CSC 252/452: Computer Organization
Fall 2025: Lecture 9

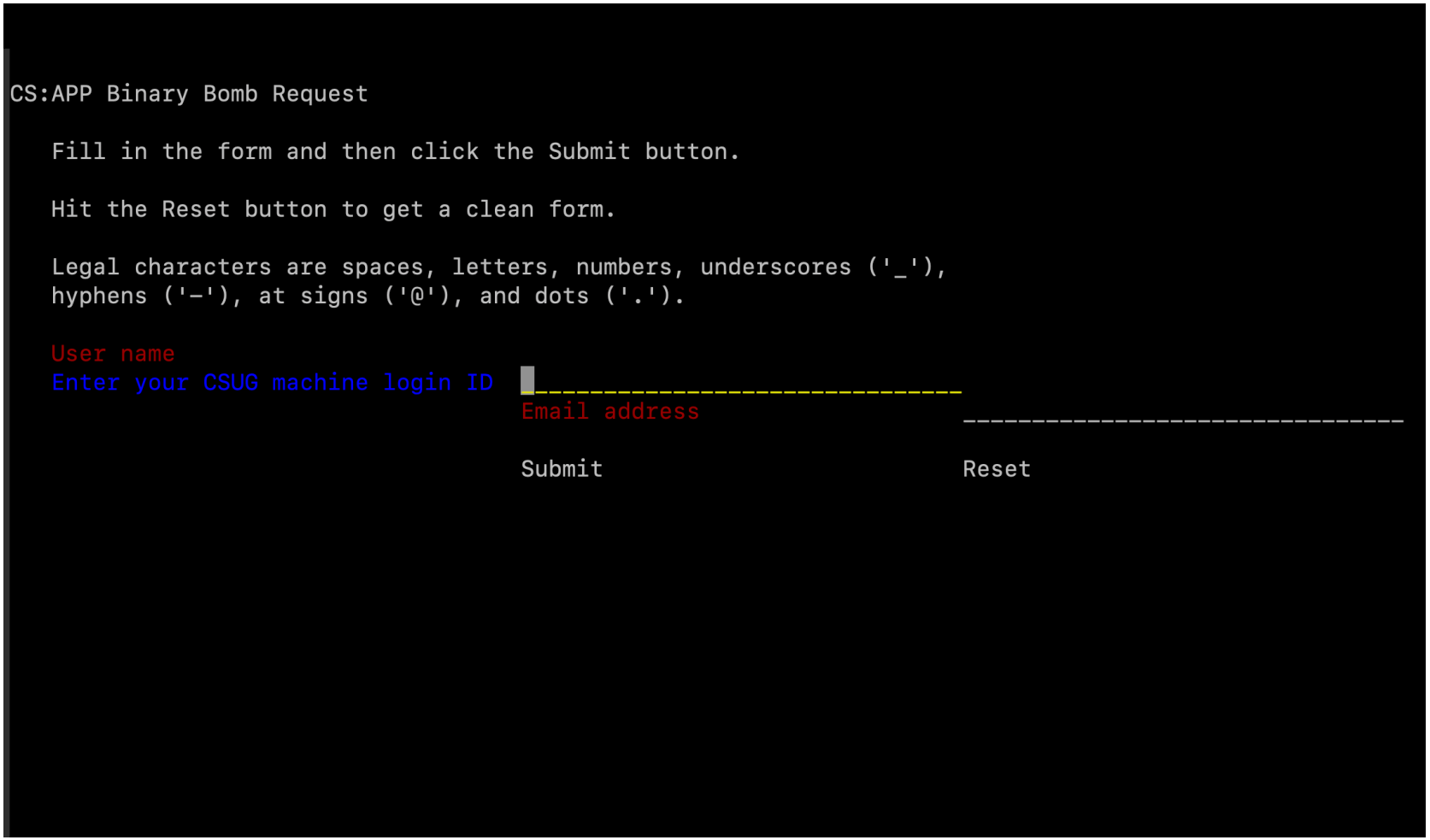
Instructor: Yanan Guo

Department of Computer Science
University of Rochester

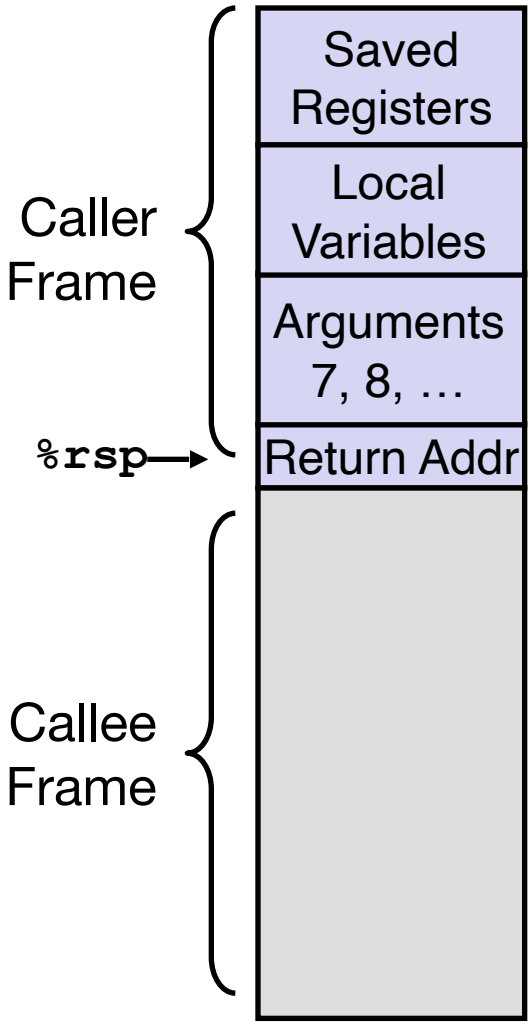
Announcement

- Programming Assignment 2 is out
  - Details: https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment2.html
  - Due on **Oct. 1st**, 11:59 PM
  - You (may still) have 3 slip days
- TA Office hours

Getting a bomb



Stack Frame: Putting It Together

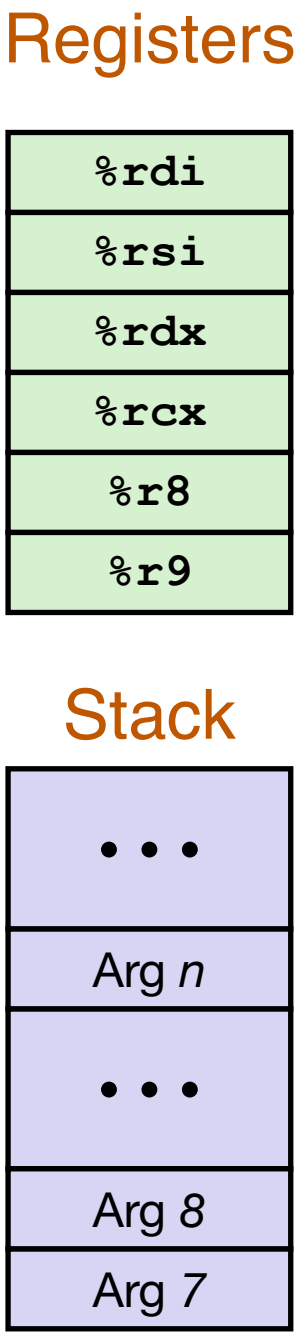


Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount

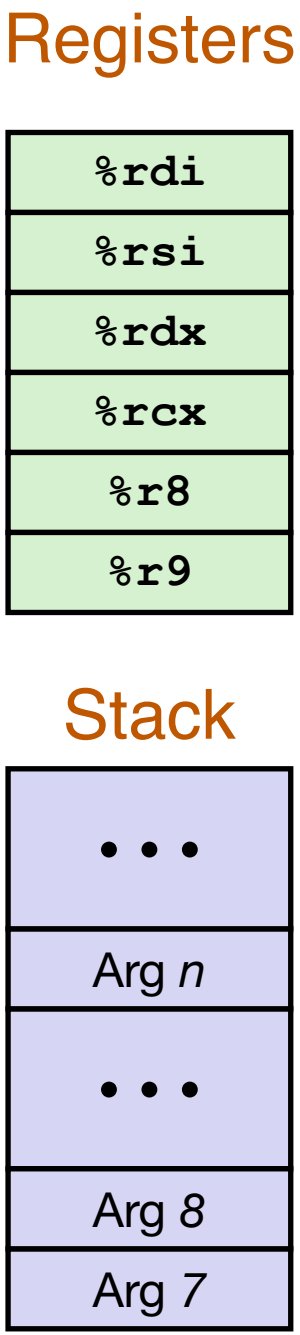
Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount
- x86-64 convention (Part of the *Calling Conventions*):
  - First 6 arguments in registers, in specific order
  - The rest are pushed to stack
  - *Return value is always in* `%rax`

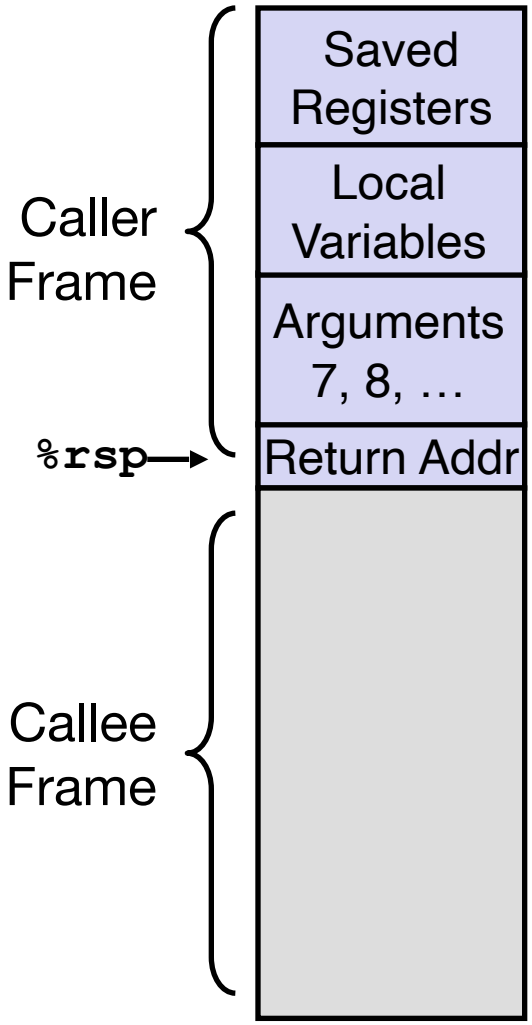


Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount
- x86-64 convention (Part of the *Calling Conventions*):
  - First 6 arguments in registers, in specific order
  - The rest are pushed to stack
  - *Return value is always in* `%rax`
- Just conventions, not laws
  - Not necessary if you write both caller and callee as long as the caller and callee agree
  - But is necessary to interface with others' code



Stack Frame: Putting It Together

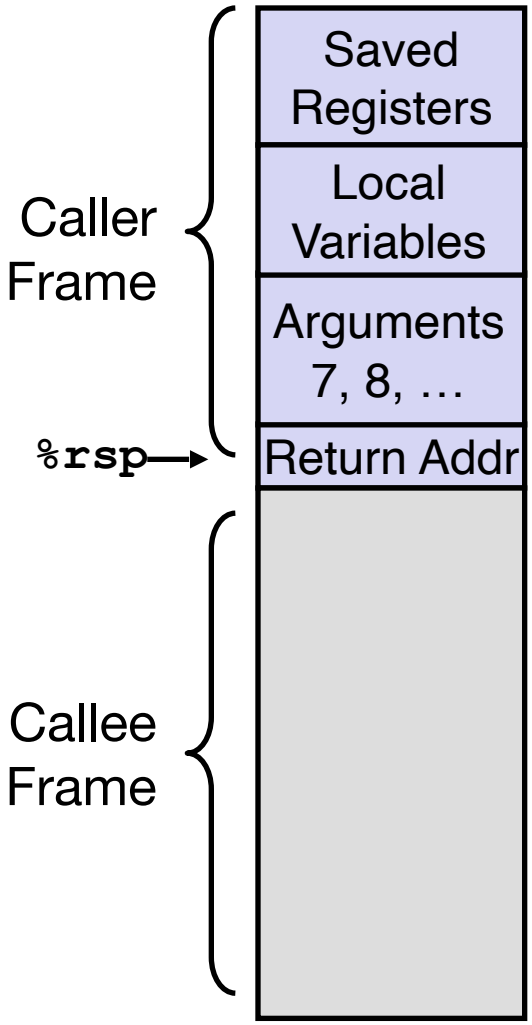


Managing Function Local Variables

- Two ways: registers and memory (stack)
- Registers are faster, but limited. Memory is slower, but large. Smart compilers will optimize the usage.

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Stack Frame: Putting It Together



Register Saving Conventions

Register Saving Conventions

- Any issue with using registers for temporary storage?

Caller

```
yoo:
    …
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    …
    ret
```

Callee

```
who:
    …
    subq $18213, %rdx
    …
    ret
```

Register Saving Conventions

- Any issue with using registers for temporary storage?
  - Contents of register **`%rdx`** overwritten by **`who()`**

Caller

```
yoo:
    …
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    …
    ret
```

Callee

```
who:
    …
    subq $18213, %rdx
    …
    ret
```

Register Saving Conventions

- Any issue with using registers for temporary storage?
  - Contents of register **`%rdx`** overwritten by **`who()`**
  - This could be trouble ➙ Need some coordination

Caller

```
yoo:
    …
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    …
    ret
```

Callee

```
who:
    …
    subq $18213, %rdx
    …
    ret
```

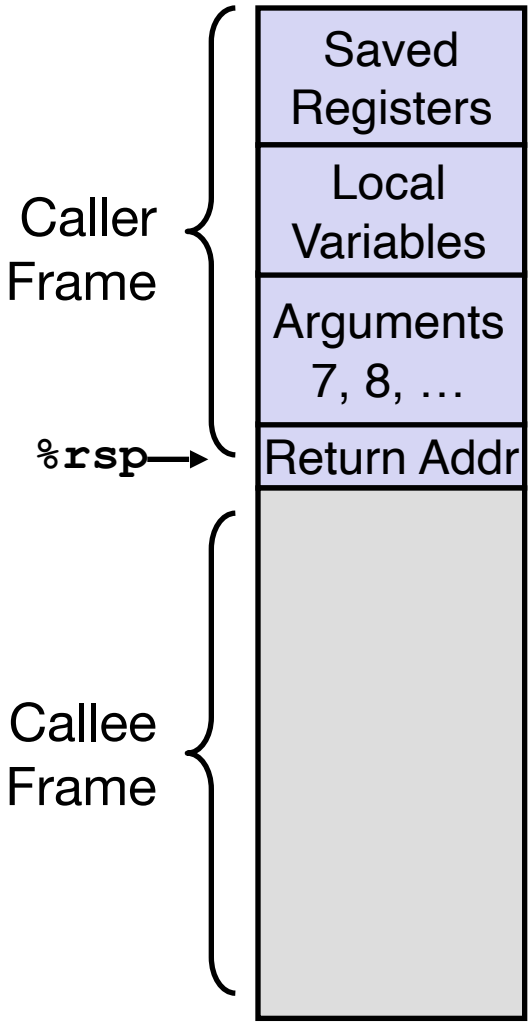
Register Saving Conventions

- Conventions used in x86-64 (*Part of the Calling Conventions*)
  - Some registers are saved by caller, some are by callee.
  - Caller saved: `%rdi,%rsi,%rdx,%rcx,%r8,%r9,%r10,%r11`
  - Callee saved: `%rbx,%rbp,%r12,%r13,%14,%r15`
  - `%rax` holds return value, so implicitly caller saved
  - `%rsp` is the stack pointer, so implicitly callee saved

Register Saving Conventions

- Common conventions
  - "*Caller Saved*"
    - Caller saves temporary values in its frame (on the stack) before the call
    - Callee is then free to modify their values
  - "*Callee Saved*"
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller
    - Caller can safely assume that register values won't change after the function call

Stack Frame: Putting It Together



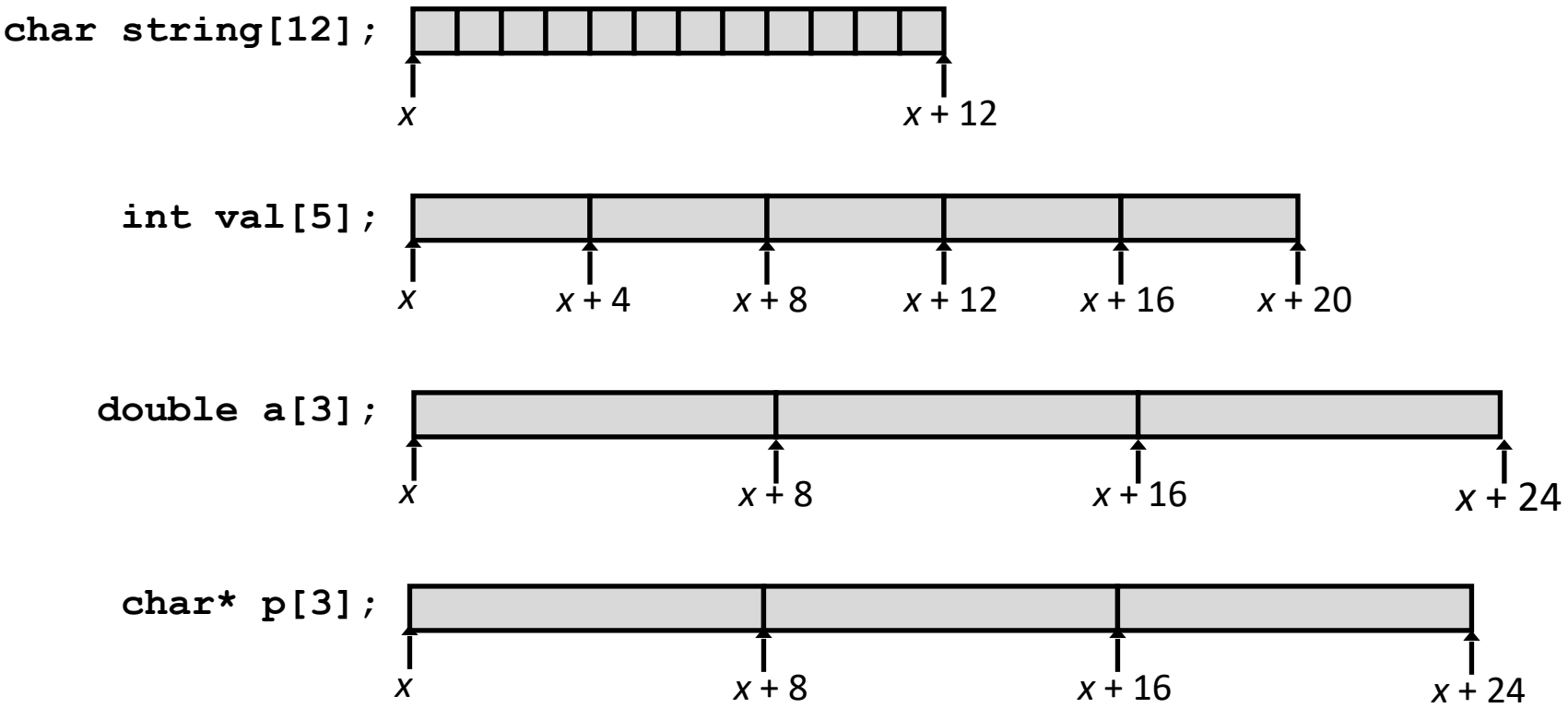
Today: Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Allocation
  - Access
  - Alignment
- Buffer Overflow

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



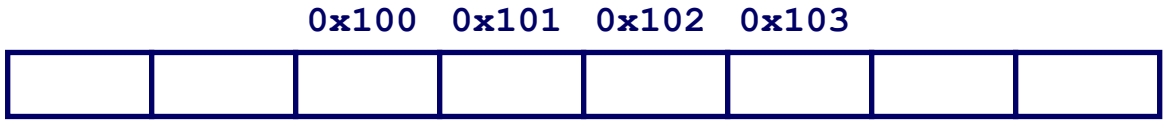
Byte Ordering

Byte Ordering

- How are the bytes of a multi-byte variable ordered in memory?

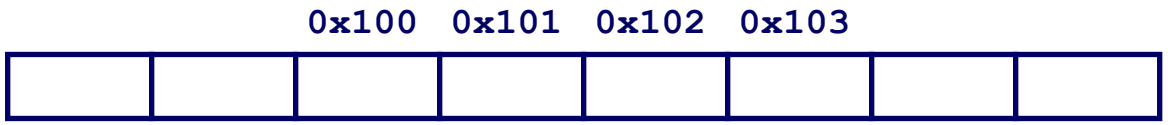
Byte Ordering

- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



Byte Ordering

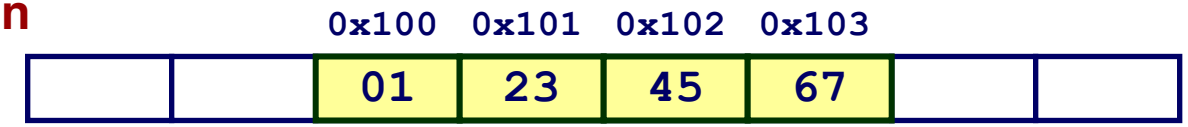
- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)



Byte Ordering

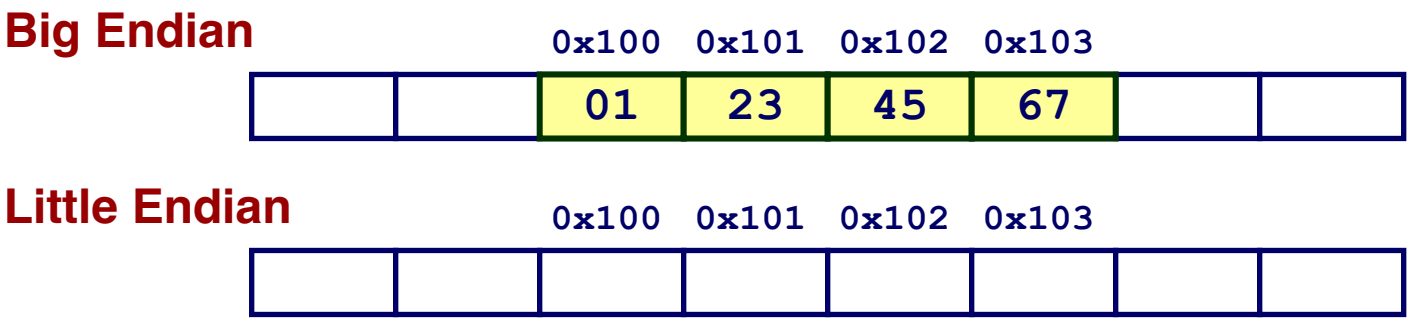
- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)

Big Endian



Byte Ordering

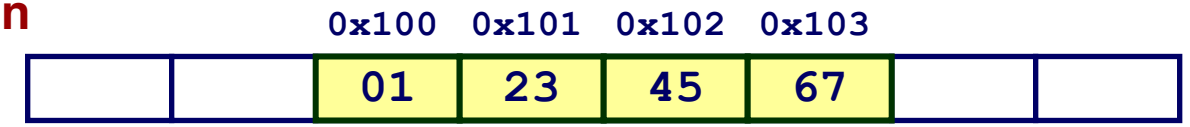
- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)



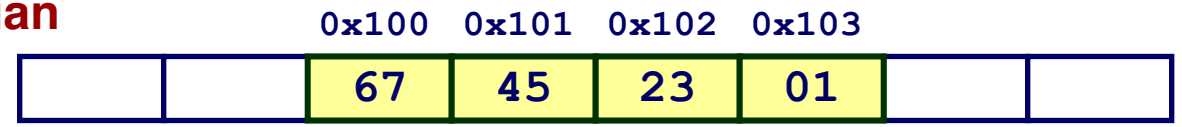
Byte Ordering

- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)

Big Endian



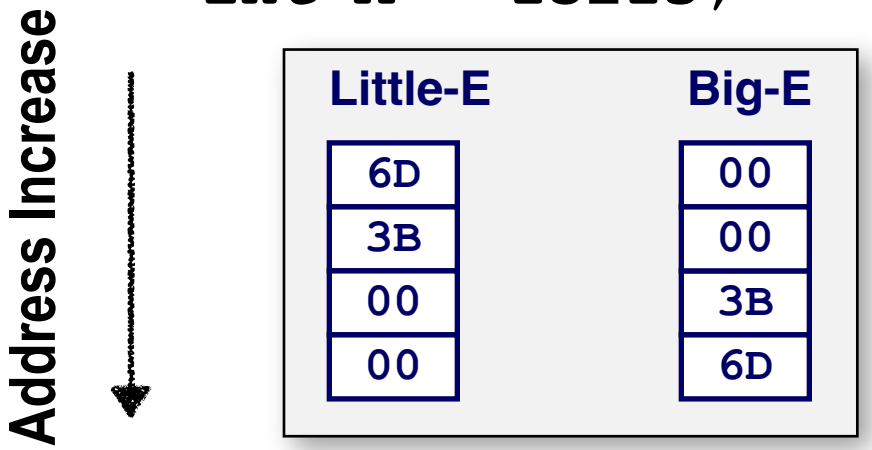
Little Endian



Representing Integers

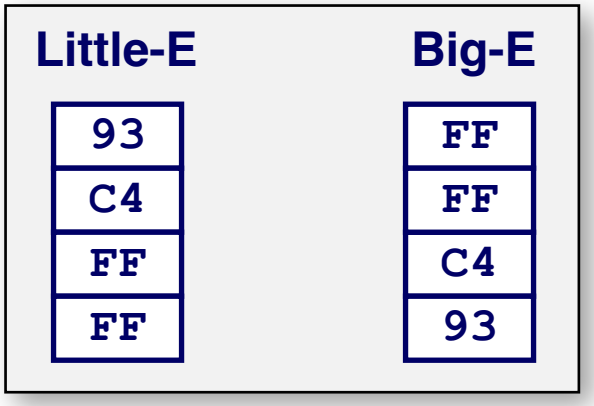
Hex: `00003B6D`

```
int A = 15213;
```



Hex: `FFFFC493`

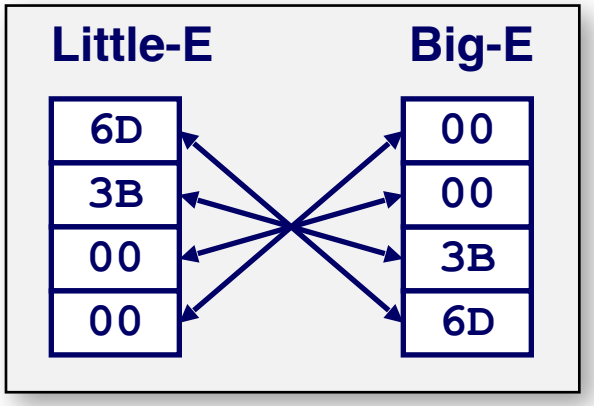
```
int B = -15213;
```



Representing Integers

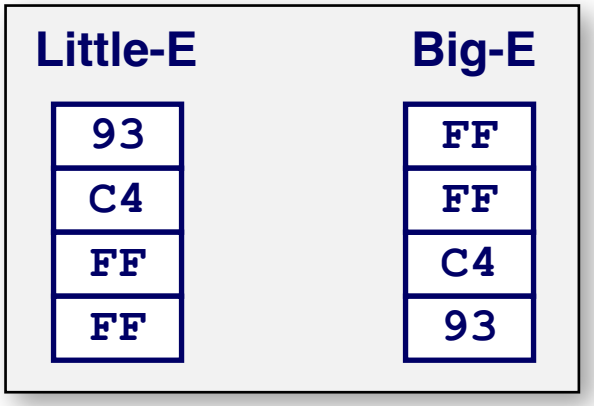
Hex: `00003B6D`

```
int A = 15213;
```



Hex: `FFFFC493`

```
int B = -15213;
```

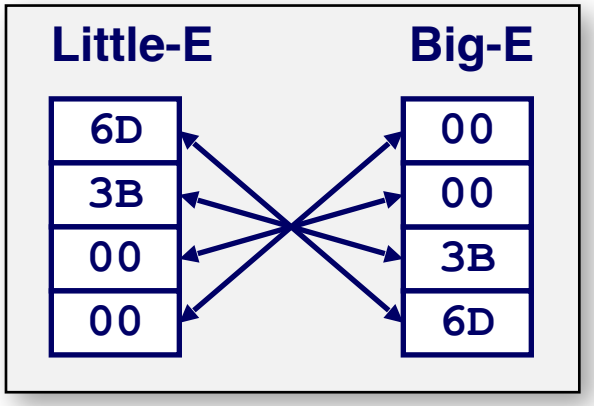


Address Increase

Representing Integers

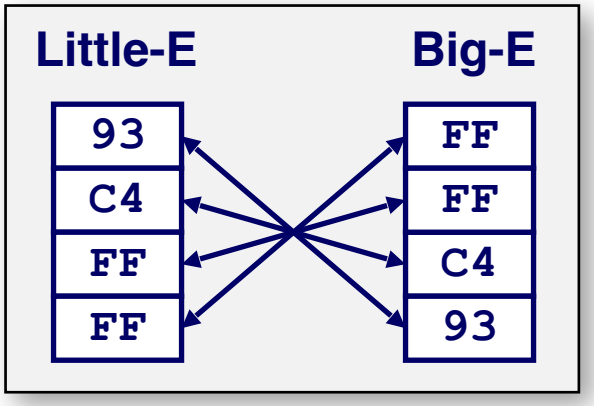
Hex: `00003B6D`

```
int A = 15213;
```



Hex: `FFFFC493`

```
int B = -15213;
```

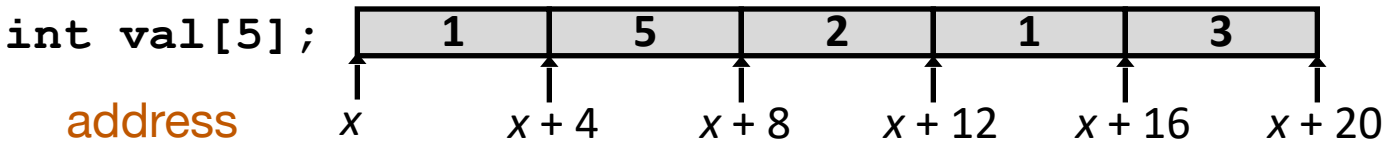


Address Increase

Array Access: Basic Principle

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

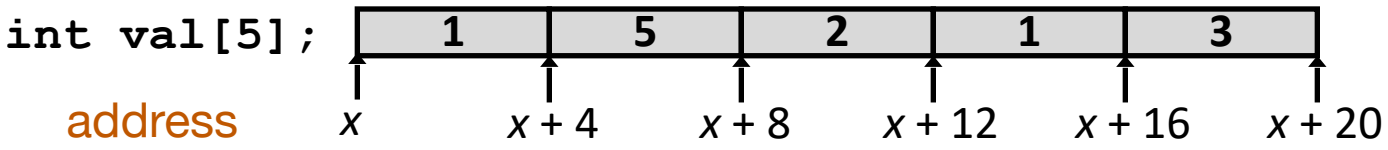
- Array of data type *T* and length *L*
- Identifier **A** can be used as a pointer to array element 0: Type *T**



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**



Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`val + i`	`int *`	*x* + 4 *i*
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5

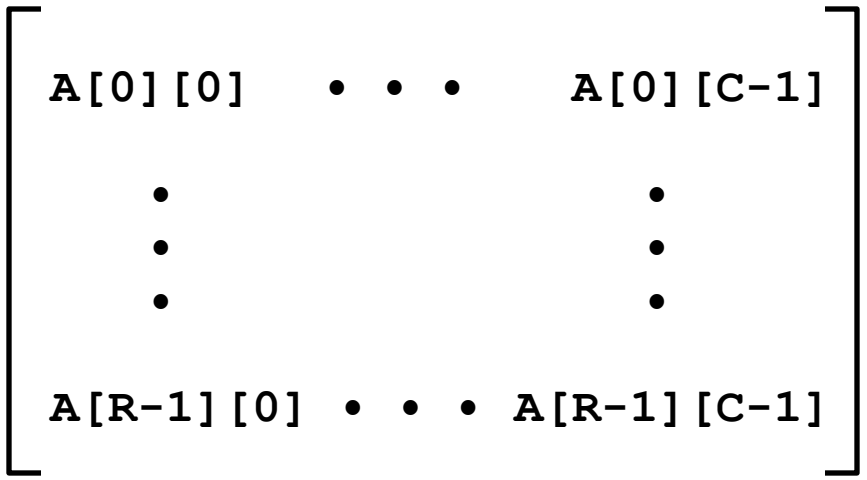
Multidimensional (Nested) Arrays

Multidimensional (Nested) Arrays

- Declaration

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

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



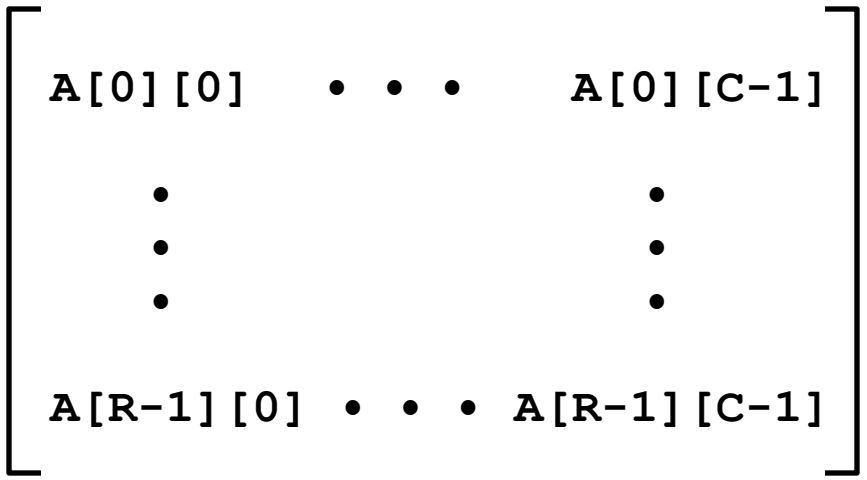
Multidimensional (Nested) Arrays

- Declaration

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

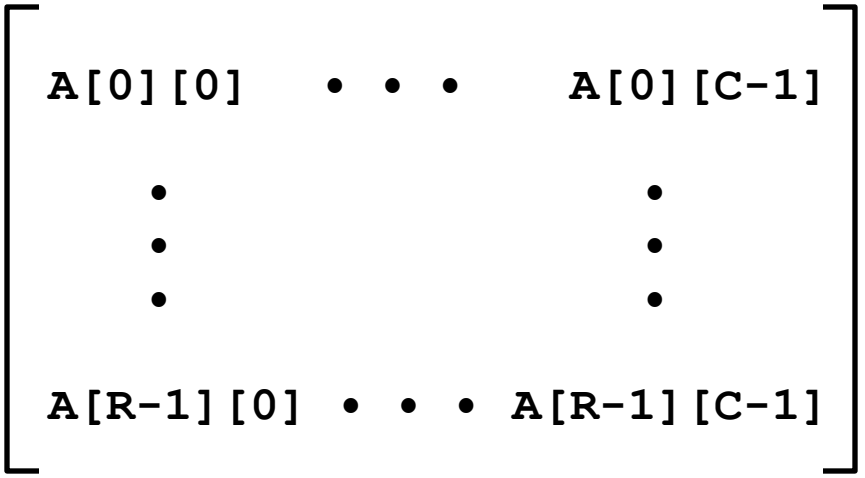
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

- Array Size
  - *R* * *C* * *K* bytes



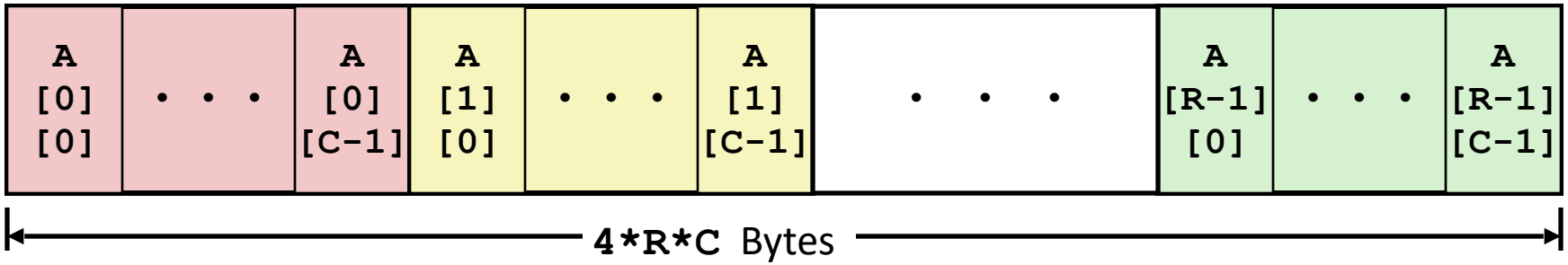
Multidimensional (Nested) Arrays

- 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
  - Row-Major Ordering in most languages, including C

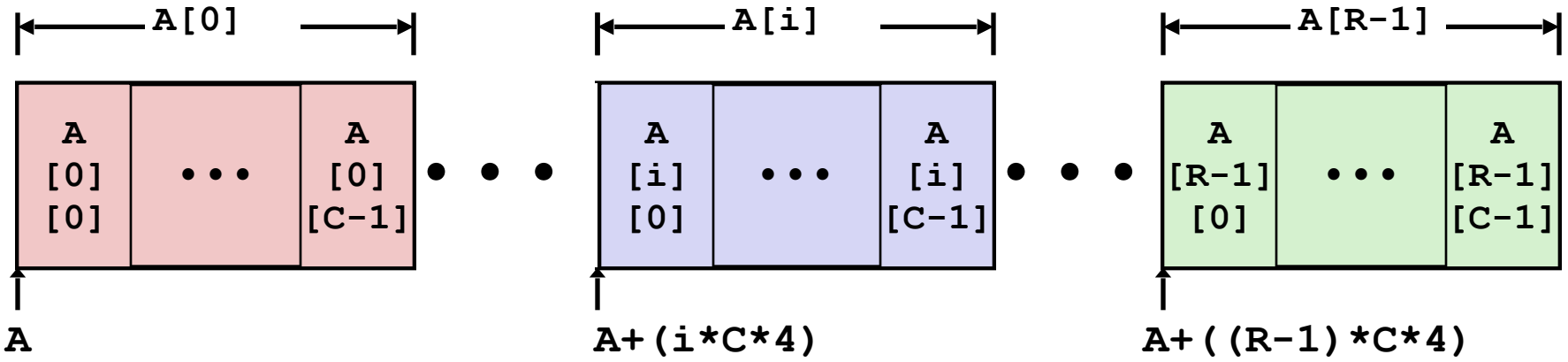
int A[R][C];



Nested Array Row Access

- T **`A[R][C];`**
  - **`A[i]`** is array of *C* elements
  - Each element of type T requires K bytes
  - Starting address A + i * (C * K)

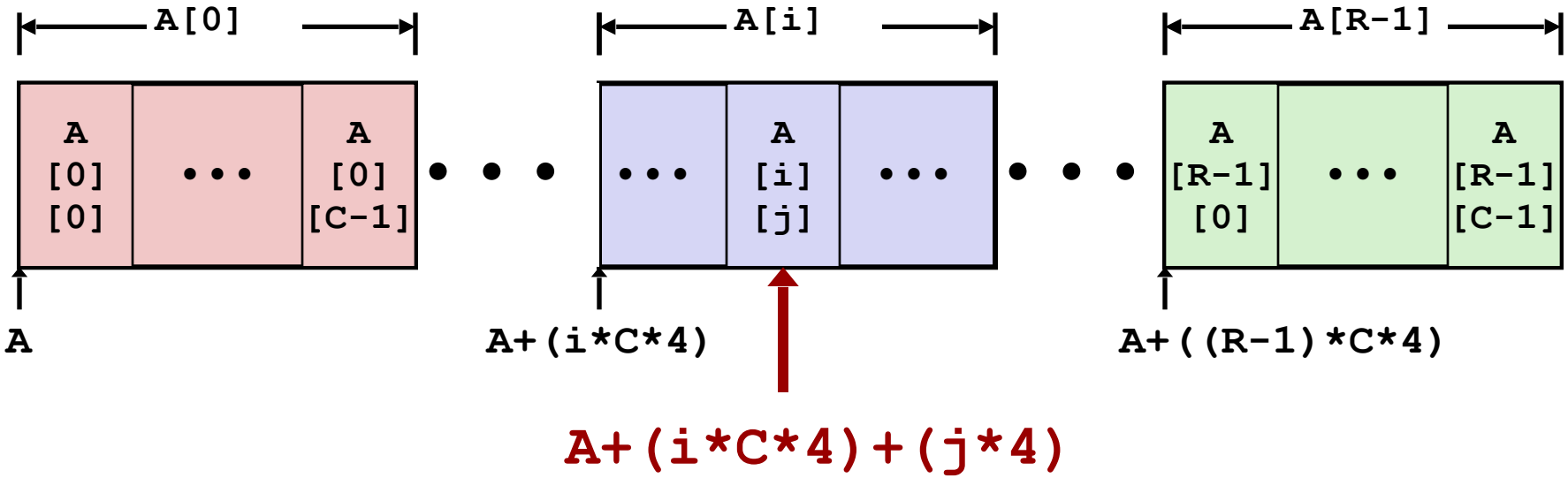
```
int A[R][C];
```



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];
```

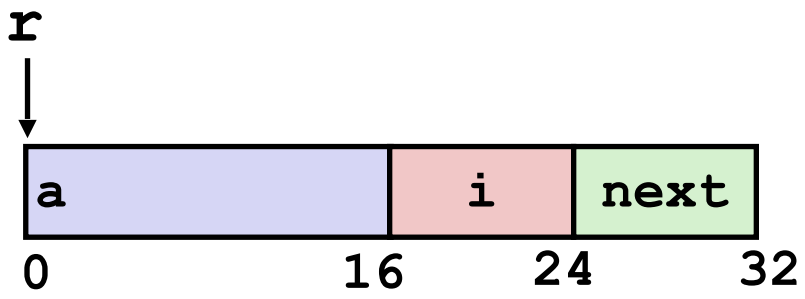


Today: Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Allocation
  - Access
  - Alignment
- Buffer Overflow

Structures

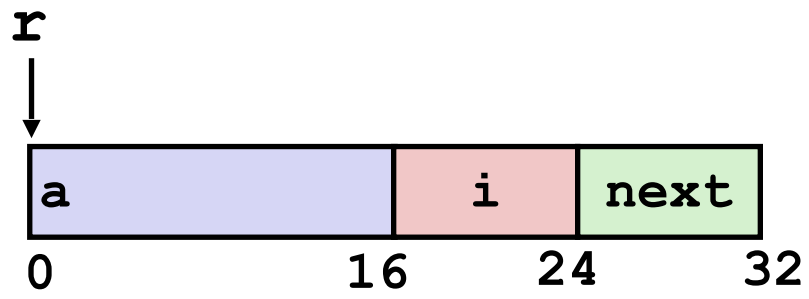
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within struct by names
  - Members may be of different types

Access Struct Members

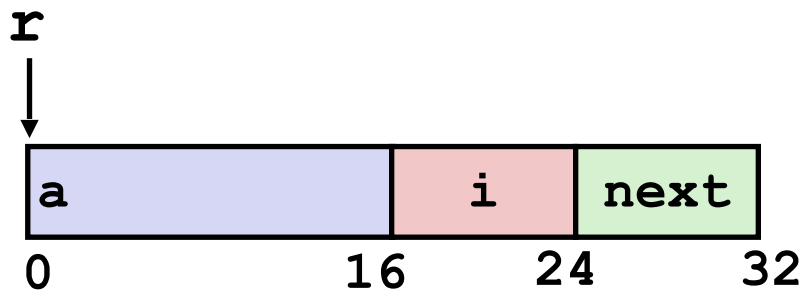
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`

Access Struct Members

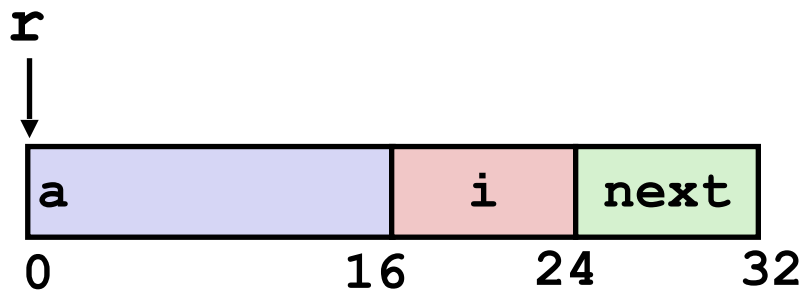
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
- Suppose we have a pointer `r` pointing to `struct res`. How to access `res`'s member using `r`?

Access Struct Members

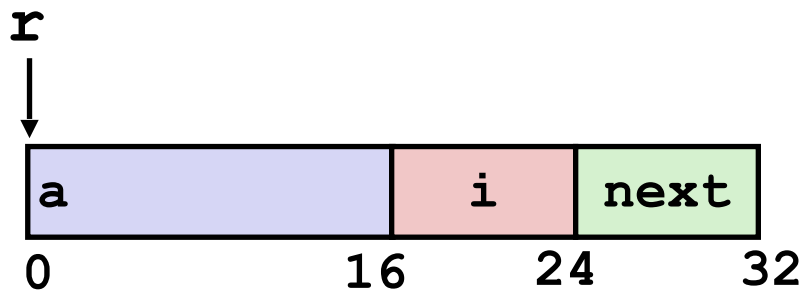
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
- Suppose we have a pointer `r` pointing to `struct res`. How to access `res`'s member using `r`?
  - Using ***** and **.** operators: `(*r).i = val;`

Access Struct Members

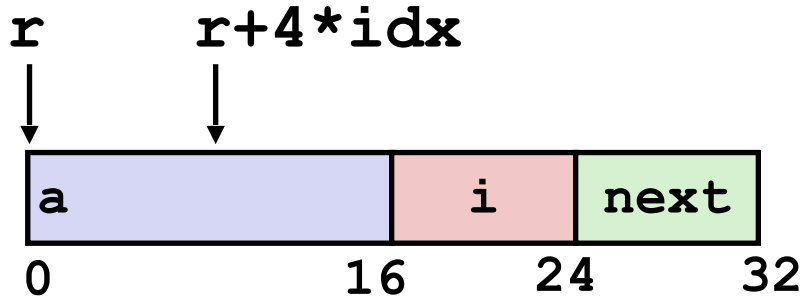
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
- Suppose we have a pointer `r` pointing to `struct res`. How to access `res`'s member using `r`?
  - Using ***** and **.** operators: `(*r).i = val;`
  - Or simply, the -> operator for short: `r->i = val;`

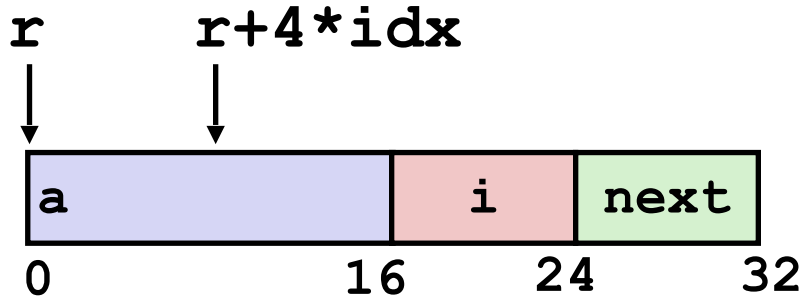
Generating Pointer to Structure Member

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



Generating Pointer to Structure Member

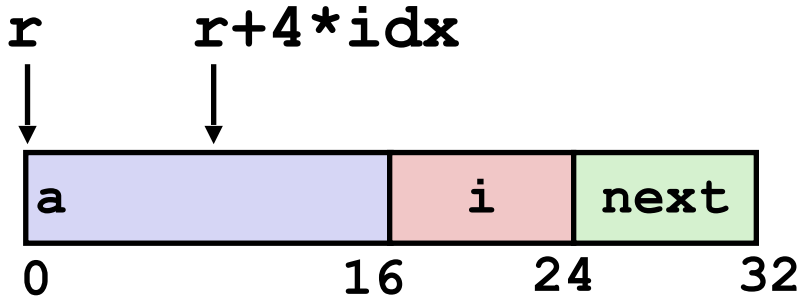
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```




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

Generating Pointer to Structure Member

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

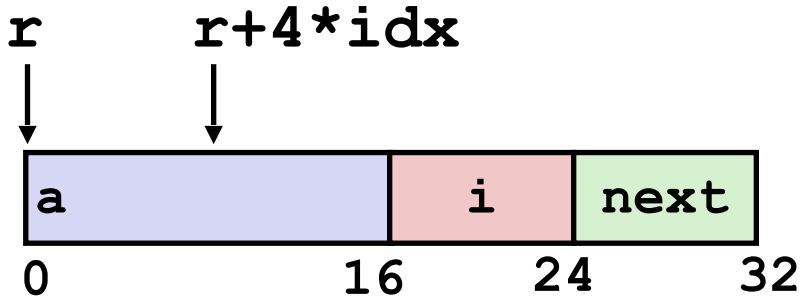



```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```

```
&((*r).a[idx])
```

Generating Pointer to Structure Member

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




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

```
&((*r).a[idx])
```

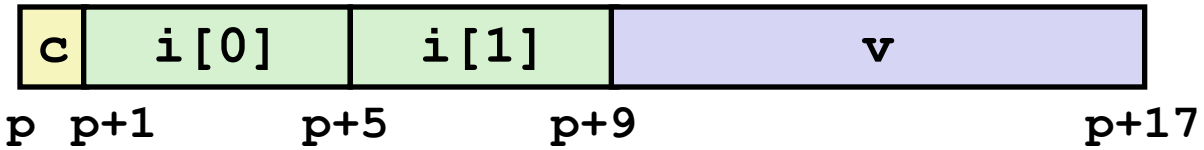
```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Alignment

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

Alignment

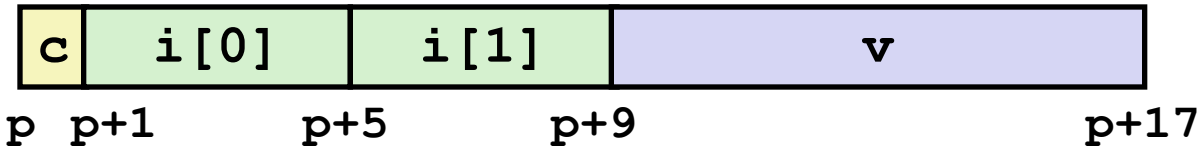
- Unaligned Data




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

Alignment

- Unaligned Data

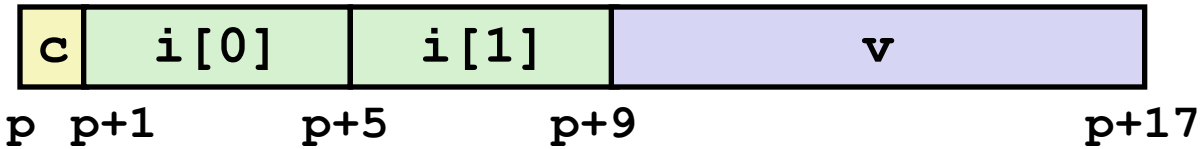



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***

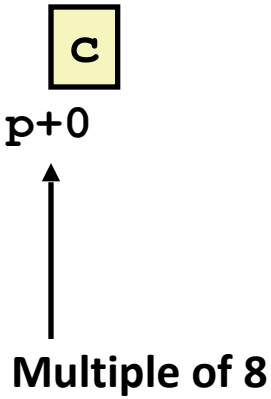
Alignment

- Unaligned Data



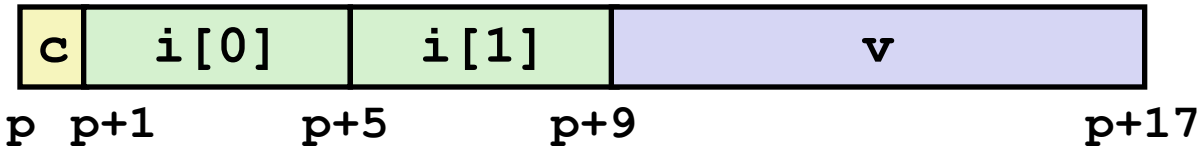

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



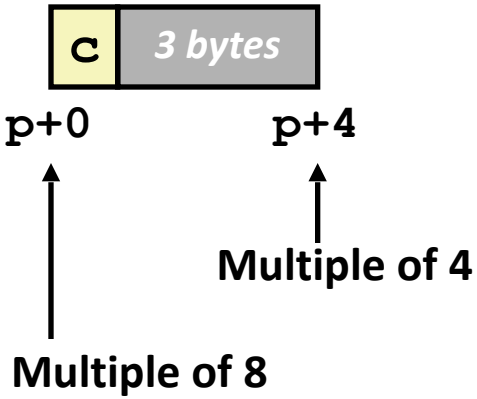
Alignment

- Unaligned Data



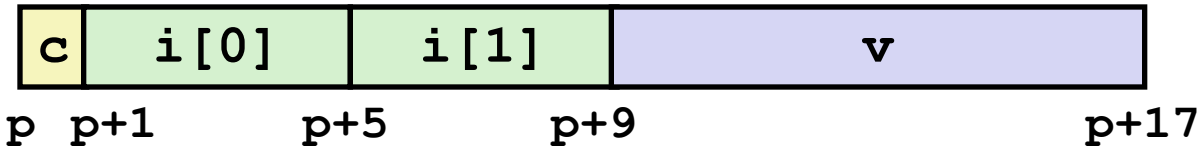

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



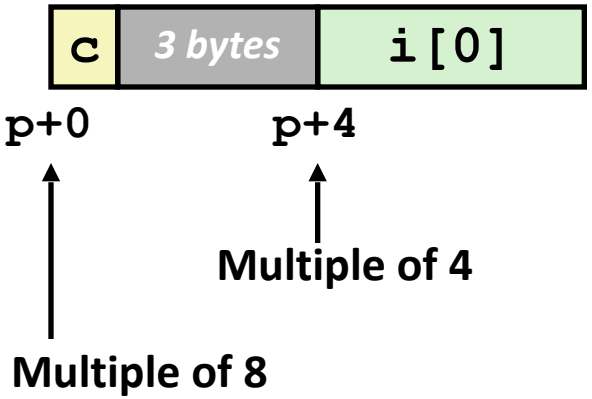
Alignment

- Unaligned Data



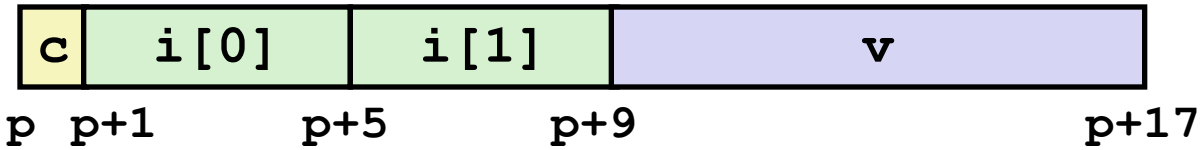

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



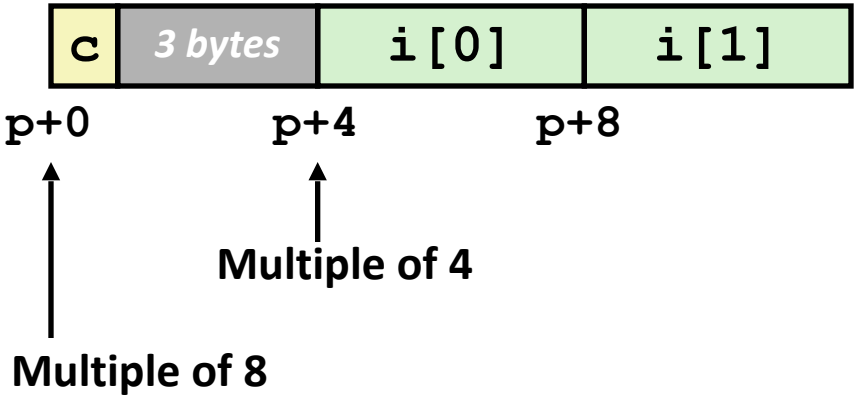
Alignment

- Unaligned Data



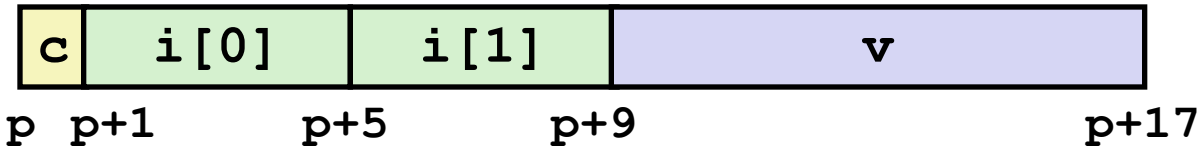

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



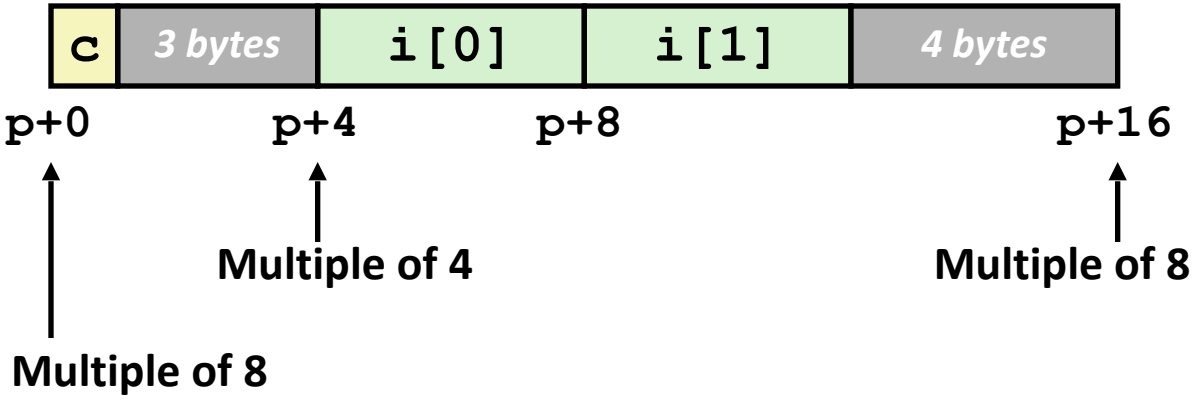
Alignment

- Unaligned Data



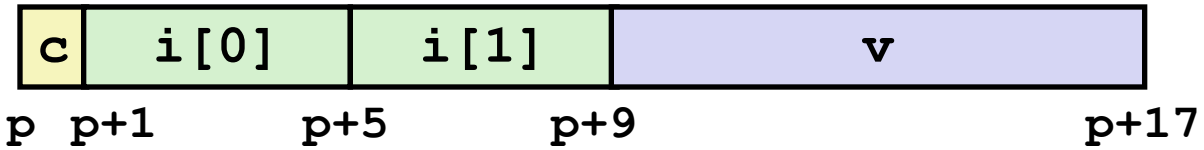

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



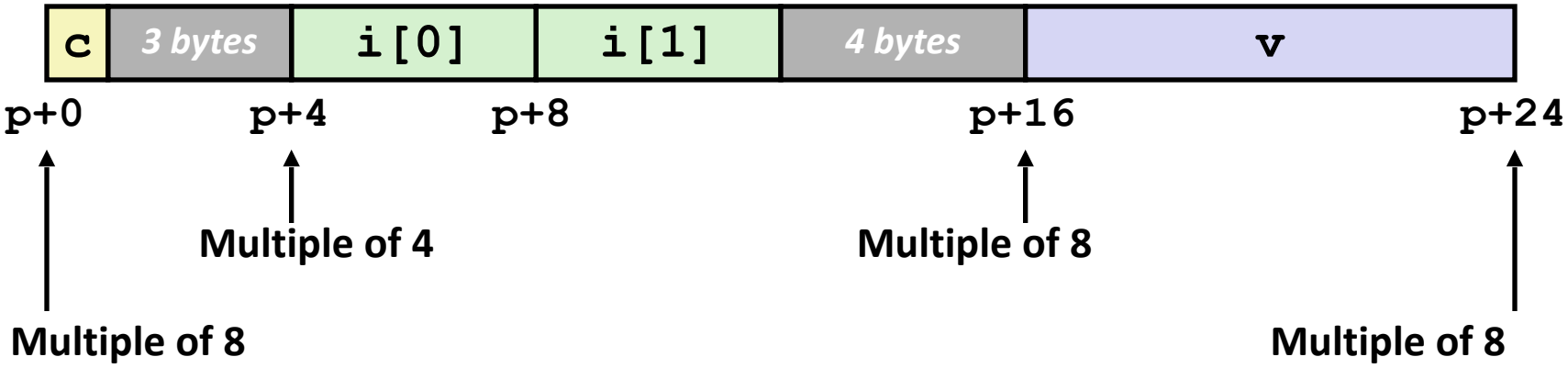
Alignment

- Unaligned Data




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

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



Alignment Principles

- Aligned Data
  - If the data type requires K bytes, address must be multiple of K
- Required on some machines; advised on x86-64
- Motivation for Aligning Data: Performance
  - Inefficient to load or store data that is unaligned
  - Some machines don't event support unaligned memory access
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields
  - `sizeof()` returns the actual size of structs (i.e., including padding)

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

Satisfying Alignment with Structures

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement

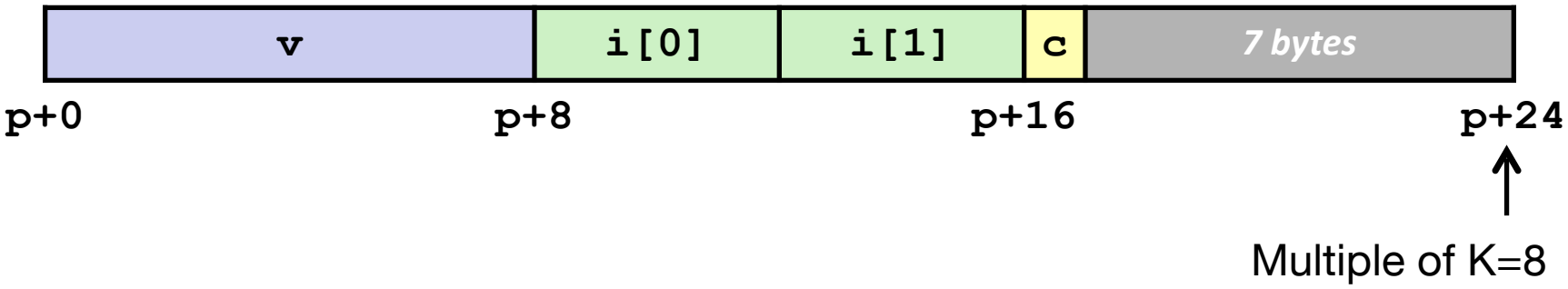
Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Structure length must be multiples of **K**, where:
    - **K** = Largest alignment of any element
  - **WHY?!**

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Structure length must be multiples of **K**, where:
    - **K** = Largest alignment of any element
  - **WHY?!**

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

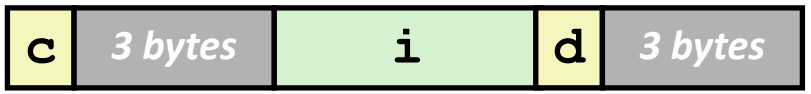


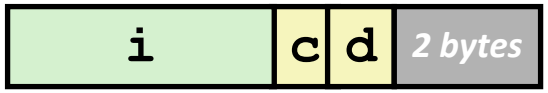
Saving Space

- Put large data types first in a Struct
- This is not something that a C compiler would always do
  - But knowing low-level details empower a C programmer to write more efficient code

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

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

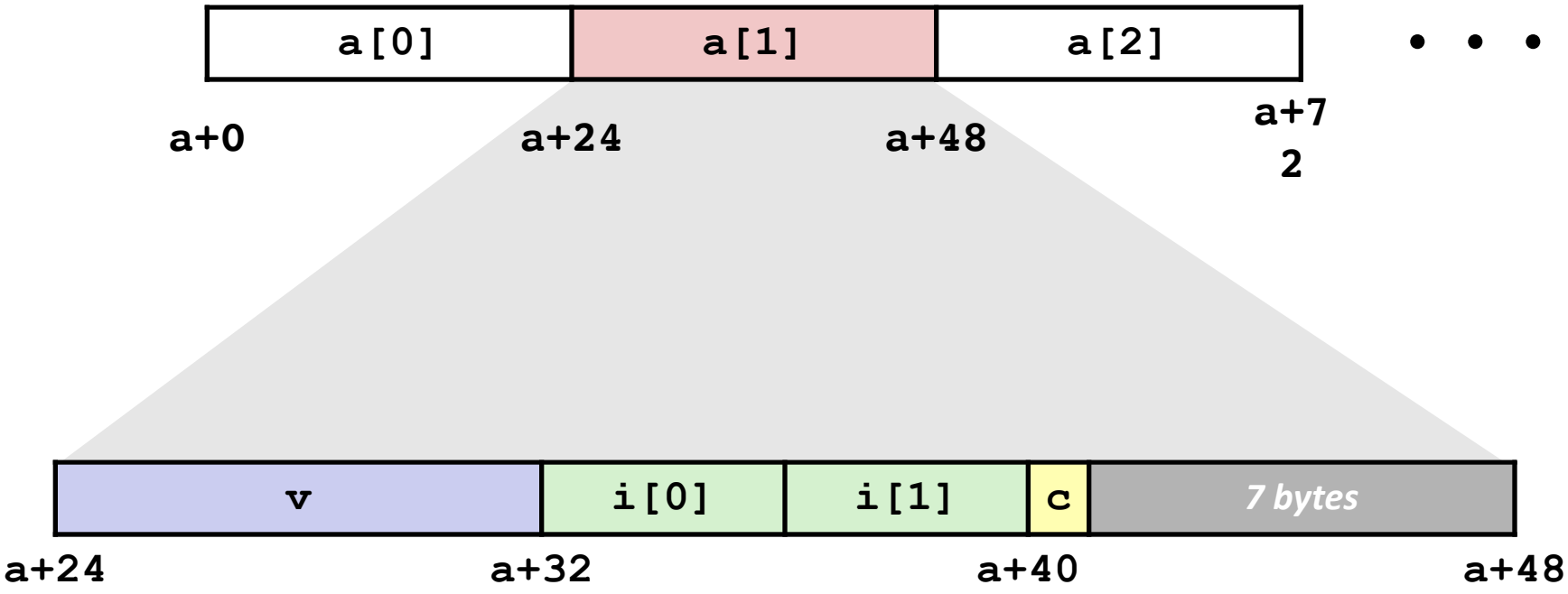




Arrays of Structures

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

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



Return Struct Values

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in **`%rax`**??

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in **`%rax`**??
- We don't have to follow that convention…

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in **`%rax`**??
- We don't have to follow that convention…
- If there are only a few members in a struct, we could return through a few registers.

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in **`%rax`**??
- We don't have to follow that convention…
- If there are only a few members in a struct, we could return through a few registers.
- If there are lots of members, we could return through memory, i.e., requires memory copy.

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in **`%rax`**??
- We don't have to follow that convention…
- If there are only a few members in a struct, we could return through a few registers.
- If there are lots of members, we could return through memory, i.e., requires memory copy.
- But either way, there needs to be some sort convention for returning struct.

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

Return Struct Values

```
struct S{
  int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
  struct S test = foo(3, 4);
  fprintf(stdout, "%d, %d\n",
test.a, test.b);
  // you will get "3, 4" from
the terminal
}
```

- The entire calling convention is part of what's called Application Binary Interface (ABI), which specifies how **two binaries** should interact.
- ABI includes: ISA, data type size, calling convention, etc.
- API defines the interface as the **source code** (e.g., C) level.
- The OS and compiler have to agree on the ABI.
- Linux x86-64 ABI specifies that returning a struct with two scalar (e.g. pointers, or long) values is done via **`%rax`** & **`%rdx`**

Today: Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Allocation
  - Access
  - Alignment
- Buffer Overflow

String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
/* 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;
}
```

Vulnerable Buffer Code

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

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

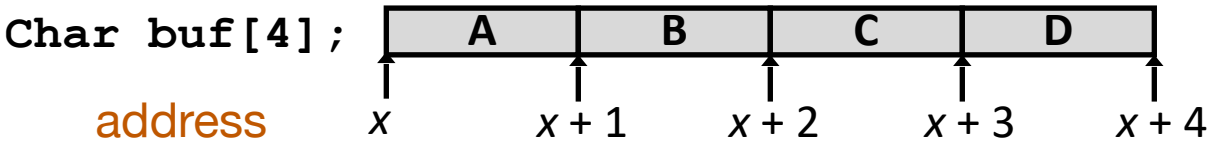
String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
/* 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;
}
```

String Library Code

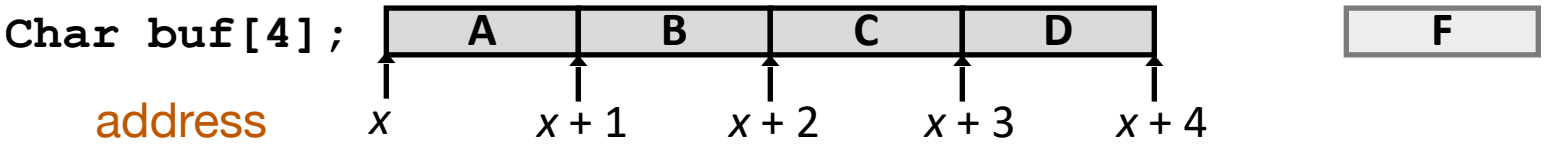
- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read




```
/* 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;
}
```

String Library Code

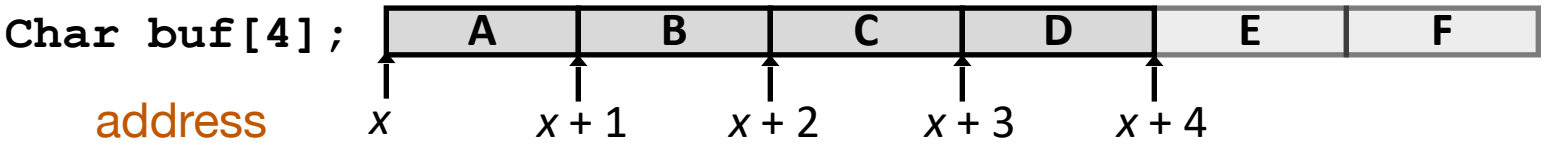
- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read




```
/* 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;
}
```

String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read




```
/* 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;
}
```

Vulnerable Buffer Code

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

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

Vulnerable Buffer Code

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

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

```
unix>./bufdemo-nsp
Type a string:0123
0123
```

Vulnerable Buffer Code

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

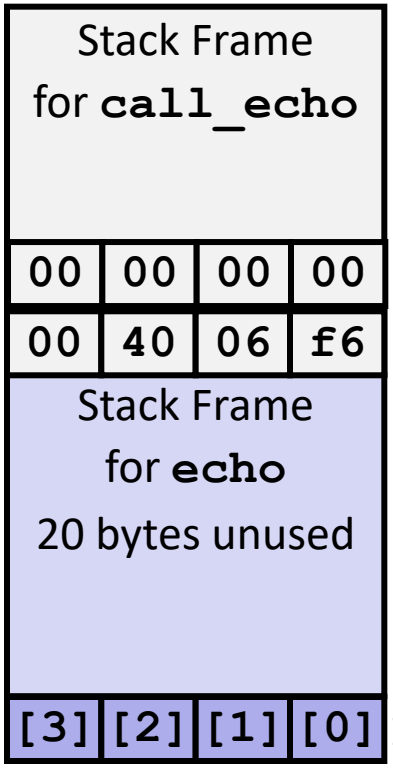
```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123
0123
```

```
unix>./bufdemo-nsp
Type a string:01234
Segmentation Fault
```

Buffer Overflow Stack Example

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

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

call_echo:

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

Buffer Overflow Stack Example #1

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

```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

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

call_echo:

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

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

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

```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

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

call_echo:

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

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

Overflowed buffer, and corrupt return address

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
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

```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

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

`call_echo:`

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

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123456789012345678901230
```

Overflowed buffer, corrupt return address, but program appears to still work!

Buffer Overflow Stack Example #4

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
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`

`register_tm_clones:`

```
  . . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

"Returns" to unrelated code

Could be code controlled by attackers!