CSC 252: Computer Organization Spring 2022: Lecture 9

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 2 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment2.html
  - Due on **Feb 15**, 11:59 PM
  - You (may still) have 3 slip days

SUN	MON	TUE	WED	THU	FRI	SAT
30	31	1	2	3	4	5
6	7	8	9	10 **Today**	11	12
13	14	15 **Due**	16	17	18	19

Announcement

- Finally is on May 4, 19:15 — 22:15.

- Programming assignment 2 is in x86 assembly language.

- Read the instructions before getting started!!!

  - You get 1/4 point off for every wrong answer

  - Maxed out at 10

- TAs are best positioned to answer your questions about programming assignments!!!

- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

- Problem set on arithmetics: https://www.cs.rochester.edu/courses/252/spring2022/handouts.html.

  - Not to be turned in.

Register Saving Conventions

Register Saving Conventions

- Any issue with using registers for temporary storage?

<div style="color:#c0520a">Caller</div>

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

<div style="color:#c0520a">Callee</div>

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

Register Saving Conventions

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

Caller

Callee

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

```
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

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

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

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
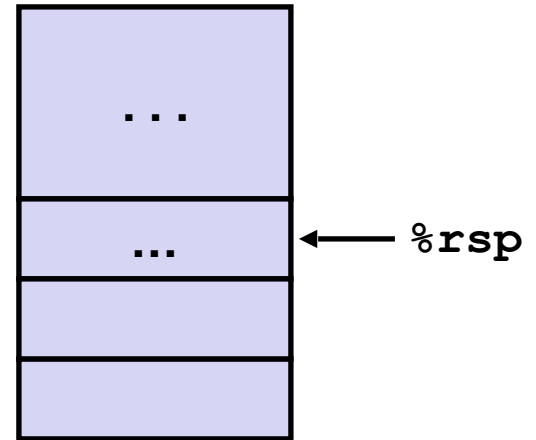   - Caller can safely assume that register values won't change after the function call

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

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```
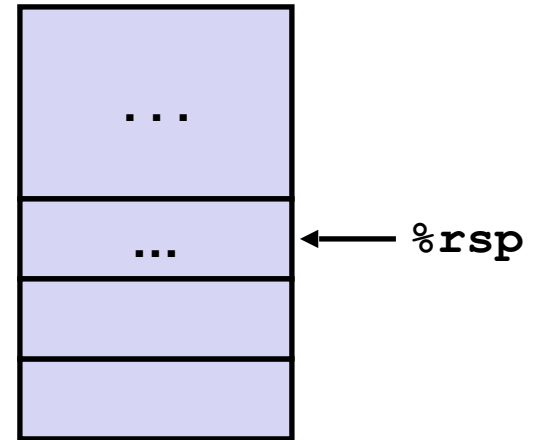
Stack


```
...
```

```
...
```

%rsp

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Stack


```
call_incr2:
  pushq    %rbx
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```
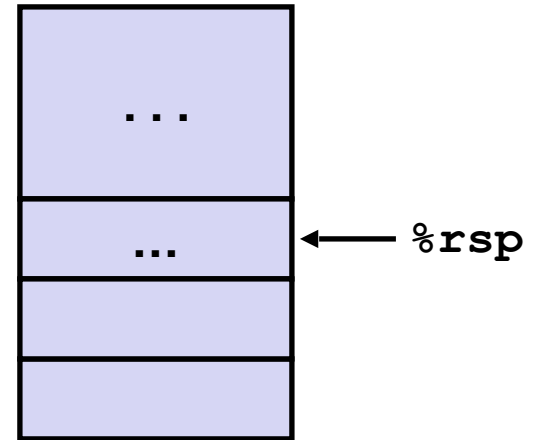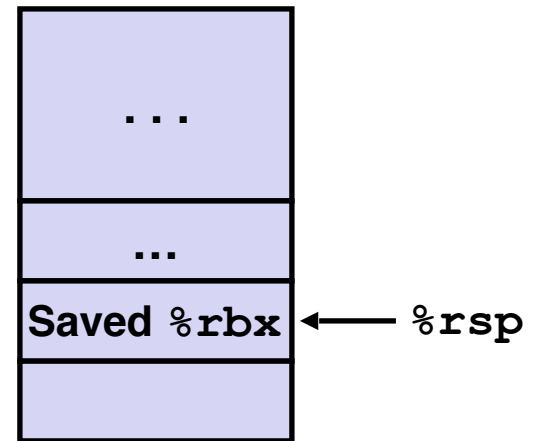
%rsp

7

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```


...

...   ← **%rsp**

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```

Stack

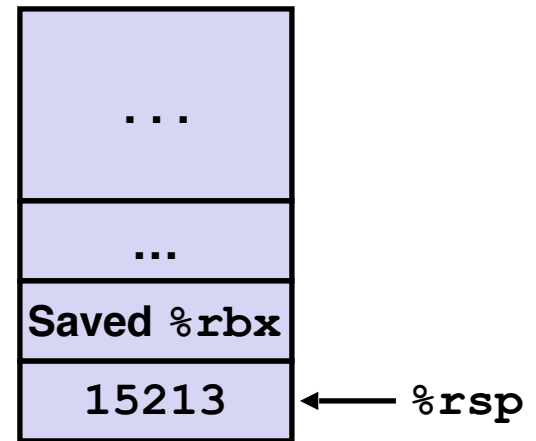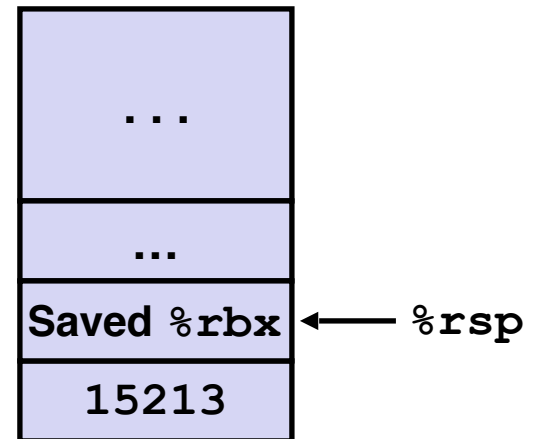| |
|---|
| ... |
| ... |
| **Saved %rbx** ← **%rsp** |
| |

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```

...
...
Saved %rbx
15213

← **%rsp**

7

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Stack

```
call_incr2:
  pushq    %rbx          ⬅
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx          ⬅
  ret
```
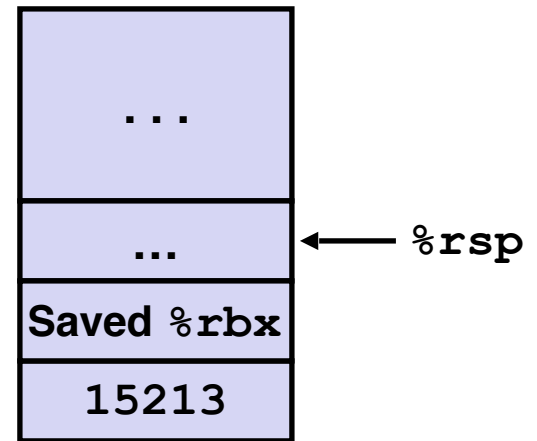
7

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx          ⬅
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx          ⬅
  ret
```

...
...	← **%rsp**
Saved %rbx
15213

7

Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Stack


```
call_incr2:
  pushq    %rbx
  pushq    $15213
  movq     %rdi, %rbx
  movl     $3000, %esi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```
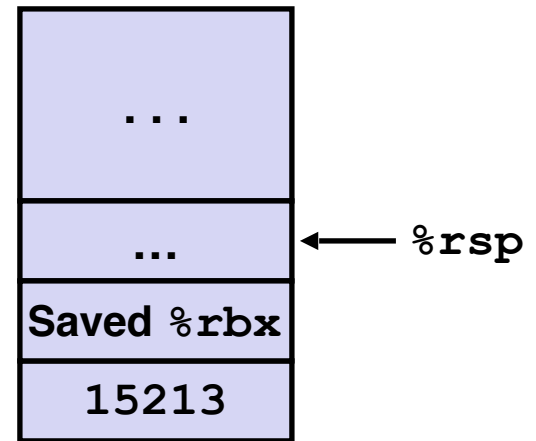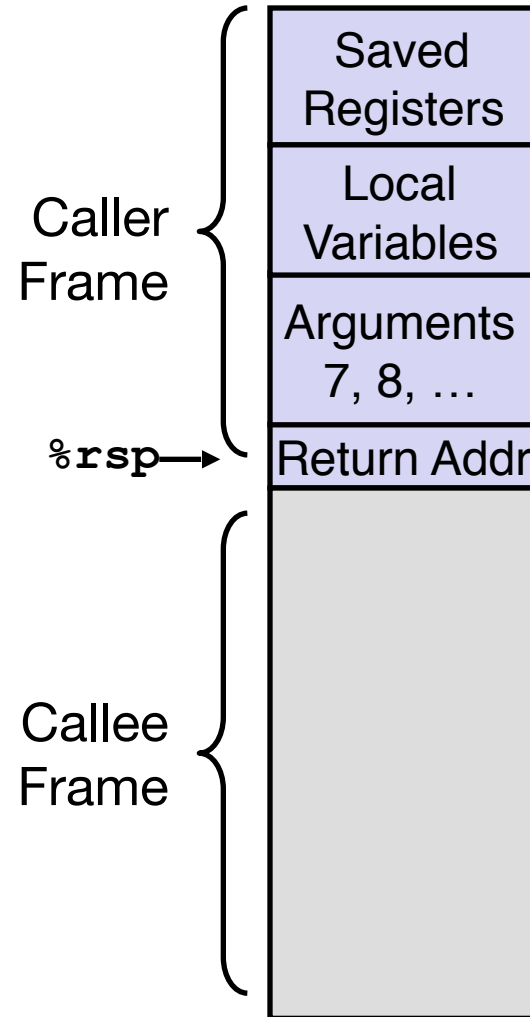
- `call_incr2` needs to save `%rbx` (callee-saved) because it will modify its value
- It can safely use `%rbx` after `call incr` because `incr` will have to save `%rbx` if it needs to use it (again, `%rbx` is callee saved)

Stack Frame: Putting It Together

Saved Registers
Local Variables
Arguments 7, 8, …
Return Addr

Caller Frame: Saved Registers, Local Variables, Arguments 7, 8, …
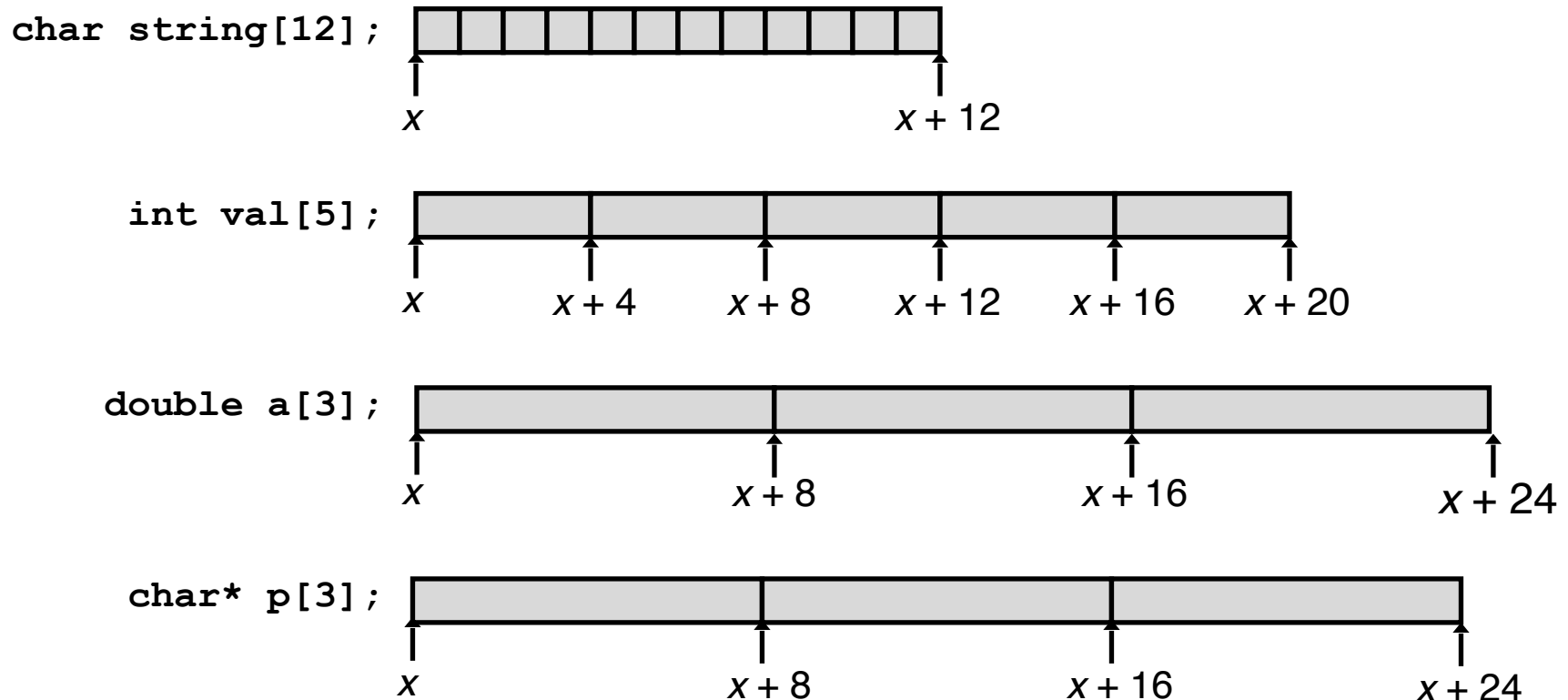
%rsp → Return Addr

Callee Frame

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

char string[12];

x $x+12$

int val[5];

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

double a[3];

x $x+8$ $x+16$ $x+24$

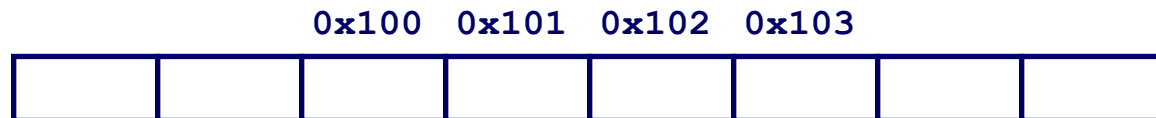**char* p[3];**

x $x+8$ $x+16$ $x+24$

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

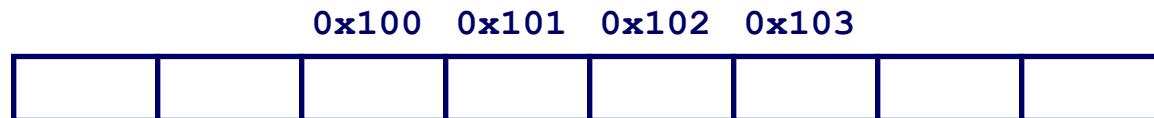```
        0x100 0x101 0x102 0x103
```
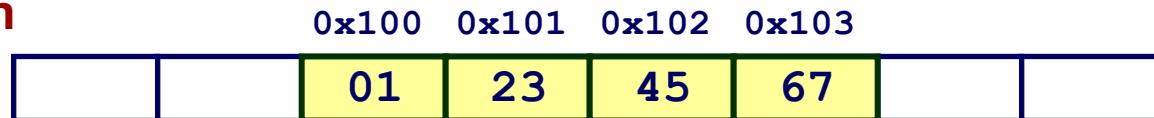
# Byte Ordering

- 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)

```
          0x100  0x101  0x102  0x103
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│      │      │      │      │      │      │      │      │
└──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘
```

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)
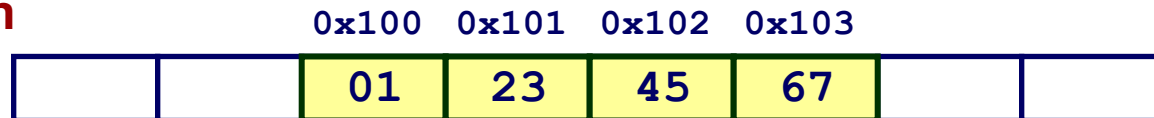
Big Endian

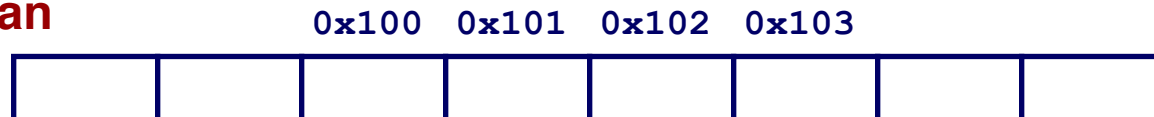|		0x100	0x101	0x102	0x103		
		01	23	45	67		

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

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		

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
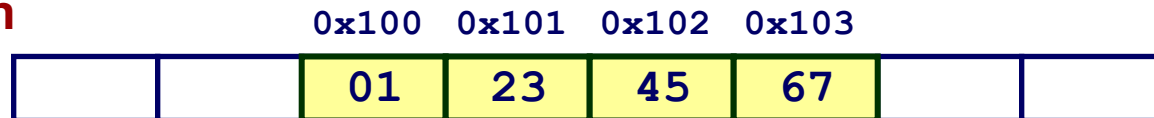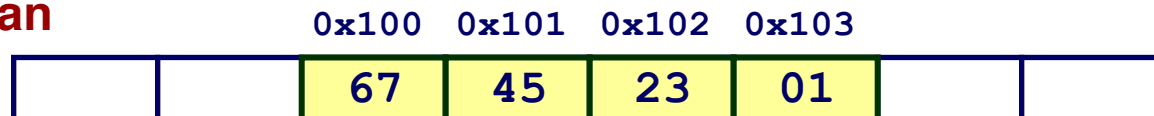   - Least significant byte has lowest address (LSB first)

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Representing Integers

Hex: 00003B6D

int A = 15213;

Address Increase

Little-E	Big-E
6D	00
3B	00
00	3B
00	6D

Hex: FFFFC493

int B = -15213;

Little-E	Big-E
93	FF
C4	FF
FF	C4
FF	93

Representing Integers

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**

int val[5]; | 1 | 5 | 2 | 1 | 3 |

address x $x+4$ $x+8$ $x+12$ $x+16$ $x+20$
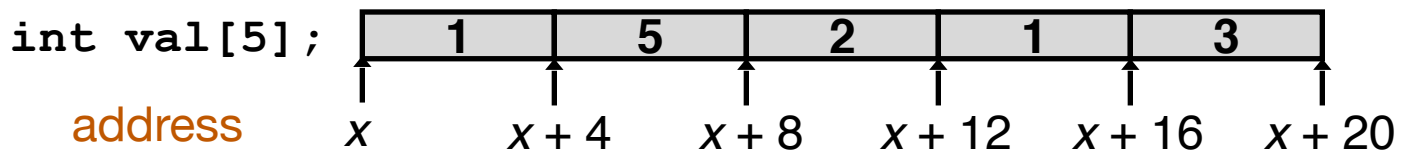
Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	$x+4$
val + *i*	int *	$x+4i$
&val[2]	int *	$x+8$
val[5]	int	??
*(val+1)	int	5

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

$$\begin{bmatrix} \texttt{A[0][0]} & \bullet\ \bullet\ \bullet & \texttt{A[0][C-1]} \\ \bullet & & \bullet \\ \bullet & & \bullet \\ \bullet & & \bullet \\ \texttt{A[R-1][0]} & \bullet\ \bullet\ \bullet & \texttt{A[R-1][C-1]} \end{bmatrix}$$

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

$$\begin{bmatrix} \texttt{A[0][0]} & \bullet\ \bullet\ \bullet & \texttt{A[0][C-1]} \\ & \bullet & \bullet \\ & \bullet & \bullet \\ & \bullet & \bullet \\ \texttt{A[R-1][0]} & \bullet\ \bullet\ \bullet & \texttt{A[R-1][C-1]} \end{bmatrix}$$
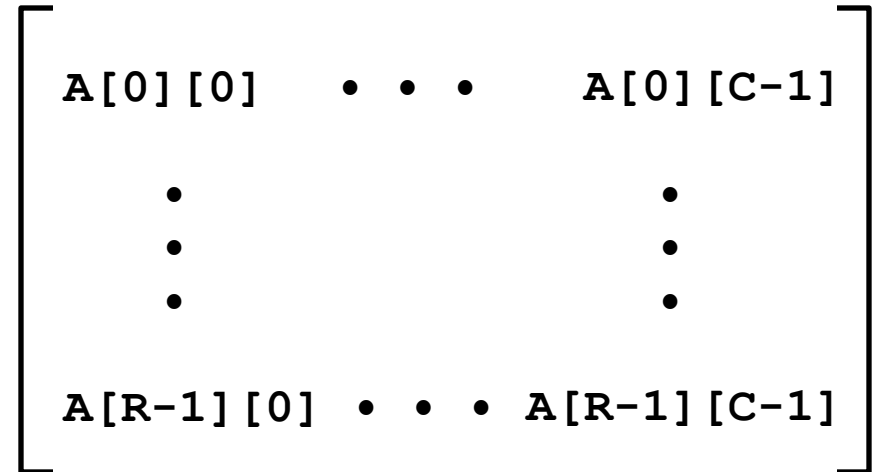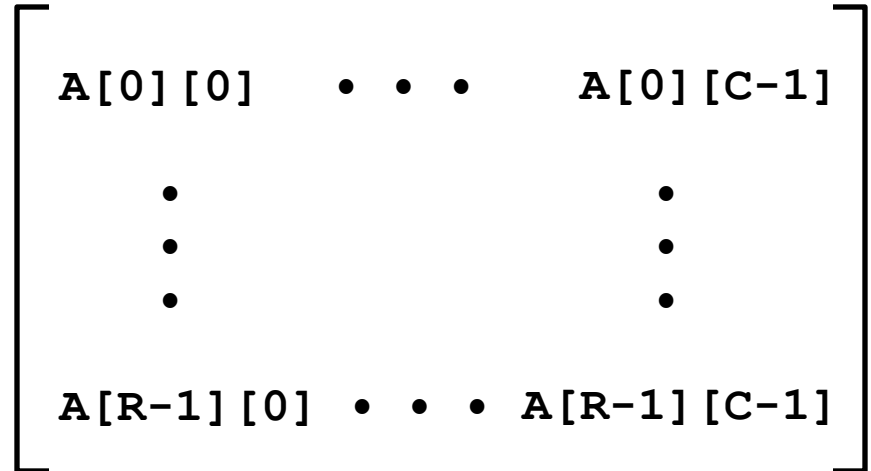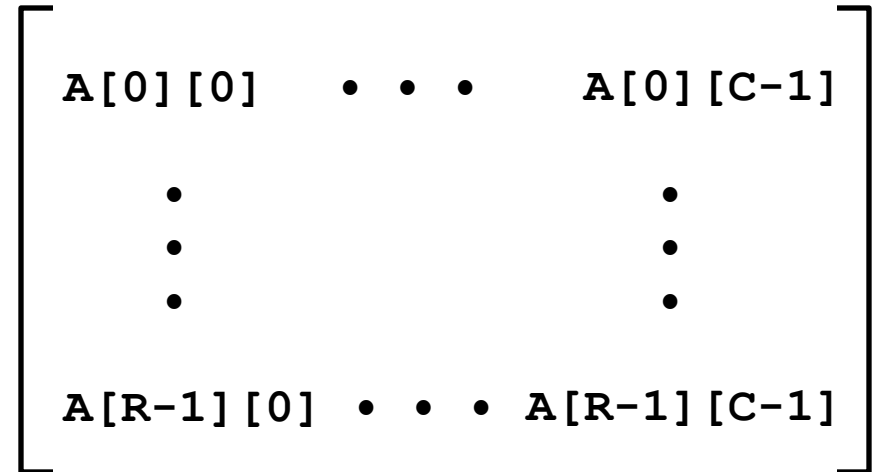
Multidimensional (Nested) Arrays

- 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];
```

$$
\begin{bmatrix}
A[0][0] & \cdots & A[0][C-1] \\
& \vdots & \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{bmatrix}
$$

| 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
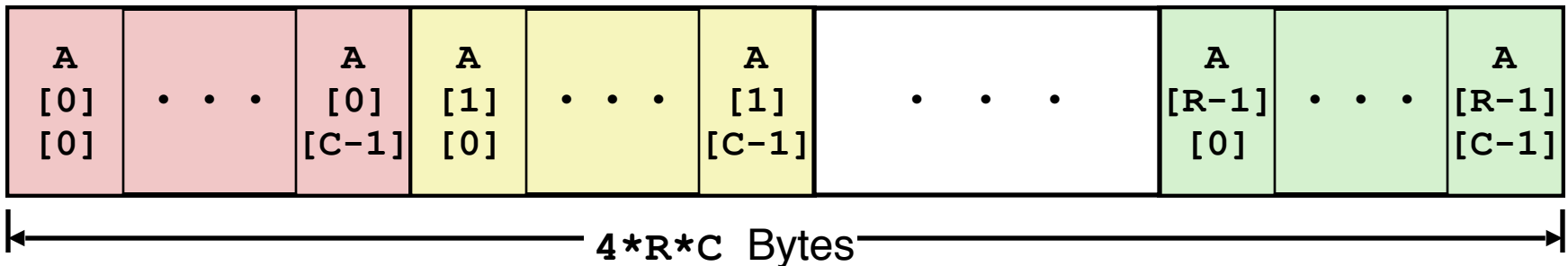
Nested Array Row Access

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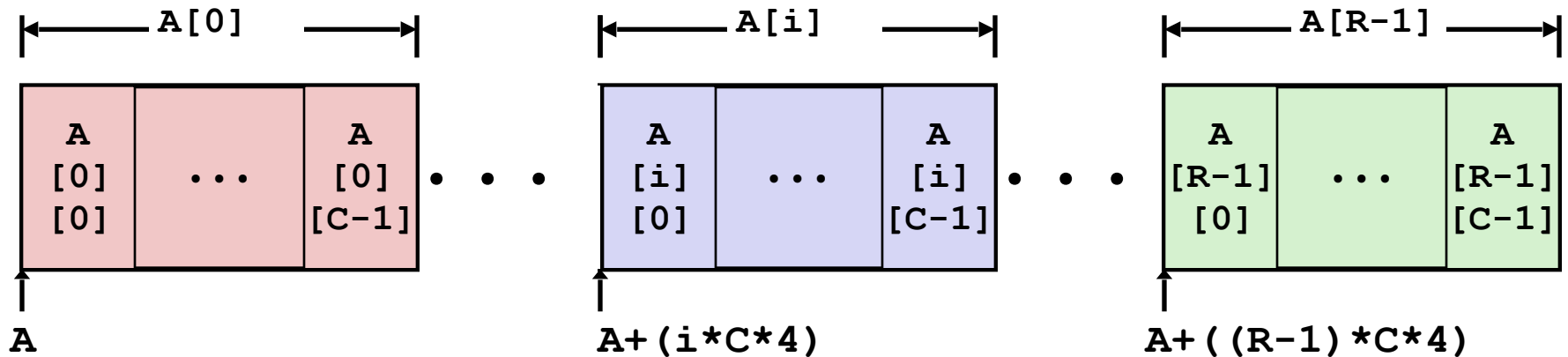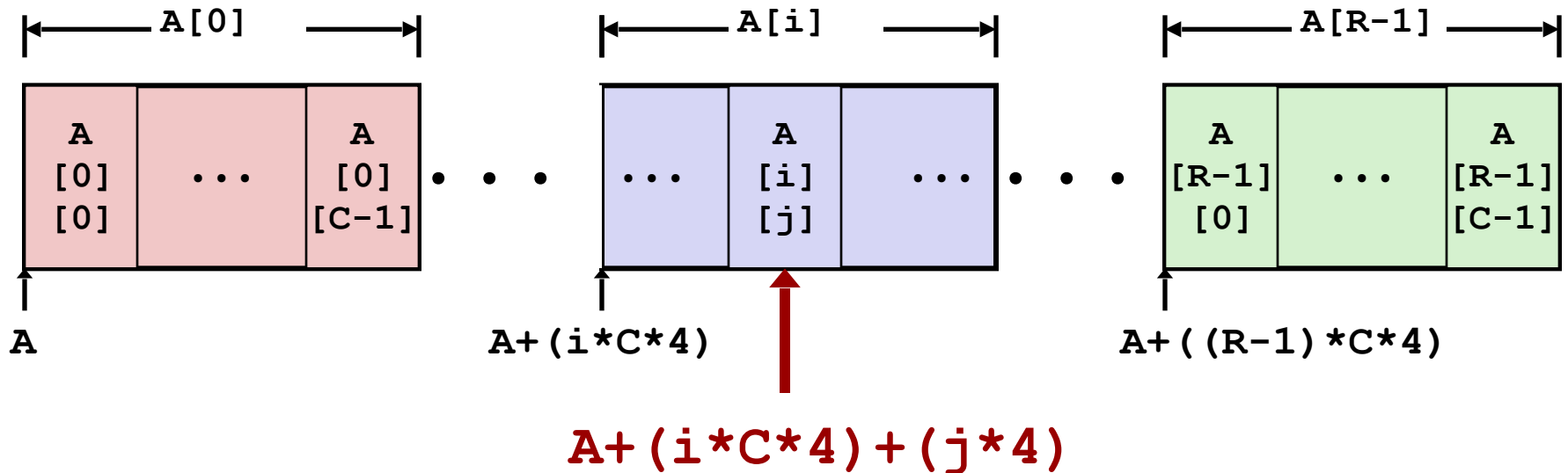 - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address `A +` $i * (C * K) + j * K = A + (i * C + j) * K$

`int A[R][C];`

A+(i*C*4)+(j*4)

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;
};
```

r

a		i	next
0	16	24	32
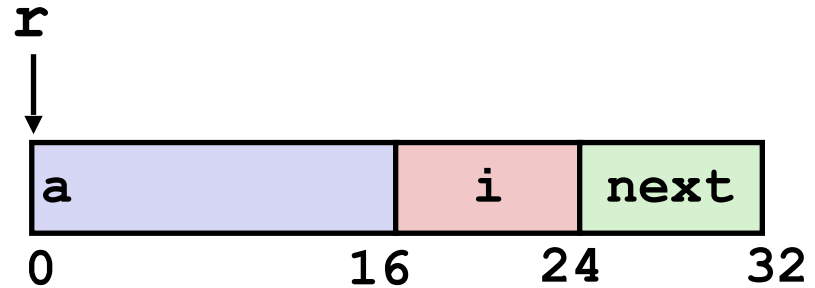
- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within struct by names
  - Members may be of different types

Access Struct Members

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

r

a	i	next

0　　　　　　　16　　24　　32
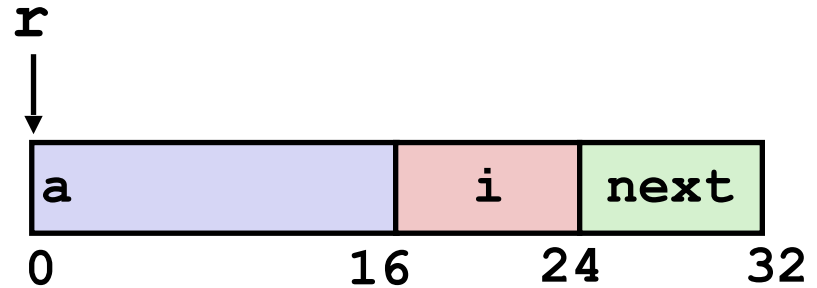
- Given a struct, we can use the . operator:
    - `struct rec r1; r1.i = val;`

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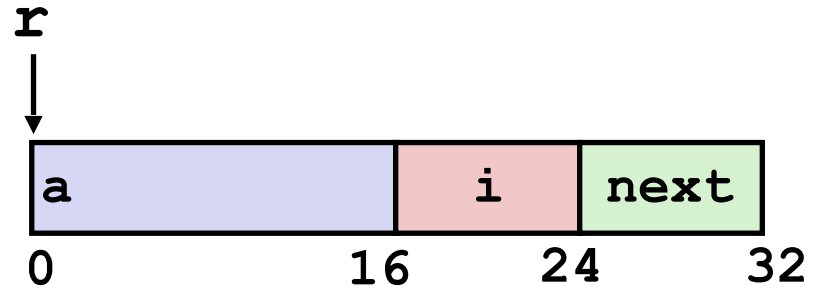
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

r

a		i	next

0   16   24   32

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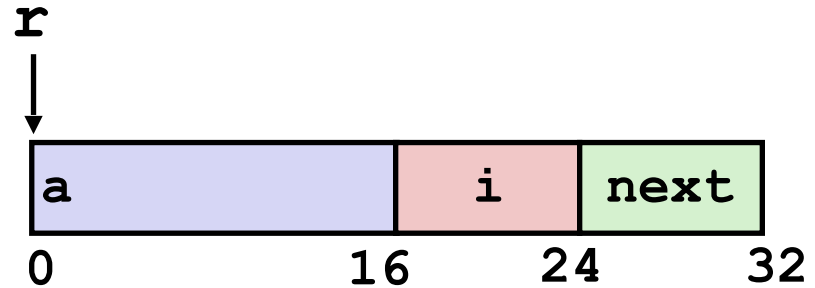
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

r

a		i	next
0	16	24	32

- 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;
};
```

r

a	i	next
0	16	24	32

- 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`?
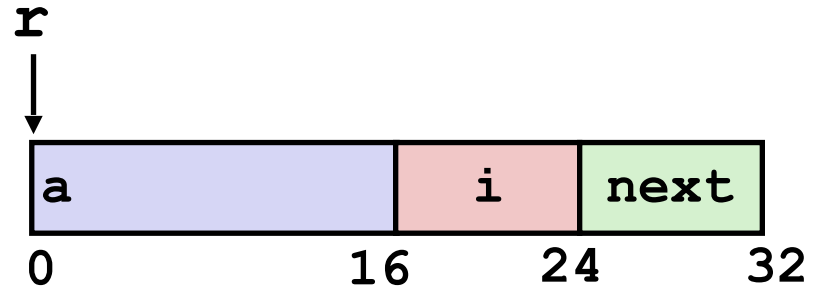   - Using ***** and **.** operators: `(*r).i = val;`
    - Or simply, the -> operator for short: `r->i = val;`

Generating Pointer to Structure Member
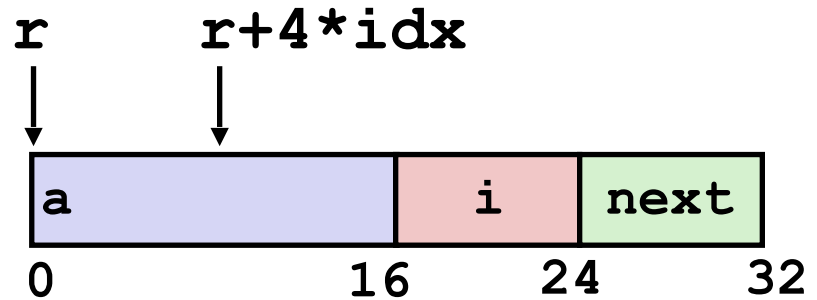
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

r   r+4*idx

a	i	next

0   16   24   32

Generating Pointer to Structure Member

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

r   **r+4*idx**

0   16   24   32
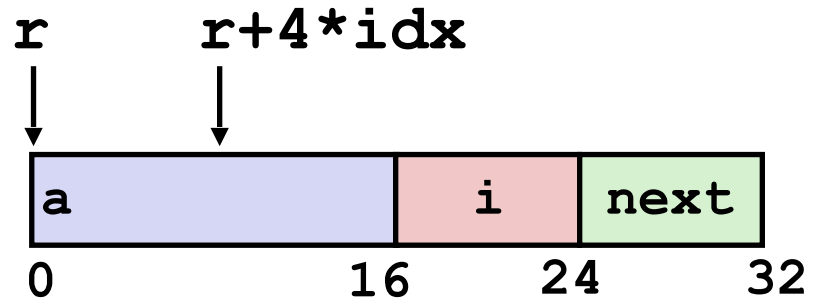
```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```

Generating Pointer to Structure Member

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

r   r+4*idx

a	i	next

0   16   24   32

```
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;
};
```
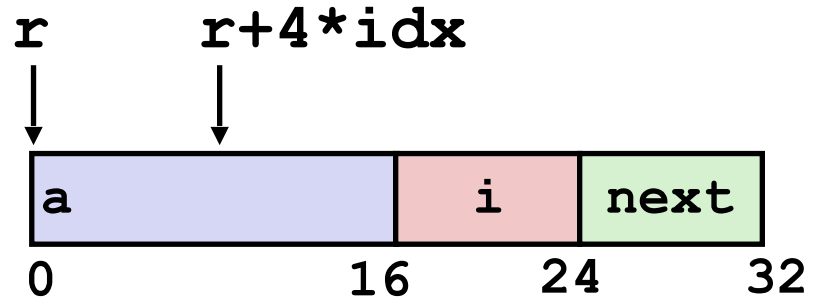
r   r+4*idx

0   16   24   32
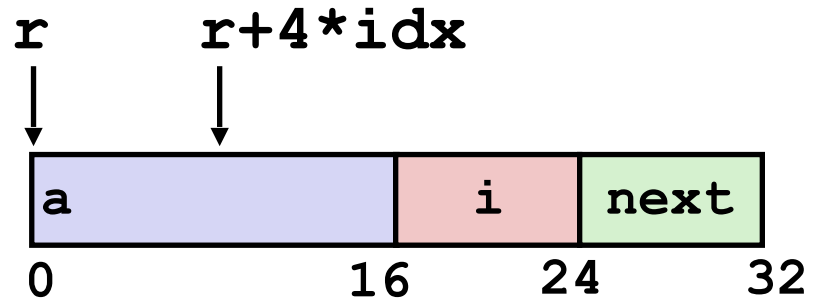
```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```

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

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

Alignment

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

Alignment

- Unaligned Data

c	i[0]	i[1]	v

p   p+1   p+5   p+9   p+17

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

Alignment

- Unaligned Data

c	i[0]	i[1]	v

p   p+1   p+5   p+9   p+17

```
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


```
c    i[0]     i[1]              v
p p+1      p+5      p+9                    p+17
```

```
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**


```
c
p+0
```

Multiple of 8

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**

Multiple of 4

Multiple of 8

Alignment

- Unaligned Data

c	i[0]	i[1]	v

p   p+1   p+5   p+9   p+17

```
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**

c	*3 bytes*	i[0]	i[1]

p+0   p+4   p+8

↑ Multiple of 4

↑ Multiple of 8

Alignment

- Unaligned Data

```
c    i[0]    i[1]    v
```
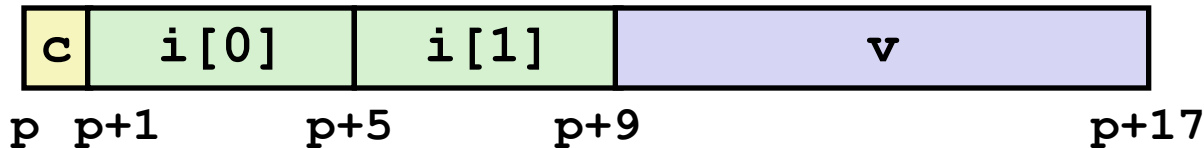p p+1   p+5   p+9   p+17

```
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**
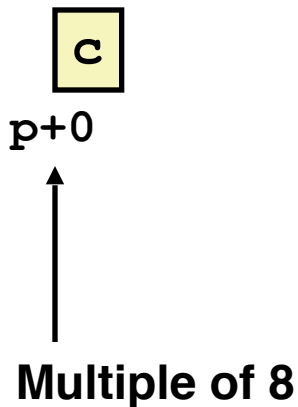
```
c   3 bytes    i[0]    i[1]    4 bytes
```
p+0   p+4   p+8   p+16

Multiple of 4   **Multiple of 8**

Multiple of 8

Alignment

- Unaligned Data

```
c   i[0]   i[1]   v
```

p p+1   p+5   p+9   p+17

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

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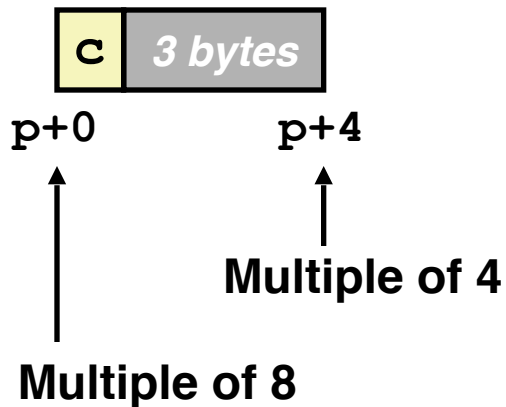 - If the data type requires **K** bytes, address must be multiple of **K**

```
c  3 bytes   i[0]   i[1]   4 bytes   v
```
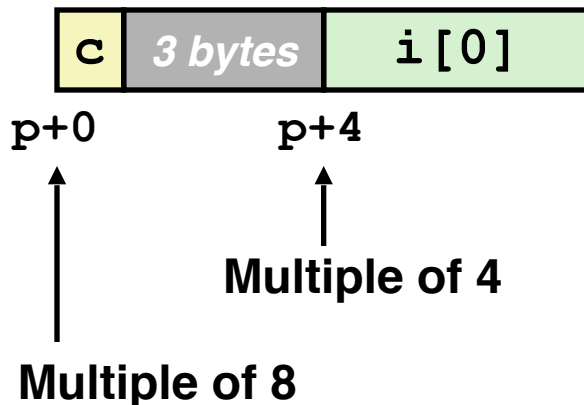
p+0   p+4   p+8   p+16   p+24

Multiple of 4

Multiple of 8

Multiple of 8

Multiple of 8

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;
```

v	i[0]	i[1]	c	7 bytes

p+0　　　　　　　p+8　　　　　　　p+16　　　　　　　p+24
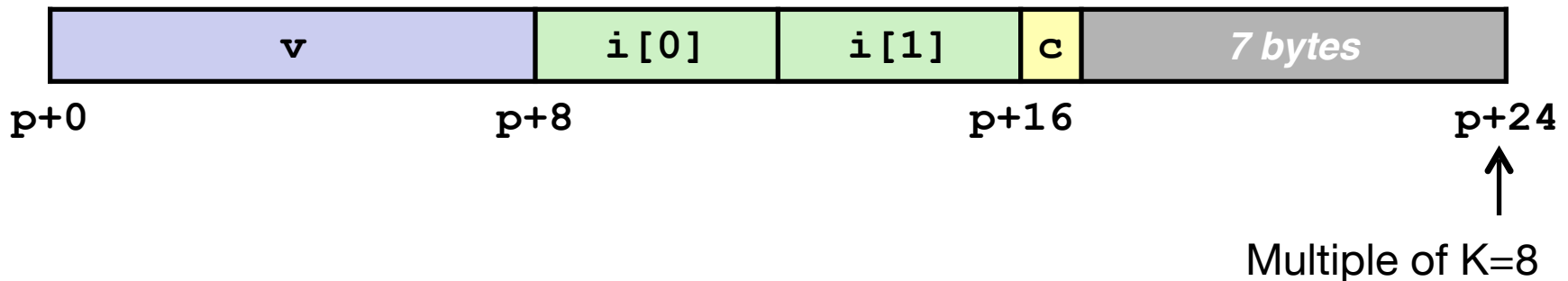
Multiple of K=8

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];
```
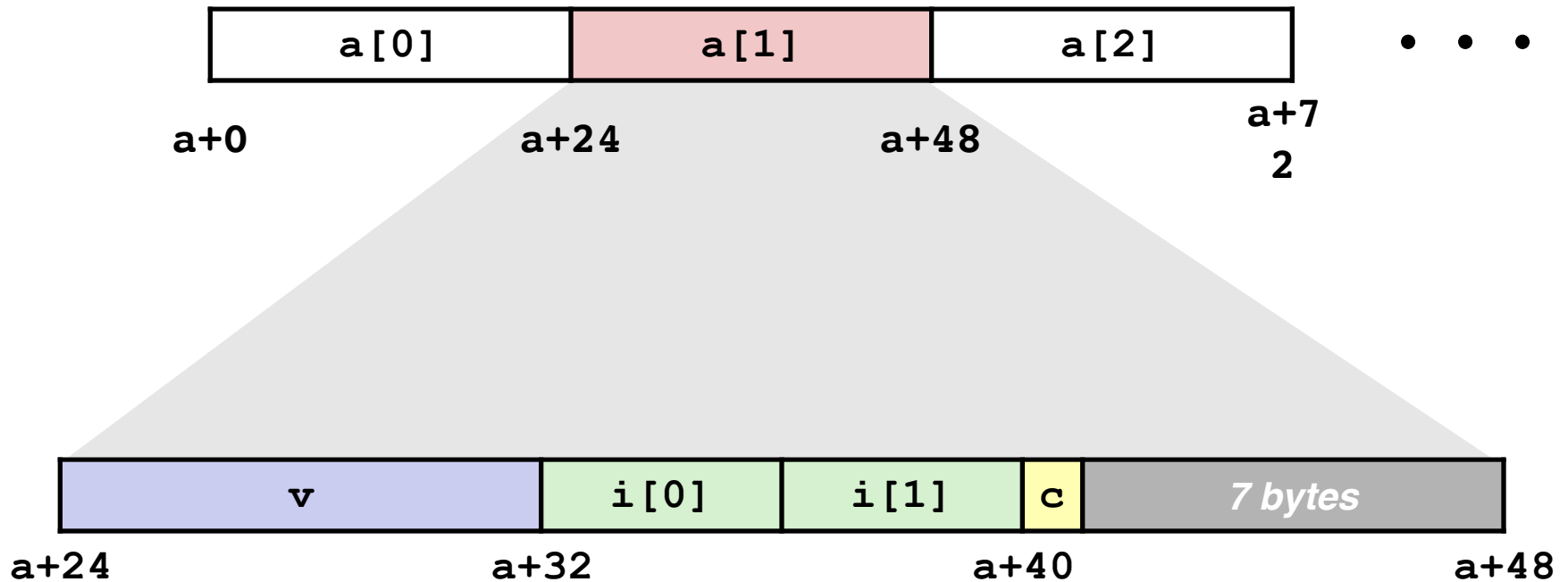
Saving Space

- Put large data types first 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;
```

| c | *3 bytes* | i | d | *3 bytes* |
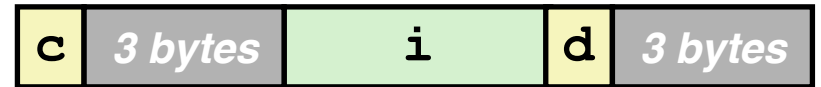
```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

| i | c | d | *2 bytes* |

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

```c
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

```c
/* 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

```c
/* 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
- Similar problems with other library functions
  - **strcpy, strcat**: Copy strings of arbitrary length
  - **scanf, fscanf, sscanf,** when given **%s** conversion specification

```
/* 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

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

```c
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:01234567890123456789 0123
01234567890123456789 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:01234567890123456789012 3
01234567890123456789012 3
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

Buffer Overflow Stack Example

Before call to gets
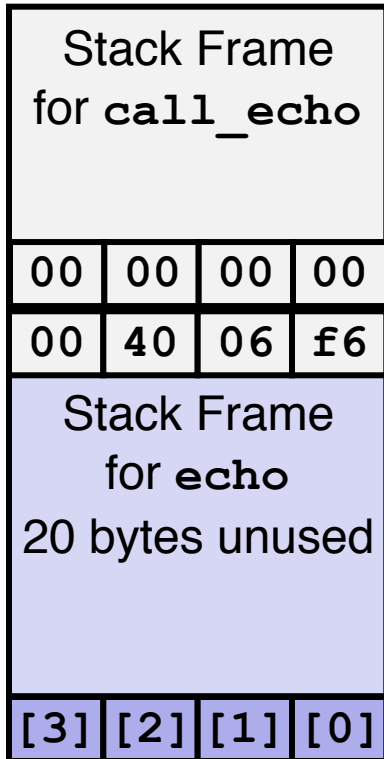
Stack Frame for **call_echo**

00	00	00	00
00	40	06	f6

Stack Frame for **echo** 20 bytes unused

[3]	[2]	[1]	[0]	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
    . . .
```

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:01234567890123456789012
01234567890123456789012
```

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:01234567890123456789901234
Segmentation Fault
```

Overflowed buffer, and corrupt return address

34

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:01234567890123456789 0123
01234567890123456789 0123
```

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!

What to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use "stack canaries"

1. Avoid Overflow Vulnerabilities in Code (!)

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

- For example, use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
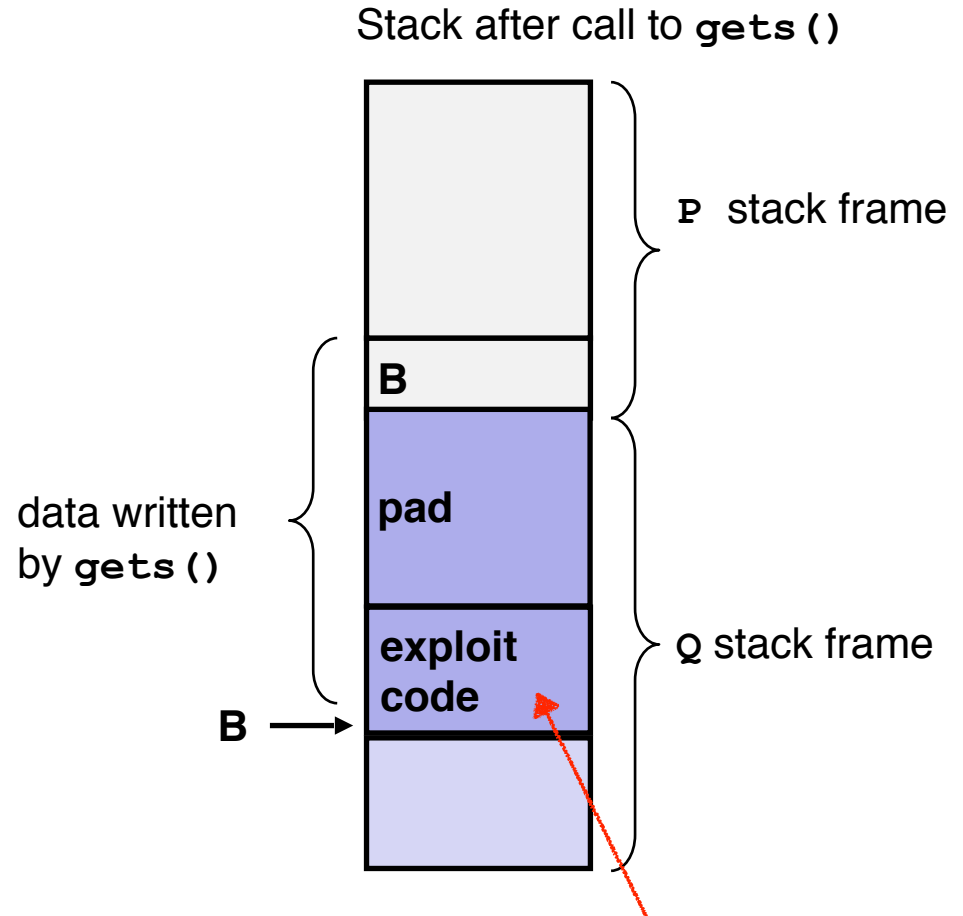   - Or use `%ns` where `n` is a suitable integer

2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code

Stack base

Random allocation

main

Other func.

B?

pad

exploit code

B? ⟶
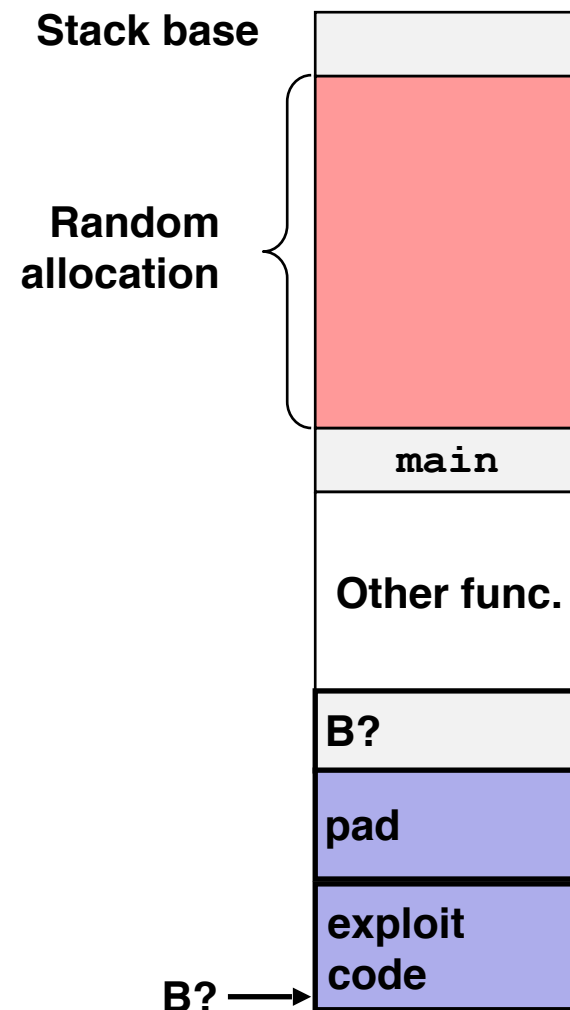
2. System-Level Protections can help

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
  - Can execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable

Stack after call to `gets()`

P stack frame

B

data written by `gets()`

pad

exploit code

B →

Q stack frame

Any attempt to execute this code will fail

3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - `-fstack-protector`
  - Now the default (disabled earlier)

3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
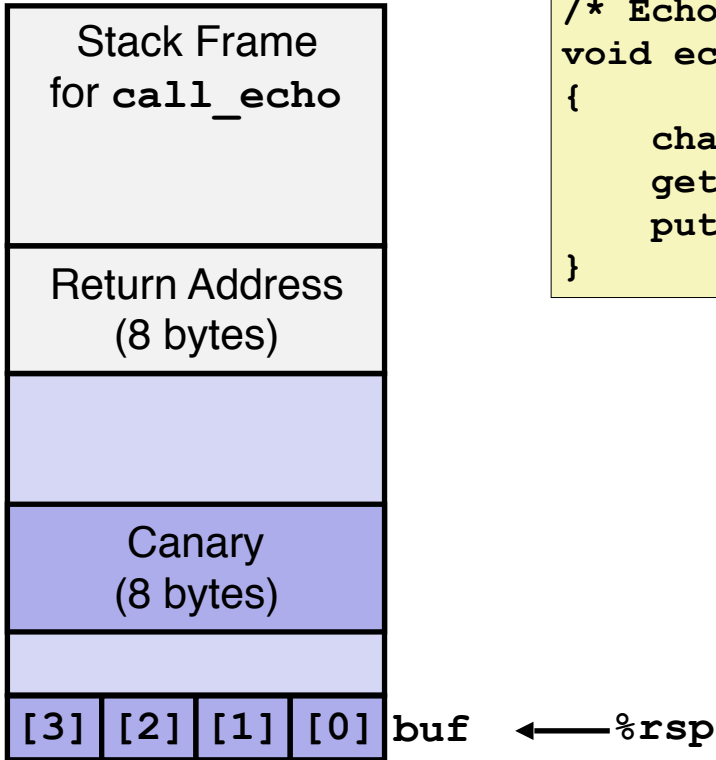 - `-fstack-protector`
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Setting Up Canary

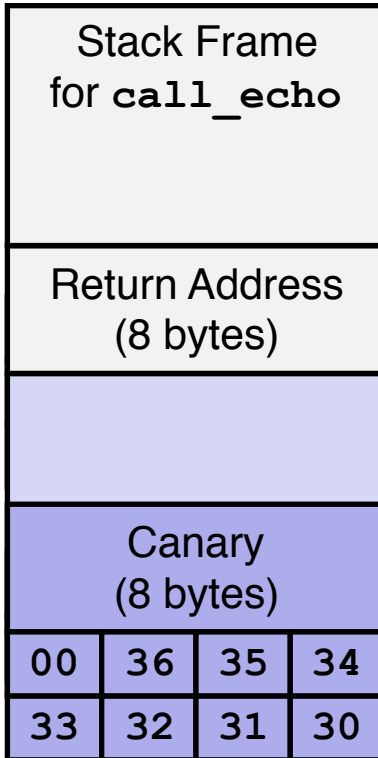Before call to gets


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

```
echo:
    . . .
    movq      %fs:40, %rax  # Get canary
    movq      %rax, 8(%rsp) # Place on stack
    xorl      %rax, %rax    # Erase canary
    . . .
```

Checking Canary

After call to gets

Stack Frame for `call_echo`
Return Address (8 bytes)
Canary (8 bytes)

00	36	35	34
33	32	31	30

`buf` ←——— `%rsp`

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

Input: *0123456*

```
echo:
    . . .
    movq      8(%rsp), %rax     # Retrieve from stack
    xorq      %fs:40, %rax      # Compare to canary
    je        .L6               # If same, OK
    call      __stack_chk_fail  # FAIL
.L6:. . .
```