# CSE 351 AA/BA
# Section 5

# Administrivia

- Hw12 due Friday
- Hw13 due Monday
- Lab 3 is out! Please start on it as soon as possible!
- Office Hours Today
  - Colton: 1:00 - 2:00
  - Tim: 3:00 - 4:00
  - Kashish: 5:00 - 6:00

# *Download the Handout!*

https://courses.cs.washington.edu/courses/cse351/21su/sections/05/cse351_sec5.pdf

Solutions will be posted this evening.

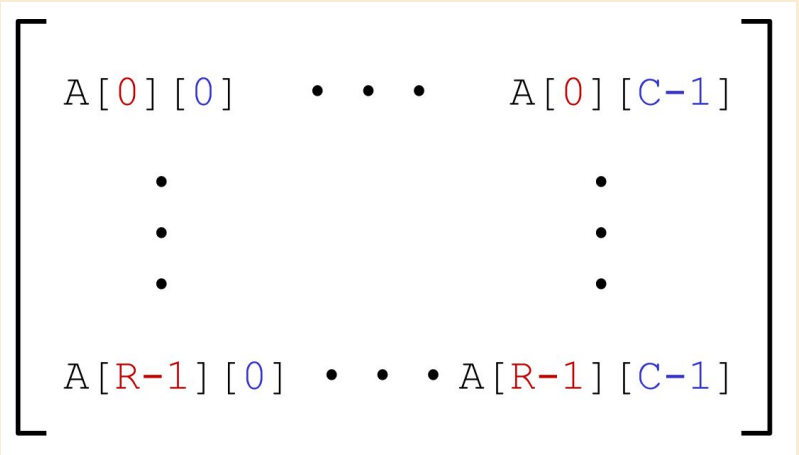
# Arrays

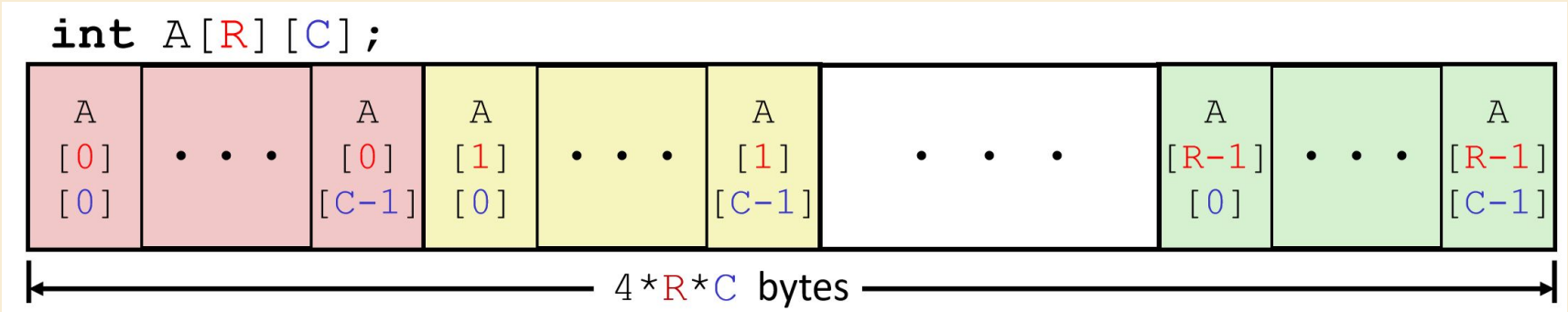
# One Dimensional Arrays

- T A[N]; → array of type T and length N
- Contiguously stored in N*sizeof(T) bytes
- A returns a T*
- A[3] is shorthand for *(A+3) → pointer arithmetic and dereference
- size is not stored like in Java, so there are no bounds checking

# Multidimensional Arrays

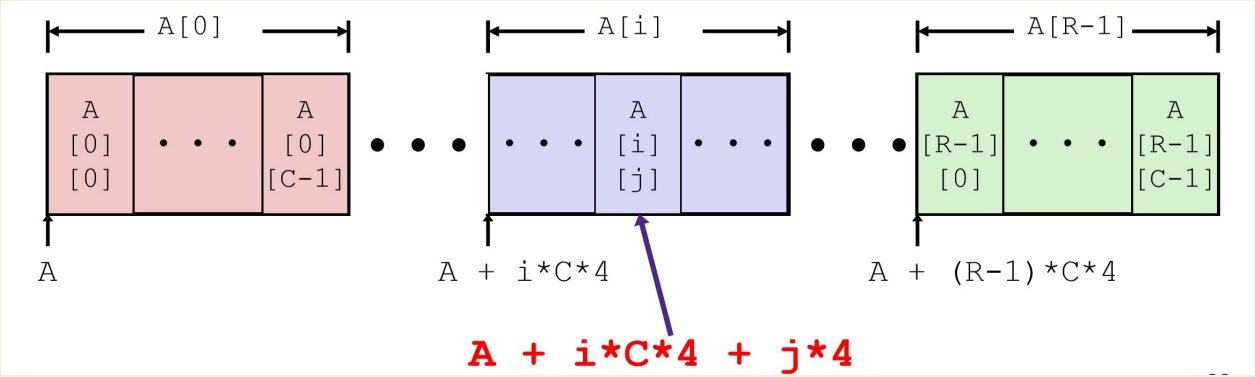
- T A[R][C]; → 2D array of type T with R rows and C columns
- Stored row major contiguously



# Multidimensional Arrays

- T A[R][C]; → A still returns a pointer to the array
- A[i] → gets a ***pointer*** to a row of the array
  - A[i] is the same as A + i * (C * sizeof(T))
- A[i][j] → gets an ***element*** of the array
  - A[i][j] is the same as *(A[i] + j*sizeof(T))

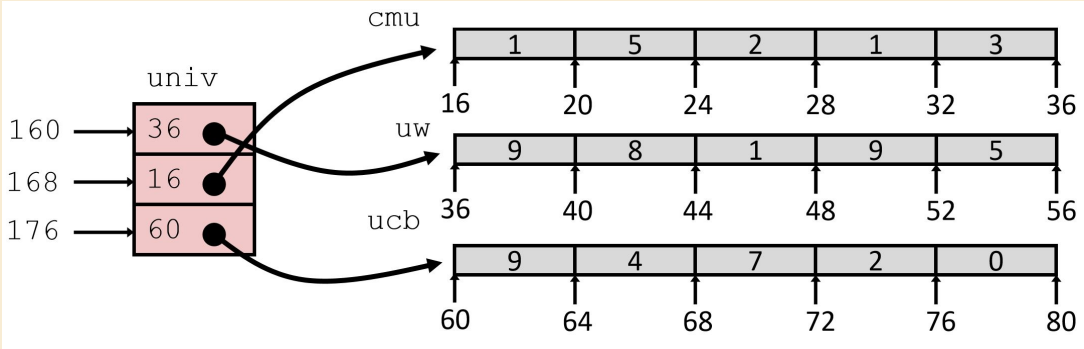


# Multilevel Arrays

- Array of pointers to arrays

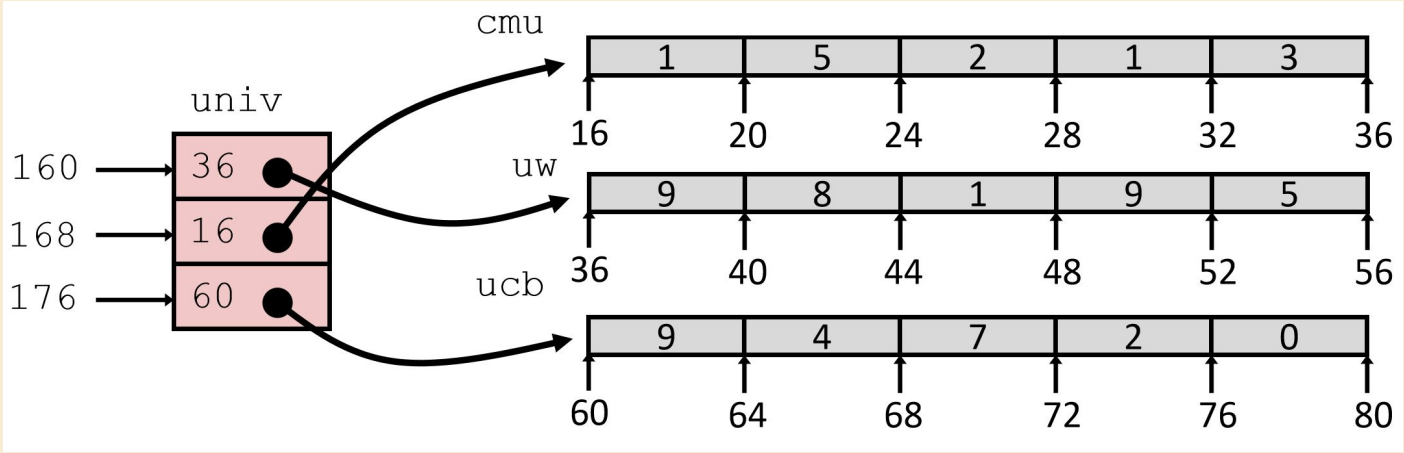
```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```



# Multilevel Arrays

- Same indexing ***notation*** as multidimensional arrays
- univ[0] gets a pointer to uw → *(univ + 0 * sizeof(int))
- univ[0][1] gets the first element of uw → *(univ[0]+1*sizeof(int))
- notice we have 2 dereferences!



# Array Exercise

| | 2-dim array | 2-level array |
|---|---|---|
| Overall Memory Used | M*N*sizeof(int) = 48 B | M*N*sizeof(int) + M*sizeof(int *) = 72 B |
| Largest *guaranteed* continuous chunk of memory | The whole array (48 B) | The array of pointers (24 B) > row array (16 B) |
| Smallest *guaranteed* continuous chunk of memory | The whole array (48 B) | Each row array (16 B) |
| Data type returned by: | `array2D[1]` `int *` | `array2L[1]` `int *` |
| Number of memory accesses to get `int` in the *BEST* case | 1 | 2 |
| Number of memory accesses to get `int` in the *WORST* case | 1 | 2 |

# Array Exercise

Provide a scenario where a 2-dimensional array would be more useful and another where a 2-level array would be more useful.

*2-dimensional:*

2D Array - Creating a table or a matrix where all rows are the same size. This way memory accesses are reduced and less memory is required.

*2-level:*

2-Level Array - When creating a list where different index sizes differ or sub-arrays are subject to replacement. In other words, when the array is more flexible to changes.

# Array Exercise

Sam wants to create a 2D array of the countries of the world that can be accessed alphabetically. Which implementation should Sam choose, and how should he implement this array?

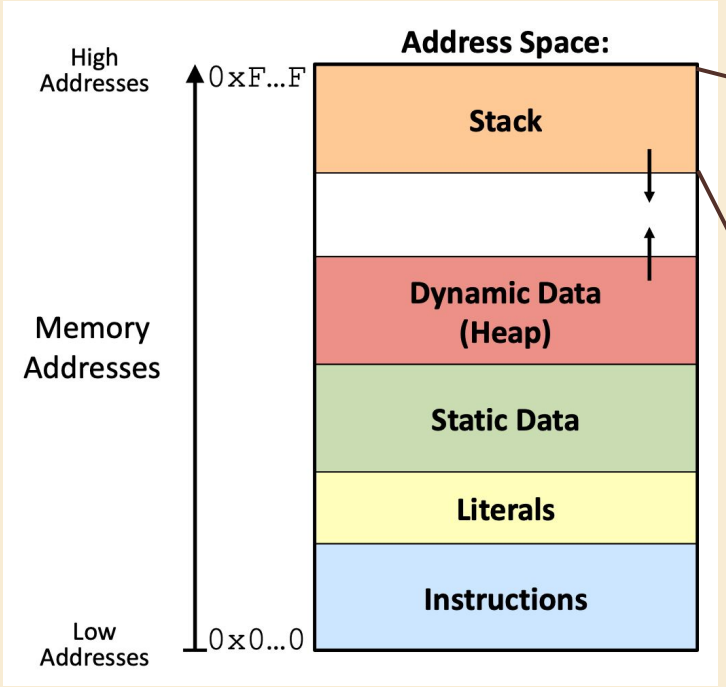
Multidimensional or multilevel?

Sam should use a 2-level array since the amount of countries starting with a given letter will vary (i.e. there are more countries that start with A than Q). He could make an array of pointers from 0 to 25 which would point to custom-sized arrays of country names starting with each corresponding letter of the alphabet.

# Stack & Procedures

# Memory Layout

- Stack is located at the top of our memory layout
- Stack is placed upside down in memory, with higher addresses considered the “bottom” and lower addresses considered the “top”
- There is a dedicated register `%rsp` that points to the current top of the stack



# Stack Frames

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```

Let’s look at how the stack grows and shrinks as the code above executes.

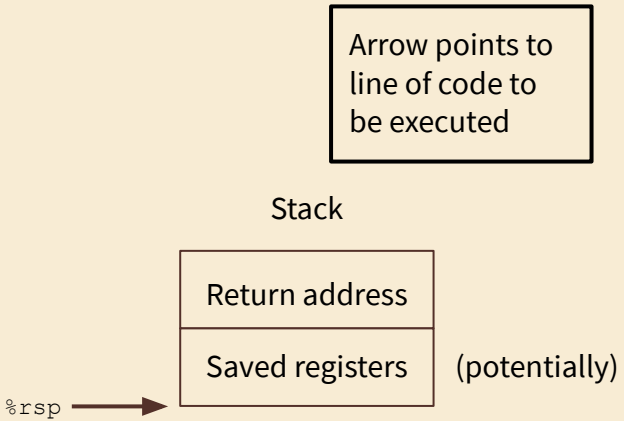
# Stack Frames

- When main starts executing, it has a return address and potentially saved register in its stack frame.

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```





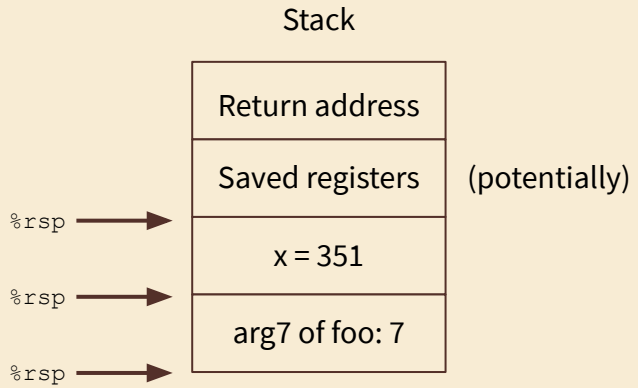
# Stack Frames

- `main` allocates its local variable `x`
- `main` prepares to call `foo` by pushing its arguments (beyond arg7) onto the stack

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```

Arrow points to line of code to be executed



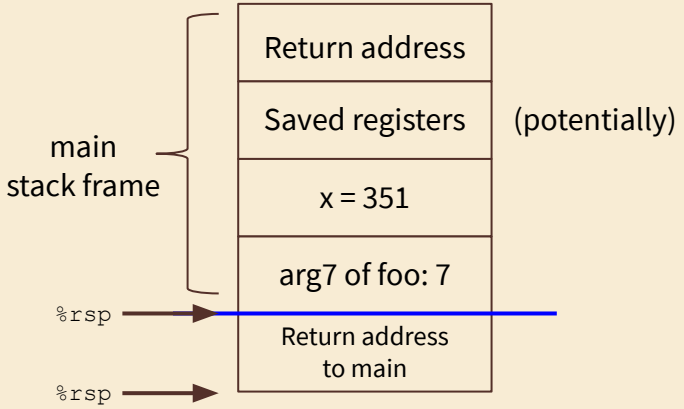
# Stack Frames

Arrow points to line of code to be executed

- main calls `foo` using `callq` in assembly, which pushes the return address to main on the stack
- Return address to main marks the end of main's stack frame and the beginning of `foo`'s stack frame

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```



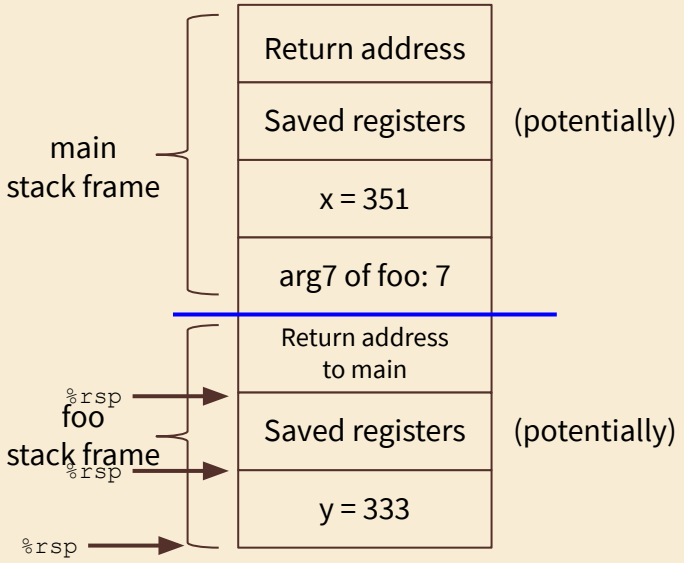
# Stack Frames

Arrow points to line of code to be executed

- `foo` potentially saves registers in its frame
- `foo` allocates its local variable on the stack and is about to return back to main

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```



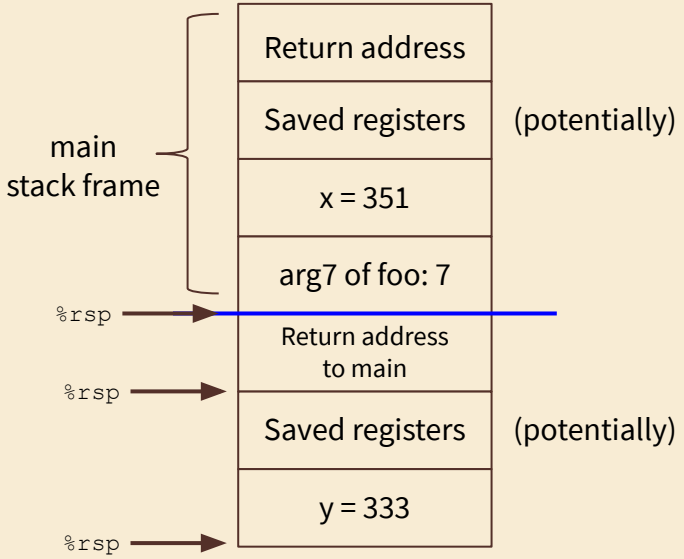
# Stack Frames

Arrow points to line of code to be executed

- `foo` deallocates its local variable and pops saved register values from the stack into corresponding registers
- `foo` returns to main using `ret` in assembly, which pops the return address to main into `%rip` from the stack
- `foo` has done executing, its stack frame is deallocated

```
int main() {
  int x = 351
  foo(1, 2, ..., 7);
}

void foo(int arg1, int arg2, ..., int arg7) {
  int y = 333;
}
```



# Stack Exercise

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07        movzbl (%rdi),%eax
 4005e9: 84 c0           test %al,%al
 4005eb: 74 13           je 400600 <rfun+0x1a>
 4005ed: 53              push %rbx
 4005ee: 48 0f be d8     movsbq %al,%rbx
 4005f2: 48 83 c7 01     add $0x1,%rdi
 4005f6: e8 eb ff ff ff  callq 4005e6 <rfun>
 4005fb: 48 01 d8        add %rbx,%rax
 4005fe: eb 06           jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00  mov $0x0,%eax
 400605: c3              retq
 400606: 5b              pop %rbx
 400607: c3              retq
```

# Stack Exercise

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

a) In terms of the C function, what value is being saved on the stack?

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07        movzbl (%rdi),%eax
 4005e9: 84 c0           test %al,%al
 4005eb: 74 13           je 400600 <rfun+0x1a>
 4005ed: 53              push %rbx
 4005ee: 48 0f be d8     movsbq %al,%rbx
 4005f2: 48 83 c7 01     add $0x1,%rdi
 4005f6: e8 eb ff ff ff  callq 4005e6 <rfun>
 4005fb: 48 01 d8        add %rbx,%rax
 4005fe: eb 06           jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00  mov $0x0,%eax
 400605: c3              retq
 400606: 5b              pop %rbx
 400607: c3              retq
```

# Stack Exercise

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

b) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)?

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07        movzbl (%rdi),%eax
 4005e9: 84 c0           test %al,%al
 4005eb: 74 13           je 400600 <rfun+0x1a>
 4005ed: 53              push %rbx
 4005ee: 48 0f be d8     movsbq %al,%rbx
 4005f2: 48 83 c7 01     add $0x1,%rdi
 4005f6: e8 eb ff ff ff  callq 4005e6 <rfun>
 4005fb: 48 01 d8        add %rbx,%rax
 4005fe: eb 06           jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00  mov $0x0,%eax
 400605: c3              retq
 400606: 5b              pop %rbx
 400607: c3              retq
```

# Stack Exercise

c) `char *s = "CSE351"`

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (l
    s++;
    return temp +
  }
  return 0;
}

// Main Function
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

c) Assume `main` calls `rfun` with `char *s = "CSE351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack?

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07        movzbl (%rdi),%eax
 4005e9: 84 c0           test %al,%al
                                  un+0x1a>
                                  0x
                                  <rfun>
 4005fe: eb 06           jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00  mov $0x0,%eax
 400605: c3              retq
 400606: 5b              pop %rbx
 400607: c3              retq
```

# Stack Exercise

c) char *s = “CSE351”

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (l
    s++;
    return temp +
  }
  return 0;
}

// Main Function – program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4) -> rfun(s+5) -> rfun(s+6) -> printf()

Total frames: 9

Max depth: 8

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07        movzbl (%rdi),%eax
 4005e9: 84 c0           test %al,%al
                                    un+0x1a>
                                    ox
                                    <rfun>
 4005fe: eb 06           jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00  mov $0x0,%eax
 400605: c3              retq
 400606: 5b              pop %rbx
 400607: c3              retq
```

# Stack Exercise

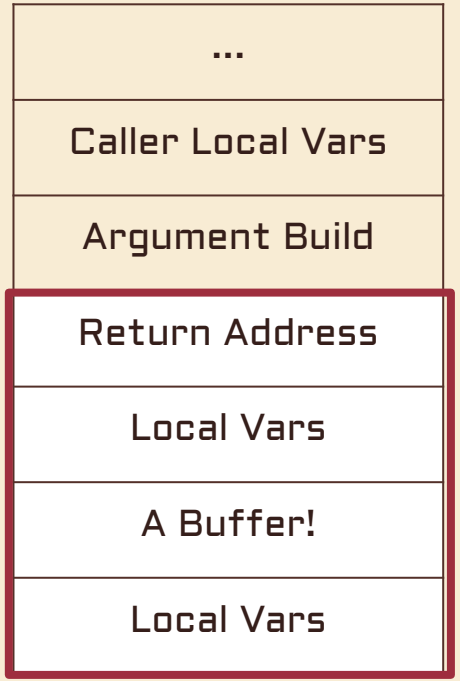
00000000004005e6 <rfun>:

| | |
|---|---|
| 4005e6: 0f b6 07 | movzbl(%rdi),%eax |
| 4005e9: 84 c0 | test %al,%al |
| 4005eb: 74 13 | je 400600 <rfun+0x1a> |
| 4005ed: 53 | push %rbx |
| 4005ee: 48 0f be d8 | movsbq %al,%rbx |
| 4005f2: 48 83 c7 01 | add $0x1,%rdi |
| 4005f6: e8 eb ff ff ff | callq 4005e6 <rfun> |
| 4005fb: 48 01 d8 | add %rbx,%rax |
| 4005fe: eb 06 | jmp 400606 <rfun+0x20> |
| 400600: b8 00 00 00 00 | mov $0x0,%eax |
| 400605: c3 | retq |
| 400606: 5b | pop %rbx |
| 400607: c3 | retq |

| Memory Address | Value | Description |
|---|---|---|
| 0x7fffffffdb48 | Unknown | %rsp when main is entered |
| 0x7fffffffdb38 | 0x400616 | Return address to main |
| 0x7fffffffdb30 | Unknown | Original %rbx |
| 0x7fffffffdb28 | 0x4005fb | Return address |
| 0x7fffffffdb20 | *s, “C”, 0x43 | Saved %rbx |
| 0x7fffffffdb18 | 0x4005fb | Return address |
| 0x7fffffffdb10 | *s, *(s+1), “S”, 0x53 | Saved %rbx |
| 0x7fffffffdb08 | 0x4005fb | Return address |
| 0x7fffffffdb00 | *s, *(s+2), “E”, 0x45 | Saved %rbx |

# Buffer Overflow

# Buffer Overflow Review

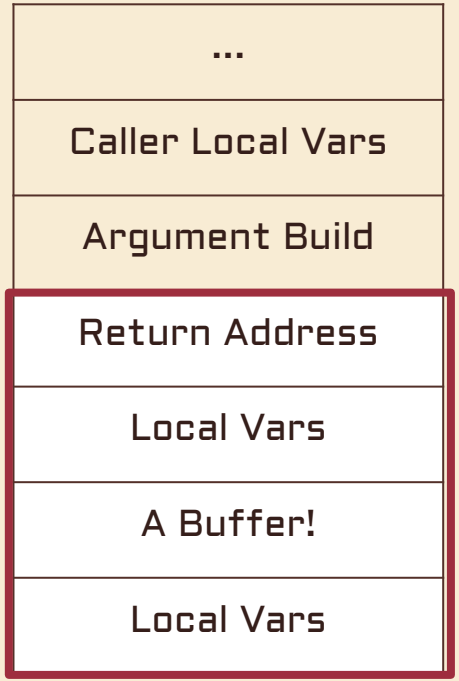



Simplified Memory Layout

Current Stack Frame

What can we overwrite and how might that affect the execution of our program?

# Buffer Overflow Review



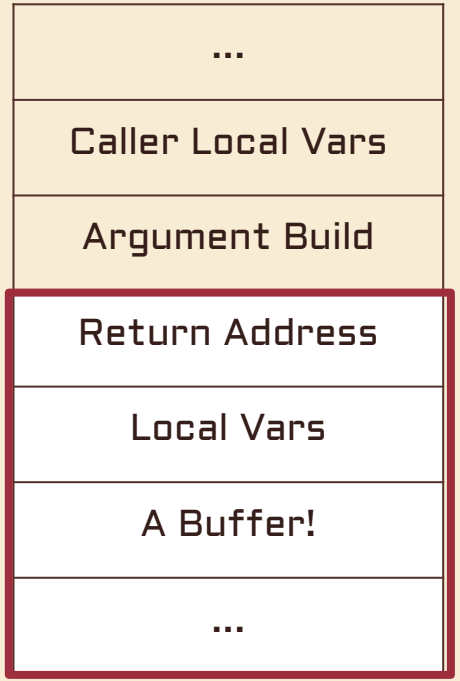

Simplified Memory Layout

Current Stack Frame

What can we overwrite and how might that affect the execution of our program?

How can we defend against it?

# Buffer Overflow Review




Simplified Memory Layout

Current Stack Frame

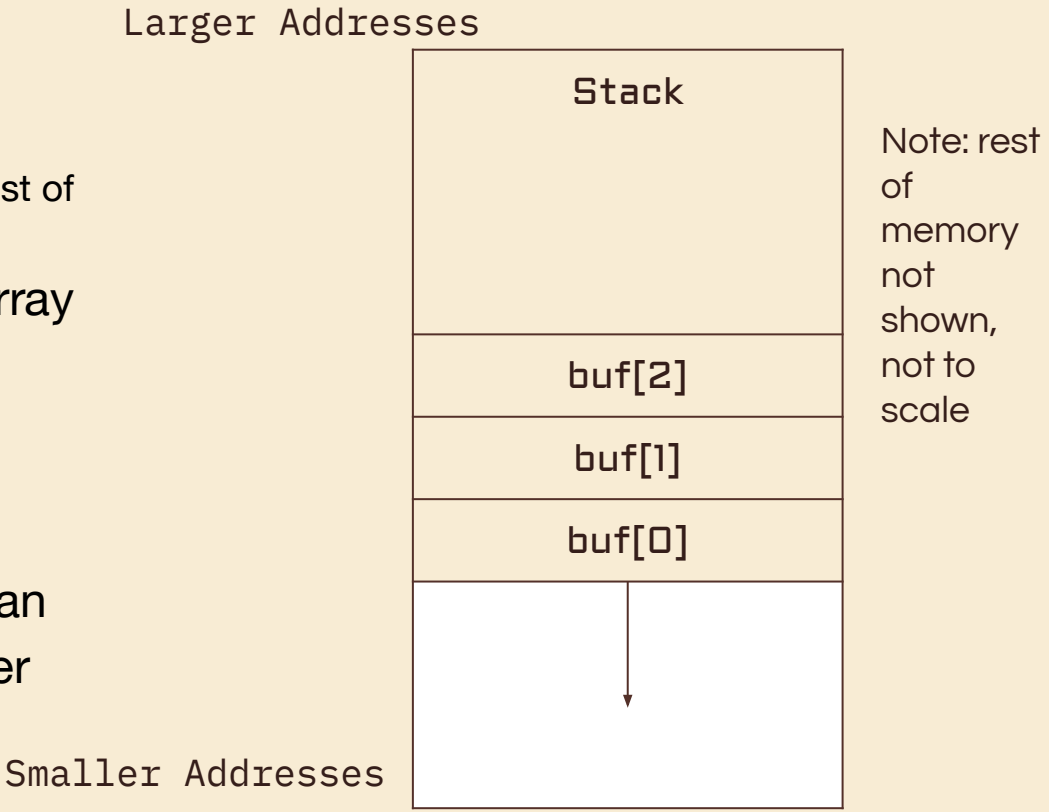
What can we overwrite and how might that affect the execution of our program?

How can we defend against it?

- Stack canaries
- Non-executable segments
- “Safe” functions or languages

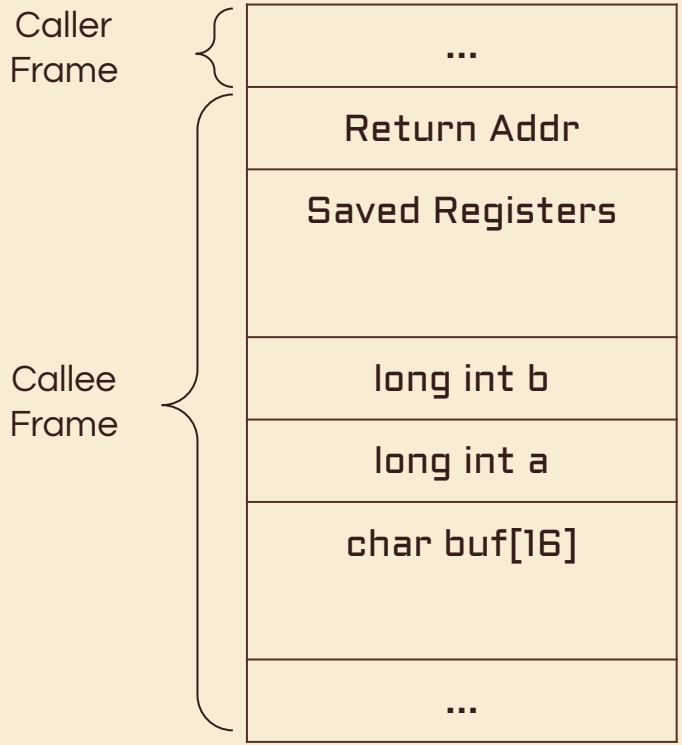
# Memory Addresses and Arrays

- Stack is “upside down” (grows down)
  - “Top” is at lower addresses than rest of stack
- Each additional element in an array is at a higher address than the previous element
  - Tip: Think about pointer arithmetic!
  - `buf[i] = *(buf + i)`
- Since each element is higher than the previous, writing to the buffer writes “up” toward the return address



# Stack Layout

- To which byte does buf[17] refer to in this example?
- In buffer overflow *attacks*, malicious users pass values to attempt to overwrite important parts of the stack or heap
- E.g. An attacker could overwrite the return instruction pointer with the address of a malicious block of code



# Exercise

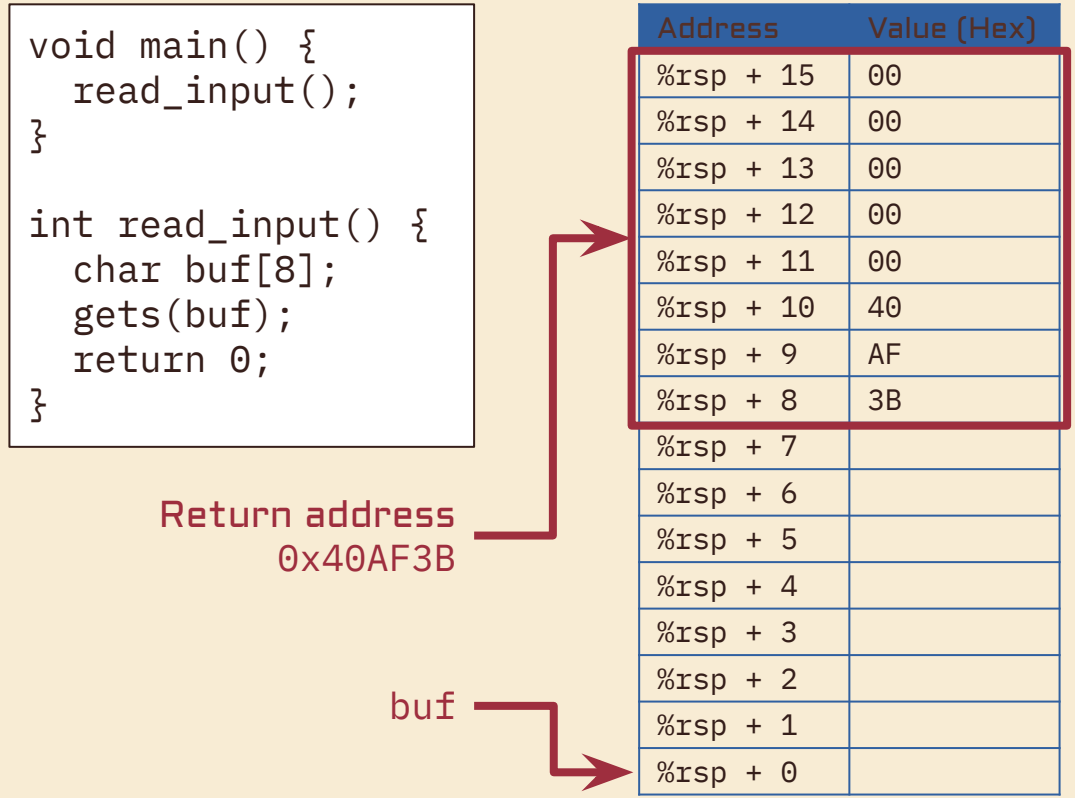
Where is the return address?
What is it?

What happens if we input the string “jklmnopqrs”?

Will the return address change?

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```



# Exercise

What happens when we try to return now?

What if we want to change the return address to 0x6A6B6C6D6E6F?

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Return address
0x7372

buf
“jklmnopqrs”

| Address | Value (Hex) |
|---|---|
| %rsp + 15 | 00 |
| %rsp + 14 | 00 |
| %rsp + 13 | 00 |
| %rsp + 12 | 00 |
| %rsp + 11 | 00 |
| %rsp + 10 | ~~40~~ 00 ‘\0’ |
| %rsp + 9 | ~~AF~~ 73 ‘s’ |
| %rsp + 8 | ~~3B~~ 72 ‘r’ |
| %rsp + 7 | 71 ‘q’ |
| %rsp + 6 | 70 ‘p’ |
| %rsp + 5 | 6F ‘o’ |
| %rsp + 4 | 6E ’n’ |
| %rsp + 3 | 6D ‘m’ |
| %rsp + 2 | 6C ‘l’ |
| %rsp + 1 | 6B ‘k’ |
| %rsp + 0 | 6A ‘j’ |

# Exercise

What about 0x7FFFFFAB1234?

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Return address
0x6A6B6C6D6E6F

buf
“aaaaaaaaonmlkj”

| Address | Value (Hex) |
|---|---|
| %rsp + 15 | 00 |
| %rsp + 14 | 00 ‘\0’ |
| %rsp + 13 | ~~00~~ 6A ‘j’ |
| %rsp + 12 | ~~00~~ 6B ‘k’ |
| %rsp + 11 | ~~00~~ 6C ‘l’ |
| %rsp + 10 | ~~40~~ 6D ‘m’ |
| %rsp + 9 | ~~AF~~ 6E ‘n’ |
| %rsp + 8 | ~~3B~~ 6F ‘o’ |
| %rsp + 7 | 61 ‘a’ |
| %rsp + 6 | 61 ‘a’ |
| %rsp + 5 | 61 ‘a’ |
| %rsp + 4 | 61 ‘a’ |
| %rsp + 3 | 61 ‘a’ |
| %rsp + 2 | 61 ‘a’ |
| %rsp + 1 | 61 ‘a’ |
| %rsp + 0 | 61 ‘a’ |

# Lab 3

# *What is* `sendstring`*?*

- Converts hexadecimal ASCII into the bytes they represent.

- Suppose we create file.txt containing the text “deadbeef”
  - `sendstring < file.txt > file.bytes` takes file.txt as input and writes file.bytes as output
  - If we look at the literal bytes in file.bytes using `hexdump`, we’ll see:

    ```
    $ hexdump file.bytes
    0000000 adde efbe 000a
    0000005
    ```

  - This is exactly the bytes of what we typed in to the original file.txt (in little endian, and with a newline “0x0a” at the end)

# Smoke Demo

# *Questions?*