X86-64

Arrays, Nested Arrays, & Structs

CS 224

Dr. Archibald

Arrays in Assembly

- *T* A[N];
- Given
  - x = starting location of array A
  - L = size (in bytes) of type *T*
- Effects
  - Allocates (N·L) bytes in memory
  - A is now pointer to beginning of array (A = x)

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element *i*
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

Practice Problem 3.36 (solution page 341)

Consider the following declarations:

```
short    S[7];
short   *T[3];
short  **U[6];
int      V[8];
double  *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element size	Total size	Start address	Element i
S	_____	_____	x_S	_____
T	_____	_____	x_T	_____
U	_____	_____	x_U	_____
V	_____	_____	x_V	_____
W	_____	_____	x_W	_____

Pointer Arithmetic

- If p is a pointer to data of type *T* (size L), and p = x
  - p + i = x + L·i
- & - generate pointer for object
  - If E is an object &E gives address of E
- * - dereference pointer to object
  - If AE is address, *AE is value at that address
- E = *&E
- A[i] = *(A + i)

%rdx : address of integer array E
%rcx : integer index i

Expanding on our earlier example, suppose the starting address of integer array E and integer index i are stored in registers %rdx and %rcx, respectively. The following are some expressions involving E. We also show an assembly-code implementation of each expression, with the result being stored in either register %eax (for data) or register %rax (for pointers).

Expression	Type	Value	Assembly code
E	int *	x_E	
E[0]	int	M[x_E]	
E[i]	int	M[$x_E + 4i$]	
&E[2]	int *	$x_E + 8$	
E+i-1	int *	$x_E + 4i - 4$	
*(E+i-3)	int	M[$x_E + 4i - 12$]	
&E[i]-E	long	i	

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index i are stored in registers `%rdx` and `%rcx`, respectively. The following are some expressions involving `E`. We also show an assembly-code implementation of each expression, with the result being stored in either register `%eax` (for data) or register `%rax` (for pointers).

Expression	Type	Value	Assembly code
`E`	`int *`	x_E	`movl %rdx,%rax`
`E[0]`	`int`	$M[x_E]$	`movl (%rdx),%eax`
`E[i]`	`int`	$M[x_E + 4i]$	`movl (%rdx,%rcx,4),%eax`
`&E[2]`	`int *`	$x_E + 8$	`leaq 8(%rdx),%rax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leaq -4(%rdx,%rcx,4),%rax`
`*(E+i-3)`	`int`	$M[x_E + 4i - 12]$	`movl -12(%rdx,%rcx,4),%eax`
`&E[i]-E`	`long`	i	`movq %rcx,%rax`

%rdx : address of short integer array S
%rcx : long integer index i

Practice Problem 3.37 (solution page 341)

Suppose x_S, the address of short integer array `S`, and long integer index i are stored in registers `%rdx` and `%rcx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in register `%rax` if it is a pointer and register element `%ax` if it has data type `short`.

Expression	Type	Value	Assembly code
`S+1`	_____	_____	_____
`S[3]`	_____	_____	_____
`&S[i]`	_____	_____	_____
`S[4*i+1]`	_____	_____	_____
`S+i-5`	_____	_____	_____

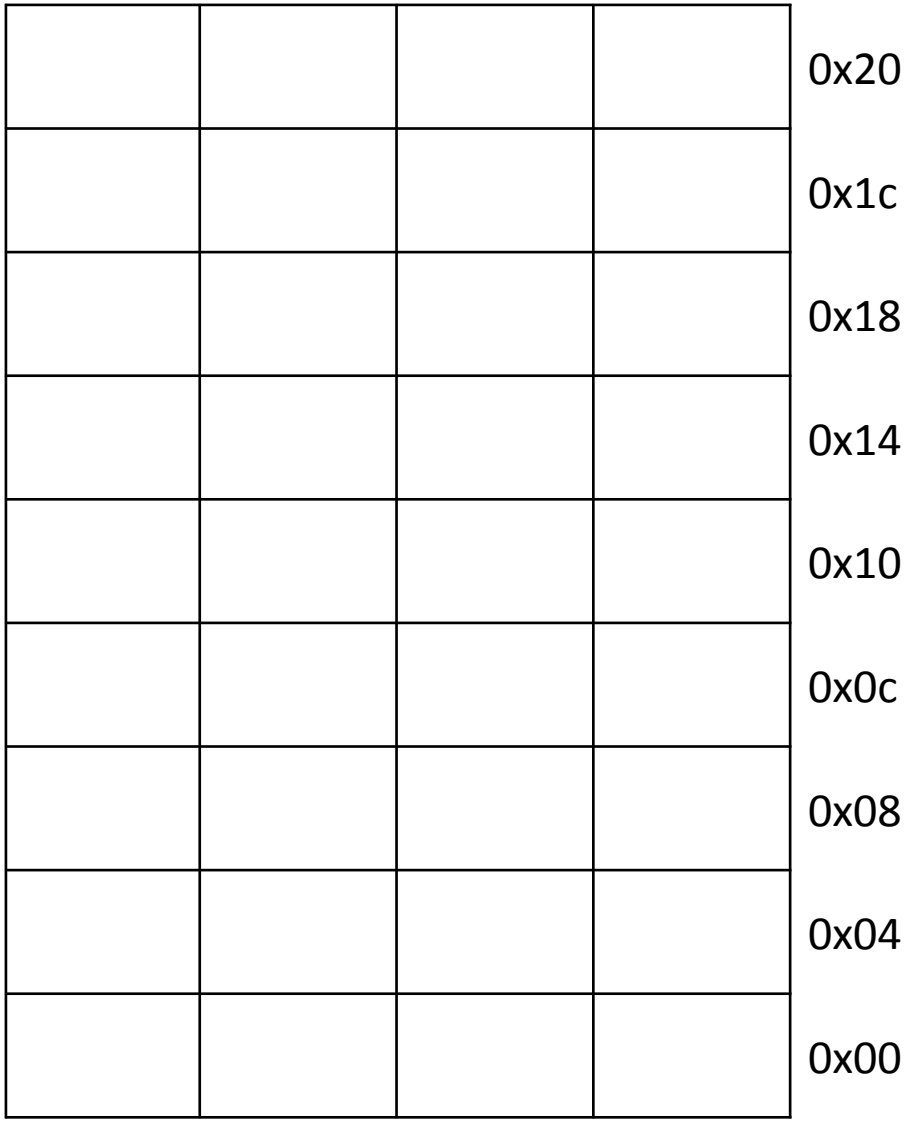
Nested Arrays

- Arrays of Arrays
- Example: int A[5][3]
  - 5 rows, with 3 columns each
- Each Row is an array of 3 integers (3 is # of columns)
  - Size of a row? $L \cdot C$
- A is an array of 5 rows (5 is # of rows)
  - Size of entire array? $L \cdot C \cdot R$
- Stored in *row-major* order

General Case:

- *T* D[R][C];
- &D[i][j] = $x + L(C \cdot i + j) = x + L \cdot C \cdot i + L \cdot j$
  - $x + L \cdot C \cdot i$ is the start of the i-th row
  - $x + L \cdot C \cdot i + L \cdot j$ is the j-th element on the i-th row

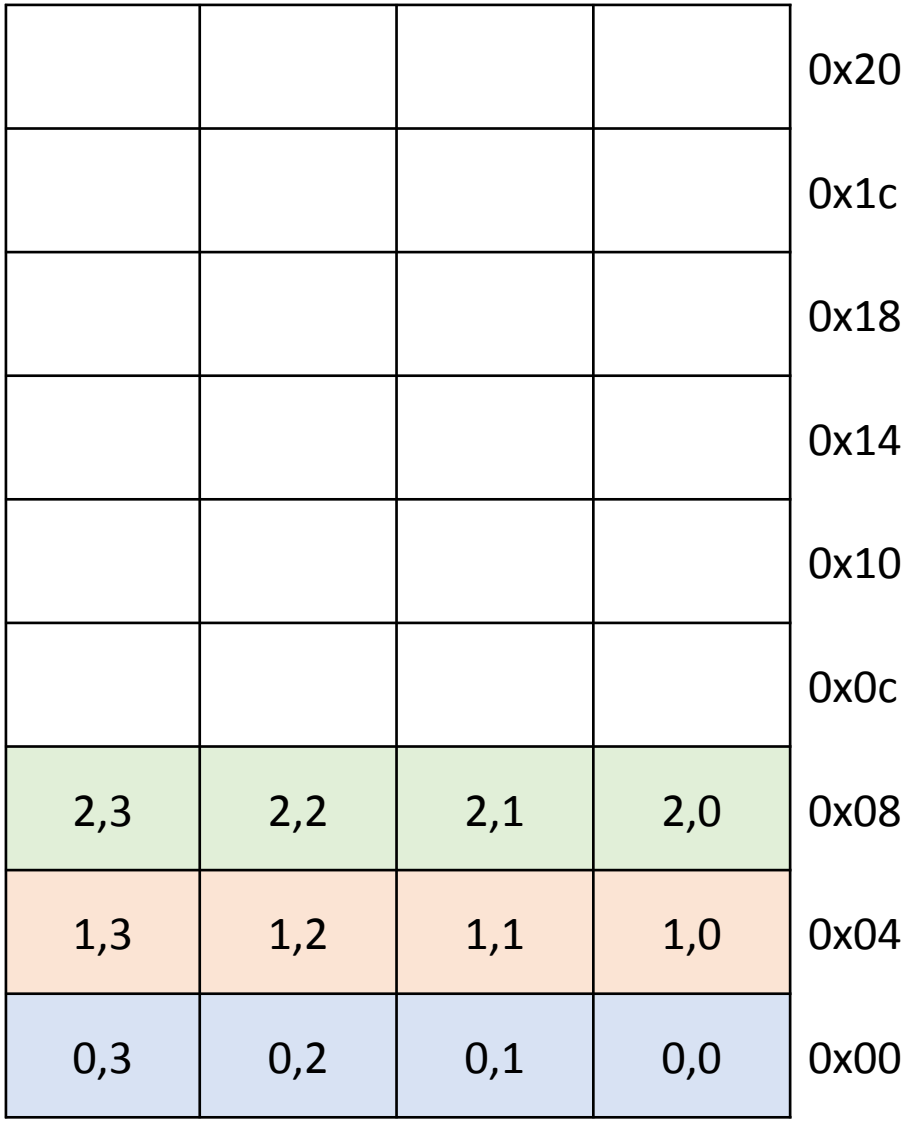
Nested Arrays Example:



char C[3][4]

- Size of each element?
  -
- Size of each row?
  -
- Size of C?
  -
- Address of element C[i][j]?
  -

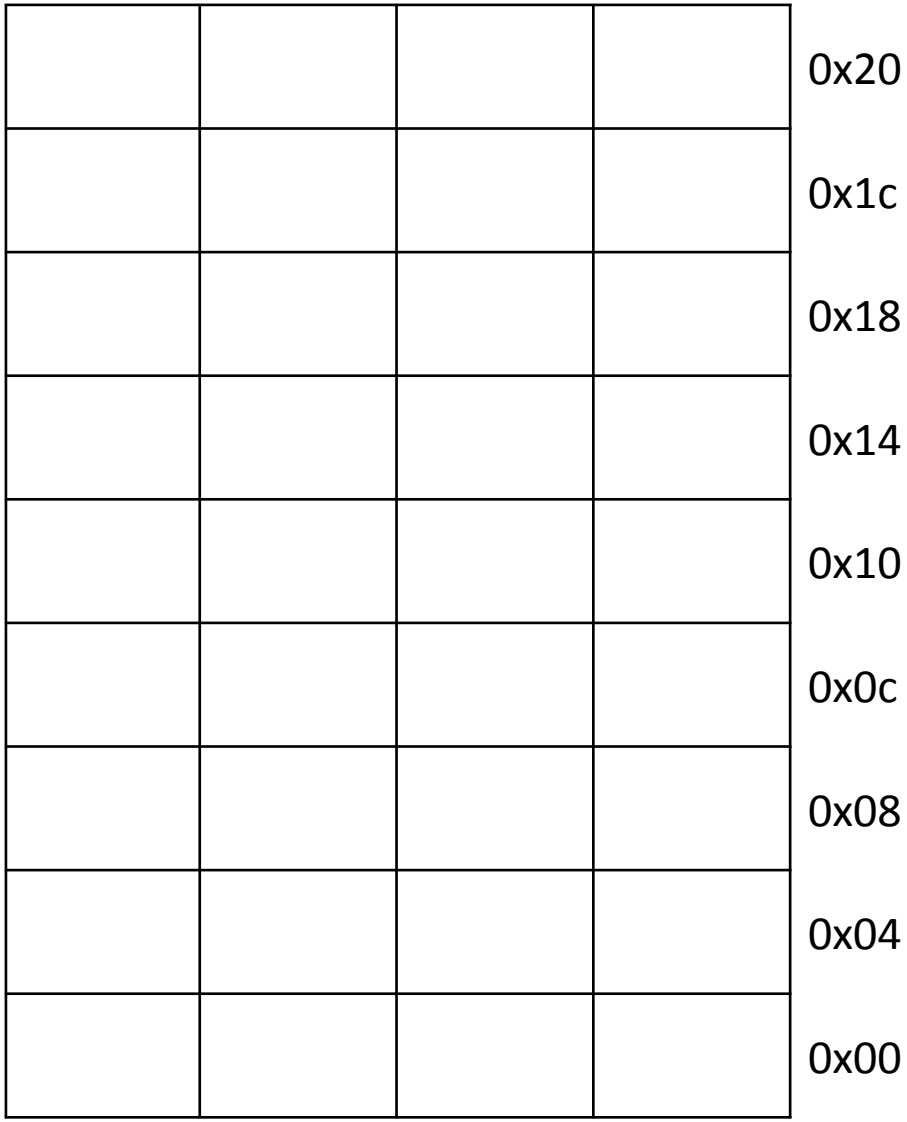
Nested Arrays Example:



char C[3][4]

- Size of each element?
  - 1 byte
- Size of each row?
  - 1 x 4 = 4 bytes
- Size of C?
  - 3 rows x 4 bytes per row = 12 bytes
- Address of element C[i][j]?
  - C + 4*i + j

Nested Arrays Example:



short S[5][3]

- Size of each element?
  -
- Size of each row?
  -
- Size of S?
  -
- Address of element S[i][j]?
  -

Nested Arrays Example:

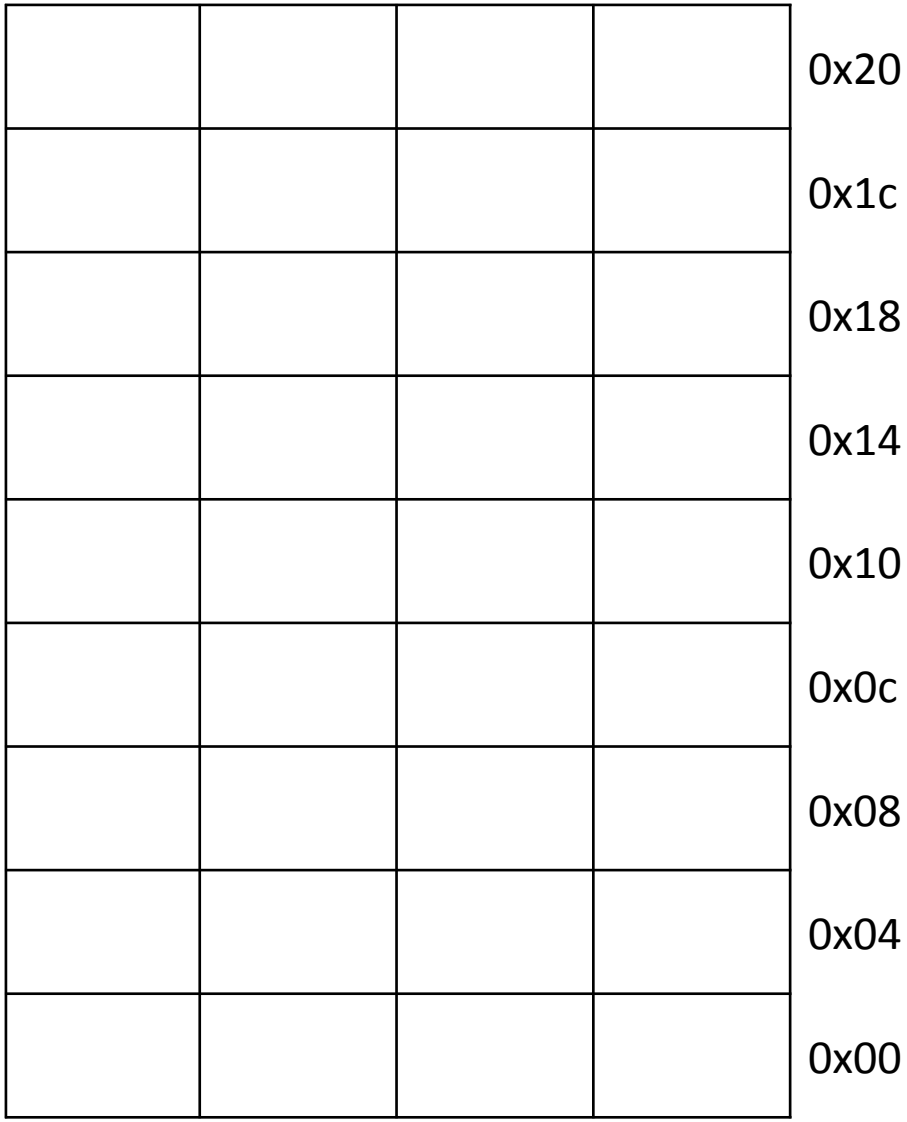
				0x20
		4,2		0x1c
4,1		4,0		0x18
3,2		3,1		0x14
3,0		2,2		0x10
2,1		2,0		0x0c
1,2		1,1		0x08
1,0		0,3		0x04
0,1		0,0		0x00

short S[5][3]

- Size of each element?
  - 2 bytes
- Size of each row?
  - 2 x 3 = 6 bytes
- Size of S?
  - 5 rows x 6 bytes per row = 30 bytes
- Address of element S[i][j]?
  - S + 6*i + 2*j

Nested Arrays Example:

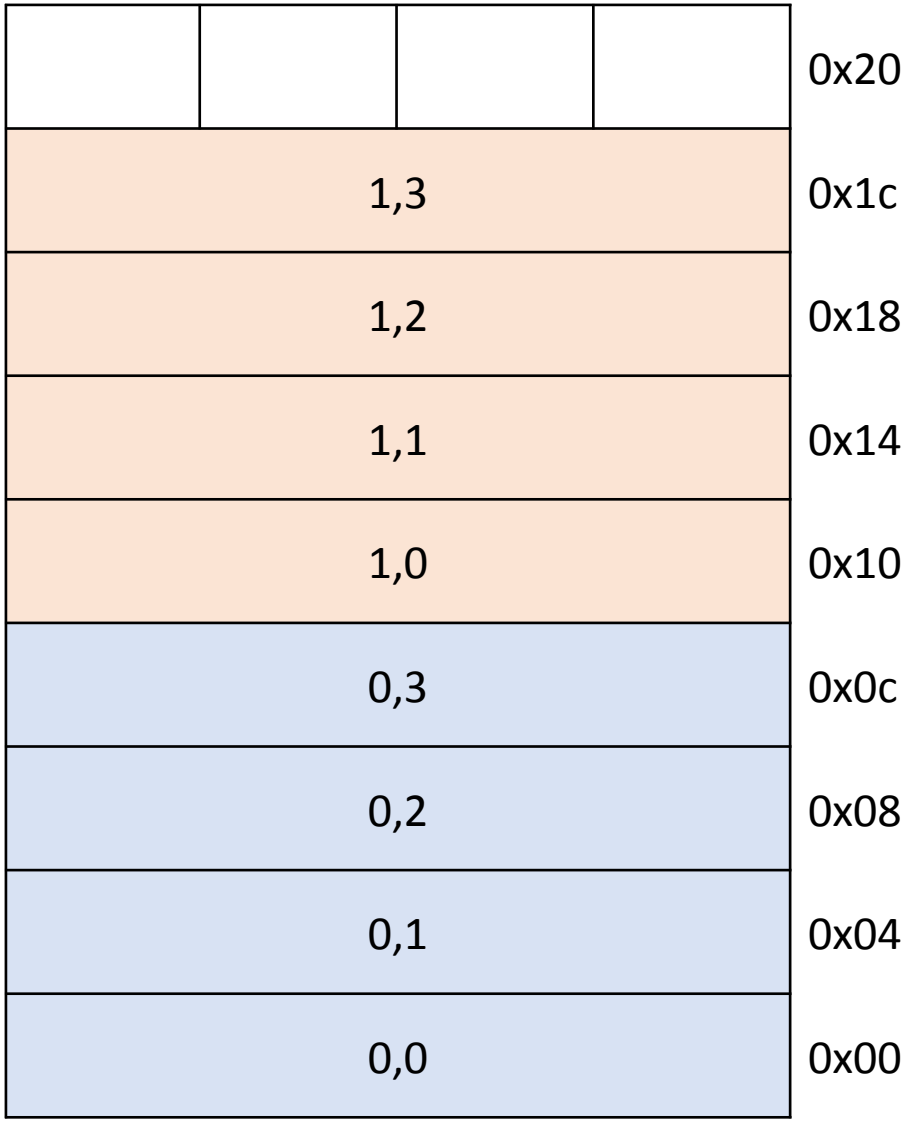
int A[2][4]



- Size of each element?
  -
- Size of each row?
  -
- Size of A?
  -

- Address of element A[i][j]?
  -

Nested Arrays Example:



int A[2][4]

- Size of each element?
  - 4 bytes
- Size of each row?
  - 4 x 4 = 16 bytes
- Size of A?
  - 2 rows x 16 bytes per row = 32 bytes
- Address of element A[i][j]?
  - A + 16*i + 4*j

Practice Problem 3.38 (solution page 341)

Consider the following source code, where M and N are constants declared with `#define`:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
1   sum_element:
2     leaq    0(,%rdi,8), %rdx
3     subq    %rdi, %rdx
4     addq    %rsi, %rdx
5     leaq    (%rsi,%rsi,4), %rax
6     addq    %rax, %rdi
7     movq    Q(,%rdi,8), %rax
8     addq    P(,%rdx,8), %rax
9     ret
```

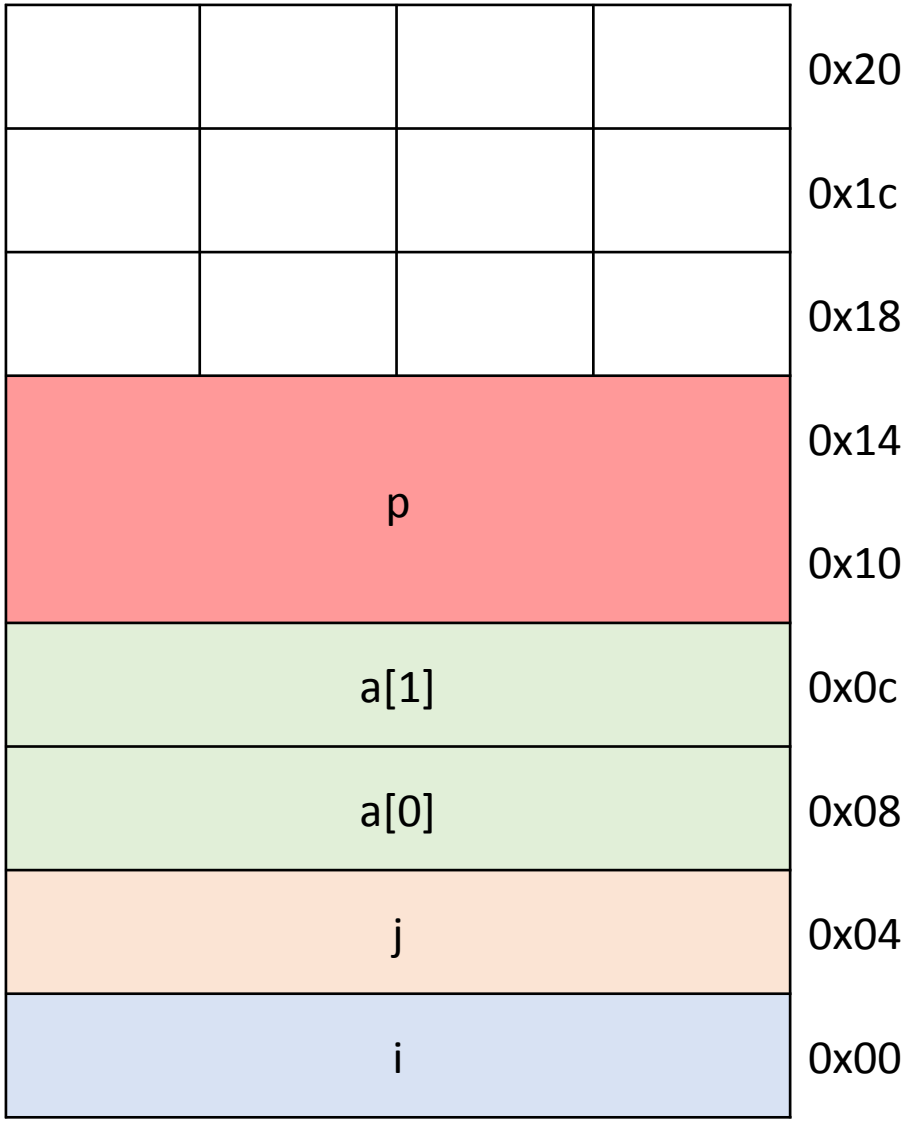
Use your reverse engineering skills to determine the values of M and N based on this assembly code.

Structs

- Data type that groups objects of different types into a single object
- Example :

```
struct rec {
        int i;
        int j;
        int a[2];
        int *p;
};
```

- Arranged in order in memory




```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

Offsets:

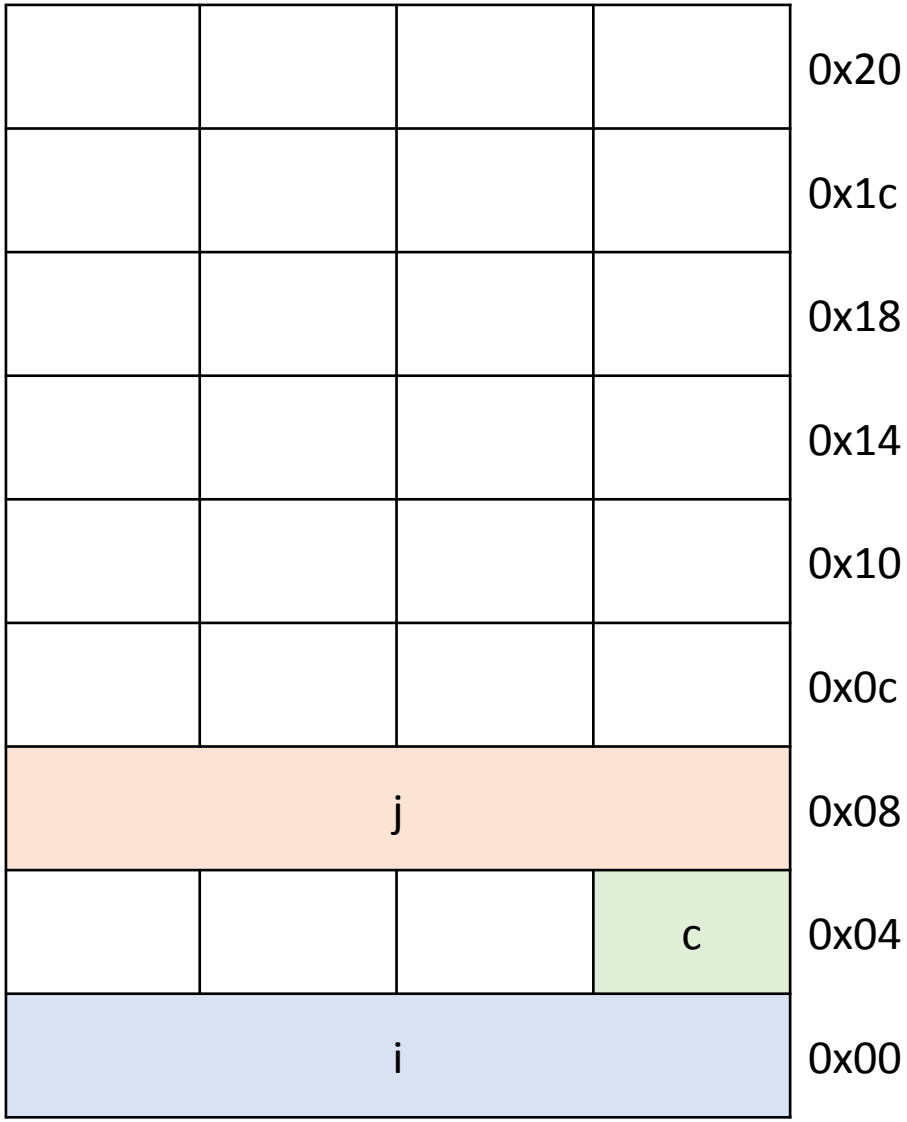
- i : 0x0 bytes
- j : 0x4 bytes
- a : 0x8 bytes
- p : 0x10 bytes

Data Alignment

- Object of *K* bytes must have address that is multiple of *K*

K	type
1	char
2	short
4	int, float
8	long, double, char*

- Spaces/Gaps inserted into structs to fulfil this requirement

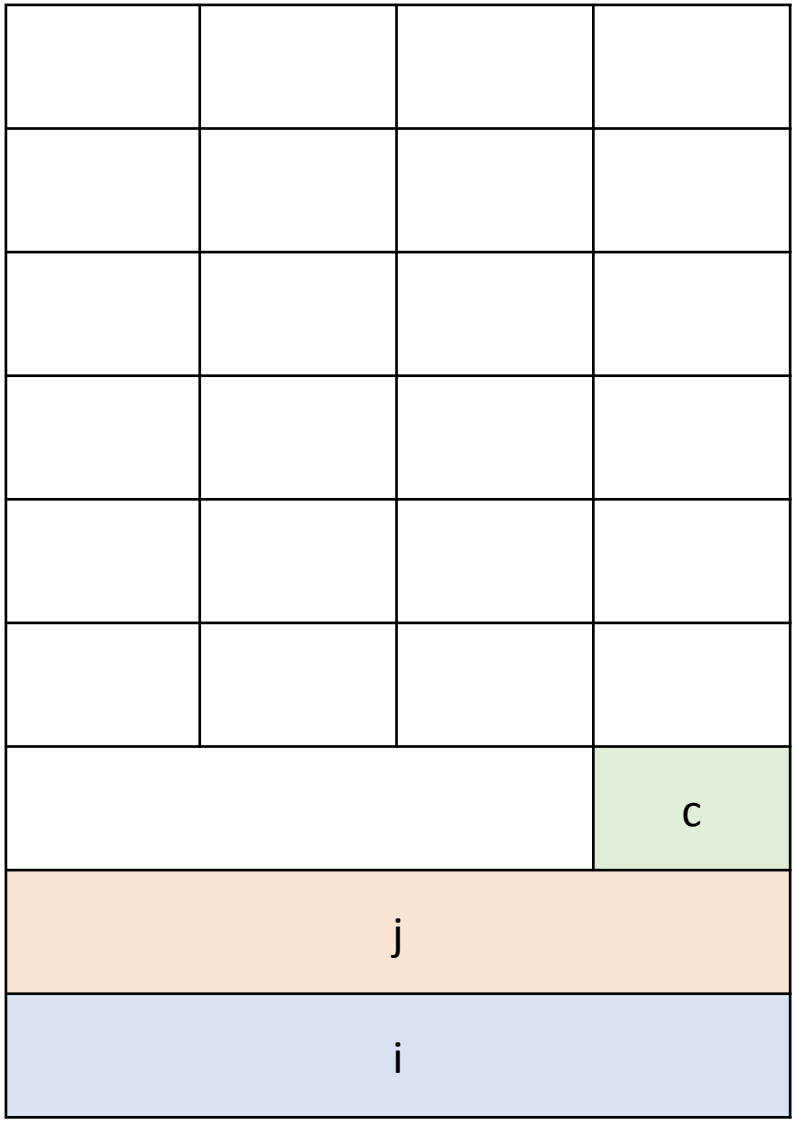



```
struct S1 {
    int i;
    char c;
    int j;
};
```

Gap

Offsets:

- i : 0 bytes
- c : 4 bytes
- j : 8 bytes




```
struct S2 {
    int i;
    int j;
    char c;
};
```

Gap

Offsets:

- i : 0 bytes
- j : 4 bytes
- c : 8 bytes

Size of S2: 12 bytes

So an array of S2s all meet requirements

Practice Problem 3.44 (solution page 345)

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64:

A. `struct P1 { int i; char c; int j; char d; };`

B. `struct P2 { int i; char c; char d; long j; };`

C. `struct P3 { short w[3]; char c[3] };`

D. `struct P4 { short w[5]; char *c[3] };`

E. `struct P5 { struct P3 a[2]; struct P2 t };`

Practice Problem 3.45 (solution page 345)

Answer the following for the structure declaration

```
struct {
    char      *a;
    short      b;
    double     c;
    char       d;
    float      e;
    char       f;
    long       g;
    int        h;
} rec;
```

A. What are the byte offsets of all the fields in the structure?

B. What is the total size of the structure?

C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.