CS107, Lecture 5
More C Strings

Reading: K&R (1.6, 5.5, Appendix B3) or Essential C section 3

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (last time)
- Helps us better understand buffer overflows, a common bug (last time)
- Introduces us to pointers, because strings can be pointers (this time)

assign2: implement 2 functions a 1 program using those functions to find the location of different built-in commands in the filesystem. You'll write functions to extract a list of possible locations and tokenize that list of locations.

Learning Goals

- Understand how to use the built-in string functions for common string tasks
- Learn more about the risks of buffer overflows and how to mitigate them
- Understand how strings are represented as pointers and how that helps us better understand their behavior

Lecture Plan

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows, Security and Valgrind
- Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Lecture Plan

- <u>**Recap: Strings so far**</u>
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows, Security and Valgrind
- Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

String operations such as `strlen` use the null-terminating character to find the end of the string.

Side note: use `strlen` to get the length of a string. Don't use `sizeof`!

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).

`strcmp(`***`str1, str2`***`),`
`strncmp(`***`str1, str2, n`***`)`
	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.

`strchr(`***`str, ch`***`)`
`strrchr(`***`str, ch`***`)`
	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.

`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***`needle`*** in ***`haystack`***, or ***`NULL`*** if ***`needle`*** was not found in ***`haystack`***.

`strcpy(`***`dst, src`***`),`
`strncpy(`***`dst, src, n`***`)`
	copies characters in ***`src`*** to ***`dst`***, including null-terminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and <u>does not</u> add null-terminating char.

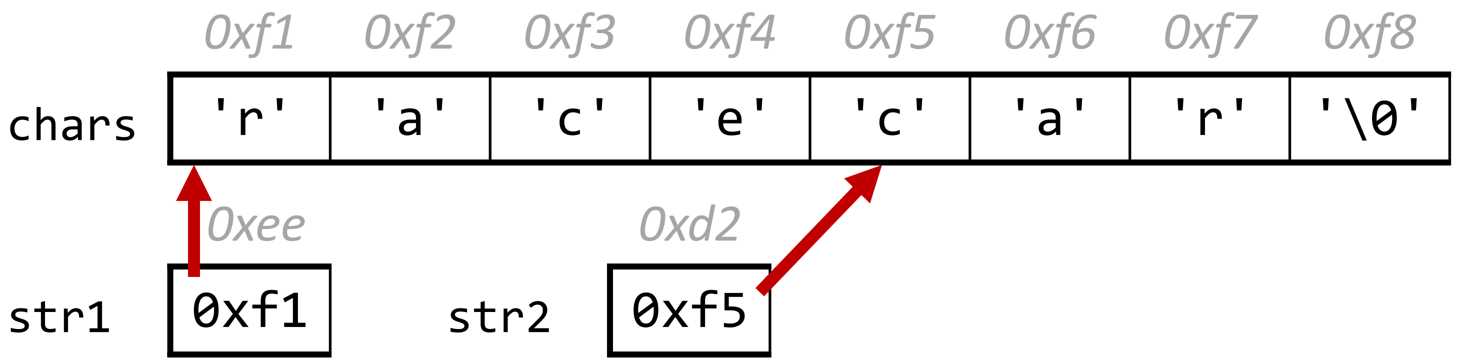
`strcat(`***`dst, src`***`),`
`strncat(`***`dst, src, n`***`)`
	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. <u>Always</u> adds a null-terminating character.

`strspn(`***`str, accept`***`),`
`strcspn(`***`str, reject`***`)`
	`strspn` returns the length of the initial part of ***`str`*** which contains <u>only</u> characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does <u>not</u> contain any characters in ***`reject`***.

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning.

```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
```



Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char str1[8];
strcpy(str1, "racecar");

char str2[5];
strncpy(str2, str1, 4);
str2[4] = '\0';
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // race
```

Substrings

We can combine pointer arithmetic and copying to make any substrings we'd like.

```
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = '\0';
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // ace
```

String Diamond

Write a function **diamond** that accepts a string parameter and prints its letters in a "diamond" format as shown below.

- For example, `diamond("BAILEY")` should print:

```
B
BA
BAI
BAIL
BAILE
BAILEY
 AILEY
  ILEY
   LEY
    EY
     Y
```

Practice: String Diamond

string_diamond.c

Lecture Plan

- **Recap:** Strings so far
- **Searching in Strings**
- **Practice:** Password Verification
- Buffer Overflows, Security and and Valgrind
- Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char bailey[7];
strcpy(bailey, "Bailey");
char *letterI = strchr(bailey, 'i');
printf("%s\n", bailey);                // Bailey
printf("%s\n", letterI);               // iley
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `str`**`r`**`chr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char bailey[11];
strcpy(bailey, "Bailey Dog");
char *substr = strstr(bailey, "Dog");
printf("%s\n", bailey);              // Bailey Dog
printf("%s\n", substr);              // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strspn(bailey, "aBeoi");      // 3
```

"How many places can we go in the first string before I encounter a character <u>not in</u> the second string?"

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strcspn(bailey, "driso");     // 2
```

"How many places can we go in the first string before I encounter a character in the second string?"

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. We can still operate on the string the same way as with a char[]. (*We'll see why today!*).

```
int doSomething(char *str) {
    char secondChar = str[1];
    ...
}

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...
```

Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", stringArray[0]);  // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what **argv** is in **main**! This means we write the parameter type as:

```
void myFunction(char **stringArray) {

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```

Practice: Password Verification

Write a function **verifyPassword** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars, char
*badSubstrings[], int numBadSubstrings);
```

password is valid if it contains only letters in **validChars**, and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars, char
*badSubstrings[], int numBadSubstrings);
```

Example:

```
char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789",
         invalidSubstrings, 1);      // true
bool valid2 = verifyPassword("141234", "0123456789",
         invalidSubstrings, 1);      // false
```

Practice: Password Verification

```
verify_password.c
```

Lecture Plan

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- **Buffer Overflows, Security and Valgrind**
- Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Recall: Buffer Overflows

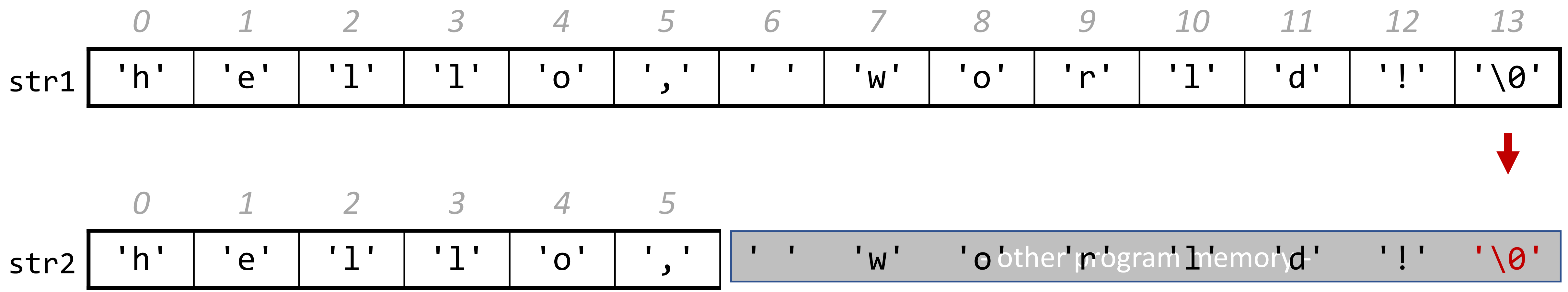
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
char str2[6];                     // not enough space!
strcpy(str2, "hello, world!");   // overwrites other memory!
```

Writing past memory bounds is called a “buffer overflow”. It can allow for security vulnerabilities!

Recall: Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
str1
'h' 'e' 'l' 'l' 'o' ',' ' ' 'w' 'o' 'r' 'l' 'd' '!' '\0'
str2
'h' 'e' 'l' 'l' 'o' ','
- other program memory -
' ' 'w' 'o' 'r' 'l' 'd' '!' '\0'

Buffer Overflow Impacts

Buffer overflows are not merely functionality bugs; they can cause a range of unintended behavior:

- Access memory you shouldn’t be able to access
- Modify memory you shouldn’t be able to access
  - Change a value that is used later in the program
  - Change the program to execute your instructions instead of its own
- And more…

It’s our job as programmers to find and fix buffer overflows and other bugs not just for the functional correctness of our programs, but to protect people who use and interact with my code.

Buffer Overflow Impacts

- AOL instant messenger buffer overflow: allowed remote attackers to execute code: https://www.cvedetails.com/cve/CVE-2002-0362/
- Morris Worm: first internet worm to gain widespread attention; exploited buffer overflow in Unix command called ”finger”: https://en.wikipedia.org/wiki/Morris_worm

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- **Carefully reading documentation**
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How can we fix buffer overflows?

MAN page for gets():

“Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.”

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- **<u>Thorough testing to uncover issues before release</u>**
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How Can We Fix Overflows?

- **Valgrind**: Your Greatest Ally
- Write your own tests
- Consider writing tests *before* writing the main program

✨ `cs107.stanford.edu/testing.html` ✨

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- **<u>Thorough documentation to document assumptions in your code</u>**
- (Where possible) use of tools that reduce the possibility for buffer overflows

How Can We Fix Overflows?

Documentation & MAN Pages (Written by Others)

“The strcpy() function copies the string pointed to by src, including the terminating null byte (‘\0’), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.) …

BUGS

If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there’s enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.”

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- **(Where possible) use of tools that reduce the possibility for buffer overflows**

Memory Safe Systems Programming

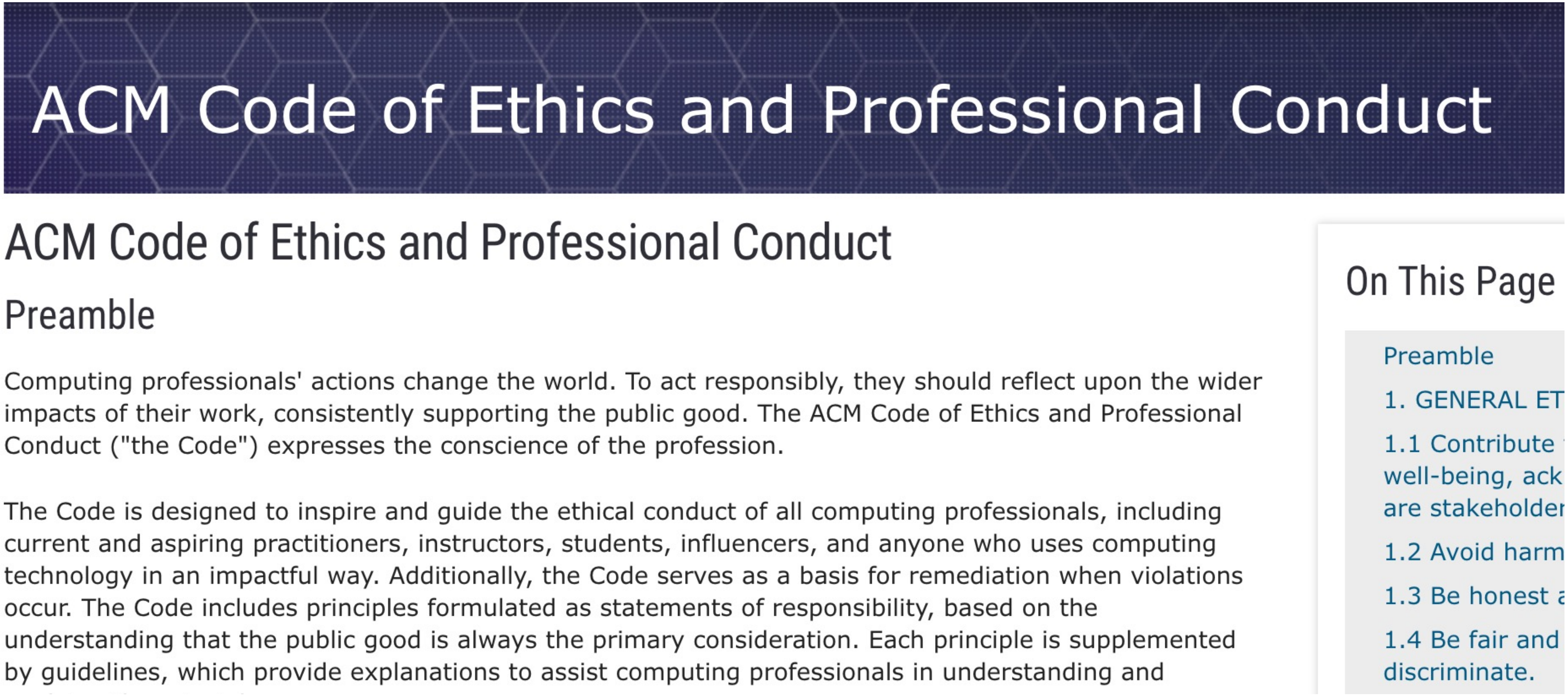
Idea 5: Choose your Tools & Languages Carefully

Existing code bases or requirements for a project may dictate what tools you use. Knowing C is crucial – it is and will remain widely used.

When you you are choosing tools for systems programming, consider languages that can help guard against programmer error.

- Rust (Mozilla)
- Go (Google)
- Project Verona (Microsoft)

Association for Computing Machinery (ACM) Code of Ethics



ACM Code of Ethics and Professional Conduct

ACM Code of Ethics and Professional Conduct

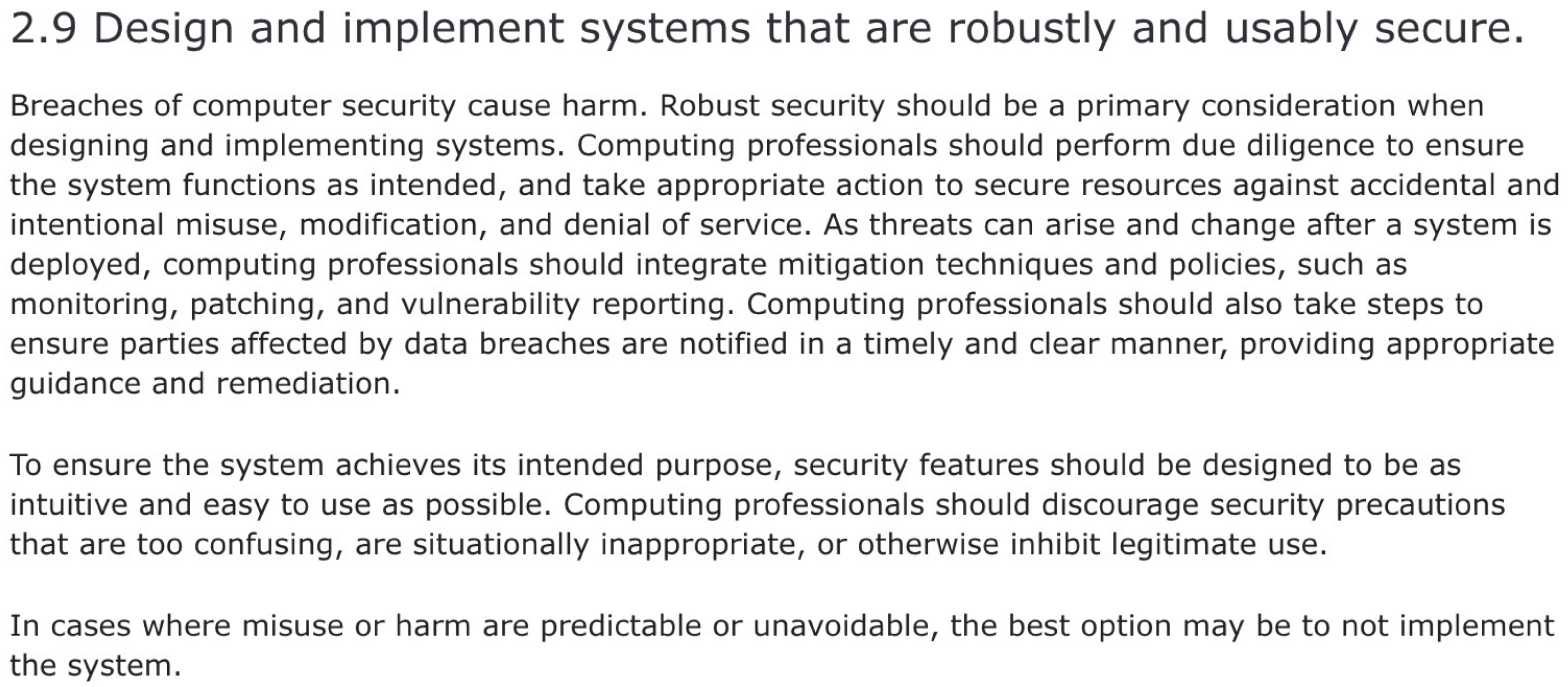
Preamble

Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good. The ACM Code of Ethics and Professional Conduct ("the Code") expresses the conscience of the profession.

The Code is designed to inspire and guide the ethical conduct of all computing professionals, including current and aspiring practitioners, instructors, students, influencers, and anyone who uses computing technology in an impactful way. Additionally, the Code serves as a basis for remediation when violations occur. The Code includes principles formulated as statements of responsibility, based on the understanding that the public good is always the primary consideration. Each principle is supplemented by guidelines, which provide explanations to assist computing professionals in understanding and

On This Page

ACM Code of Ethics on Security



2.9 Design and implement systems that are robustly and usably secure.

Breaches of computer security cause harm. Robust security should be a primary consideration when designing and implementing systems. Computing professionals should perform due diligence to ensure the system functions as intended, and take appropriate action to secure resources against accidental and intentional misuse, modification, and denial of service. As threats can arise and change after a system is deployed, computing professionals should integrate mitigation techniques and policies, such as monitoring, patching, and vulnerability reporting. Computing professionals should also take steps to ensure parties affected by data breaches are notified in a timely and clear manner, providing appropriate guidance and remediation.

To ensure the system achieves its intended purpose, security features should be designed to be as intuitive and easy to use as possible. Computing professionals should discourage security precautions that are too confusing, are situationally inappropriate, or otherwise inhibit legitimate use.

In cases where misuse or harm are predictable or unavoidable, the best option may be to not implement the system.

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

Buffer Overflows

- We must always ensure that memory operations we perform don’t improperly read or write memory.
  - E.g. don’t copy a string into a space that is too small!
  - E.g. don’t ask for the string length of an uninitialized string!
- The **Valgrind** tool may be able to help track down memory-related issues.
  - See cs107.stanford.edu/resources/valgrind
  - We’ll talk about Valgrind more when we talk about dynamically-allocated memory.

Demo: Memory Errors

`memory_errors.c`

Lecture Plan

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows, Security and Valgrind
- **<u>Pointers</u>**

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- One (8 byte) pointer can refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

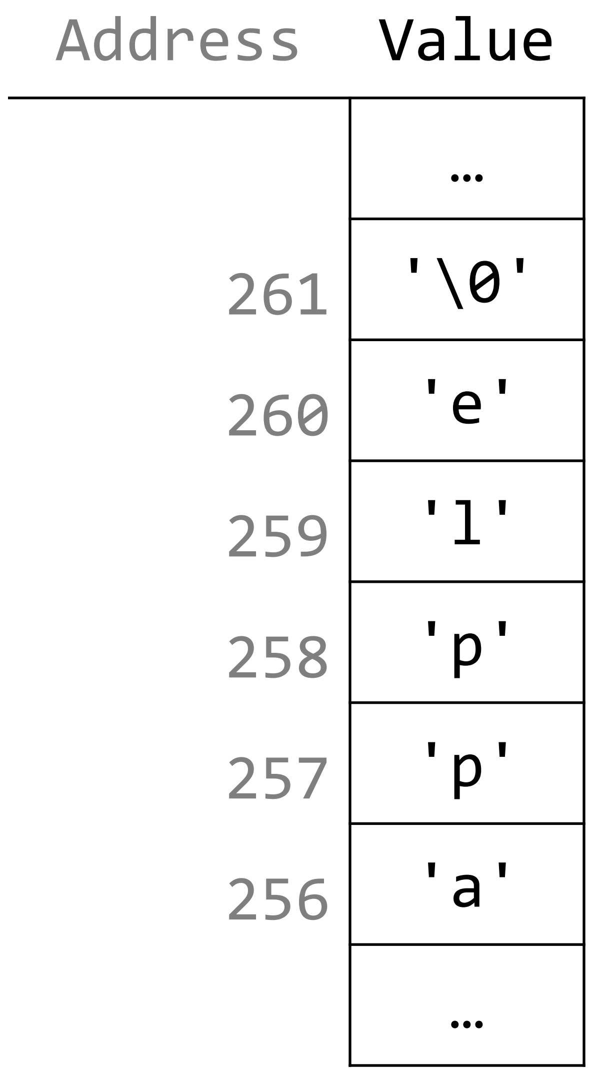
Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.

Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.



Looking Back at C++

How would we write a program with a function that takes in an **`int`** and modifies it? We might use *pass by reference*.

```
void myFunc(int& num) {
	num = 3;
}

int main(int argc, char *argv[]) {
	int x = 2;
	myFunc(x);
	printf("%d", x);   // 3!
	...
}
```

Looking Ahead to C

- All parameters in C are "pass by value." For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
- This means whenever we pass something as a parameter, we pass a copy.
- If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the *address* instead of a copy of the *value*.

Pointers

```c
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

Recap

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows, Security and Valgrind
- Pointers
- Strings in Memory

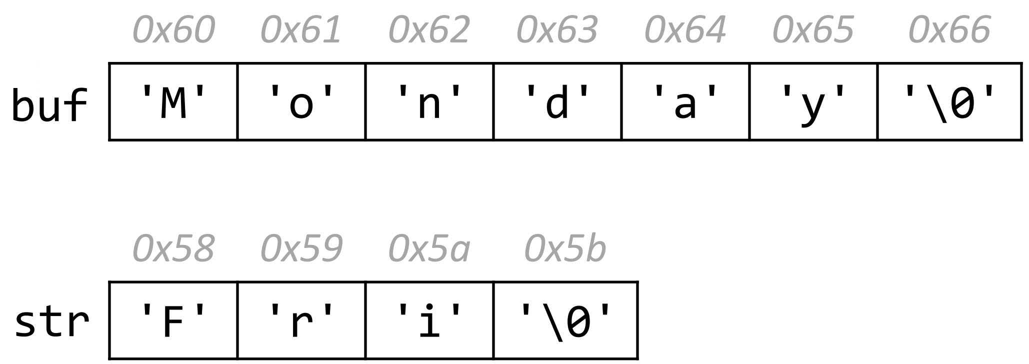
Lecture 5 takeaway: C strings are pointers and arrays. C strings are error-prone, and issues like buffer overflows can arise!

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Extra Practice

2. Code study: strncpy

```
STRCPY(3)                 Linux Programmer's Manual                STRCPY(3)

DESCRIPTION
       The strncpy() function is similar, except that at most n bytes of src  are
       copied.  Warning: If there is no null byte among the first n bytes of src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than  n,  strncpy()  writes  additional  null
       bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:
```

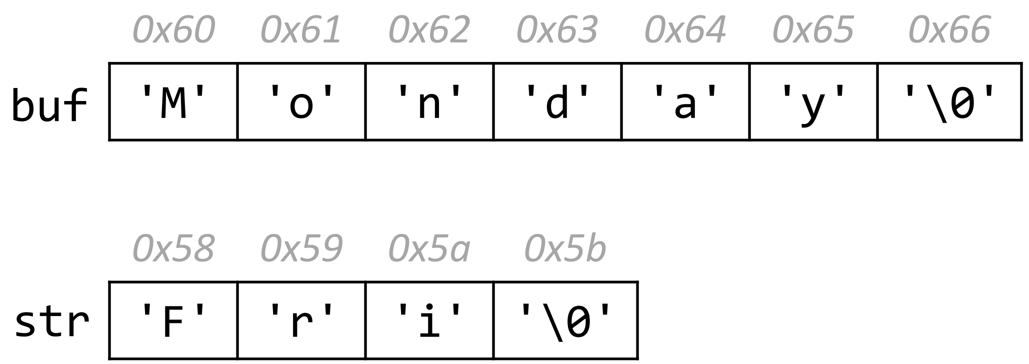


```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
       dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

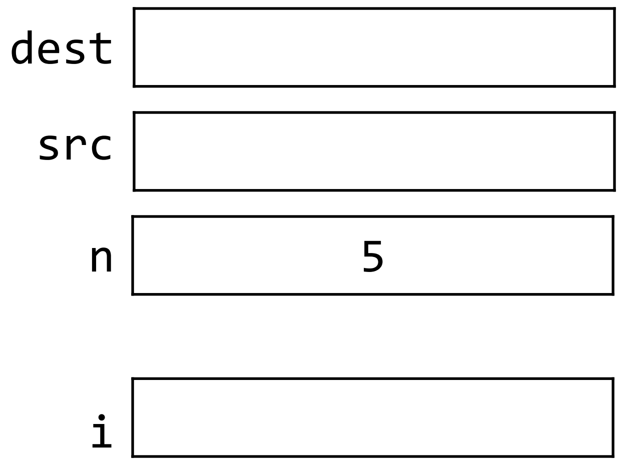
What happens if we call `strncpy(buf, str, 5);`?

2. Code study: strncpy

```
STRCPY(3)                  Linux Programmer's Manual                 STRCPY(3)

DESCRIPTION
       The strncpy() function is similar, except that at most n bytes of src  are
       copied.  Warning: If there is no null byte among the first n bytes of src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than  n,  strncpy()  writes  additional  null
       bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:
```



```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
       dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```



What happens if we call `strncpy(buf, str, 5);`?