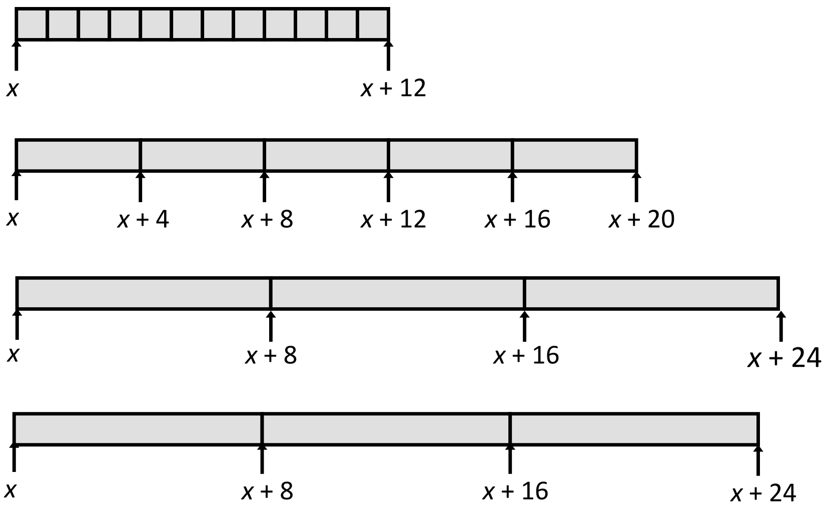
Array Allocation



`char string[12];`

x   x + 12

`int val[5];`

x   x + 4   x + 8   x + 12   x + 16   x + 20

`double a[3];`

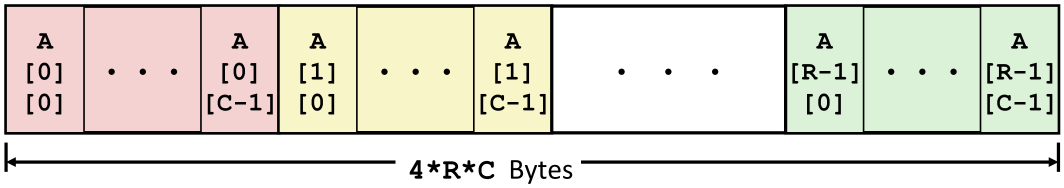
x   x + 8   x + 16   x + 24

`char* p[3];`

x   x + 8   x + 16   x + 24

Nested Arrays

`int A[R][C];`

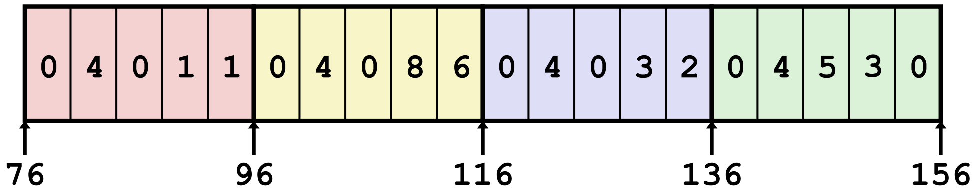


A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

4*R*C Bytes

Nested Array Example

```
int zips[4][5];
```

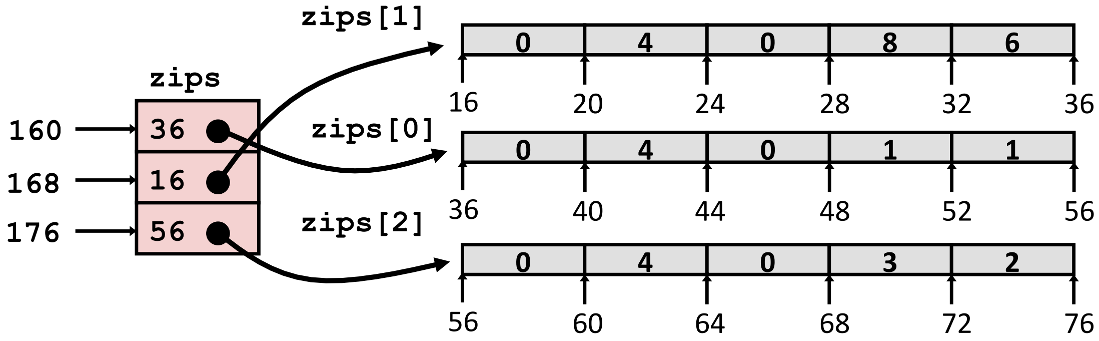


```
int* get_zip(int index)
{
  return zips[index];
}
```

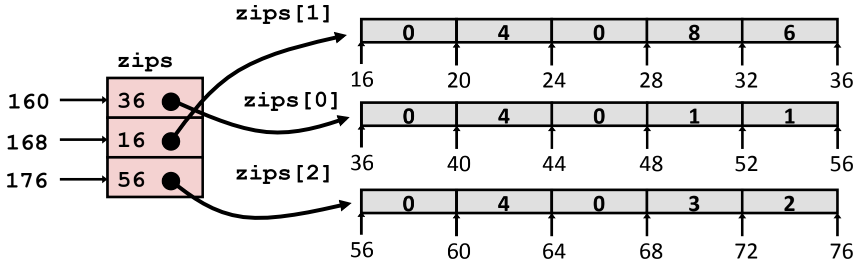
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq 76(,%rax,4),%rax   # zips + (20 * index)
```

Multi-Level Array Example (1)

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



Multi-Level Array Example (2)

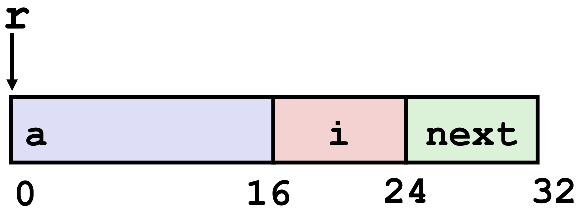


```
int get_zip_digit
  (size_t index, size_t digit)
{
  return zips[index][digit];
}
```

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

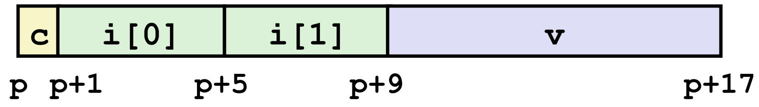
Structures

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



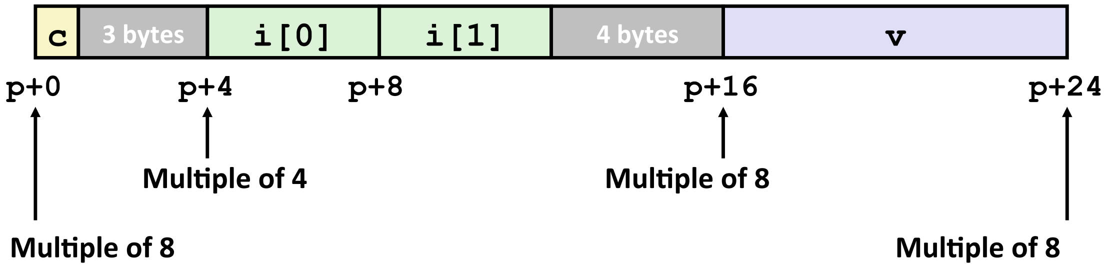
Data Alignment

Unaligned Data



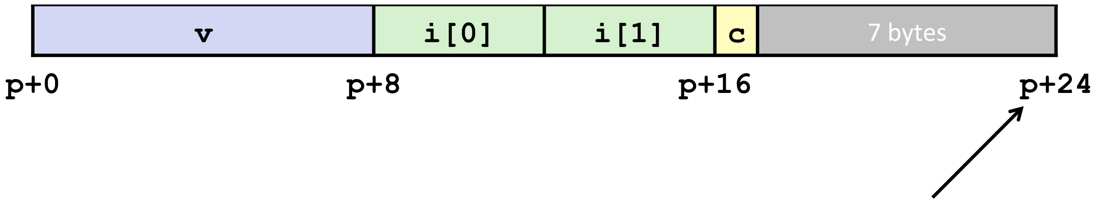
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Aligned Data



Struct Data Alignment

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



Multiple of 8 (largest alignment in struct)

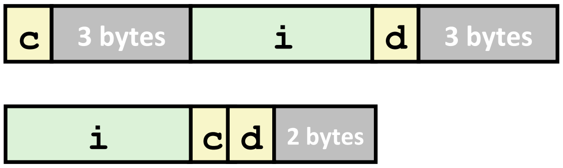
Saving Space

Put large data types first

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

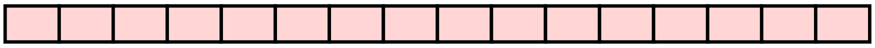
→

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

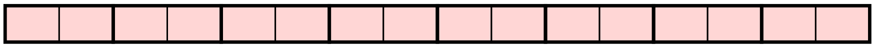


Floating Point: YMM/XMM Registers

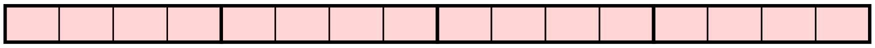
- 16 single-byte integers



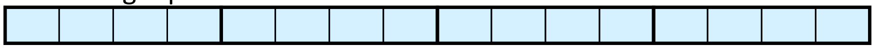
- 8 16-bit integers



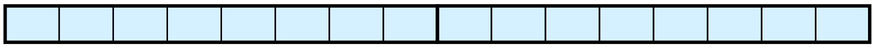
- 4 32-bit integers



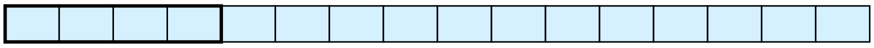
- 4 single-precision floats



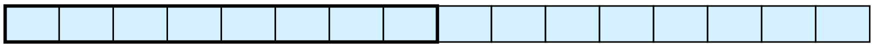
- 2 double-precision floats



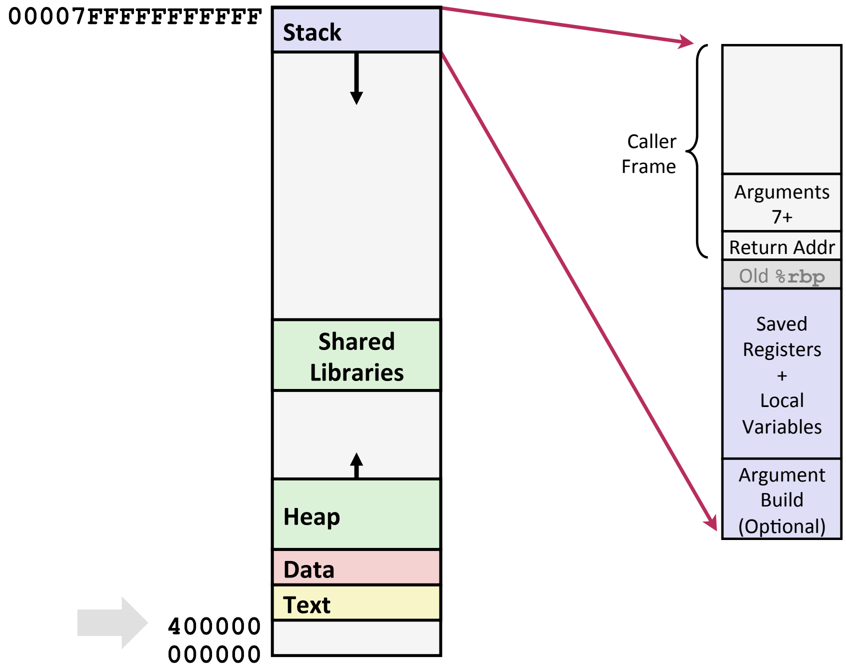
- 1 single-precision float



- 1 double-precision float



x86-64 Linux Memory Layout



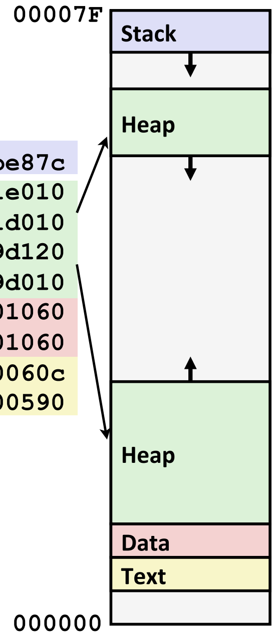
Memory Allocation Example

```
char big_array[1L<<24];
char huge_array[1L<<31];

int foo() { return 0; }

int main() {
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28);
    p2 = malloc(1L << 8);
    p3 = malloc(1L << 32);
    p4 = malloc(1L << 8);
 /* Some print statements ... */
}
```

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
foo()	0x0000000000400590



String Library Code

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

See also: `strcpy`, `strcat`, `scanf`, `fscanf`, `sscanf`, …

Vulnerable Buffer Code

```
/* Echo Line */
void echo() {
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./buftest
Type a string:012345678901234567890123
012345678901234567890123
```

```
unix>./buftest
Type a string:0123456789012345678901234
Segmentation Fault
```

Buffer Overflow Assembly

echo:

```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18              sub    $0x18,%rsp
 4006d3:  48 89 e7                 mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff           callq  400680 <gets>
 4006db:  48 89 e7                 mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff           callq  400520 <puts@plt>
 4006e3:  48 83 c4 18              add    $0x18,%rsp
 4006e7:  c3                       retq
```

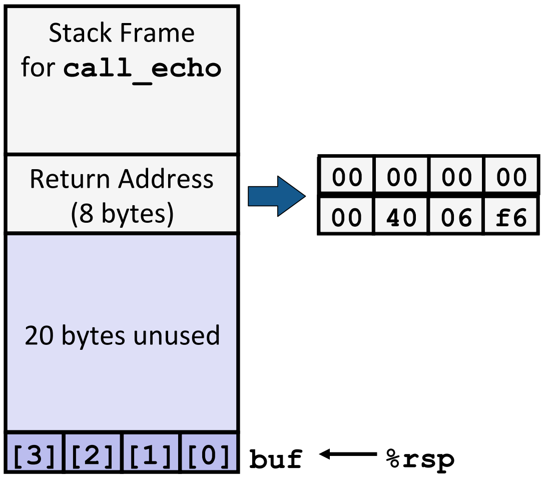
call_echo:

```
4006e8:   48 83 ec 08              sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00           mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff           callq  4006cf <echo>
 4006f6:  48 83 c4 08              add    $0x8,%rsp
 4006fa:  c3                       retq
```

Buffer Overflow Stack

```
/* Echo Line */
void echo() {
    char buf[4];
    gets(buf);
    puts(buf);
}
```

Before call to gets

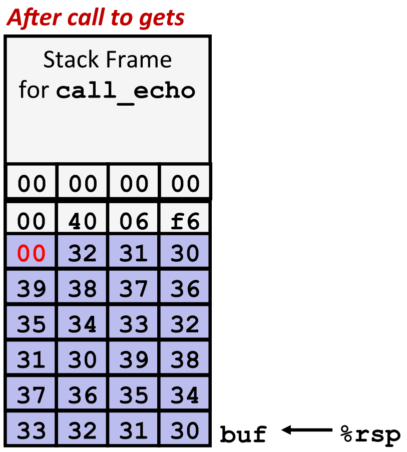


```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
 . . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
 . . .
```

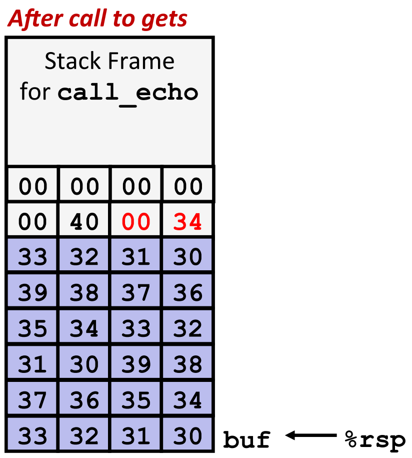
Buffer Overflow Examples



After call to gets

Stack Frame for `call_echo`

00	00	00	00	
00	40	06	f6	
00	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ← %rsp



After call to gets

Stack Frame for `call_echo`

00	00	00	00	
00	40	00	34	
33	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ← %rsp

```
unix>./buftest
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed, but did not corrupt state

```
unix>./buftest
Type a string:0123456789012345678901234
Segmentation Fault
```

Overflowed and corrupted return pointer