CSE 351 Section 5

Download the Handout!

https://courses.cs.washington.edu/courses/cse351/20su/sections/05/cse351_sec5.pdf

Solutions will be posted this evening.

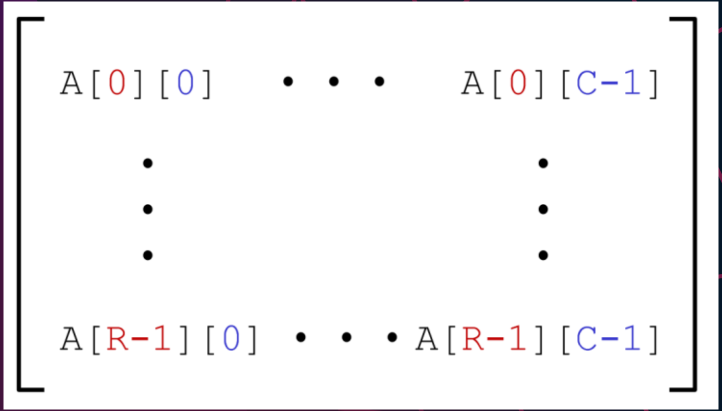
Arrays

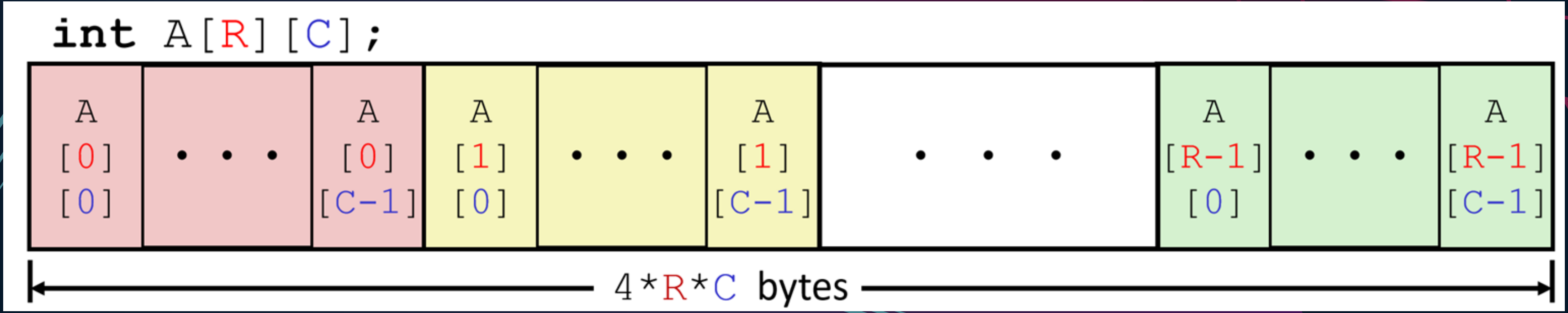
One Dimensional Arrays

- `T A[N];`
  - Declares an array of type T and length N
  - Contiguously stored in `N*sizeof(T)` bytes
    - Note that separately declared arrays are not necessarily contiguous
  - The symbol A returns a `T*`
  - `A[3]` is shorthand for `*(A+3)`
    - Pointer arithmetic and dereference
- The length of the array is not stored...
  - This means no bounds checking like Java!
- Arrays are like pointers, but they're not exactly the same...

Multidimensional Arrays

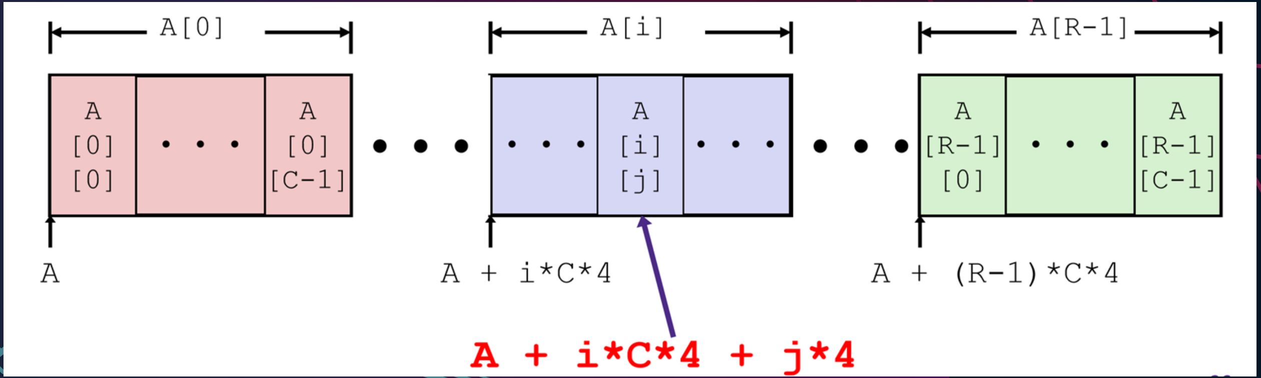
- `T A[R][C];`
  - 2D array of type T with R rows and C columns
  - Row-major
  - Contiguously
- Can extend to more dimensions
  - Add more brackets: `T B[X][Y][Z];`





Multidimensional Arrays

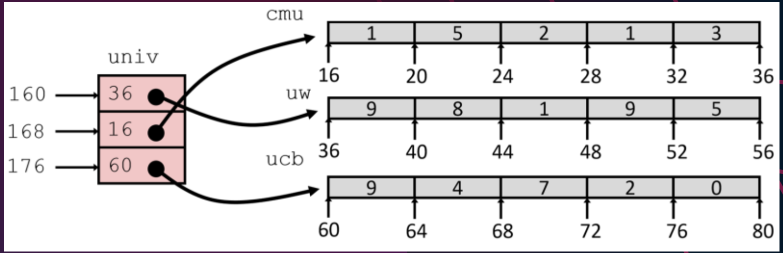
- `T A[R][C];`
  - A still returns a pointer to the array
- `A[i]` gets a *pointer* to a row of the array
  - The same as `A + i * (C * sizeof(T))`
- `A[i][j]` gets an *element* of the array
  - The same as `*(A[i] + j * sizeof(T))`



Multilevel Arrays

- `T* A[C];`
    - Array of pointers to 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!

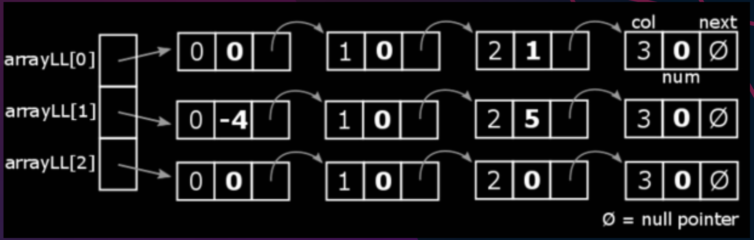
```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

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



Array Exercise (a)

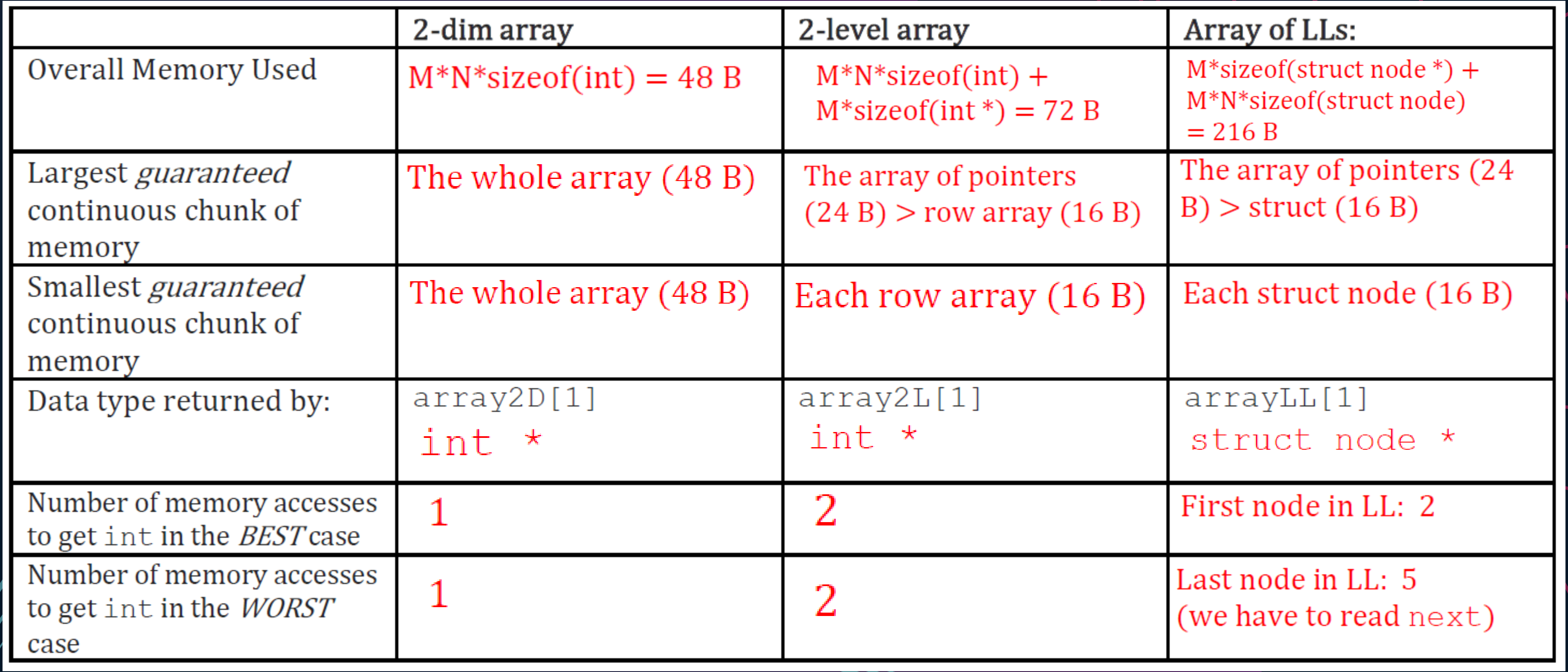
Consider an array with 3 rows and 4 columns.

- 2-dimensional
  - `int array2D[3][4];`
- 2-level
  - `int r0[4], r1[4], r2[4];`
  - `int* array2D[] = {r0, r1, r2};`
- Array of linked lists
  - `struct node { int col, num; struct node* next; };`
  - `struct node* arrayLL[3];`
  - …
  - NOTE: sizeof(struct node) is 16 bytes



Array Exercise (a)

Remember!
M=3, N=4



	2-dim array	2-level array	Array of LLs:
Overall Memory Used	M*N*sizeof(int) = 48 B	M*N*sizeof(int) + M*sizeof(int *) = 72 B	M*sizeof(struct node *) + M*N*sizeof(struct node) = 216 B
Largest *guaranteed* continuous chunk of memory	The whole array (48 B)	The array of pointers (24 B) > row array (16 B)	The array of pointers (24 B) > struct (16 B)
Smallest *guaranteed* continuous chunk of memory	The whole array (48 B)	Each row array (16 B)	Each struct node (16 B)
Data type returned by:	`array2D[1]` `int *`	`array2L[1]` `int *`	`arrayLL[1]` `struct node *`
Number of memory accesses to get `int` in the *BEST* case	1	2	First node in LL: 2
Number of memory accesses to get `int` in the *WORST* case	1	2	Last node in LL: 5 (we have to read `next`)

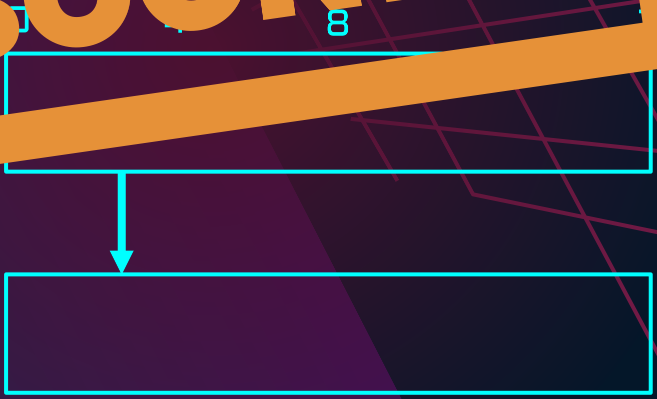
Array Exercise (b)

Sam Student claims that since our arrays are relatively small ($N < 256$), we can save space by storing the col field as a char in struct node. Is this corr ... , how much space do we save? If not, is this an ... internal or external fragmentation?

COMING SOON!

```
struct node {
  int co   nu  ;
  struct      e* next;
};
```

```
struct node {
  char col;
  int num;
  struct node* next;
};
```



Array Exercise (c)

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:

Matrix or table where all rows are the same size
This way we reduce memory usage and accesses

2-level:

Table or list where rows may vary in size or change frequently
This way we can more easily manipulate rows

Array Exercise (d)

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

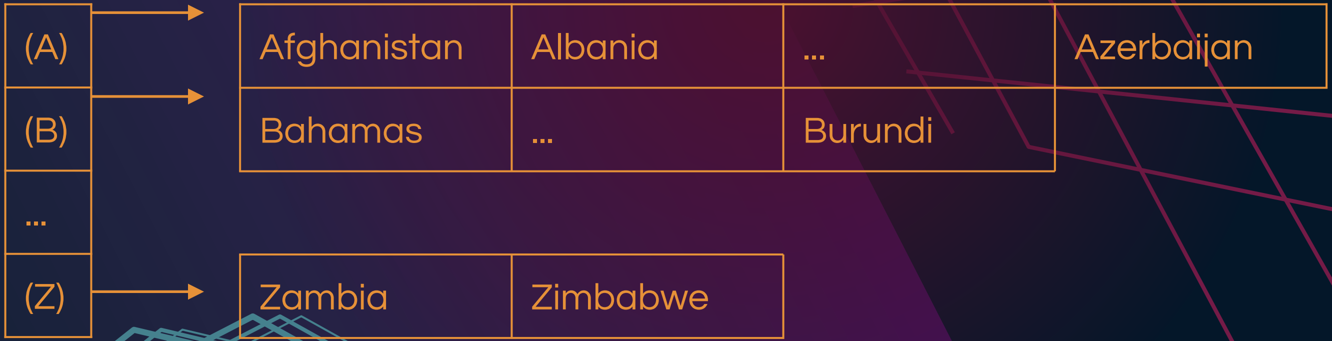
Multidimensional or multilevel?

Array Exercise (d)

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

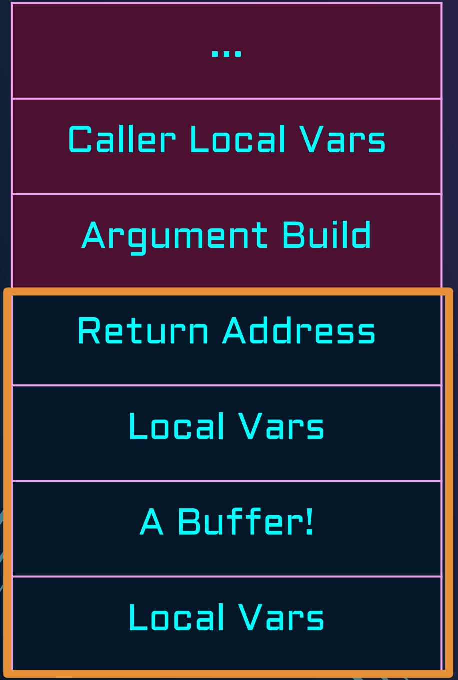
Multidimensional or ***multilevel***

We could have an array of pointers corresponding to letters, pointing to arrays of country names that start with that letter.



Buffer Overflow

Buffer Overflow Review

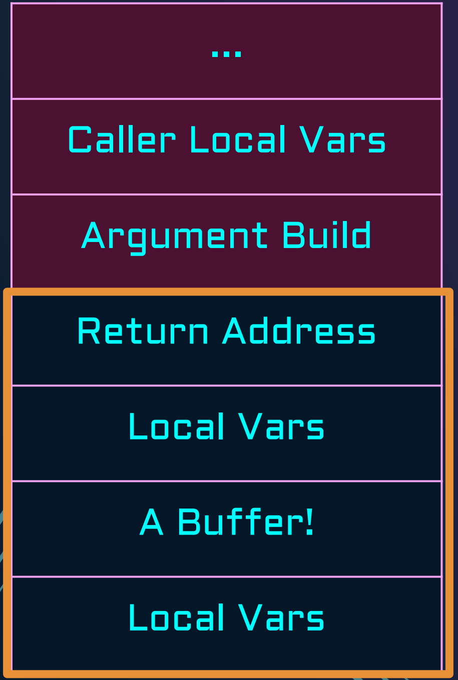


Simplified
Memory Layout

Current Stack
Frame

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

Buffer Overflow Review



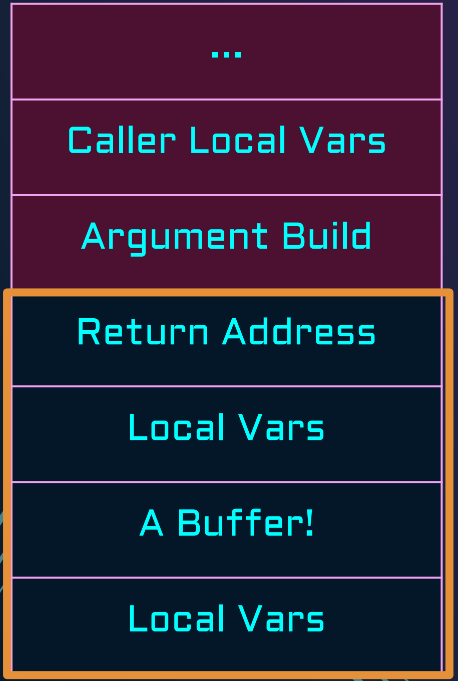
Simplified Memory Layout

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

Exercise (a)

Where is the return address?

What is it?

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

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

Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

Return address
0x40AF3B

buf

Exercise (b) - (d)

What happens if we input the string "jklmnopqrs"?

Will the return address change?

What happens when we try to return now?

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

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

Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

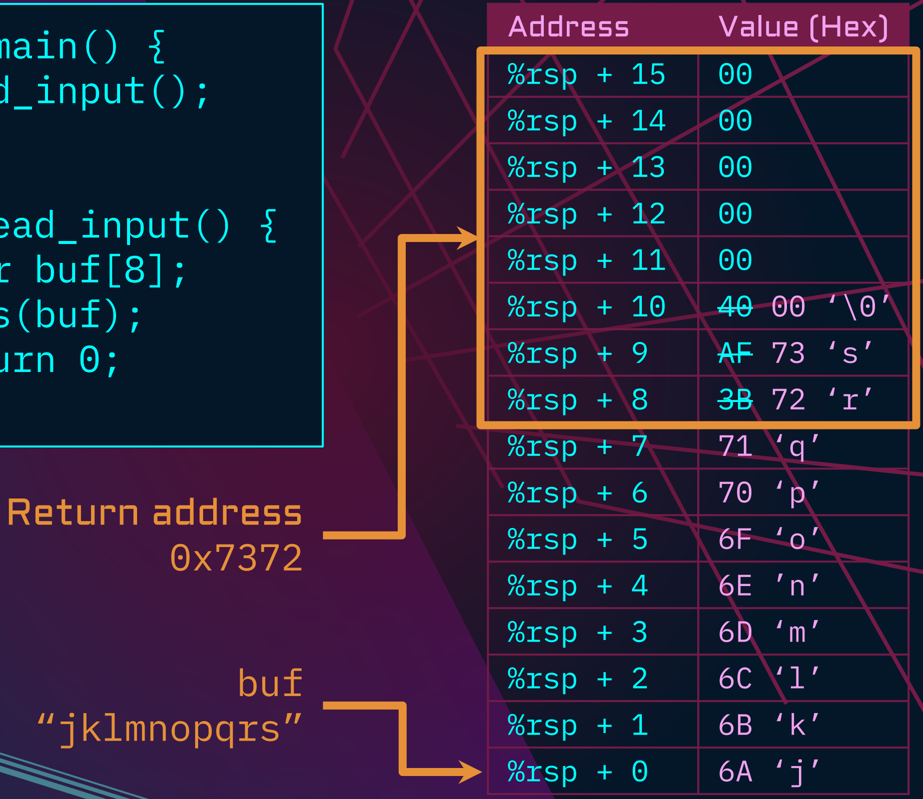
Exercise (b) - (d)

What happens if we input the string "jklmnopqrs"?

Will the return address change?

What happens when we try to return now?

```
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 (e) + (f)

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

How long should our string be?

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

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

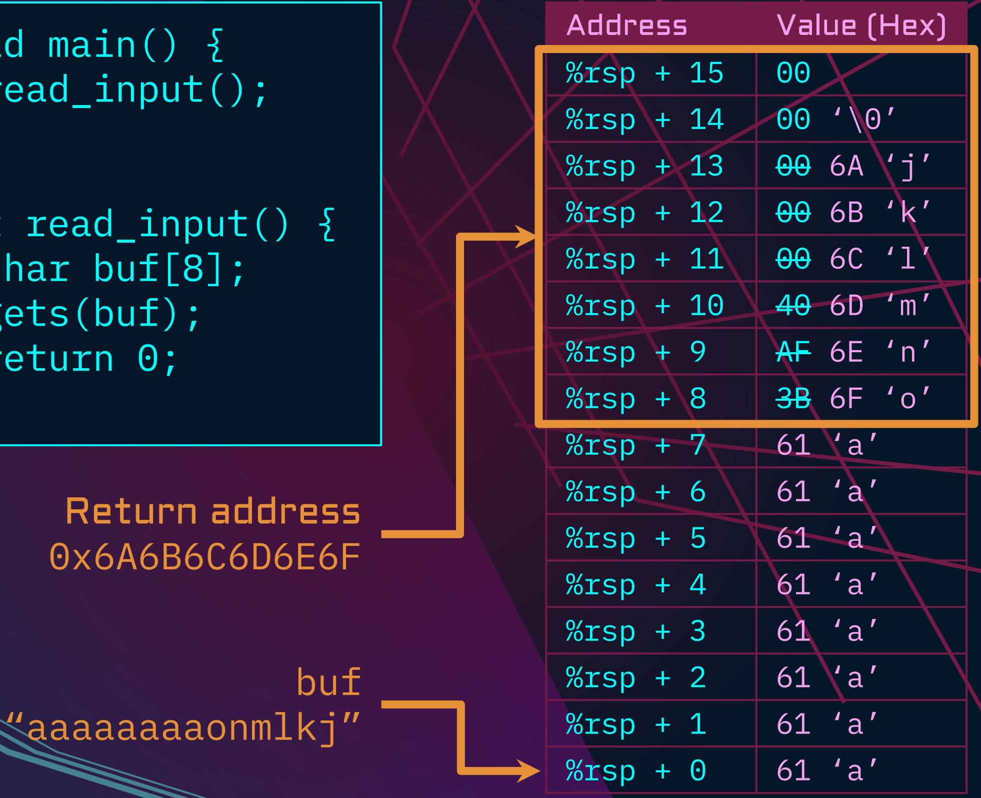
Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

Exercise (e) + (f)

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

How long should our string be?

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

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



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'

Exercise (g)

What about 0x7FFFFFAB1234?

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

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

Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

Exercise (g)

What about `0x7FFFFFAB1234`?

Impossible with traditional ASCII, which is a 7-bit code!

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

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

Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

Lab 3

Handout!

https://courses.cs.washington.edu/courses/cse351/20su/sections/05/cse351_lab3.pdf

This walks you through the first phase of lab 3. Take a look so you can follow along or do it yourself later!

Make sure you're familiar with the `sendstring` utility: how we use it, and why we need it.

Lab 3 Demo