CS107, Lecture 7
C Strings, Buffer Overflows and Security

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 (this time)
- Introduces us to pointers, because strings can be pointers (next 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

Lecture Plan

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

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

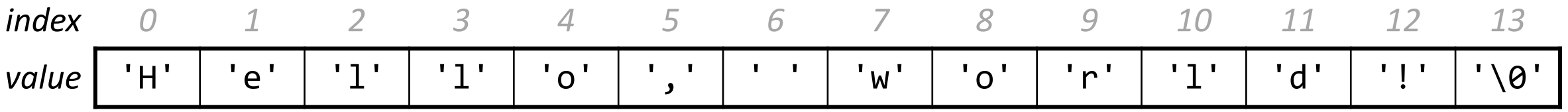
Lecture Plan

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

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

C Strings

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

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
```

char * vs. char[]

- char * is an 8-byte pointer – it stores an address of a character
- char[] is an array of characters – it stores the actual characters in a string
- When you pass a char[] as a parameter, it is automatically passed as a char * (pointer to its first character)

char * vs. char[]

```
char myString[]
```

vs

```
char *myString
```

You can create char * pointers to point to any character in an existing string and reassign them since they are just pointer variables. You **cannot** reassign an array.

```
char myString[6];
strcpy(myString, "Hello");
myString = "Another string";                    // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;                  // ok
```

Lecture Plan

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

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

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 `strrchr` 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 not in 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

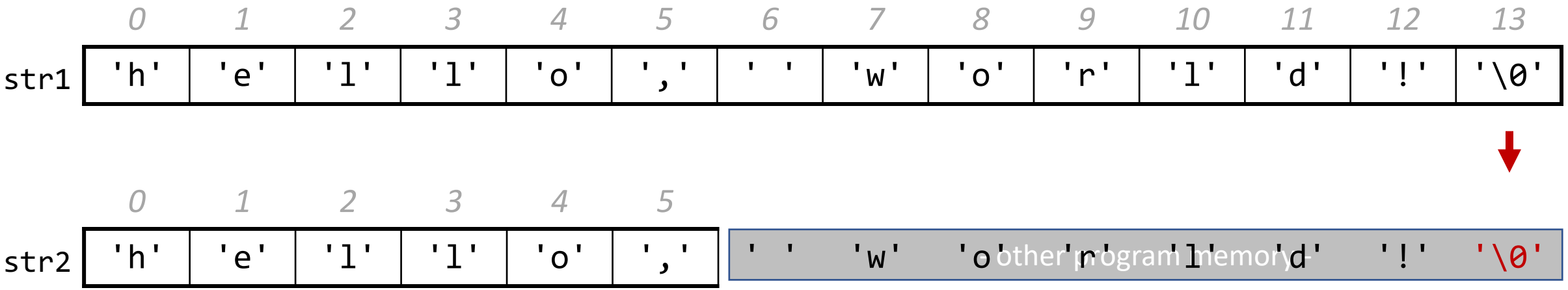
- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- **Buffer Overflows and Security**

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Recall: Buffer Overflows

We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*. Writing past memory bounds is called a "buffer overflow". It can allow for security vulnerabilities!

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



Buffer Overflow Impacts

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

- Let the user access memory they shouldn't be able to access
- Let the user modify memory they shouldn't be able to access
  - Change a value that is used later in the program
  - User changes the program to execute their own custom instructions instead
  - 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 our code.

Buffer Overflow Example: ./buf

```
int main(int argc, char *argv[]) {
    char secret[4] = "123";
    // assume secret comes right after name in memory
    // (this is not always true)
    char name[4];
    strcpy(name, argv[1]);

    if (!strcmp(secret, argv[2])) {
        printf("You're in!\n");
    }
    return 0;
}
```

Which of these arguments would cause the program to print “You’re in!”?

Respond with your thoughts on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.

A. `./buf abcdefgh efgh`
B. `./buf abcd abcd`
C. `./buf a a`
D. `./buf abcdefgh abcd`

Which of these arguments would cause the program to print "You're in!"?

./buf abcdefgh efgh

./buf abcd abcd

./buf a a

./buf abcdefgh abcd

Recap

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

Lecture 7 takeaway: C strings are pointers and arrays. C strings are error-prone, and issues like buffer overflows can arise! Valgrind is a tool that can help detect memory errors.

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Extra Practice

2. Code study: strncpy

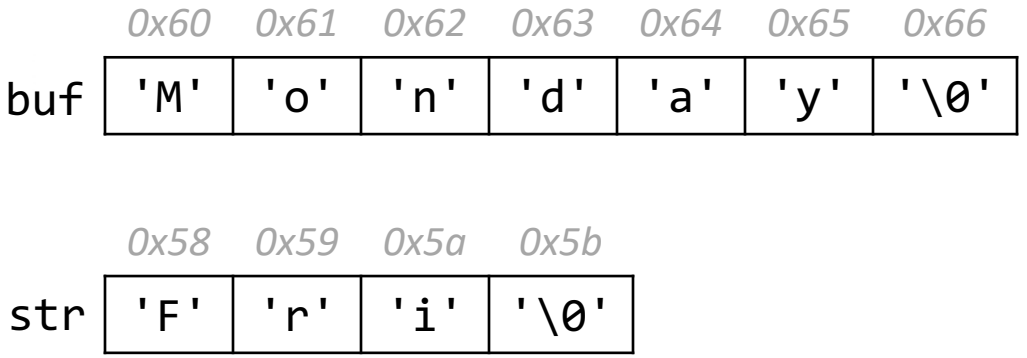
STRCPY(3) Linux Programmer's Manual STRCPY(3)

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

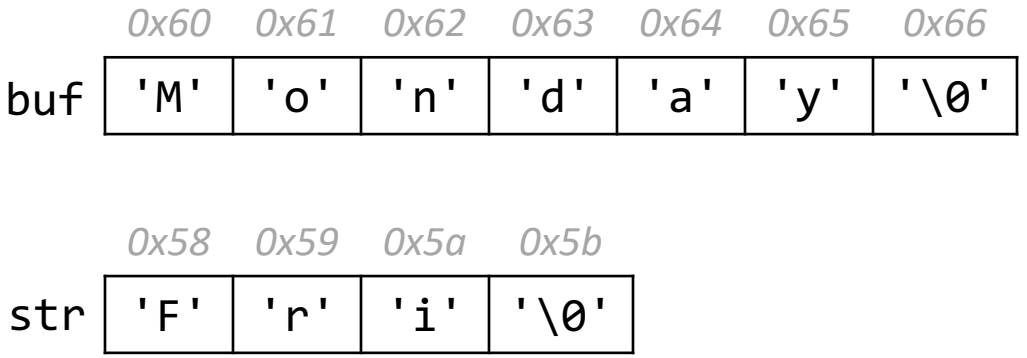
STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

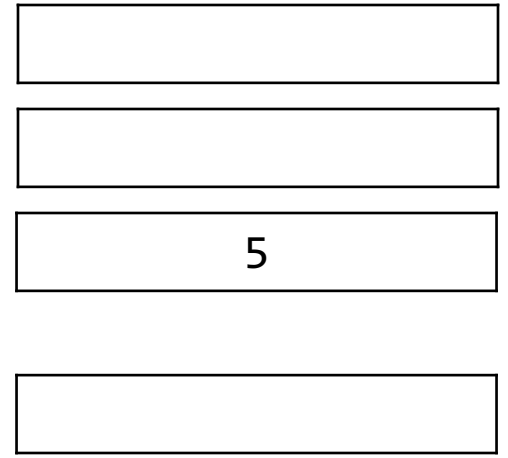
The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

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:




```
1 char *strncpy(char *dest, const char *src, size_t n) {
2     size_t i;
3     for (i = 0; i < n && src[i] != '\0'; i++)
4         dest[i] = src[i];
5     for ( ; i < n; i++)
6         dest[i] = '\0';
7     return dest;
8 }
```



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