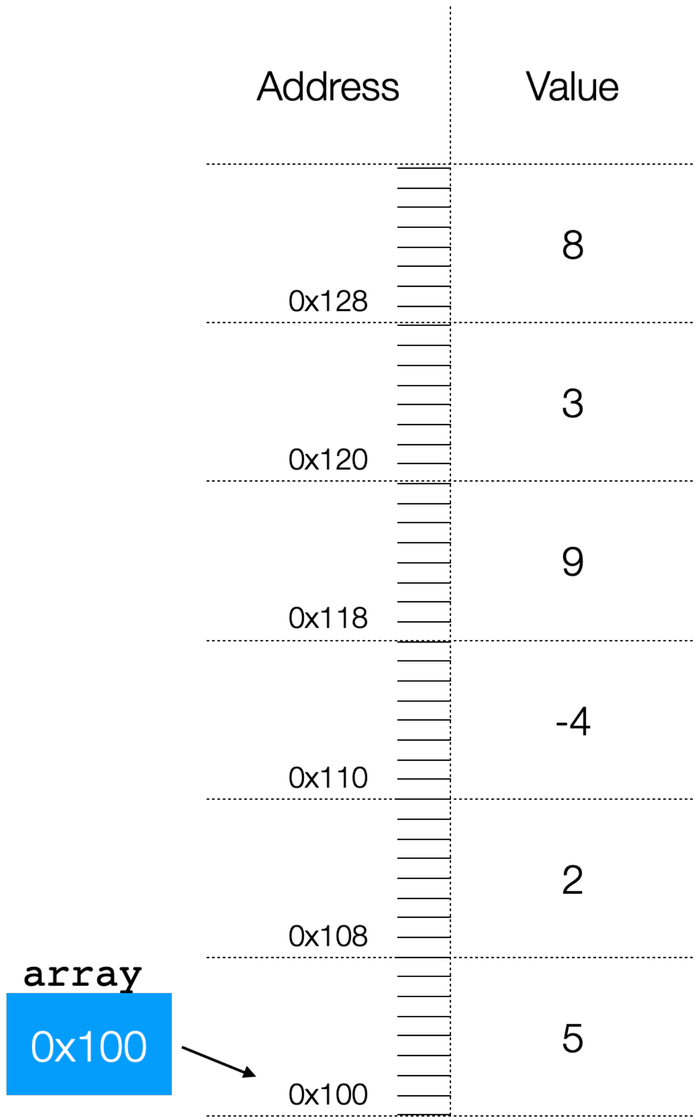
CS 107
Lecture 9: Arrays and Pointers in C

Friday, January 29, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Reader: Ch 4, *C Primer,* K&R Ch 1.6, 5.1-5.5

Lecturer: Chris Gregg



Today's Topics

- Logistics
  - Assign2 due Wednesday
  - Midterm coming up next week — we will have review materials out this week
- Reading: Reader: *C Primer*
  - Pointers and memcpy / memmove
  - Arrays of strings

Pointers to Arrays — Memory Footprint

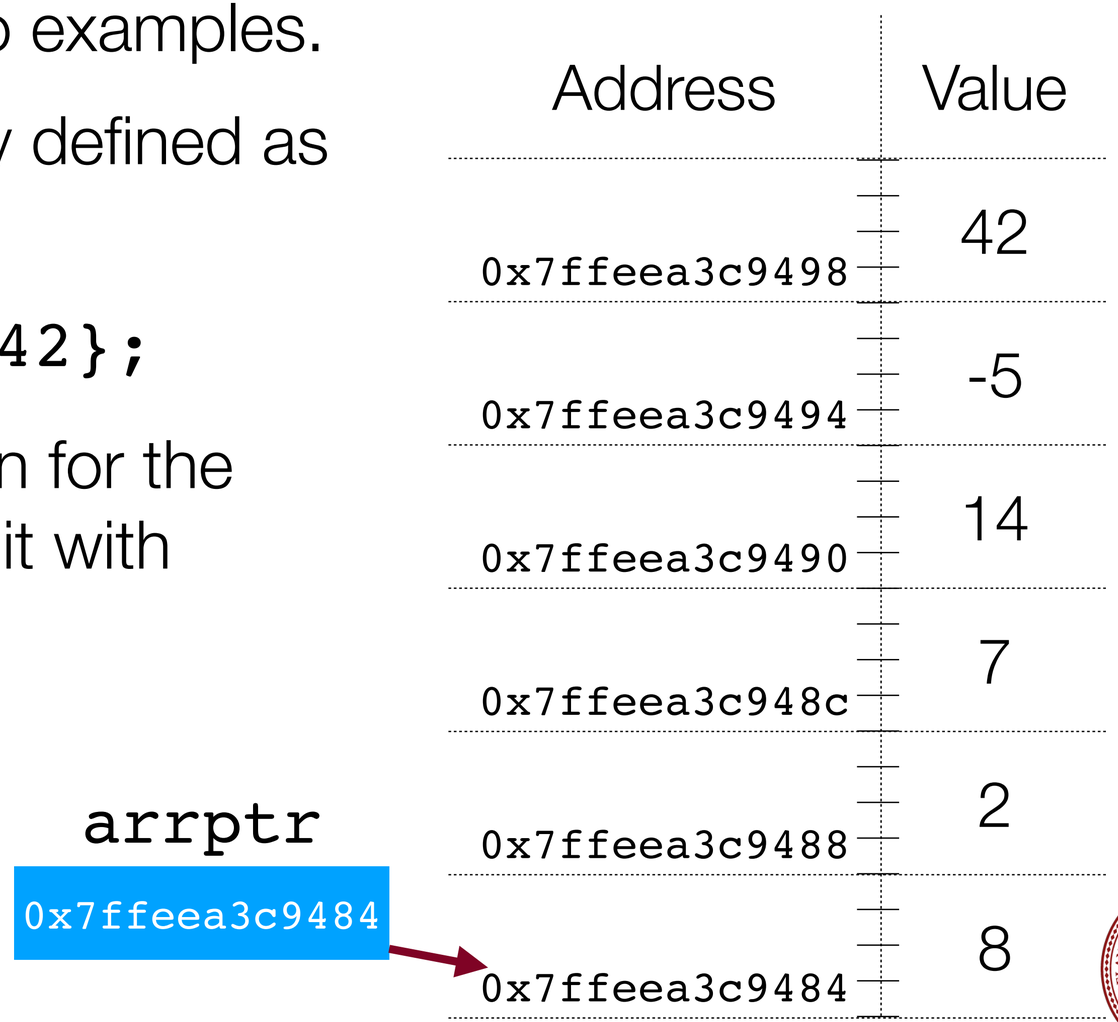
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

For the first example, let's look at the array defined as follows:

```
int arr[] = {8, 2, 7, 14, -5, 42};
```

What if we wanted to write a swap function for the array (to swap two elements)? We can do it with regular `int` variables:

```
void swapA(int *arr, int index_x, int index_y)
{
    int tmp = *(arr + index_x);
    *(arr + index_x) = *(arr + index_y);
    *(arr + index_y) = tmp;
}
```



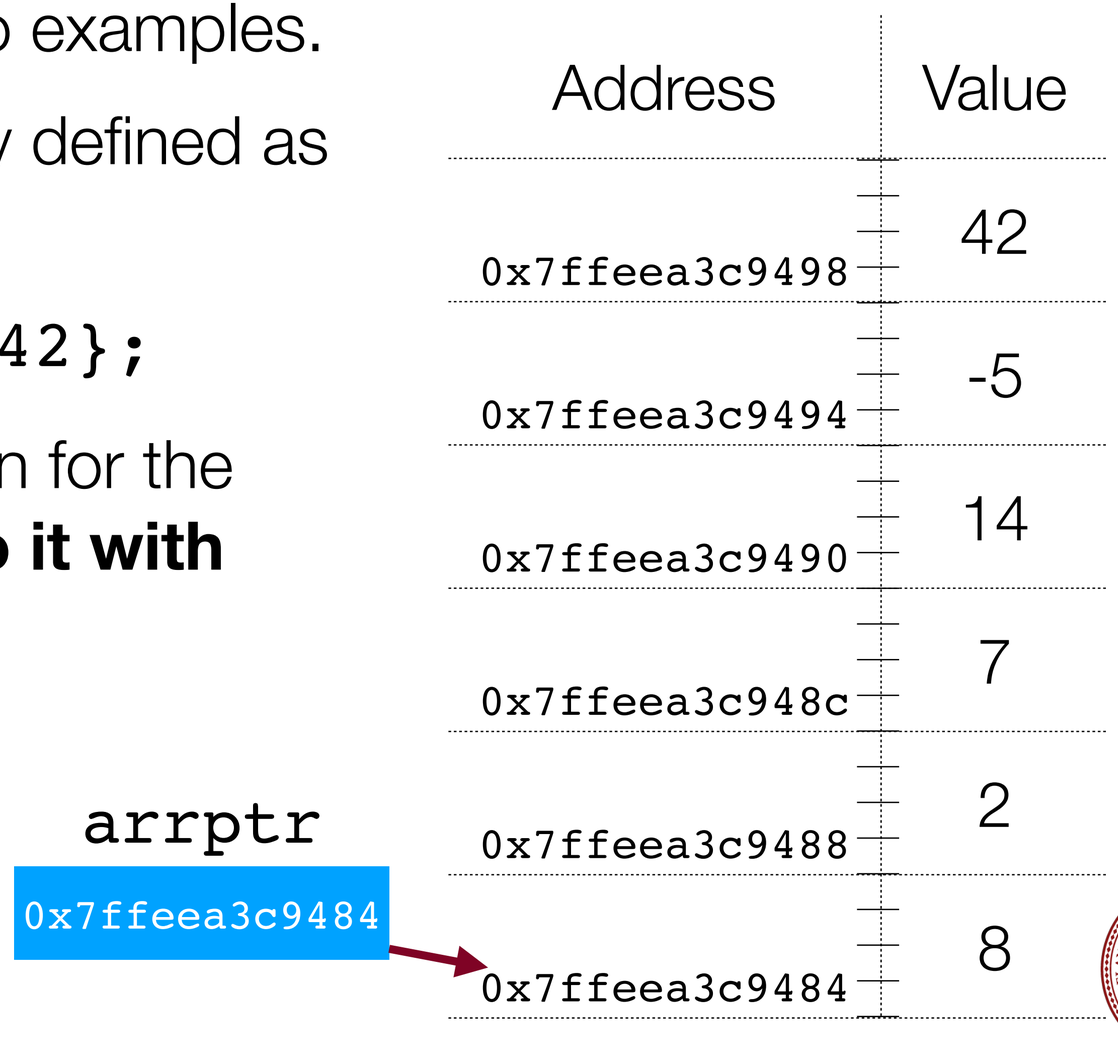
Pointers to Arrays — Memory Footprint

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

For the first example, let's look at the array defined as follows:

```
int arr[] = {8, 2, 7, 14, -5, 42};
```

What if we wanted to write a swap function for the array (to swap two elements)? **Can we do it with memmove?**



Pointers to Arrays — Memory Footprint

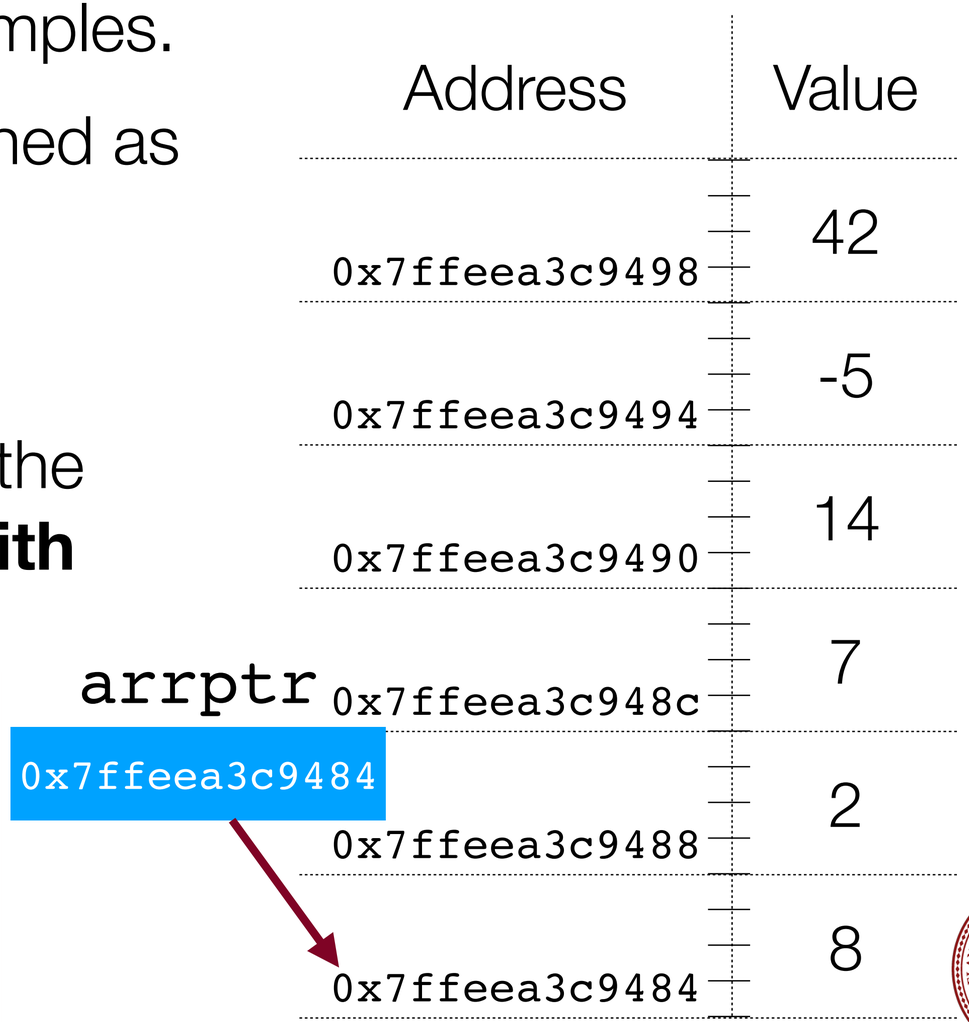
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

For the first example, let's look at the array defined as follows:

```
int arr[] = {8, 2, 7, 14, -5, 42};
```

What if we wanted to write a swap function for the array (to swap two elements)? **Can we do it with memmove? Sure:**

```
void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```



Pointers to Arrays — Memory Footprint

What if we wanted to write a swap function for the array (to swap two elements)? **Can we do it with memove? Sure:**

```
void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```

This works because we *know the size of the elements in the array (they are `int`s)*

As long as we know the size of the elements, we can always swap (or compare, or whatever) two elements in an array!

Pointers to Arrays — Memory Footprint

Full example:

```
// file: pointer_to_array1.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swapA(int *arr, int index_x, int index_y)
{
   int tmp = *(arr + index_x);
   *(arr + index_x) = *(arr + index_y);
   *(arr + index_y) = tmp;
}

void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```

```
int main(int argc, char **argv)
{
    int arr[] = {8, 2, 7, 14, -5, 42};
    swapA(arr, 0, 5); // swaps 8 and 42
    swapB(arr, 1, 2); // swaps 2 and 7
    int nelems = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < nelems; i++) {
        printf("%d", arr[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

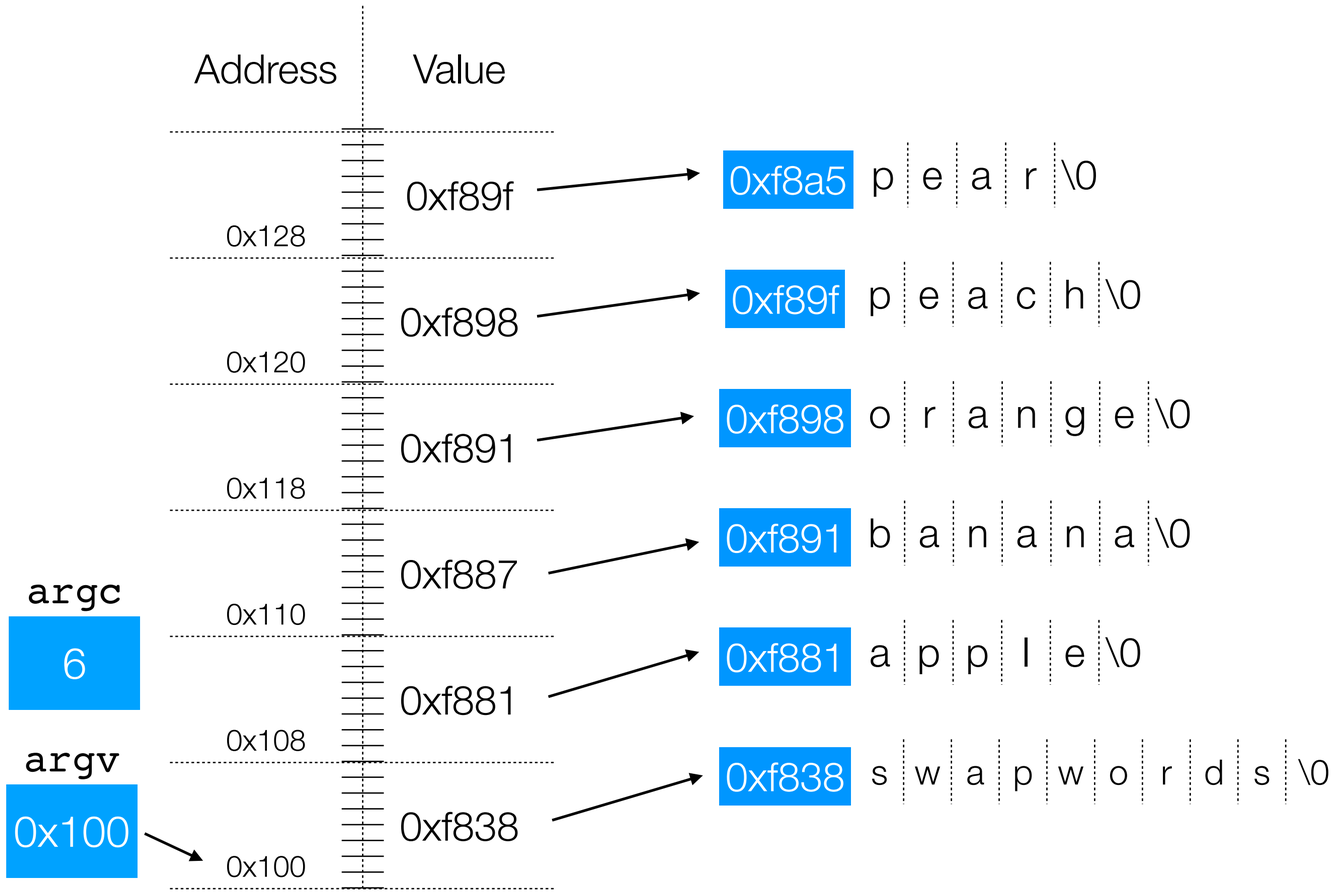
```
$ ./pointer_to_array1
42, 7, 2, 14, -5, 8
```

Pointers to Arrays — Memory Footprint

For our second example, let's look at **`argv`**, which is an array of **`char *`** pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array?



Pointers to Arrays — Memory Footprint

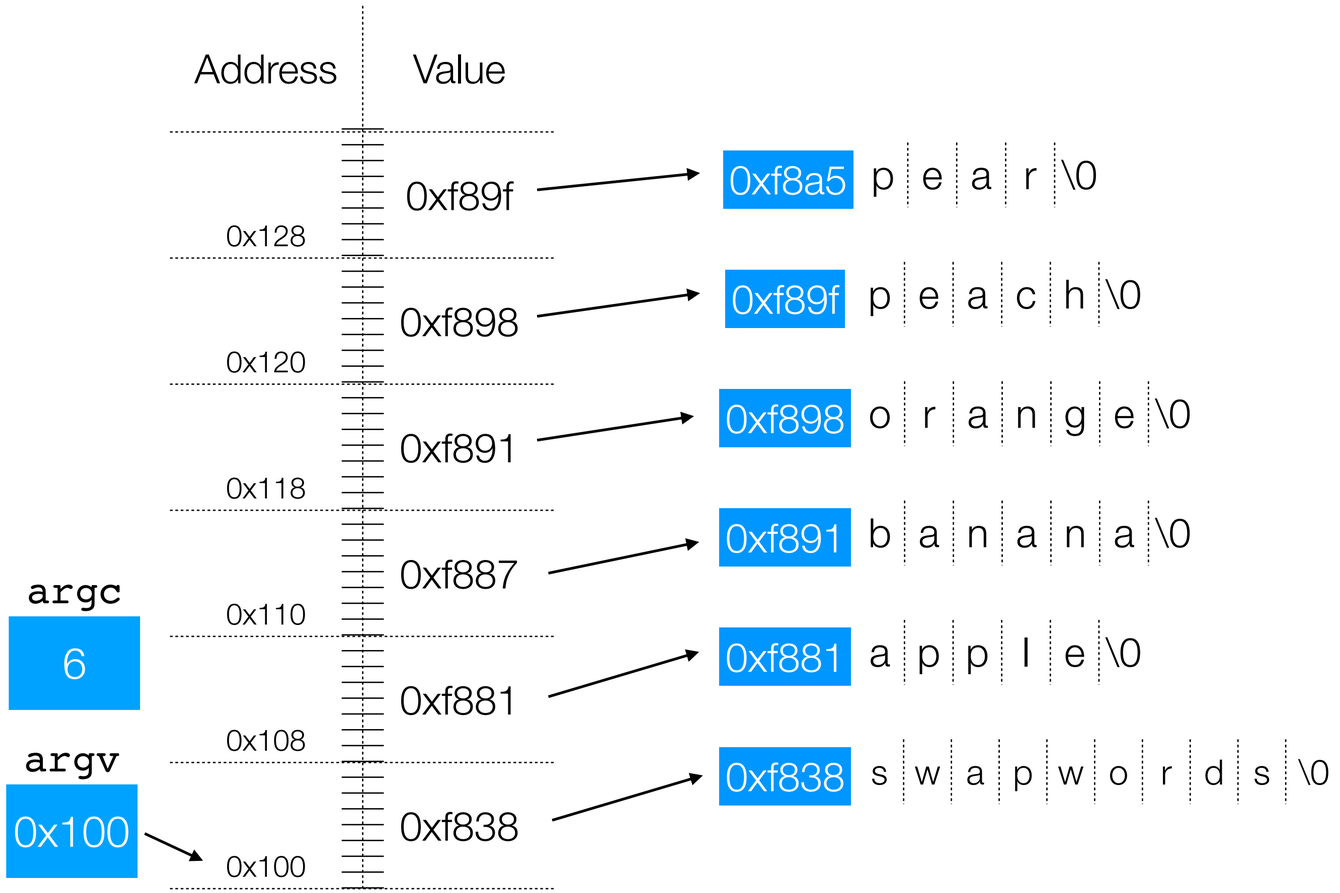
For our second example, let's look at **`argv`**, which is an array of **`char *`** pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array? Sure:

```
void swapA(char **arr, int index_x, int index_y)
{
   char *tmp = *(arr + index_x);
   *(arr + index_x) = *(arr + index_y);
   *(arr + index_y) = tmp;
}
```

Note (very important!) -- Only the pointers are getting swapped! We are *not* copying the text from each string, at all. For all we know, the strings might be any type!

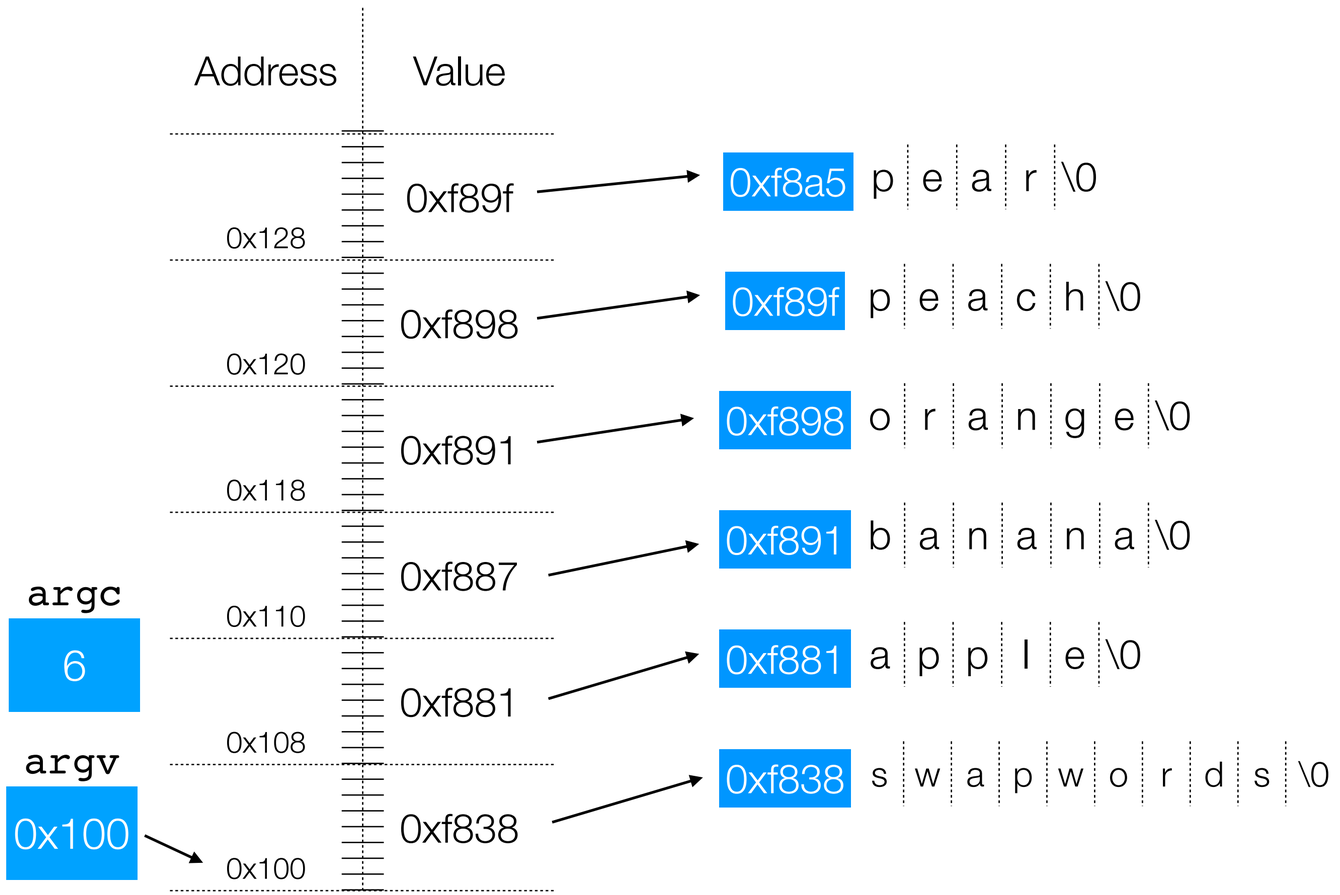


Pointers to Arrays — Memory Footprint

For our second example, let's look at **`argv`**, which is an array of **`char *`** pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array **using memove?**



Pointers to Arrays — Memory Footprint

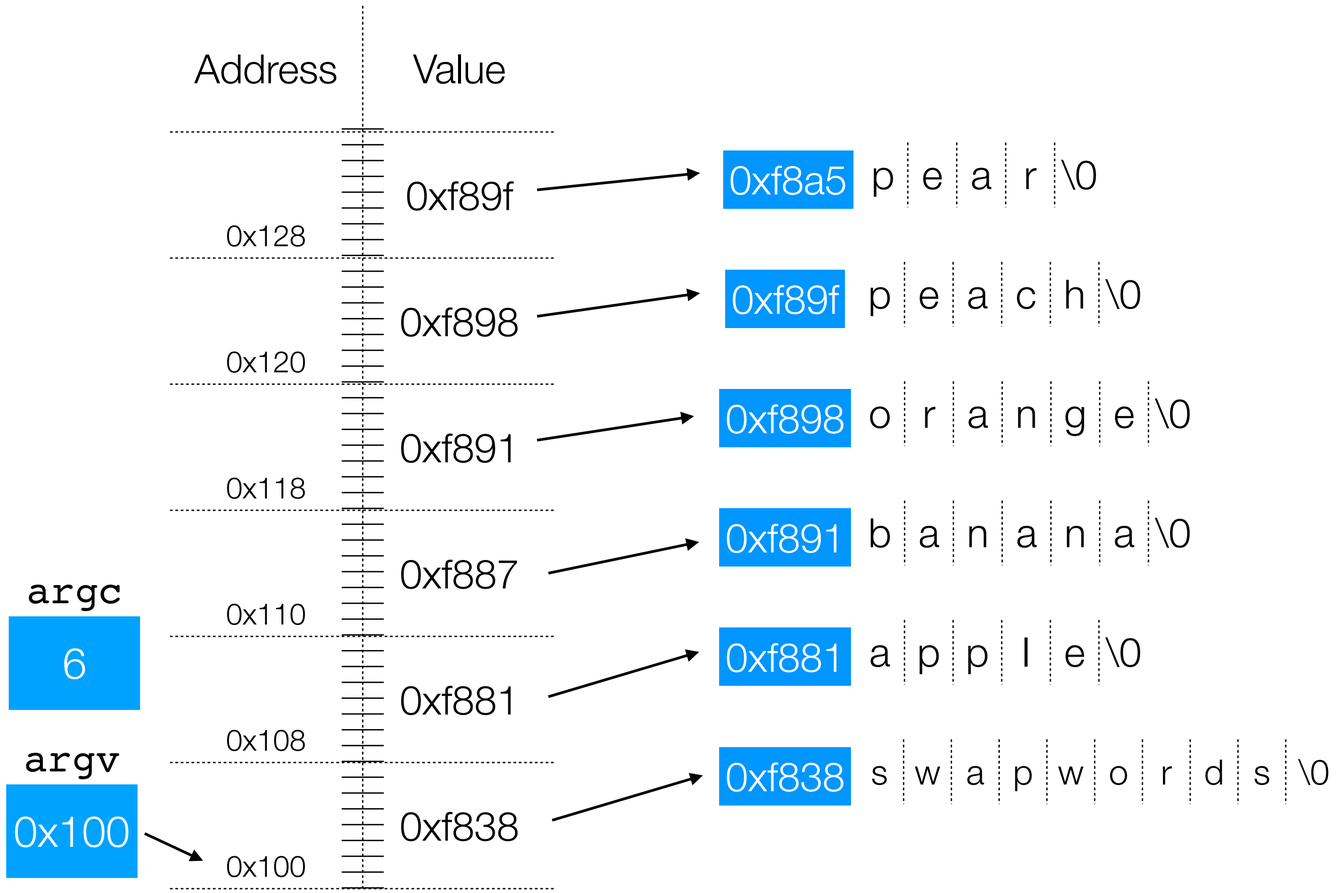
For our second example, let's look at **`argv`**, which is an array of **`char *`** pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array **using memove?**

```
void swapB(char **arr, int index_x, int index_y)
{
    char *tmp;
    memmove(&tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, &tmp, sizeof(char *));
}
```

In this case, we need to move 8 bytes at a time, and we conveniently get that value using **`sizeof()`**.



Pointers to Arrays — Memory Footprint

Full example:

```
// file: swapwords.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swapA(char **arr, int index_x, int index_y)
{
   char *tmp = *(arr + index_x);
   *(arr + index_x) = *(arr + index_y);
   *(arr + index_y) = tmp;
}

void swapB(char **arr, int index_x, int index_y)
{
    char *tmp;
    memmove(&tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, &tmp, sizeof(char *));
}
```

```
int main(int argc, char **argv)
{
    if (argc < 6) {
        printf("Usage:\n\t%s s1 s2 s3 s4 s5\n",argv[0]);
        return -1;
    }
    // assume:
    // ./swapwords apple banana orange peach pear
    swapA(argv, 1, 5); // swaps apple and pear
    swapB(argv, 2, 3); // swaps banana and orange
    for (int i = 1; i < argc; i++) { // skip progname
        printf("%s",argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./swapwords apple banana orange peach pear
pear, orange, banana, peach, apple
```

Pointers to Arrays — Memory Footprint

You might be asking -- why would we ever want to use the **`memmove`** function, if we already know the type?

Ah -- this is a key insight that we will discuss soon! When we get to "**`void *`**" pointers, we will find out that there is no way to do this without memmove, and we will actually need information about the width of the type itself!

Preview:

```
void swap_generic(void *arr, int index_x, int index_y, int width)
{
    char tmp[width];
    void *x_loc = (char *)arr + index_x * width;
    void *y_loc = (char *)arr + index_y * width;

    memmove(tmp, x_loc, width);
    memmove(x_loc, y_loc, width);
    memmove(y_loc, tmp, width);
}
```

Double pointers — why are they needed?

Let's take an in-depth look at the following example:

```
#include<stdio.h>
#include<stdlib.h>

// print the next character in p
// and update the local pointer, p (which does nothing)
char nextCharA(char *p) {
    char next = p[0];
    p++; // this does not do anything except inside this function
    // and, we are returning here, so it really doesn't
    // do anything productive
    return next;
}

// print the next character in the string pointed to by p
// and update the string pointer by one to go to the next character
char nextCharB(char **p) {
    char next = (*p)[0];
    (*p)++; // now the original pointer gets updated!
    // we return here, but the calling function has the
    // details it needs for the next call
    return next;
}
```

```
int main() {
    char *myString = "hello";
    char *pA = myString;
    char *pB = myString;

    for (int i = 0; i < 5; i++) {
        printf("nextCharA(pA): %c ", nextCharA(pA));
        printf("nextCharB(&pB): %c\n",
nextCharB(&pB));
    }
    return 0;
}
```

Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.

Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

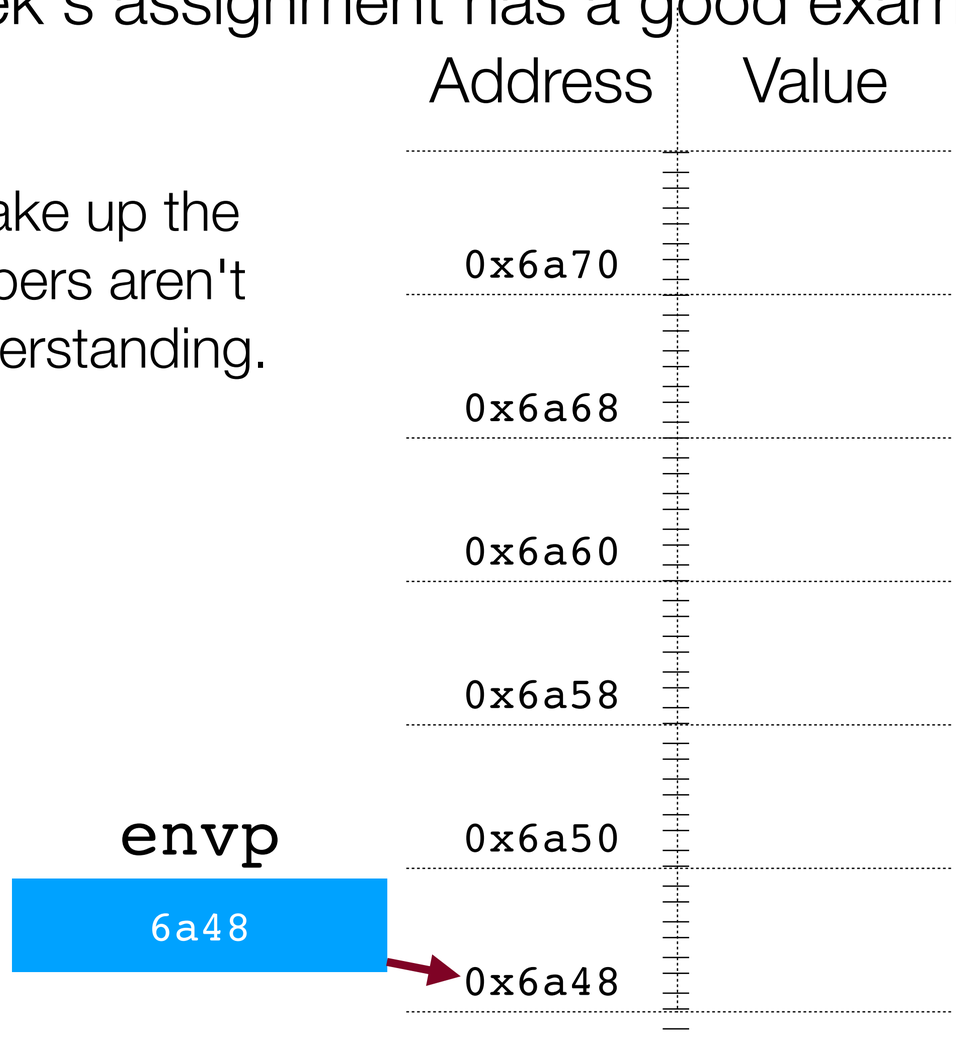
1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.

Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.

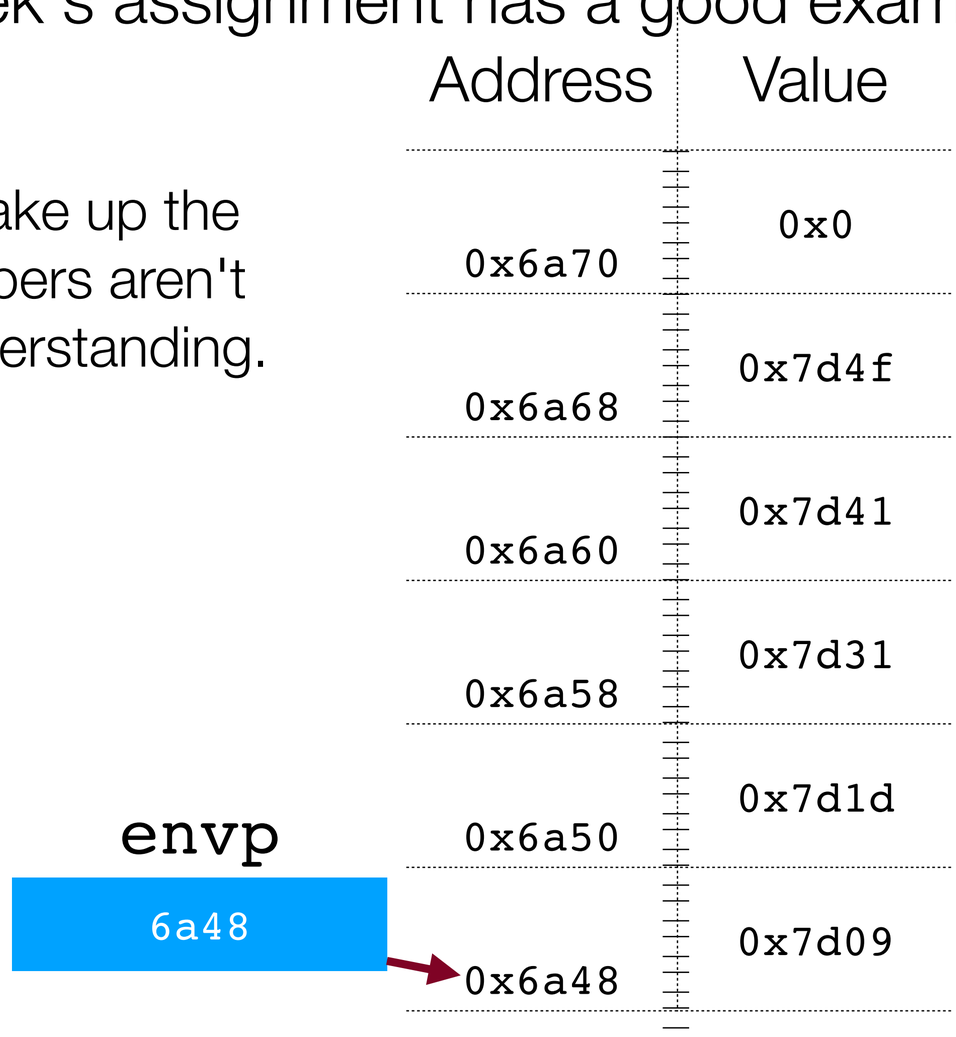


Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.

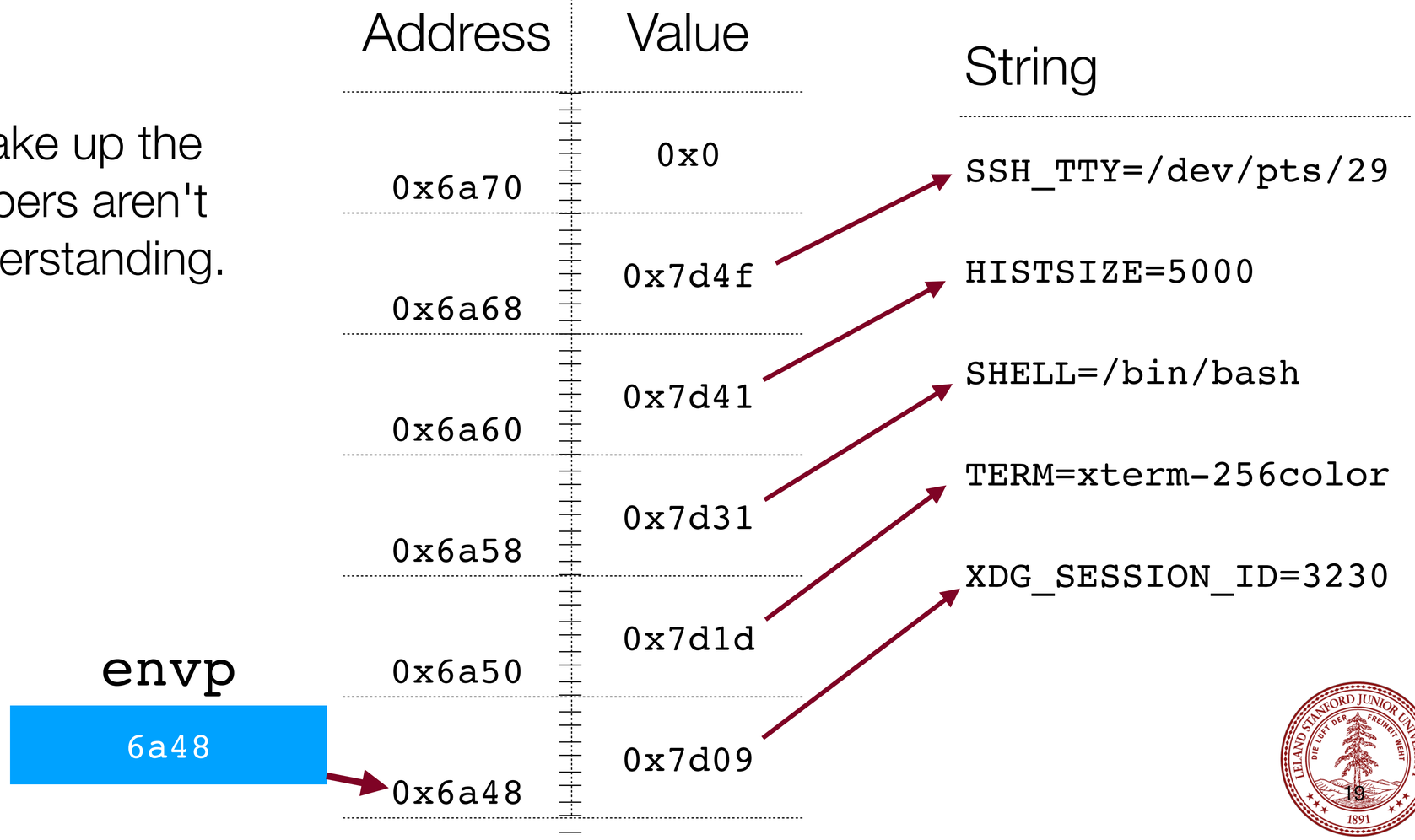


Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.



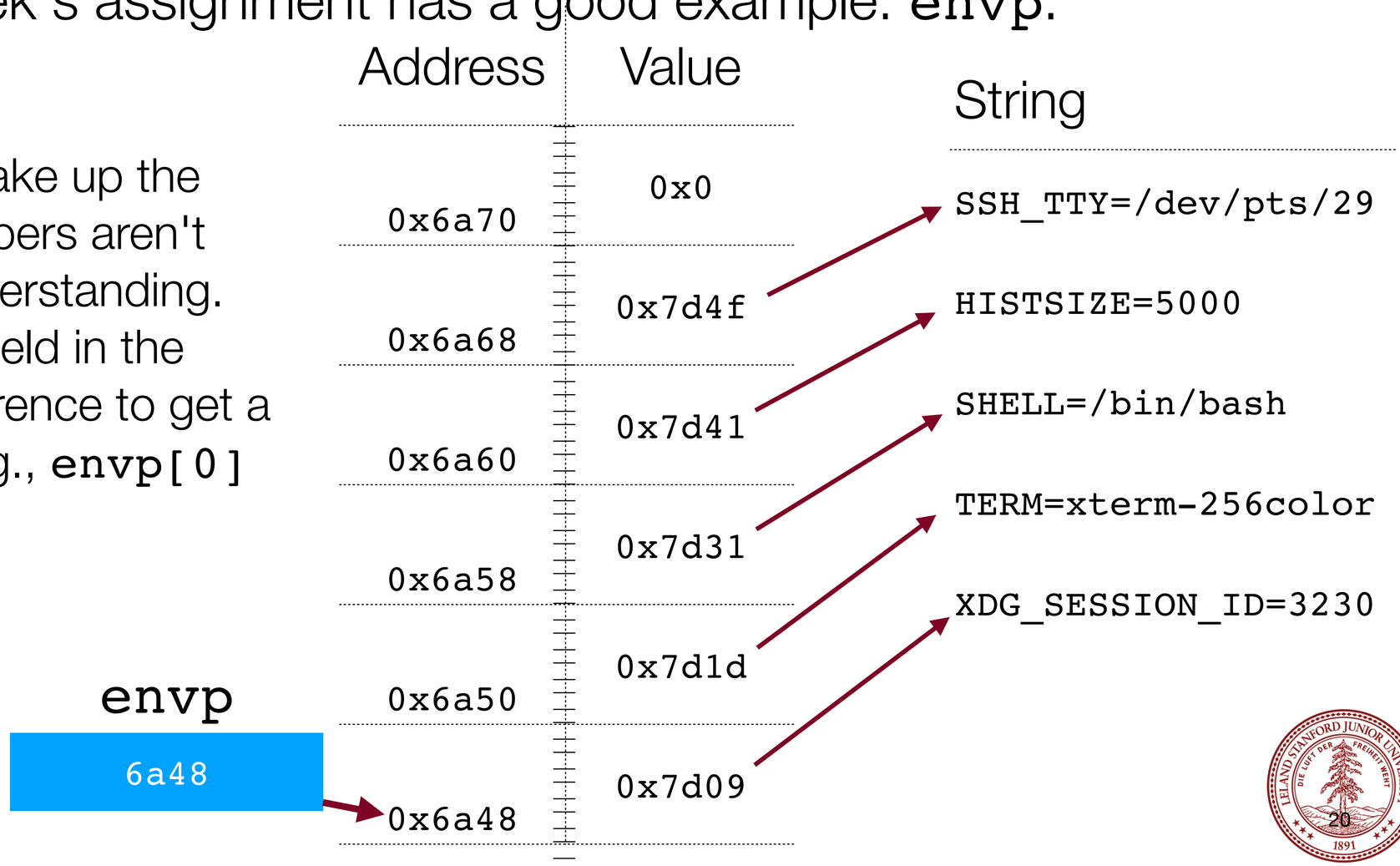
Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., **`envp[0]`** is a pointer to the string **`"XDG_SESSION_ID=3230"`**

What is *the value* of **`envp[2]`** for the diagram?



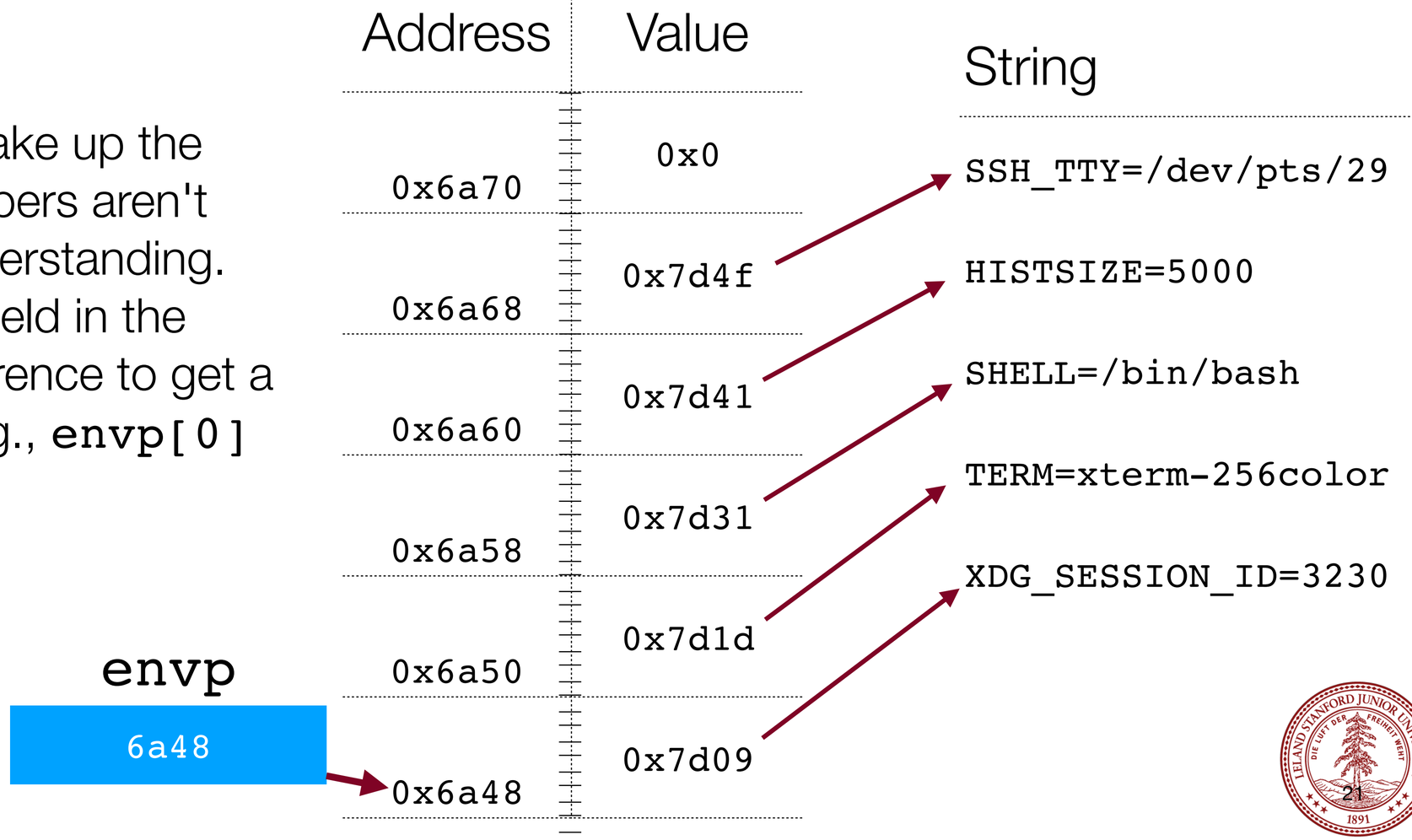
Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., **`envp[0]`** is a pointer to the string **`"XDG_SESSION_ID=3230"`**

What is *the value* of **`envp[2]`** for the diagram? **`0x7d31`**



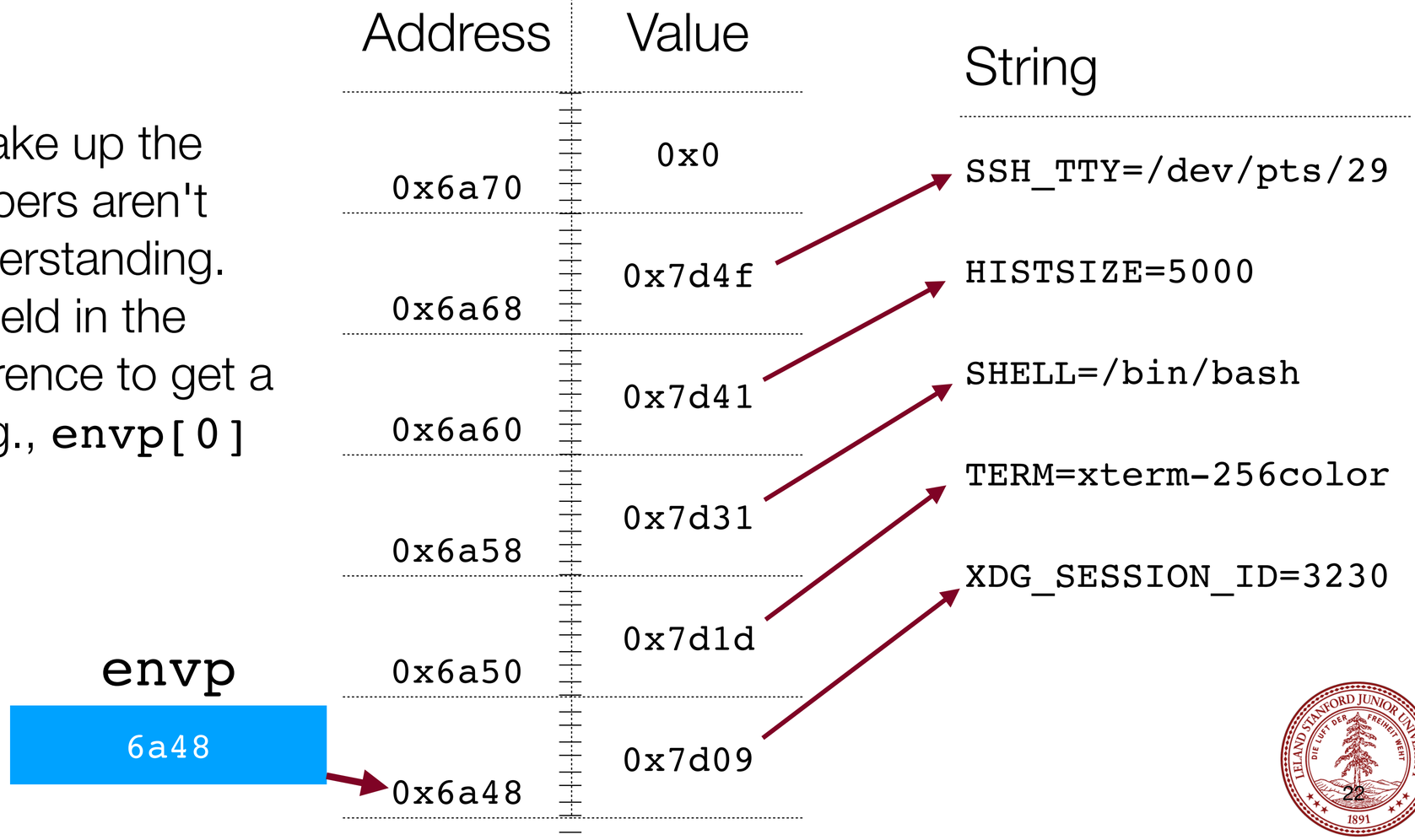
Pointers to Arrays — char *envp[]

One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., **`envp[0]`** is a pointer to the string **`"XDG_SESSION_ID=3230"`**

What type is **`envp[2]`**?



Pointers to Arrays — char *envp[]

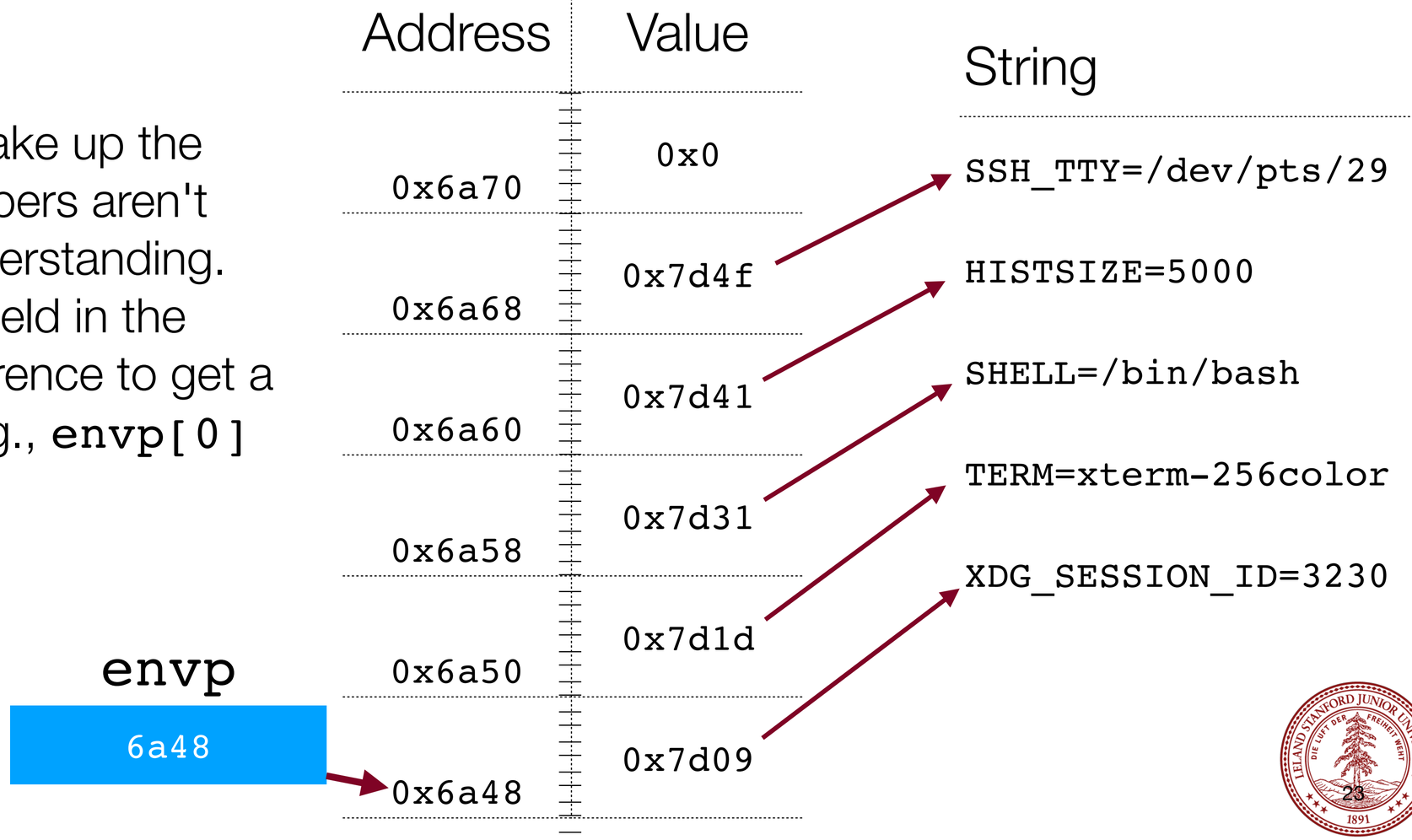
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. This week's assignment has a good example: **`envp`**.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., **`envp[0]`** is a pointer to the string **`"XDG_SESSION_ID=3230"`**

What type is **`envp[2]`**?

char *

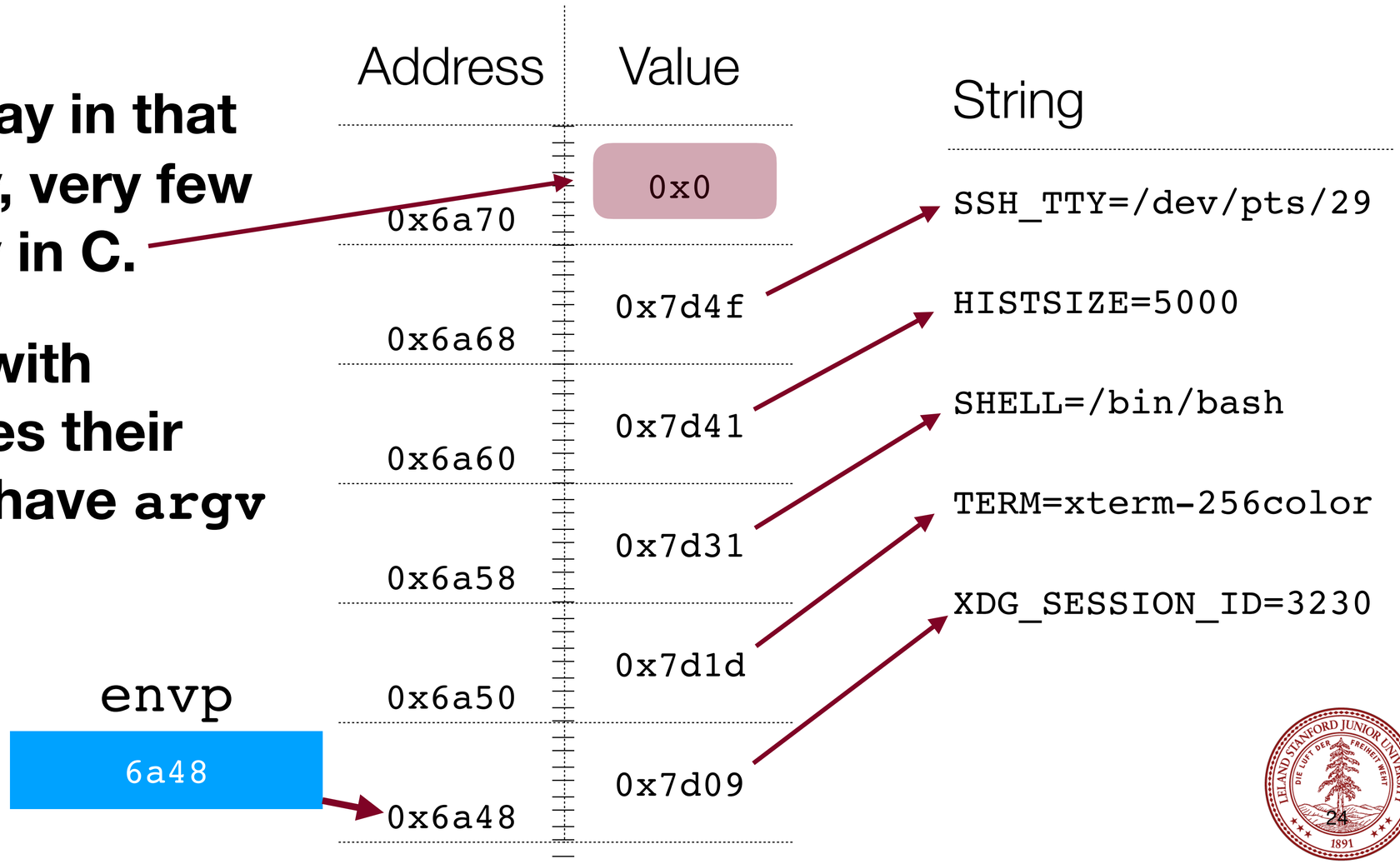


Pointers to Arrays — char *envp[]

Note: `envp` is a weird array in that it is null-terminated! Very, very few arrays have this property in C.

Most arrays are passed with another variable that gives their length. For example, we have `argv` and `argc`.*

*Note: `argv[argc]` *is* defined to be `NULL`, but that still an anomaly for C arrays in general.



References and Advanced Reading

- **References:**
  - K&R C Programming (from our course)
  - Course Reader, C Primer
  - Awesome C book: http://books.goalkicker.com/CBook

- **Advanced Reading:**
  - https://www.cs.bu.edu/teaching/cpp/string/array-vs-ptr/
  - https://en.wikibooks.org/wiki/C_Programming/Pointers_and_arrays