Data & Memory Management

CS4410: Spring 2013

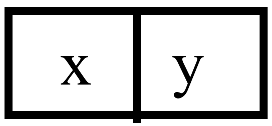
Records in C:

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

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

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

Representation:

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

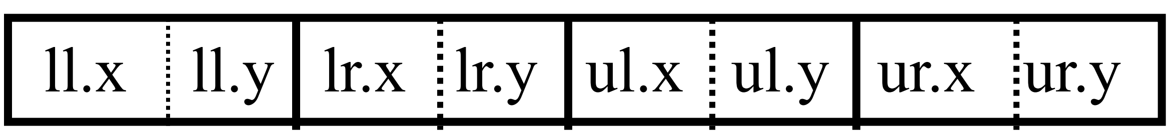
- Two contiguous words. Use base address.



- Alternatively, dedicate two registers?

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

- 8 contiguous words.



Member Access

i = rect.ul.y

- Assuming $t holds address of p:
- Calculate offsets of path relative to base:
  - .ul = sizeof(struct Point)+sizeof(struct Point), .y = sizeof(int)
  - So lw $t2, 36($t)

Copy-in/Copy-out

When we do an assignment as in:

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

then we copy all of the elements out of the source and put them in the target. Same as doing word-level opn's:

```
struct Rect mkSquare(struct Point ll, int elen) {
  struct Rect res;
  res.lr.x = ll.x;
  res.lr.y = ll.x;
  ...
```

For really large copies, we use something like memcpy.

Procedure Calls:

- Similarly, when we call a procedure, we copy arguments in, and copy results out.
  - Caller sets aside extra space in its frame to store results that are bigger than 2-words.
  - We do the same with scalar values such as integers or doubles.
- Sometimes, this is termed "call-by-value".
  - This is bad terminology.
  - Copy-in/copy-out is more accurate.
- Problem: expensive for large records…

Arrays

```
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;
   }
```

Space is allocated on the stack for buf.

(note, without alloca, need to know size of buf at compile time…)

buf[i] is really just base of array + i * elt_size

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[i][j] compiles to?
- In Fortran, arrays are laid out in *column major order*.
- In ML, there are no multi-dimensional arrays -- (int array) array.

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.
- Rookie mistake:

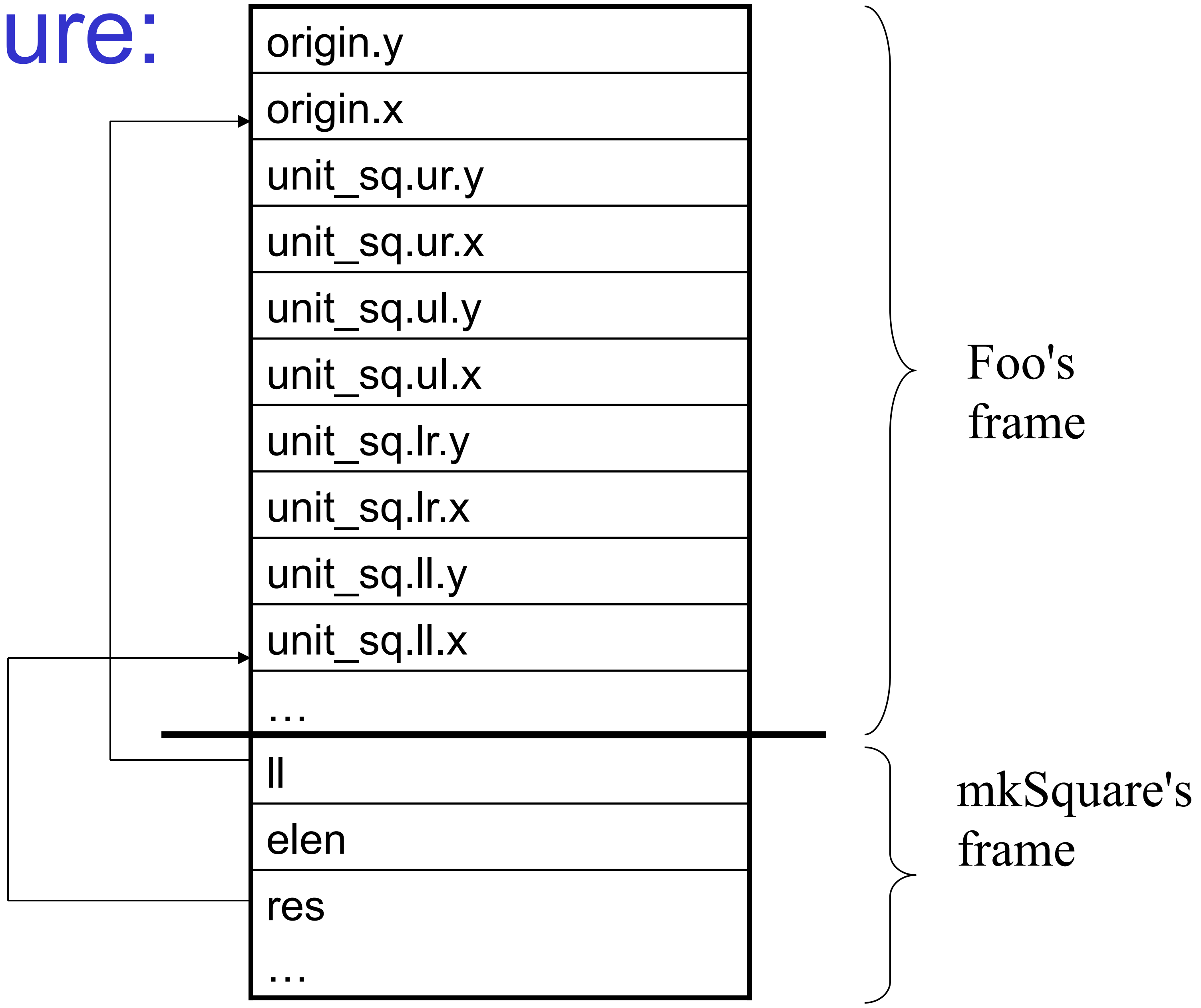
```
char *p = "foo";
p[0] = 'b';
```

Pass-by-Reference:

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

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

The caller passes in the address of the point and the address of the result (1 word each).

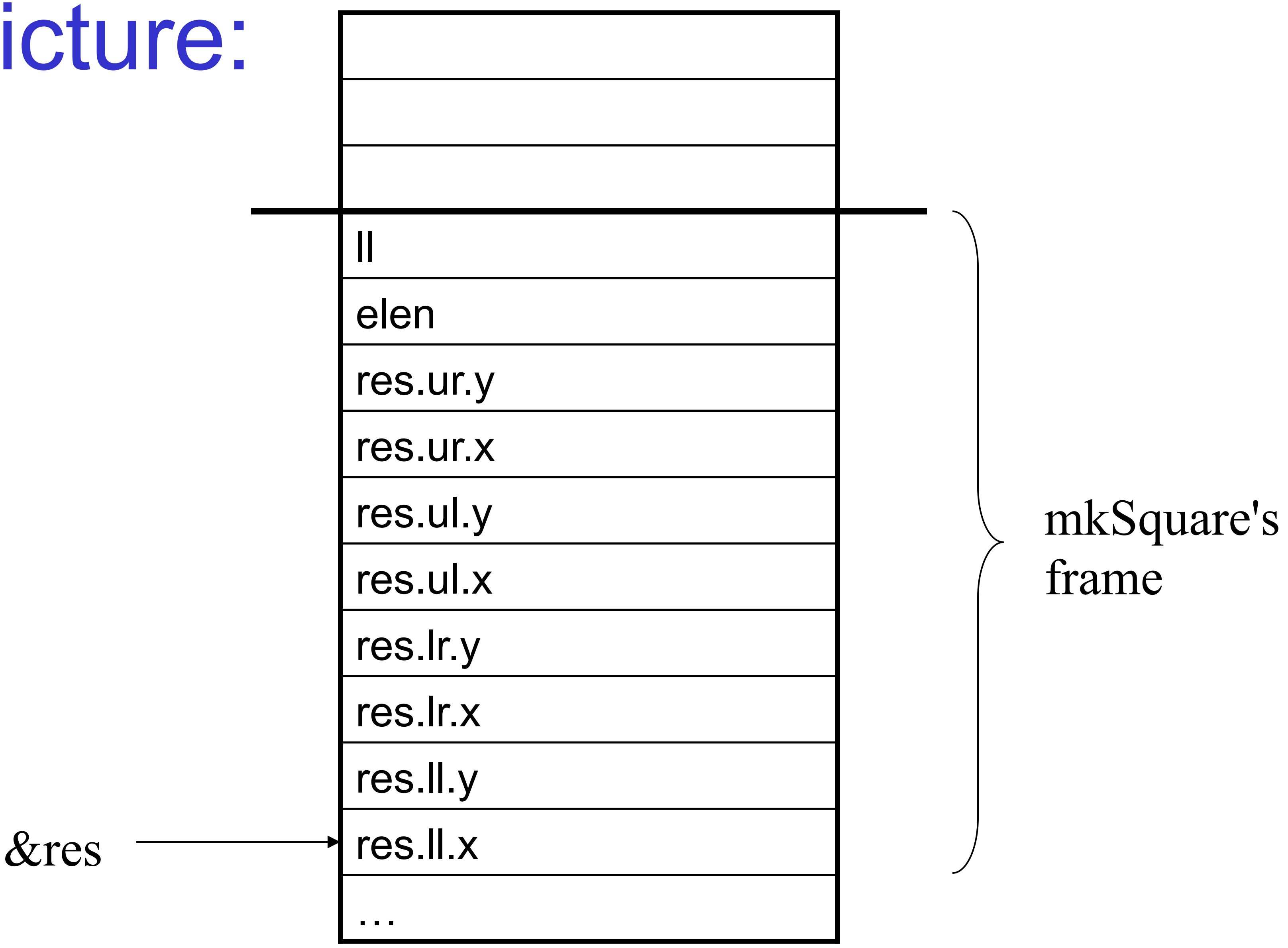
Picture:


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
…
ll
elen
res
…
Foo's frame
mkSquare's frame

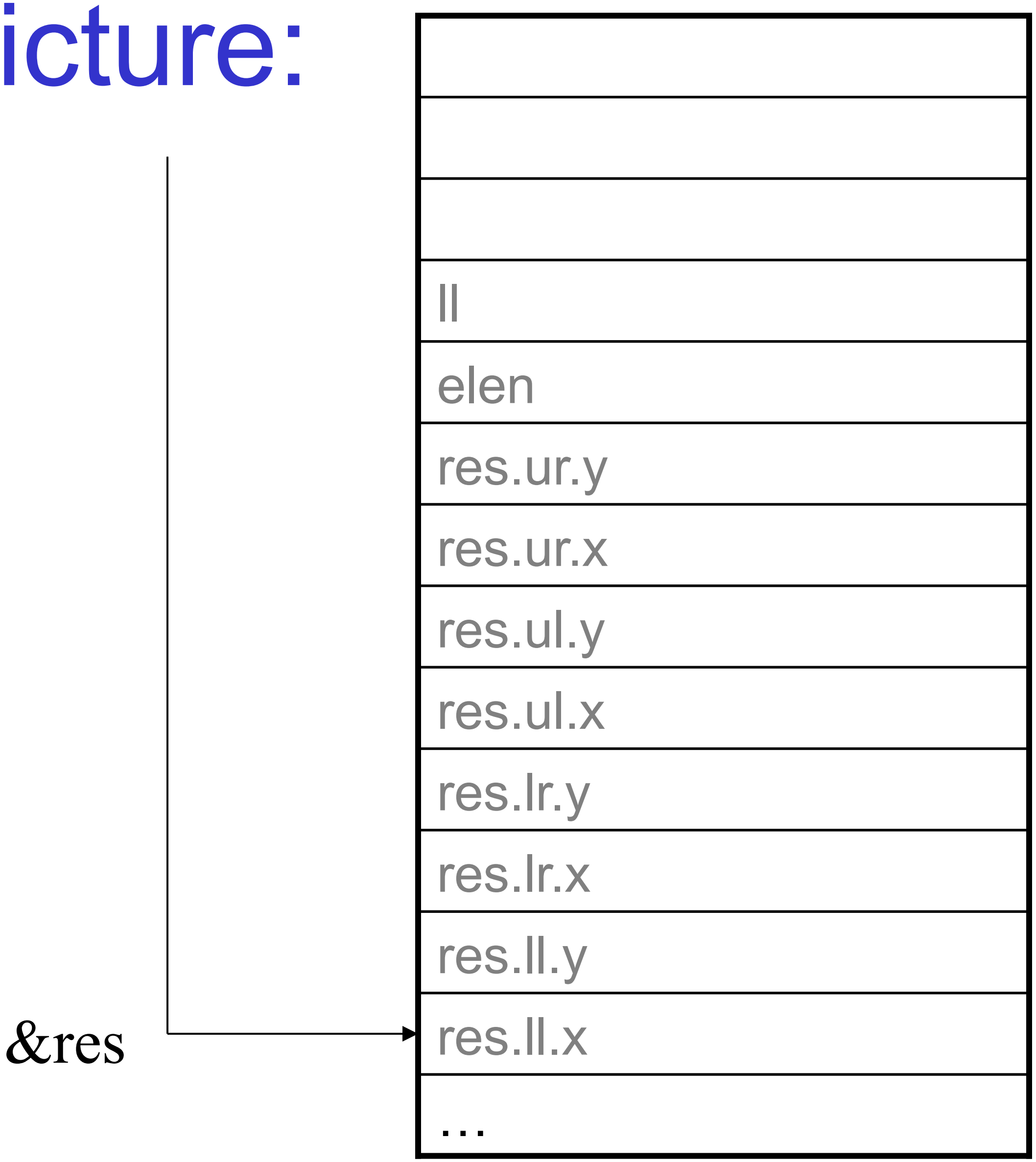
What's wrong with this?

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

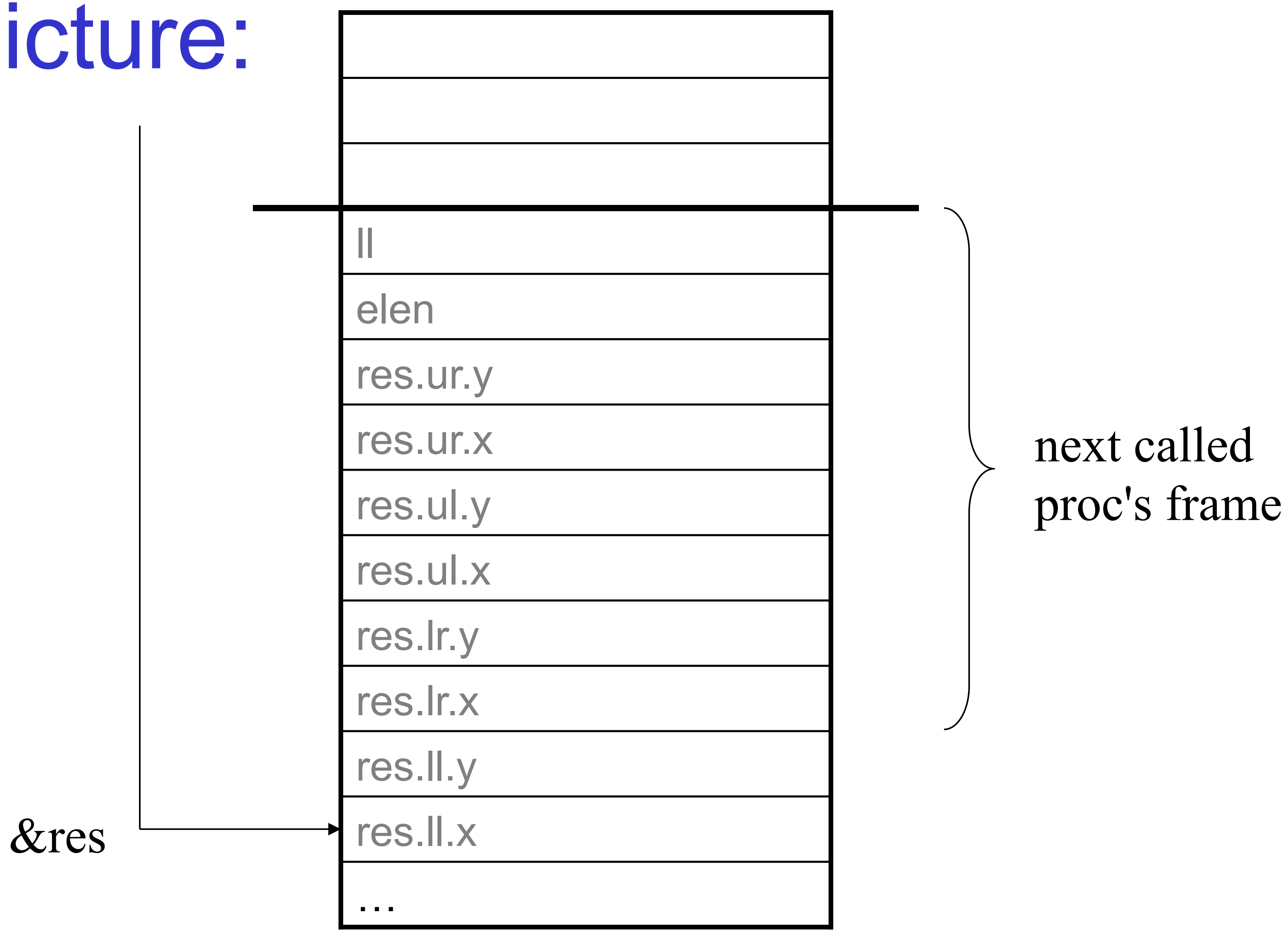
Picture:


ll
elen
res.ur.y
res.ur.x
res.ul.y
res.ul.x
res.lr.y
res.lr.x
res.ll.y
res.ll.x
…
&res
mkSquare's frame

Picture:


ll
elen
res.ur.y
res.ur.x
res.ul.y
res.ul.x
res.lr.y
res.lr.x
res.ll.y
res.ll.x
…
&res

Picture:


ll
elen
res.ur.y
res.ur.x
res.ul.y
res.ul.x
res.lr.y
res.lr.x
res.ll.y
res.ll.x
…
&res
next called
proc's frame

Stack vs. Heap Allocation

- We can only allocate an object on the stack when it is no longer used after the procedure returns.
  - NB: it's possible to exploit bugs like this in C code to hijack the return address. Then an attacker can gain control of the program…
- For other objects, we must use the heap (i.e., malloc).
  - And of course, we must remember to free the object when it is no longer used! Also a big source of bugs in C/C++ code.
  - Java, ML, C#, etc. use a garbage collector instead.

Program Fixed:

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

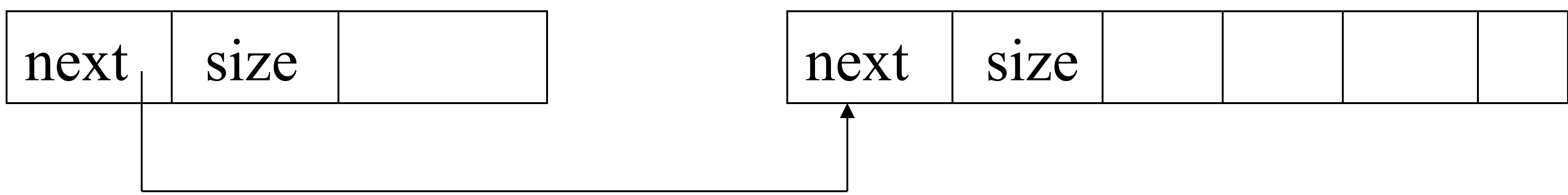
How do malloc/free work?

- Upon malloc(n):
  - Find an unused space 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.)

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.



Malloc and Free

- To malloc, run down the list until you find a spot that's big enough to satisfy the request.
  - Take left-overs and put them back in the free-list.
  - First-fit vs. Best-fit?
- To free, put the object back in the list.
  - Perhaps keep chunks sorted so that adjacent chunks can be coalesced.
- Pros and Cons?
- What happens if you free something twice or free the middle of an object?

Exponential Scaling:

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

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: they are 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.
  - Because pointers are *abstract*, has the freedom to re-arrange the data in the heap to support compaction.

Allocation in SML/NJ

- Reserve two registers:
  - allocation pointer (like stack pointer)
  - limit pointer
- To allocate a record of size n:
  - checks that limit-alloc > n. If not, invokes garbage collector.
  - Adds n+1 to the alloc pointer, returns old value of alloc pointer as result.
  - Extra word holds meta-data (e.g., size.)
  - Actually, amortizes the limit check across a bunch of allocations (just as we amortize stack pointer adjustment.)
  - Result: 3-5 instructions to allocate a record.