CSC 252: Computer Organization
Spring 2025: Lecture 10

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Mid-term exam March 7 (Friday) in class.
- Programming assignment 3 will be out day, and it’s due the Friday after the spring break.

Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

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.
    - Take 255/455.

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

Register Example: `incr`

Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

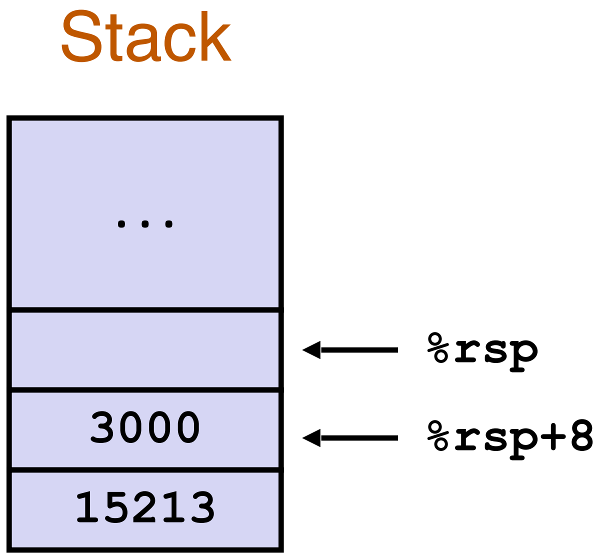
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Stack Example: `call_add`

```
long call_add() {
    long v1 = 15213;
    long v2 = 3000;
    long v3 = add(&v1, &v2);
    return v2+v3;
}
```

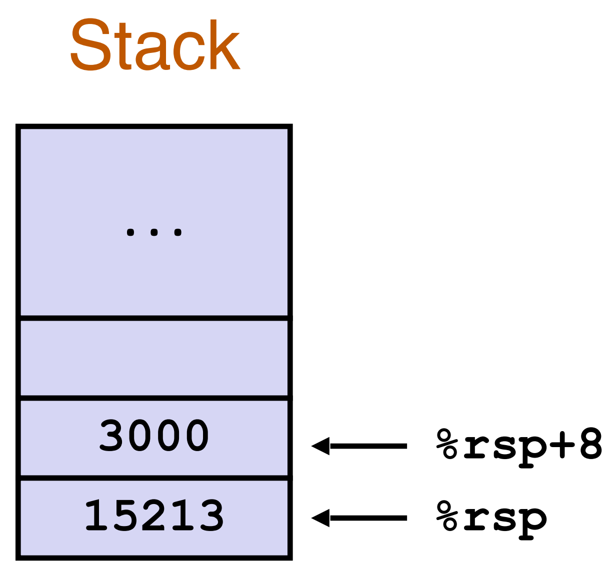
```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Example: `call_add`

```c
long call_add() {
    long v1 = 15213;
    long v2 = 3000;
    long v3 = add(&v1, &v2);
    return v2+v3;
}
```

```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

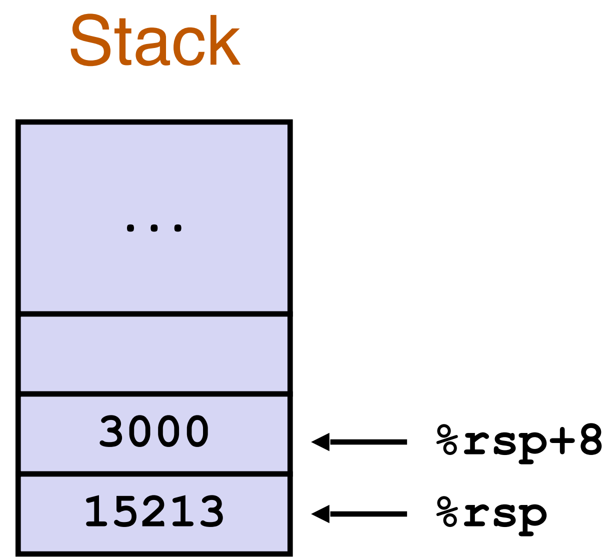


Register	Value(s)
%rdi	&v1
%rsi	&v2

Stack Example: `call_add`

```
long call_add() {
    long v1 = 15213;
    long v2 = 3000;
    long v3 = add(&v1, &v2);
    return v2+v3;
}
```

```
call_add:
  subq     $16, %rsp
  movq     $15213, (%rsp)
  movq     $3000, 8(%rsp)
  leaq     (%rsp), %rdi
  leaq     8(%rsp), %rsi
  call     add
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

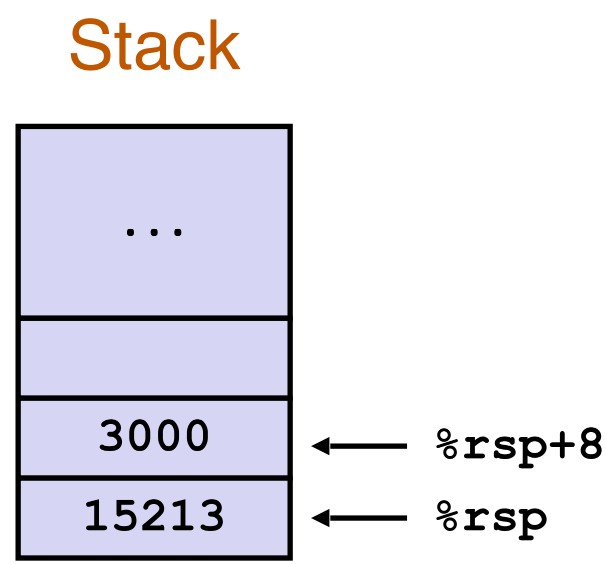


Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	18213

Stack Example: `call_add`

```
long call_add() {
    long v1 = 15213;
    long v2 = 3000;
    long v3 = add(&v1, &v2);
    return v2+v3;
}
```

```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

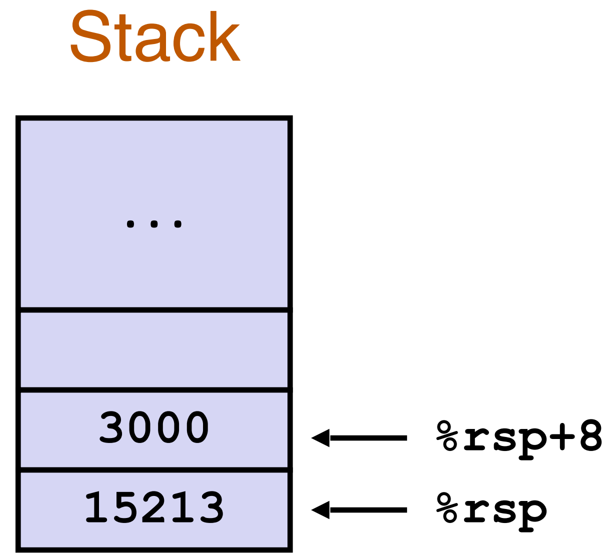


Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	21213

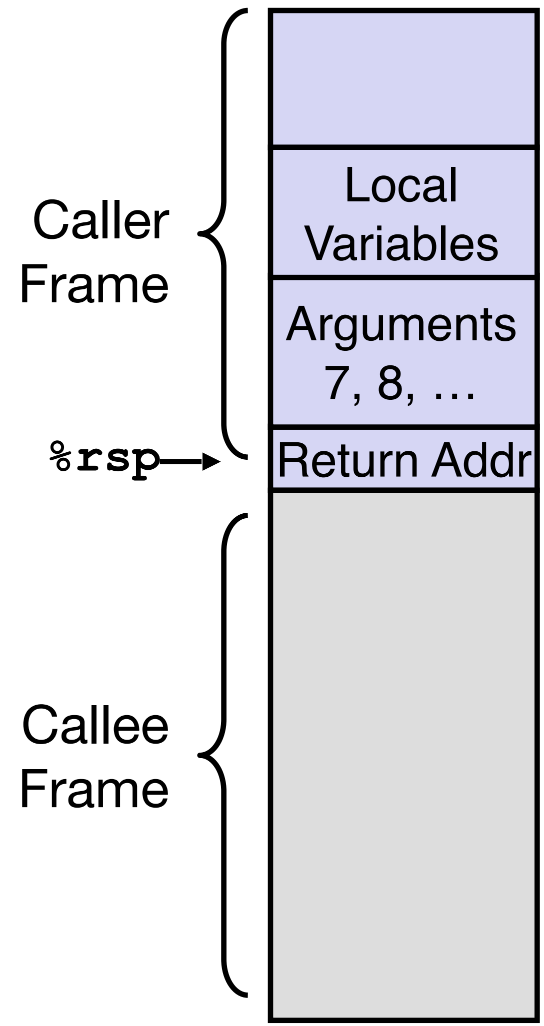
Stack Example: `call_add`

```c
long call_add() {
    long v1 = 15213;
    long v2 = 3000;
    long v3 = add(&v1, &v2);
    return v2+v3;
}
```

```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Frame (So Far…)



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

- 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

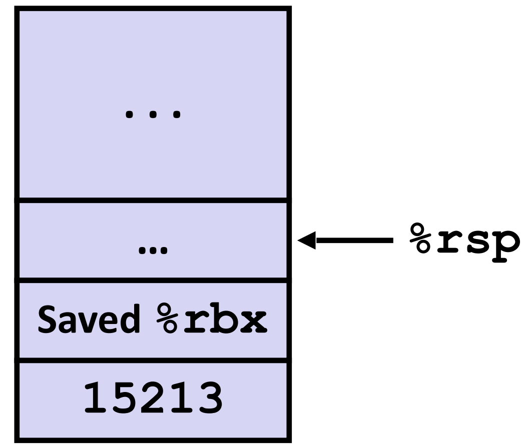
- Conventions used in x86-64 (*Part of the Calling Conventions*)
  - Some registers are saved by caller, some are by callee.
  - Caller saved: `%rdi,%rsi,%rdx,%rcx,%r8,%r9,%r10,%r11`
  - Callee saved: `%rbx,%rbp,%r12,%r13,%14,%r15`
  - `%rax` holds return value, so implicitly caller saved
  - `%rsp`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

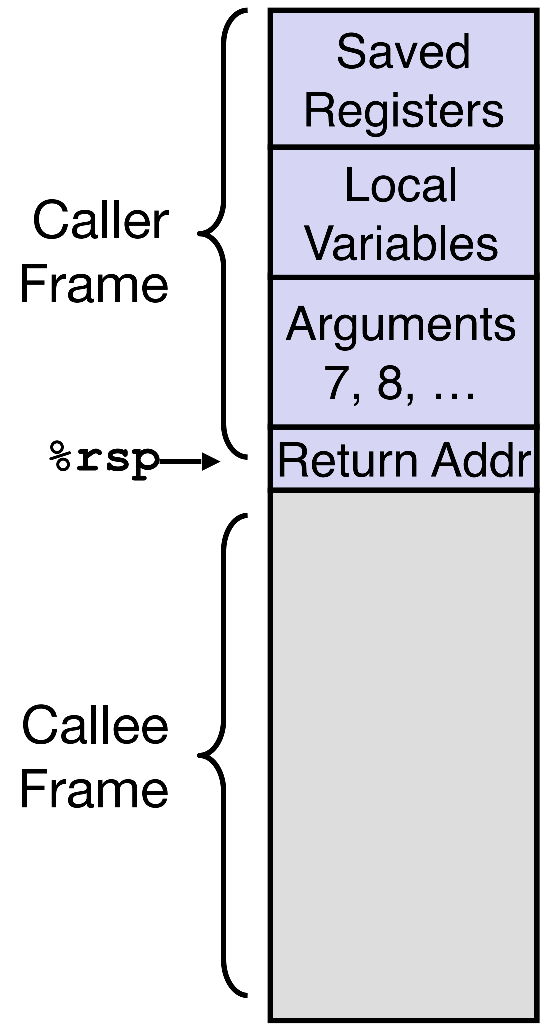
```
call_incr2:
  pushq   %rbx
  pushq   $15213
  movq    %rdi, %rbx
  movl    $3000, %esi
  leaq    (%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $8, %rsp
  popq    %rbx
  ret
```

Stack



- `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



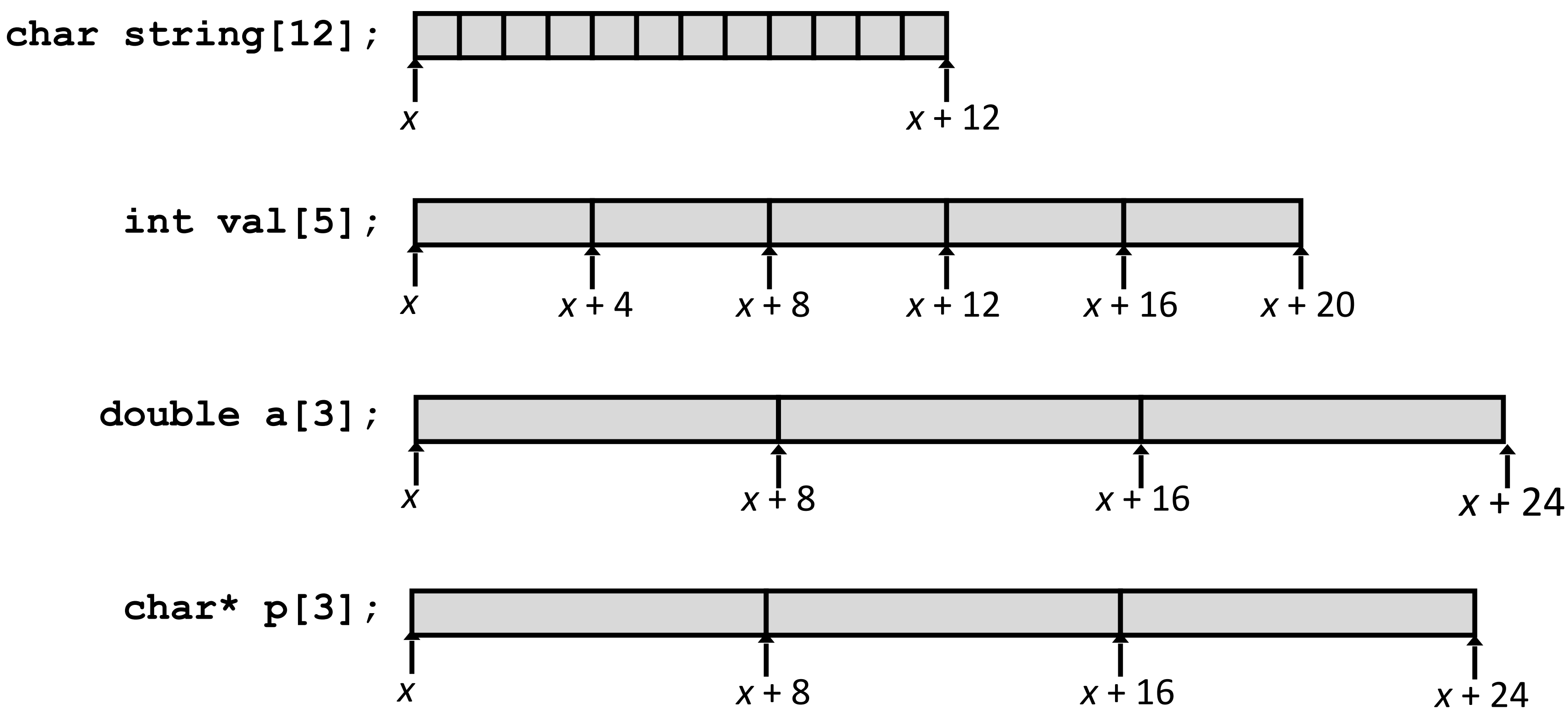
Today: Data Structures and Buffer Overflow

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

Array Allocation: Basic Principle

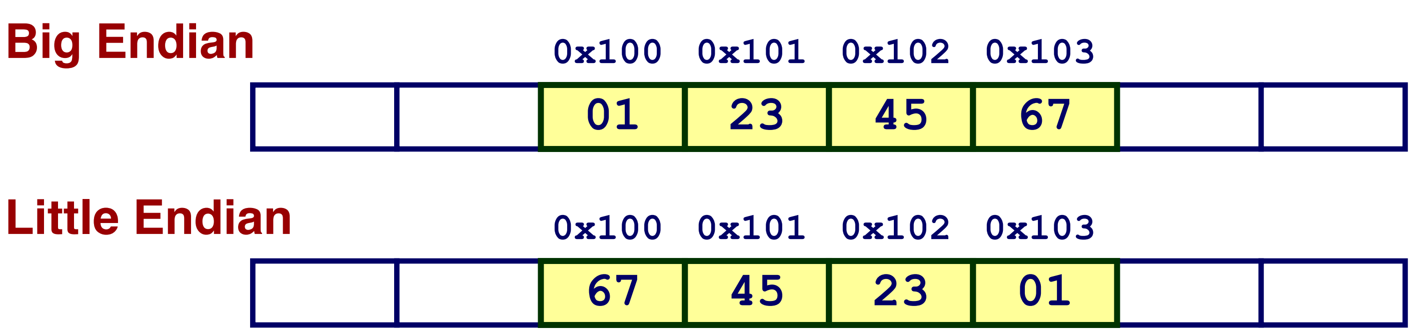
T **A[***L***];**

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



Byte Ordering

- 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)



Representing Integers

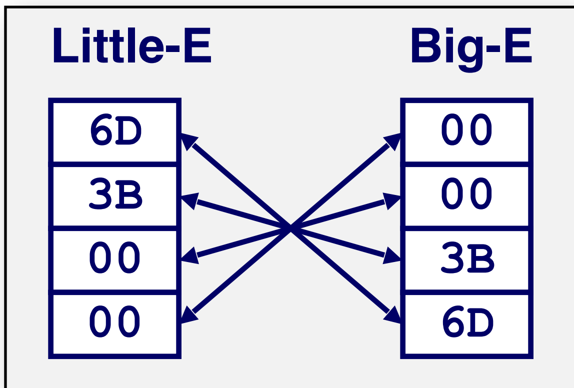
Hex: `00003B6D`

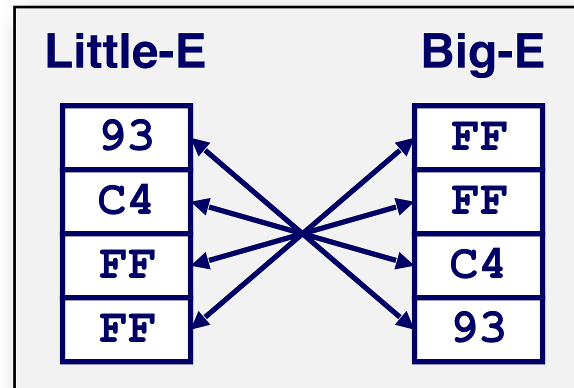
Hex: `FFFFC493`

```
int A = 15213;
```

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

Address Increase





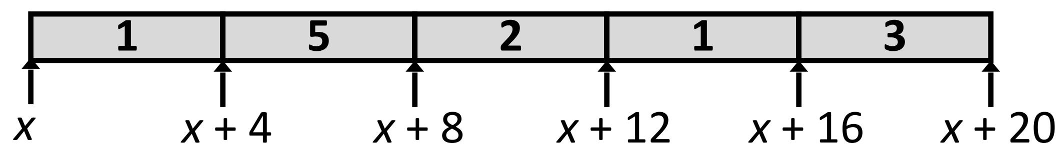
Array Access: Basic Principle

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];

address



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

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

- 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];`

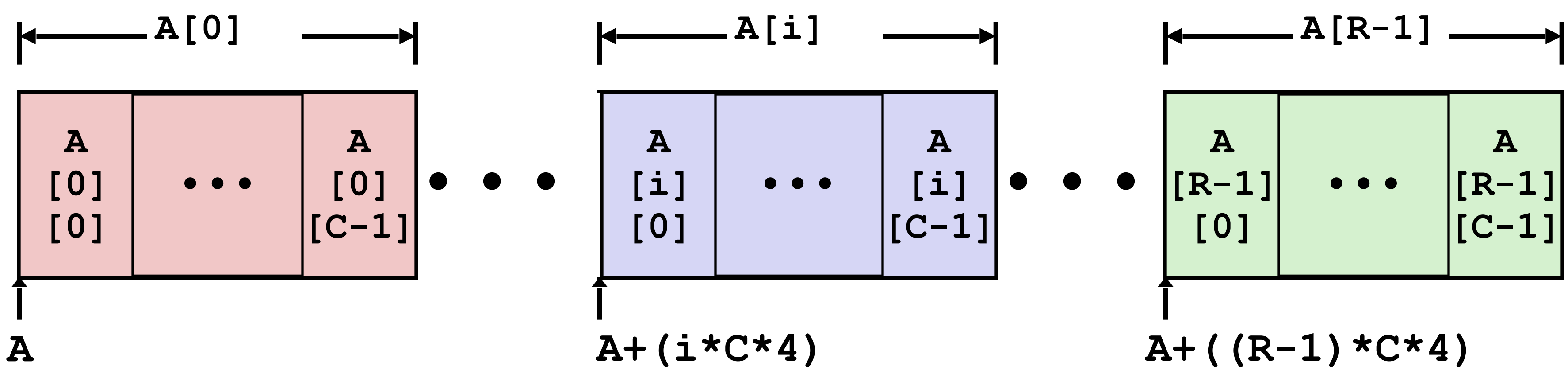
A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

|← **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)

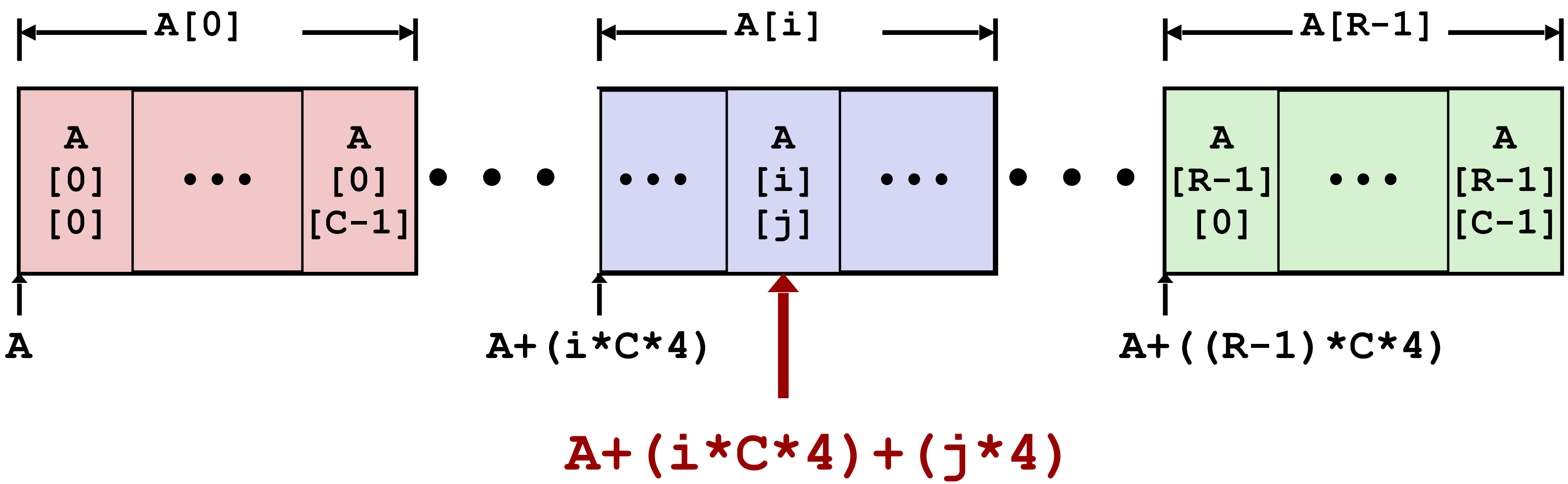
```
int A[R][C];
```



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];
```

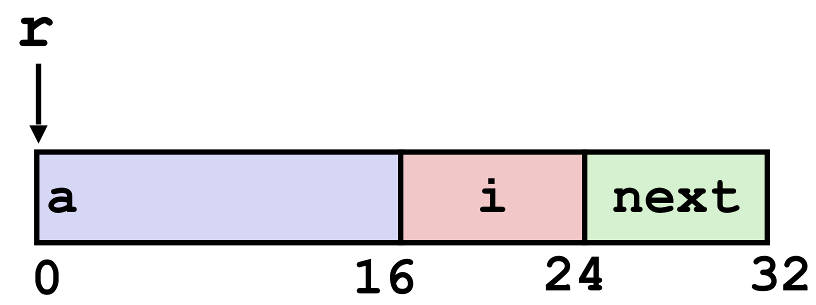


Today: Data Structures and Buffer Overflow

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

Structures

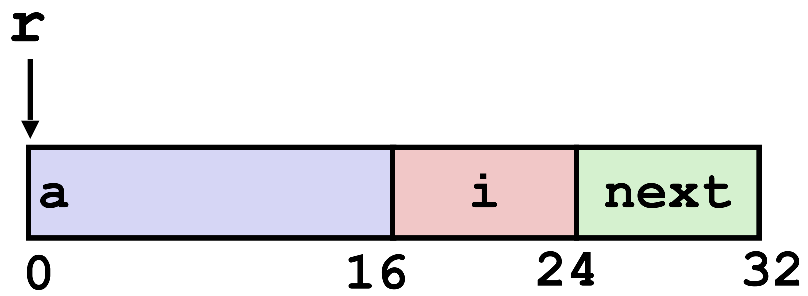
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



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

Access Struct Members

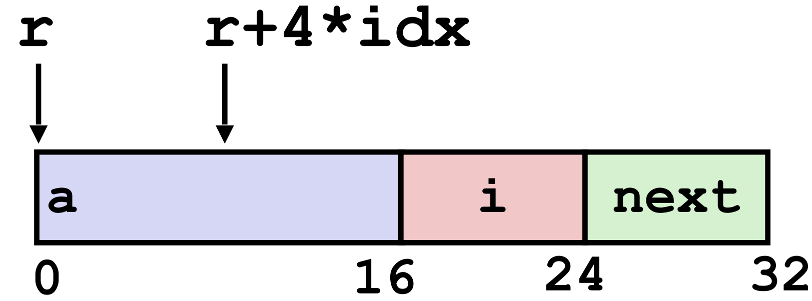
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



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



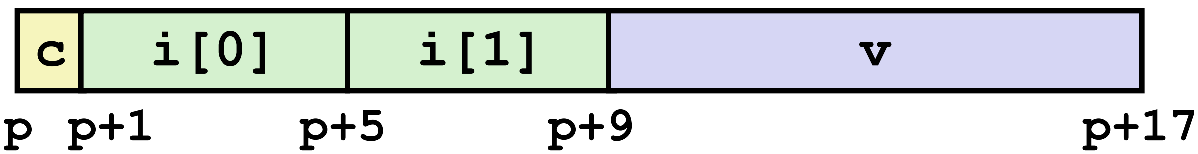

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```

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

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

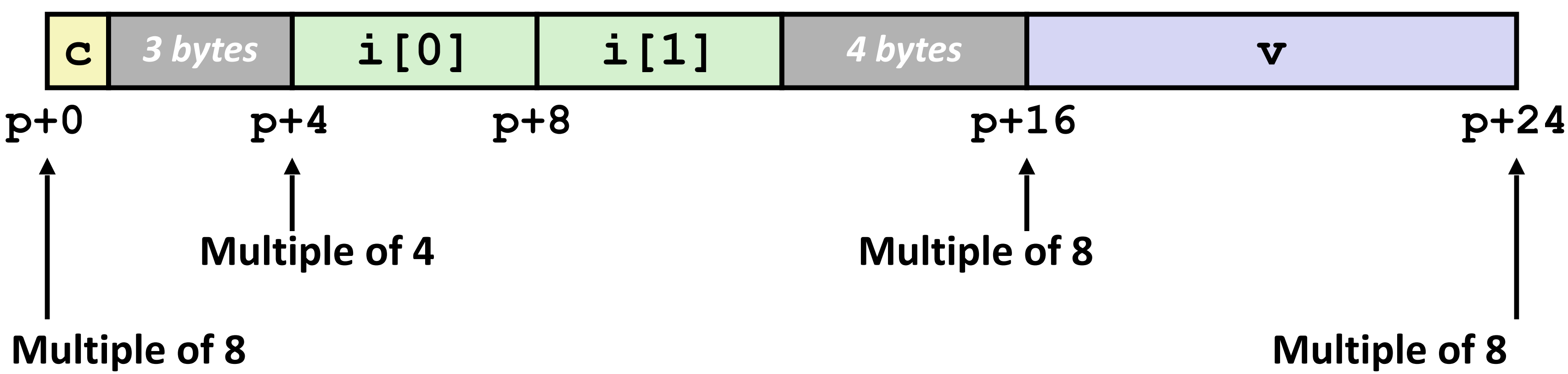
Alignment

- 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 even 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)

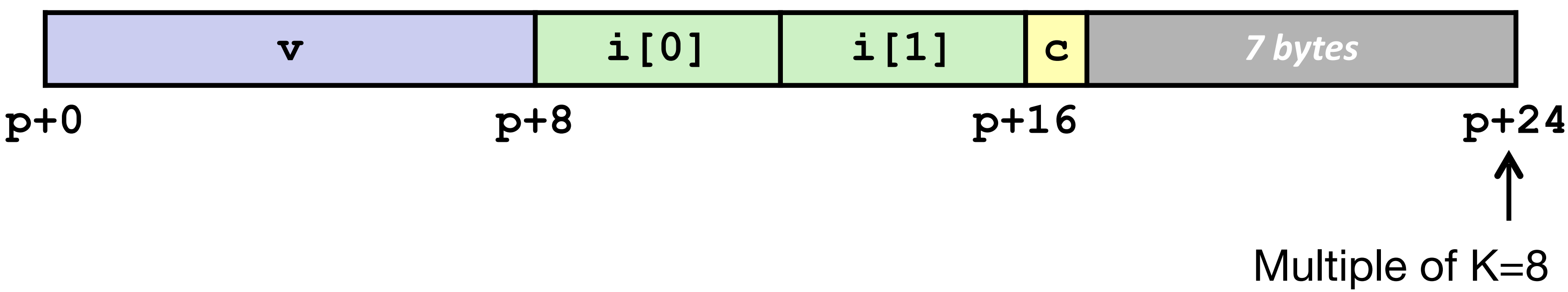
Specific Cases of Alignment (x86-64)

- 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

- 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?!**

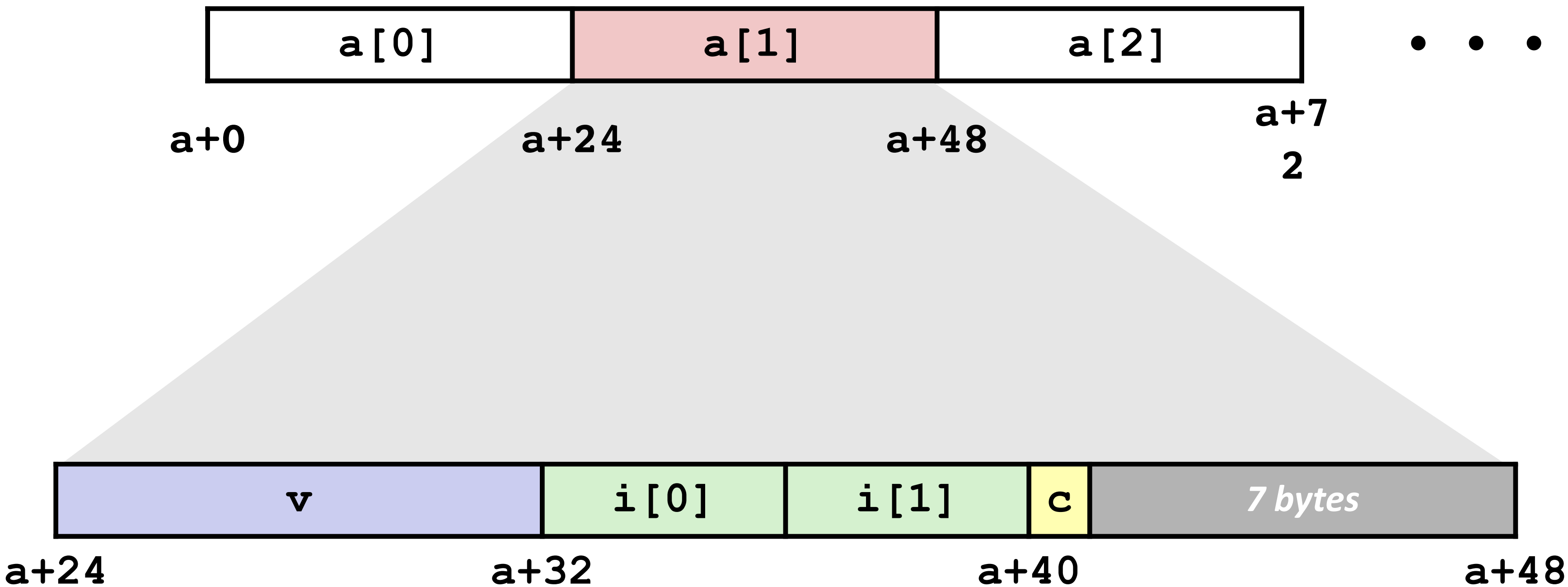
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

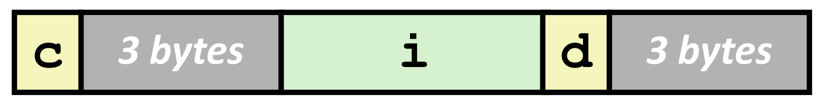
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



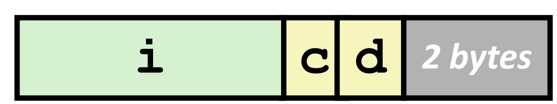
Saving Space

- 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;
```




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



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
}
```

- 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
- 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

```
/* 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
01234567890123456789012
```

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