Systems and Internet Infrastructure Security

Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Strings

Devin J. Pohly <djpohly@cse.psu.edu>

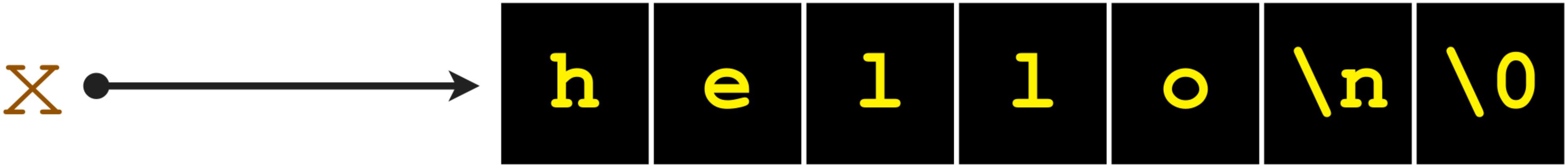
Strings are arrays…

- C handles ASCII text through *strings*
- Standard library functions for managing strings are found in `<string.h>`
- A string is just an array of characters
    - And arrays are… (all together now)

… which are pointers!

- Variable just points to first character
  - No length information stored
- Strings are *null-terminated*: the end is denoted by the null character `'\0'`
  - Remember this requires an extra byte!

```
// All of these are equivalent
char *x0 = "hello\n";
char x1[] = "hello\n";
char x2[7] = "hello\n";   // Why 7?
```



ASCII

- American Standard Code for Information Interchange

0	NUL	16	DLE	32	SPC	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	TAB	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

```c
char a = 65;
printf("Decimal %d is ASCII '%c'\n", a, a);
```

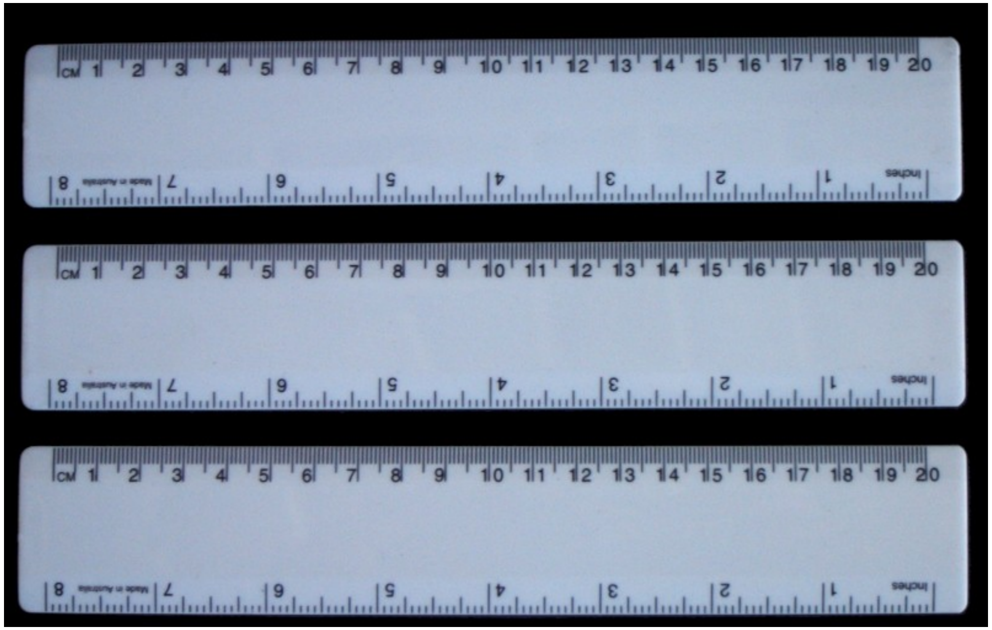
```
Decimal 65 is ASCII 'A'
```

String length

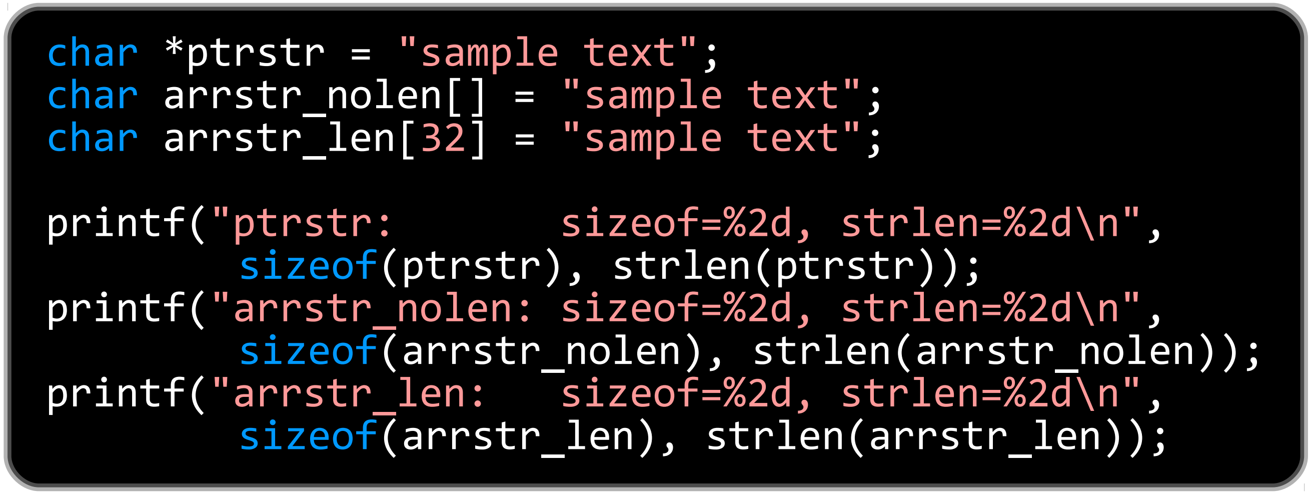
- C library function `strlen(str)` returns the number of characters before the null terminator
- Be careful with `sizeof`: it may not do what you want!
  - Can return the size of the array or the size of a pointer!
  - Depends on how and where you use it
  - Does not change if you modify the string!

Aside: arrays and `sizeof`

- Use **caution** with `sizeof` and arrays
  - Including strings!
  - Compiler only knows the array size if the array declaration is in scope
  - Otherwise it has already been converted to a pointer
    - `sizeof` will just give you the size of a pointer
    - E.g., as a function argument
- Easy, lazy rule: just avoid it



sizeof vs. strlen



```c
char *ptrstr = "sample text";
char arrstr_nolen[] = "sample text";
char arrstr_len[32] = "sample text";

printf("ptrstr:       sizeof=%2d, strlen=%2d\n",
        sizeof(ptrstr), strlen(ptrstr));
printf("arrstr_nolen: sizeof=%2d, strlen=%2d\n",
        sizeof(arrstr_nolen), strlen(arrstr_nolen));
printf("arrstr_len:   sizeof=%2d, strlen=%2d\n",
        sizeof(arrstr_len), strlen(arrstr_len));
```

- What will this print?

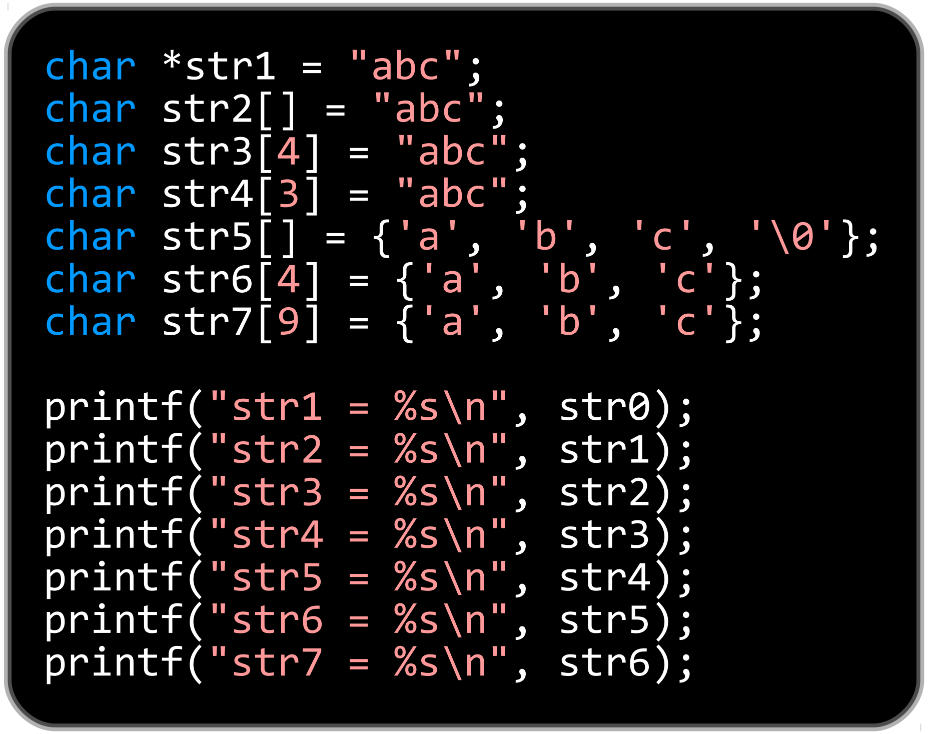
sizeof vs. strlen

```
char *ptrstr = "sample text";
char arrstr_nolen[] = "sample text";
char arrstr_len[32] = "sample text";

printf("ptrstr:       sizeof=%2d, strlen=%2d\n",
        sizeof(ptrstr), strlen(ptrstr));
printf("arrstr_nolen: sizeof=%2d, strlen=%2d\n",
        sizeof(arrstr_nolen), strlen(arrstr_nolen));
printf("arrstr_len:   sizeof=%2d, strlen=%2d\n",
        sizeof(arrstr_len), strlen(arrstr_len));
```

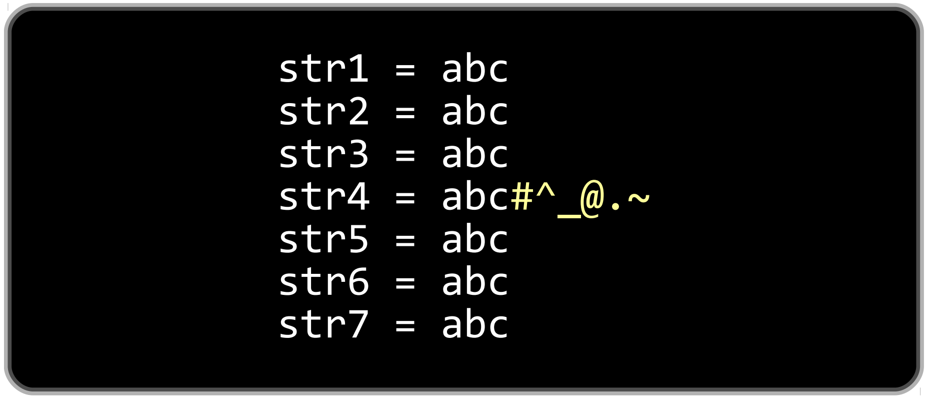
```
ptrstr:       sizeof= 8, strlen=11
arrstr_nolen: sizeof=12, strlen=11
arrstr_len:   sizeof=32, strlen=11
```

- Notice:
  - `strlen` always gives the same result, and it excludes the null terminator. (It is calculated at run-time.)
  - `sizeof` is unrelated to the contents of the string, and it includes the null terminator if any. (It is calculated by the compiler.)

Initializing strings



```
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = "abc";
char str4[3] = "abc";
char str5[] = {'a', 'b', 'c', '\0'};
char str6[4] = {'a', 'b', 'c'};
char str7[9] = {'a', 'b', 'c'};

printf("str1 = %s\n", str0);
printf("str2 = %s\n", str1);
printf("str3 = %s\n", str2);
printf("str4 = %s\n", str3);
printf("str5 = %s\n", str4);
printf("str6 = %s\n", str5);
printf("str7 = %s\n", str6);
```



```
str1 = abc
str2 = abc
str3 = abc
str4 = abc#^_@.~
str5 = abc
str6 = abc
str7 = abc
```

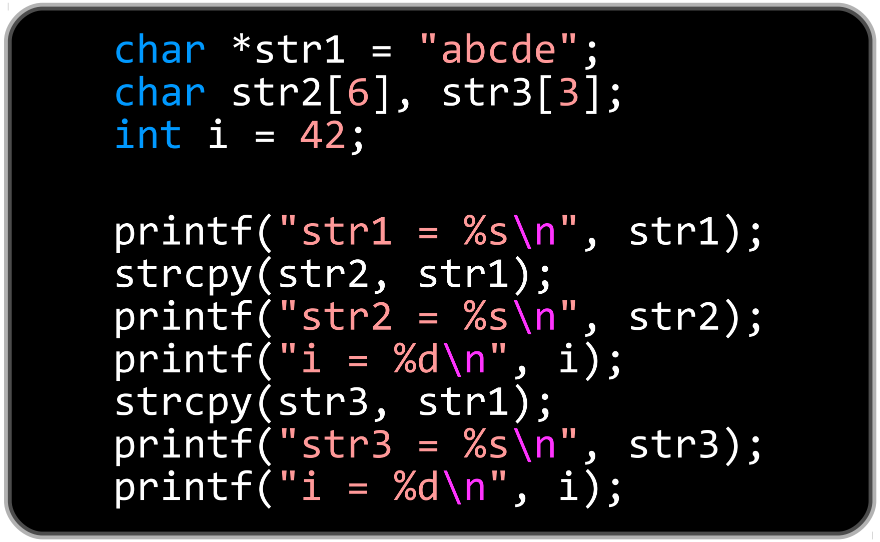
- All of these work except for `str4`
- Why?
  - The array declaration did not leave space for a null terminator!
  - So there is no `'\0'` at the end of the string
  - This is called an *unterminated string*
  - Which can cause **bad, scary things** to happen!
    - So don't do it!

Copying strings

- The `strcpy` function copies bytes from one string to another
    - It searches for the null terminator and copies everything up to that point, plus the terminator
    - Copying from “source” string to “destination” string:

      `strcpy(dest, src);`

    - Mnemonic: the order is the same as `dest = src;`.



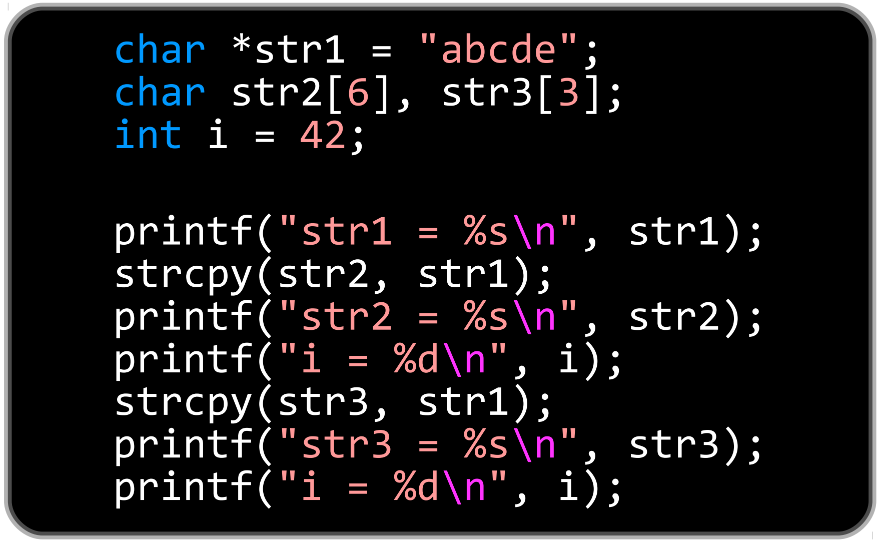
```
char *str1 = "abcde";
char str2[6], str3[3];
int i = 42;

printf("str1 = %s\n", str1);
strcpy(str2, str1);
printf("str2 = %s\n", str2);
printf("i = %d\n", i);
strcpy(str3, str1);
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```

Copying strings

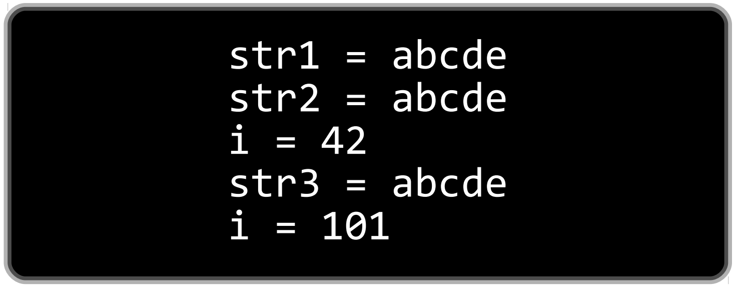
- The `strcpy` function copies bytes from one string to another
  - It searches for the null terminator and copies everything up to that point, plus the terminator
  - Copying from “source” string to “destination” string:

    `strcpy(dest, src);`

  - Mnemonic: the order is the same as `dest = src;`.



```c
char *str1 = "abcde";
char str2[6], str3[3];
int i = 42;

printf("str1 = %s\n", str1);
strcpy(str2, str1);
printf("str2 = %s\n", str2);
printf("i = %d\n", i);
strcpy(str3, str1);
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```



```
str1 = abcde
str2 = abcde
i = 42
str3 = abcde
i = 101
```

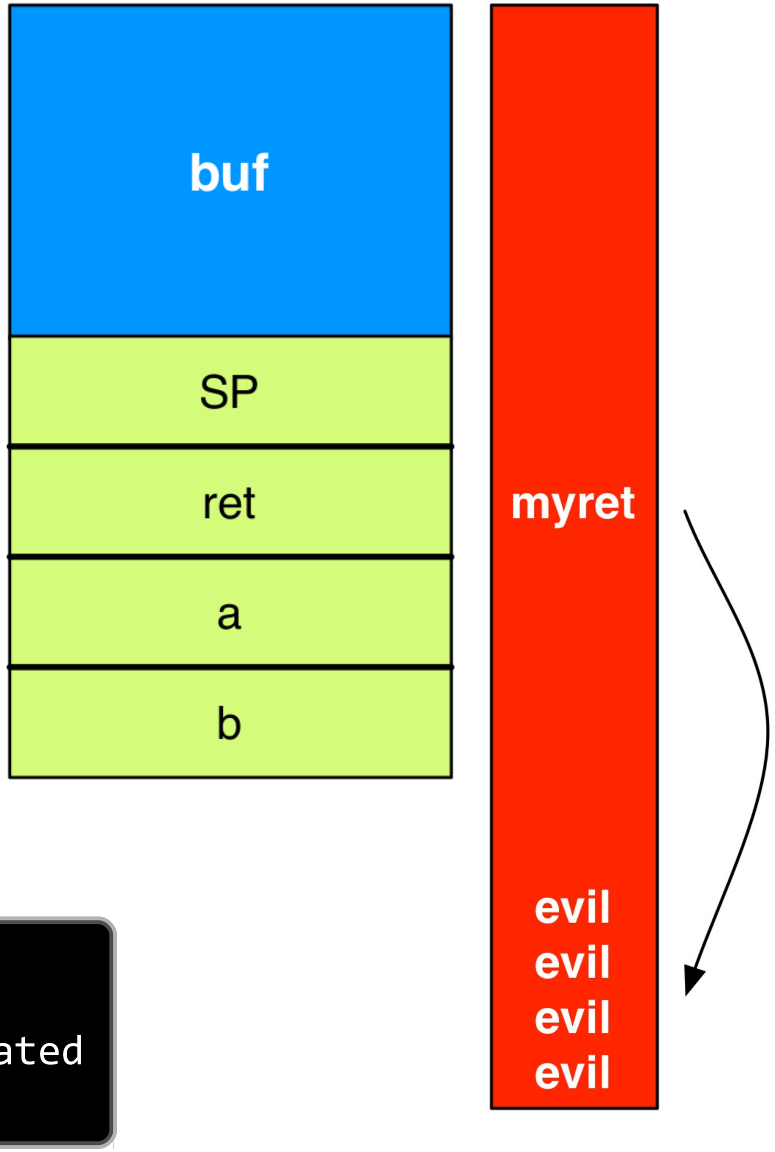
Wait, what??

Buffer overflows

- A buffer overflow is when you overwrite data outside the buffer (on the stack)
  - Specifically the *return address*
  - Under adversary control, this can take over the process

```
    char buf[5];

    printf("Enter some text:\n");
    scanf("%s", buf);
```

```
Enter some text:
oeuaoeuaoeuoaeuaoeuaoeuaoeuaoeuoaeuaoeuaoeuoe
*** stack smashing detected***: ./test terminated
Segmentation fault (core dumped)
```

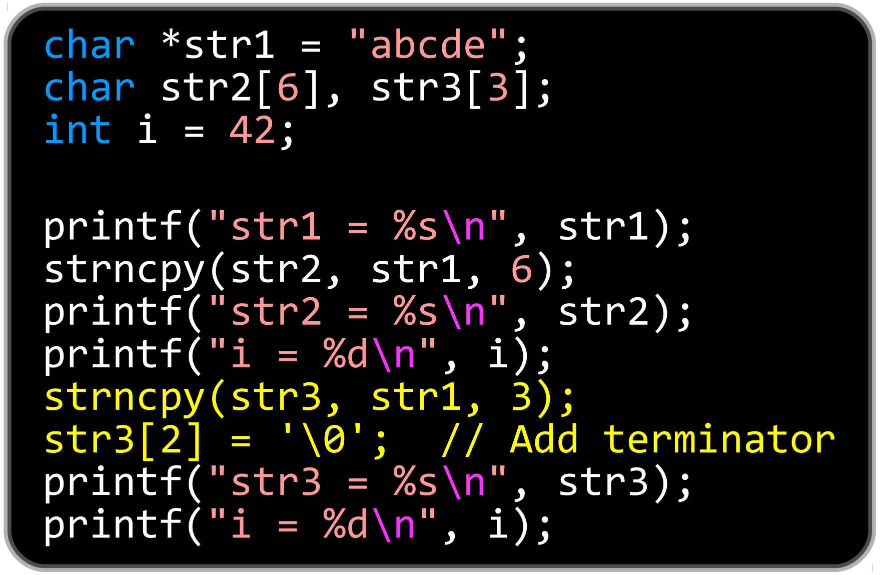


“n” variants

- To work safely with strings, use the “n” variants of the string functions.
  - These take an extra parameter: the maximum number of bytes to copy
  - For example, to copy a string safely:

```
strncpy(dest, src, n);
```

  - Natural choice for n is the buffer size of the destination!
    - **Caution**: if we hit this maximum, the destination string will not be terminated!



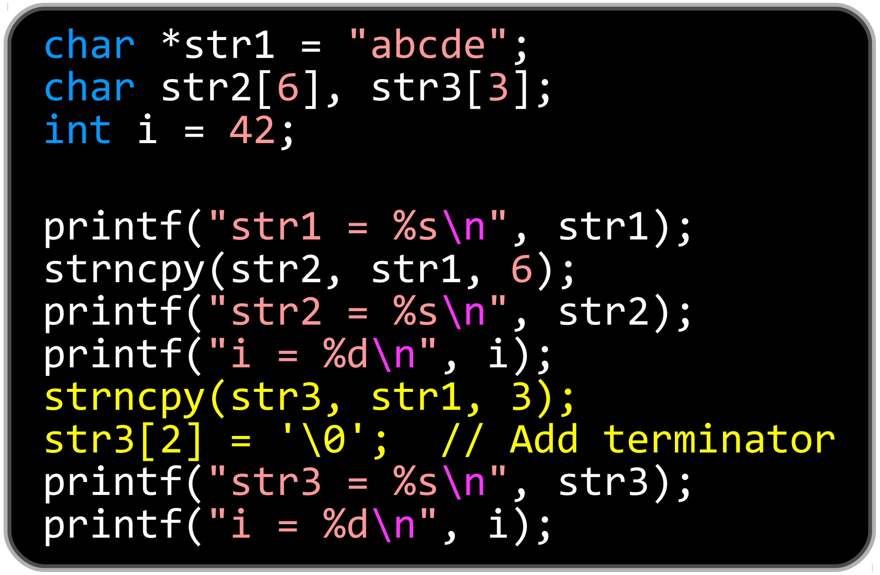
```
char *str1 = "abcde";
char str2[6], str3[3];
int i = 42;

printf("str1 = %s\n", str1);
strncpy(str2, str1, 6);
printf("str2 = %s\n", str2);
printf("i = %d\n", i);
strncpy(str3, str1, 3);
str3[2] = '\0';  // Add terminator
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```

“n” variants

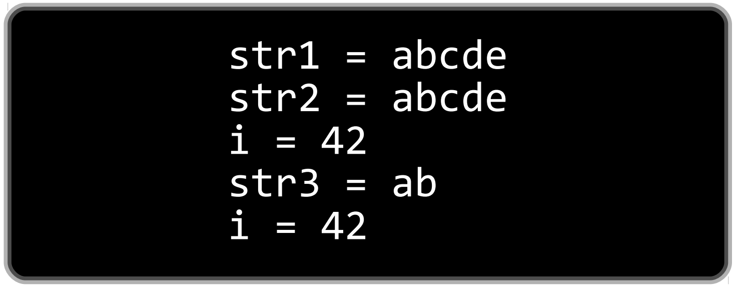
- To work safely with strings, use the “n” variants of the string functions.
  - These take an extra parameter: the maximum number of bytes to copy
  - For example, to copy a string safely:

```
strncpy(dest, src, n);
```

  - Natural choice for n is the buffer size of the destination!
    - **Caution**: if we hit this maximum, the destination string will not be terminated!



```
char *str1 = "abcde";
char str2[6], str3[3];
int i = 42;

printf("str1 = %s\n", str1);
strncpy(str2, str1, 6);
printf("str2 = %s\n", str2);
printf("i = %d\n", i);
strncpy(str3, str1, 3);
str3[2] = '\0';  // Add terminator
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```



```
str1 = abcde
str2 = abcde
i = 42
str3 = ab
i = 42
```

Concatenating strings

- Often we want to put two strings together to make one long string
  - In C++, the + operator was overloaded for this
  - In C, we use the `strcat` function to append src to dest:

```
strcat(dest, src);
```

  - The `strncat` variant copies at most n bytes of src:

```
strncat(dest, src, n);
```

String comparison

- We often want to compare strings to see if they match or are lexicographically smaller or larger
- In C, we use `strcmp` (which compares s1 to s2):

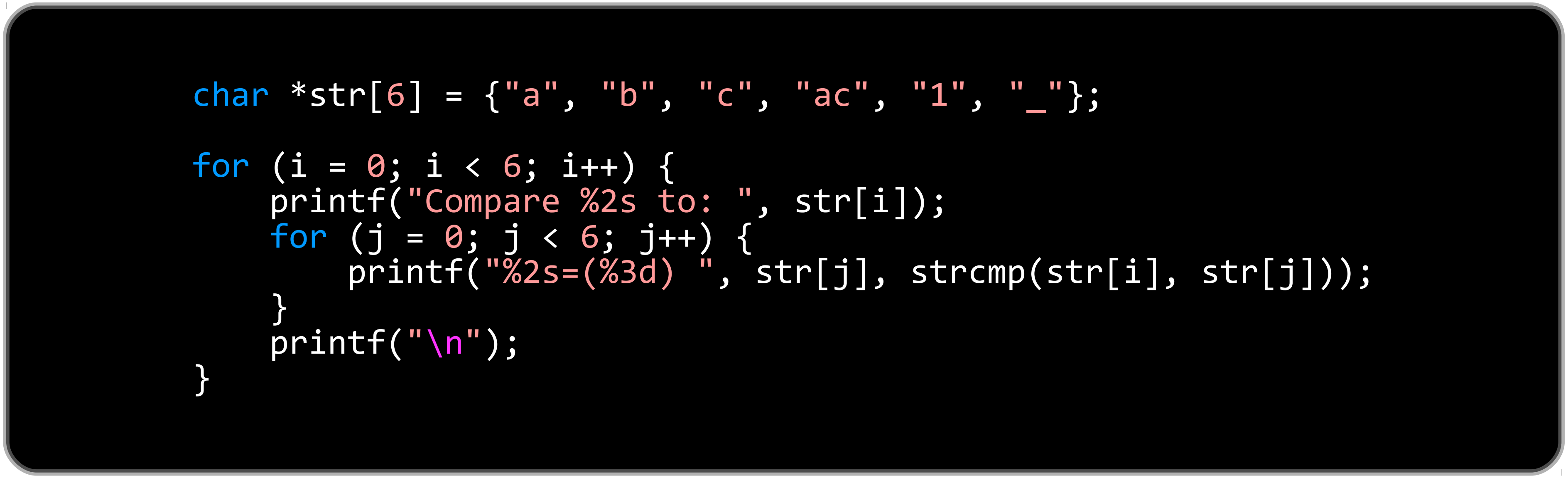
```
strcmp(s1, s2);
```

- `strncmp` compares first n bytes of strings:

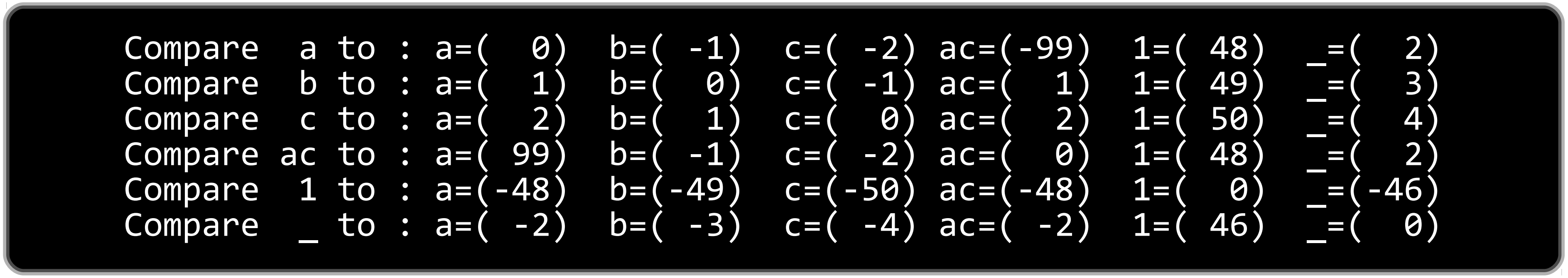
```
strncmp(s1, s2, n);
```

- The comparison functions return
  - negative integer if s1 is less than s2
  - 0 if s1 is equal to s2
  - positive integer if s1 is greater than s2

Lexicographical order



```
char *str[6] = {"a", "b", "c", "ac", "1", "_"};

for (i = 0; i < 6; i++) {
    printf("Compare %2s to: ", str[i]);
    for (j = 0; j < 6; j++) {
        printf("%2s=(%3d) ", str[j], strcmp(str[i], str[j]));
    }
    printf("\n");
}
```



```
Compare  a to : a=(  0)  b=( -1)  c=( -2) ac=(-99)  1=( 48)  _=(  2)
Compare  b to : a=(  1)  b=(  0)  c=( -1) ac=(  1)  1=( 49)  _=(  3)
Compare  c to : a=(  2)  b=(  1)  c=(  0) ac=(  2)  1=( 50)  _=(  4)
Compare ac to : a=( 99)  b=( -1)  c=( -2) ac=(  0)  1=( 48)  _=(  2)
Compare  1 to : a=(-48)  b=(-49)  c=(-50) ac=(-48)  1=(  0)  _=(-46)
Compare  _ to : a=( -2)  b=( -3)  c=( -4) ac=( -2)  1=( 46)  _=(  0)
```

Searching strings

- Often we want to search through strings to find something we are looking for:
  - `strchr` searches front-to-back for a character
  - `strrchr` searches back-to-front for a character

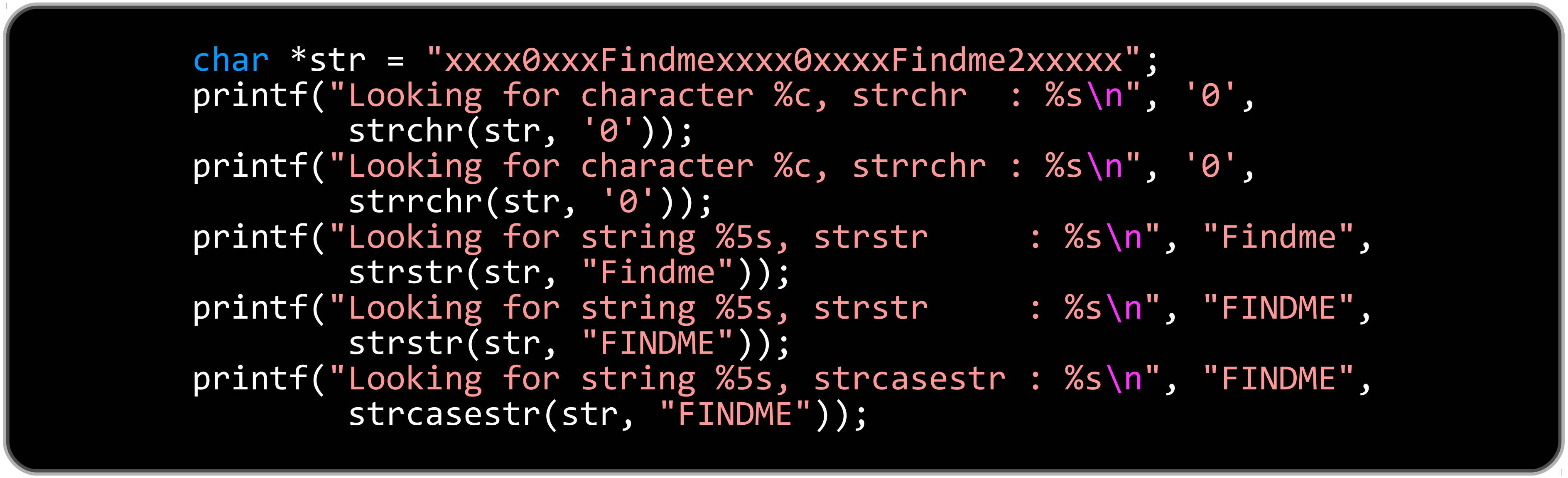
```
strchr(str, ch);
strrchr(str, ch);
```

  - `strstr` searches for a substring
  - `strcasestr` searches for a substring ignoring case

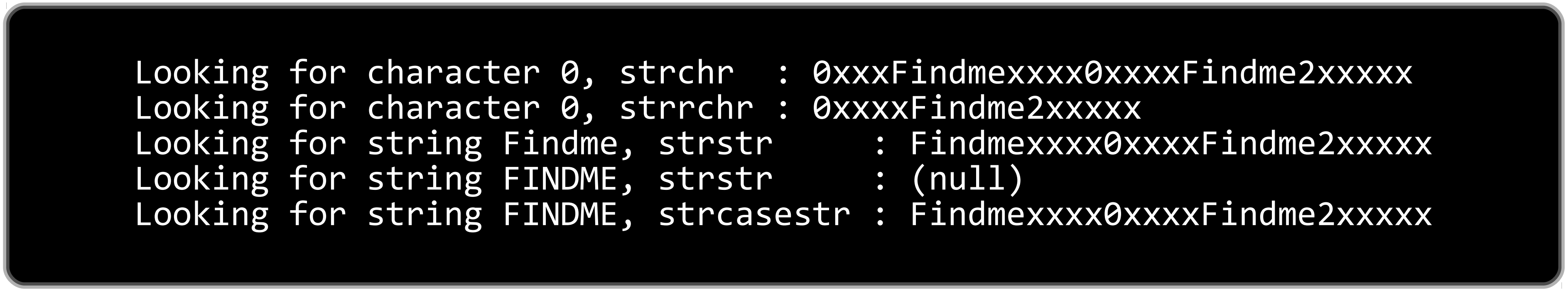
```
strstr(haystack, needle);
strcasestr(haystack, needle);
```

- All of these functions return a pointer to the found value within the string, or NULL if not found.

Example searches



```c
char *str = "xxxx0xxxFindmexxxx0xxxxFindme2xxxxx";
printf("Looking for character %c, strchr  : %s\n", '0',
        strchr(str, '0'));
printf("Looking for character %c, strrchr : %s\n", '0',
        strrchr(str, '0'));
printf("Looking for string %5s, strstr     : %s\n", "Findme",
        strstr(str, "Findme"));
printf("Looking for string %5s, strstr     : %s\n", "FINDME",
        strstr(str, "FINDME"));
printf("Looking for string %5s, strcasestr : %s\n", "FINDME",
        strcasestr(str, "FINDME"));
```



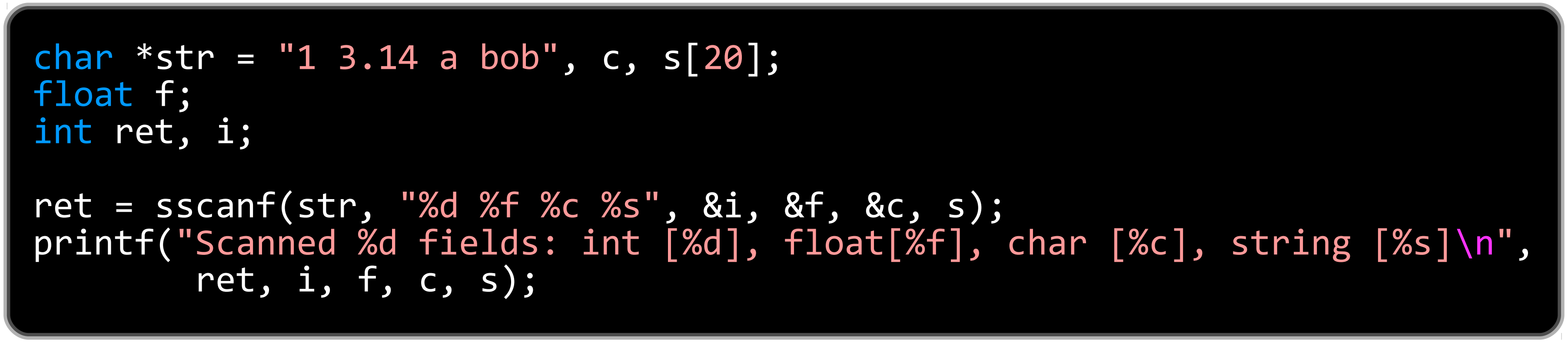
```
Looking for character 0, strchr  : 0xxxFindmexxxx0xxxxFindme2xxxxx
Looking for character 0, strrchr : 0xxxxFindme2xxxxx
Looking for string Findme, strstr     : Findmexxxx0xxxxFindme2xxxxx
Looking for string FINDME, strstr     : (null)
Looking for string FINDME, strcasestr : Findmexxxx0xxxxFindme2xxxxx
```

Parsing strings

- Strings carry information we want to parse (break down into separate variables)
- In C, we use `sscanf` to extract data by format:

```
sscanf(str, "format", ...);
```

- The syntax is very similar to that of `printf`, but your arguments must be passed by reference
  - Returns the number successfully parsed



```
char *str = "1 3.14 a bob", c, s[20];
float f;
int ret, i;

ret = sscanf(str, "%d %f %c %s", &i, &f, &c, s);
printf("Scanned %d fields: int [%d], float[%f], char [%c], string [%s]\n",
        ret, i, f, c, s);
```

Tokenizing strings

- Input is often in a form ready for parsing, such as the CSV (comma-separated value) format:

```
Devin,Pohly,CMPSC311,Instructor
Junpeng,Qiu,CMPSC311,TA
Prashanth,Thinakaran,CMPSC311,TA
```

- We want to be able to pull the data apart so we can process it
  - "Tokenize": each field is a "token"
  - We use the `strtok` function:

```
strtok(str, delim);
```

  - On first run pass the string to parse, then pass NULL

Tokenizing example

```c
char *ptr, *nptr, *input[3] = {
    "Devin,Pohly,CMPSC311,Instructor",
    "Junpeng,Qiu,CMPSC311,TA",
    "Prashanth,Thinakaran,CMPSC311,TA",
};

for (i = 0; i < 3; i++) {
    // Duplicate the string (avoid modifying original)
    nptr = strdup(input[i]);

    // First time, supply the string to parse
    ptr = strtok(nptr, ",");
    while (ptr != NULL) {
        // On subsequent calls, pass NULL
        printf("Next token [%s]\n", ptr);
        ptr = strtok(NULL, ",");
    }
    printf(" -- no more tokens\n");
    free(nptr);
}
```

```
Next token [Devin]
Next token [Pohly]
Next token [CMPSC311]
Next token [Instructor]
 -- no more tokens
Next token [Junpeng]
Next token [Qiu]
Next token [CMPSC311]
Next token [TA]
 -- no more tokens
Next token [Prashanth]
Next token [Thinakaran]
Next token [CMPSC311]
Next token [TA]
 -- no more tokens
```