ECS 36A,
April 16 and 18, 2024

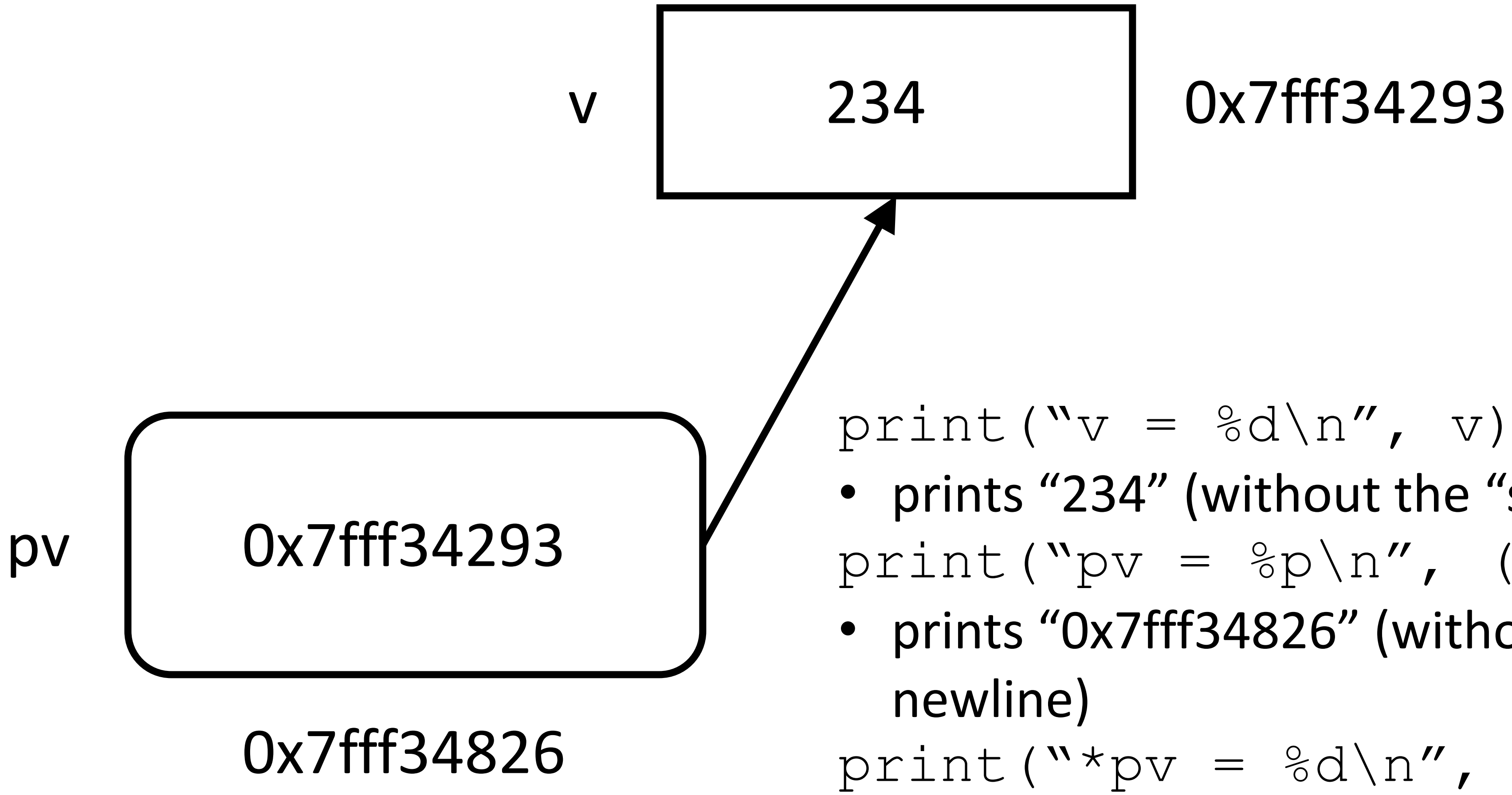
Pointers

- A variable containing the address of another variable
- Example:

```
int x = 0;
int *px;
px = &x;
printf("x = %d, px = %p, *px = %d\n", x, (void *)px, *px);
```

- Operators:
  - &*variable*: address of *variable*
  - **variable*: what is in the memory location with the address stored in *variable*

In Pictures

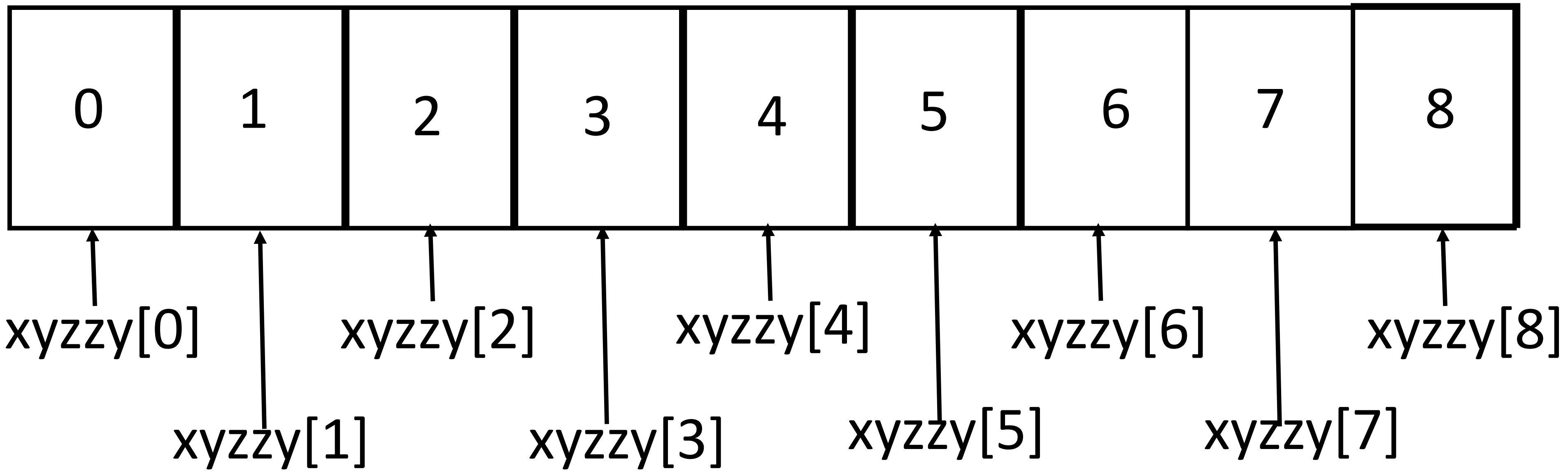


```
print("v = %d\n", v);
```
- prints "234" (without the "s, ending in newline)

```
print("pv = %p\n", (void *)pv);
```
- prints "0x7fff34826" (without the "s, ending in newline)

```
print("*pv = %d\n", *py);
```
- prints "234" (without the "s, ending in newline)

C Arrays


0
1
2
3
4
5
6
7
8
xyzzy[0]
xyzzy[1]
xyzzy[2]
xyzzy[3]
xyzzy[4]
xyzzy[5]
xyzzy[6]
xyzzy[7]
xyzzy[8]

Arrays as Pointers and *Vice Versa*

- Arrays are simply another way to express pointers
  - So xarray[0] and *xarray refer to the same memory location
  - And xarray[12] and *(xarray+12) refer to the same memory location

Pointer Arithmetic

- *type* *x;
  - x + 10 refers to the 10th *type* object; so if *type* is an int, x + 10 refers to the 10th integer memory location beyond that which x points to
  - This is why pointers and array names are equivalent
- x + *n*: refers to the *n*th *type* object beyond x
- x – *n*: refers to the *n*th *type* object before x
- x – y: refers to the number of *type* objects between x and y
- x + y: meaningless!!!

Multidimensional Arrays

- A 2-dimensional array look like this:

x[0]	x[0][0]	x[0][1]	x[0][2]	x[0][3]
x[1]	x[1][0]	x[1][1]	x[1][2]	x[1][3]
x[2]	x[2][0]	x[2][1]	x[2][2]	x[2][3]

- Stored in row-major order as consecutive elements of a row are stored next to each other
  - Column-major order has consecutive elements of a column stored next to each other
- x[*i*] refers to row *i*

Initializations

- Initializing an array

```
int iarr[5] = { 1, 2, 3, 4, 5 };
```

or

```
int iarr[] = { 1, 2, 3, 4, 5 };
```

- Initializing a pointer

```
int ivar;
int *iptr = &ivar;
```

Strings

- An array of characters terminated with a 0 byte
  - 0 byte is a byte with all bits set to 0; also called a NUL byte
  - You can use either an array or a pointer
- Examples:

```
char carr[6] = { 'h', 'e’, 'l', 'l', 'o', '\0' };
char carr[] = { 'h', 'e’, 'l', 'l', 'o', '\0' };
char *cstr = "hello";
```

  - For the last, when a string (in "…") ends, the compiler adds a NUL byte

A Warning

- You want to make a copy of a string

```
char *cstr = "hello";
```

- Do *not* do this:

```
char *cdupstr;
. . .
cdupstr = cstr;
```

- This simply copies the *pointer*, so `cdupstr` and `cstr` point to the same string; if `cdupstr` is declared as an array, you get an error

Doing It Right

- You want to make a copy of a string

```
char *cstr = "hello";
char cdupstr[100];
```

  - Be sure `cdupstr` is an array with enough room to hold "hello" *plus the trailing NUL byte!*
- This works:

```
(void) strcpy(cdupstr, cstr);
```

- But this is better!

```
(void) strncpy(cdupstr, cstr, 99);
cdupstr[99] = '\0';
```

Reading a Line of Input

- Use `fgets(buf, n, stdin)`
  - On success, returns address of buf
  - On failure or EOF, if nothing has been read, returns a NULL pointer; otherwise, it returns all the characters read up to that error or the end of file
- Example use:

```
if (fgets(buf, 100, stdin) == NULL){
   fprintf(stderr, "Bad input\n"); . . .
```

  - If there is a new line, it reads up to that and *then* appends the '\0' byte
- Another way (but do *not* do this!)

```
if (gets(buf) == NULL){ fprintf(stderr, "Bad input\n"); . . . }
```

Command-Line Arguments

- Command is loopy 5 9
- Declaration of main function:

```
int main(int argc, char *argv[])
```

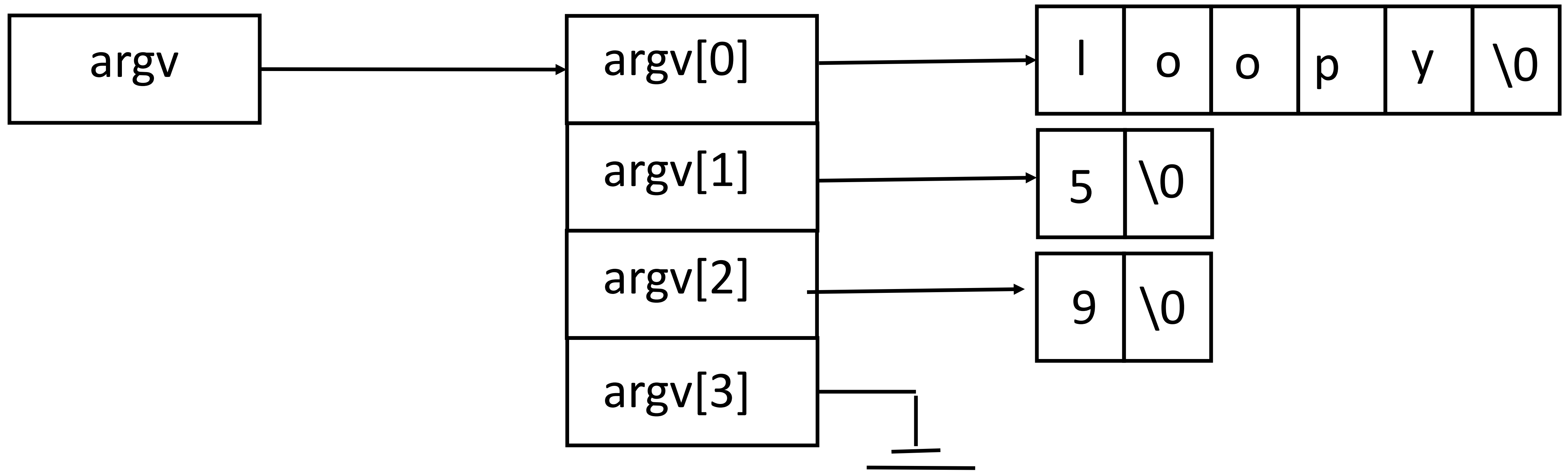
- Sometimes written as:

```
int main(int argc, char **argv)
```

number of arguments
(command is argument 0
So argc is always at least 1)

list of arguments
(in array of char pointers)

Visually:



Passing Strings as Arguments

- Function prototype:

```
void strfunc(char *, char *)
```

- Actual call (x, y are strings):

```
strfunc(x, y)
```

- Function definition header:

```
void strfunc(char *first, char *second){
```

String Idioms

- These mean the same thing when used as function arguments:

```
char *x
char x[]
```

Common Ways to “Walk Down” Strings

```
char *c = "hello";
char *cp = c;

while(*cp != '\0')
    printf("%c", *cp++);
printf("\n");
```

Another Idiom: Copy a String

```
char *c = "hello";
char cd[100];
char *cp = c;
char *cpd = cd;

while(*cpd++ = *cp++)
    ;
```

But . . .

- It’s better to use *strcpy* or *strncpy*
  - Because these may be faster, using assembly language optimizations
  - Also they are easier to understand!

Types of Characters

#include <ctype.h>

isprint(ch)	check for printing characters
isspace(ch)	check for space (for example, space, newline, tab)
isalpha(ch)	check for (capital or small) letter
isdigit(ch)	check for a digit ('0' … '9')
isalnum(ch)	same as isalpha(ch) || isdigit(ch)

- Note: ch is a character (technically, EOF or unsigned short int)
- Returns 0 if above check fails, non-zero if not

Converting Chars to Numbers

- Convert printing digit ch to integer

  `ch – '0'`

- Convert integer (between 0 and 9 inclusive) to printing char

  `ch + '0'`

- Find out which number a letter of the alphabet is

  `ch – 'a'` (for lower case), `ch – 'A'` (for upper case)

- Find out which letter of the alphabet a number between 0 and 25 inclusive) is

  `ch + 'a'` (for lower case), `ch + 'A'` (for upper case)