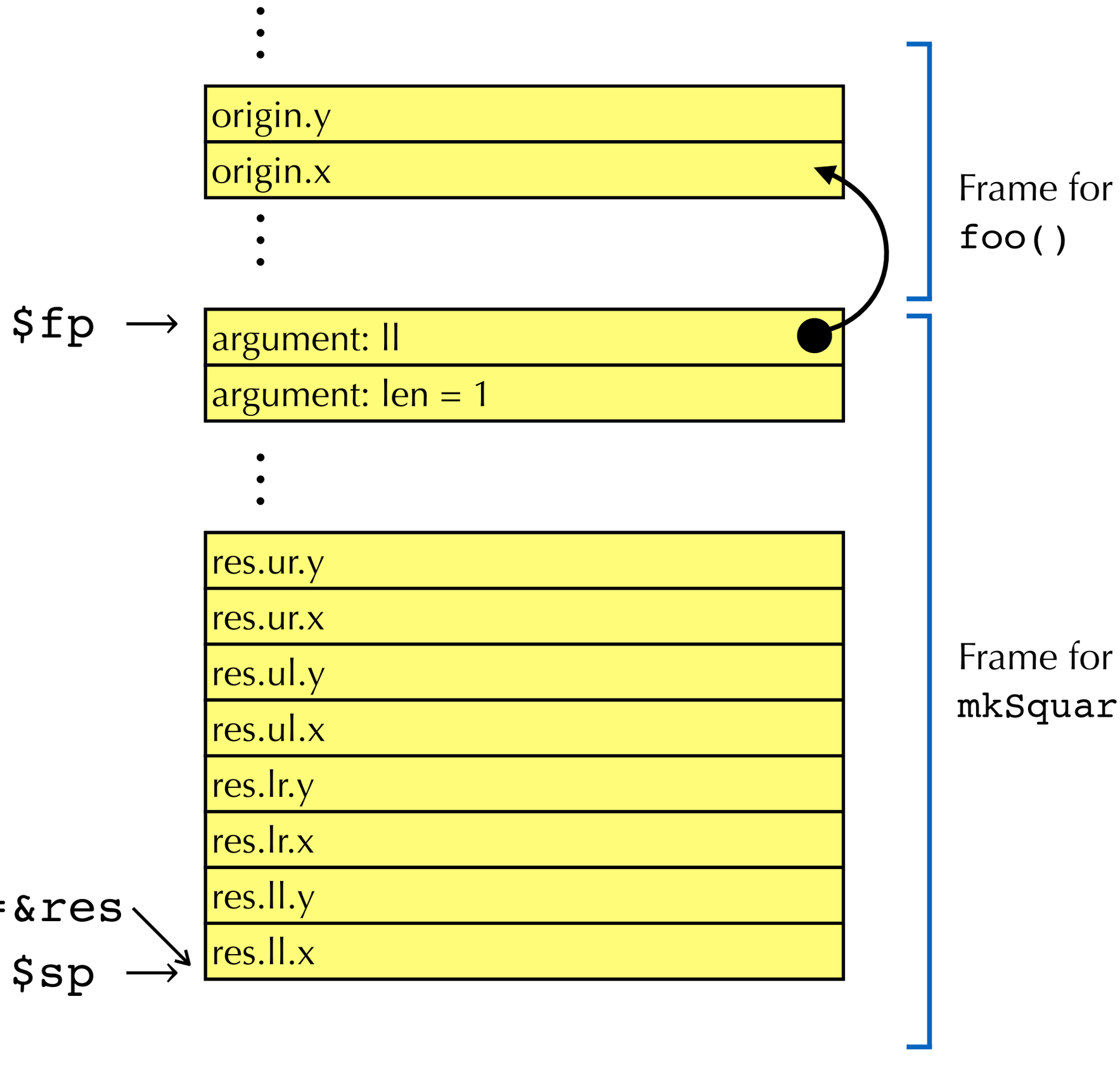
Pass-by-Reference

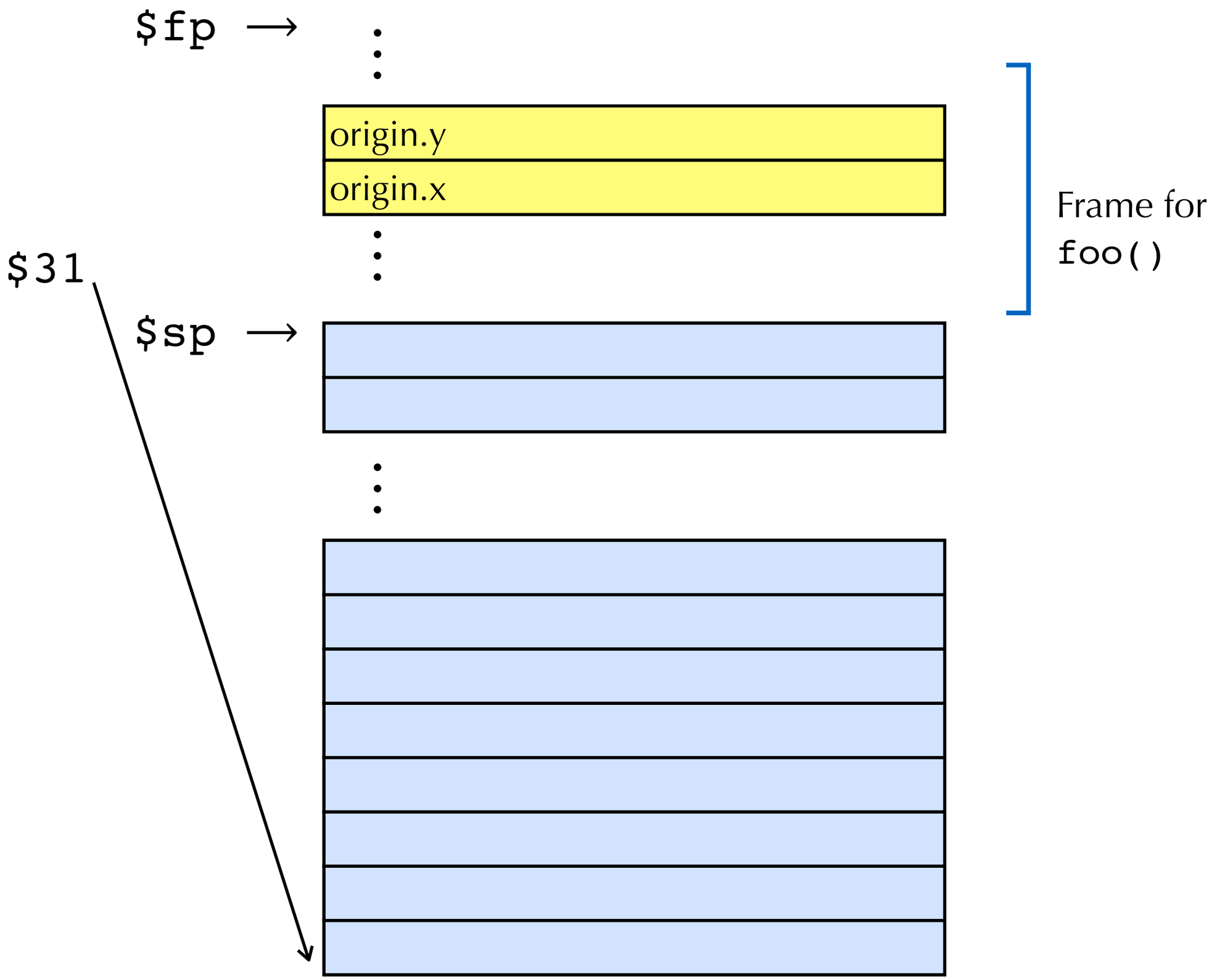
- In some languages, up to programmer
  - E.g., C, C++ argument type indicates if passing value, pointer, reference
- In other languages, it is a language design decision
  - E.g., Java is call-by-reference only

Puzzle: what's wrong with this?

```
struct Rect * mkSquare(struct Point *ll, int len) {
  struct Rect res;
  res.lr = res.ul = res.ur = res.ll = *ll;
  res.lr.x += len;
  res.ur.x += len;
  res.ur.y += len;
  res.ul.y += len;

  return &res;
}
```


origin.y
origin.x
$fp
argument: ll
argument: len = 1
Frame for
foo()
res.ur.y
res.ur.x
res.ul.y
res.ul.x
res.lr.y
res.lr.x
res.ll.y
res.ll.x
$31 = &res
$sp
Frame for
mkSquare(...)


$fp →
origin.y
origin.x
Frame for
foo()
$31
$sp →

Stack vs Heap Allocation

- When should we allocate data on stack?
- Only when we know the size of the data
- Only when the data is **not used** after the procedure returns!
  - Previous example: pointer to `res` returned by `mkSquare`, but `res` was in stack frame for `mkSquare`
  - Failure to do this leads to bugs and security vulnerabilities...
- Other data must be allocated in heap
  - i.e., use `malloc()`

Managing the Heap

- Some languages (C, C++, ...) have **manual memory management**
  - Programmers explicitly call `malloc()` to allocate memory in heap
  - Must remember to call `free()` to release the memory
    - Forgetting to call `free` results in **memory leaks**
    - Calling `free` more than once on same pointer (aka **double free**) results in bugs and vulnerabilities
- Other languages (Java, OCaml, Python, JavaScript, ...) have **managed memory** garbage collection
  - Language runtime looks after allocation and **garbage collection**
  - More on this next lecture...

Recall Puzzle...

```
struct Rect * mkSquare(struct Point *ll, int len) {
  struct Rect res;
  res.lr = res.ul = res.ur = res.ll = *ll;
  res.lr.x += len;
  res.ur.x += len;
  res.ur.y += len;
  res.ul.y += len;

  return &res;
}
```

Solution

```
struct Rect *mkSquare(struct Point *ll, int len) {
  struct Rect *res = malloc(sizeof(struct Rect));
  res->lr = res->ul = res->ur = res->ll = *ll;
  (*res).lr.x += len;
  res->ur.x += len;
  res->ur.y += len;
  (*res).ul.y += len;
  return res;
}
```

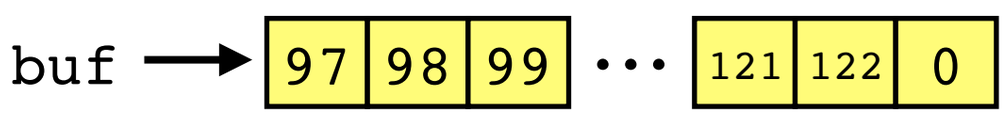
Representation of Arrays and Strings

```
void foo() {
  char buf[27];

  buf[0] = 'a';
  buf[1] = 'b';
  ...
  buf[25] = 'z';
  buf[26] = 0;
}
```

```
void foo() {
  char buf[27];

  *(buf) = 'a';
  *(buf+1) = 'b';
  ...
  *(buf+25) = 'z';
  *(buf+26) = 0;
}
```



- An array in C is a contiguous region of memory
  - A string is just an zero-terminated array of `char`
- Accessing element: `buf[`*`i`*`]` is (base of array `buf`) + *`i`* * `size_of(`*`element`*`)`
- Same issues of allocating on stack or heap...

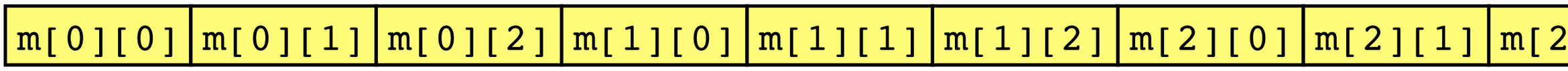
Multi-dimensional Arrays

- In C `int m[4][3]` yields an array with 4 rows and 3 columns.
  - Laid out in row-major order:
  - `m[0][0]`, `m[0][1]`, `m[0][2]`, `m[1][0]`, `m[1][1]`, ...

m[0][0]	m[0][1]	m[0][2]
m[1][0]	m[1][1]	m[1][2]
m[2][0]	m[2][1]	m[2][2]
m[3][0]	m[3][1]	m[3][2]

Multi-dimensional Arrays

- In C `int m[4][3]` yields an array with 4 rows and 3 columns.
  - Laid out in row-major order:
  - `m[0][0]`, `m[0][1]`, `m[0][2]`, `m[1][0]`, `m[1][1]`, ...



- So `m[i][j]` is located where?
  - (base address of `m`) + (`i` * 3 * `sizeof(int)`) + `j` * `sizeof(int)`

Multi-dimensional Arrays

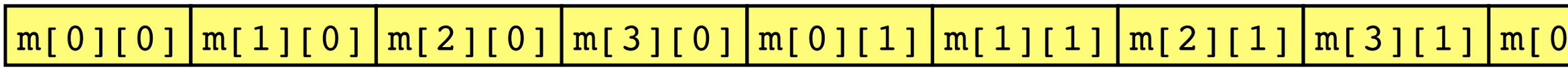
- In Fortran, arrays are laid out in column major order

m[0][0]	m[0][1]	m[0][2]
m[1][0]	m[1][1]	m[1][2]
m[2][0]	m[2][1]	m[2][2]
m[3][0]	m[3][1]	m[3][2]

- In ML, there are no multi-dimensional arrays int array) array.
  - Why is knowing this important?

Multi-dimensional Arrays

- In Fortran, arrays are laid out in column major order



- In ML, there are no multi-dimensional arrays int array) array.
  - Why is knowing this important?

Constant Strings

- A string constant `"foo"` is represented as global data:

```
_string42: 102 111 111 0
```

- It's usually placed in the text segment so it's read only.
  - allows all copies of the same string to be shared.
- Typical mistake:

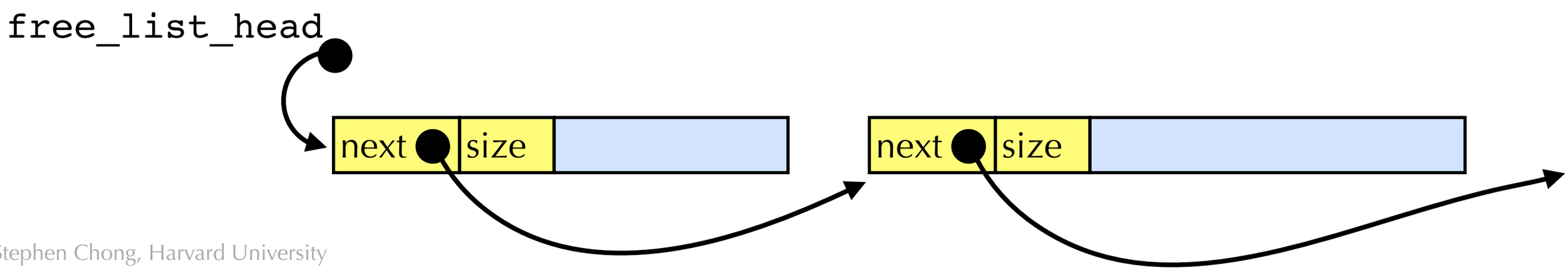
```
char *p = "foo";
p[0] = ‘b';
```

How do malloc and free work?

- (High-level view only; for more info, see CS61!)
- Upon `malloc(`*`n`*`)`:
  - Find an unused space on heap of at least size *n*
  - (Need to mark space as in use)
  - Return address of that space
- Upon `free(`*`p`*`)`:
  - Mark space pointed to by *p* as free
  - (Need to keep track of how big object is to know how much space to free)

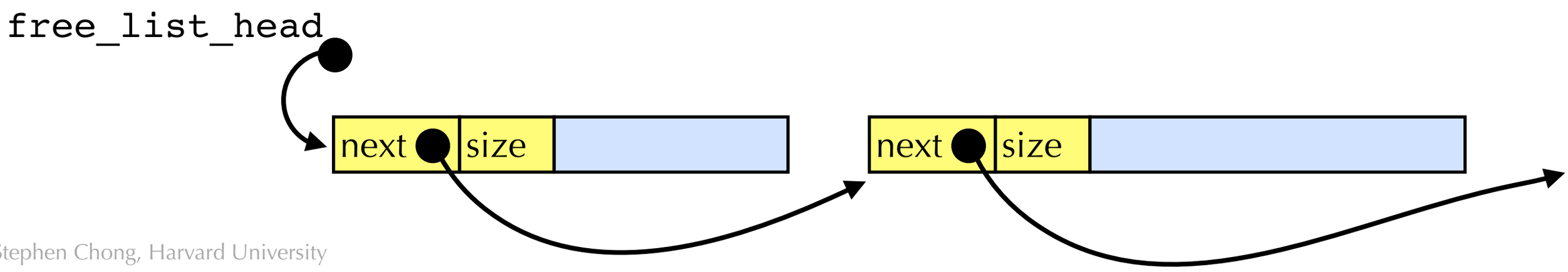
One Option: Free List

- Keep a linked list of contiguous chunks of free memory
  - Each component of list has two words of meta-data
    - 1 word points to the next element in the free list
    - The other word says how big the object is



One Option: Free List

- To `malloc(`*n*`)`:
  - Run down list until find a block that is big enough (size ≥ ***n***)
  - Divide block, put left overs back on free list
  - First-fit vs best-fit?
- To `free(`*p*`)`:
  - Put object back in free list
  - Metadata lets us know how size of object
  - Keep list sorted to allow coalescing of adjacent free blocks



Multiple free lists!

- Keep an array of free lists!
  - Each list has chunks the same size
  - `free_list[i]` holds chunks of size 2^i
  - Round requests up to the next power of 2
  - When `free_list[i]` is empty, take a block from `free_list[i+1]` and divide it in half, putting both chunks in `free_list[i]`
  - Alternatively, run through `free_list[i-1]` and merge contiguous blocks

Modern Languages

- Represent all records (tuples, objects, etc.) using pointers.
  - Makes it possible to support polymorphism.
    - e.g., ML doesn't care whether we pass an integer, two-tuple, or record to the identity function: all represented with 1 word
  - Price paid: lots of loads/stores…
- By default, allocate records on the heap.
  - Programmer doesn't have to worry about lifetimes.
  - Compiler may determine that it's safe to allocate a record on the stack instead.
  - Uses a garbage collector to safely reclaim data.