CSCI 2330


Does your dog bite?
No, but it can hurt you in other ways.
Array indexing starts at 1

TODO

- Assignment 3 assigned due October 28

```
j.knockel@krebs:~/assignments/lab3$ cat README
This is bombseed116081297159636014777127300996828590641394618974O.

It belongs to j.knockel (j.knockel@bowdoin.edu)

See https://tildesites.bowdoin.edu/~j.knockel/csci2330f26/assignments/?id=lab3-
xxxxxxxxxxxxxxx for more assignment instructions.
```

Overview

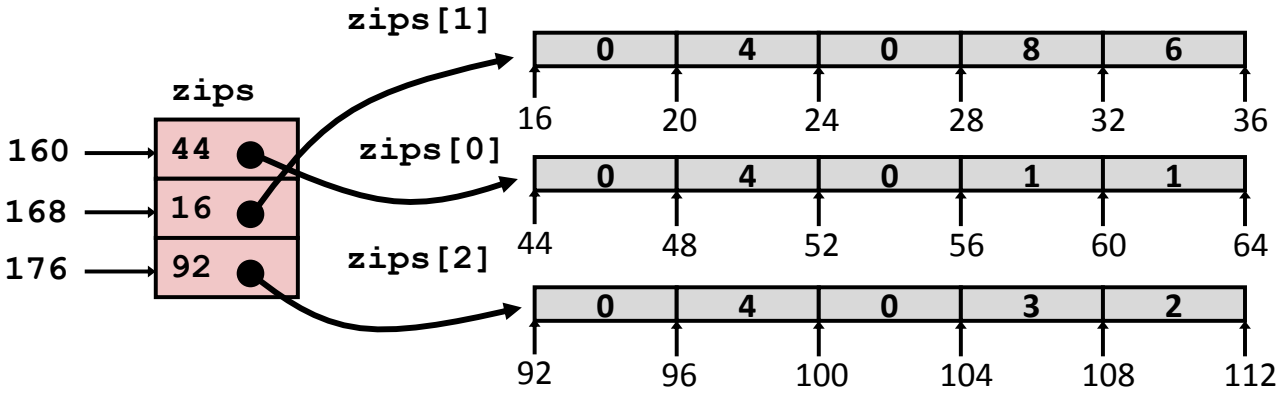
- Finish multidimensional arrays
- Structs

Multidimensional arrays

- Arrays that have two or more dimensions
- E.g., for three dimensions, we would index like `x[5][4][7]`

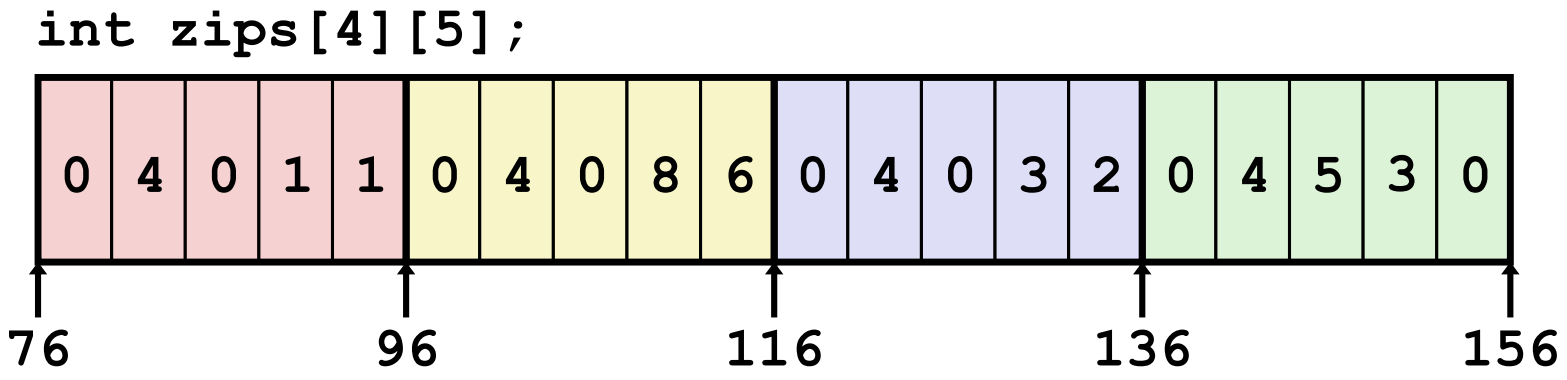
Multilevel arrays

```
int *zips[3];  // stored at address 160
zips[0] = (int *) malloc(sizeof(int) * 5);
zips[0][1] = 4;
...
```



Nested array on stack

```
int zips[4][5];  // stored at address 76
zips[0][1] = 4;
...
```



Nested array on heap

```
int (*zips)[4][5] = malloc(sizeof(int[4][5]));
(*zips)[0][1] = 4
...
```

Nested array on heap

```
int (*zips)[4][5] = malloc(sizeof(int[4][5]));
(*zips)[0][1] = 4
...
```

- Note:

  `(*zips)[x][y]`

- is equivalent to:

```
int *ptr = (int *) zips;
ptr[x * 5 + y] // column-major would be different
```

%rdx = %rsi[%rax][%rcx]

In C:

- `int (*%rsi)[4][5] = malloc(sizeof(int[4][5]));`
- `%rdx = %rsi[%rax][%rcx]`

The latter line in x86-64:

- `leaq (%rax,%rax,4), %rdx # %rdx = %rax * 5`
- `addq %rcx, %rdx # %rdx += %rcx`
- `movq (%rsi,%rdx,4), %rdx # %rdx = %rsi[%rdx]`

Multilevel vs. nested arrays

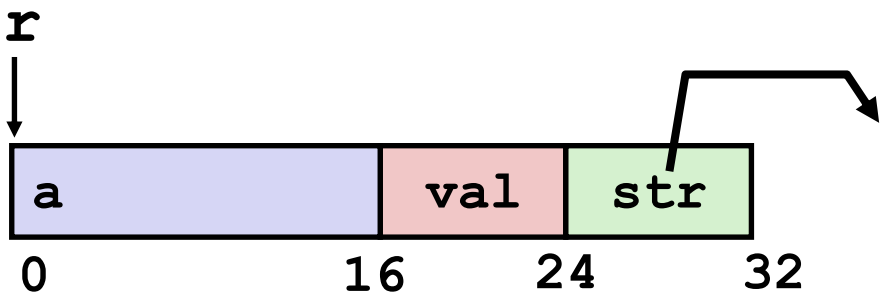
- Multilevel arrays can be jagged: different rows can have different lengths
- Nested arrays are faster (no pointer chasing)
- Nested arrays take less memory (no overhead pointer)

Structs

- “Structures”
- Closest thing to objects in C
- Collections of sequential *fields*
- Unlike arrays, where every item has the same type, struct fields can have different types
- A struct definition is itself a type…

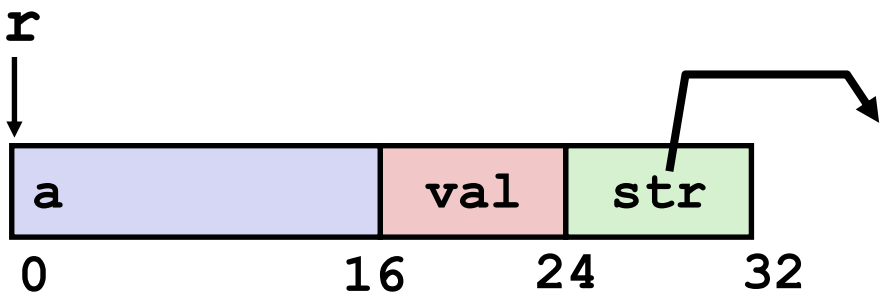
New 32-byte type!

```
struct thing {
    int a[4];
    long val;
    char *str;
};
```



New 32-byte type!

```
struct thing {
    int a[4];
    long val;
    char *str;
};
```



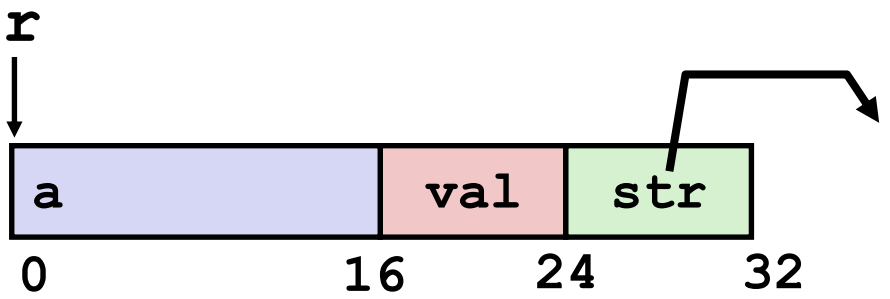

```
struct thing x; // 32 bytes
struct thing y; // 32 bytes

x.val = 5;
x.a[1] = 2;
x.str = “hello”;

y = x; // copy full struct
```

New 32-byte type!

```
struct thing {
    int a[4];
    long val;
    char *str;
};
```




```
struct thing x; // 32 bytes
struct thing y; // 32 bytes

x.val = 5;
x.a[1] = 2;
x.str = "hello";

y = x; // copy full struct
```

```
struct thing *p; // 8 bytes
p = malloc(sizeof(struct thing));

// form 1
(*p).val = 7; // NOT p.val = 7

// form 2 (preferred)
p->val = 7; // same as form 1

struct thing *p2;
p2 = p; // just a pointer copy
```

Comparison to Java

- `struct thing a; // this is not possible`
- `struct thing b = a; // nor is this`
- In Java, you can only have “references” (pointers) to objects
  - In C you can have pointers and structs in themselves
- In Java, you cannot allocate objects to the stack; you can only allocate via “new” (heap)
  - In C you can allocate structs to the stack, heap, or static memory.

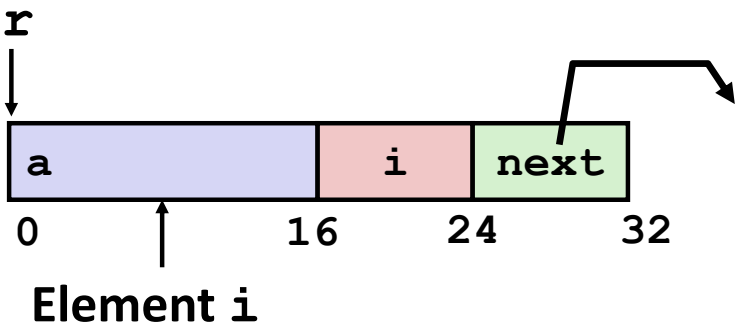
Comparison to Java

- `struct thing a;`
- `struct thing *p = ...;`
- `p->x // this is p.x in Java`
- `a.x // this has no equivalent in Java`
  `// because you cannot have a thing`
  `// in itself, only a reference to one`

The `.` operator in C

- Similar to `[]`, `.` accesses a memory offset
- With `[]`, the offset depends on the index
- With `.`, the offset depends on the field
- You can have an array of structs, in which case the final offset depends on both `[]` and `.`
- Let’s look at x86-64 generation…

```
struct list_el {
    int a[4];
    long i;
    struct list_el *next;
};
```




```
void set_val
    (struct list_el *n, int val) {
  do {
    long i = n->i;
    n->a[i] = val;
    n = n->next;
  } while (n);
}
```

Register	Value
%rdi	n
%rsi	val

```
.L1:                                    # loop:
  movq    16(%rdi), %rax                #    i = M[n+16]
  movl    %esi, (%rdi,%rax,4)           #    M[n+4*i] = val
  movq    24(%rdi), %rdi                #    n = M[n+24]
  testq   %rdi, %rdi                    #    Test n
  jne     .L1                           #    if !=0 goto loop
```

Initialize structs

- `struct point { long x; long y; };`
- `struct point a;`
- `a.x = 1;`
- `a.y = 2;`
- `struct point b;`
- `b.x = 3;`
- `b.y = 4;`

Initialize structs

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `a.x` → ?

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `a.x` → 1

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `a = b;`
- `a.x` → ?

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `a = b;`
- `a.x` → 3

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `struct point *p = &a;`
- `struct point *q = &b;`
- `p = q;`
- `a.x` → ?

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `struct point *p = &a;`
- `struct point *q = &b;`
- `p = q;`
- `a.x` → 1

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `struct point *p = &a;`
- `struct point *q = &b;`
- `*p = *q;`
- `a.x` → ?

Quiz

- `struct point { long x; long y; };`
- `struct point a = {1, 2}; // sets a.x = 1, a.y = 2`
- `struct point b = {3, 4}; // sets b.x = 3, b.y = 4`
- `struct point *p = &a;`
- `struct point *q = &b;`
- `*p = *q;`
- `a.x` → 3

Problem

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

struct point add_points(struct point a, struct point b) {
    struct point sum = struct point a;
    sum.x += b.x;
    sum.y += b.y;
    return sum;
}
```

Problem

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

struct point add_points(struct point a, struct point b) {
    struct point sum = struct point a;
    sum.x += b.x;
    sum.y += b.y;
    return sum;
}
```

Passing structs to/from functions

- There is a calling convention for this too!
- But it is beyond the scope of this class :)

Passing structs to/from functions

- Copying large structs can be expensive
- C programmers usually pass/return structs by pointer (e.g., `struct point *`), not by value (e.g., `struct point`)

typedefs

- Create type aliases.
- Commonly used for structs to create more succinct types:
- `struct point { int x; int y; };`
- `typedef struct point point_t;`
- `point_t a; // can now say point_t`
- `point_t b; // instead of struct point`

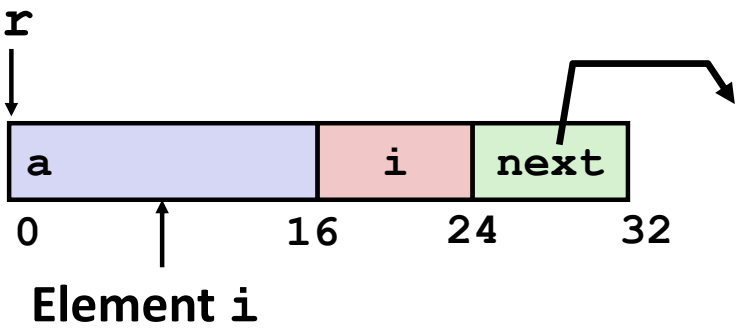
typedefs

- Other uses:
- …
- `typedef short s16;`
- `typedef unsigned short u16;`
- `typedef int s32;`
- `typedef unsigned int u32;`
- …

Structs in this class

- Looking for their use in Lab 3
- Using them for programming in Labs 5 and 6
- What should we look for in Lab 3?

```
struct list_el {
    int a[4];
    long i;
    struct list_el *next;
};
```




```
void set_val
    (struct list_el *n, int val) {
  do {
    long i = n->i;
    n->a[i] = val;
    n = n->next;
  } while (n);
}
```

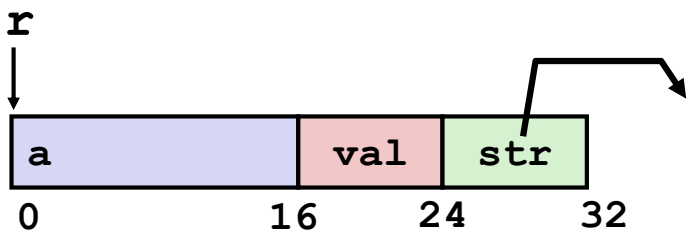
Register	Value
%rdi	n
%rsi	val

```
.L1:                                    # loop:
  movq    16(%rdi), %rax                #    i = M[n+16]
  movl    %esi, (%rdi,%rax,4)           #    M[n+4*i] = val
  movq    24(%rdi), %rdi                #    n = M[n+24]
  testq   %rdi, %rdi                    #    Test n
  jne     .L1                           #    if !=0 goto loop
```

Structs exercises

New 32-byte type!

```
struct thing {
    int a[4];
    long val;
    char *str;
};
```




```
struct thing x; // 32 bytes
struct thing y; // 32 bytes

x.val = 5;
x.a[1] = 2;
x.str = "hello";

y = x; // copy full struct
```

```
struct thing *p; // 8 bytes
p = malloc(sizeof(struct thing));

// form 1
(*p).val = 7; // NOT p.val = 7

// form 2 (preferred)
p->val = 7; // same as form 1

struct thing *p2;
p2 = p; // just a pointer copy
```

Remainder of today

- Work on Lab 3!

Next week

- We will learn how we can use our knowledge of machine code to exploit bugs and take control of machines

TODO

- Assignment 3 assigned due October 28