CS 240
Foundations of Computer Systems

WELLESLEY

Representing
Data Structures

Multidimensional arrays

C structs

1

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

2

C: Array layout and indexing

ex

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


+0   +4   +8   +12   +16

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

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

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

1. Assume:
   - Base address of val is in %rdi
   - i is in %rsi

2. Assume:
   - Base address of val is 28(%rsp)
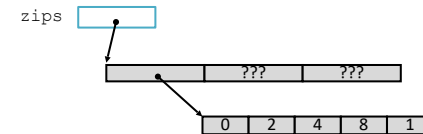   - i is in %rcx

3

3

C: Arrays of pointers to arrays of …

reminder

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

zips

???	???

0	2	4	8	1

```
int[][] zips = new int[3][];          Java
zips[0] = new int[5] {0, 2, 4, 8, 1};
```

4

4

C: Arrays of pointers to arrays in x86

ex

```
                  %rdi        %rsi         %rdx
void copyfromleft(int** zips, long i, long j) {
  zipCodes[i][j] = zipCodes[i][j - 1];
}
```
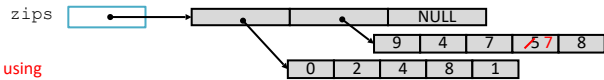
```
copyleft(zips, 1, 3)
```

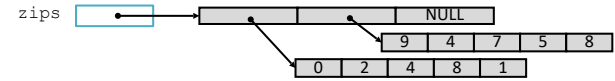zips → NULL
9 4 7 5̶ 7 8
0 2 4 8 1

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)

5

5

C: Arrays of pointers to arrays: Pros/Cons
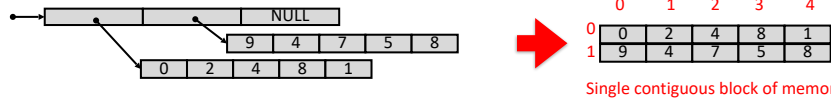
zips → NULL
9 4 7 5 8
0 2 4 8 1

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

6

6

Alternative: row-major nested arrays

NULL
9 4 7 5 8
0 2 4 8 1

→

       0   1   2   3   4
0   0   2   4   8   1
1   9   4   7   5   8
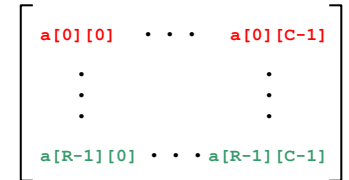
Single contiguous block of memory

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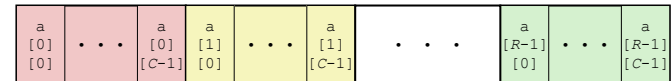- Calculations can be done ahead of time, via arithmetic

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

7

7

C: Row-major nested arrays

$$\begin{bmatrix} a[0][0] & \cdots & a[0][C-1] \\ \vdots & & \vdots \\ a[R-1][0] & \cdots & a[R-1][C-1] \end{bmatrix}$$

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

a[0][0]	$\cdots$	a[0][C−1]	a[1][0]	$\cdots$	a[1][C−1]	$\cdots$	a[R−1][0]	$\cdots$	a[R−1][C−1]

Suppose a's base address is *A*.

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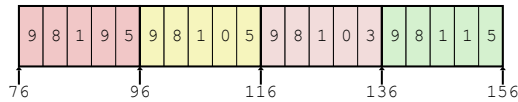*(regular unscaled arithmetic)*

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

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

8

8

C: Strange array indexing examples

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

| 9 | 8 | 1 | 9 | 5 | 9 | 8 | 1 | 0 | 5 | 9 | 8 | 1 | 0 | 3 | 9 | 8 | 1 | 1 | 5 |

76   96   116   136   156

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.

9

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;
};
```

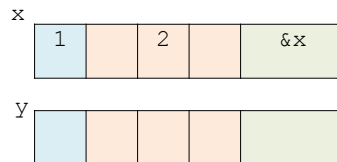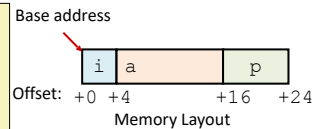10

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);
```

Base address

| i | a | | p |

Offset: +0 +4   +16 +24

Memory Layout

x

| 1 | | 2 | | &x |

y

| | | | |

11

C structs

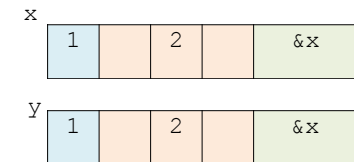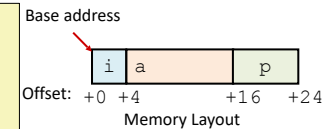Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

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

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

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

Base address

| i | a | | p |

Offset: +0 +4   +16 +24

Memory Layout

x

| 1 | | 2 | | &x |

y

| 1 | | 2 | | &x |

12

C structs

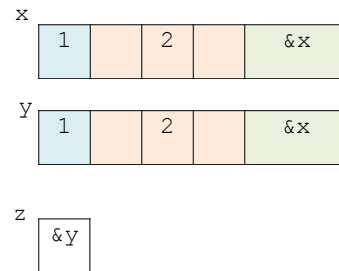Like Java class/object without methods.

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;

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

Base address

i	a		p

Offset: +0 +4 +16 +24

Memory Layout

x

| 1 | | 2 | | &x |

y

| 1 | | 2 | | &x |

z

| &y |

13

C structs

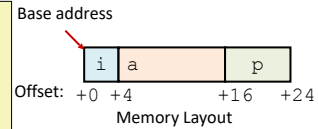Like Java class/object without methods.

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;
(*z).i++;
// same as:
// z->i++
```

Base address

i	a		p

Offset: +0 +4 +16 +24

Memory Layout

x

| 1 | | 2 | | &x |

y

| 1 | | 2 | | &x |

z

| &y |

14

C structs

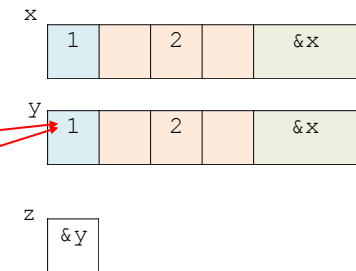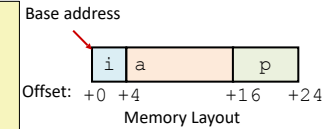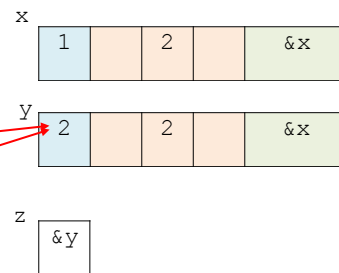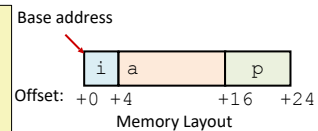Like Java class/object without methods.

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++
```

Base address

i	a		p

Offset: +0 +4 +16 +24

Memory Layout

x

| 1 | | 2 | | &x |

y

| 2 | | 2 | | &x |

z

| &y |

15

C: Accessing struct fields

ex

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

Example: traversing a list of structs

```
// Given a null-terminated list of 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;



}
```

16

C: Accessing struct fields

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

Example: traversing a list of structs

```
// Given a null-terminated list of 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 field

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

r   r+4+4*index

| i | a | | p |

0   4   16   24
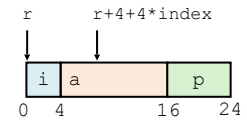
```
int get_i_plus_elem(struct rec* r, int index) {
    return r->i + r->a[index];
}
```

```
movl 0(%rdi),%eax          # Mem[r+0]
addl 4(%rdi,%rsi,4),%eax.   # Mem[r+4*index+4]
retq
```
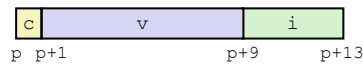
18

C: Struct field alignment

Alignment is especially important for structs
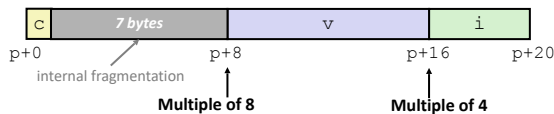
```
struct S1 {
  char c;
  double v;
  int i;
}* p;
```

Defines new struct type and declares variable p of type struct S1*

Unaligned Data (not what C does)

| c | v | i |

p p+1   p+9   p+13

Aligned Data (what C does)

Primitive data type requires **K** bytes
Address must be multiple of **K**
C: align every struct field accordingly.

| c | 7 bytes | v | i |

p+0   p+8   p+16   p+20

internal fragmentation

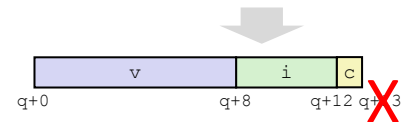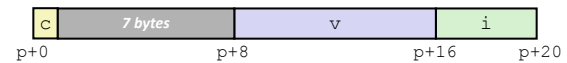Multiple of 8   Multiple of 4

19

C: Struct packing

Put large data types first:

```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

programmer

```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```

| c | 7 bytes | v | i |

p+0   p+8   p+16   p+20

| v | i | c |

q+0   q+8   q+12 q+13   ✗

but actually…

20

C: Struct alignment (full)

Base *and total size* must align largest internal primitive type.
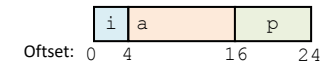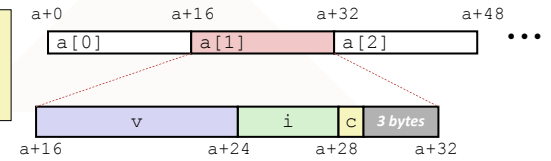Fields must align their type's largest alignment requirement.

```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

"external fragmentation"

c	7 bytes	v	i	4 bytes

p+0   p+8   p+16 p+20   p+24

"internal fragmentation"

"external fragmentation"

v	i	c	3 bytes

q+0   q+8   q+12 q+16

```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```
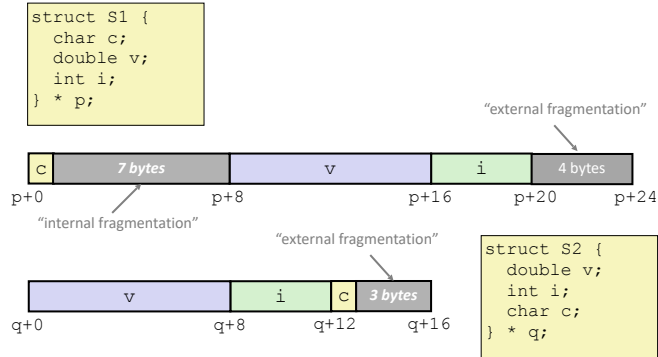
Array in struct

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

i	a	p

Offset: 0 4   16   24

Struct in array

```
struct S2 {
  double v;
  int i;
  char c;
} a[10];
```

a+0   a+16   a+32   a+48

| a[0] | a[1] | a[2] | •••

v	i	c	3 bytes

a+16   a+24   a+28   a+32

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 = ...;
```
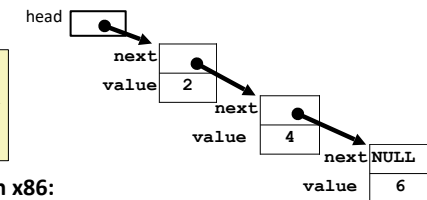

Linked Lists

head

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

next
value 2

next
value 4

next NULL
value 6

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;
}
```

Linked Lists ex

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

head

next
value 2

next
value 4

next
val...

append:
```
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
```

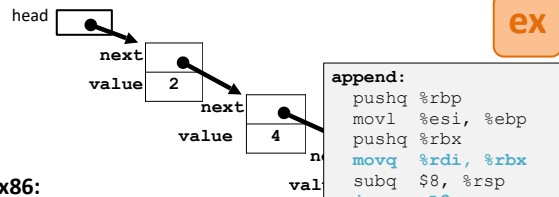**Implement append in x86:**

```
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!
```

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.

Struct practice problem (similar to CSAPP 3.45) ex

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

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);

a	b		c	d		e	f

+0   +8 +10   +16   +24,+25 +28 +32,+33

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).

a	b		c	d		e	f	

+0   +8 +10   +16   +24,+25 +28 +32,+33   +40

Recall: a short is 2 bytes in C

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

a	c	e	b	e f

+0   +8   +16   +20   +24