Lecture 7: Data Structures in Assembly

CS 105

February 13, 2019

Array Example

```
int proc (int *p);

int overflow (int x) {
  int a[4];
  a[3] = 10;
  return proc(a);
}
```

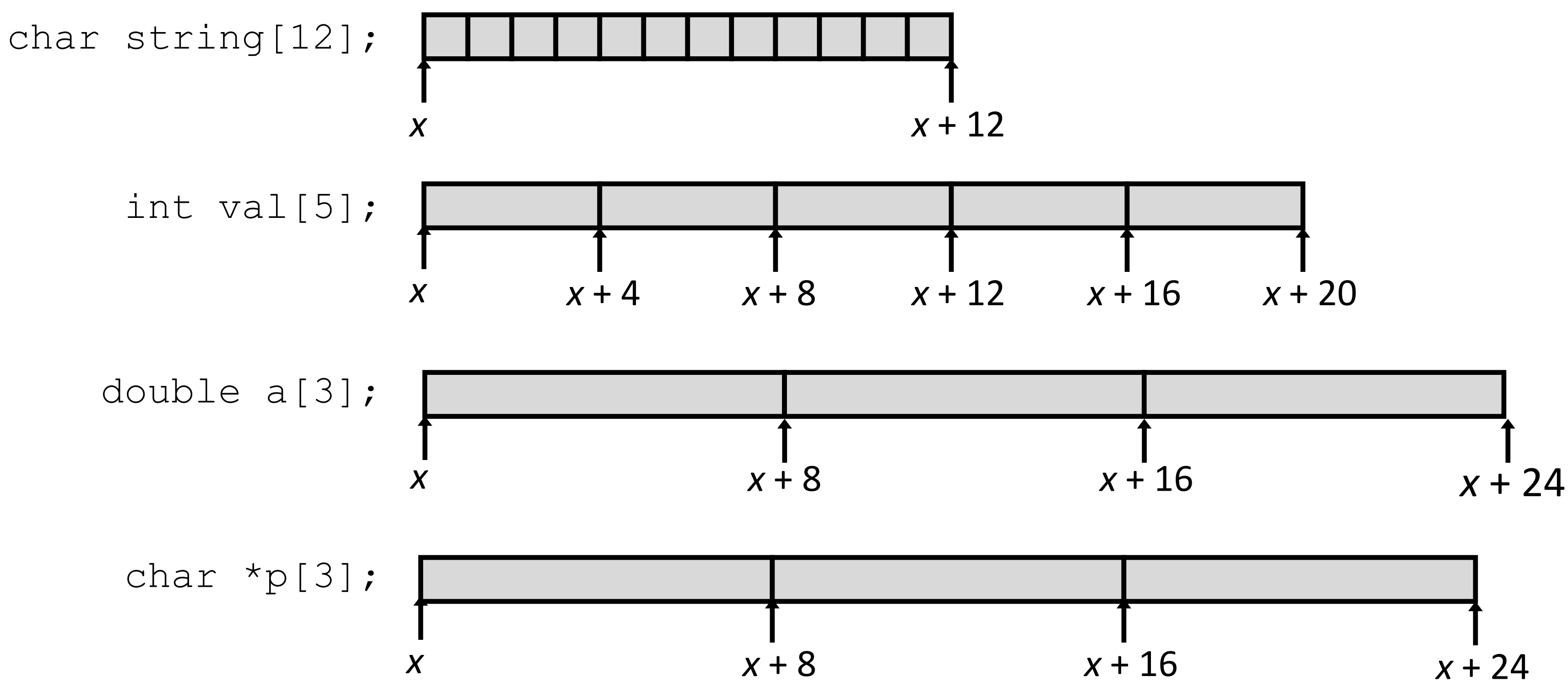
```
overflow:
  subq   $16, %rsp
  movl   $10, 12(%rsp)
  movq   %rsp, %rdi
  call   proc
  addq   $16, %rsp
  ret
```

Array Allocation

- Basic Principle

  T **A[*L*]**;

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory



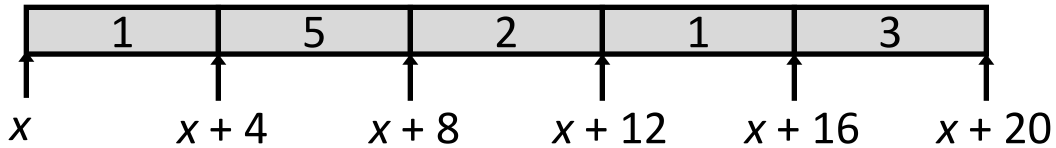
Array Access

- Basic Principle

  T **A[*L*];**

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

`int val[5];`



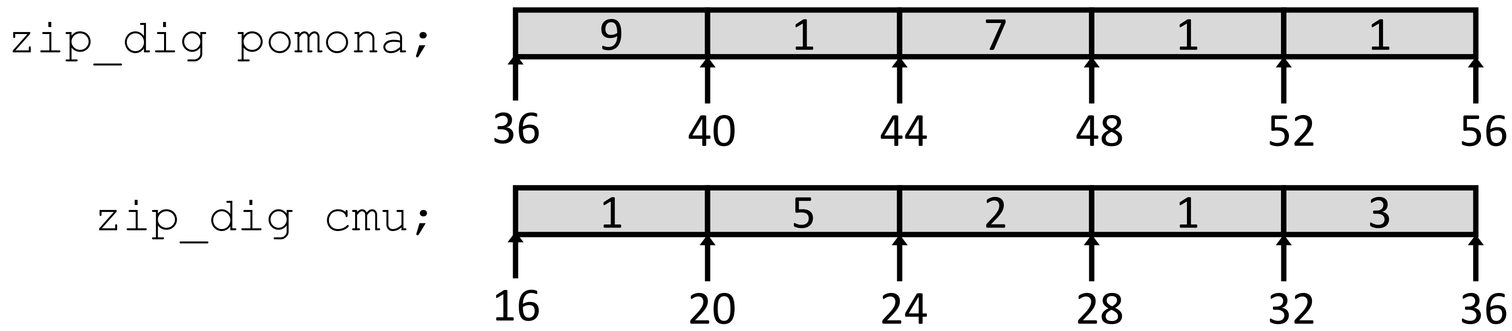
- Reference / Type / Value

Reference	Type	Value
`val[4]`	`int`	
`val`	`int *`	
`val+1`	`int *`	
`&val[2]`	`int *`	
`val[5]`	`int`	
`*(val+1)`	`int`	
`val + i`	`int *`	

Array Example

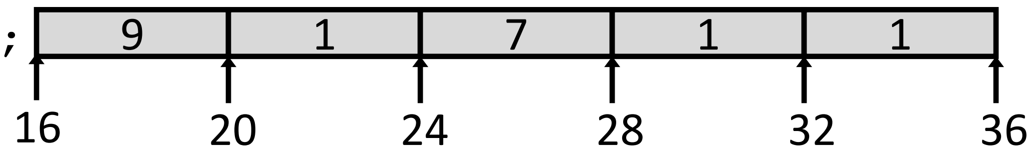
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig pomona = { 9, 1, 7, 1, 1 };
zip_dig cmu = { 1, 5, 2, 1, 3 };
```

- Declaration "`zip_dig pomona`" equivalent to "`int pomona[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general



Array Accessing Example

`zip_dig pomona;`




```
int get_digit(zip_dig z, int digit){
  return z[digit];
}
```

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax   # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at
  `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Array Loop Example

```
void zip_inc(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

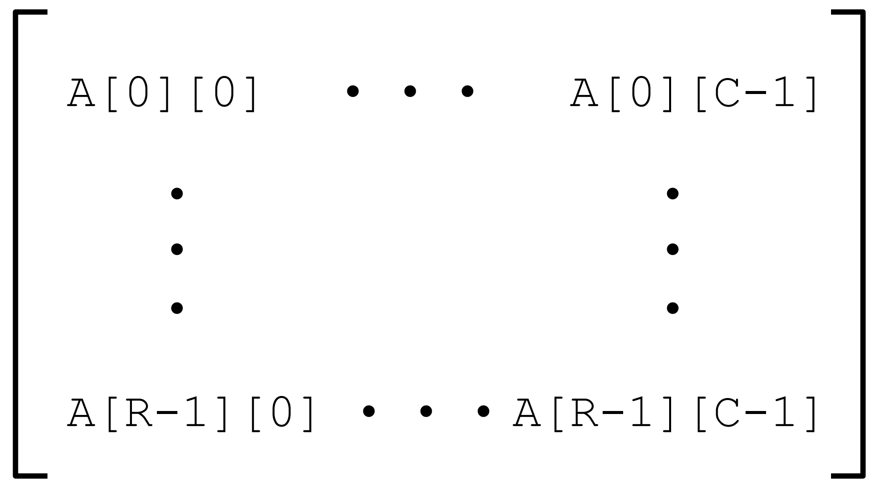
```
  # %rdi = z
  movl    $0, %eax             #   i = 0
  jmp     .L3                  #   goto middle
.L4:                           # loop:
  addl    $1, (%rdi,%rax,4)    #   z[i]++
  addq    $1, %rax             #   i++
.L3:                           # middle
  cmpq    $4, %rax             #   i:4
  jle     .L4                  #   if <=, goto loop
  rep; ret
```

Multidimensional (Nested) Arrays

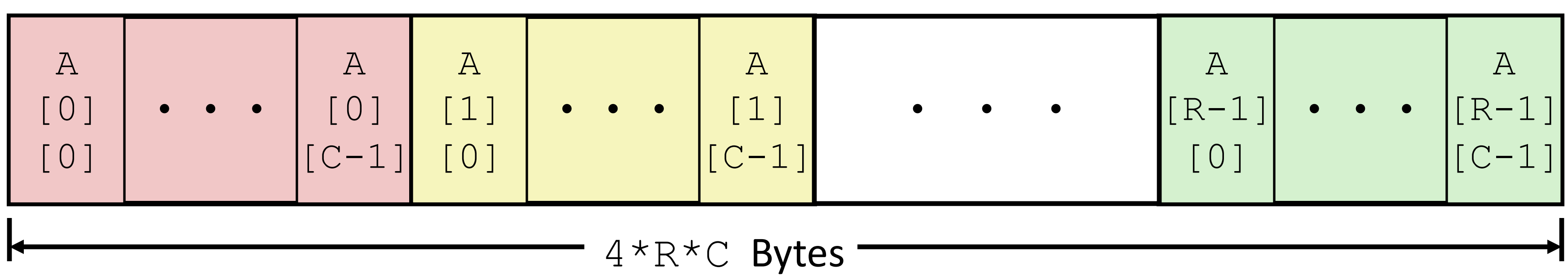
- Declaration

  T **A**[*R*][*C*];

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes
- Arrangement:



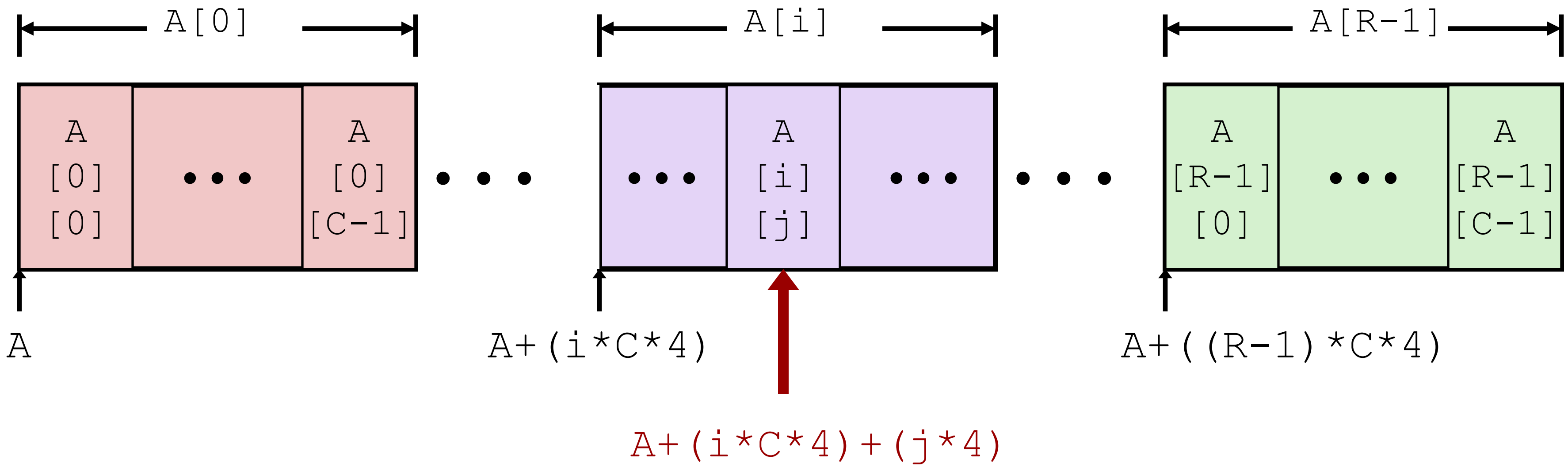

```
int A[R][C];
```



Nested Array Element Access

- Array Elements
  - **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - Address **`A +`** $i * (C * K) + j * K = A + (i * C + j) * K$

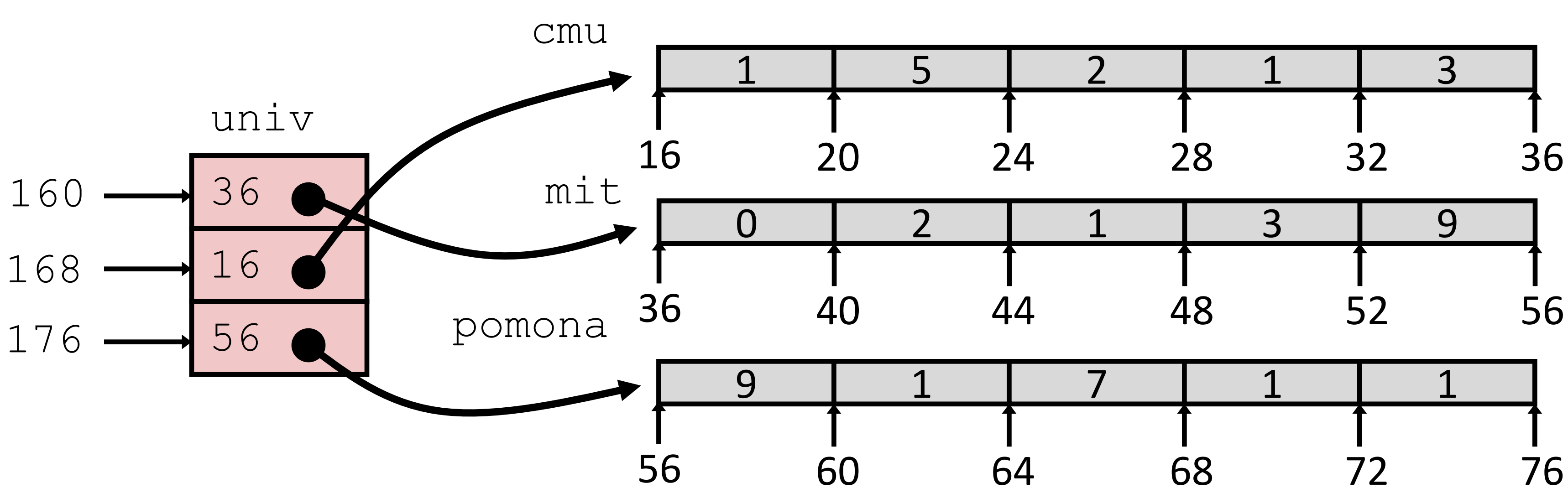
```
int A[R][C];
```



Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig pomona = { 9, 1, 7, 1, 1 };
```

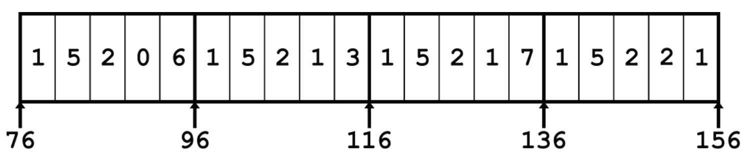
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, pomona};
```



Array Element Accesses

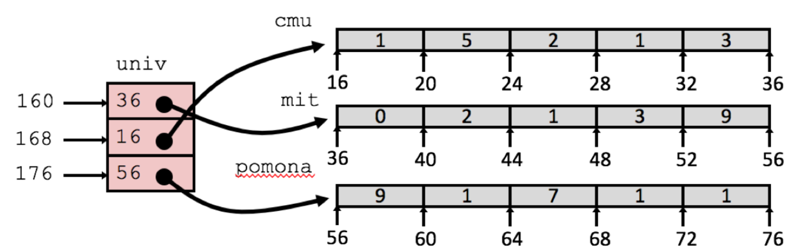
Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```

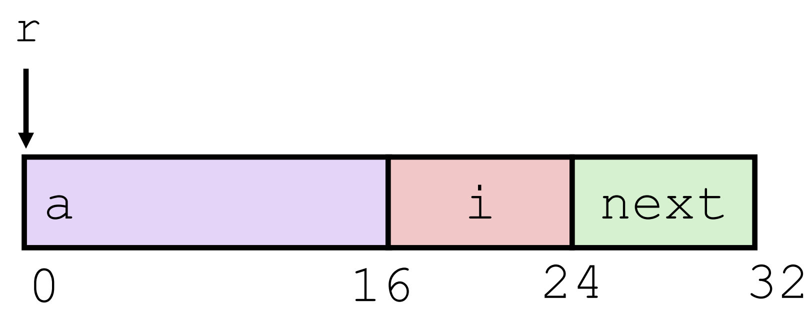


Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`   `Mem[Mem[univ+8*index]+4*digit]`

Structure Representation

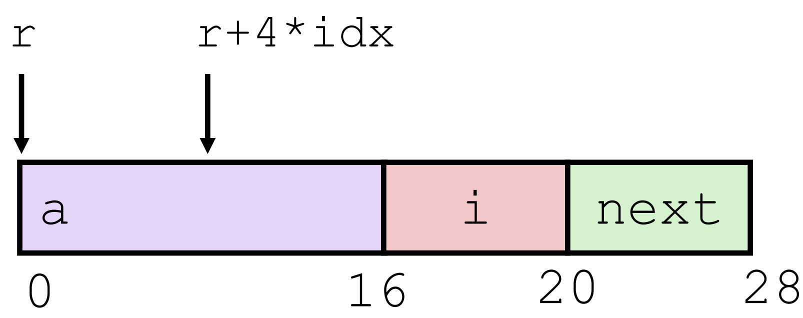
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap(struct rec *r, int idx){
  return &r->a[idx];
}
```

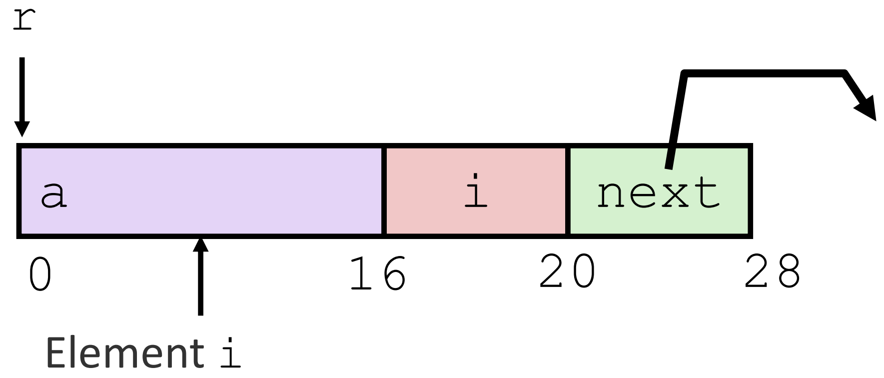
```
   # r in %rdi, idx in %rsi
   leaq  (%rdi,%rsi,4), %rax
   ret
```

Following Linked List

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

- C Code

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	r
%rsi	val

```
.L11:                          # loop:
  movslq  16(%rdi), %rax       #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)  #   M[r+4*i] = val
  movq    20(%rdi), %rdi       #   r = M[r+20]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

Exercise

```
struct ELE {
    long v;
    struct ELE *p;
};
```

```
long fun(struct ELE *ptr);
```

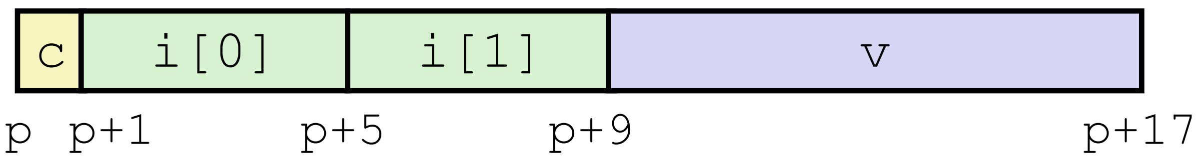
What does the following code do?

```
fun:
  movl $0, %eax
  jmp .L2
.L3:
  addq (%rdi), %rax
  movq 8(%rdi), %rdi
.L2:
  testq %rdi, %rdi
  jne .L3
  rep; ret
```

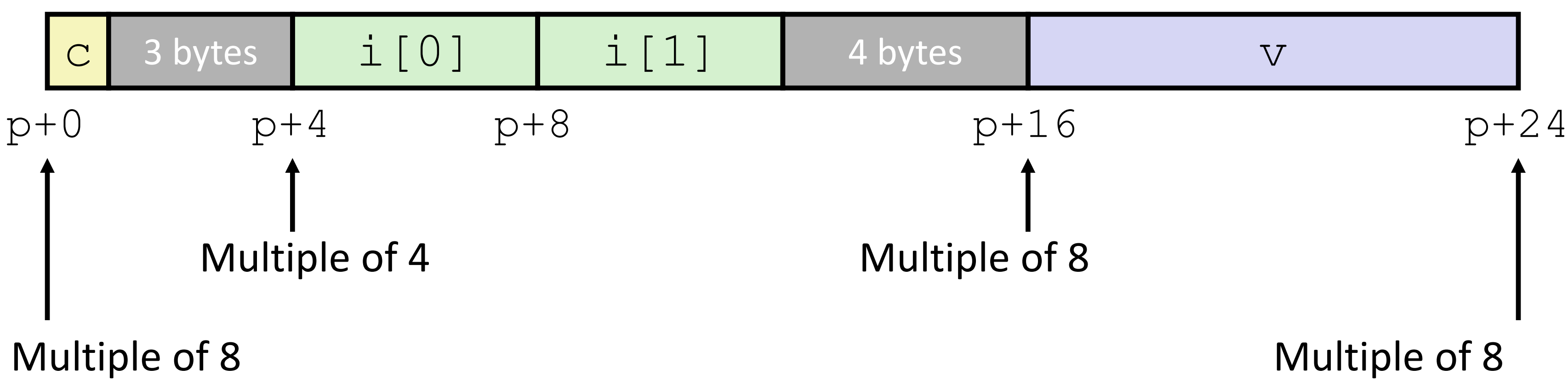
Structures & Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Unaligned Data



- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

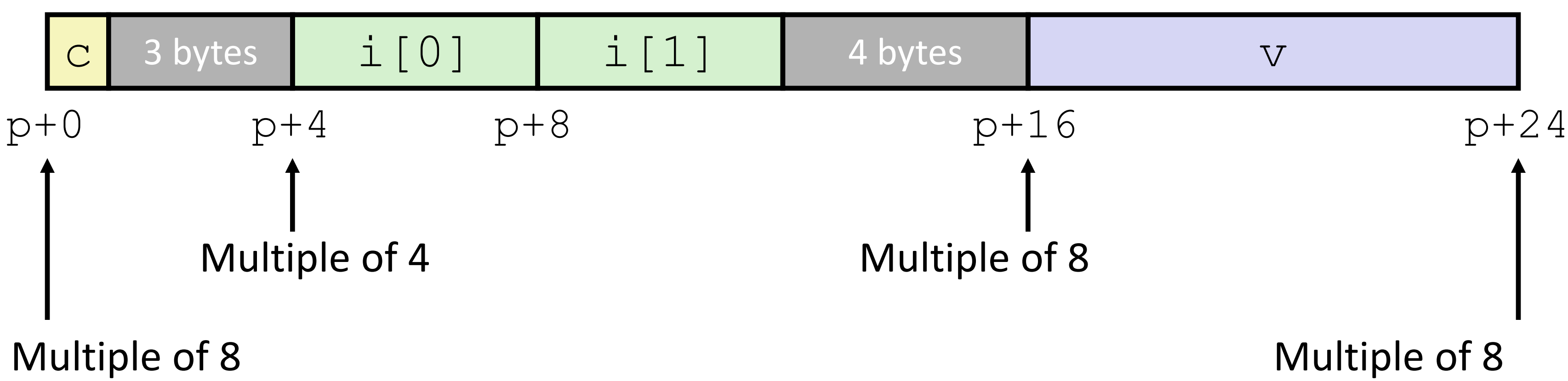
Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, `long`, **`char *`**, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

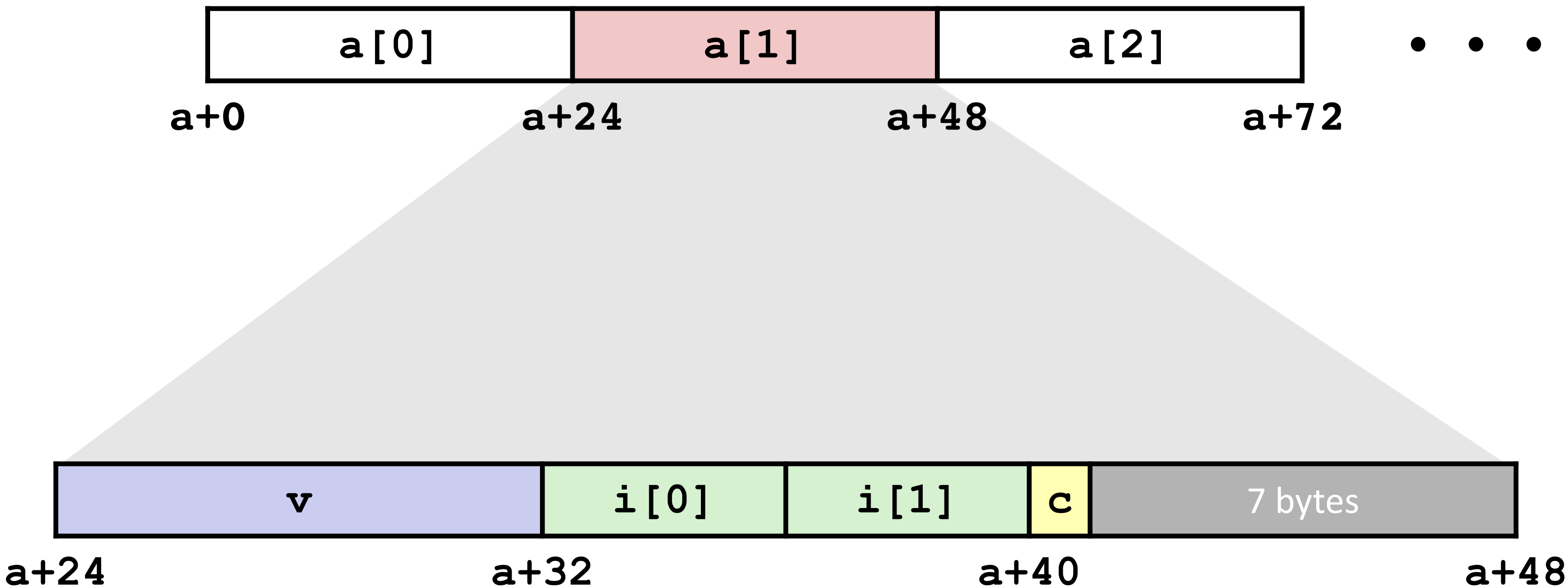
- Within structure:
  - Must satisfy each element’s alignment requirement
- Overall structure placement
  - Each structure has alignment requirement `K`
    - `K` = Largest alignment of any element
  - Initial address & structure length must be multiples of `K`
- Example:
  - K = 8, due to **`double`** element



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Saving Space

- Put large data types first

```
struct S4 {
   char c;
   int i;
   char d;
} *p;
```

```
struct S5 {
   int i;
   char c;
   char d;
} *p;
```

- Effect (K=4)