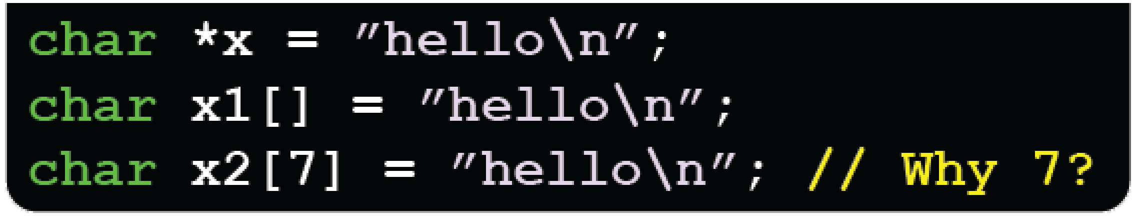
Strings and Commonly used String Functions in C

Outline

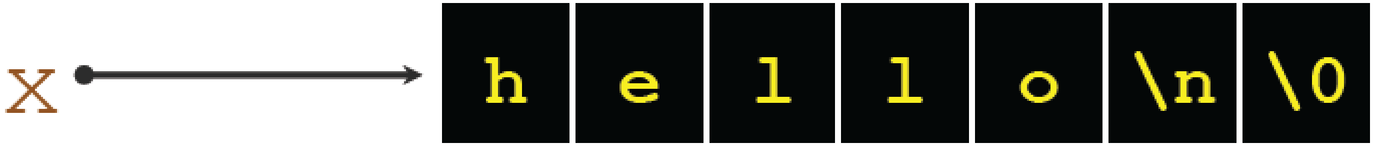
- String implementation in C
- Reading and Printing Strings
- Frequently used <string.h> library functions

A string is just an array ...

- C handles ASCII text through *strings*
- A string is just an array of characters
    - Which is just pointed by a pointer



```
char *x = "hello\n";
char x1[] = "hello\n";
char x2[7] = "hello\n"; // Why 7?
```

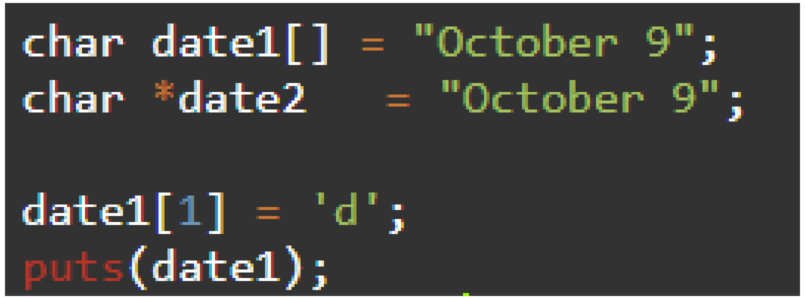


- There are a large number of interfaces for managing strings available in the C library, i.e., string.h.

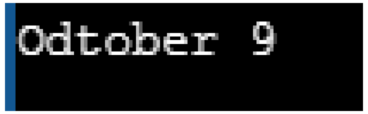
How you declare matters

1. char date [] = “October 9”;

- Date is an array and stored in stack
- Characters stored in date can be modified



```
char date1[] = "October 9";
char *date2    = "October 9";

date1[1] = 'd';
puts(date1);
```

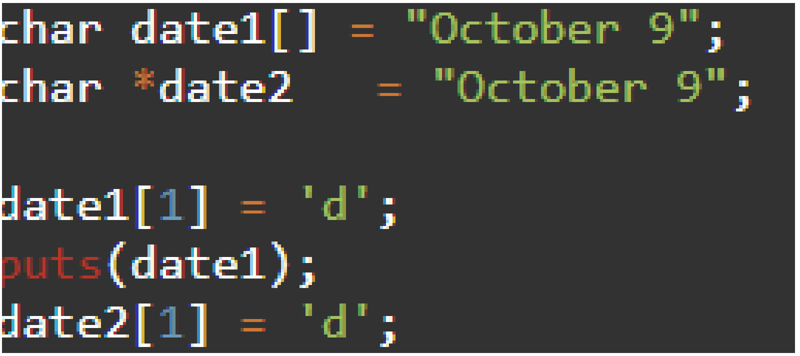


```
Odtober 9
```

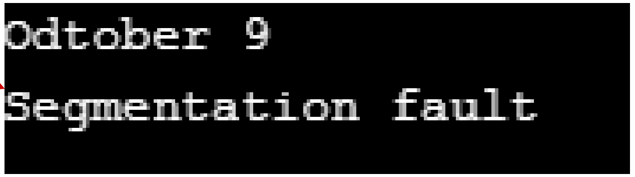
2. char *date= “October 9”;

- Date is a pointer, and points to a literal
- String literals cannot be modified.

https://onlinegdb.com/H1i_gYodB



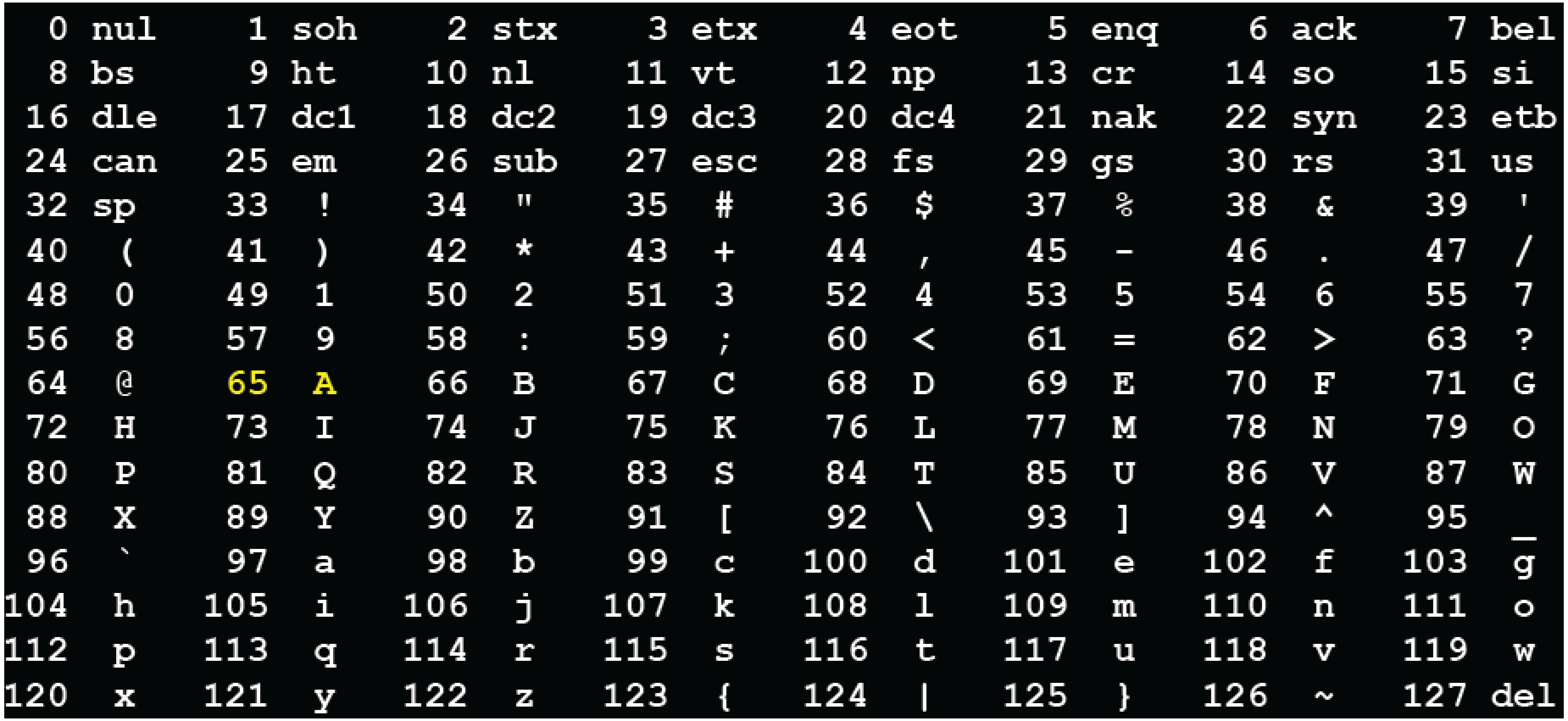
```
char date1[] = "October 9";
char *date2    = "October 9";

date1[1] = 'd';
puts(date1);
date2[1] = 'd';
```



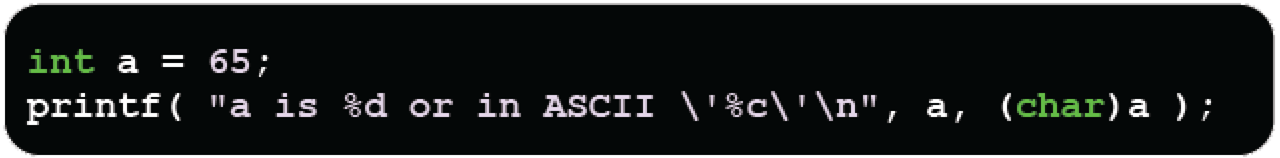
```
Odtober 9
Segmentation fault
```

ASCII

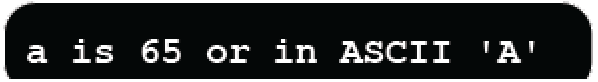
- American Standard Code for Information Interchange



0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del



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



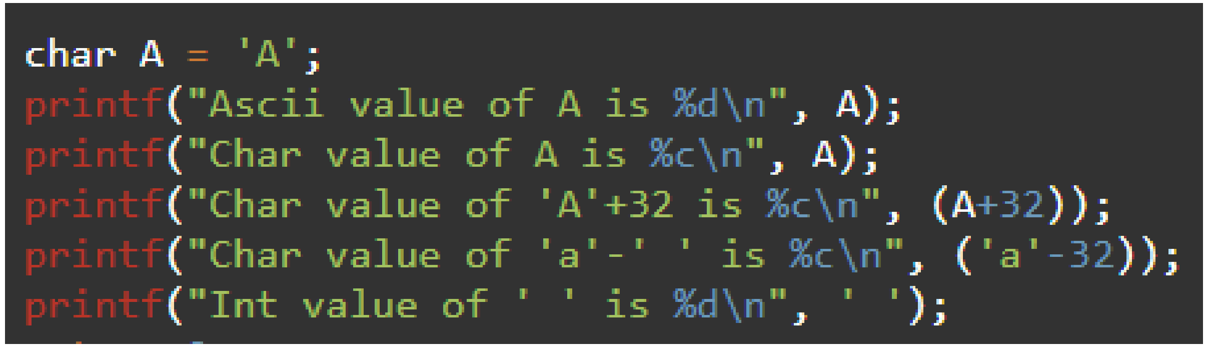
```
a is 65 or in ASCII 'A'
```

Convert ACII lowercase to uppercase

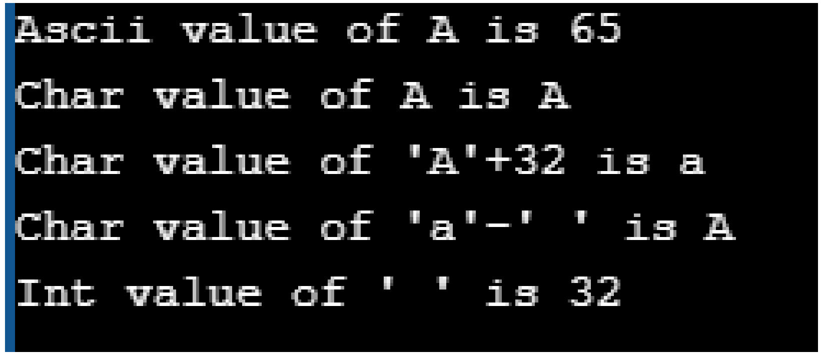
Difference between Uppercase ACII and lowercase is 0x20 (32 in decimal) or space, or decimal 32

uppercase = a - 32

lowercase = A + 32



```
char A = 'A';
printf("Ascii value of A is %d\n", A);
printf("Char value of A is %c\n", A);
printf("Char value of 'A'+32 is %c\n", (A+32));
printf("Char value of 'a'-' ' is %c\n", ('a'-32));
printf("Int value of ' ' is %d\n", ' ');
```

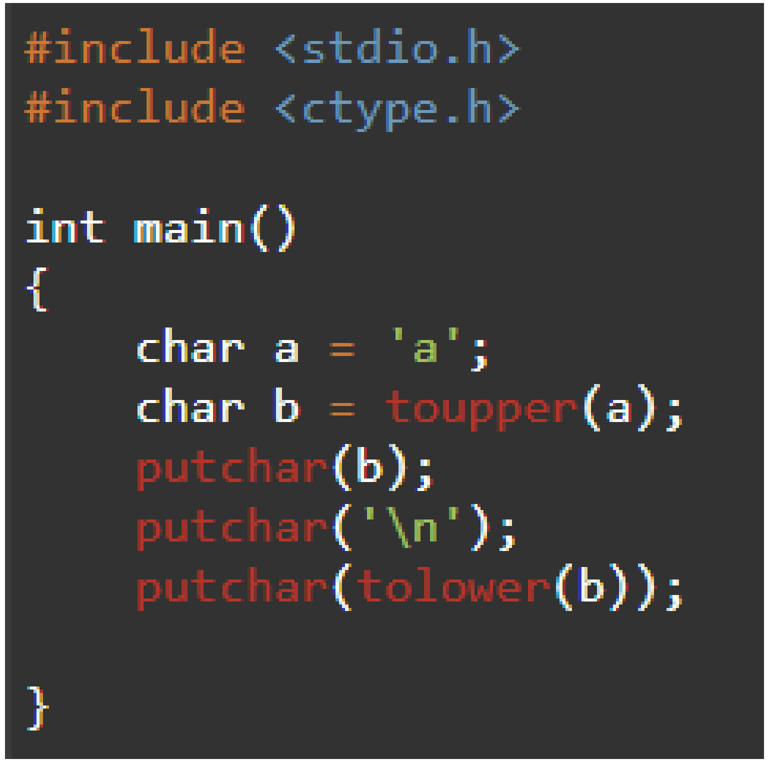


```
Ascii value of A is 65
Char value of A is A
Char value of 'A'+32 is a
Char value of 'a'-' ' is A
Int value of ' ' is 32
```

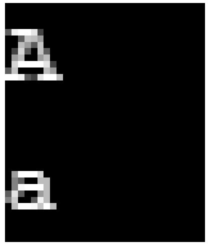
https://onlinegdb.com/r1H3R79_r

Conversion to uppercase and lowercase using ctype.h

- int toupper(int ch) -> return uppercase
- int tolower(int ch) -> return lowercase

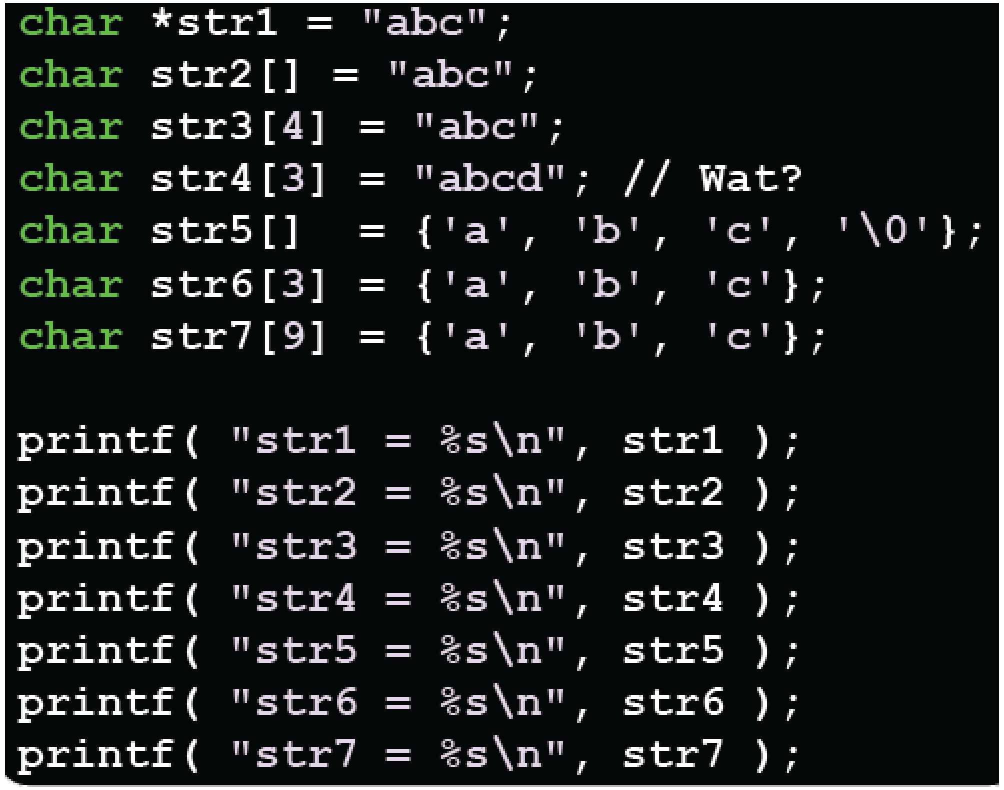


```
#include <stdio.h>
#include <ctype.h>

int main()
{
    char a = 'a';
    char b = toupper(a);
    putchar(b);
    putchar('\n');
    putchar(tolower(b));

}
```

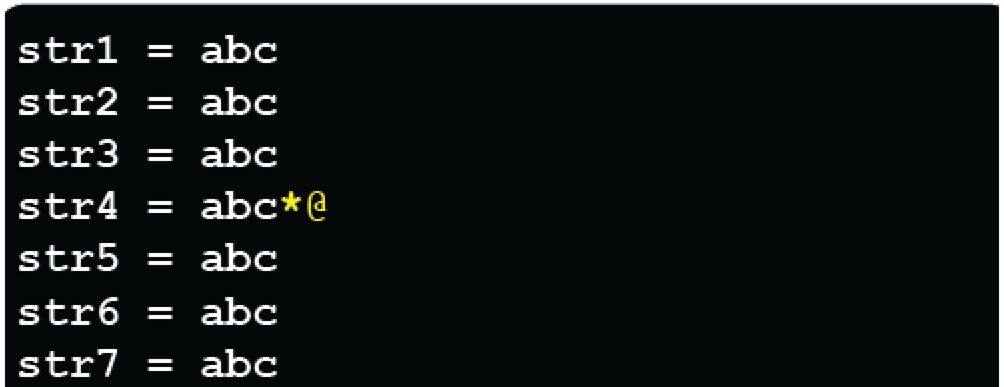


```
A
a
```

Initializing strings ...



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

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



```
str1 = abc
str2 = abc
str3 = abc
str4 = abc*@
str5 = abc
str6 = abc
str7 = abc
```

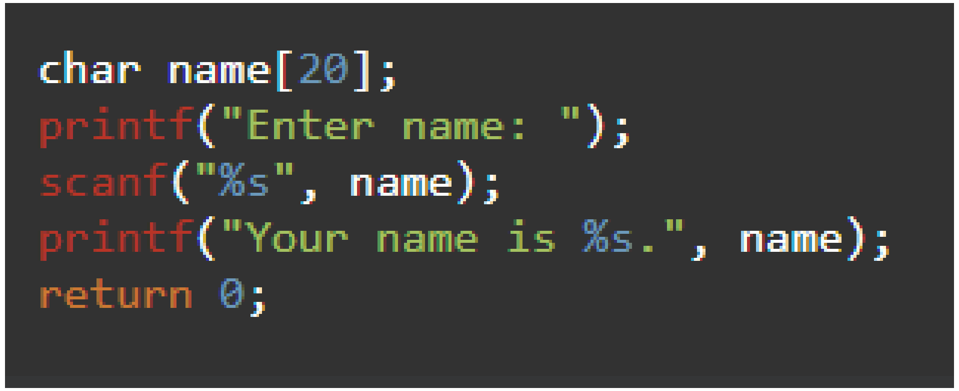
- All legitimate except
  str4 str6
- The bad strings have no NULL terminator
  - This is called an *unterminated string*
  - Big, scary things can happen when you work with unterminated strings (don’t do it).
- Str7, characters after c will be initialized to 0. So automatically null terminated

Outline

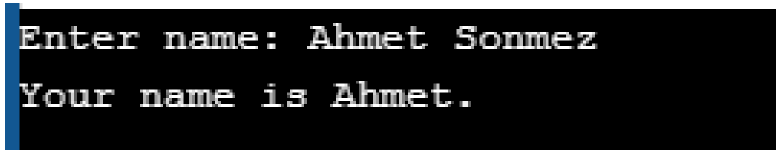
- String implementation in C
- Reading and Printing Strings
- Frequently used <string.h> library functions

Reading and printing strings

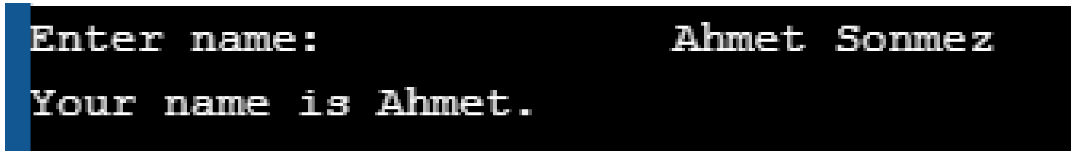
- Using scanf and %s format specifier



```
char name[20];
printf("Enter name: ");
scanf("%s", name);
printf("Your name is %s.", name);
return 0;
```



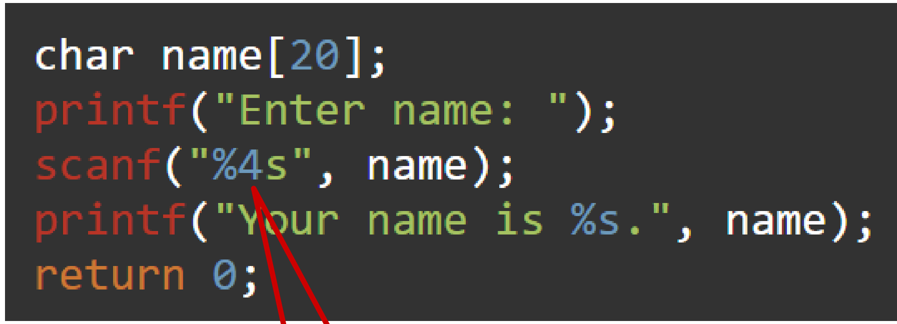
```
Enter name: Ahmet Sonmez
Your name is Ahmet.
```



```
Enter name:           Ahmet Sonmez
Your name is Ahmet.
```

https://onlinegdb.com/H1jRAUi_S

- Will read until the whitespace
- Will skip the whitespace before the first word

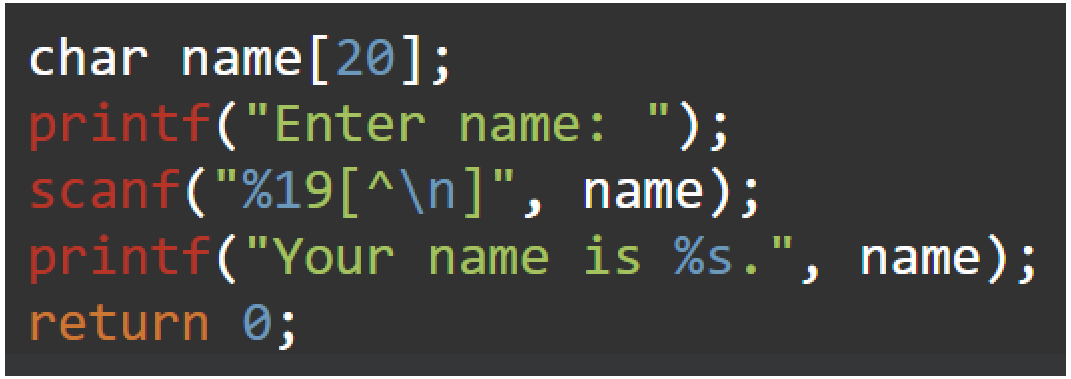


```
char name[20];
printf("Enter name: ");
scanf("%4s", name);
printf("Your name is %s.", name);
return 0;
```

A better way to prevent memory smashing

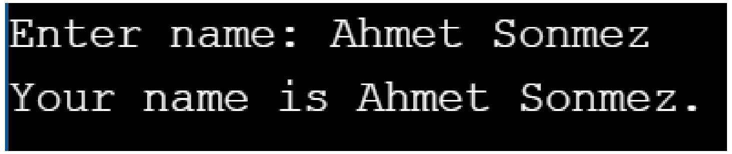
Reading and printing strings

- Using scanf and %[] format specifier

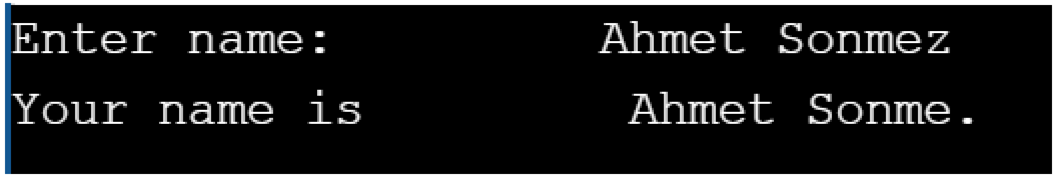


```c
char name[20];
printf("Enter name: ");
scanf("%19[^\n]", name);
printf("Your name is %s.", name);
return 0;
```

- Will read up to 19 characters
- Will read until reaching the next line character



```
Enter name: Ahmet Sonmez
Your name is Ahmet Sonmez.
```



```
Enter name:         Ahmet Sonmez
Your name is         Ahmet Sonme.
```

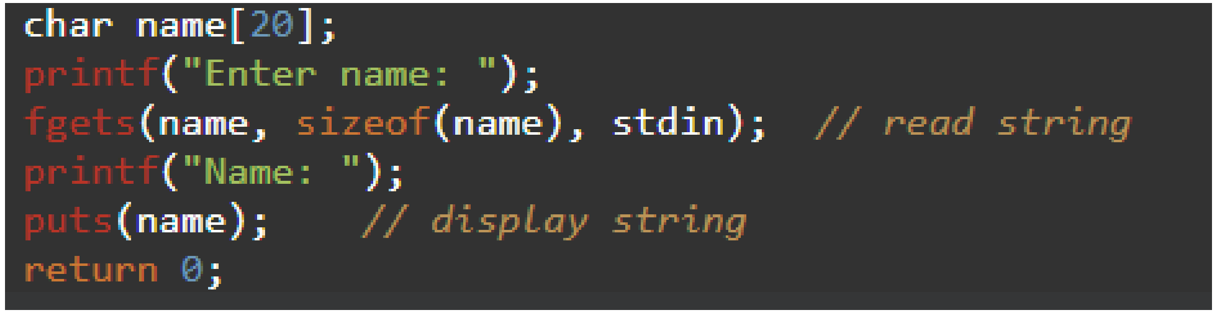
Refer to man pages for scanf for more on %[format specifier

https://www.man7.org/linux/man-pages/man3/scanf.3.html

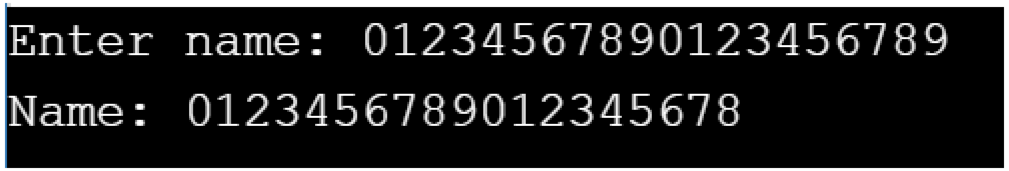
https://onlinegdb.com/B1p84k22U

Reading and printing strings

- Using gets, fgets and puts, to read or print full line



```
char name[20];
printf("Enter name: ");
fgets(name, sizeof(name), stdin);  // read string
printf("Name: ");
puts(name);    // display string
return 0;
```



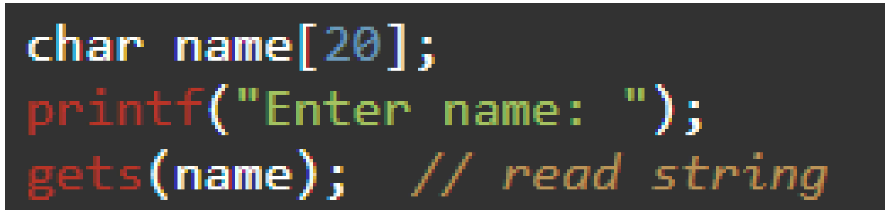
```
Enter name: 01234567890123456789
Name: 0123456789012345678
```

- Will read a single line but limited to the sizeof (name)
- Will allocate one character space at the end for null character
- Will not skip whitespaces

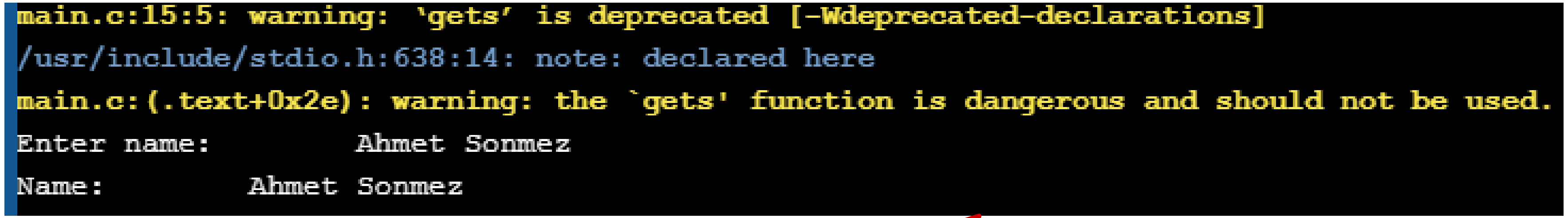
https://onlinegdb.com/SJe61XDj_H

Reading string input using gets

- gets is deprecated and dangerous to use
- Vulnerable to Buffer Overflow attacks



```
char name[20];
printf("Enter name: ");
gets(name);  // read string
```



```
main.c:15:5: warning: ‘gets’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.c:(.text+0x2e): warning: the `gets' function is dangerous and should not be used.
Enter name:        Ahmet Sonmez
Name:        Ahmet Sonmez
```

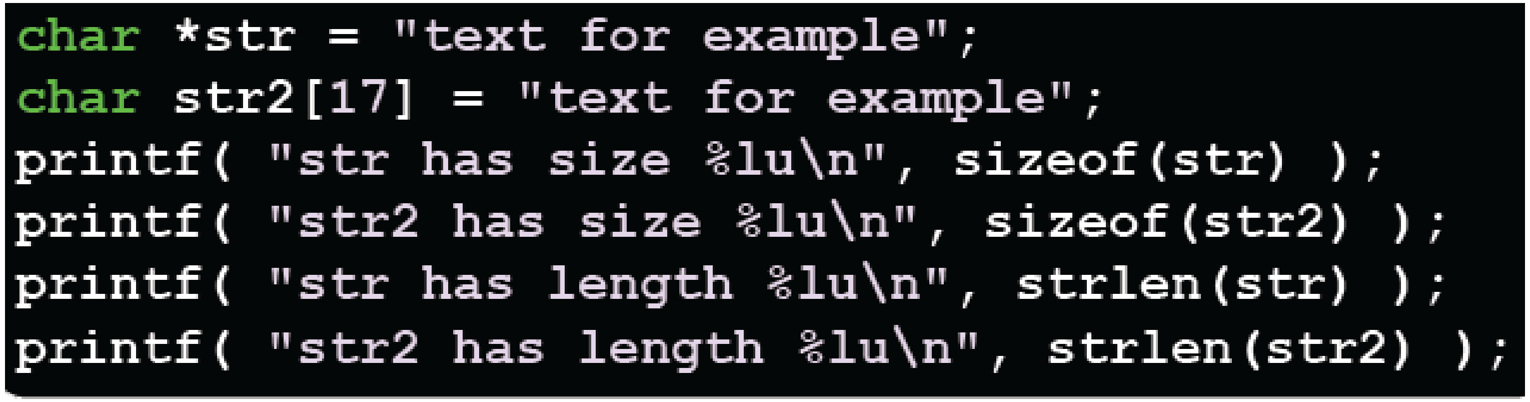
Don’t use gets!

Outline

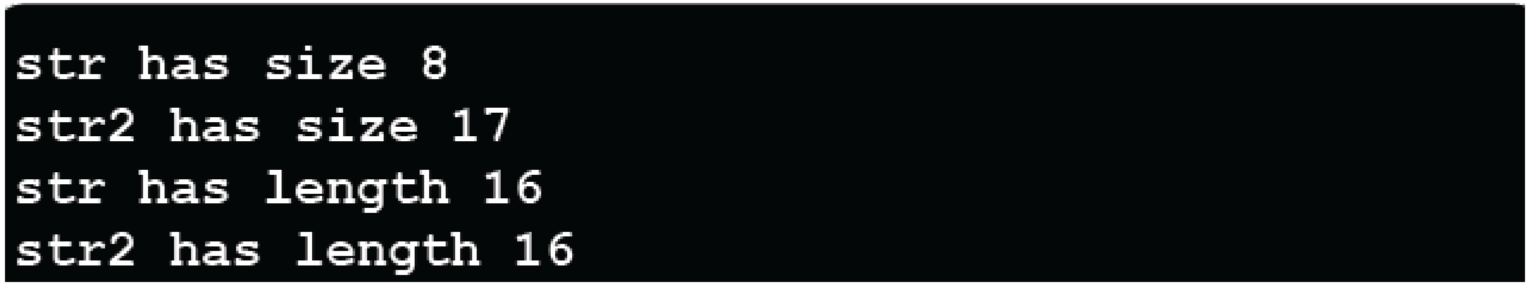
- String implementation in C
- Reading and Printing Strings
- Frequently used <string.h> library functions

sizeof vs strlen

- There are two ways of determining the “size” of the string, each with their own semantics
  - ▸ sizeof(string) returns the size of the declaration
    (sometimes beware: for literals returns pointer size.)
  - ▸ strlen(string) returns number of characters until hitting ‘\0’. (null terminator can be anywhere within)



```
char *str = "text for example";
char str2[17] = "text for example";
printf( "str has size %lu\n", sizeof(str) );
printf( "str2 has size %lu\n", sizeof(str2) );
printf( "str has length %lu\n", strlen(str) );
printf( "str2 has length %lu\n", strlen(str2) );
```



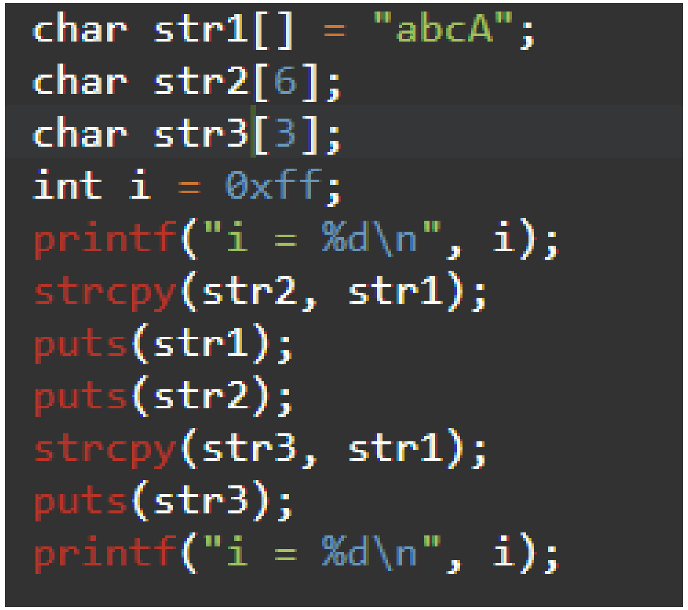
```
str has size 8
str2 has size 17
str has length 16
str2 has length 16
```

Copying strings

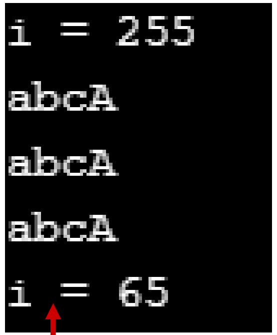
- strcpy allows you to copy one string to another
  - It searches NULL terminator and copies everything up to that point, plus the terminator
  - Copy from “source” string to “destination” string

    strcpy(dest, src) is kinda like dest = src

- strcpy has no size control, so vulnerable to buffer overflow.



```c
char str1[] = "abcA";
char str2[6];
char str3[3];
int i = 0xff;
printf("i = %d\n", i);
strcpy(str2, str1);
puts(str1);
puts(str2);
strcpy(str3, str1);
puts(str3);
printf("i = %d\n", i);
```



```
i = 255
abcA
abcA
abcA
i = 65
```

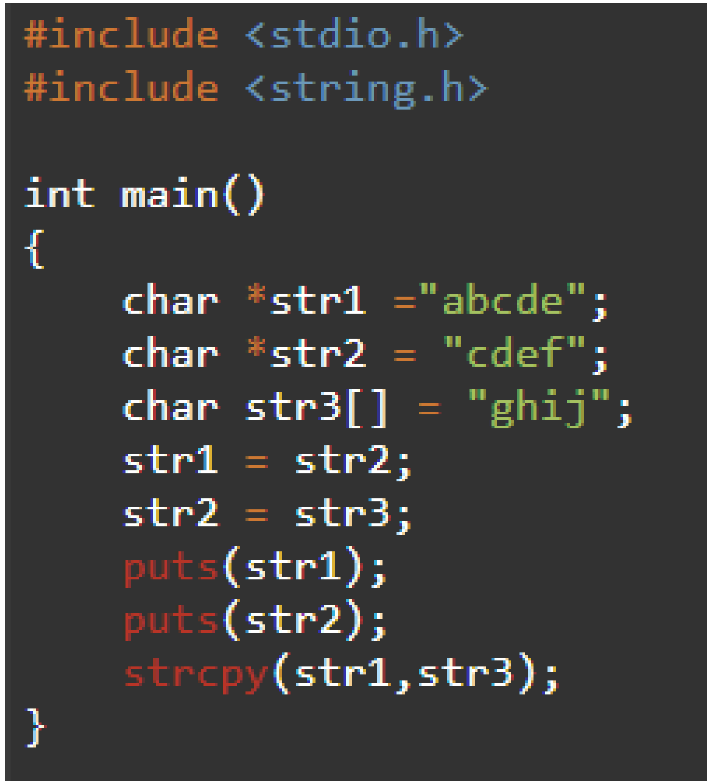
Stomp!

Vulnerable to Buffer overflow attacks

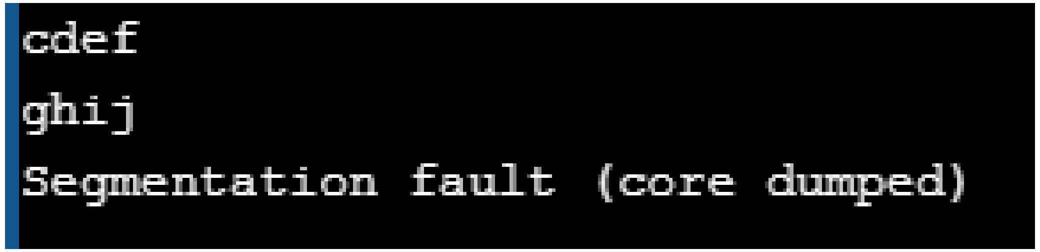
https://onlinegdb.com/rJnovZU4U

Copying strings

- When you use = operator, you only change the address stored in pointer variable.



```c
#include <stdio.h>
#include <string.h>

int main()
{
    char *str1 ="abcde";
    char *str2 = "cdef";
    char str3[] = "ghij";
    str1 = str2;
    str2 = str3;
    puts(str1);
    puts(str2);
    strcpy(str1,str3);
}
```



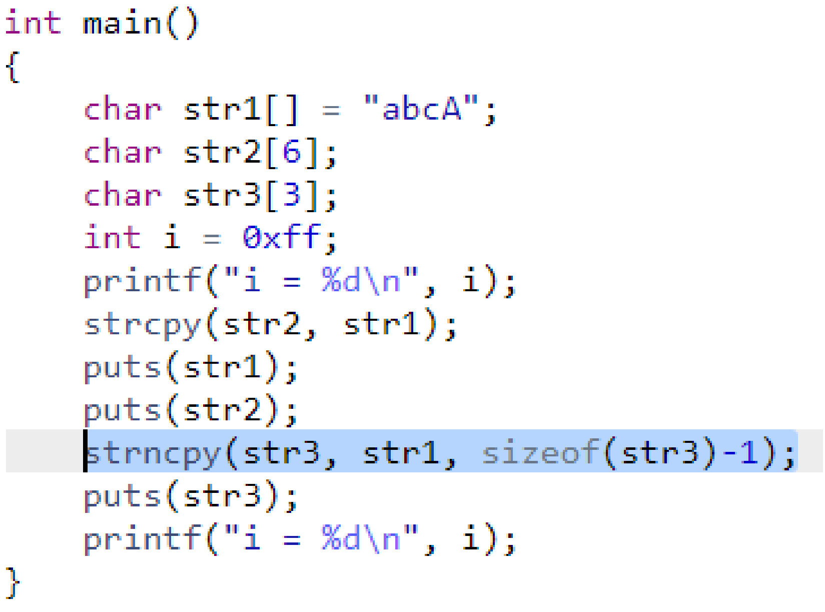
```
cdef
ghij
Segmentation fault (core dumped)
```

Reason of segmentation fault is copying from stack to literal

n-variants of string functions

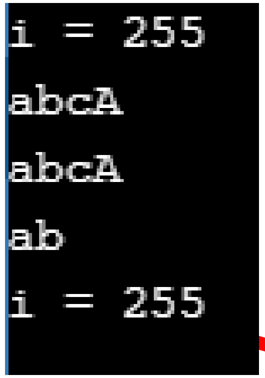
- The best way to thwart buffer overflows (and generally make more safe code) is to use the “n” variants of the string functions
  - For example, you can copy a string to make it safer

strncpy(dest, src, n)



```
int main()
{
    char str1[] = "abcA";
    char str2[6];
    char str3[3];
    int i = 0xff;
    printf("i = %d\n", i);
    strcpy(str2, str1);
    puts(str1);
    puts(str2);
    strncpy(str3, str1, sizeof(str3)-1);
    puts(str3);
    printf("i = %d\n", i);
}
```

https://onlinegdb.com/HyjQcZUEL

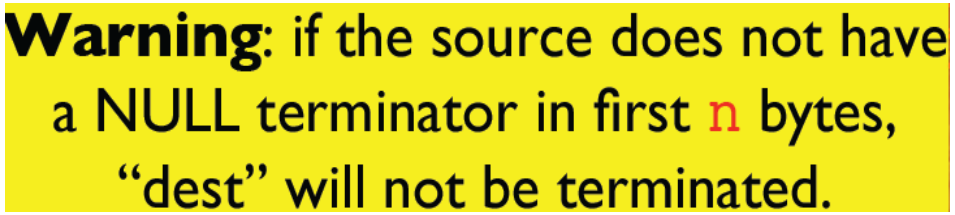


```
i = 255
abcA
abcA
ab
i = 255
```

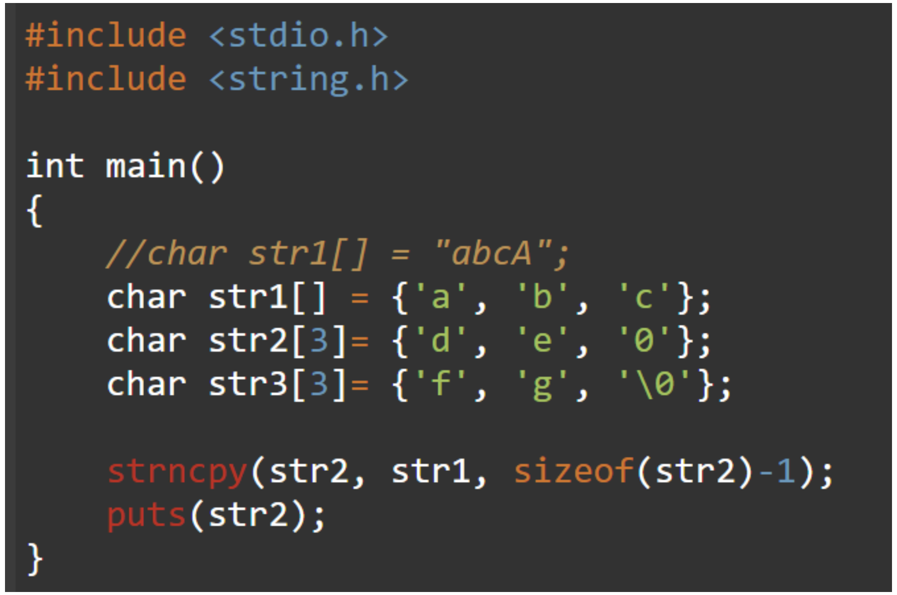
No Stomp

Warning: if the source does not have a NULL terminator in first n bytes, “dest” will not be terminated.

n-variants of string functions



Warning: if the source does not have a NULL terminator in first n bytes, “dest” will not be terminated.



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

int main()
{
    //char str1[] = "abcA";
    char str1[] = {'a', 'b', 'c'};
    char str2[3]= {'d', 'e', '0'};
    char str3[3]= {'f', 'g', '\0'};

    strncpy(str2, str1, sizeof(str2)-1);
    puts(str2);
}
```

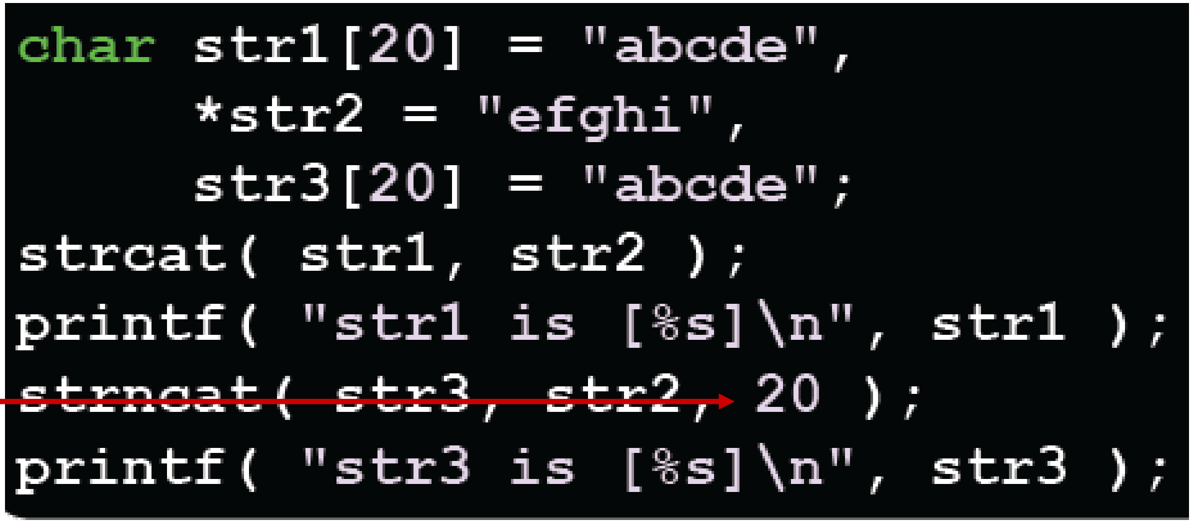
https://onlinegdb.com/ryu-wL03I

Concatenating strings ...

- Often, we want to “add” strings together to make one long string, e.g., as in C++ or Java (str = str1 + str2)
- In C, we use strcat (which appends src to dest)

  strcat(dest, src);
- The strncat variant copies at most n bytes of src

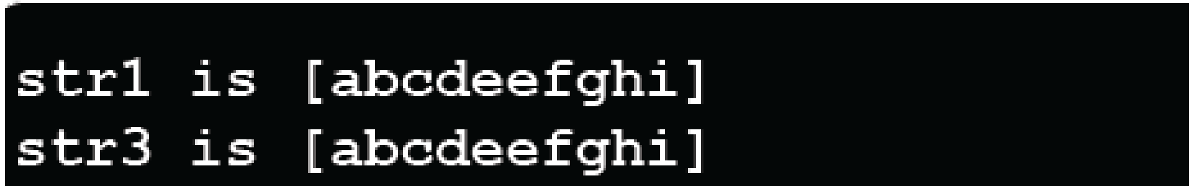
  strncat(dest, src, n);



```
char str1[20] = "abcde",
     *str2 = "efghi",
     str3[20] = "abcde";
strcat( str1, str2 );
printf( "str1 is [%s]\n", str1 );
strncat( str3, str2, 20 );
printf( "str3 is [%s]\n", str3 );
```

Total size of concat maybe more than 20 characters, still not safe

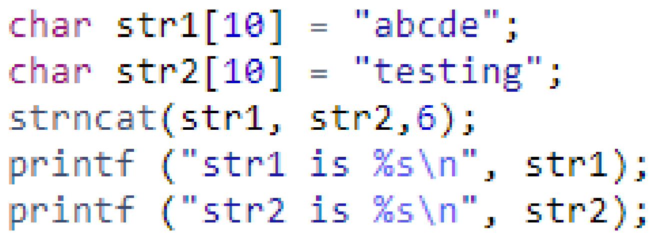
Be very careful



```
str1 is [abcdeefghi]
str3 is [abcdeefghi]
```

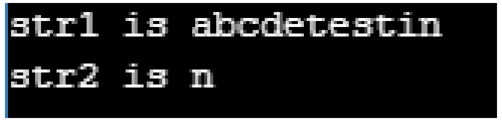
Concatenating strings

- What if we pass larger sizes?
  - You still need to be careful when using strncat
  - Undefined behavior if total size of concatenated string is bigger than destination



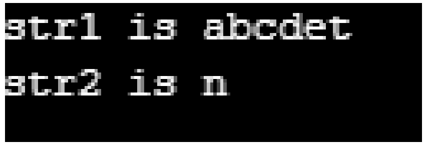
```
char str1[10] = "abcde";
char str2[10] = "testing";
strncat(str1, str2,6);
printf ("str1 is %s\n", str1);
printf ("str2 is %s\n", str2);
```

run1



```
str1 is abcdetestin
str2 is n
```

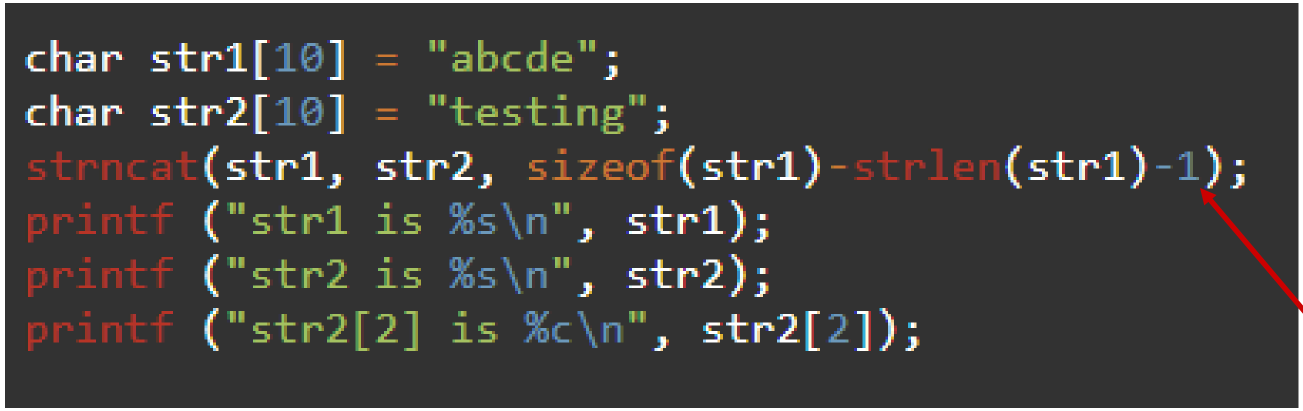
run2



```
str1 is abcdet
str2 is n
```

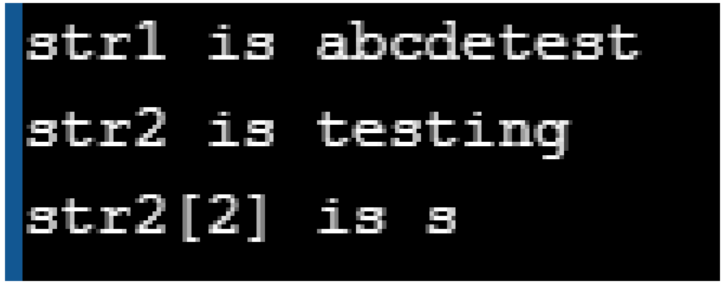
https://onlinegdb.com/SkePLc6mU

Concatenating String



```
char str1[10] = "abcde";
char str2[10] = "testing";
strncat(str1, str2, sizeof(str1)-strlen(str1)-1);
printf ("str1 is %s\n", str1);
printf ("str2 is %s\n", str2);
printf ("str2[2] is %c\n", str2[2]);
```

This is to leave space for null character



```
str1 is abcdetest
str2 is testing
str2[2] is s
```

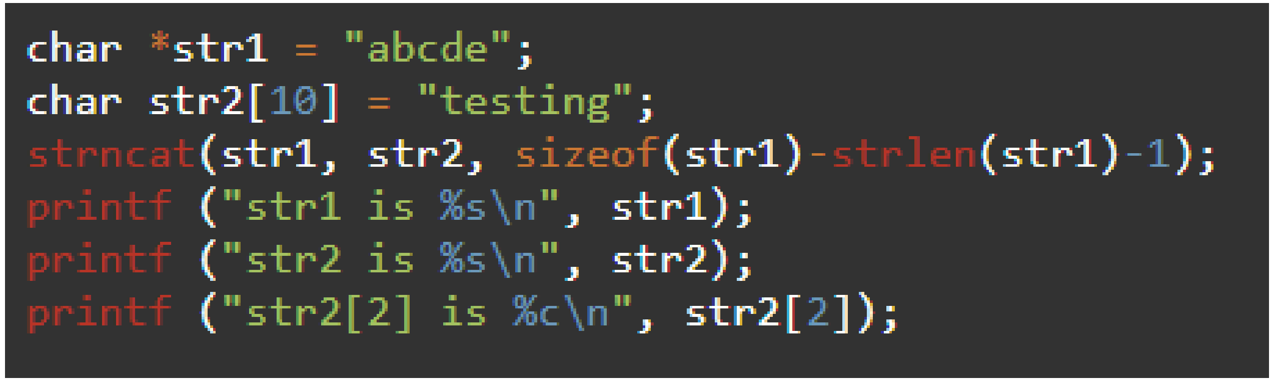
This worked!, why?

https://onlinegdb.com/ryQyR9idB

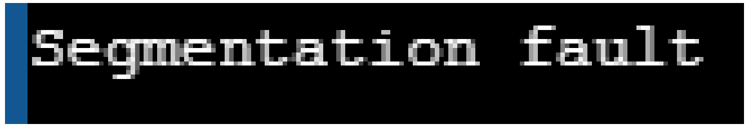
https://onlinegdb.com/HkZK2wV5B

Concatenating String

- What if we try to concatenate into a literal.



```
char *str1 = "abcde";
char str2[10] = "testing";
strncat(str1, str2, sizeof(str1)-strlen(str1)-1);
printf ("str1 is %s\n", str1);
printf ("str2 is %s\n", str2);
printf ("str2[2] is %c\n", str2[2]);
```



```
Segmentation fault
```

Literals cannot be modified

https://onlinegdb.com/HJkdC5j_r

String comparisons ...

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

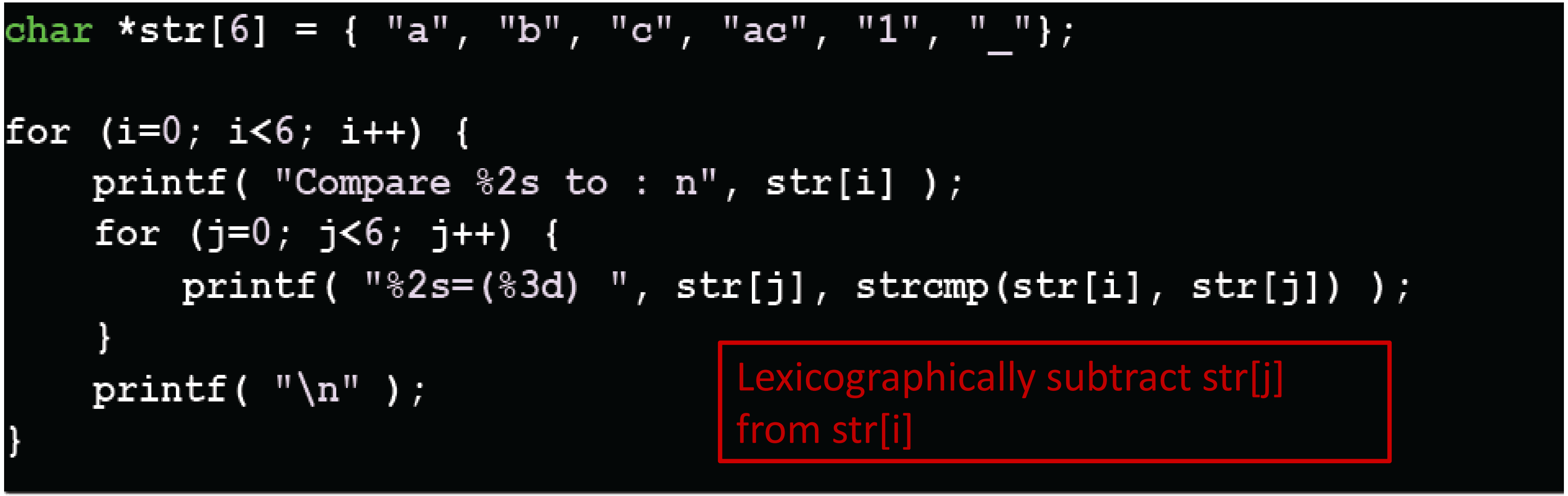
```
strcmp(s1, s2);
```

- strncmp compares first n bytes of strings

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

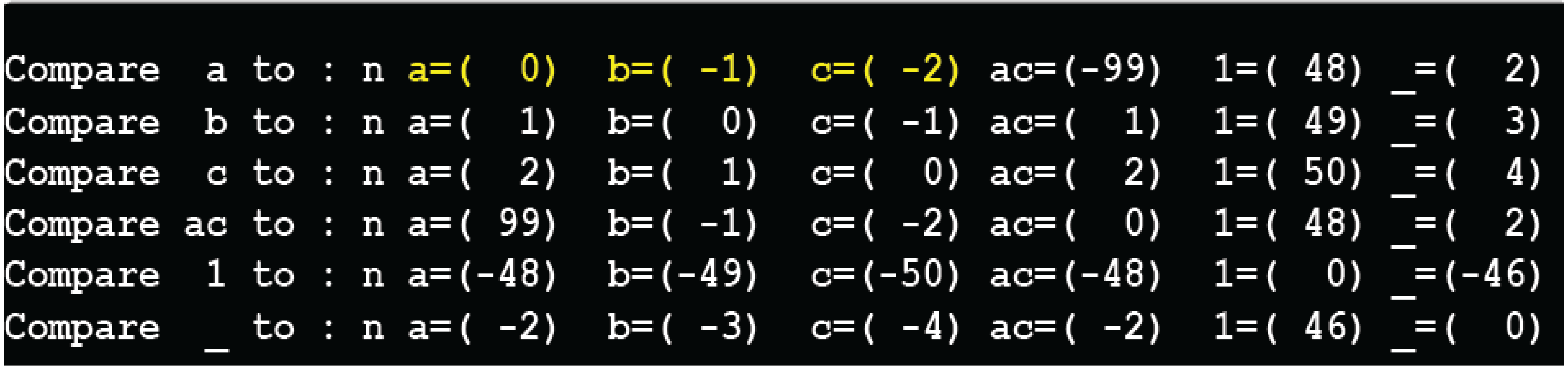
- The comparison functions return
  - negative integer if s1 is less than s2
  - 0 if s1 is equal to s2
  - positive integer is s1 greater than s2

How is a string greater than?



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

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

Lexicographically subtract str[j] from str[i]



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

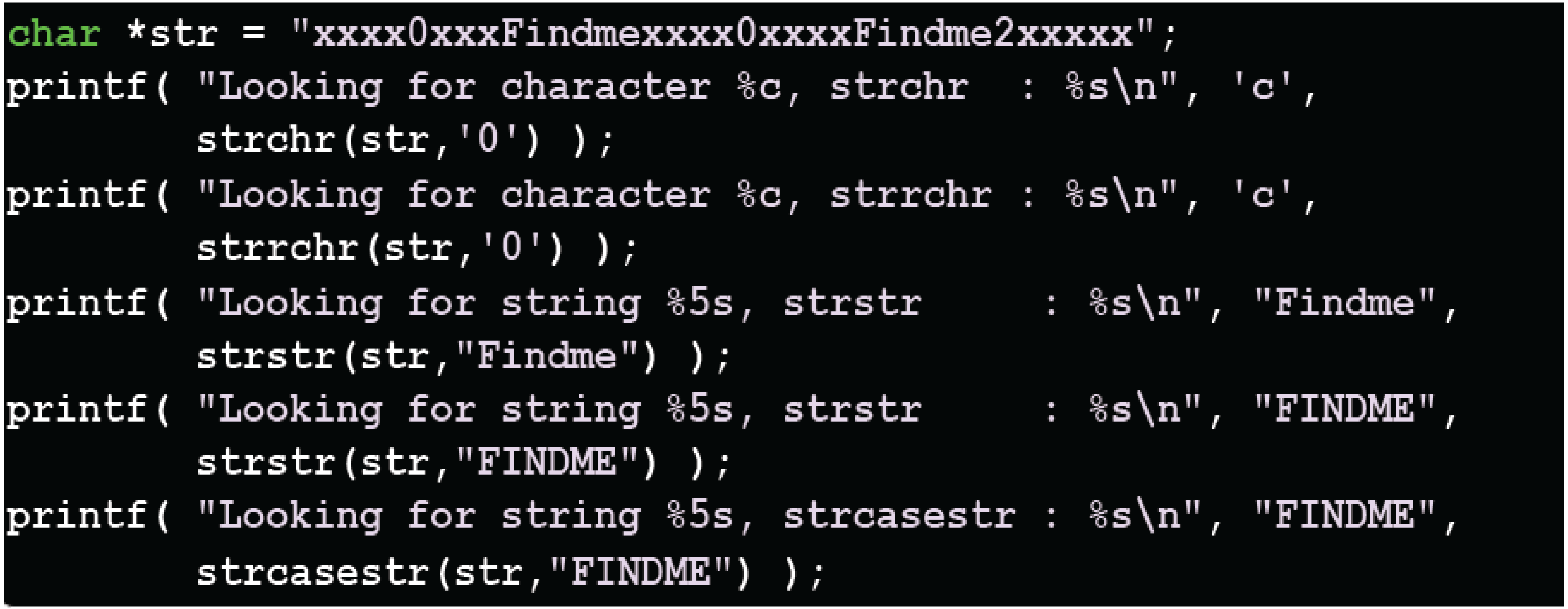
```
strchr(str, char_to_find);
strrchr(str, char_to_find);
```

  - strstr searches front to back for a string
  - strcasestr searches from front for a string (ignoring case)

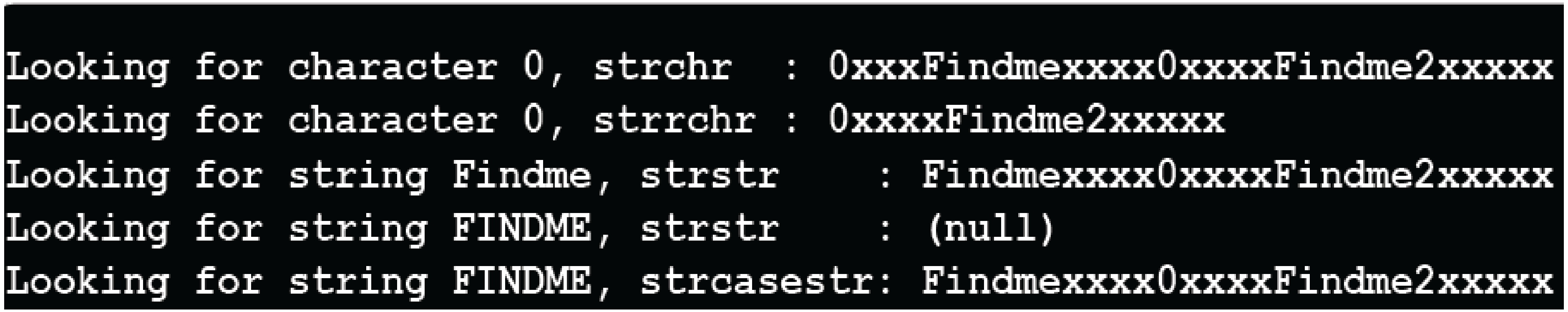
```
strstr(str, str_to_find);
strcasestr(str, str_to_find);
```

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

Example searches



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

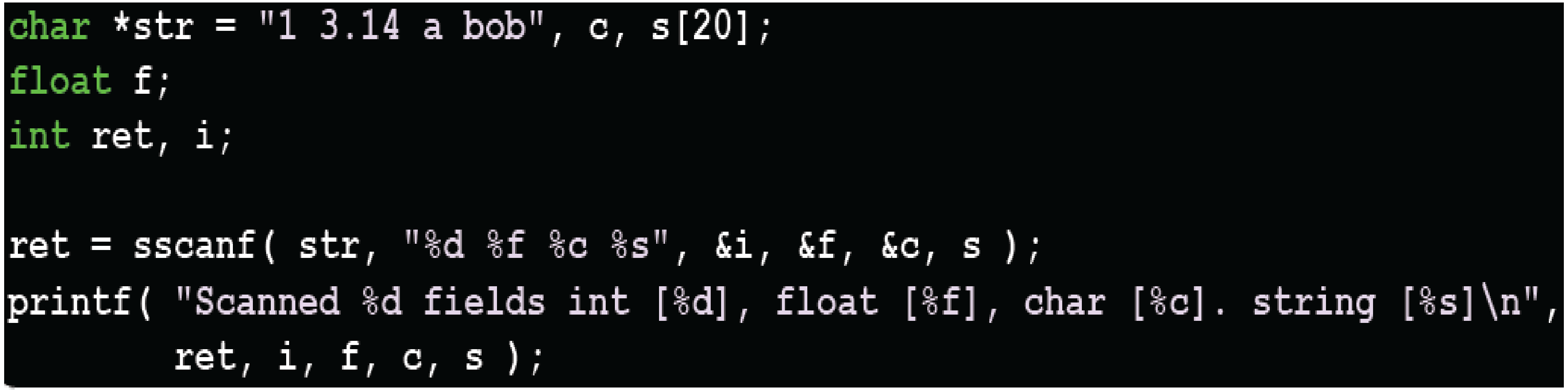
Parsing strings ...

- Strings carry information we want to translate (parse) into other forms (variables)
- In C, we use sscanf which extracts data by format

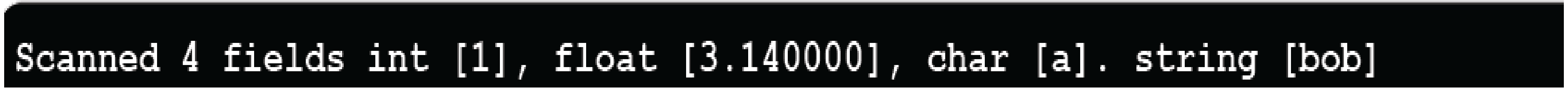
int sscanf(const char *str, const char *format, ...)

Sequence of pointers

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

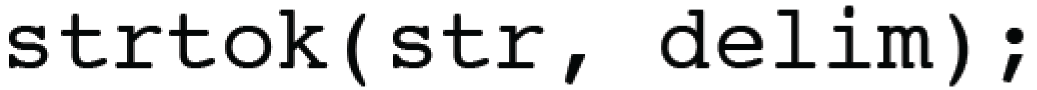


```
Scanned 4 fields int [1], float [3.140000], char [a]. string [bob]
```

Tokenizing strings ...

- Input is often in a form ready for parsing, such as the **.csv** format (**comma separated values**)

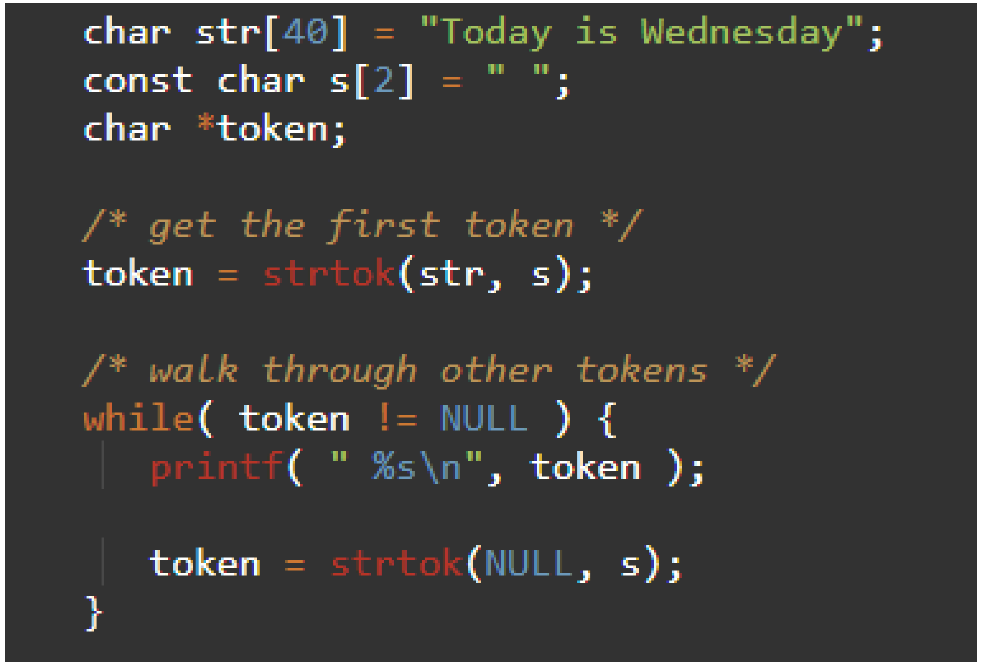
- We want to be able to pull that data apart so we can process it, where each field is a token
  - Here we use the strtok function



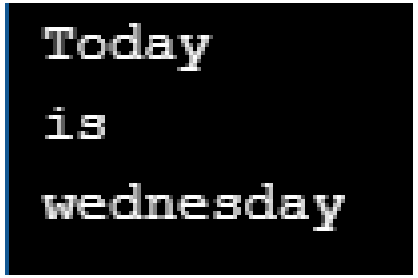
```
strtok(str, delim);
```

  - First use pass the string to parse, thereafter NULL

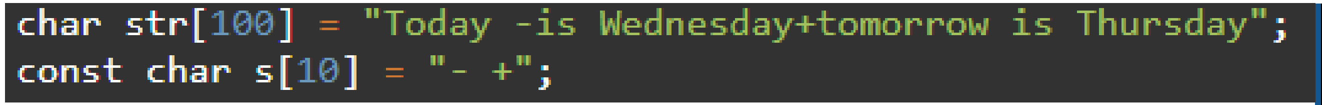
Tokenizing strings ...



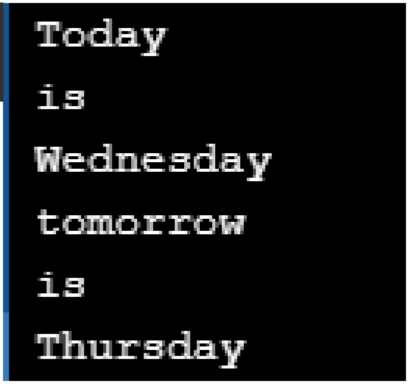
```c
char str[40] = "Today is Wednesday";
const char s[2] = " ";
char *token;

/* get the first token */
token = strtok(str, s);

/* walk through other tokens */
while( token != NULL ) {
   printf( " %s\n", token );

   token = strtok(NULL, s);
}
```



```
Today
is
wednesday
```



```c
char str[100] = "Today -is Wednesday+tomorrow is Thursday";
const char s[10] = "- +";
```



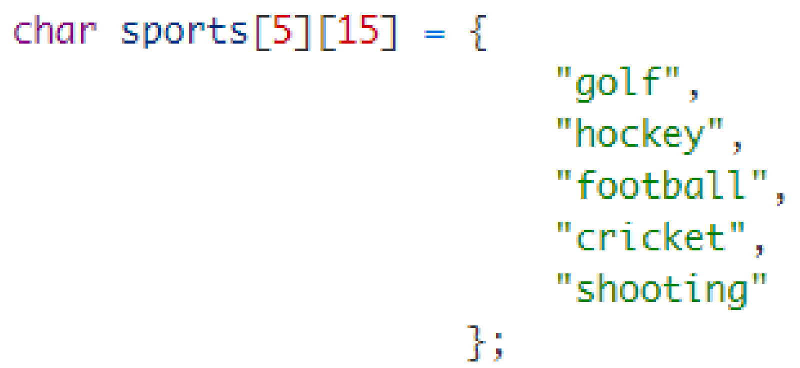
```
Today
is
Wednesday
tomorrow
is
Thursday
```

https://onlinegdb.com/HkN89_5OS

Extra

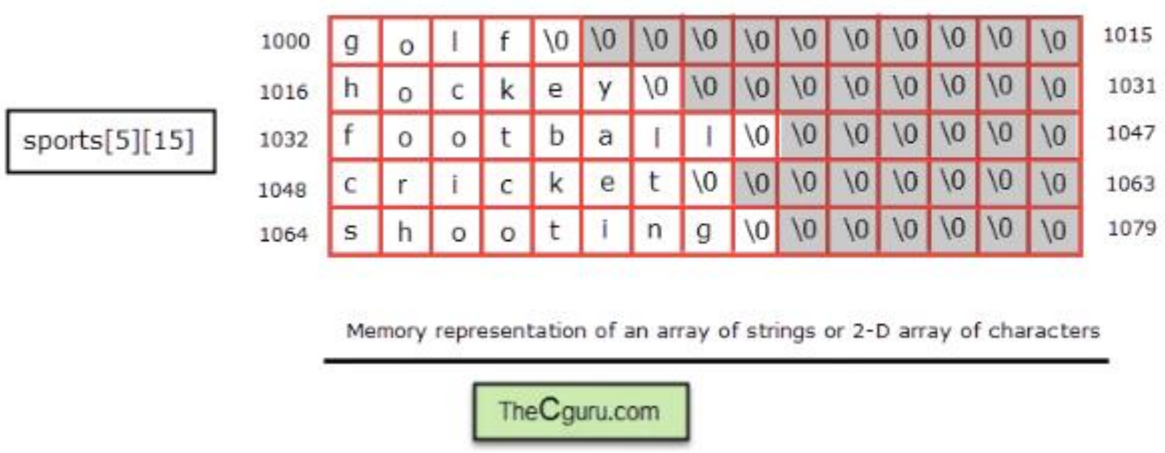
Array of strings

- String can be stored in two dimensional array as follows.



```
char sports[5][15] = {
                        "golf",
                        "hockey",
                        "football",
                        "cricket",
                        "shooting"
                     };
```

https://onlinegdb.com/B1xxvU903I



sports[5][15]

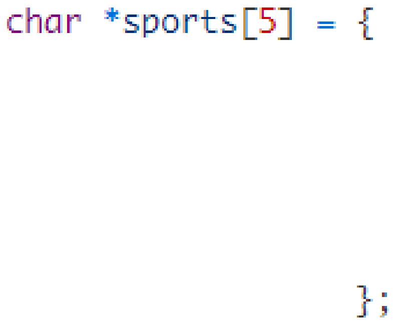
1000	g	o	l	f	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	1015
1016	h	o	c	k	e	y	\0	\0	\0	\0	\0	\0	\0	\0	\0	1031
1032	f	o	o	t	b	a	l	l	\0	\0	\0	\0	\0	\0	\0	1047
1048	c	r	i	c	k	e	t	\0	\0	\0	\0	\0	\0	\0	\0	1063
1064	s	h	o	o	t	i	n	g	\0	\0	\0	\0	\0	\0	\0	1079

Memory representation of an array of strings or 2-D array of characters

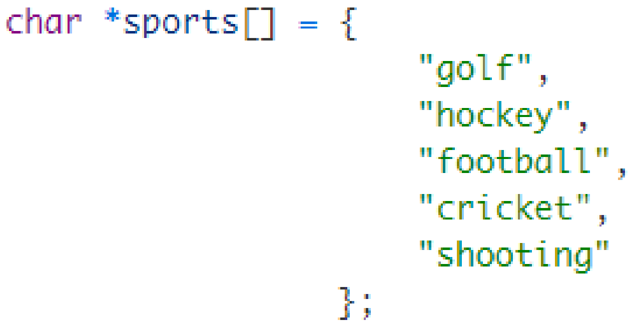
TheCguru.com

Ragged Array of strings

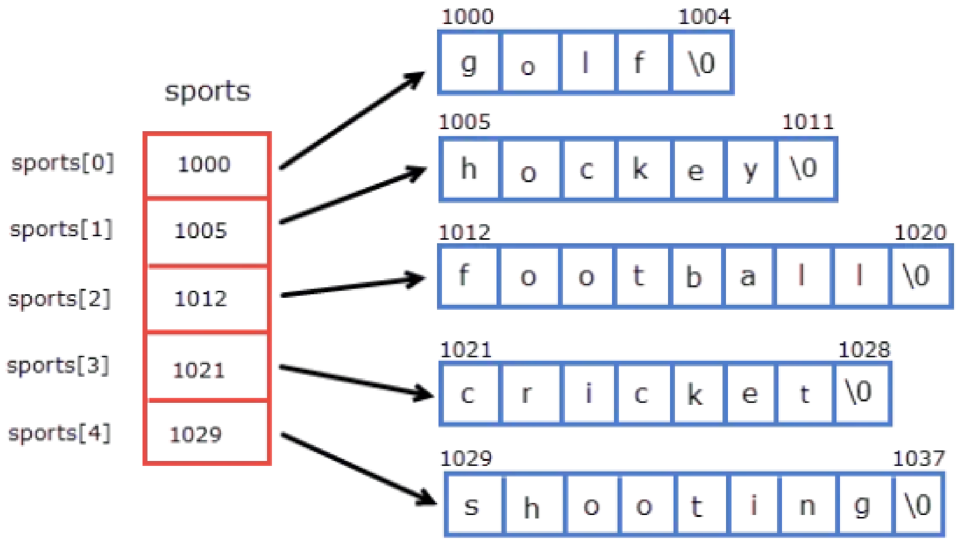
- We can use string literals and pointers instead.



```
char *sports[5] = {
                     "golf",
                     "hockey",
                     "football",
                     "cricket",
                     "shooting"
                 };
```



```
char *sports[] = {
                     "golf",
                     "hockey",
                     "football",
                     "cricket",
                     "shooting"
                 };
```

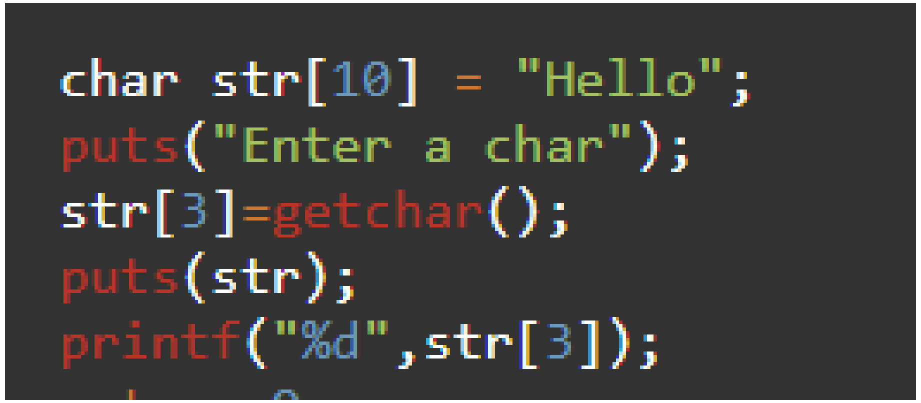


Memory representation of array of pointers

TheCguru.com

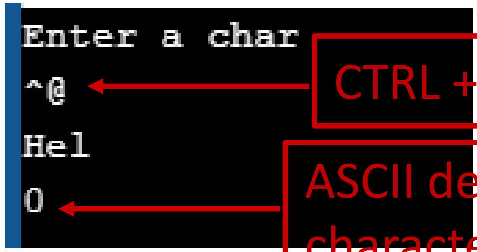
Entering null (\0) character from keyboard as program input

- This is dependent to the system.
- In most Linux and Windows systems. CTRL + SPACE will type a null character.
- Ascii null character has a decimal 0 value as show in ASCII table



```
char str[10] = "Hello";
puts("Enter a char");
str[3]=getchar();
puts(str);
printf("%d",str[3]);
```

https://onlinegdb.com/BydGdMh_B



```
Enter a char
^@
Hel
0
```

CTRL + SPACE

ASCII decimal 0 is null character. Since we cannot print null character we printed decimal value