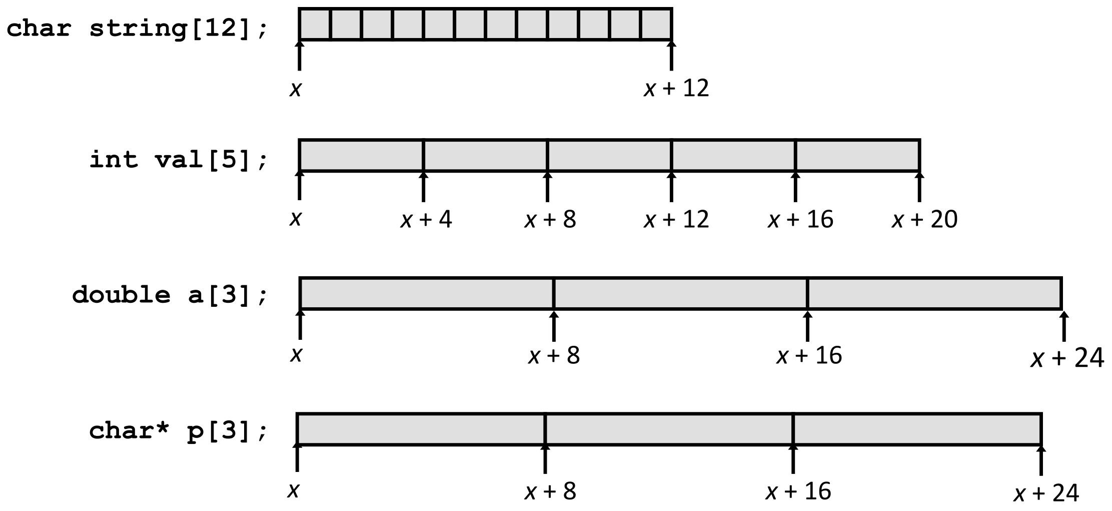
Array Allocation



```
char string[12];
```

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

```
double a[3];
```

```
char* p[3];
```

Array Access

```
int get_val(int a[], int i) {
  return a[i];
}
```

```
  # %rdi = a
  # %rsi = i
movl (%rdi,%rsi,4), %eax  # a[i]
```

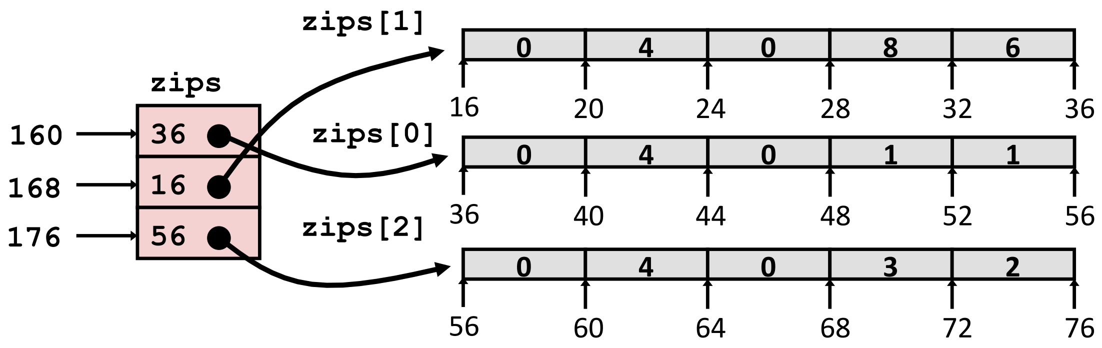
Array Looping

```
void inc5(int a[]) {
  size_t i;
  for (i = 0; i < 5; i++)
    a[i]++;
}
```

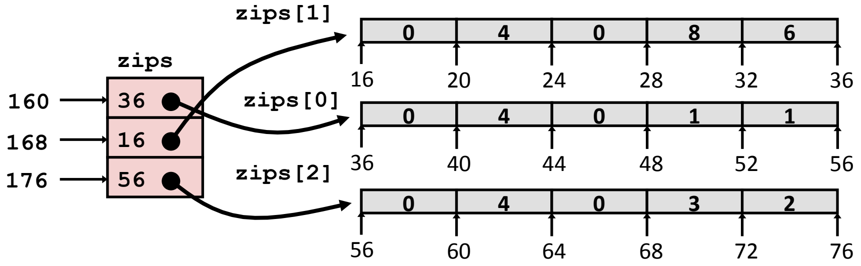
```
  # %rdi = a
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   a[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe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

Multi-Level Array Example

```
int* zips[3];
zips[0] = (int*) malloc(sizeof(int)*5);
...
```



Multi-Level Array Example

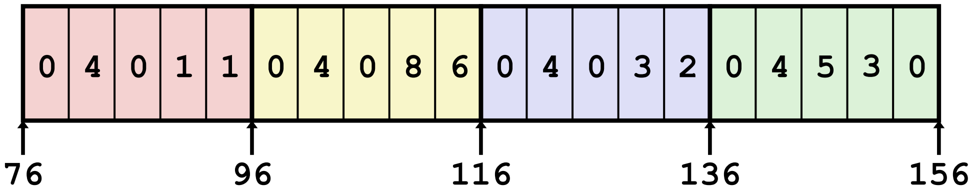


```
int get_zip_digit
  (size_t index, size_t digit)
{
  return zips[index][digit];
}
```

```
salq    $2, %rsi               # 4*digit
addq    160(,%rdi,8), %rsi     # p = zips[index] + 4*digit
movl    (%rsi), %eax           # return *p
ret
```

Nested Array Example

`int zips[4][5];`

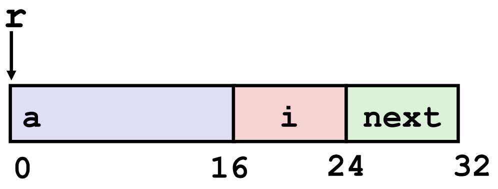


```
int* get_zip(int index)
{
  return zips[index];
}
```

```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq 76(,%rax,4),%rax   # zips + (20 * index)
```

C Structs

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



```
struct rec x;
struct rec y;

x.i = 5;
x.a[1] = 2;
x.next = &y;

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

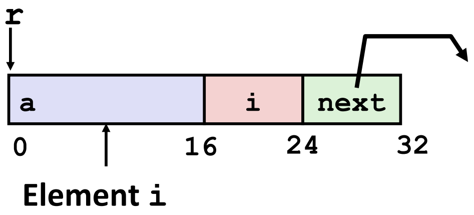
```
struct rec* z;
z = &y; // just a pointer copy

// form 1
(*z).i = 7; // NOT z.i = 7;

// form 2 (preferred)
z->i = 7;
```

typedef

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


// example: defines a type “struct rec”
// and typedefs it with another name “rec”
typedef struct rec {
  ...
} rec;


rec x; // now can omit “struct”
x.i = 5;


rec* p = (rec*) malloc(sizeof(rec));
p->i = 3;
```

Linked List Example

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



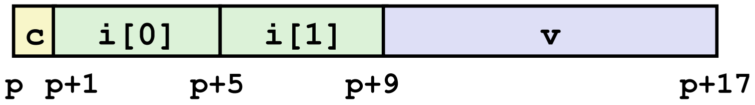
```
void set_val
  (struct rec* r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

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    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

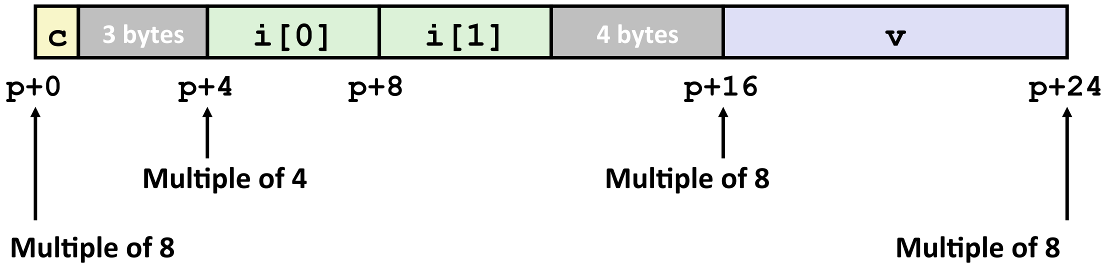
Data Alignment

Unaligned Data



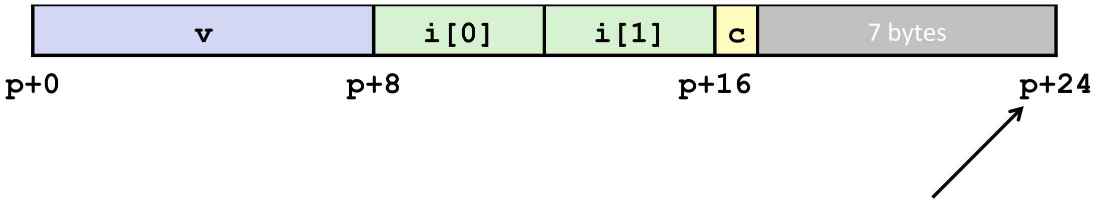
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Aligned Data



Struct Data Alignment

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



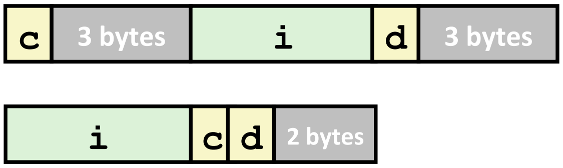
Multiple of 8 (largest alignment in struct)

Saving Space

Put large data types first

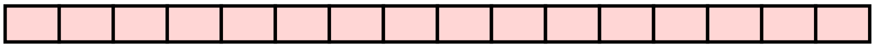
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

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

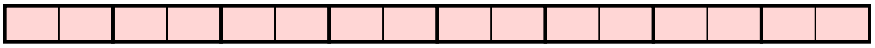


Floating Point: YMM/XMM Registers

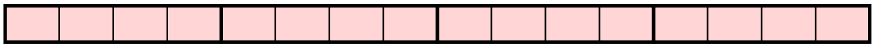
- 16 single-byte integers



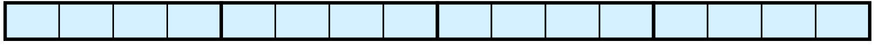
- 8 16-bit integers



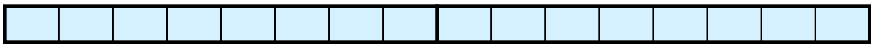
- 4 32-bit integers



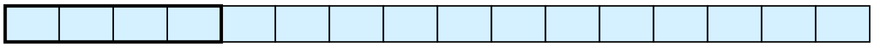
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float



- 1 double-precision float