

CS107, Lecture 6

C Strings

Reading: K&R (1.9, 5.5, Appendix B3) or Essential
C section 3

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under
Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared,
uploaded, or distributed. (without expressed written permission)

**CS107 Topic 2: How can a
computer represent and
manipulate more complex
data like text?**

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 (this 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 and 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

- Learn how strings are represented in C; as an array of null-terminated characters.
- Understand how to use the built-in string functions for common string tasks
- Learn about buffer overflow and what might cause it

Goal: 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
```



Lecture Plan

- Characters
- Strings
- Common String Operations
 - Comparing
 - Copying
 - Concatenating
 - Substrings

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Lecture Plan

- Characters
- Strings
- Common String Operations
 - Comparing
 - Copying
 - Concatenating
 - Substrings

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Char

A **char** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
```

```
char plus = '+';
```

```
char zero = '0';
```

```
char space = ' ';
```

```
char newLine = '\n';
```

```
char tab = '\t';
```

```
char singleQuote = '\'';
```

```
char backSlash = '\\';
```

ASCII

Under the hood, C represents each **char** as an *integer* (its “ASCII value”).

- Uppercase letters are sequentially numbered
- Lowercase letters are sequentially numbered
- Digits are sequentially numbered
- Lowercase letters are 32 more than their uppercase equivalents (cool trick – can bit flip!)

```
char uppercaseA = 'A';           // Actually 65
```

```
char lowercaseA = 'a';          // Actually 97
```

```
char zeroDigit = '0';          // Actually 48
```

ASCII

We can take advantage of C representing each **char** as an *integer*:

```
bool areEqual = 'A' == 'A';           // true
bool earlierLetter = 'f' < 'c';       // false
char uppercaseB = 'A' + 1;
int diff = 'c' - 'a';                  // 2
int numLettersInAlphabet = 'z' - 'a' + 1;
// or
int numLettersInAlphabet = 'Z' - 'A' + 1;
```

ASCII

We can take advantage of C representing each **char** as an *integer*:

```
// prints out every lowercase character
for (char ch = 'a'; ch <= 'z'; ch++) {
    printf("%c", ch);
}
```

Common ctype.h Functions

Function	Description
<code>isalpha(<i>ch</i>)</code>	true if <i>ch</i> is 'a' through 'z' or 'A' through 'Z'
<code>islower(<i>ch</i>)</code>	true if <i>ch</i> is 'a' through 'z'
<code>isupper(<i>ch</i>)</code>	true if <i>ch</i> is 'A' through 'Z'
<code>isspace(<i>ch</i>)</code>	true if <i>ch</i> is a space, tab, new line, etc.
<code>isdigit(<i>ch</i>)</code>	true if <i>ch</i> is '0' through '9'
<code>toupper(<i>ch</i>)</code>	returns uppercase equivalent of a letter
<code>tolower(<i>ch</i>)</code>	returns lowercase equivalent of a letter

Remember: these **return** a char; they cannot modify an existing char!

More documentation with `man isalpha`, `man tolower`

Common ctype.h Functions

```
bool isLetter = isalpha('A');           // true
bool capital = isupper('f');           // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');         // true
```

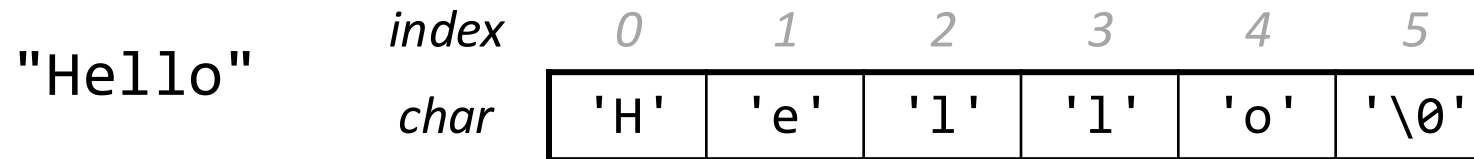
Lecture Plan

- Characters
- **Strings**
- Common String Operations
 - Comparing
 - Copying
 - Concatenating
 - Substrings

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

C Strings

C has no dedicated variable type for strings. Instead, a string is represented as an **array of characters** with a special ending sentinel value.



'\0' is the **null-terminating character**; you always need to allocate one extra space in an array for it. '\0' is the character with numerical value 0. (Note: different from NULL – NULL is 0 pointer, null terminator is 0 character).

C Strings

```
char myString[6];  
myString[0] = 'H';  
myString[1] = 'e';  
myString[2] = 'l';  
...  
myString[5] = '\0';
```

String Length

Strings are **not** objects. They do not embed additional information (e.g., string length). We must calculate this!

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

We can use the provided **strlen** function to calculate string length. The null-terminating character does *not* count towards the length.

```
int length = strlen(myStr);           // e.g. 13
```

Caution: `strlen` is $O(N)$ because it must scan the entire string!
Tip: save the value if you plan to refer to the length later.

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. C passes the location of the first character rather than a copy of the whole array.

```
int doSomething(char *str) {  
    ...  
}
```

```
char myString[6];  
...  
doSomething(myString);
```

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. C passes the location of the first character rather than a copy of the whole array.

```
int doSomething(char *str) {  
    ...  
    str[0] = 'c'; // modifies original string!  
    printf("%s\n", str); // prints cello  
}
```

We can still use a `char *` the same way as a `char[]`.

```
char myString[6];  
... // e.g. this string is "Hello"  
doSomething(myString);
```

Lecture Plan

- Characters
- Strings
- Common String Operations
 - Comparing
 - Copying
 - Concatenating
 - Substrings

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Common string.h Functions

Function	Description
<code>strlen(<i>str</i>)</code>	returns the # of chars in a C string (before null-terminating character).
<code>strcmp(<i>str1</i>, <i>str2</i>)</code> , <code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code>	compares two strings; returns 0 if identical, <0 if <i>str1</i> comes before <i>str2</i> in alphabet, >0 if <i>str1</i> comes after <i>str2</i> in alphabet. <i>strncmp</i> stops comparing after at most <i>n</i> characters.
<code>strchr(<i>str</i>, <i>ch</i>)</code> <code>strrchr(<i>str</i>, <i>ch</i>)</code>	character search: returns a pointer to the first occurrence of <i>ch</i> in <i>str</i> , or <i>NULL</i> if <i>ch</i> was not found in <i>str</i> . <code>strrchr</code> find the last occurrence.
<code>strstr(<i>haystack</i>, <i>needle</i>)</code>	string search: returns a pointer to the start of the first occurrence of <i>needle</i> in <i>haystack</i> , or <i>NULL</i> if <i>needle</i> was not found in <i>haystack</i> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	copies characters in <i>src</i> to <i>dst</i> , including null-terminating character. Assumes enough space in <i>dst</i> . Strings must not overlap. <i>strncpy</i> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char.
<code>strcat(<i>dst</i>, <i>src</i>)</code> , <code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	concatenate <i>src</i> onto the end of <i>dst</i> . <i>strncat</i> stops concatenating after at most <i>n</i> characters. <u>Always</u> adds a null-terminating character.
<code>strspn(<i>str</i>, <i>accept</i>)</code> , <code>strcspn(<i>str</i>, <i>reject</i>)</code>	<i>strspn</i> returns the length of the initial part of <i>str</i> which contains <u>only</u> characters in <i>accept</i> . <i>strcspn</i> returns the length of the initial part of <i>str</i> which does <u>not</u> contain any characters in <i>reject</i> .

Common string.h Functions

Function	Description
<code>strlen(<i>str</i>)</code>	returns the # of chars in a C string (before null-terminating character).
<code>strcmp(<i>str1</i>, <i>str2</i>)</code> , <code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code>	compares two strings; returns 0 if identical, <0 if <i>str1</i> comes before <i>str2</i> in alphabet, >0 if <i>str1</i> comes after <i>str2</i> in alphabet. <i>strncmp</i> stops comparing after at most <i>n</i> characters.
<code>strchr(<i>str</i>, <i>ch</i>)</code> <code>strrchr(<i>str</i>, <i>ch</i>)</code>	character search: returns a pointer to the first occurrence of <i>ch</i> in <i>str</i> , or <i>NULL</i> if <i>ch</i> was not found in <i>str</i> . <i>strrchr</i> find the last occurrence.
<code>strstr(<i>haystack</i>, <i>needle</i>)</code>	returns a pointer to the first occurrence of <i>needle</i> in <i>haystack</i> , or <i>NULL</i> if not found in <i>haystack</i> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	copies <i>src</i> into <i>dst</i> until a null-terminating character. Assumes enough space in <i>dst</i> . Strings must not overlap. <i>strncpy</i> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char.
<code>strcat(<i>dst</i>, <i>src</i>)</code> , <code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	concatenate <i>src</i> onto the end of <i>dst</i> . <i>strncat</i> stops concatenating after at most <i>n</i> characters. <u>Always</u> adds a null-terminating character.
<code>strspn(<i>str</i>, <i>accept</i>)</code> , <code>strcspn(<i>str</i>, <i>reject</i>)</code>	<i>strspn</i> returns the length of the initial part of <i>str</i> which contains <u>only</u> characters in <i>accept</i> . <i>strcspn</i> returns the length of the initial part of <i>str</i> which does <u>not</u> contain any characters in <i>reject</i> .

Many string functions assume **valid string** input; i.e., ends in a null terminator.

Comparing Strings

We cannot compare C strings using comparison operators like `==`, `<` or `>`. This compares addresses!

```
// e.g. str1 = 0x7f42, str2 = 0x654d
void doSomething(char *str1, char *str2) {
    if (str1 > str2) { ... // compares 0x7f42 > 0x654d!
```

Instead, use **`strcmp`**.

The string library: strcmp

`strcmp(str1, str2)`: compares two strings.

- returns 0 if identical
- <0 if **str1** comes before **str2** in alphabet
- >0 if **str1** comes after **str2** in alphabet.

```
int compResult = strcmp(str1, str2);
if (compResult == 0) {
    // equal
} else if (compResult < 0) {
    // str1 comes before str2
} else {
    // str1 comes after str2
}
```

Copying Strings

We cannot copy C strings using =. This copies addresses!

```
// e.g. param1 = 0x7f42, param2 = 0x654d
void doSomething(char *param1, char *param2) {
    param1 = param2;    // copies 0x654d. Points to same string!
    param2[0] = 'H';    // modifies the one original string!
```

Instead, use **strcpy**.

The string library: strcpy

strcpy(dst, src): copies the contents of **src** into the string **dst**, including the null terminator. (Note: string literal has automatic ending null terminator)

```
char str1[6];  
strcpy(str1, "hello");
```

```
char str2[6];  
strcpy(str2, str1);  
str2[0] = 'c';
```

```
printf("%s", str1);           // hello  
printf("%s", str2);           // cello
```

Copying Strings - strcpy

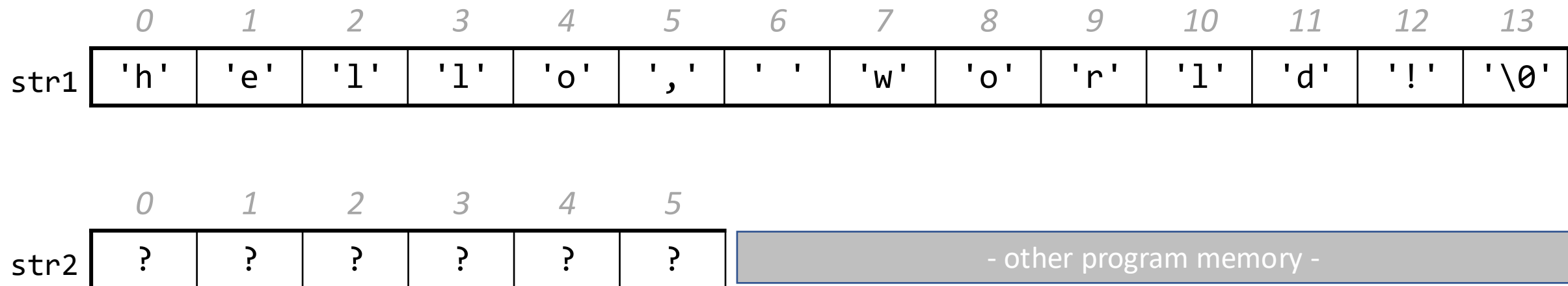
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
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!

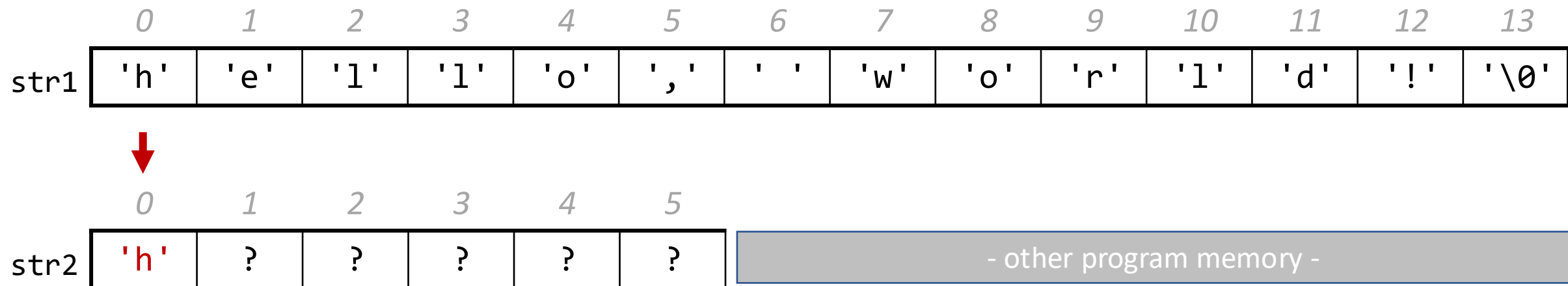
Copying Strings – Buffer Overflows

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



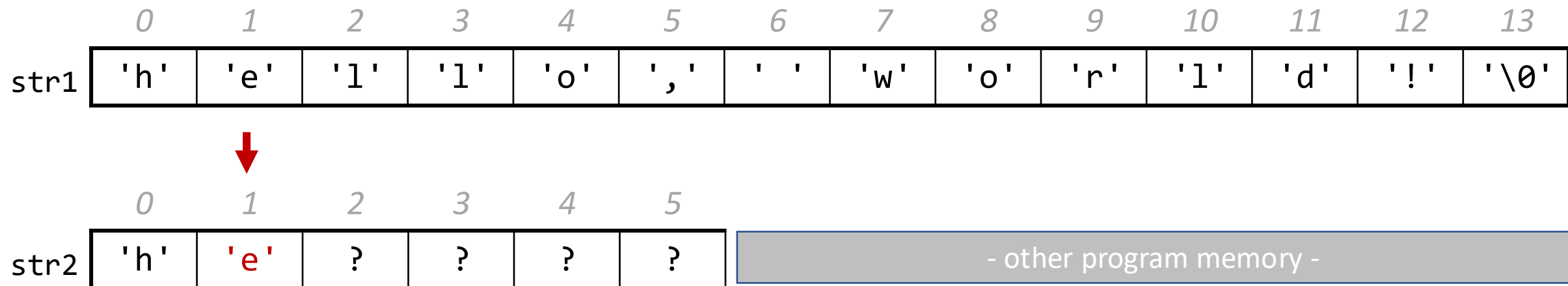
Copying Strings – Buffer Overflows

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



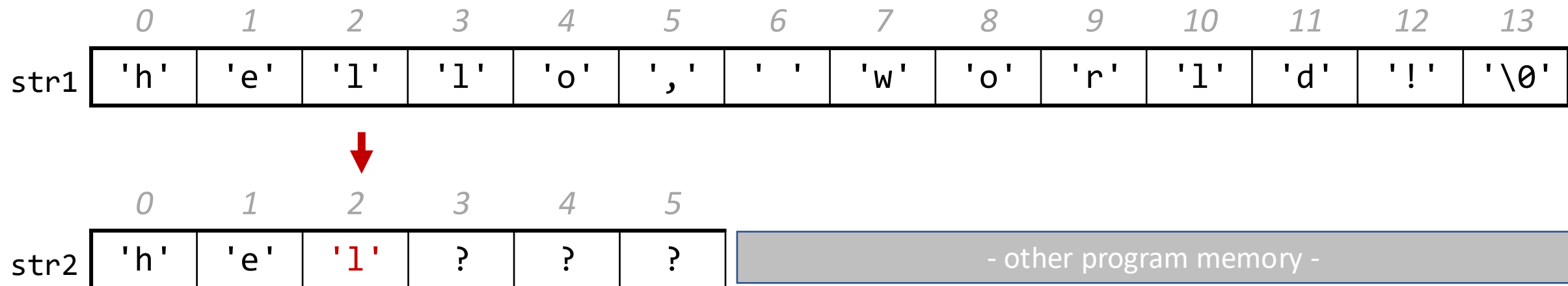
Copying Strings – Buffer Overflows

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



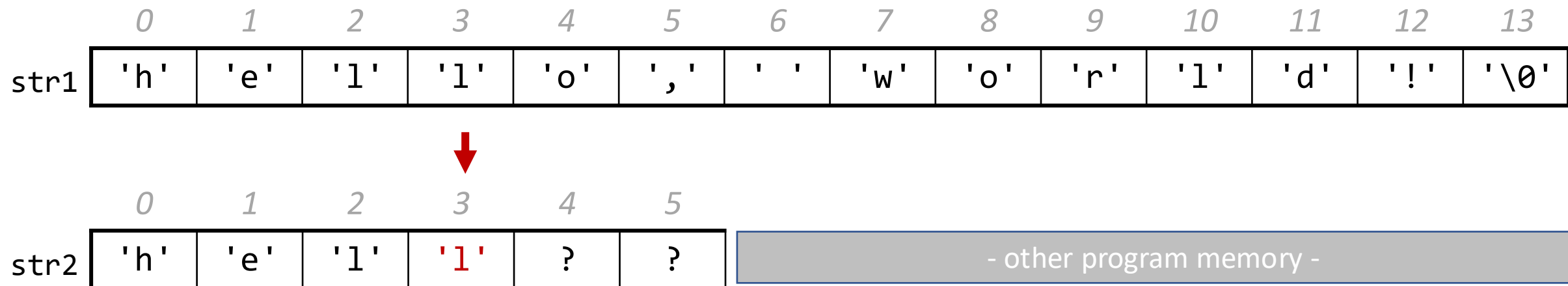
Copying Strings – Buffer Overflows

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



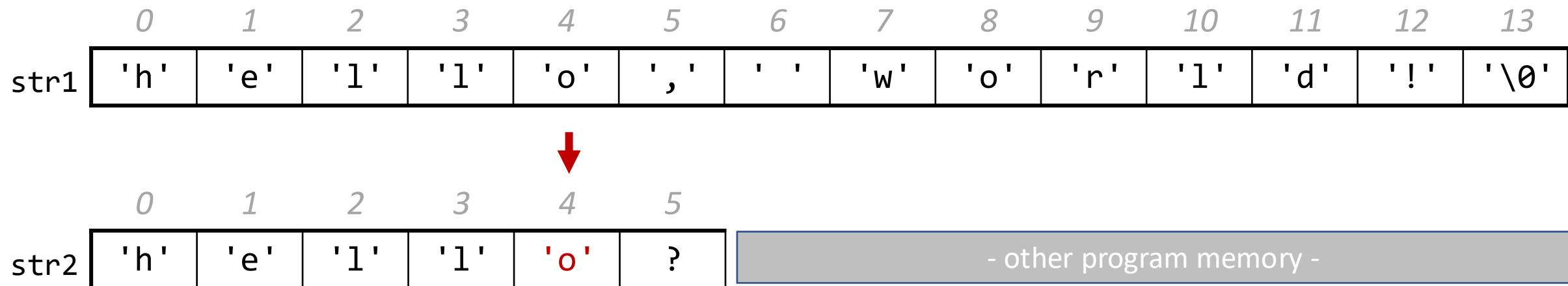
Copying Strings – Buffer Overflows

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



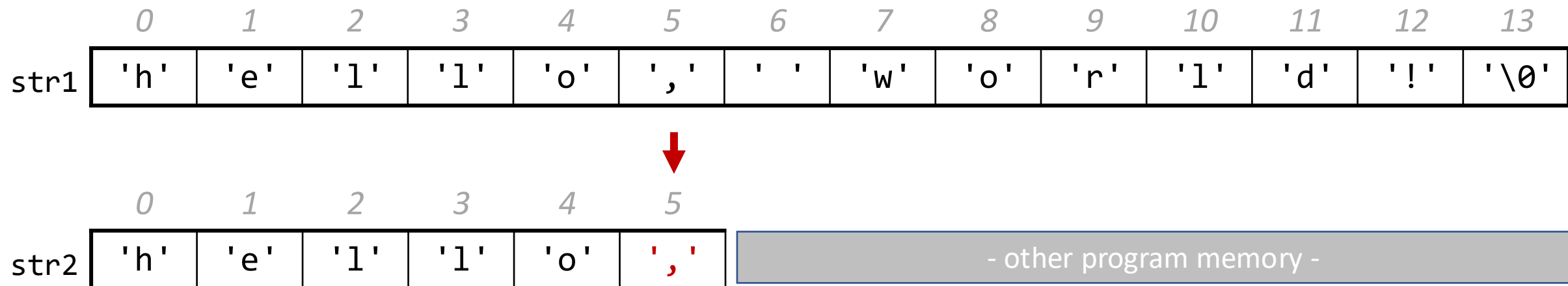
Copying Strings – Buffer Overflows

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



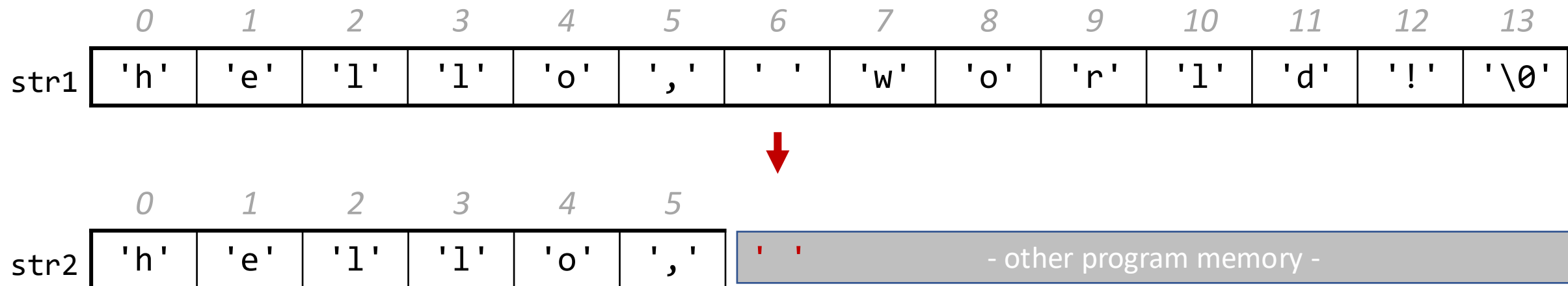
Copying Strings – Buffer Overflows

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



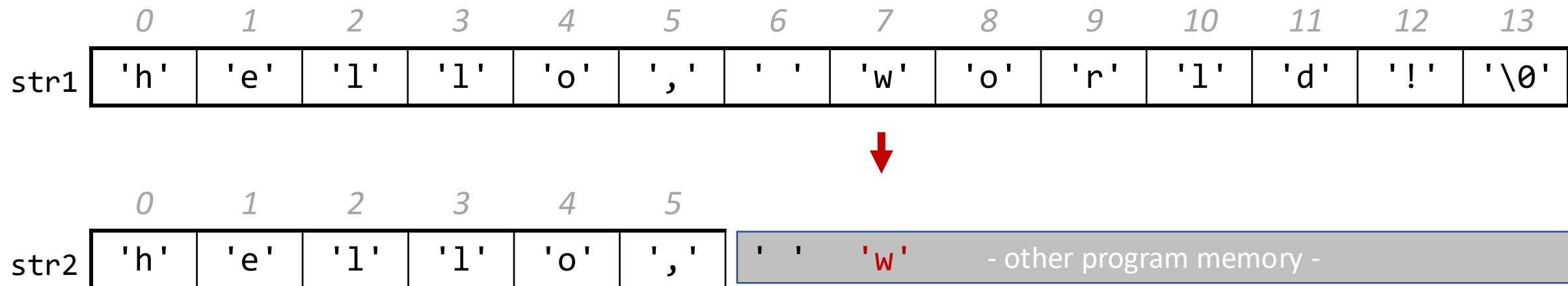
Copying Strings – Buffer Overflows

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



Copying Strings – Buffer Overflows

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



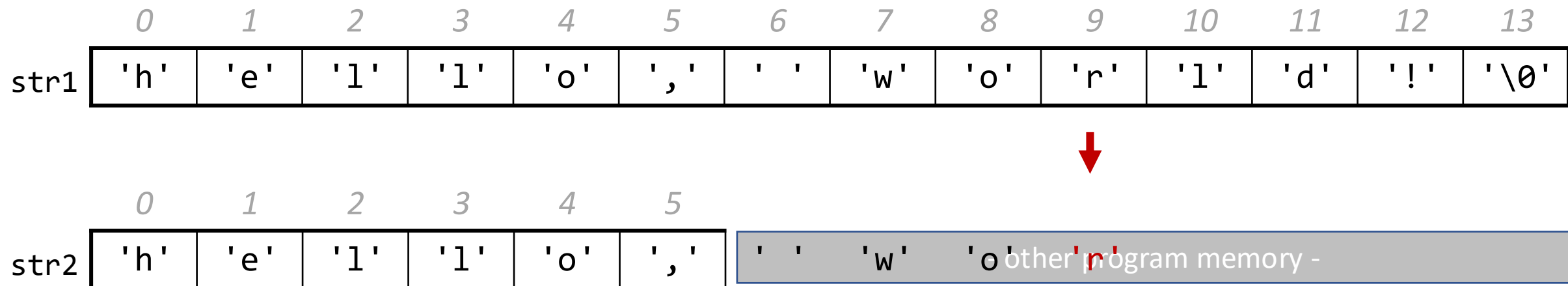
Copying Strings – Buffer Overflows

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



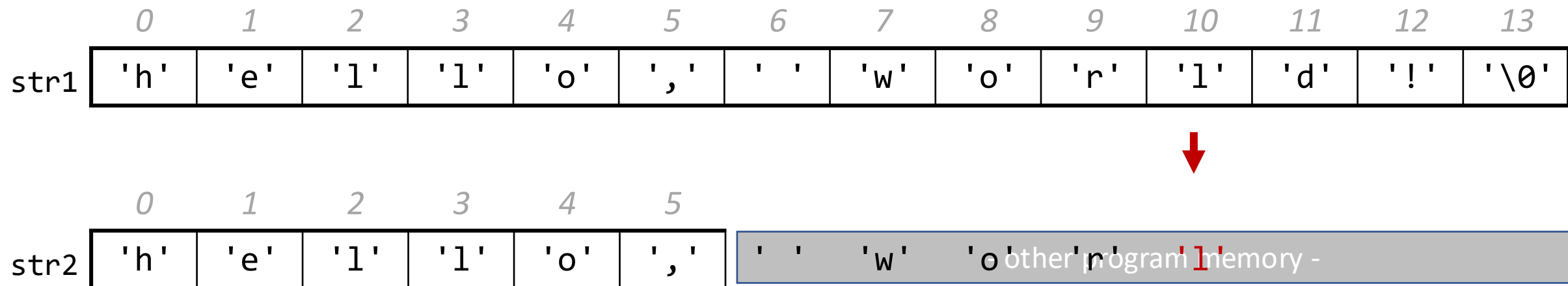
Copying Strings – Buffer Overflows

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



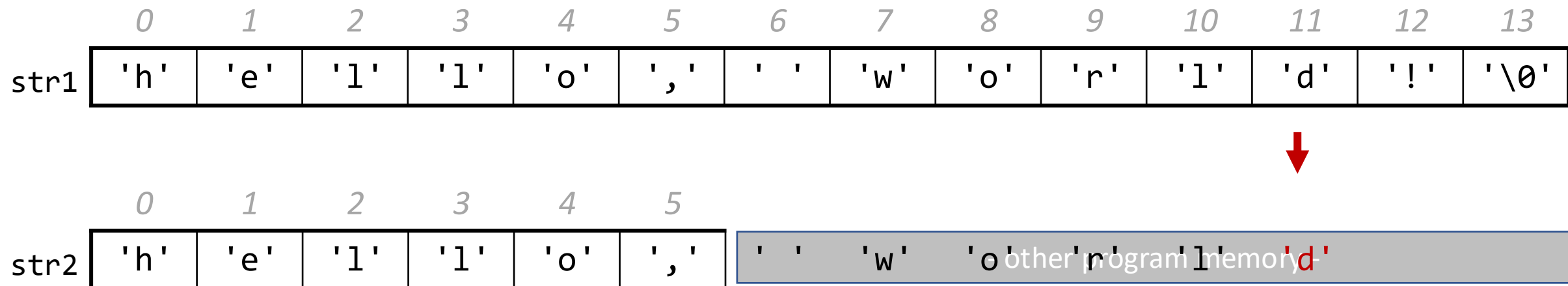
Copying Strings – Buffer Overflows

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



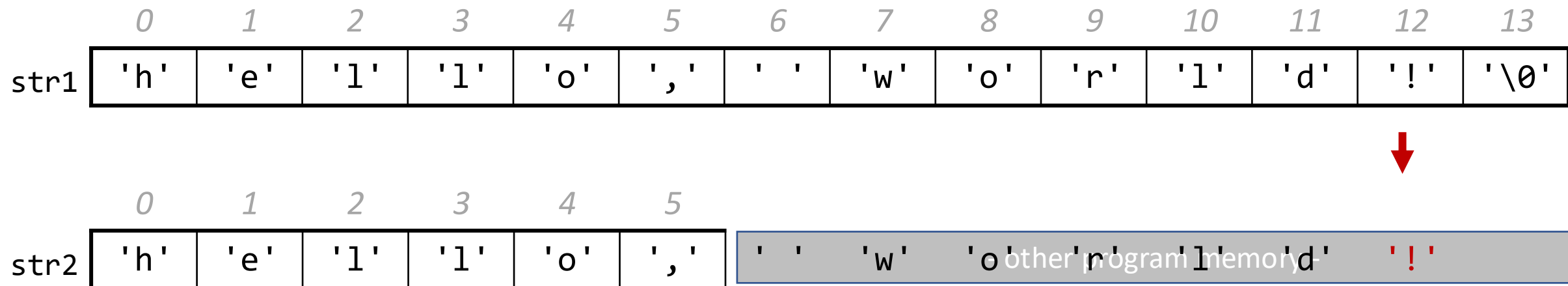
Copying Strings – Buffer Overflows

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



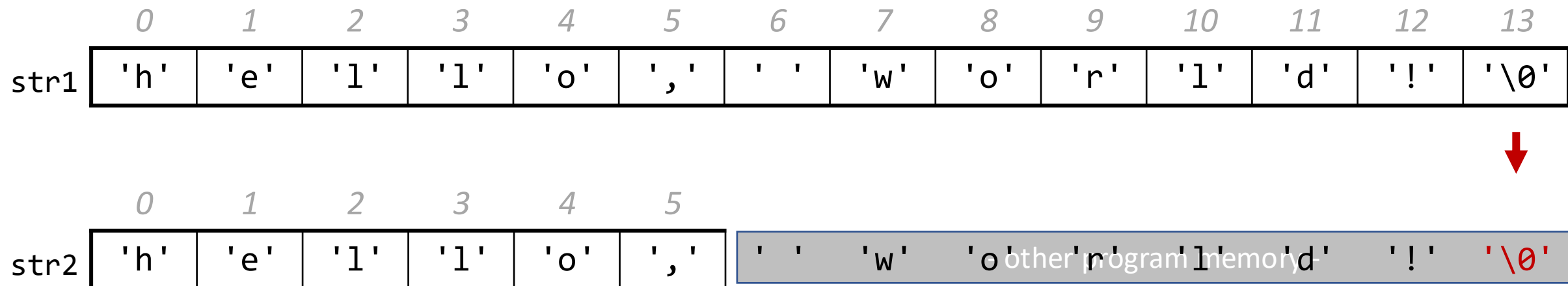
Copying Strings – Buffer Overflows

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



Copying Strings – Buffer Overflows

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



Copying Strings – Buffer Overflows

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



Copying Strings - strncpy

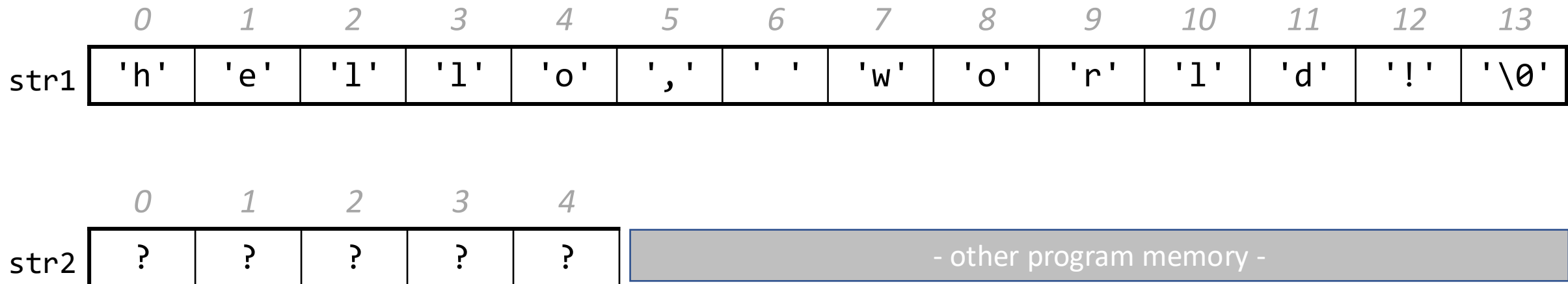
strncpy(dst, src, n): copies at most the first **n** bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **dst** will *not be null terminated!*

```
// copying "hello"  
char str2[5];  
strncpy(str2, "hello, world!", 5);    // doesn't copy '\0'!
```

If there is no null-terminating character, we may not be able to tell where the end of the string is anymore. E.g. `strlen` may continue reading into some other memory in search of `'\0'`!

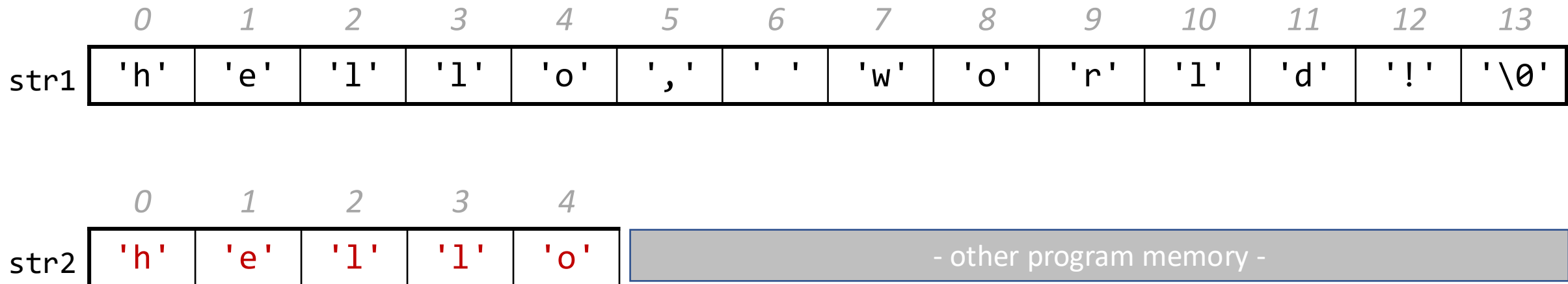
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



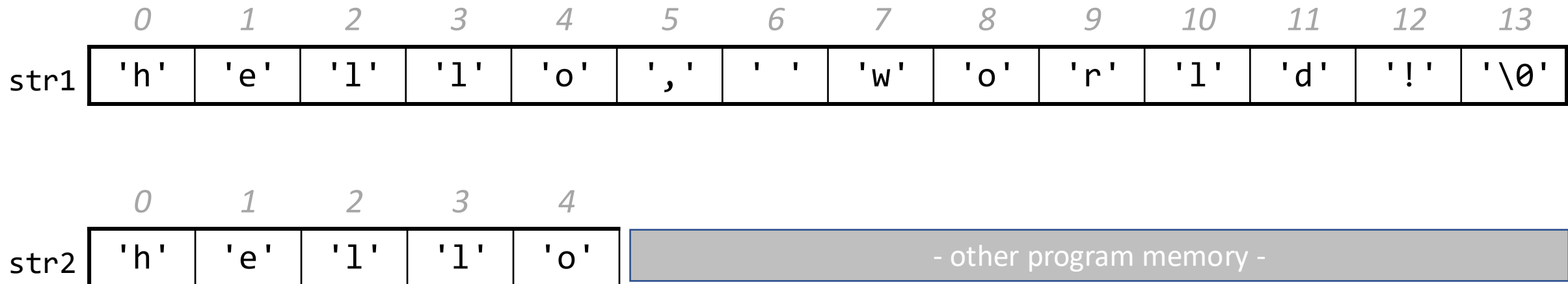
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



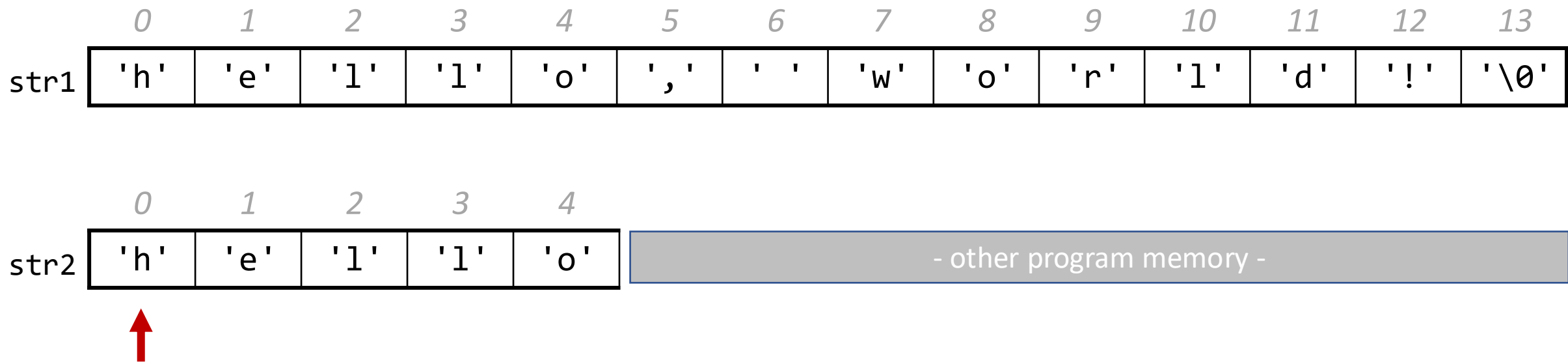
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



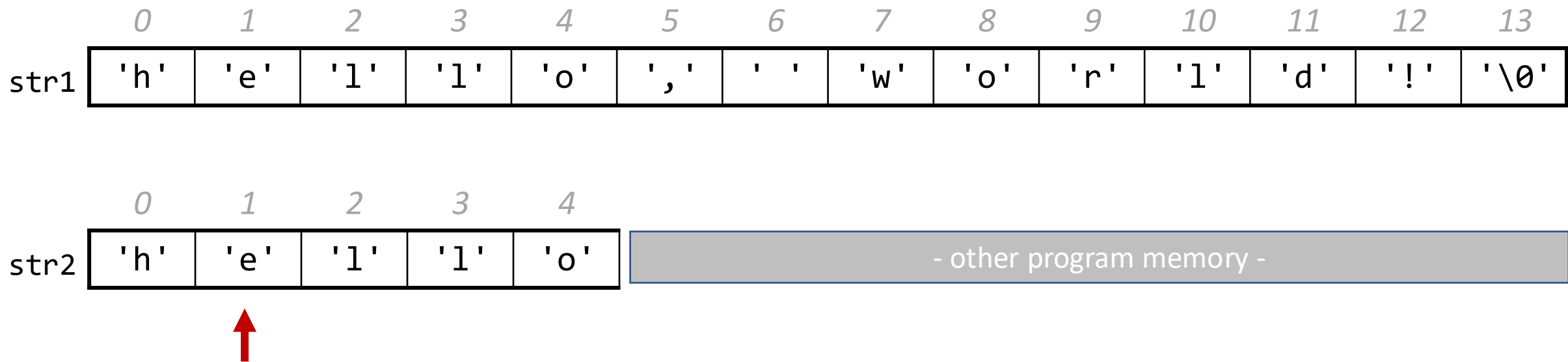
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



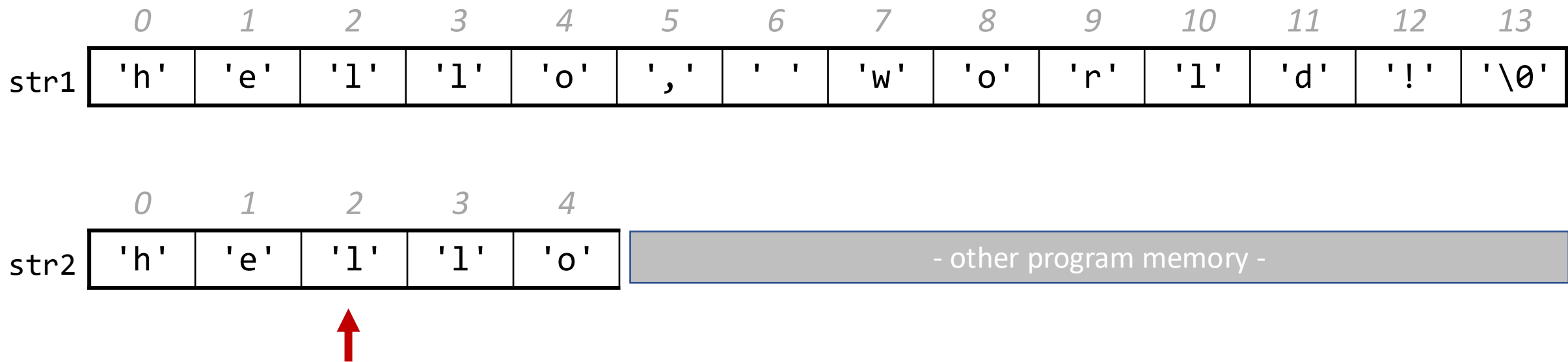
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



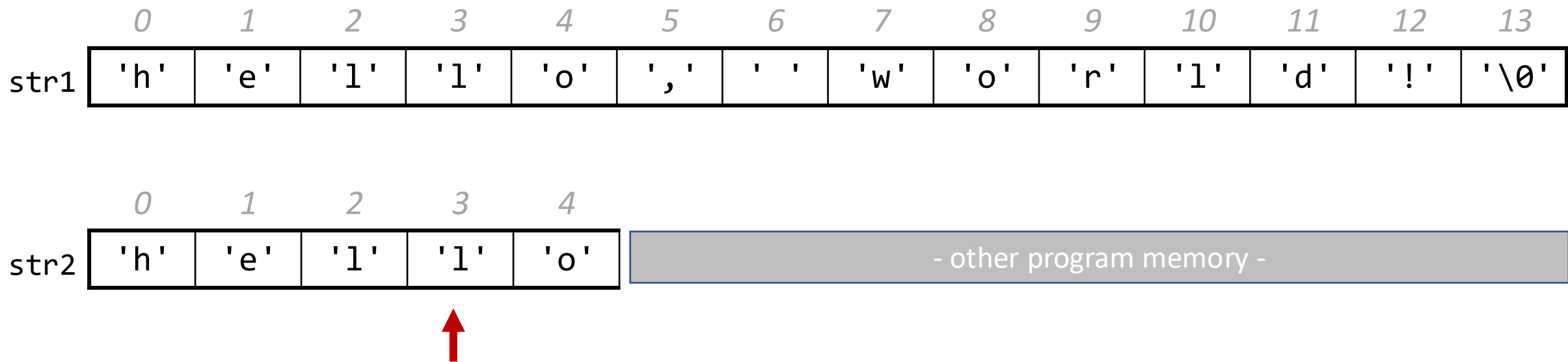
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



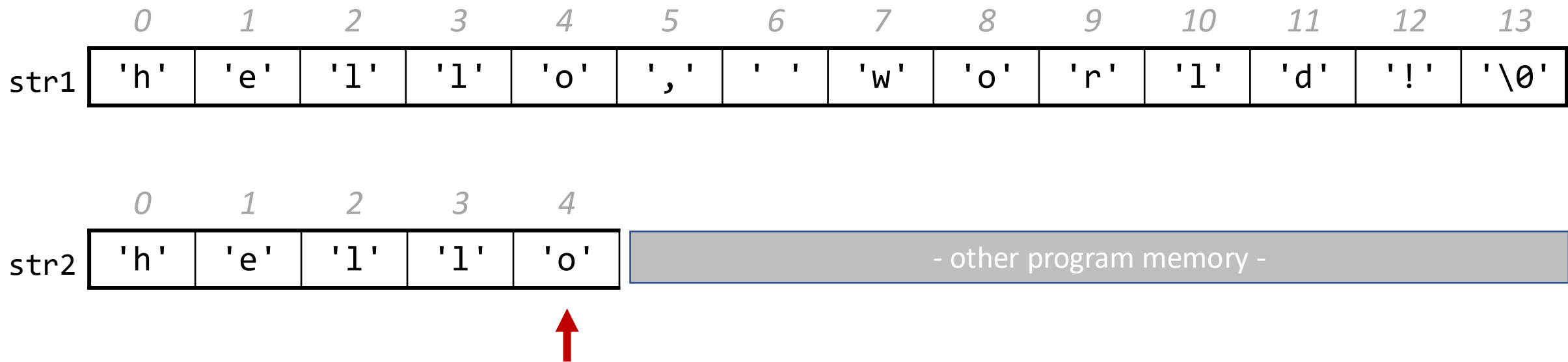
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



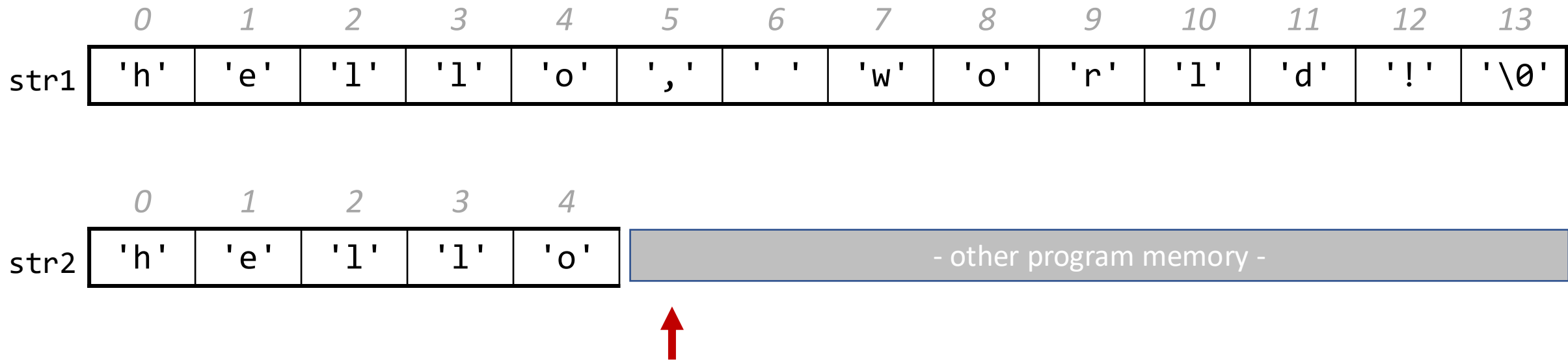
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



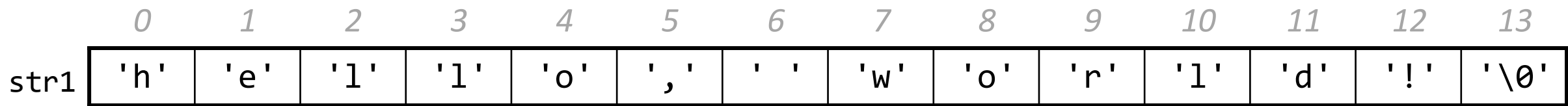
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



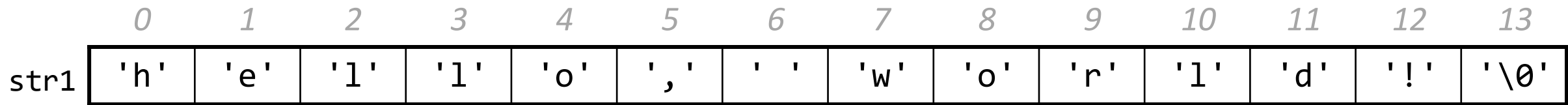
Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



Copying Strings - strncpy

```
char str2[5];  
strncpy(str2, "hello, world!", 5);  
int length = strlen(str2);
```



Copying Strings - strncpy

If necessary, we can add a null-terminating character ourselves.

```
// copying "hello"
char str2[6]; // room for string and '\0'
strncpy(str2, "hello, world!", 5); // doesn't copy '\0'!
str2[5] = '\0'; // add null-terminating char
```

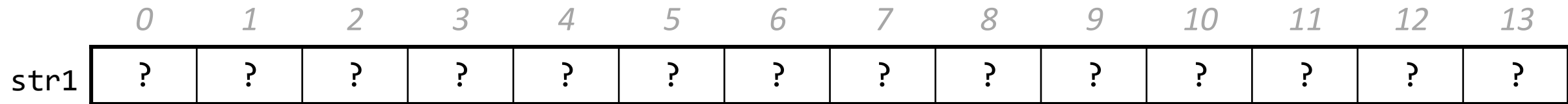
C Doesn't Automatically Initialize

Important note: C doesn't automatically initialize variables or values to a default value.

```
int x;    // contains garbage value  
char str[6]; // contains garbage characters
```

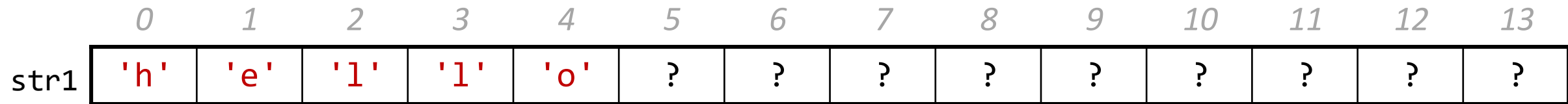
Copying Strings - strncpy

```
char str1[14];  
strncpy(str1, "hello there", 5);
```



Copying Strings - strncpy

```
char str1[14];  
strncpy(str1, "hello there", 5);
```



Copying Strings - strncpy

```
char str1[14];  
strncpy(str1, "hello there", 5);  
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	'?'	'?'	'?'	'?'	'?'	'?'	'?'	'?'	'?'

```
hello[?][?]J[?][?][?]
```

String Exercise

What is printed out by this program? (%zu is for “size_t”, \approx unsigned long)

```
1 int main(int argc, char *argv[]) {  
2     char str[9];  
3     strcpy(str, "Hi earth");  
4     str[2] = '\0';  
5     printf("str = %s, len = %zu\n",  
6           str, strlen(str));  
7     return 0;  
8 }
```

- A. str = Hi, len = 8
- B. str = Hi, len = 2
- C. str = Hi earth, len = 8
- D. str = Hi earth, len = 2
- E. None/other

Respond on Pollev:
pollev.com/cs107



L6 Q1: What is printed out by the example string program?

str = Hi, len = 8

0%

str = Hi, len = 2

0%

str = Hi earth, len = 8

0%

str = Hi earth, len = 2

0%

None/other

0%

Concatenating Strings

We cannot concatenate C strings using +. This adds addresses!

```
// e.g. param1 = 0x7f, param2 = 0x65
void doSomething(char *param1, char *param2) {
    printf("%s", param1 + param2);    // adds 0x7f and 0x65!
```

Instead, use **strcat**.

The string library: `str(n)cat`

`strcat(dst, src)`: concatenates the contents of **`src`** into the string **`dst`**.

`strncat(dst, src, n)`: same, but concats at most **`n`** bytes from **`src`**.

```
char str1[13];           // enough space for strings + '\0'
strcpy(str1, "hello ");
strcat(str1, "world!");  // removes old '\0', adds new '\0' at end
printf("%s", str1);     // hello world!
```

Both **`strcat`** and **`strncat`** remove the old `\0` and add a new one at the end.

Concatenating Strings

```
char str1[13];  
strcpy(str1, "hello ");  
char str2[7];  
strcpy(str2, "world!");  
  
strcat(str1, str2);
```



Concatenating Strings

```
char str1[13];  
strcpy(str1, "hello ");  
char str2[7];  
strcpy(str2, "world!");  
  
strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

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
```

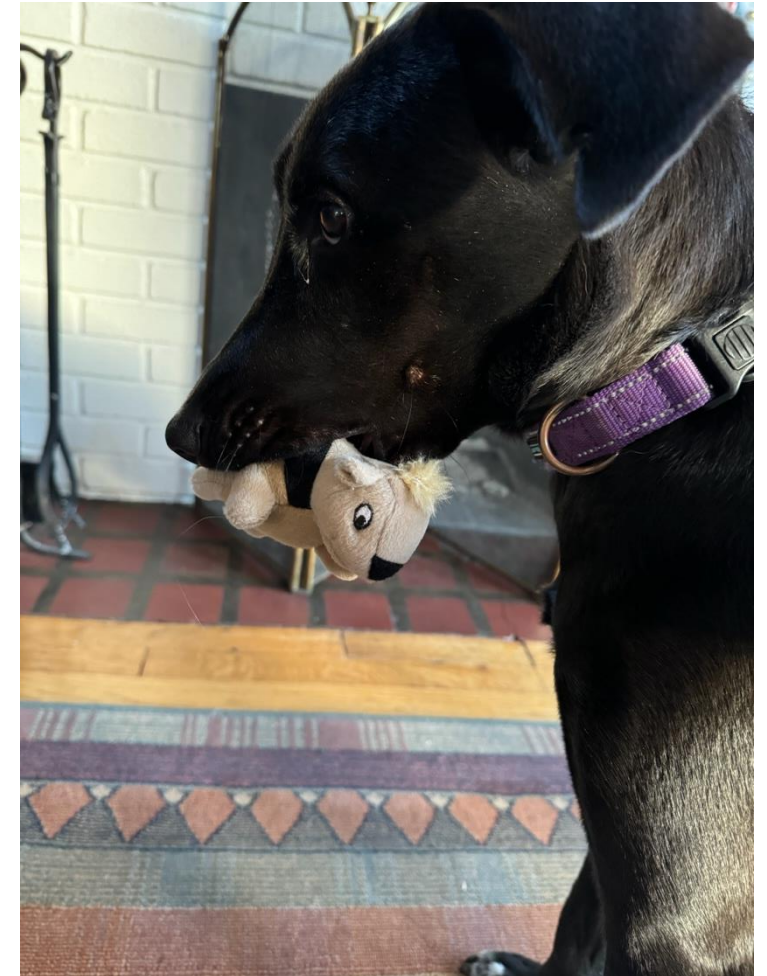
Goal: String Diamond

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
```

We can use this to print the top half of the diamond!
What about the bottom half?



Substrings and char *

You can also create a char * variable yourself that points to an address within in an existing string.

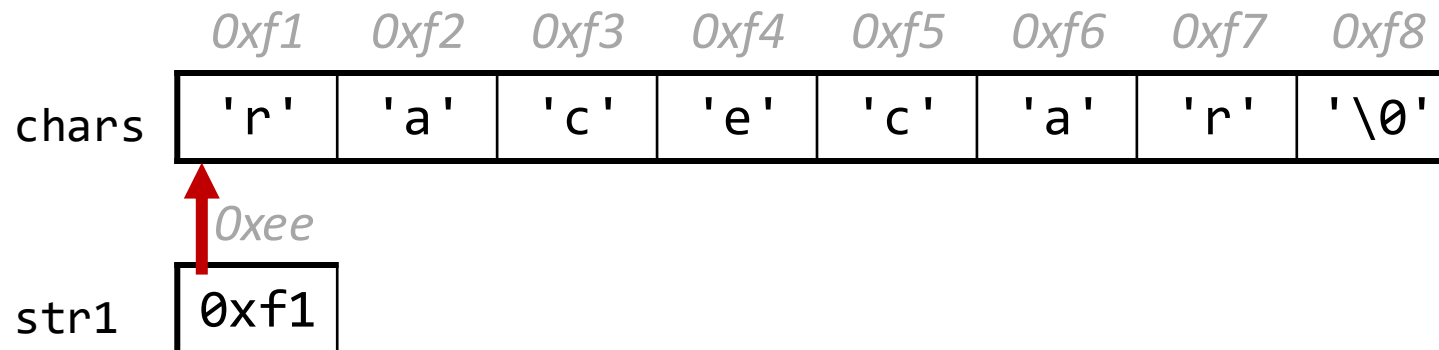
```
char myString[3];  
myString[0] = 'H';  
myString[1] = 'i';  
myString[2] = '\0';
```

```
char *otherStr = myString; // points to 'H'
```

Substrings

`char *s` (pointers to characters) *are strings*. We can use them to create substrings of larger strings.

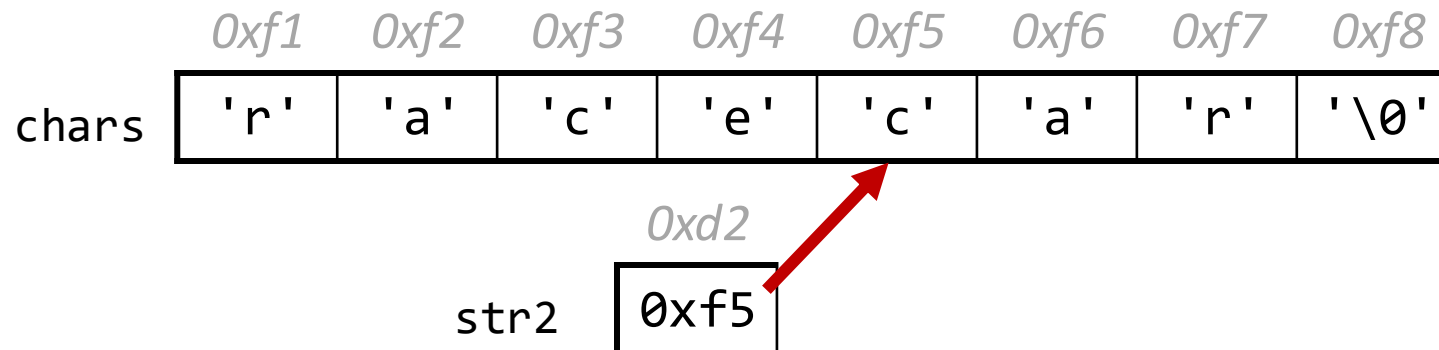
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
printf("%s\n", str1);           // racecar
```



Substrings

`char *s` (pointers to characters) *are strings*. We can use them to create substrings of larger strings - e.g. we can adjust the pointer to omit characters at the beginning.

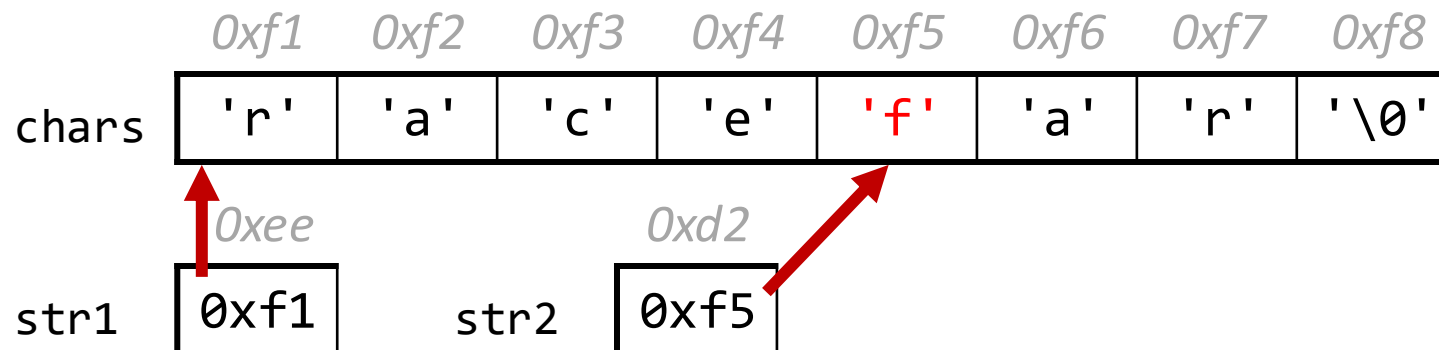
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars + 4;  
printf("%s\n", str2);           // car
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

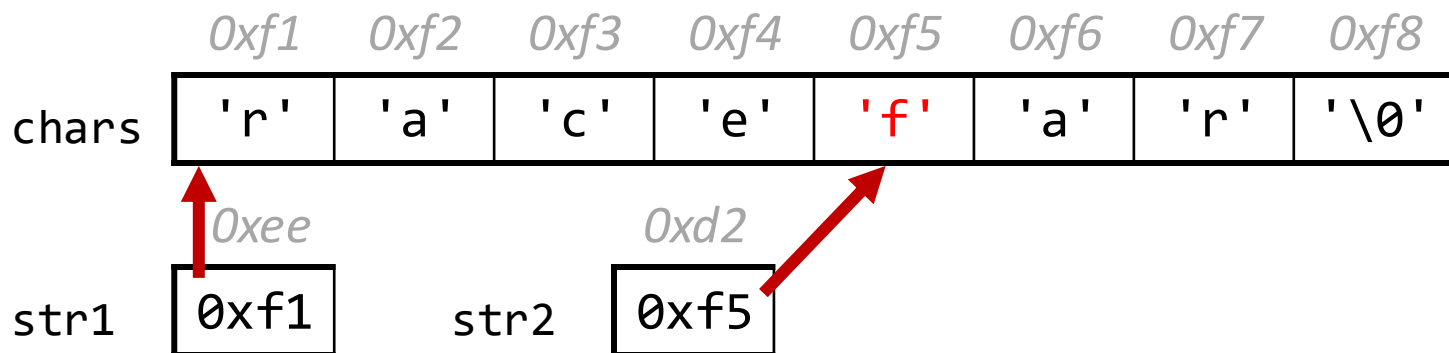
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
str2[0] = 'f';  
printf("%s %s\n", chars, str1);  
printf("%s\n", str2);
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
str2[0] = 'f';  
printf("%s %s\n", chars, str1);           // racefar racefar  
printf("%s\n", str2);                     // far
```



Substrings

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

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

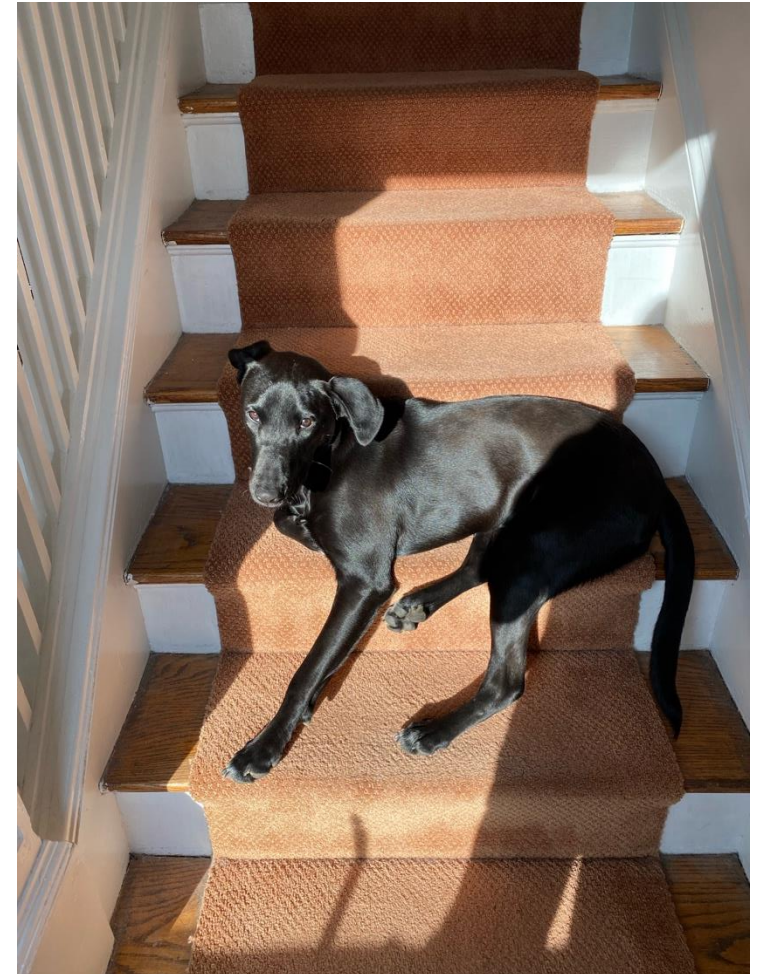
char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = '\0';
printf("%s\n", str1);           // racecar
printf("%s\n", str2);          // ace
```

Goal: 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
```



Demo: Diamond



diamond.c

Recap

- Characters
- Strings
- Common String Operations
 - Comparing
 - Copying
 - Concatenating
 - Substrings

Lecture 6 takeaway: C strings are null-terminated arrays of characters. We can manipulate them using string and pointer operations.

Next time: more strings

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Extra Practice

Copycat exercise

Challenge: implement **strcat** using other string functions.

```
char src[9];  
strcpy(src, "We Climb");  
char dst[200]; // lots of space  
strcpy(dst, "The Hill ");  
  
strcat(dst, src);
```

How could we replace a call to **strcat** with a call to **strcpy** instead?



Copycat exercise

Challenge: implement **strcat** using other string functions.

```
char src[9];  
strcpy(src, "We Climb");  
char dst[200]; // lots of space  
strcpy(dst, "The Hill ");  
  
strcat(dst, src);  equivalent strcpy(dst + strlen(dst), src);
```

Strings Practice

```
1 char buf[9];
2 strcpy(buf, "Potatoes");
3 printf("%s\n", buf);
4 char *word = buf + 2;
5 strncpy(word, "mat", 3);
6 printf("%s\n", buf);
```

Line 6: What is printed?

- A. matoes
- B. mattoes
- C. Pomat
- D. Pomatoes
- E. Something else
- F. Compile error

