



Representing Data Structures

Multidimensional arrays
C structs

Big picture: how does larger data get stored in x86?

<https://cs.wellesley.edu/~cs240/>

1

Outline

Goal: understand how we represented structured data in C and x86

- Arrays in x86
- Array indexing
- Arrays of pointers to arrays
- 2-dimensional arrays (defer details to video)
- C structs (simpler version of objects)
- Overview and accessing fields
- Alignment
- LinkedList example

2

C: Array layout and indexing

ex

```
int val[5];
```



Recall:

- Array layout will be contiguous block of memory
- The base address will be aligned based on the element type: here, a multiple of 4

Write x86 code to load **val[i]** into **%eax**.

1. Assume:

- Base address of val is in %rdi
- i is in %rsi

2. Assume:

- Base address of val is 28 (%rsp)
- i is in %rcx

For: `T a[N]`
Address of **a[i]** is:
`a + i * sizeof(T)`

3

Which expression correctly loads val[i] into %rax? Assume val is in %rdi and i is in %rsi.

`movq (%rsi,%rdi,4), %rax`

`movq (%rdi,%rsi,4), %rax`

`movq (%rdi,%rsi,8), %rax`

`movq (%rsi,%rdi,8), %rax`

None of the above

```
long val[4];
```

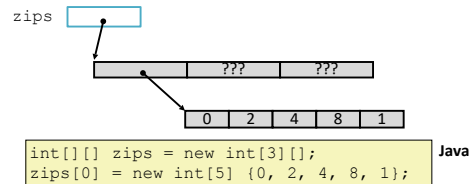


Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

C: Arrays of pointers to arrays of ...

reminder

```
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int)*5);
...
int* zip0 = zips[0];
zip0[0] = 0;
zip0[1] = 2;
zip0[2] = 4;
zip0[3] = 8;
zip0[4] = 1;
```



Java

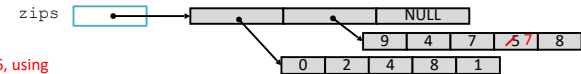
5

C: Arrays of pointers to arrays in x86

ex

```
void copyfromleft(int** zips, long i, long j) {
    zipCodes[i][j] = zipCodes[i][j - 1];
}
```

copyleft(zips, 1, 3)

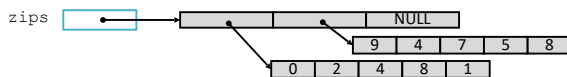


Goal: translate to x86, using
two scratch registers
%rax, %ecx (why 32 bits?)

1. Put zips[i] in a reg
2. Access element [j-1]
3. Set element [j]
4. Return (nothing)

6

C: Arrays of pointers to arrays: Pros/Cons



Pros:

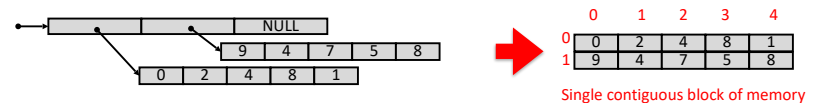
- Flexible array lengths
- Different elements can be different lengths
- Lengths can change as the program runs
- Representation of empty elements saves space

Cons:

- Accessing a nested element requires multiple memory operations

7

Alternative: row-major nested arrays



Pros:

- Accessing nested elements now a single memory operation!
- Calculations can be done ahead of time, via arithmetic

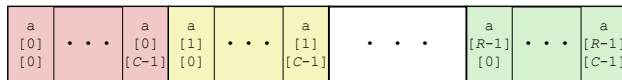
Cons:

- Less space efficient depending on the shape of the data
- Need to be careful with our order of indexing!

8

C: Row-major nested arrays

```
int a[R][C];
```



Suppose a's base address is A.

$\&a[i][j]$ is $\frac{A + C \times \text{sizeof}(\text{int}) \times i + \text{sizeof}(\text{int}) \times j}{\text{(regular unscaled arithmetic)}}$

```
int* b = (int*)a; // Treat as larger 1D array
```

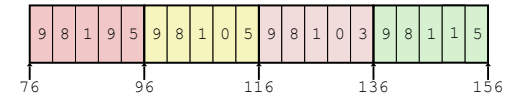
```
&a[i][j] == &b[ C*i + j ]
```

9

C: Strange array indexing examples

ex

```
int sea[4][5];
```



Reference Address Value

```
sea[3][3]
sea[2][5]
sea[2][-1]
sea[4][-1]
sea[0][19]
sea[0][-1]
```

C does not do any bounds checking.
Row-major array layout is guaranteed.

10

C structs

Like Java class/object, without methods.

Models structured, but not necessarily list-like, data.

Combines other, simpler types.

```
struct point {
    int xcoordinate;
    int ycoordinate;
};
```

```
struct student {
    int classyear;
    int id;
    char* name;
};
```

11

C structs

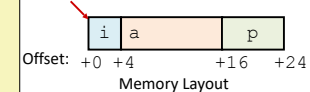
Like Java class/object without methods.

Compiler determines:

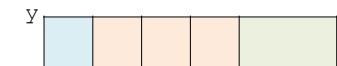
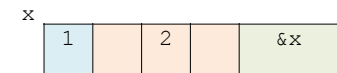
- Total size
- Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```

Base address



```
struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);
```



12

C structs

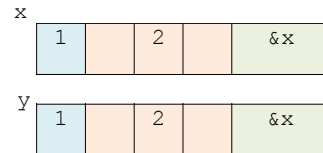
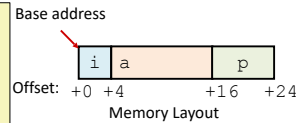
Like Java class/object without methods.

- Compiler determines:
- Total size
 - Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};

struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);

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



13

C structs

Like Java class/object without methods.

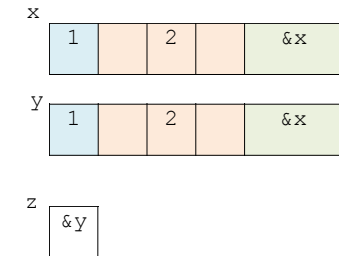
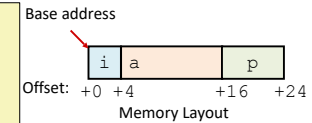
- Compiler determines:
- Total size
 - Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};

struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);

// copy full struct
y = x;

struct rec* z;
z = &y;
```



14

C structs

Like Java class/object without methods.

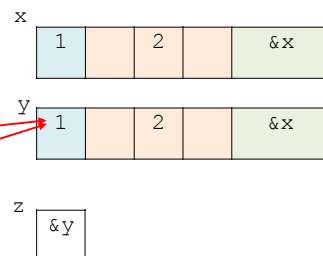
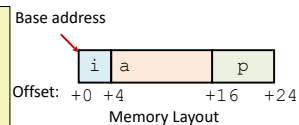
- Compiler determines:
- Total size
 - Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};

struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);

// copy full struct
y = x;

struct rec* z;
z = &y;
(*z).i++;
// same as:
// z->i++
```



15

C structs

Like Java class/object without methods.

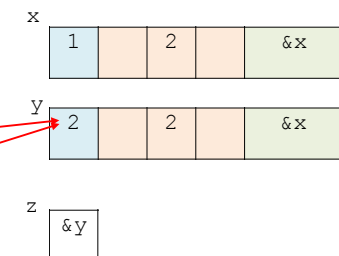
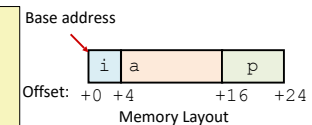
- Compiler determines:
- Total size
 - Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};

struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);

// copy full struct
y = x;

struct rec* z;
z = &y;
(*z).i++;
// same as:
// z->i++
```



16

C: Accessing struct fields

ex

```
struct student {
    int classyear;
    int id;
    char* name;
};
```

Example: traversing a list of struct pointers

```
// Given a null-terminated list of pointers to students,
// return the name of the student with a given ID, or null
// if there is no student with that ID.
char* getStudentNameWithId(struct student *s[], int id) {
    struct student **curr = s;

    while (*curr) {
        if ((*curr)->id == id)
            return (*curr)->name;
        curr++;
    }
    return NULL;
}
```

17

C: Accessing struct fields

ex

```
struct student {
    int classyear;
    int id;
    char* name;
};
```

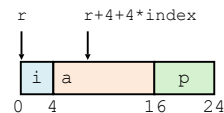
Example: traversing a list of struct pointers

```
// Given a null-terminated list of pointers to students,
// return the name of the student with a given ID, or null
// if there is no student with that ID.
char* getStudentNameWithId(struct student *s[], int id) {
    struct student **curr = s;
    while (*curr) {
        if ((*curr)->id == id)
            return (*curr)->name;
        curr++;
    }
    return NULL;
}
```

18

C: Accessing struct field

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```



```
int get_i_plus_elem(struct rec* r, int index) {
    return r->i + r->a[index];
}
```

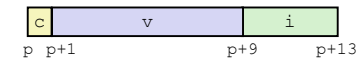
```
movl 0(%rdi),%eax    # Mem[r+0]
addl 4(%rdi,%rsi,4),%eax. # Mem[r+4*index+4]
retq
```

19

C: Struct field alignment

Alignment is especially important for structs

Unaligned Data (not what C does)

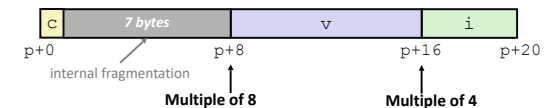


Aligned Data (what C does)

Primitive data type requires K bytes

Address must be multiple of K

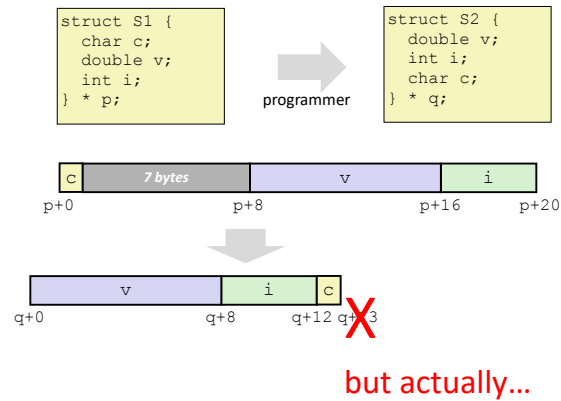
C: align every struct field accordingly.



20

C: Struct packing

Put large data types first:



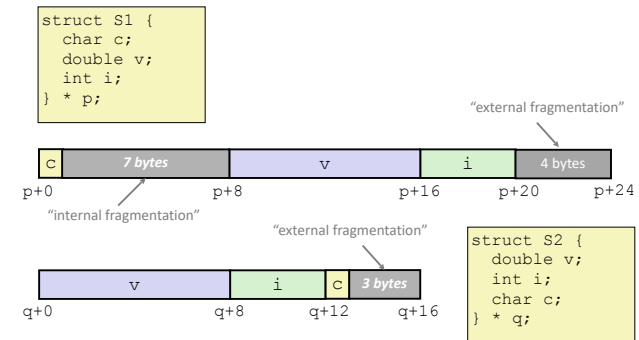
but actually...

21

C: Struct alignment (full)

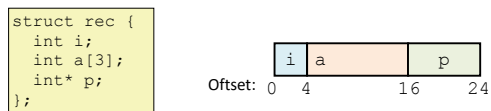
Base *and* total size must align largest internal primitive type.

Fields must align their type's largest alignment requirement.

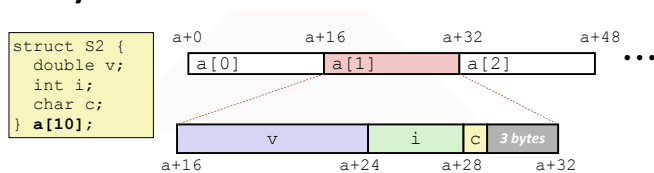


22

Array in struct



Struct in array



23

C: typedef

```
// give type T another name: U
typedef T U;
```

```
// struct types can be verbose
struct Node { ... };
...
struct Node* n = ...;
```

```
// typedef can help
typedef struct Node {
    ...
} Node;

...
Node* n = ...;
```

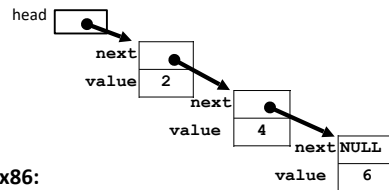
24

Linked Lists

```
typedef
struct Node {
    struct Node* next;
    int value;
} Node;
```

Implement append in x86:

```
void append(Node* head, int x) {
    // assume head != NULL
    Node* cursor = head;
    // find tail
    while (cursor->next != NULL) {
        cursor = cursor->next;
    }
    Node* n = (Node*)malloc(sizeof(Node));
    // error checking omitted
    // for x86 simplicity
    cursor->next = n;
    n->next = NULL;
    n->value = x;
}
```



ex

25

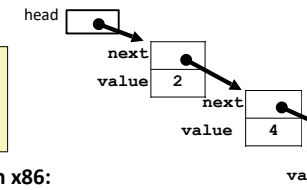
Linked Lists

```
typedef
struct Node {
    struct Node* next;
    int value;
} Node;
```

Implement append in x86:

```
void append(Node* head, int x) {
    // assume head != NULL
    Node* cursor = head;
    // find tail
    while (cursor->next != NULL) {
        cursor = cursor->next;
    }
    Node* n = (Node*)malloc(sizeof(Node));
    // error checking omitted
    // for x86 simplicity
    cursor->next = n;
    n->next = NULL;
    n->value = x;
}
```

Extra fun: try a recursive version too!



```
append:
    pushq %rbp
    movl %esi, %ebp
    pushq %rbx
    movq %rdi, %rbx
    subq $8, %rsp
    jmp .L3
.L6:
    movq %rax, %rbx
.L3:
    movq (%rbx), %rax
    testq %rax, %rax
    jne .L6
    movl $16, %edi
    call malloc
    movq %rax, (%rbx)
    movq $0, (%rax)
    movl %ebp, 8(%rax)
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret
```

ex

26

Where is data stored?

	Registers	Stack	Heap
What?	Fast storage inside the CPU	Contiguous region of memory managed by the compiler, grows downward, one frame per call	Dynamically allocated data, programmer controlled. Always returned from malloc calls.
Size?	On CS Linux: 64 bits (or a subset of those bits)	On CS Linux: grows in multiples of 8. Can store larger data, e.g., an array of 64 bytes	Determined by the argument to malloc
Examples	Function arguments 1-6 Function return values The stack pointer %rsp Operands for instructions, including pointers to heap/stack data	Local variables that don't fit in a register or where an address is needed Function arguments 7+ Return addresses Saved values for caller/callee saved logic.	An array created with malloc then initialized with indexing Linked list nodes created with malloc Both levels of arrays in our zipcode and command array examples

27

Struct practice problem (similar to CSAPP 3.45)

```
struct s {
    char *a;
    short b;
    int *c;
    char d;
    int e;
    char f;
};
```

Recall: a short is 2 bytes in C

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with a at offset +0);

2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).

3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

ex

28

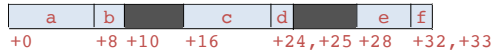
Struct practice problem (similar to CSAPP 3.45)

ex

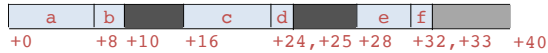
```
struct s {  
    char *a;  
    short b;  
    int *c;  
    char d;  
    int e;  
    char f;  
};
```

Recall: a short is
2 bytes in C

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with a at offset +0);



2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).



3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

