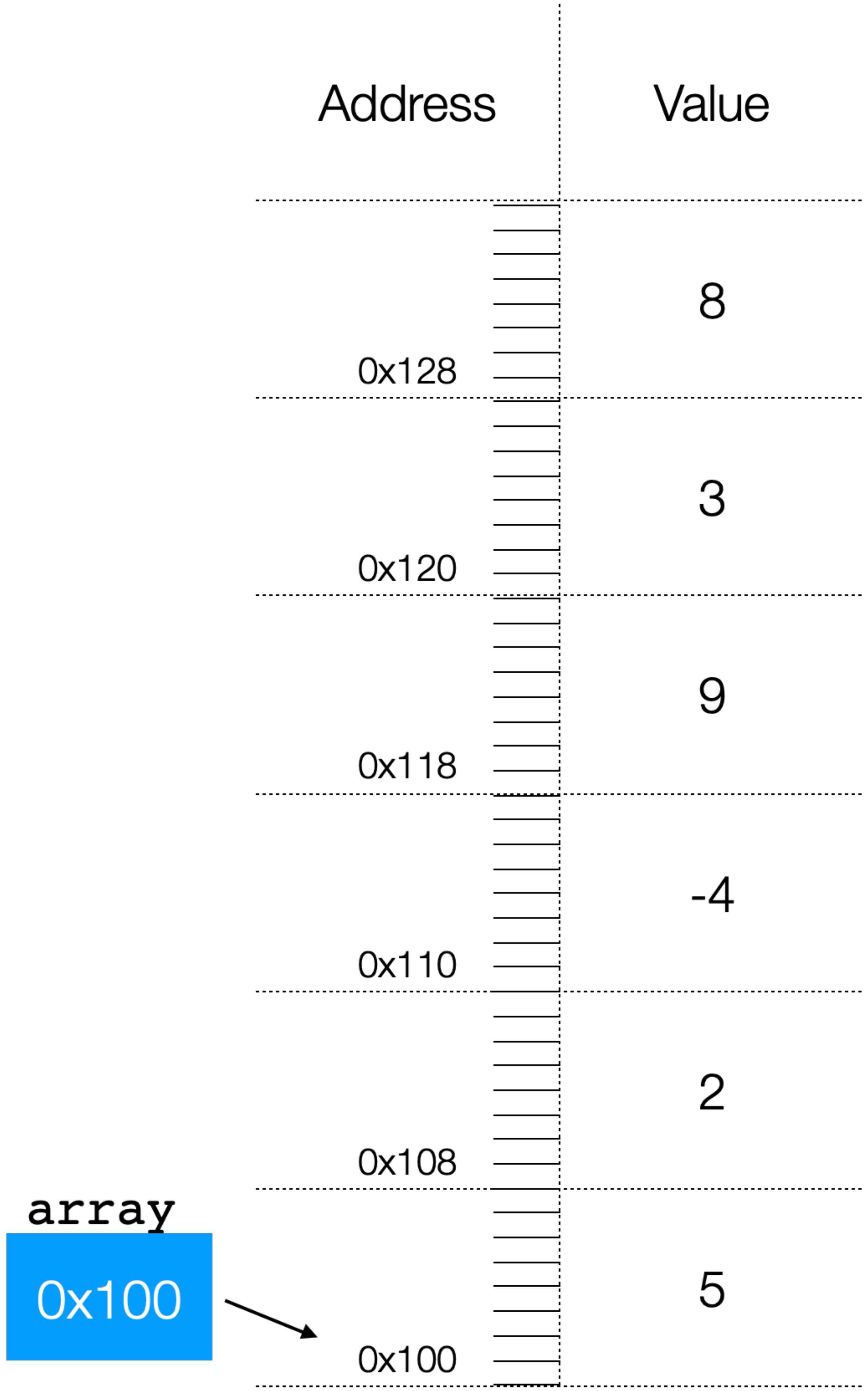
CS 107
Lecture 5: Arrays and Pointers in C

Friday, January 21, 2022

Computer Systems
Winter 2022
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 at 11:59pm
- Reading: Reader: *C Primer*
  - Pointers
  - Arrays in C
  - Pointers to arrays
  - Pointer arithmetic -vs- bracket notation
  - Pointers -vs- arrays
  - When is 1 + 1 ≠ 2 ?
  - Arrays passed by reference
  - memcpy / memmove
  - literal strings
  - dereferencing NULL, bogus address…

C Pointers

1. Review of `strspn` and `strstr`. Also: can we determine the length of an array that is passed into a function? (no!) `verify_password.c` (from lect5 code).
2. A preview of some pointer math. `live_session.c`

If you took CS 106B (or another C++ based class), you learned about pointers, which are simply *memory addresses*. A pointer is an integer, and on the myth machines, all pointers are **64-bits**, or **8 bytes long**.

It is difficult to stress how important understanding that a pointer is **just a memory address**. Let's look at some examples.

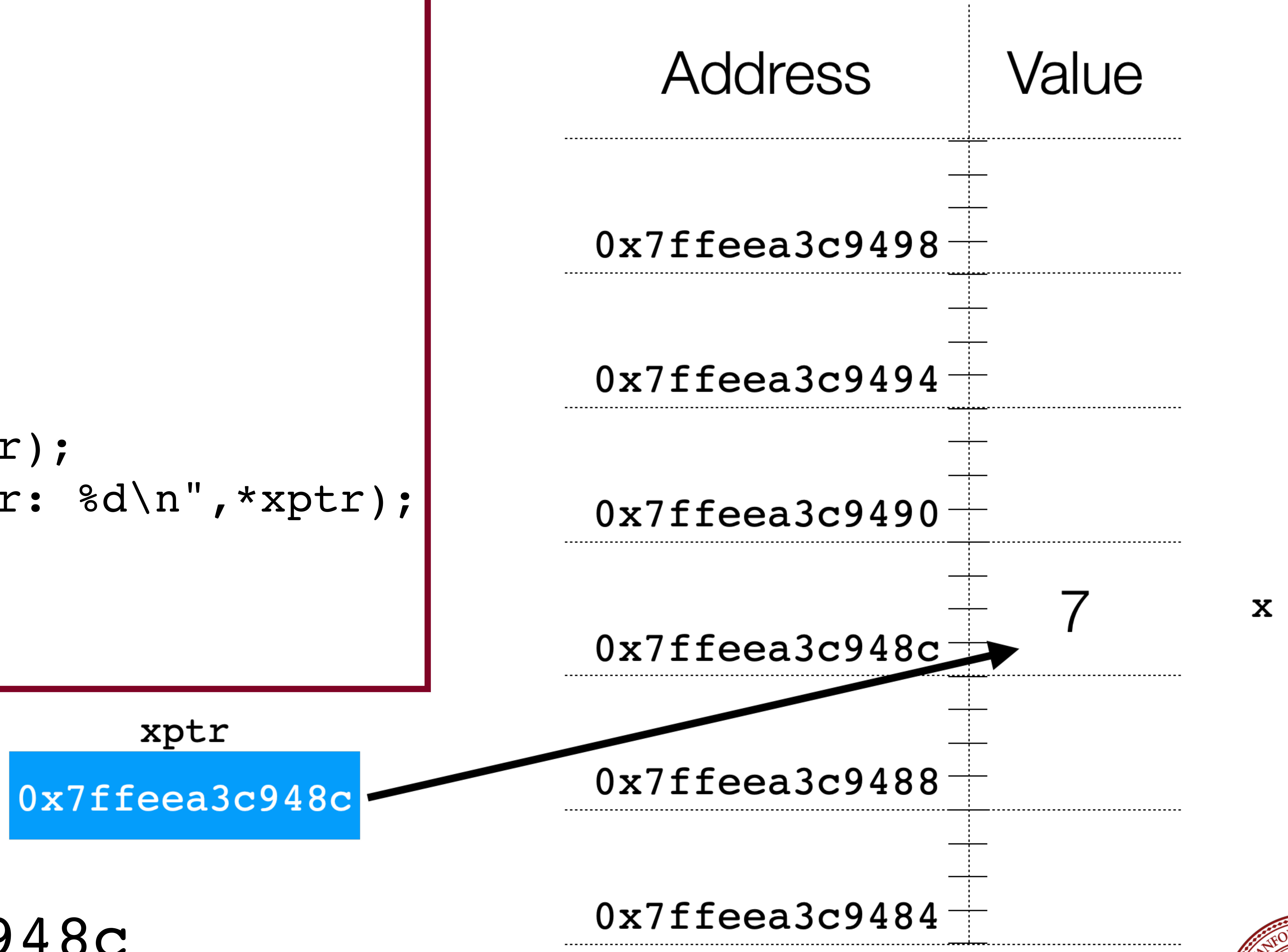
C Pointers

```
// file: pointer_ex1.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int x = 7;
    int *xptr = &x; // pointer to x
    printf("x: %d\n",x);
    printf("address of x: %p\n",xptr);
    printf("x via dereferencing xptr: %d\n",*xptr);

    return 0;
}
```

```
$ ./pointer_ex1
x: 7
address of x: 0x7ffeea3c948c
x via dereferencing xptr: 7
```



C Pointers to Pointers

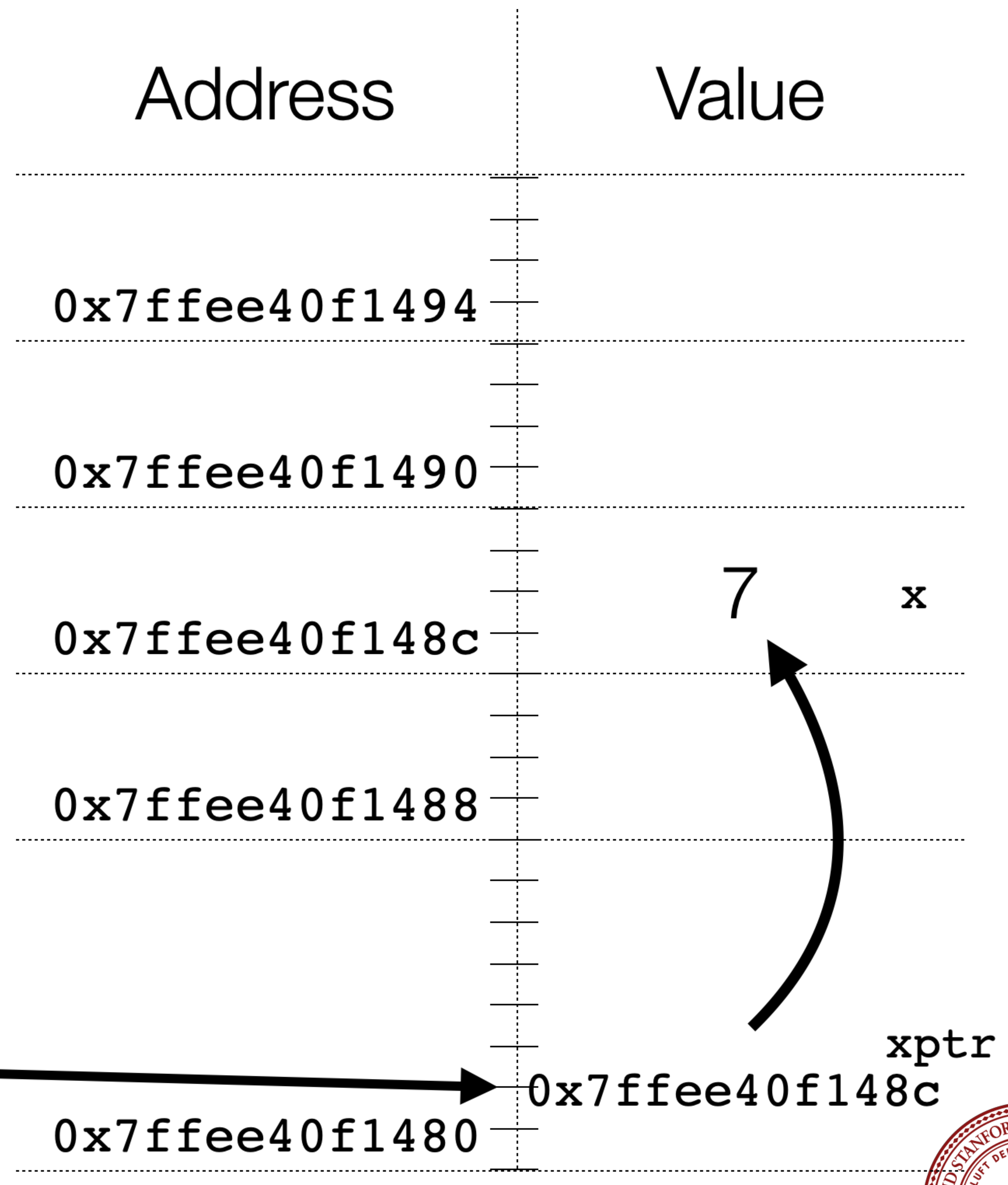
We will often use *pointers to pointers*, which means that we have the *address of a pointer.*

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

int main(int argc, char **argv)
{
    int x = 7;
    int *xptr = &x; // pointer to x
    int **xptrptr = &xptr; // pointer to xptr
    printf("x: %d\n",x);
    printf("address of x: %p\n",xptr);
    printf("address of xptr: %p\n",xptrptr);
    printf("address of x via dereferencing xptrptr: %p\n",*xptrptr);
    printf("x via double-dereferencing xptrptr: %d\n",**xptrptr);

    return 0;
}
```

```
$ ./ptrptr_ex
x: 7
address of x: 0x7ffee40f148c
address of xptr: 0x7ffee40f1480
address of x via dereferencing xptrptr: 0x7ffee40f148c
x via double-dereferencing xptrptr: 7
```



C Pointers to Pointers

Question: what does the following code print out?

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

int main(int argc, char **argv)
{
    char *str = "CS107";
    char **strptr = &str;

    char mystery = **strptr;
    printf("Mystery: %c\n",mystery);
    return 0;
}
```

C Pointers to Pointers

Question: what does the following code print out?

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

int main(int argc, char **argv)
{
    char *str = "CS107";
    char **strptr = &str;

    char mystery = **strptr;
    printf("Mystery: %c\n",mystery);
    return 0;
}
```

Answer: C

A single dereference `*strptr` produces the address of `str`.

If we were to dereference `str`:

`*str`

this would give us the first character of `str`, or '`C`'.

So, a double-dereference of `strptr`, or `**strptr` produces the character C.

Why do we use pointers?

Pointers are used as *references* to values (and for arrays, which we will get to next).

Pointers let us write functions that can modify and use values that are created elsewhere in the program, without having to make a copy of the values themselves.

Pointers allow us to refer to large data structures in a compact way. One 8-byte pointer can refer to any size data structure.

Pointers allow us to reserve new memory during a program (using **`malloc`**, **`calloc`**, and **`realloc`**, which we will cover later). It is convenient to request memory inside a function that you do not have details about at compile time (e.g., the length of a potential array).

Pointers to pointers allow us to refer to a particular element in an array generically, without even knowing about the details of what the array has in it. We will see this next week.

Arrays in C

Arrays in C are contiguous blocks of memory with a fixed length.

A programmer can access elements in an array using either bracket notation (e.g., `arr[2]`) or by dereferencing an offset from the beginning of the array (e.g., `*(arr + 2)`, more on this in a couple of slides).

Although arrays sometimes behave like pointers, they are distinct and should not be confused with pointers.

Address	Value
`0x7ffeea3c9498`	13
`0x7ffeea3c9494`	5
`0x7ffeea3c9490`	9
`0x7ffeea3c948c`	7
`0x7ffeea3c9488`	-4
`0x7ffeea3c9484`	2

Array Example

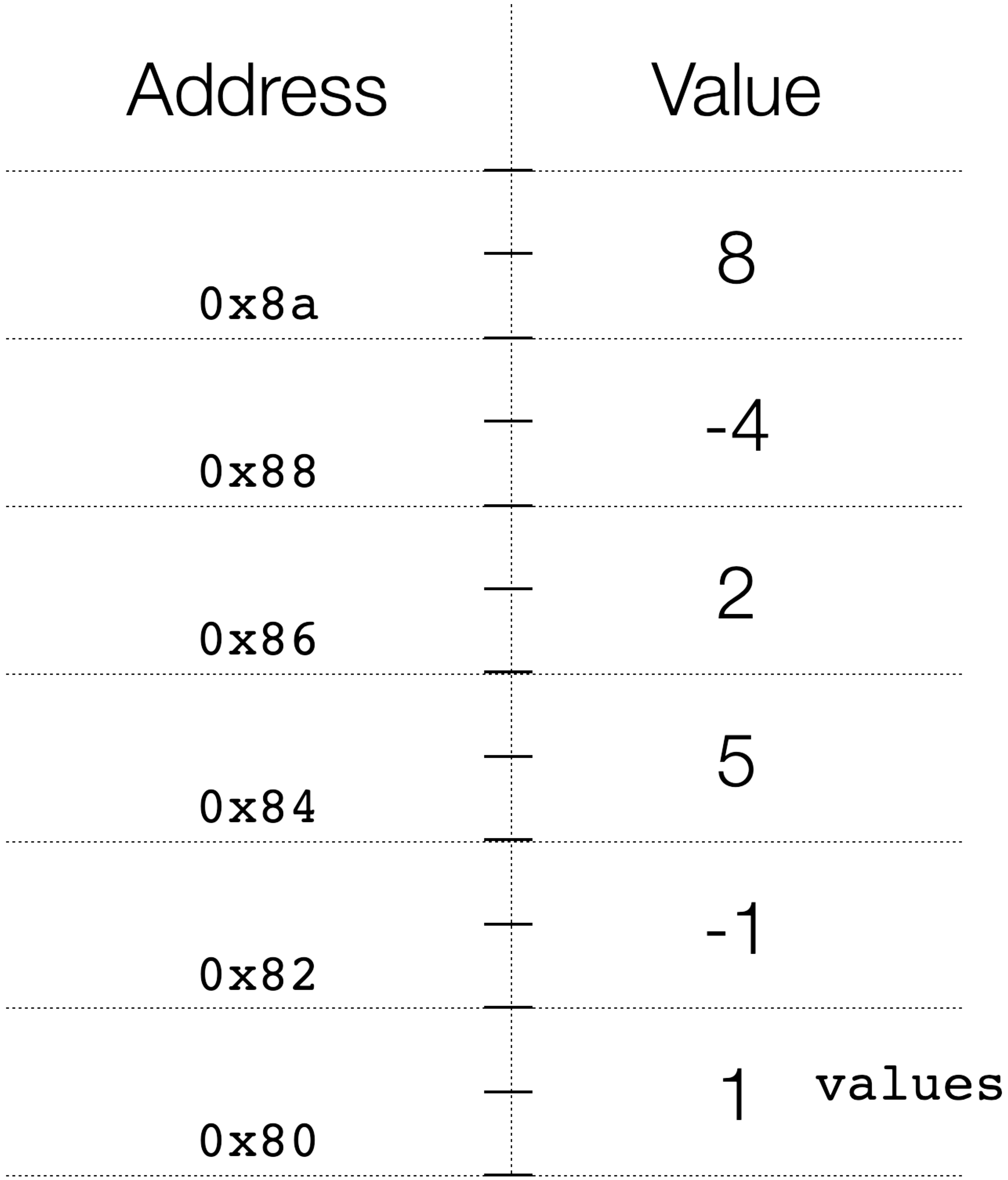
```
// file: array_ex.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    short values[] = {1, -1, 5, 2, -4, 8};

    int nelems = sizeof(values) / sizeof(values[0]);

    for (int i=0; i < nelems; i++) {
        printf("%d",values[i]);
        i == nelems -1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./array_ex
1, -1, 5, 2, -4, 8
```



Address	Value	
0x8a	8	
0x88	-4	
0x86	2	
0x84	5	
0x82	-1	
0x80	1	values

The `sizeof` for an array reports the *total number of bytes in the array*. ***Not true for pointers!***

Arrays are not pointers!

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

int main(int argc, char **argv)
{
    short values[] = {1, -1, 5, 2, -4, 8};

    printf("sizeof(values) = %lu\n",sizeof(values));
    printf("sizeof(values[0]) = %lu\n",sizeof(values[0]));

    return 0;
}
```

```
$ ./array_sizeof
sizeof(values) = 12
sizeof(values[0]) = 2
```

Arrays can behave like pointers, but there is no pointer associated with the array **`values`** in the code to the left.

The compiler keeps track of where in memory the array is, and the compiler also knows how big the array is (which is why **`sizeof(values)`** is 12. Each element is 2 bytes (because it is an array of **`short`**s, and there are six elements).

Arrays are not pointers!

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

int main(int argc, char **argv)
{
    int arr[3] = {1,2,3};
    int a = 4;
    arr = &a; // produces compile error
    return 0;
}
```

You cannot assign a different value to an array, because it isn't a pointer. (An array name is not a variable! The array name is the address of the zeroth element. See here for more info: https://stackoverflow.com/questions/62817460/why-cant-i-assign-an-array-to-another-array-in-c)

```
$ make arrays_not_pointers
gcc -g -O0 -std=gnu99 -Wall $warnflags    arrays_not_pointers.c   -o arrays_not_pointers
arrays_not_pointers.c:9:9: error: array type 'int [3]' is not assignable
    arr = &a; // produces compile error
    ~~~ ^
1 error generated.
make: *** [arrays_not_pointers] Error 1
```

You can assign array addresses to pointers

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

int main(int argc, char **argv)
{
    int arr[3] = {1,2,3};
    int *arrptr = arr;
    arrptr = &arr[0]; // also works
    printf("%p\n",arrptr);
    printf("%p\n",arr);
    printf("%p\n",&arr[0]);
    printf("%p\n",&arr); // also works, but try to avoid this
    return 0;
}
```

Here, `arr` is "decaying" into a pointer when we assign its value to `arrptr`.

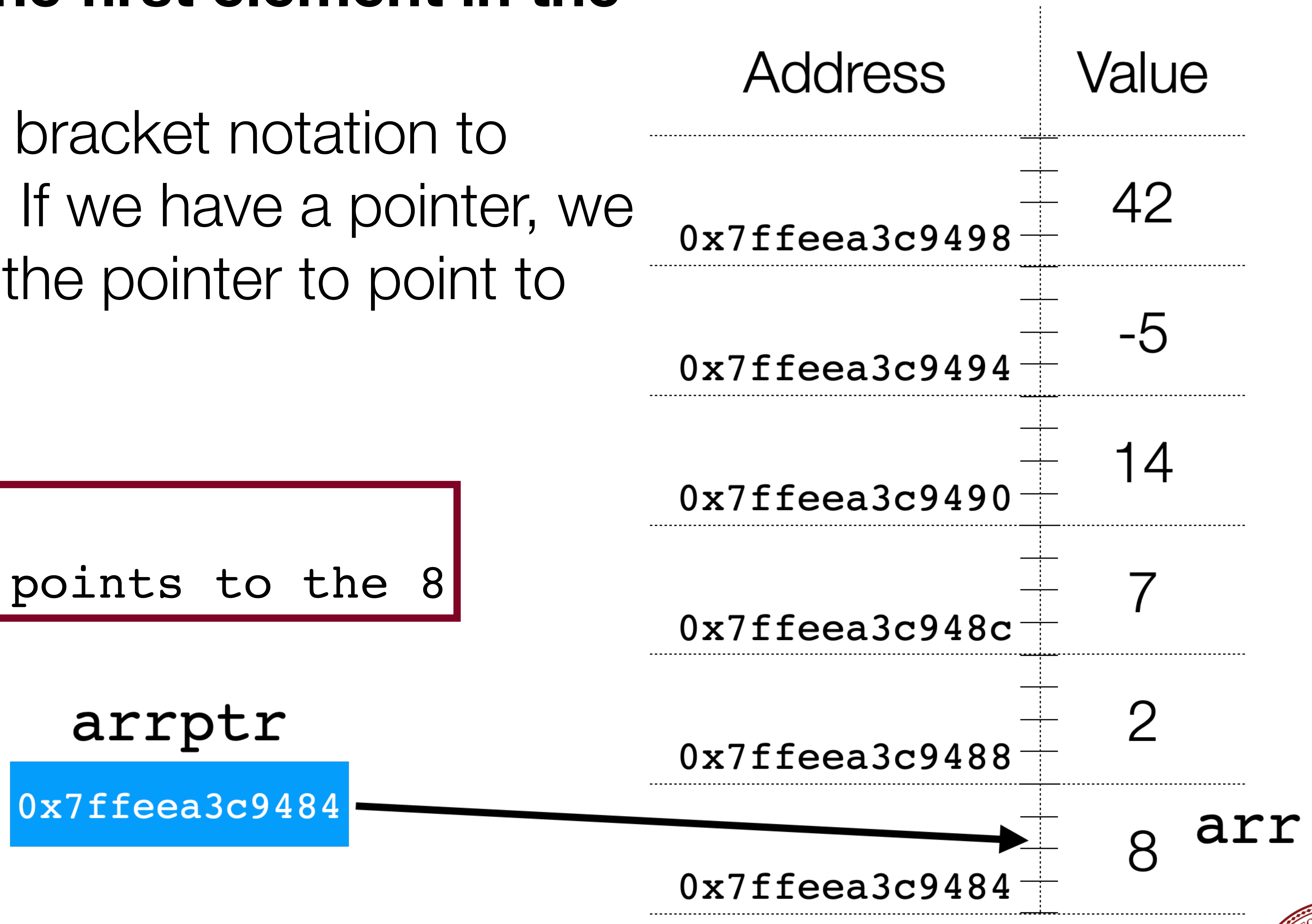
```
$ ./assign_array_to_pointer
0x7ffee363d60c
0x7ffee363d60c
0x7ffee363d60c
0x7ffee363d60c
```

Pointers to arrays

A pointer to an array **points to the first element in the array.**

We can use *pointer arithmetic* or bracket notation to access the elements in the array. If we have a pointer, we can actually change the value of the pointer to point to another place in the array.

```
int arr[] = {8,2,7,14,-5,42};
int *arrptr = arr; // arr now points to the 8
```

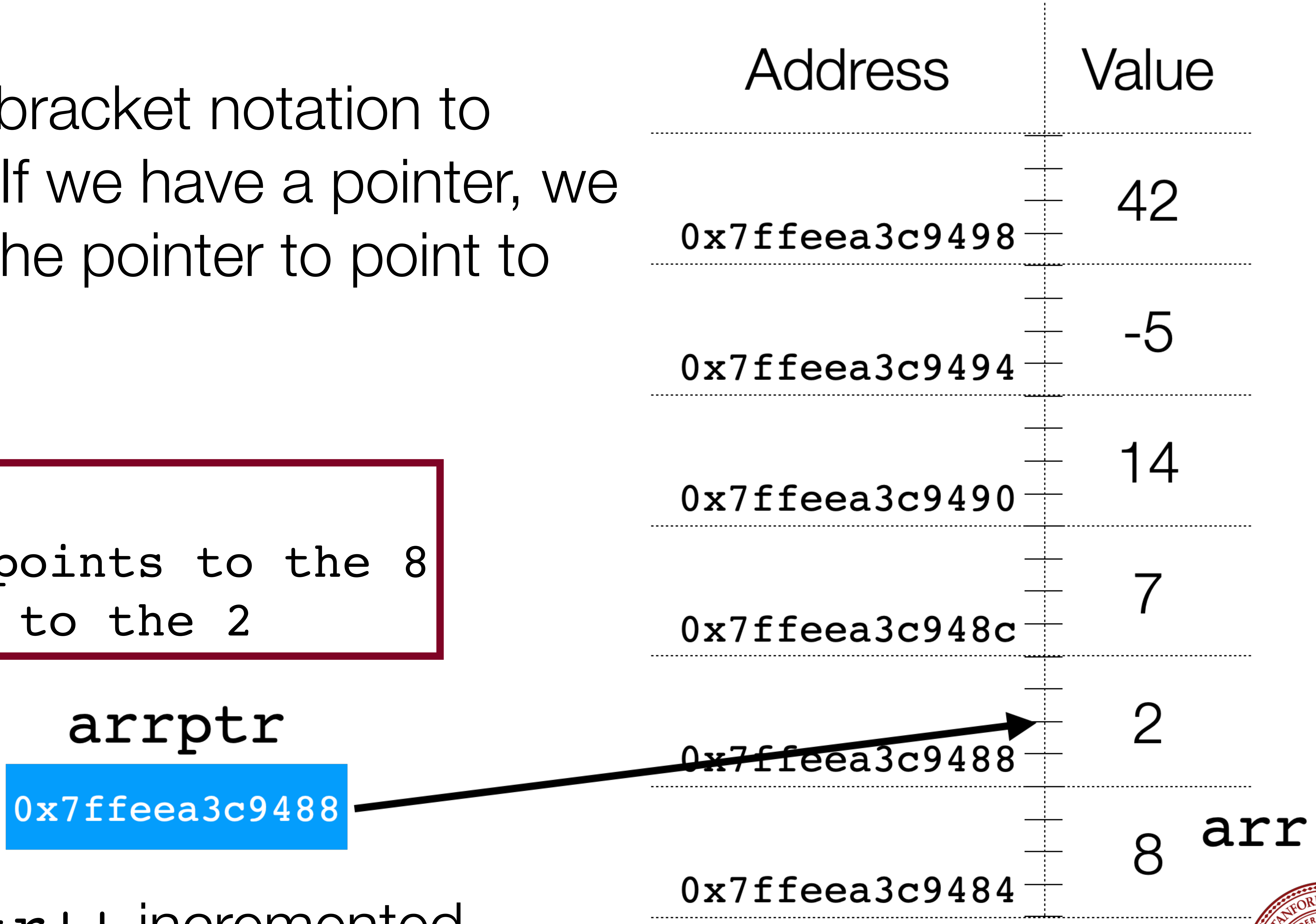


Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

Pointers to arrays

A pointer to an array **points to the first element in the array.**
We can use *pointer arithmetic* or bracket notation to access the elements in the array. If we have a pointer, we can actually change the value of the pointer to point to another place in the array.

```
int arr[] = {8,2,7,14,-5,42};
int *arrptr = arr; // arr now points to the 8
arrptr++; // arrptr now points to the 2
```



Notice that the expression `arrptr++` incremented `arrptr` by 4! The compiler knows how wide the type is!

Pointers to arrays

A very important piece of information was on the bottom of the last slide:

Notice that the expression **`arrptr++`** incremented **`arrptr`** by 4! The compiler knows how wide the type is!

One of the main reasons we have pointer types is so the compiler knows how big each element in the array is. Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    int arr[] = {1,2,3,4,5,6};
    int *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffedfe72780
```

Pointers to arrays

A very important piece of information was on the bottom of the last slide:

Notice that the expression **`arrptr++`** incremented **`arrptr`** by 4! The compiler knows how wide the type is!

One of the main reasons we have pointer types is so the compiler knows how big each element in the array is. Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    int arr[] = {1,2,3,4,5,6};
    int *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffedfe72780
0x7ffedfe72784
```

Pointers to arrays

A very important piece of information was on the bottom of the last slide:

Notice that the expression **`arrptr++`** incremented **`arrptr`** by 4! The compiler knows how wide the type is!

One of the main reasons we have pointer types is so the compiler knows how big each element in the array is. Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    int arr[] = {1,2,3,4,5,6};
    int *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffedfe72780
0x7ffedfe72784
0x7ffedfe72788
0x7ffedfe7278c
0x7ffedfe72790
0x7ffedfe72794
```

Pointers to arrays

Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    long arr[] = {1,2,3,4,5,6};
    long *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffedfc23770
```

Pointers to arrays

Continue the printout from the following program:

```c
int main(int argc, char **argv)
{
    long arr[] = {1,2,3,4,5,6};
    long *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffedfc23770
0x7ffedfc23778
0x7ffedfc23780
0x7ffedfc23788
0x7ffedfc23790
0x7ffedfc23798
```

Note that 0x8 + 0x8 = 0x10

Pointers to arrays

Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    char arr[] = {1,2,3,4,5,6};
    char *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffee9fe678b
```

Pointers to arrays

Continue the printout from the following program:

```
int main(int argc, char **argv)
{
    char arr[] = {1,2,3,4,5,6};
    char *arrptr = arr; // arr decays to a pointer

    printf("%p\n",arrptr);
    printf("%p\n",arrptr+1);
    printf("%p\n",arrptr+2);
    printf("%p\n",arrptr+3);
    printf("%p\n",arrptr+4);
    printf("%p\n",arrptr+5);
    return 0;
}
```

```
./ptr_arith
0x7ffee9fe678b
0x7ffee9fe678c
0x7ffee9fe678d
0x7ffee9fe678e
0x7ffee9fe678f
0x7ffee9fe6790
```

3 minute break

Pointers to arrays

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

int main(int argc, char **argv)
{
    int arr[] = {8,2,7,14,-5,42};
    int nelems = sizeof(arr) / sizeof(arr[0]);

    int *arrptr = arr; // arr now points to the 8
    printf("Address of array:  %p\n",arr);
    printf("Address of arrptr: %p\n",arrptr);

    for (int i=0; i < nelems; i++) {
        printf("%d",*arrptr++); // we need to unpack this!
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./pointer_to_array
Address of array:  0x7ffeea3c9484
Address of arrptr: 0x7ffeea3c9484
8, 2, 7, 14, -5, 42
```

Address	Value	
0x7ffeea3c9498	42	
0x7ffeea3c9494	-5	
0x7ffeea3c9490	14	
0x7ffeea3c948c	7	
0x7ffeea3c9488	2	
0x7ffeea3c9484	8	arr

Pointers to arrays

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

int main(int argc, char **argv)
{
    int arr[] = {8,2,7,14,-5,42};
    int nelems = sizeof(arr) / sizeof(arr[0]);

    int *arrptr = arr; // arr now points to the 8
    printf("Address of array:  %p\n",arr);
    printf("Address of arrptr: %p\n",arrptr);

    for (int i=0; i < nelems; i++) {
        printf("%d",*arrptr++); // we need to unpack this!
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

`*arrptr++`

means:

1. dereference `arrptr` and return the value.
2. increment *the pointer value* for `arrptr`.

It does not mean "add one to the value that `arrptr` points to! To do would need to write: `(*arrptr)++`

Pointers to arrays

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

int main(int argc, char **argv)
{
    int arr[] = {8,2,7,14,-5,42};
    int nelems = sizeof(arr) / sizeof(arr[0]);

    int *arrptr = arr; // arr now points to the 8

    // increment all values in array
    for (int i=0; i < nelems; i++) {
        (*arrptr)++;
        arrptr++;
    }

    for (int i=0; i < nelems; i++) {
        printf("%d",arr[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

This is not the cleanest code, but you will see things like this in C quite often.

You could actually combine the highlighted line and the next line into the even more unreadable:

`(*arrptr++)++;`

Please try to write nice clean code. (i.e., not this ☝)

BTW, the combined line of code does the following: 1. returns the value pointed to by `arrptr`, 2. increments both `arrptr` and the value in the array.

More examples of arrays decaying to pointers

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

void sizeof_test(int arr[]) {
    printf("sizeof(arr) in function: %lu\n", sizeof(arr));
}

int main()
{
    int arr[] = {1,3,4,2,7,9};

    printf("%d\n",arr[2]); // prints 4
    printf("%d\n",*(arr+5)); // prints 9

    // set the value of the third element
    arr[2] = 42;
    printf("%d\n",arr[2]); // prints 42

    printf("sizeof(arr) in main: %lu\n",sizeof(arr));
    sizeof_test(arr);

    return 0;
}
```

When you pass an array into a function that expects a pointer, the array decays into a pointer, and it loses the ability to determine its size using **`sizeof`**.

In the **`sizeof_test`** function, even though the declaration looks like an array, it is really an int pointer.

```
$ ./array
4
9
42
sizeof(arr) in main: 24
sizeof(arr) in function: 8
```

More examples of arrays decaying to pointers

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

void sizeof_test(int arr[]) {
    printf("sizeof(arr) in function: %lu\n", sizeof(arr));
}

int main()
{
    int arr[] = {1,3,4,2,7,9};

    printf("%d\n",arr[2]); // prints 4
    printf("%d\n",*(arr+5)); // prints 9

    // set the value of the third element
    arr[2] = 42;
    printf("%d\n",arr[2]); // prints 42

    printf("sizeof(arr) in main: %lu\n",sizeof(arr));
    sizeof_test(arr);

    return 0;
}
```

Note the use of pointer arithmetic and bracket notation — either is fine for an array, because the array variable decays to a pointer when used in these circumstances.

Arrays are passed by reference

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

void scale(int array[], int factor, size_t nelems)
{
    for (size_t i = 0; i < nelems; i++) {
        array[i] *= factor;
    }
}
int main(int argc, char **argv)
{
    int arr[] = {1,2,3,4,5};
    size_t nelems = sizeof(arr) / sizeof(arr[0]);

    scale(arr,10,nelems);
    for (int i=0; i < nelems; i++) {
        printf("%d",*(arr + i));
        i == nelems - 1 ? printf("\n") : printf(",");
    }

    return 0;
}
```

Arrays are passed by reference when used as arguments in a function, because the array variable decays to a pointer, which is just an address.

In other words, a function has access to the array in the function.

```
$ ./array_pass_by_ref
10,20,30,40,50
```

Arrays are passed by reference

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

long dot_prod(const long a[], const long b[], size_t nelems)
{
    long result = 0;
    for (size_t i=0; i < nelems; i++) {
        result += a[i] * b[i];
    }
    return result;
}
int main(int argc, char **argv)
{
    long a[] = {1,2,3};
    long b[] = {8,9,10};
    size_t nelems = sizeof(a) / sizeof(a[0]);
    printf("Dot product of a . b: %lu\n",dot_prod(a,b,nelems));
    return 0;
}
```

If a programmer wants to indicate that an array elements will not be modified, they pass it as a const array.

```
$ ./dot_product
Dot product of a . b: 56
```

memcpy

In the last lecture, we discussed **`strcpy`** and **`strncpy`**, which are used to make copies of strings into buffers large enough to hold them. A more generic memory copying function, **`memcpy`**, is used to copy array data.

```
void *memcpy(void *dest, const void *src, size_t n);
```

We haven't yet covered the "**`void *`**" pointer yet, but for now it is enough to know that any pointer can be passed into the **`memcpy`** function. Note that, like **`strncpy`**, **`memcpy`** has a parameter for the *number of bytes to be copied*, **`n`**. Unlike **`strncpy`**, exactly **`n`** bytes will always be copied.

When using **`memcpy`**, **src and dest may not overlap**. If you have overlapping areas, use **`memmove`**, instead (covered next).

memcpy

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

int main(int argc, char **argv)
{
    int src[] = {1,2,3,4,5};
    int nelems = sizeof(src) / sizeof(src[0]);

    int dest[nelems]; // will set aside enough bytes

    memcpy(dest,src,nelems * sizeof(int));

    for (int i=0; i < nelems; i++) {
        printf("%d",dest[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./memcpy_ex
1, 2, 3, 4, 5
```

Note that we passed the number of bytes. We often have to use `sizeof(type)` to determine how wide each element in the array is.

memmove

The **memmove** function performs the same operation as memcpy, but it allows copying from arrays that overlap.

```
void *memmove(void *dest, const void *src, size_t n);
```

The reason for using **memcpy** over **memmove** is that sometimes **memcpy** can be faster, as it does not have to do anything special to determine if there are overlapping areas.

memmove

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

int main(int argc, char **argv)
{
    int arr[] = {1,2,3,4,5};
    int nelems = sizeof(arr) / sizeof(arr[0]);

    memove(arr,arr+2, 3 * sizeof(int));

    for (int i=0; i < nelems; i++) {
        printf("%d",arr[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./memmove_ex
3, 4, 5, 4, 5
```

Note that **`memmove`** does not do any "moving" in the sense that the elements that it copied from are still present where they were originally, unless they became written over because of overlap.

declaring strings as arrays or pointers

It is possible to declare a specific string using a *string literal* in two different ways in C, and you should carefully note the differences:

```
char s1[] = "string in an array"; // modifiable
char *s2 = "string literal with pointer"; // not modifiable
```

In the first case, the string literal is used to initialize an array that you can modify (and we have seen examples of this already).

In the second case, there is no array, and there is just a pointer to the string literal. The following would likely cause your program to crash:

```
s2[0] = 'S';
```

declaring strings as arrays or pointers

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

int main(int argc, char **argv)
{
    char *s2 = "string literal with pointer";
    s2[0] = 'S'; // crashes here

    printf("s2: %s\n",s2);
    return 0;
}
```

```
$ ./string_literal_crash
Segmentation fault (core dumped)
```

```
$ gdb string_literal_crash
(gdb) run
Starting program: /
afs/.ir.stanford.edu/users/c/g/
cgregg/tmp/string_literal_crash

Program received signal SIGSEGV,
Segmentation fault.
0x0000000000400541 in main (argc=1,
argv=0x7fffffffea78) at
string_literal_crash.c:8
8       s2[0] = 'S'; // crashes here
```

be careful dereferencing!

Be careful when dereferencing pointers. NULL pointers or bogus addresses lead to segfaults:

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

int main(int argc, char **argv)
{
    char *err;
    size_t address = strtoul(argv[1],&err,0);
    printf("Let's see what int is at address %lx:\n",address);
    printf("%d\n",*(char *)address);
    return 0;
}
```

```
$ ./deref_problems 0x1234
Let's see what int is at address 1234:
Segmentation fault: 11
$ ./deref_problems 0x7ffeea3c948c
Let's see what int is at address 7ffeea3c948c:
Segmentation fault: 11
```

be careful with your array bounds!

C does **not** check for the end of your array, and it will gladly let you walk right off the end of the array:

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

#define NUMS 10000

int main(int argc, char **argv)
{
    int arr[] = {0,1,2,3,4};
    for (int i=0; i < NUMS; i++) {
        printf("%d",*(arr + i));
        i == NUMS ? printf("\n") : printf(",");
    }

    return 0;
}
```

```
$ ./buffer_overflow
0,1,2,3,4,32767,-1052713984,1018355125,41959
36,0,-1863256016,32660,0,0,935138104,32767,0
,1,4195734,0,0,0,1702390139,-2045811791,4195
488,0,935138096,32767,0,0,0,0,1299736955,204
5840459,-2039977605,2032550608,0,0,0,0,0,0,1
,0,4195734,0,4196048,0,0,0,0,0,4195488,0,935
138096,32767,0,0,4195529,0,935138088,32767,2
8,0,1,0,935140602,32767,0,0,935140620,32767,
935140640,32767,935140651,32767,935140667,32
767,935140681,32767,935140714,32767,93514073
4,32767,935140746,32767,935140768,32767,9351
40938,32767,935140971,32767,935141005,32767,
935141016,32767,935141033,32767,935141079,32
767,935141087,32767,935141118,32767,93514113
4,32767,935141149,32767,935141214,32767,9351
41264,32767,935141297,32767,935141317,32767,
0,0,33,0,935301120,32767,16,0,-1075053569,0,
6,0,4096,0,17,0,100,0,3,0,4194368,0,4,0,56,0
,5,0,9,0,7,0,-1859416064,32660,8,0,0,0,9,0,4
195488,0,11,0,306920,0,12,0,Segmentation
fault (core dumped)
```

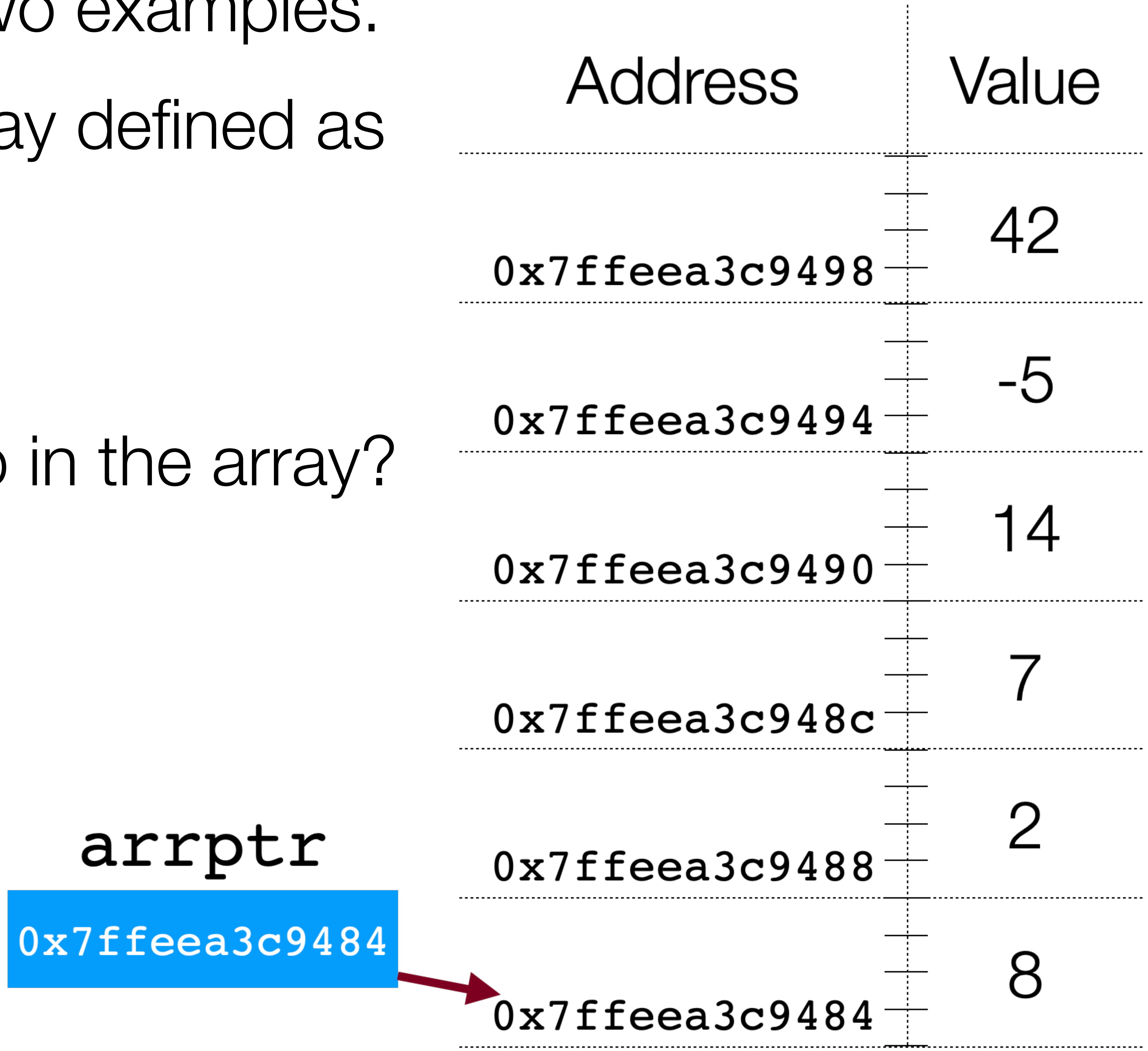
Pointers to Arrays — Memory Footprint

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};
```

How many bytes does each `int` take up in the array?



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

Pointers to Arrays — Memory Footprint

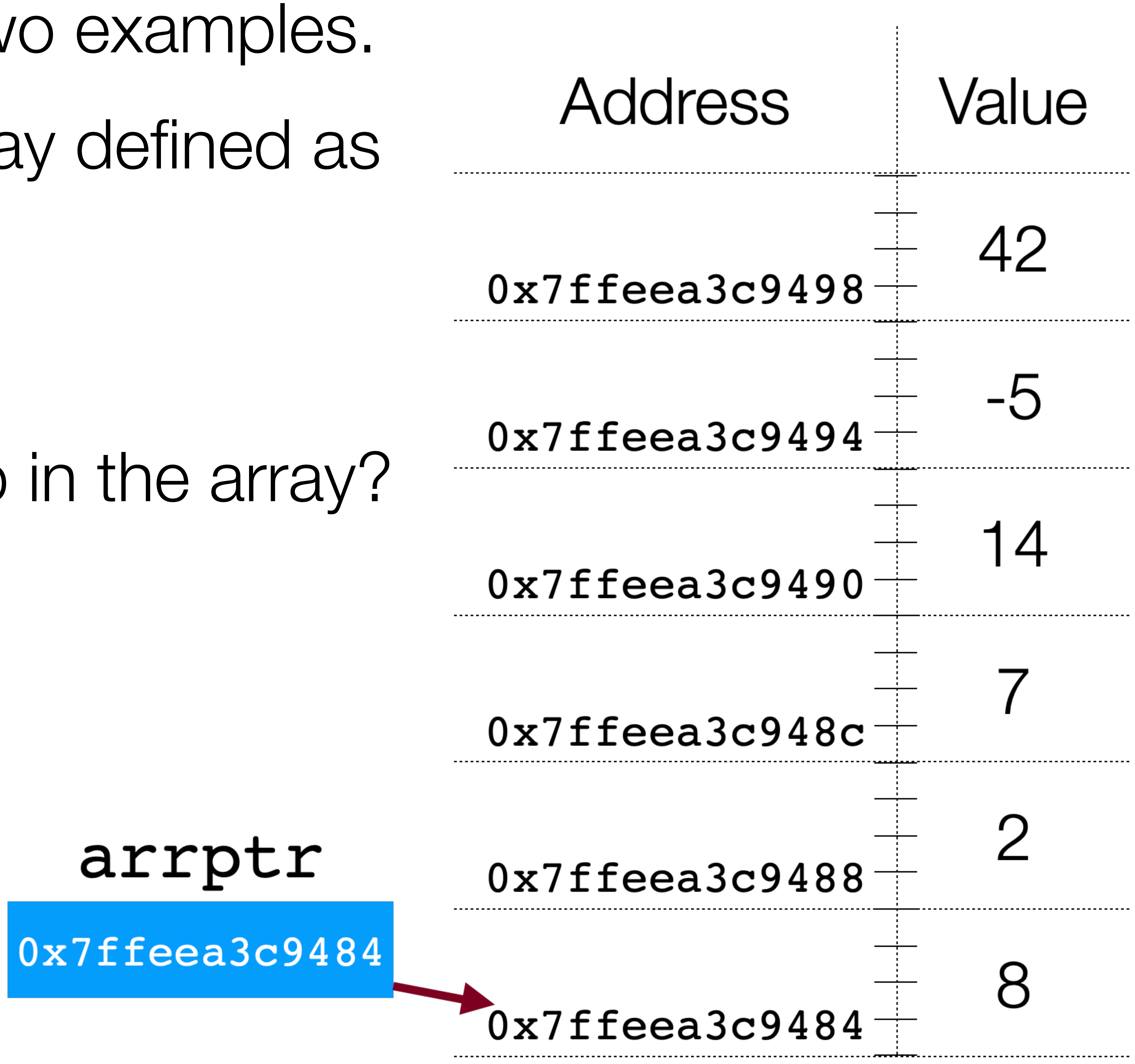
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

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

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

How many bytes does each `int` take up in the array?

4



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

Pointers to Arrays — Memory Footprint

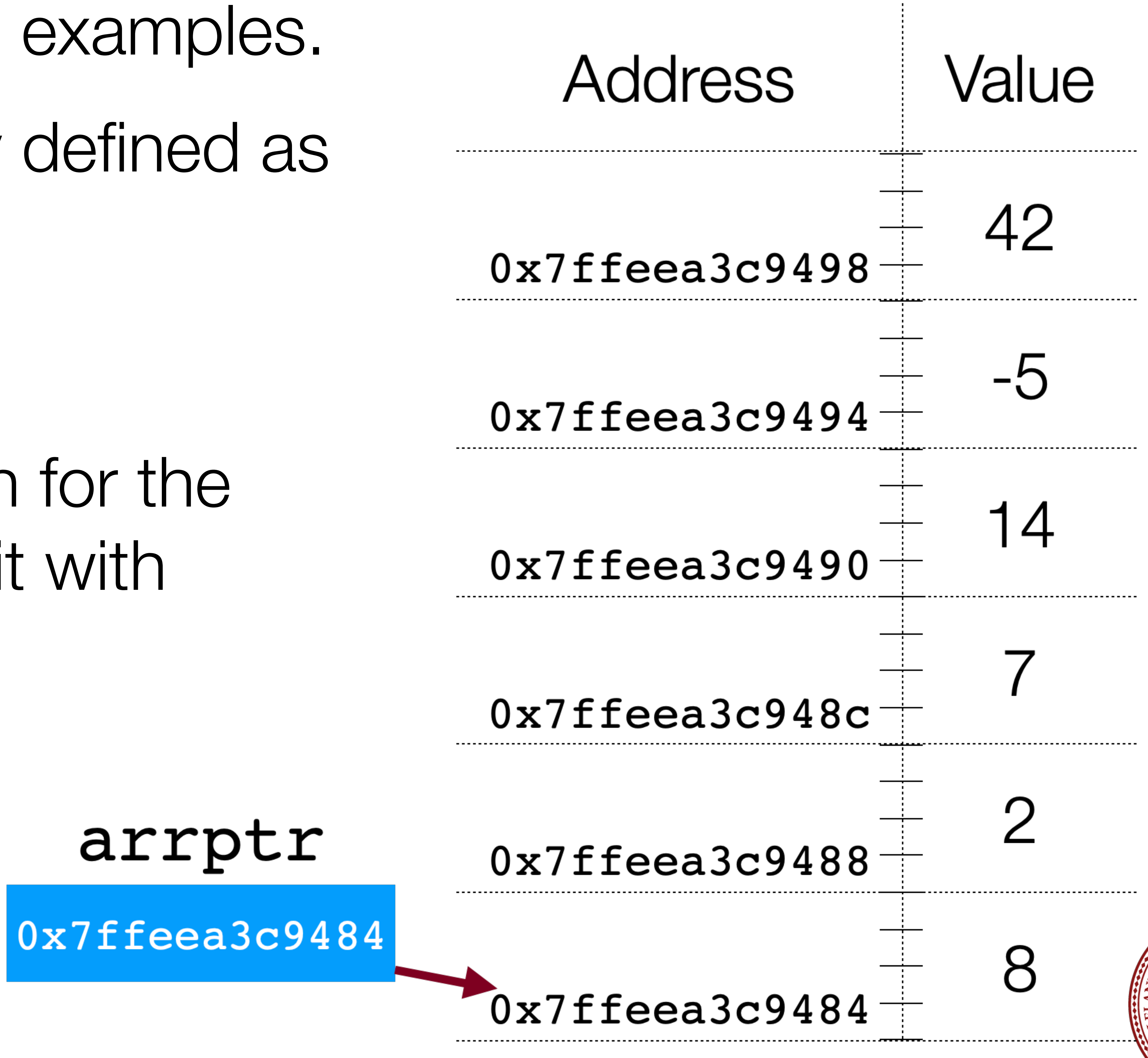
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

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;
}
```



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

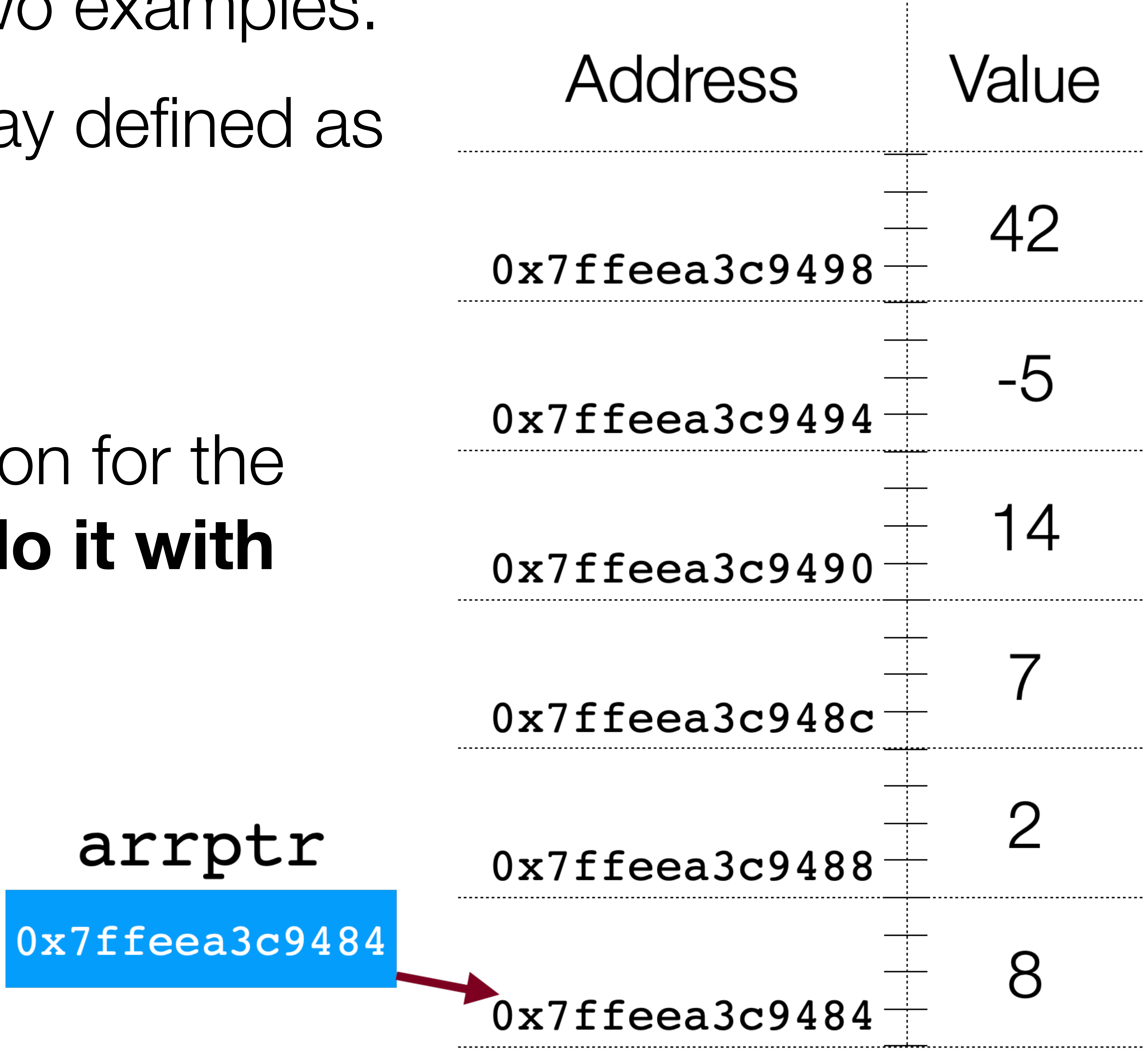
Pointers to Arrays — Memory Footprint

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?**



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

Pointers to Arrays — Memory Footprint

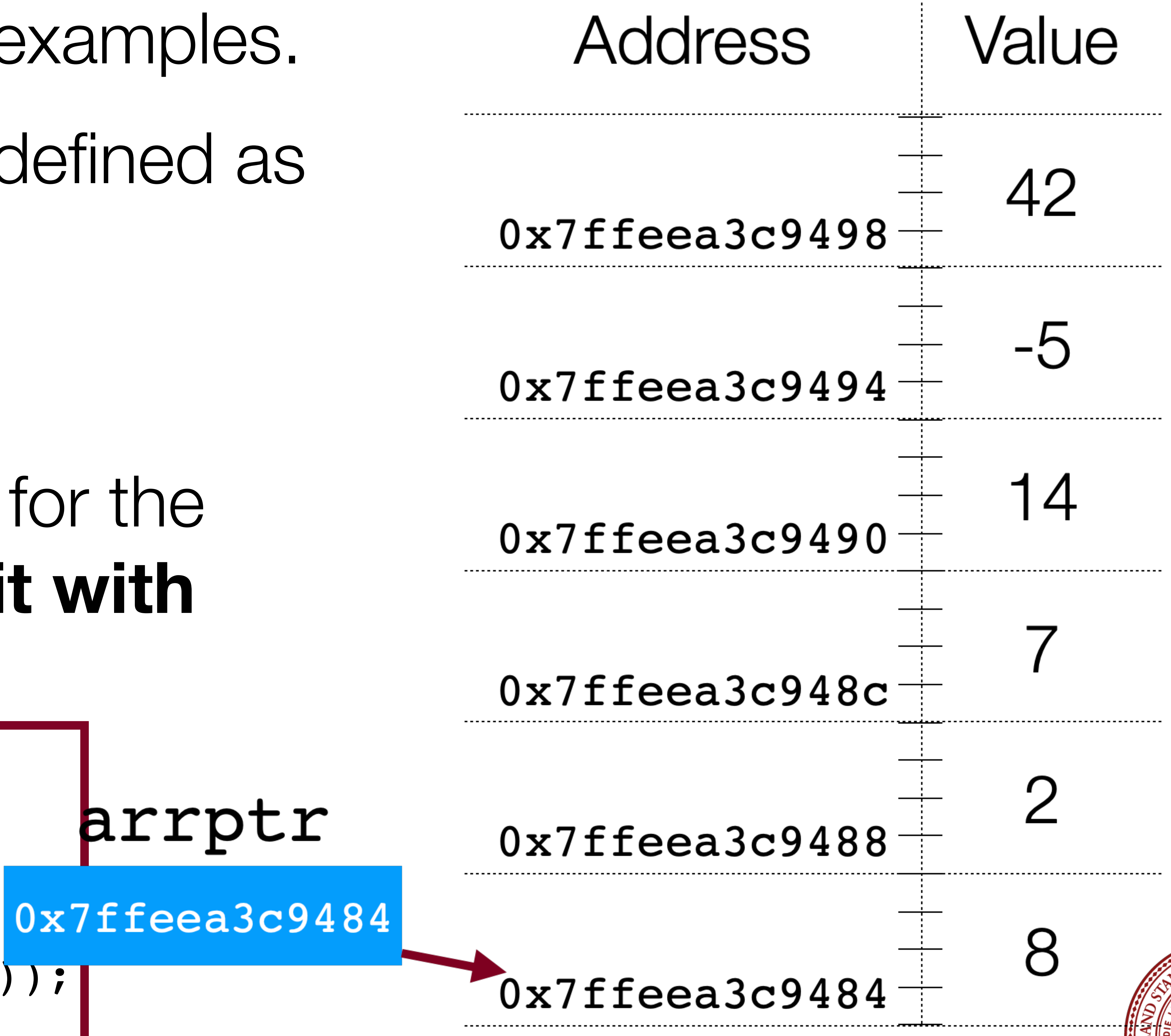
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. Let's take a look at two examples.

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));
}
```



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

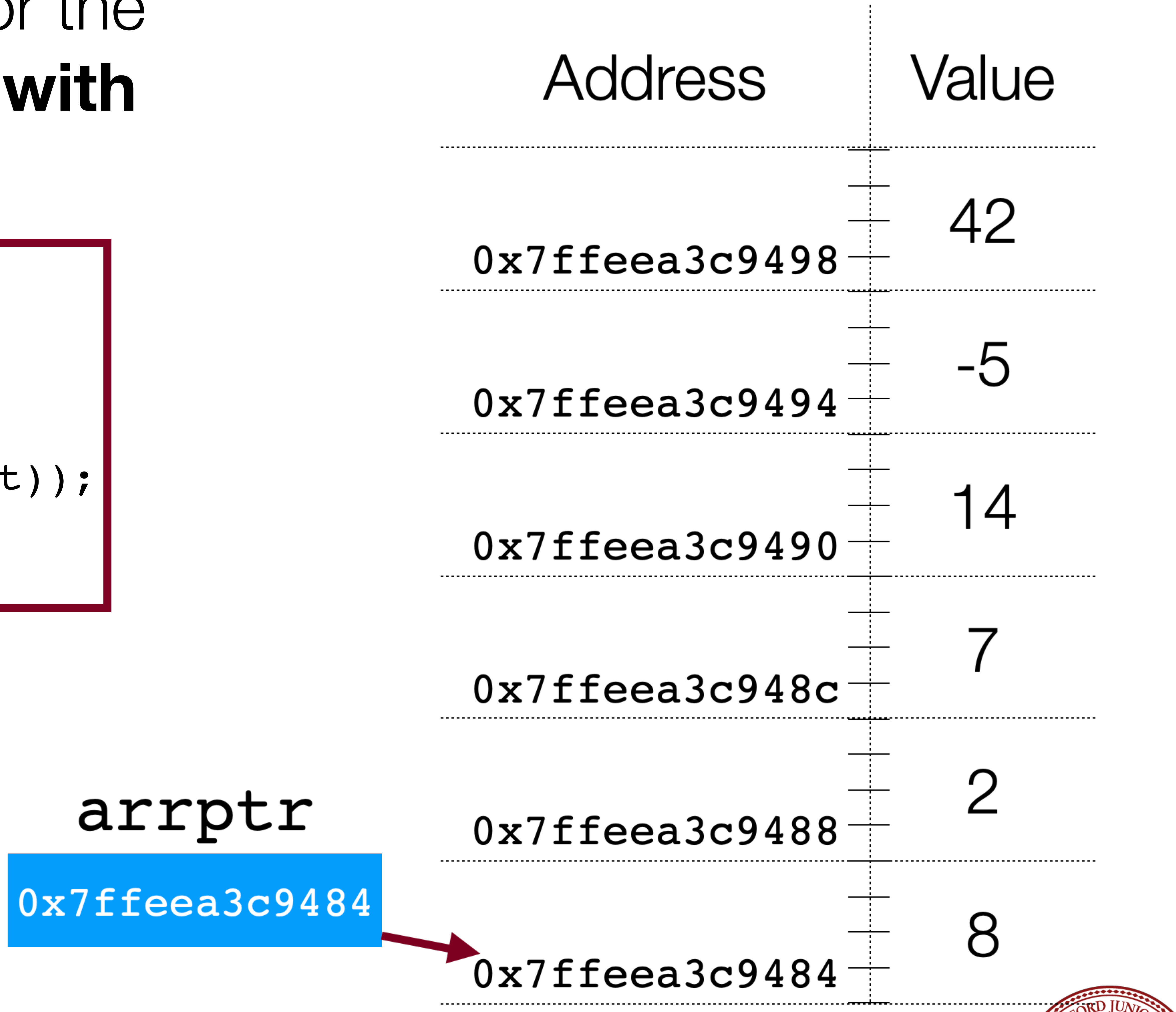
Pointers to Arrays — Memory Footprint

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* `ints`*)*

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



Address	Value
0x7ffeea3c9498	42
0x7ffeea3c9494	-5
0x7ffeea3c9490	14
0x7ffeea3c948c	7
0x7ffeea3c9488	2
0x7ffeea3c9484	8

`arrptr`

`0x7ffeea3c9484`

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;
}
```

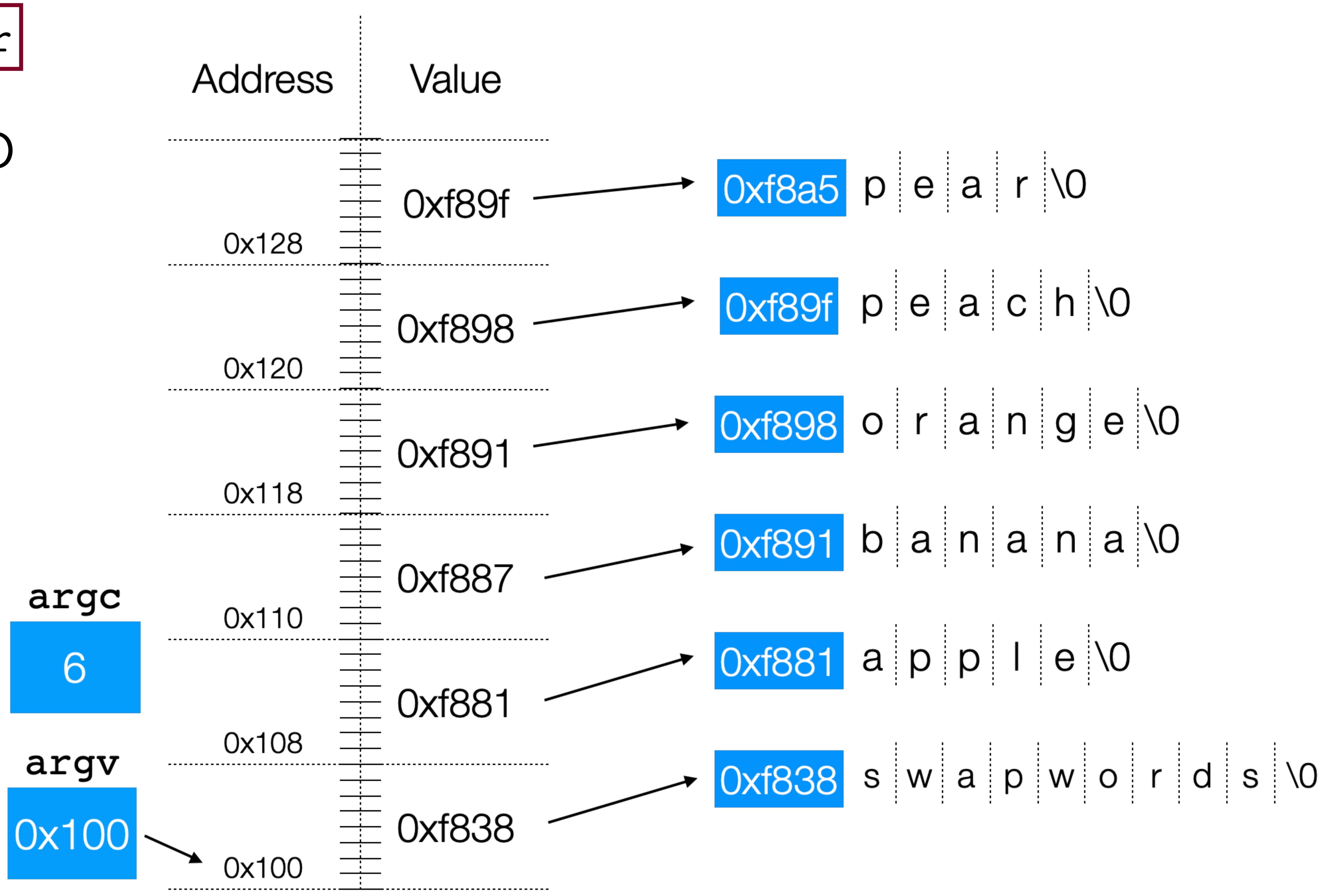
```
$ ./pointer_to_array1
42, 7, 2, 14, -5, 8
```

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

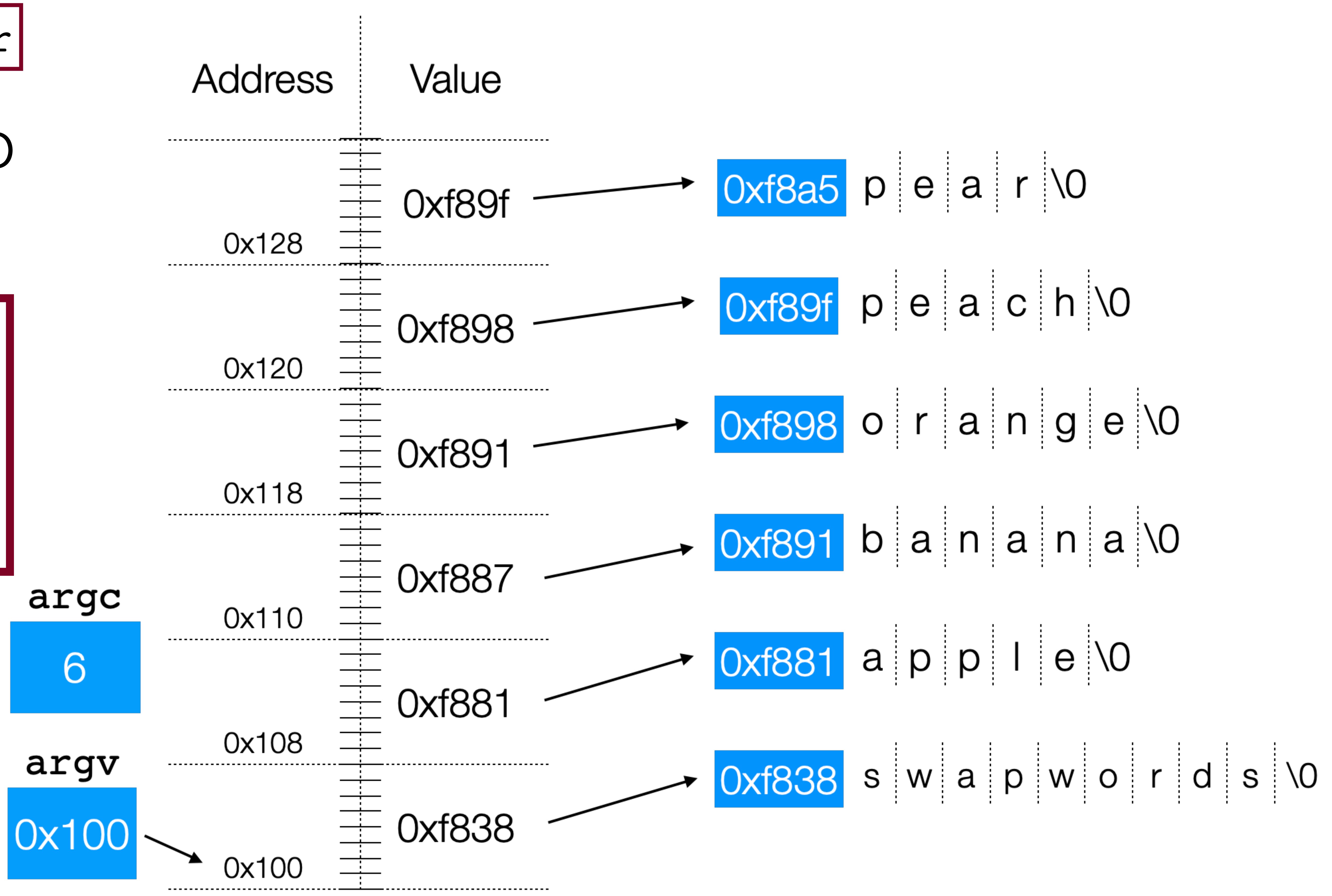
For our second example, let's look at **`argv`**, which is an array of **`char *`** pointers:

```
./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!

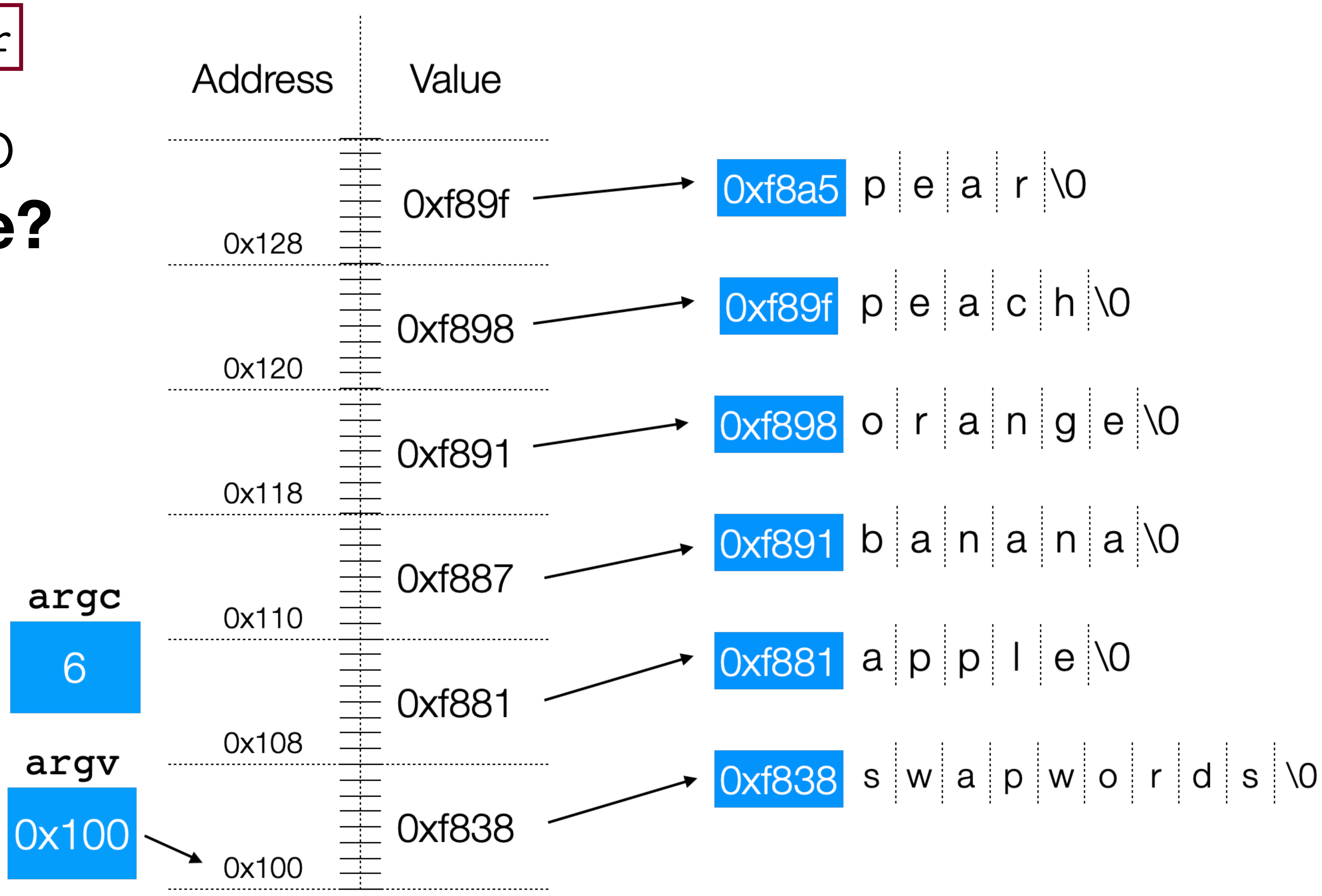


Pointers to Arrays — Memory Footprint

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

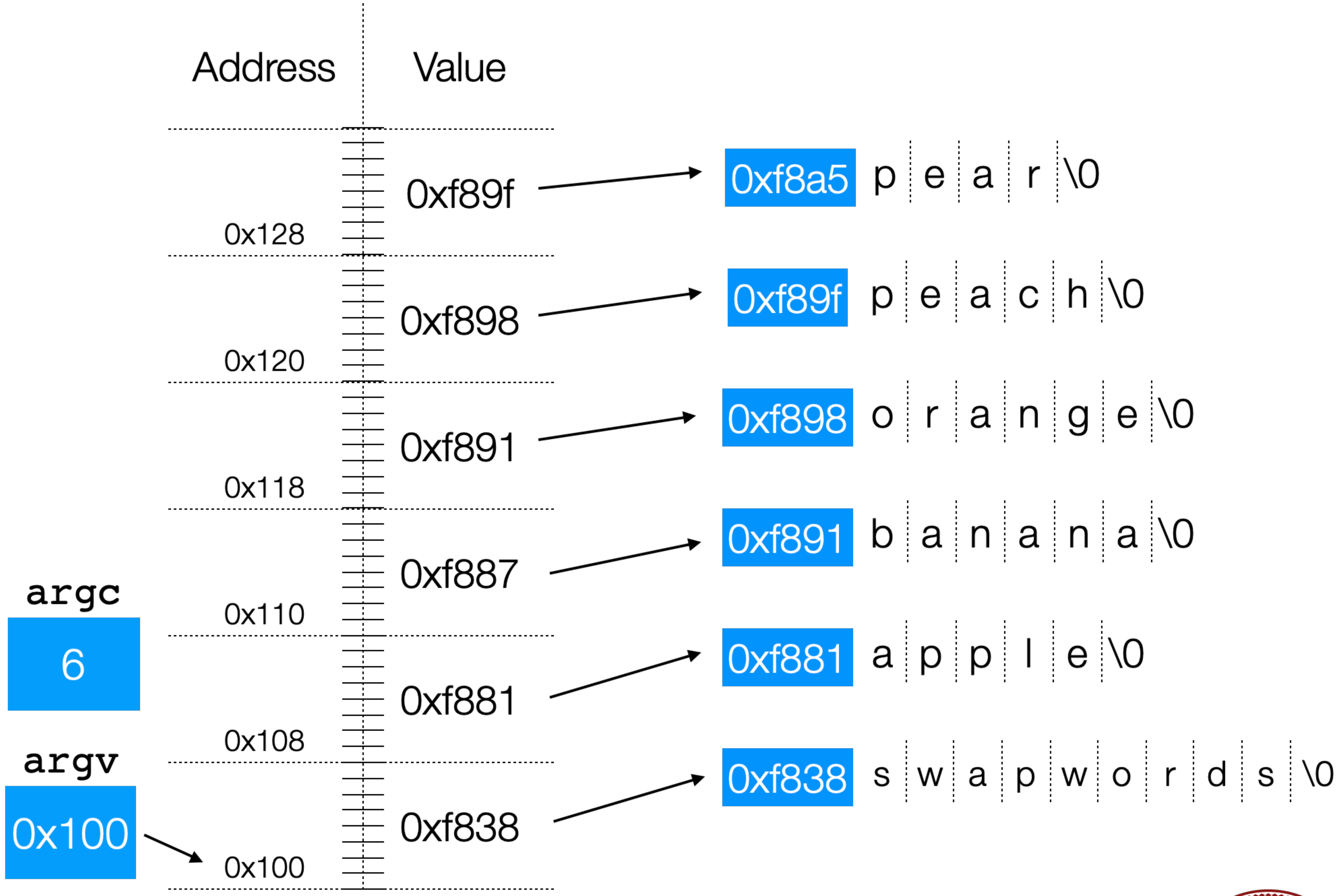
For our second example, let's look at `argv`, which is an array of `char *` pointers:

```
./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);
}
```

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