

# CS107, Lecture 4

## C Strings

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

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

# Lecture Plan

• Characters	4
• Strings	11
• Common String Operations	16
• Comparing	19
• Copying	21
• Concatenating	79
• Substrings	90
• <b>Practice: Diamond</b>	99
• Live Session	105

# Lecture Plan

• Characters	4
• Strings	11
• Common String Operations	16
• Comparing	19
• Copying	21
• Concatenating	79
• Substrings	90
• <b>Practice: Diamond</b>	99
• Live Session	105

# 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 (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	4
• <b>Strings</b>	<b>11</b>
• Common String Operations	16
• Comparing	19
• Copying	21
• Concatenating	79
• Substrings	90
• <b>Practice: Diamond</b>	<b>99</b>
• Live Session	105

# 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.

"Hello"	<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>
	<i>char</i>	'H'	'e'	'l'	'l'	'o'	'\0'

'\0' is the **null-terminating character**; you always need to allocate one extra space in an array for it.

# 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!  
We should save the value if we 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	4
• Strings	11
• Common String Operations	16
• Comparing	19
• Copying	21
• Concatenating	79
• Substrings	90
• <b>Practice: Diamond</b>	99
• Live Session	105

# 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 <b><i>str1</i></b> comes before <b><i>str2</i></b> in alphabet, >0 if <b><i>str1</i></b> comes after <b><i>str2</i></b> in alphabet. <b><i>strncmp</i></b> stops comparing after at most <b><i>n</i></b> 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 <b><i>ch</i></b> in <b><i>str</i></b> , or <b><i>NULL</i></b> if <b><i>ch</i></b> was not found in <b><i>str</i></b> . <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 <b><i>needle</i></b> in <b><i>haystack</i></b> , or <b><i>NULL</i></b> if <b><i>needle</i></b> was not found in <b><i>haystack</i></b> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	copies characters in <b><i>src</i></b> to <b><i>dst</i></b> , including null-terminating character. Assumes enough space in <b><i>dst</i></b> . Strings must not overlap. <b><i>strncpy</i></b> stops after at most <b><i>n</i></b> 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 <b><i>src</i></b> onto the end of <b><i>dst</i></b> . <b><i>strncat</i></b> stops concatenating after at most <b><i>n</i></b> 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>	<b><i>strspn</i></b> returns the length of the initial part of <b><i>str</i></b> which contains <u>only</u> characters in <b><i>accept</i></b> . <b><i>strcspn</i></b> returns the length of the initial part of <b><i>str</i></b> which does <u>not</u> contain any characters in <b><i>reject</i></b> .

# 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 <b><i>str1</i></b> comes before <b><i>str2</i></b> in alphabet, >0 if <b><i>str1</i></b> comes after <b><i>str2</i></b> in alphabet. <b><i>strncmp</i></b> stops comparing after at most <b><i>n</i></b> 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 <b><i>ch</i></b> in <b><i>str</i></b> , or <b><i>NULL</i></b> if <b><i>ch</i></b> was not found in <b><i>str</i></b> . <code>strrchr</code> find the last occurrence.
<code>strstr(<i>haystack</i>, <i>needle</i>)</code>	returns a pointer to the first occurrence of <b><i>needle</i></b> in <b><i>haystack</i></b> , or <b><i>NULL</i></b> if not found in <b><i>haystack</i></b> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	Assumes enough space in <b><i>dst</i></b> . Strings must not overlap. <b><i>strncpy</i></b> stops after at most <b><i>n</i></b> 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 <b><i>src</i></b> onto the end of <b><i>dst</i></b> . <b><i>strncat</i></b> stops concatenating after at most <b><i>n</i></b> 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>	<b><i>strspn</i></b> returns the length of the initial part of <b><i>str</i></b> which contains <u>only</u> characters in <b><i>accept</i></b> . <b><i>strcspn</i></b> returns the length of the initial part of <b><i>str</i></b> which does <u>not</u> contain any characters in <b><i>reject</i></b> .

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.

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

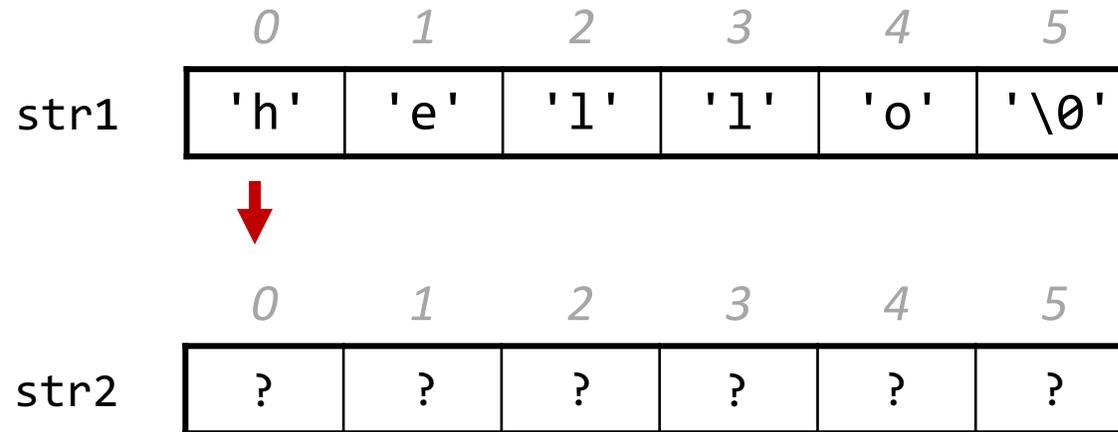
```
char str2[6];  
strcpy(str2, str1);  
str2[0] = 'c';
```

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

# Copying Strings - strcpy

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

```
char str2[6];  
strcpy(str2, str1);
```



# Copying Strings - strcpy

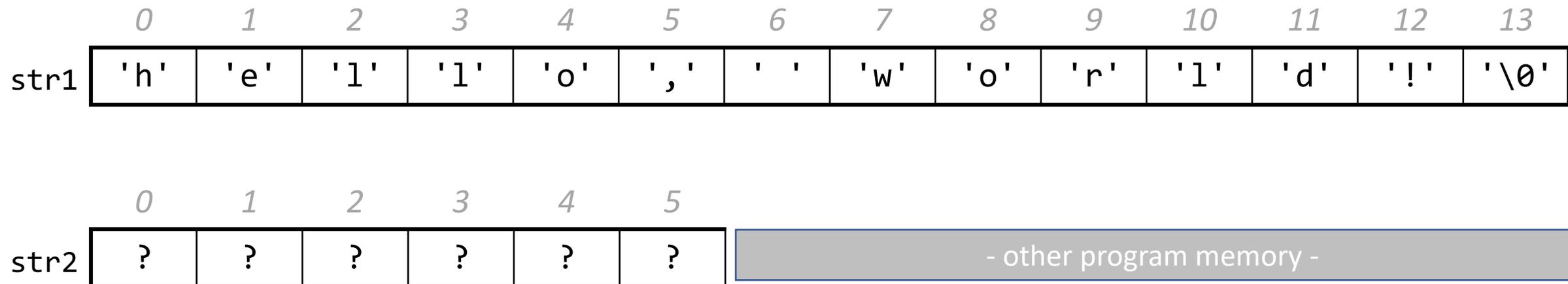
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
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!

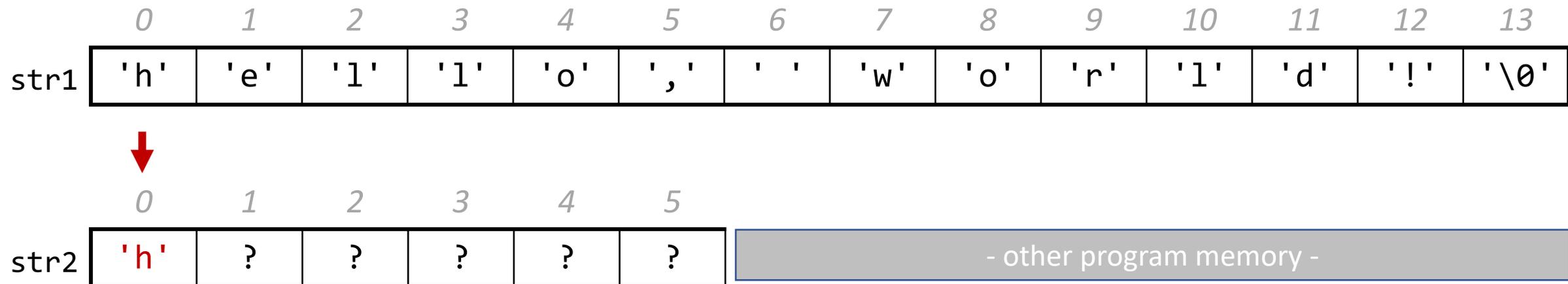
# Copying Strings – Buffer Overflows

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



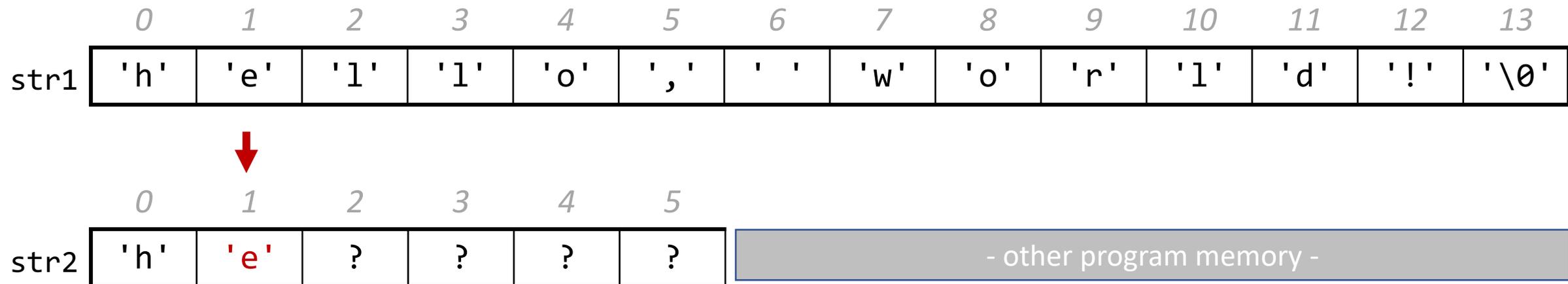
# Copying Strings – Buffer Overflows

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



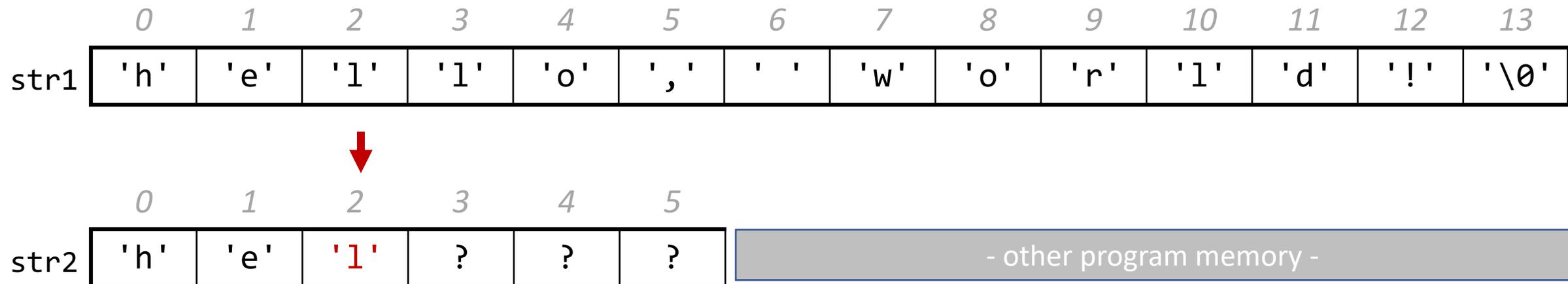
# Copying Strings – Buffer Overflows

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



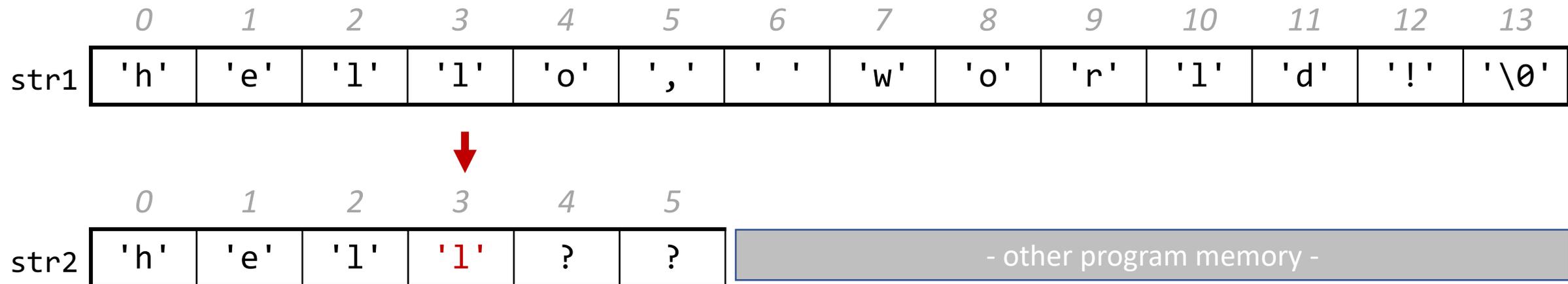
# Copying Strings – Buffer Overflows

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



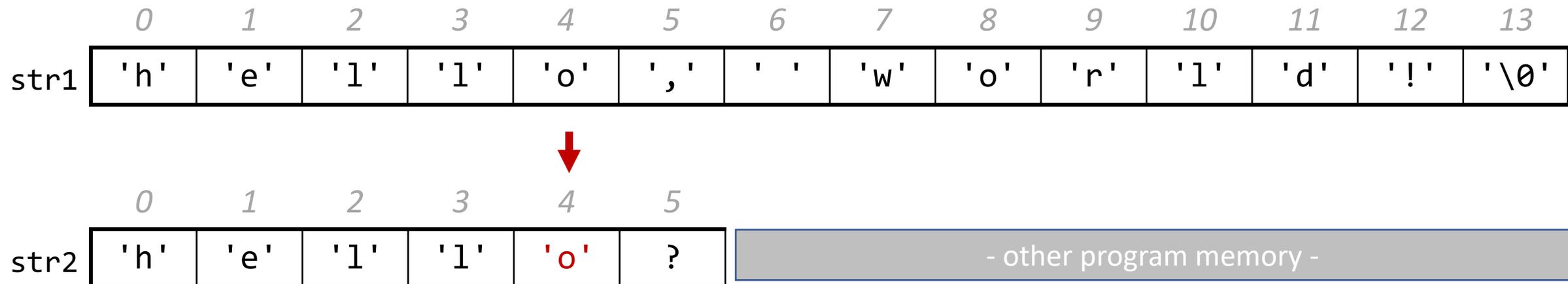
# Copying Strings – Buffer Overflows

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



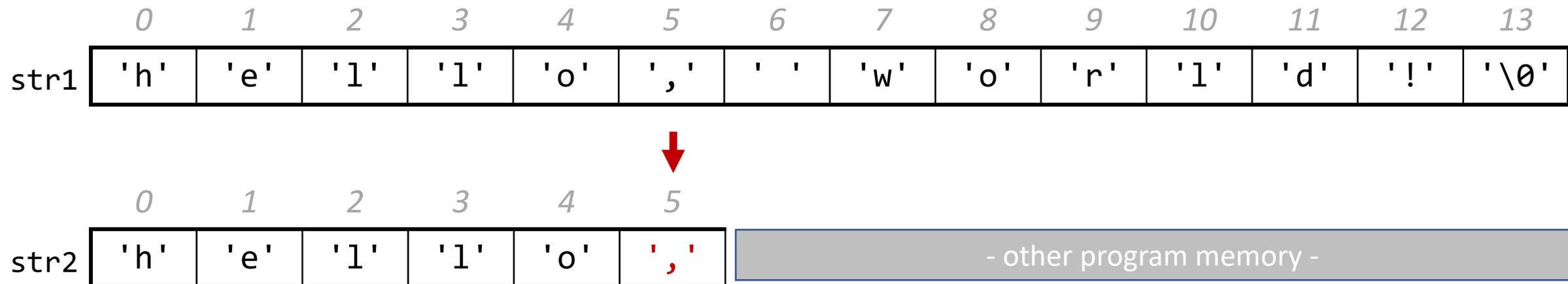
# Copying Strings – Buffer Overflows

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



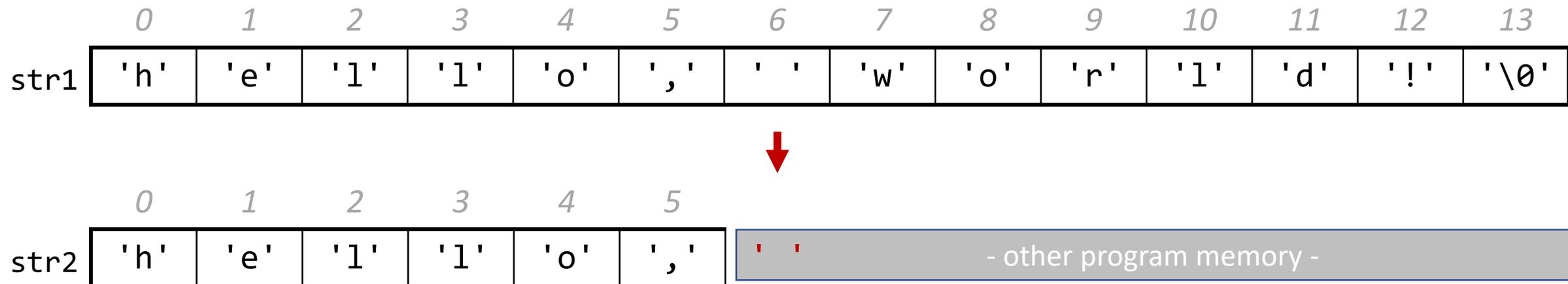
# Copying Strings – Buffer Overflows

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



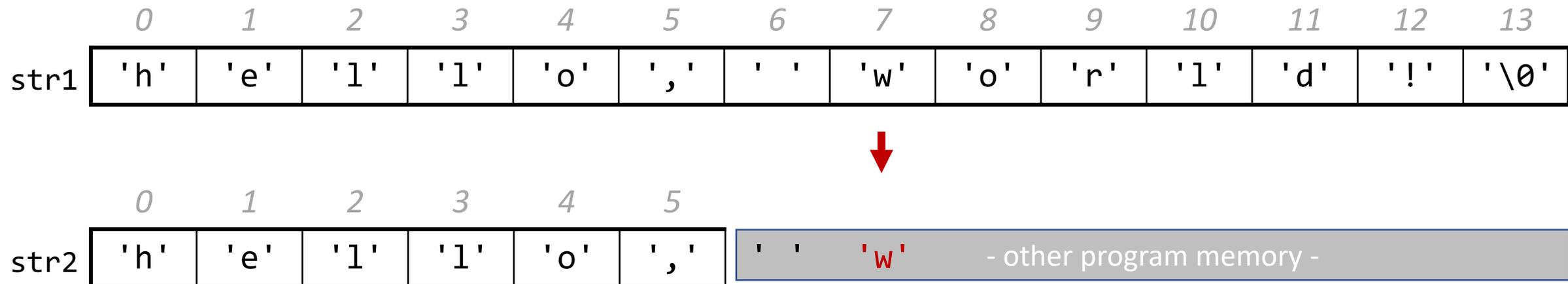
# Copying Strings – Buffer Overflows

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



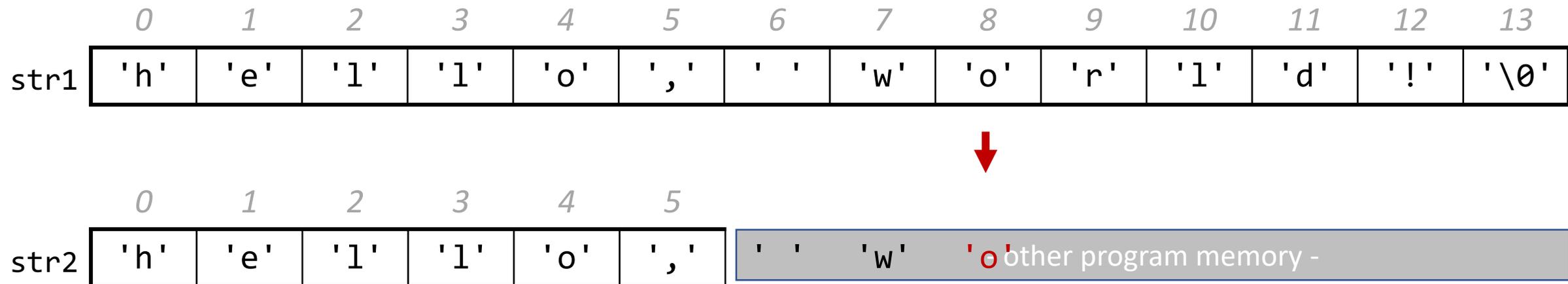
# Copying Strings – Buffer Overflows

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



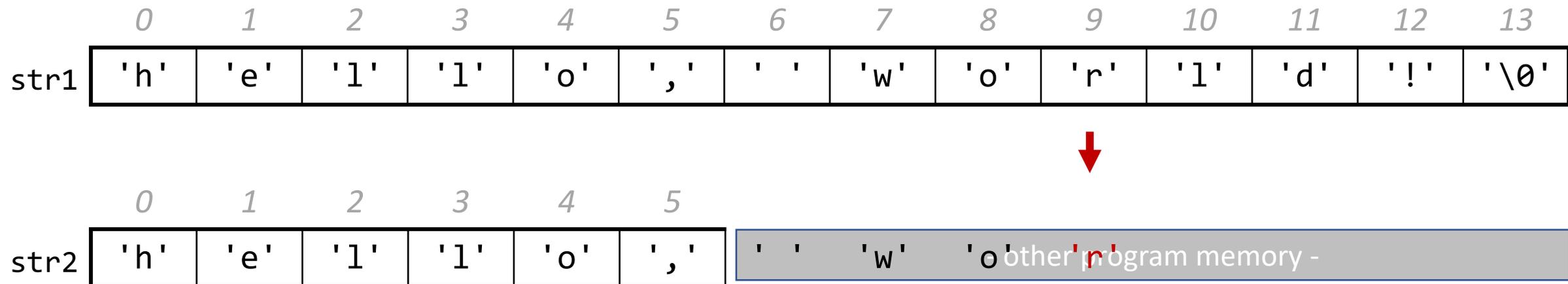
# Copying Strings – Buffer Overflows

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



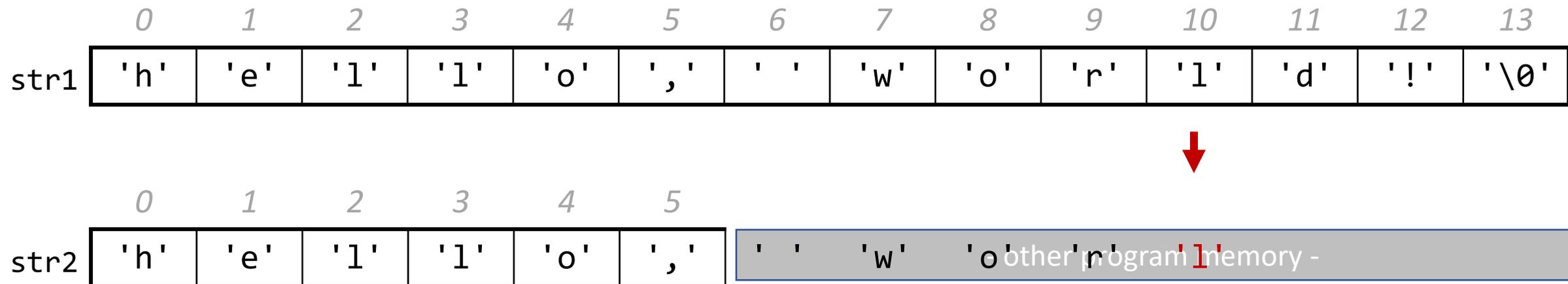
# Copying Strings – Buffer Overflows

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



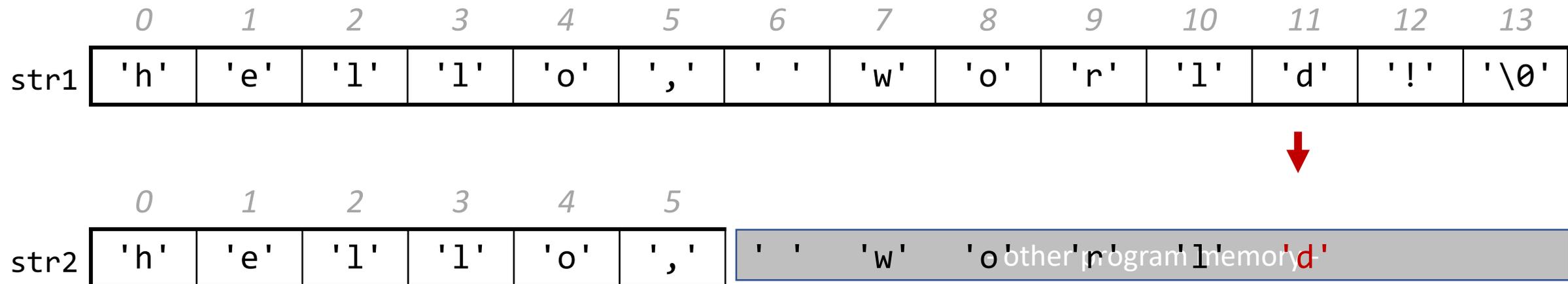
# Copying Strings – Buffer Overflows

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



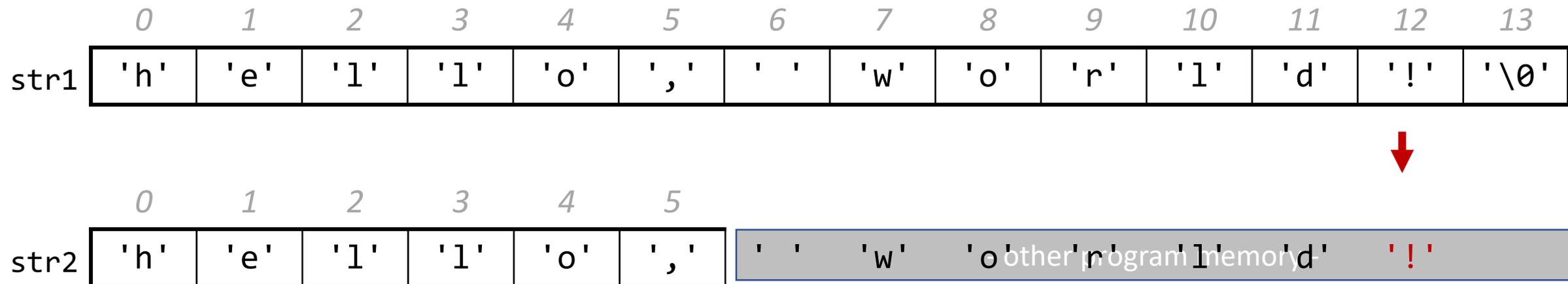
# Copying Strings – Buffer Overflows

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



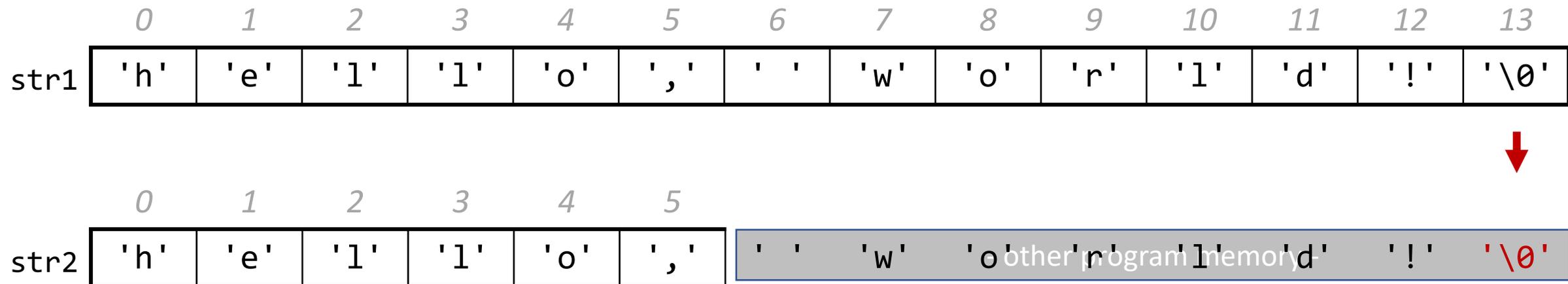
# Copying Strings – Buffer Overflows

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



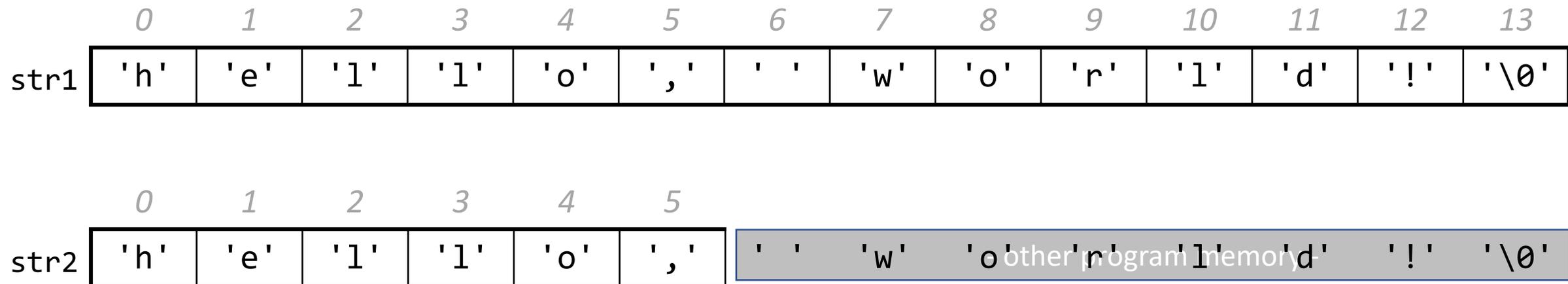
# Copying Strings – Buffer Overflows

```
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

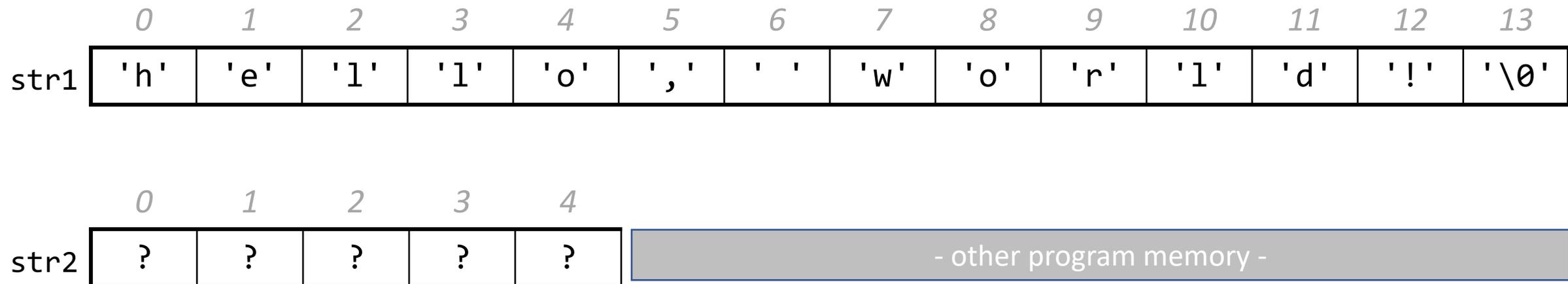
**strncpy(dst, src, n)**: copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **dst** will *not be null terminated!*

```
// 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'`!

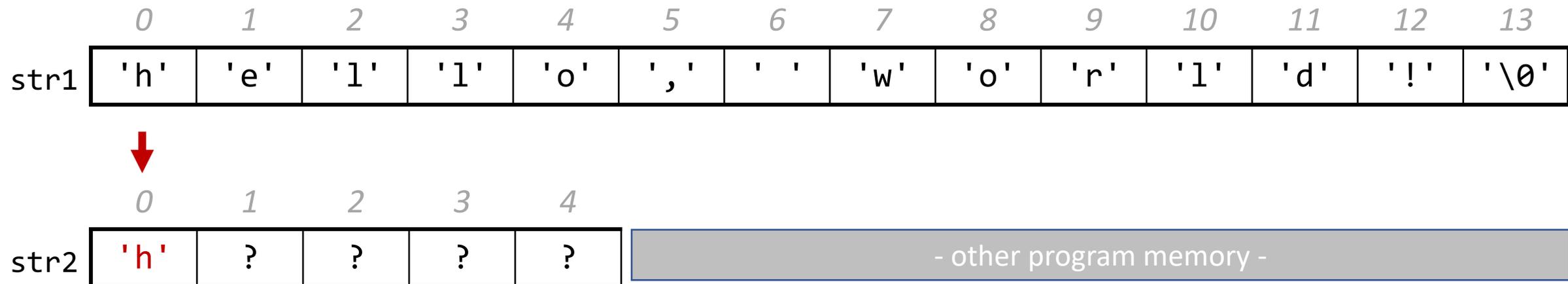
# Copying Strings - strncpy

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



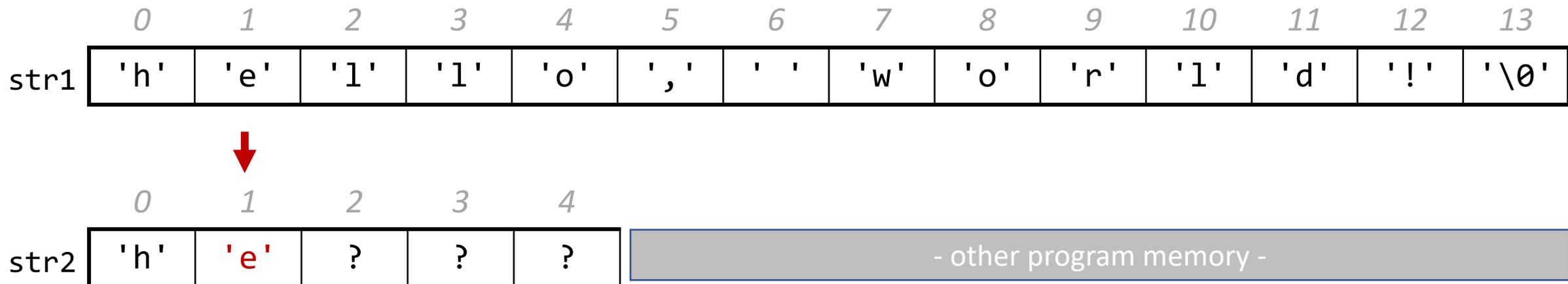
# Copying Strings - strncpy

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



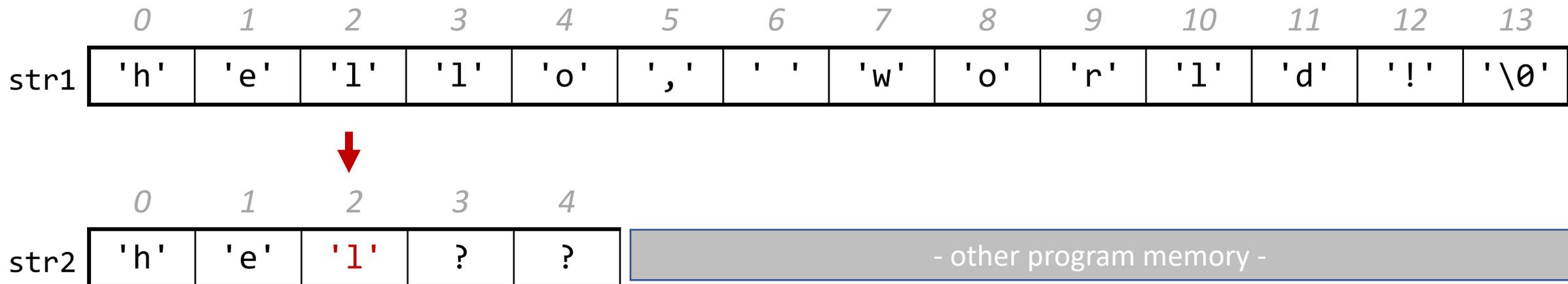
# Copying Strings - strncpy

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



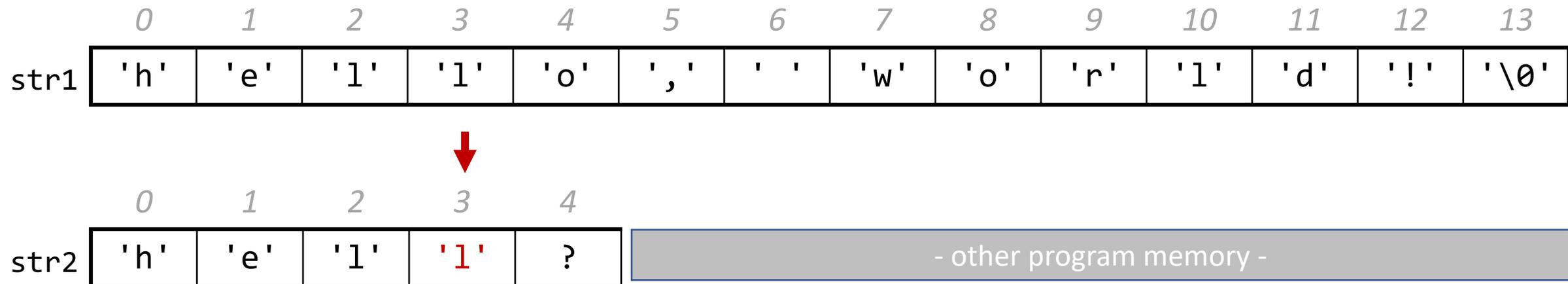
# Copying Strings - strncpy

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



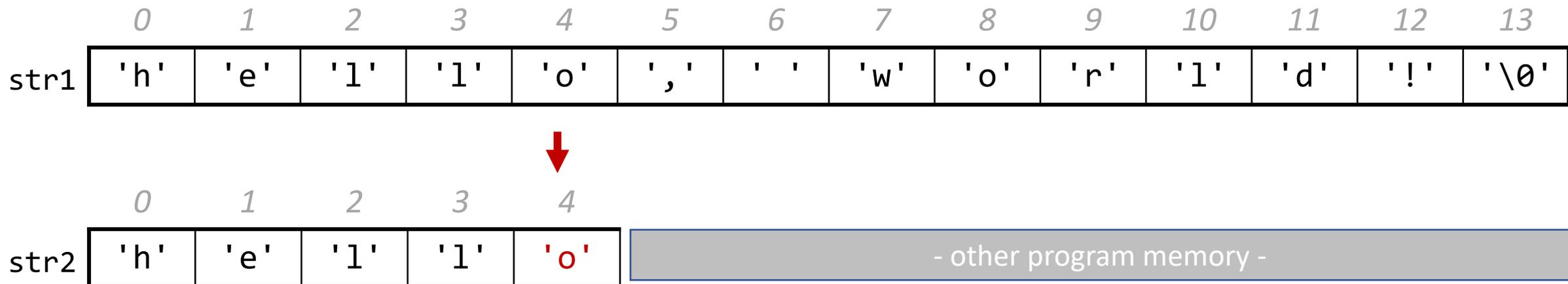
# Copying Strings - strncpy

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



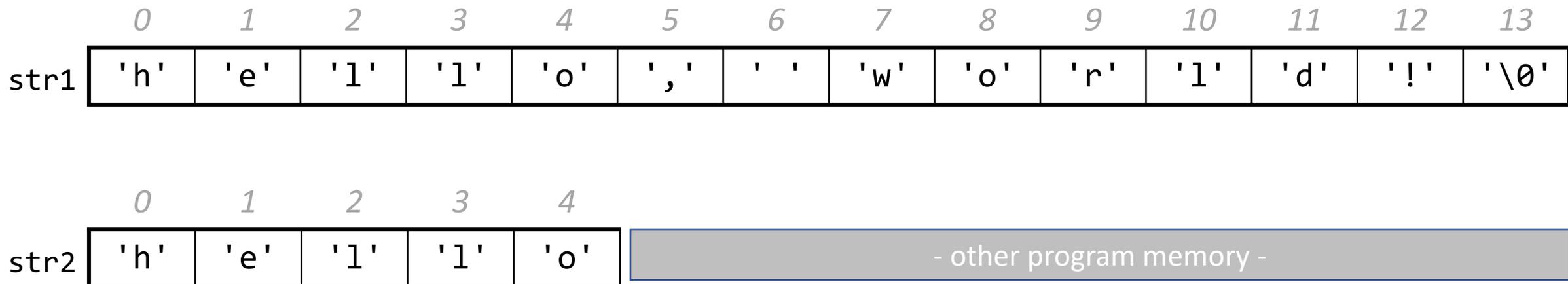
# Copying Strings - strncpy

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



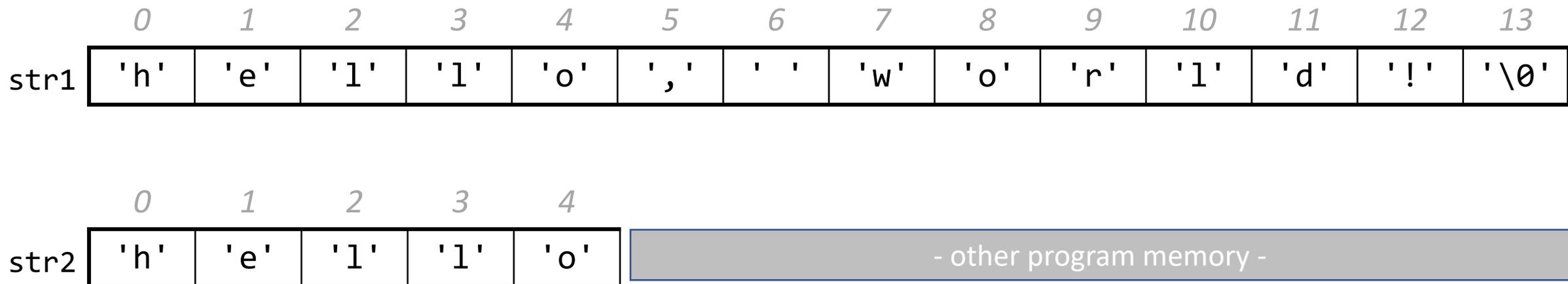
# Copying Strings - strncpy

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



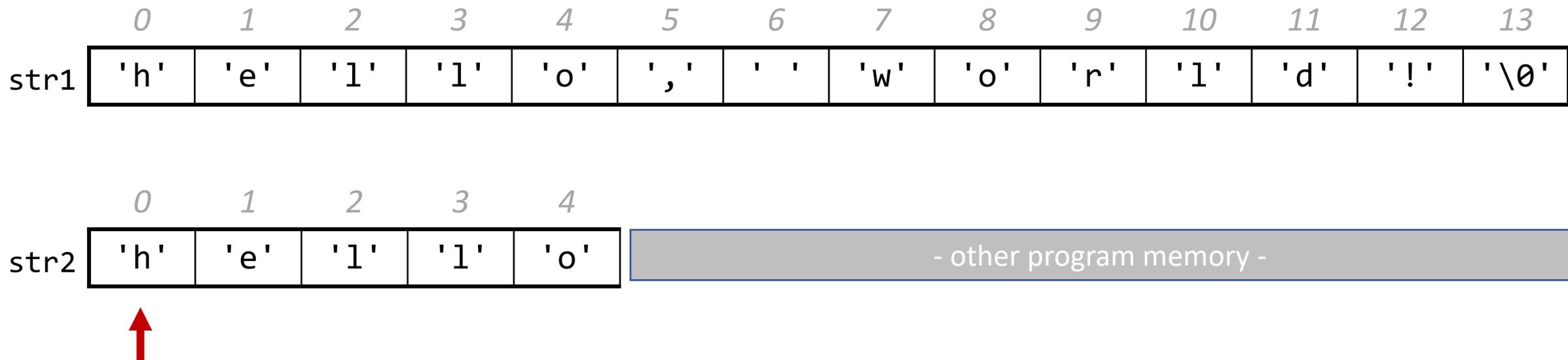
# Copying Strings - strncpy

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



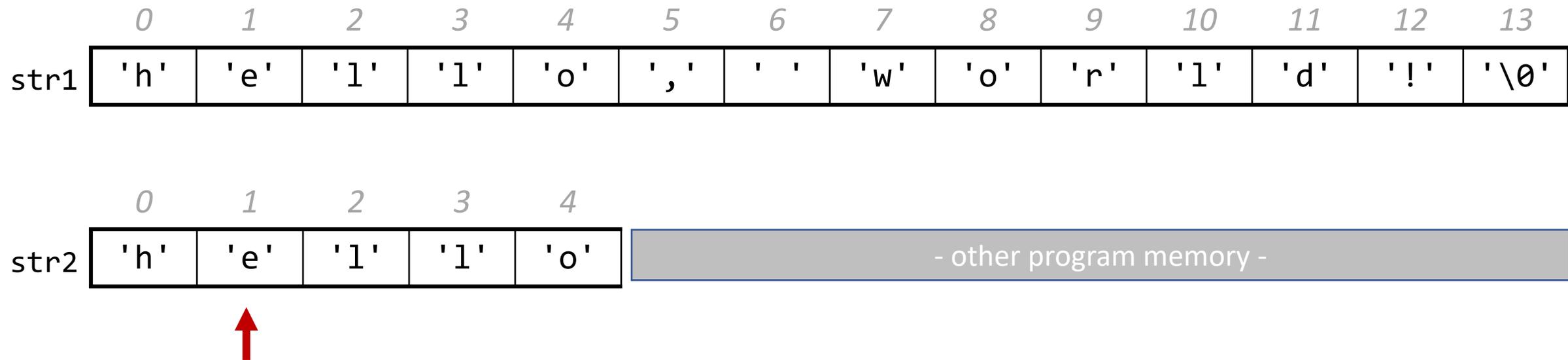
# Copying Strings - strncpy

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



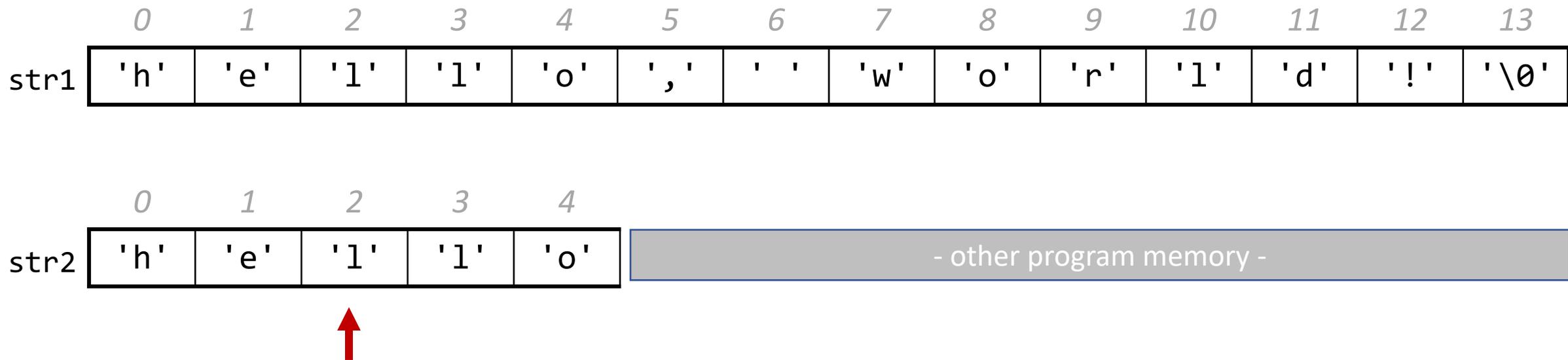
# Copying Strings - strncpy

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



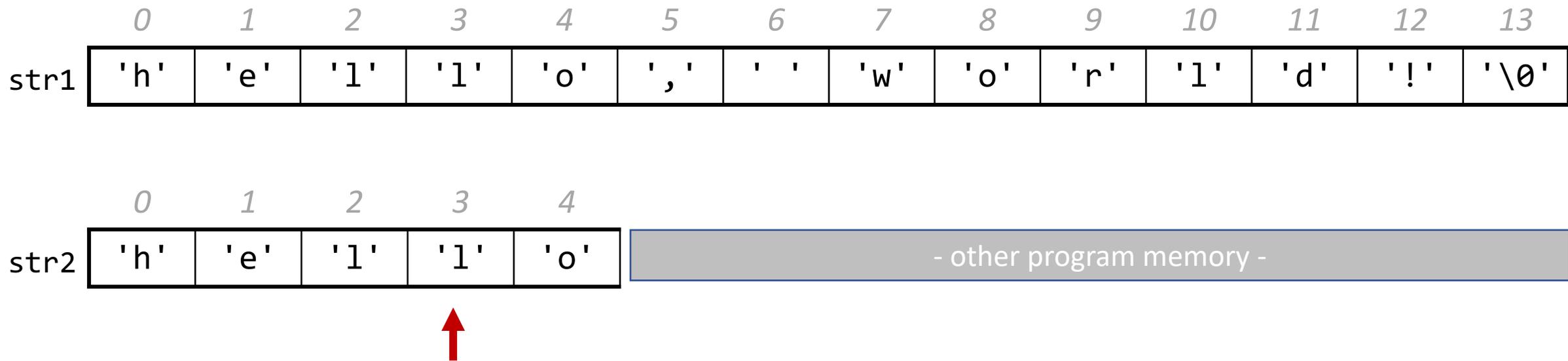
# Copying Strings - strncpy

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



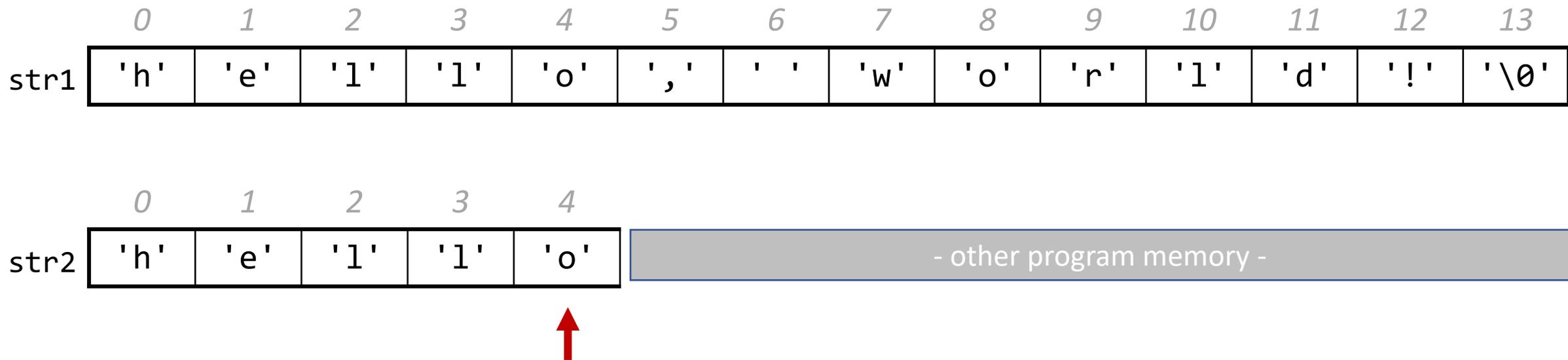
# Copying Strings - strncpy

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



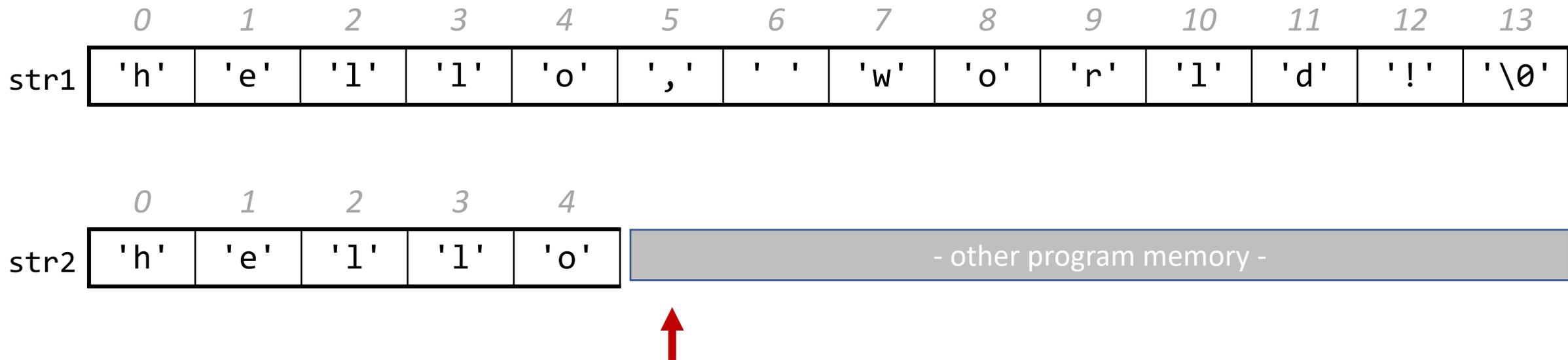
# Copying Strings - strncpy

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



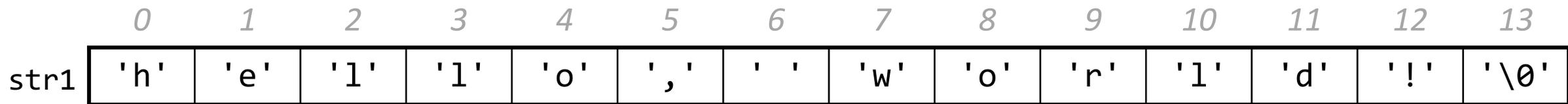
# Copying Strings - strncpy

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



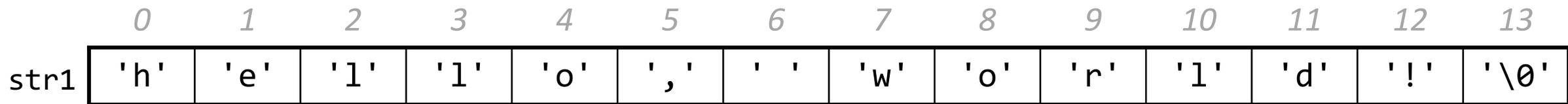
# Copying Strings - strncpy

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



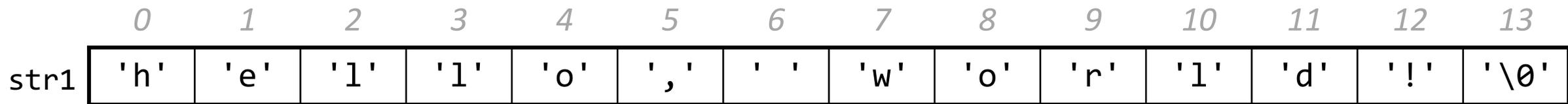
# Copying Strings - strncpy

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



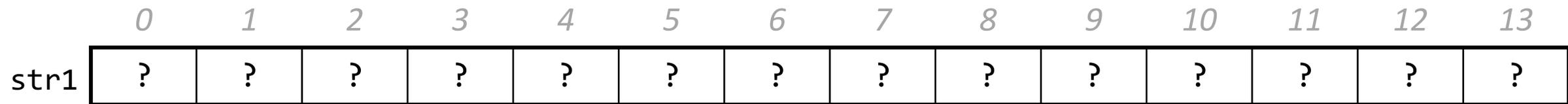
# Copying Strings - strncpy

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



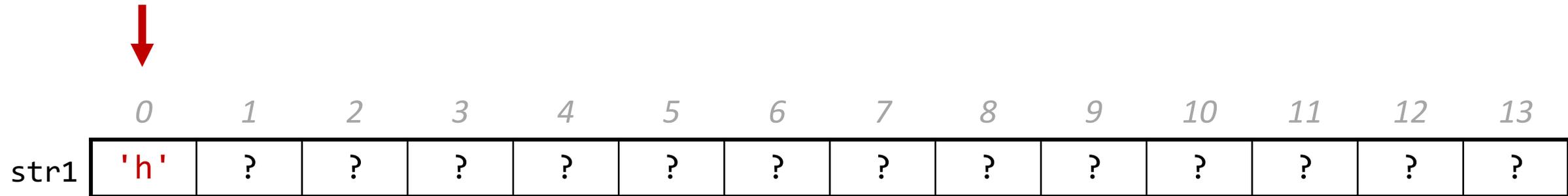
# Copying Strings - strncpy

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



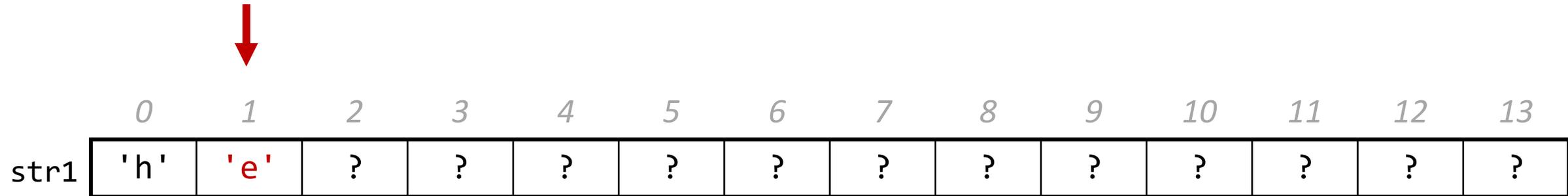
# Copying Strings - strncpy

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



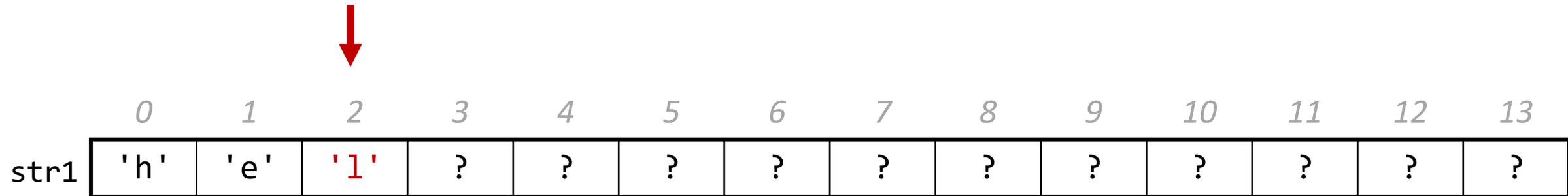
# Copying Strings - strncpy

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



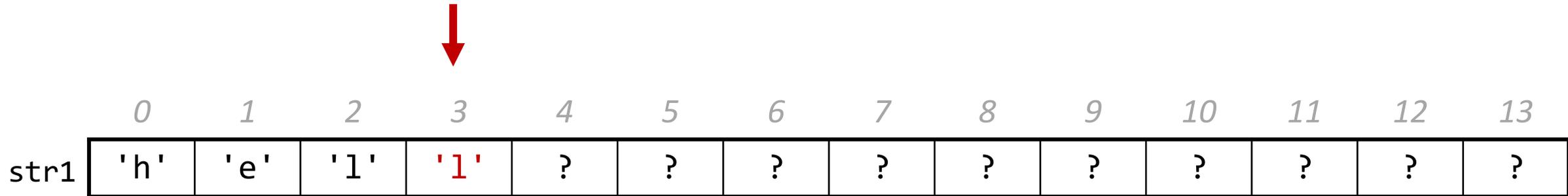
# Copying Strings - strncpy

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



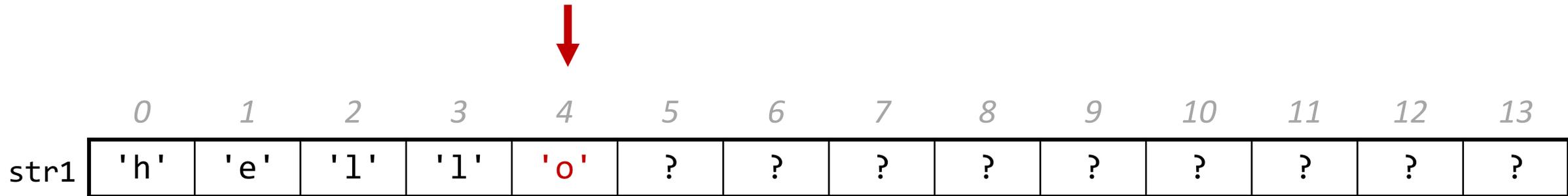
# Copying Strings - strncpy

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



# Copying Strings - strncpy

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



# Copying Strings - strncpy

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

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

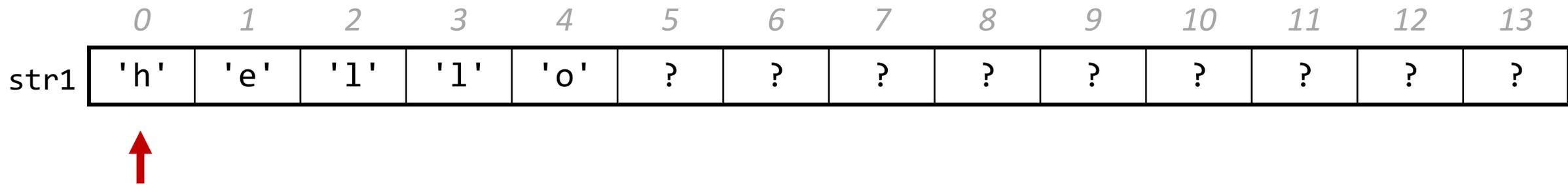
# Copying Strings - strncpy

```
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'	'?'	'?'	'?'	'?'	'?'	'?'	'?'	'?'	'?'

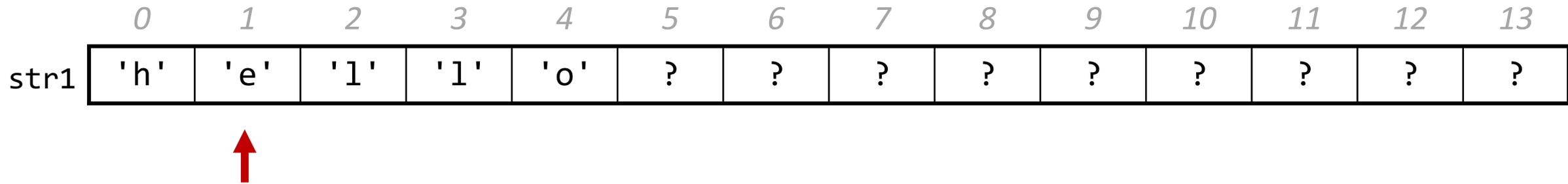
# Copying Strings - strncpy

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



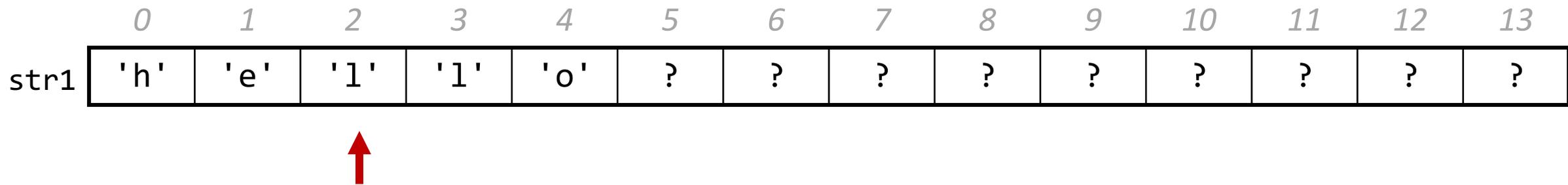
# Copying Strings - strncpy

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



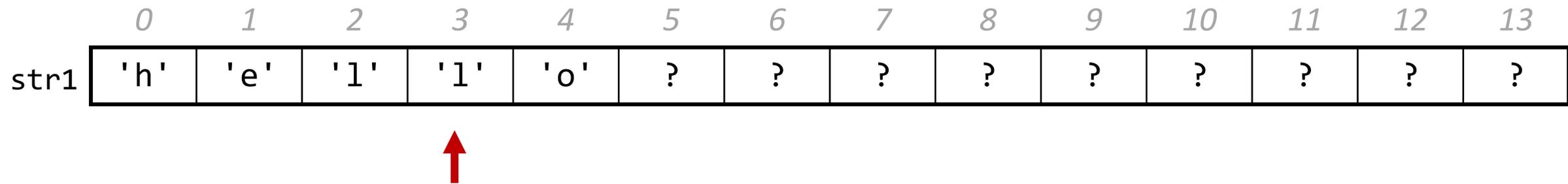
# Copying Strings - strncpy

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



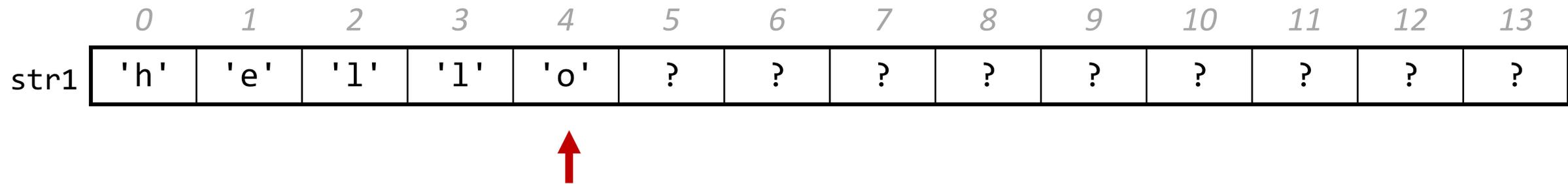
# Copying Strings - strncpy

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



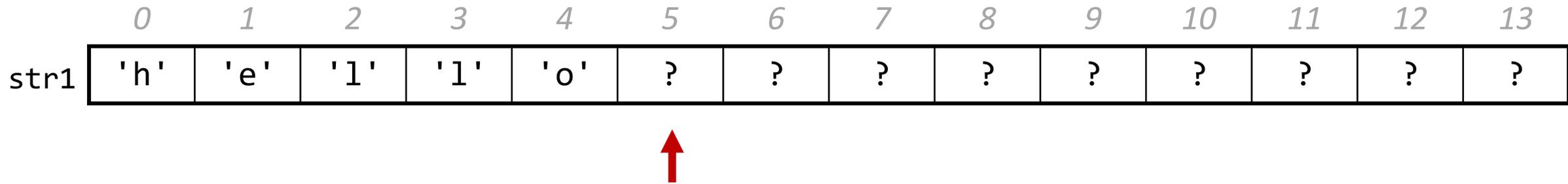
# Copying Strings - strncpy

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



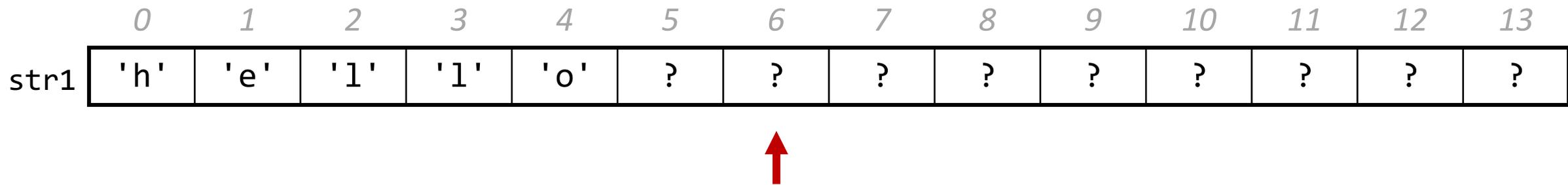
# Copying Strings - strncpy

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



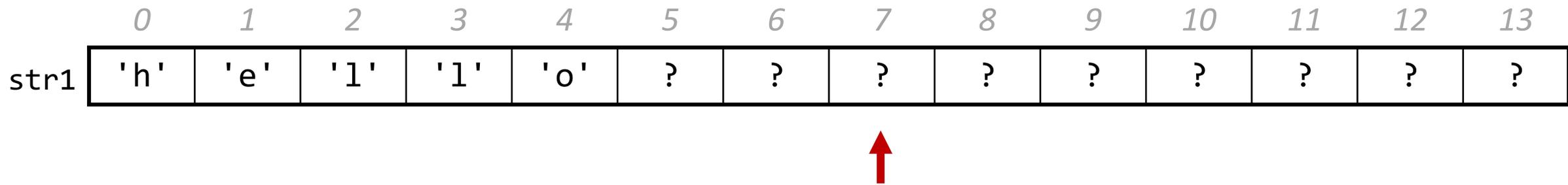
# Copying Strings - strncpy

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



# Copying Strings - strncpy

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



# 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 ? ? ?
```

# 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
```

# String Copying Exercise

What value should go in the blank at right?

- A. 4
- B. 5
- C. 6
- D. 12
- E. strlen("hello")
- F. Something else

```
char str[_____];  
strcpy(str, "hello");
```

# String Exercise

What is printed out by the following program?

```
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



# 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);
```

	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>	<i>11</i>	<i>12</i>
str1	'h'	'e'	'l'	'l'	'o'	' '	'\0'	'?	'?	'?	'?	'?	'?

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

# Concatenating Strings

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

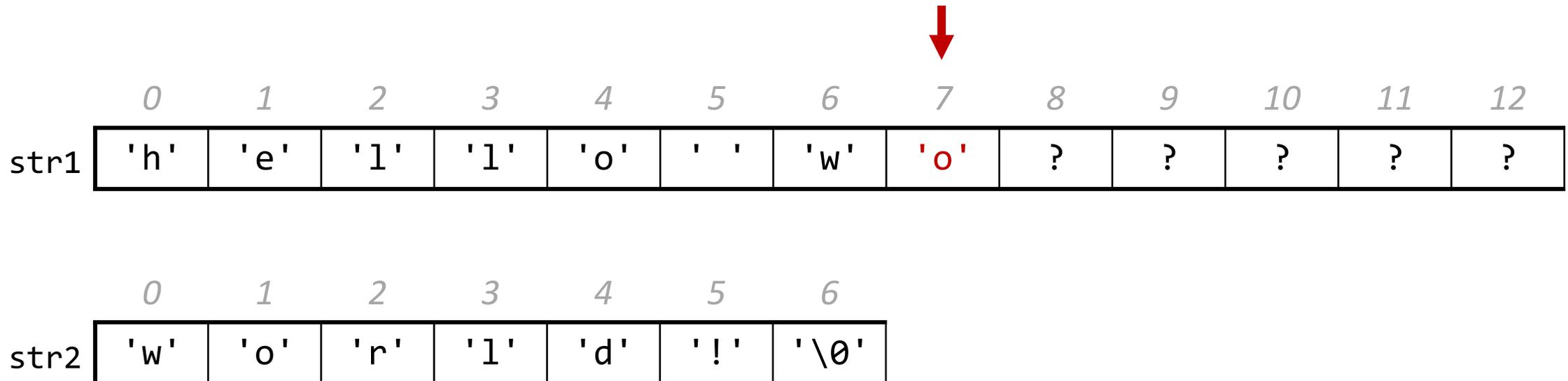


	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'?	'?	'?	'?	'?	'?

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

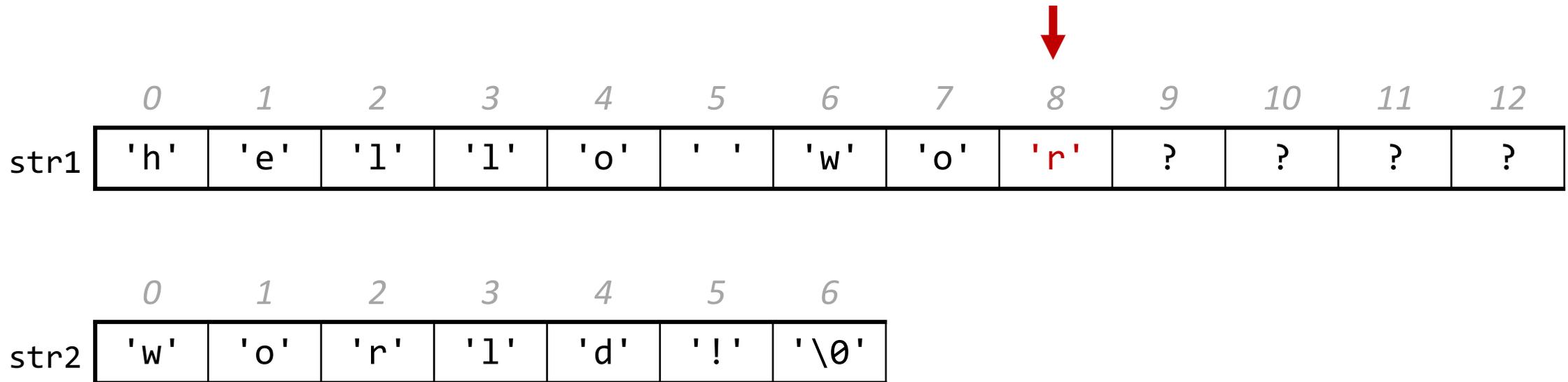
# Concatenating Strings

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



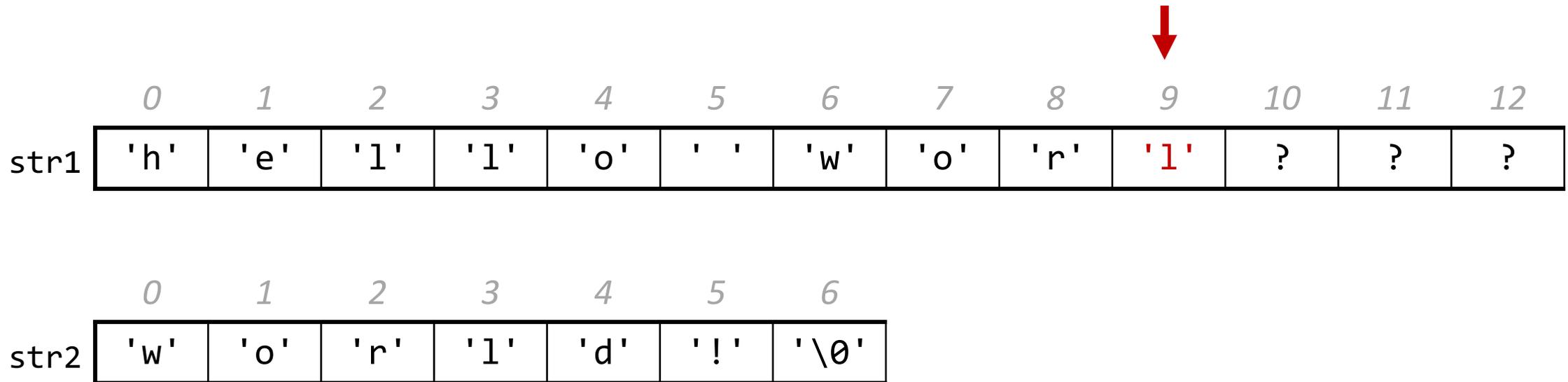
# Concatenating Strings

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



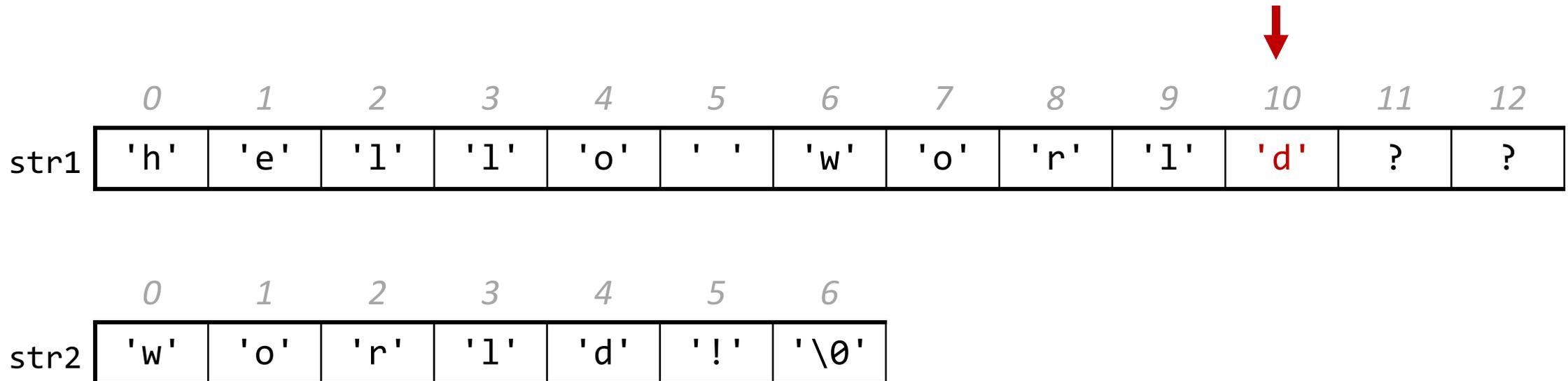
# Concatenating Strings

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



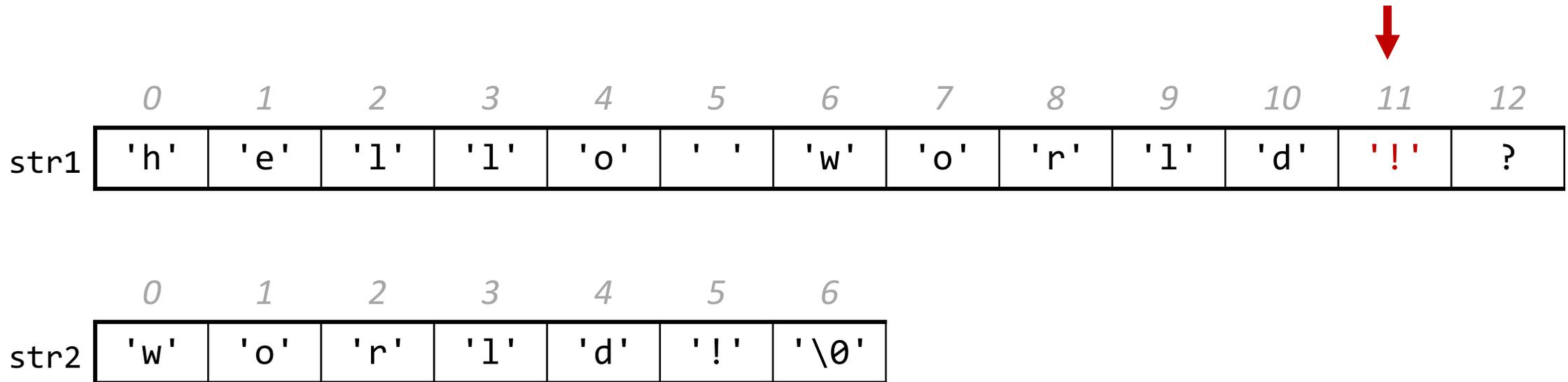
# Concatenating Strings

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



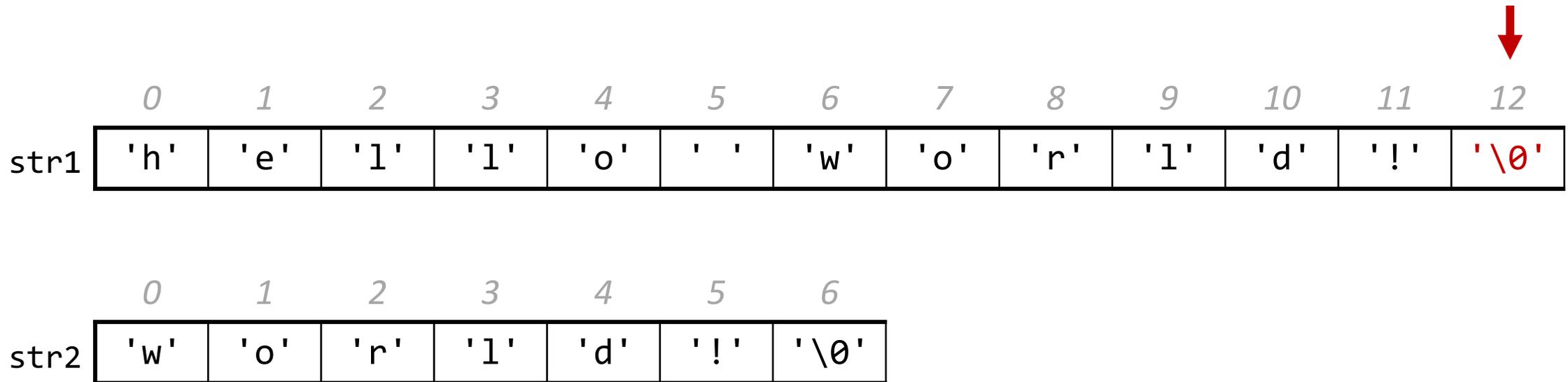
# Concatenating Strings

```
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);
```



# 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 and char \*

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

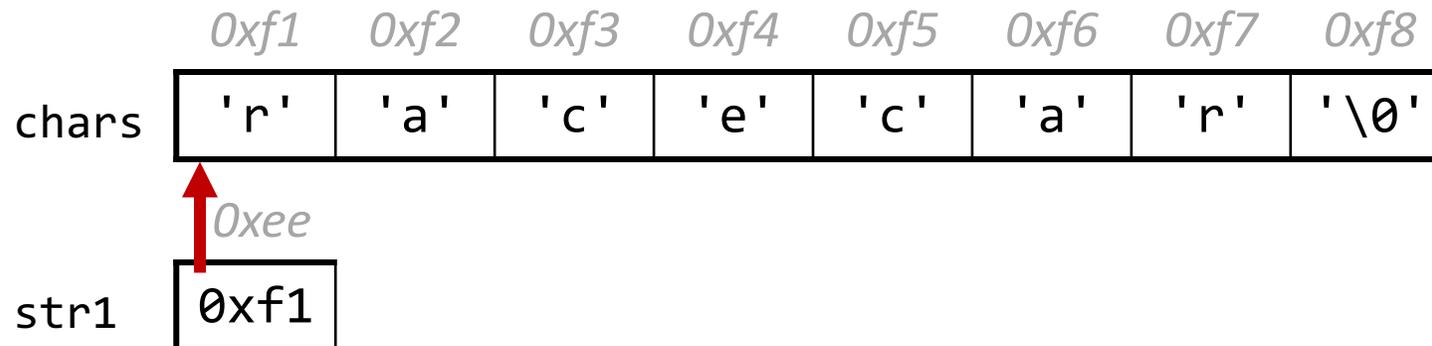
```
char myString[3];  
myString[0] = 'H';  
myString[1] = 'i';  
myString[2] = '\0';
```

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

# Substrings

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

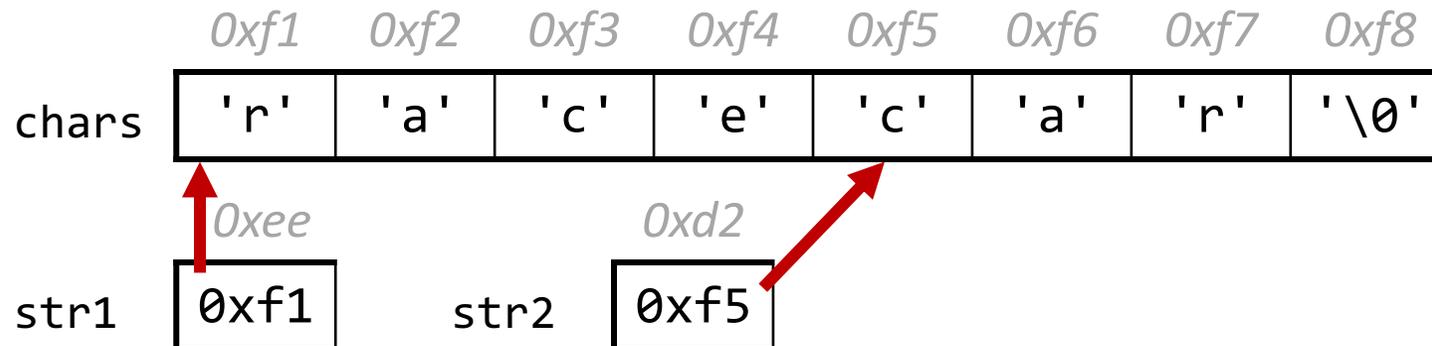
```
// Want just "car"  
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;
```



# Substrings

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

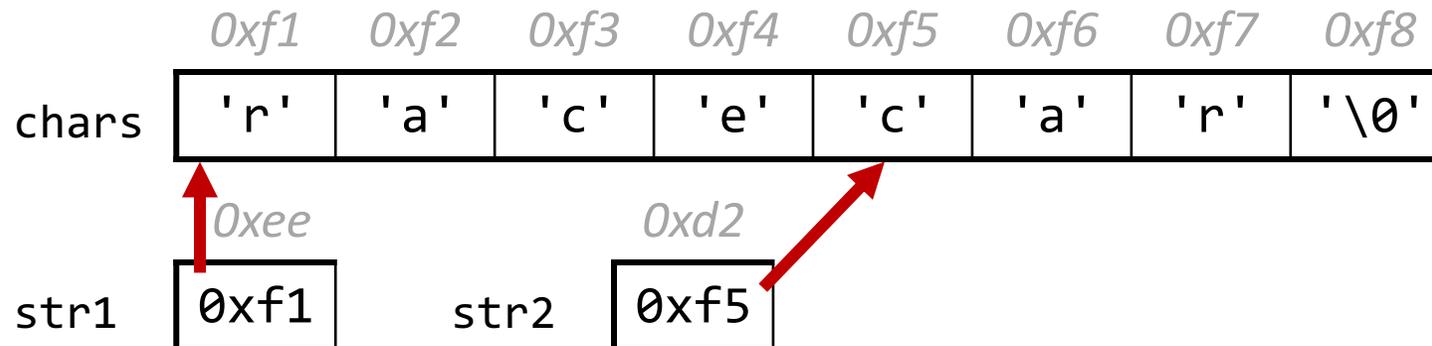
```
// Want just "car"  
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;
```



# Substrings

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

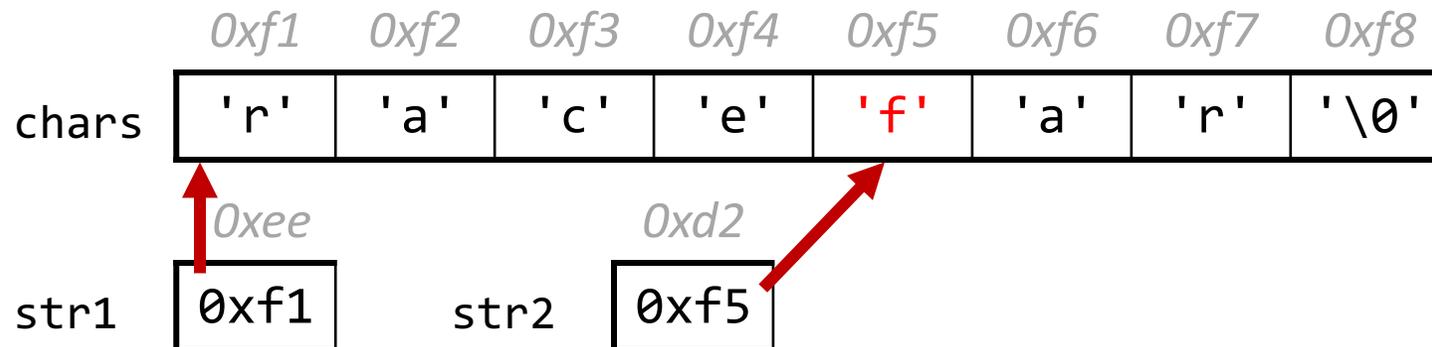
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
printf("%s\n", str1);           // racecar  
printf("%s\n", str2);           // car
```



# 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!

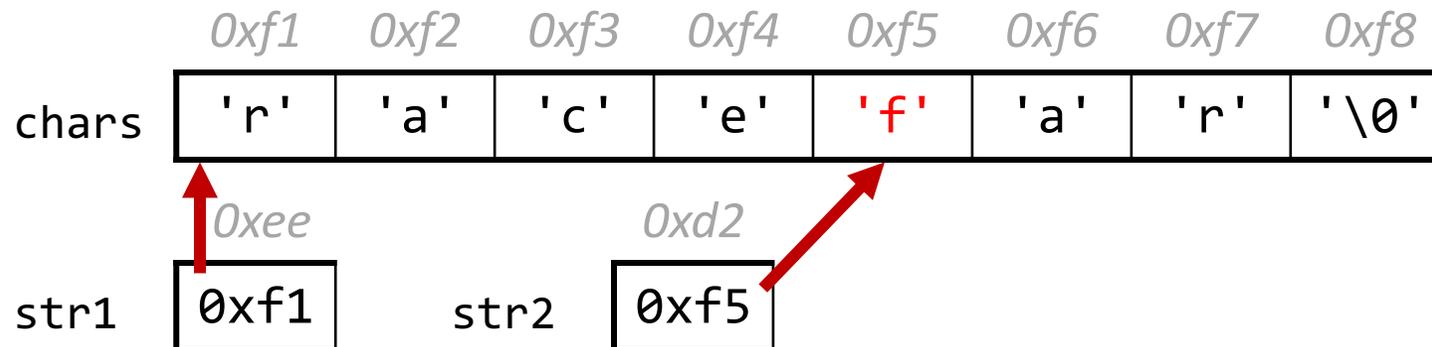
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
str2[0] = 'f';  
printf("%s %s\n", chars, str1);  
printf("%s\n", str2);
```



# 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
```



# 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
```

# 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
```

# Lecture Plan

• Characters	4
• Strings	11
• Common String Operations	16
• Comparing	19
• Copying	21
• Concatenating	79
• Substrings	90
• <b>Practice: Diamond</b>	<b>99</b>
• Live Session	105

# 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("DAISY")` should print:

```
D
DA
DAI
DAIS
DAISY
 AISY
  ISY
   SY
    Y
```

# 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("DAISY")` should print:

```
D
DA
DAI
DAIS
DAISY
 AISY
  ISY
   SY
    Y
```



# Daisy!



# Practice: Diamond



```
cp -r /afs/ir/class/cs107/lecture-code/lect4 .
```

# Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Practice:** Diamond

**Next time:** more strings

# Additional Live Session Slides

Lecture 04

Lecture 3 Ed thread: <https://edstem.org/us/courses/3085/discussion/205867>

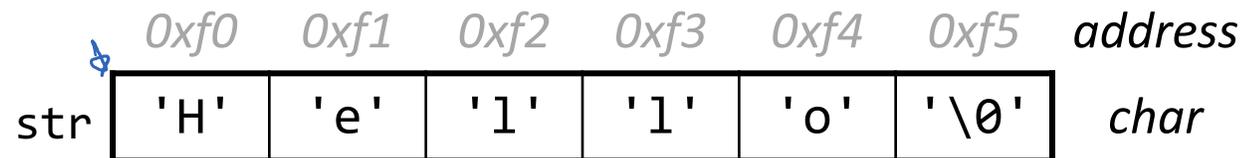
Lecture 4 Ed thread: <https://edstem.org/us/courses/3085/discussion/214248>

# Outline of the next few lectures

- Lecture 04 (today): An introduction to chars, C strings, and string.h
  - Lecture 05 (Monday): More string.h, char\* vs char [], pointers
  - Lecture 06: Arrays, Pointers, Arrays of Pointers, some keywords (const, etc.)
  - Lecture 07: Layout of Memory (Stack and Heap)
- 

# Key takeaways from this time

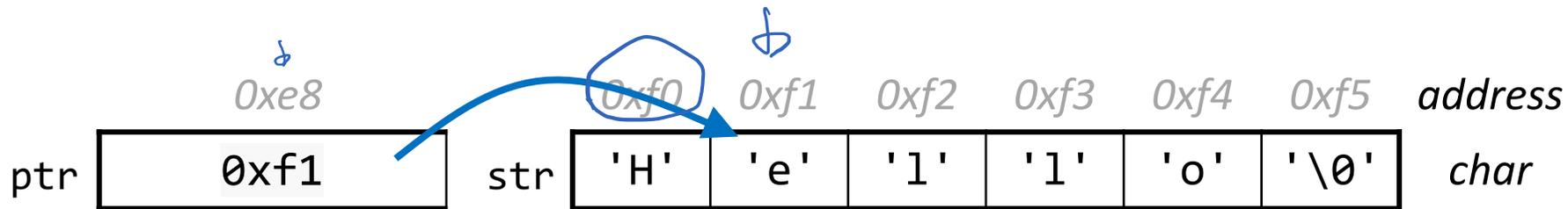
1. Valid strings are null-terminated.



```
char str[6];  
strcpy(str, "Hello");  
int length = strlen(str); // 5
```

# Key takeaways from this time

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.



64-bit  
0x80007634

```
char str[6];  
strcpy(str, "Hello");  
int length = strlen(str); // 5  
char *ptr = str + 1; // 0xf1  
printf("%s\n", ptr); // ello
```

# Key takeaways from this time

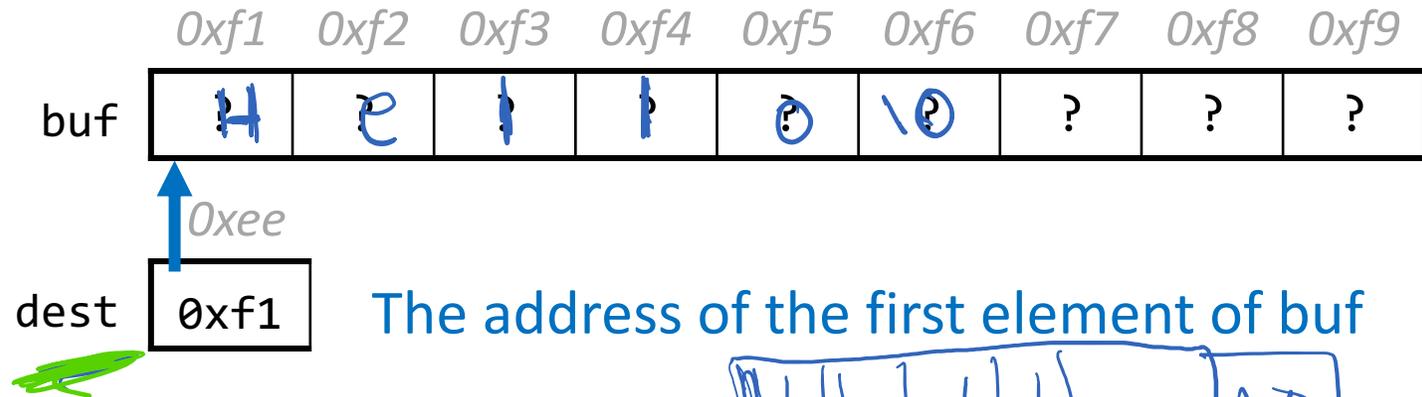
1. Valid strings are null-terminated. 
2. An array name (and a string name, by extension) is the address of the first element.
3. When you pass a `char[]` as a parameter, it is automatically passed as a `char *` (pointer to its first character)

## Why did C bother with this representation?

- C is a powerful, **efficient** language that requires a solid understanding of computer memory.
- We'll hone this understanding over these next two weeks!

# Takeaway #3 : man strcpy

```
1 char buf[6];
2 strcpy(buf, "Hello");
3 printf("%s\n", buf);
... ..
```



STRCPY(3)

Linux Programmer's Manual

## NAME

strcpy, strncpy - copy a string

## SYNOPSIS

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

```
char *strcpy(char *dest, const char *src);
```

- Lecture 5: where string constants like "hello" are stored.
- Lecture 6: what const means

# String copying exercise

```
1 char buf[      ];  
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 1: What value should go in the blank?

- A. 7
- B. 8
- C. 9
- D. 12
- E. strlen("Potatoes")
- F. Something else

Line 6: What is printed?

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



# String copying exercise

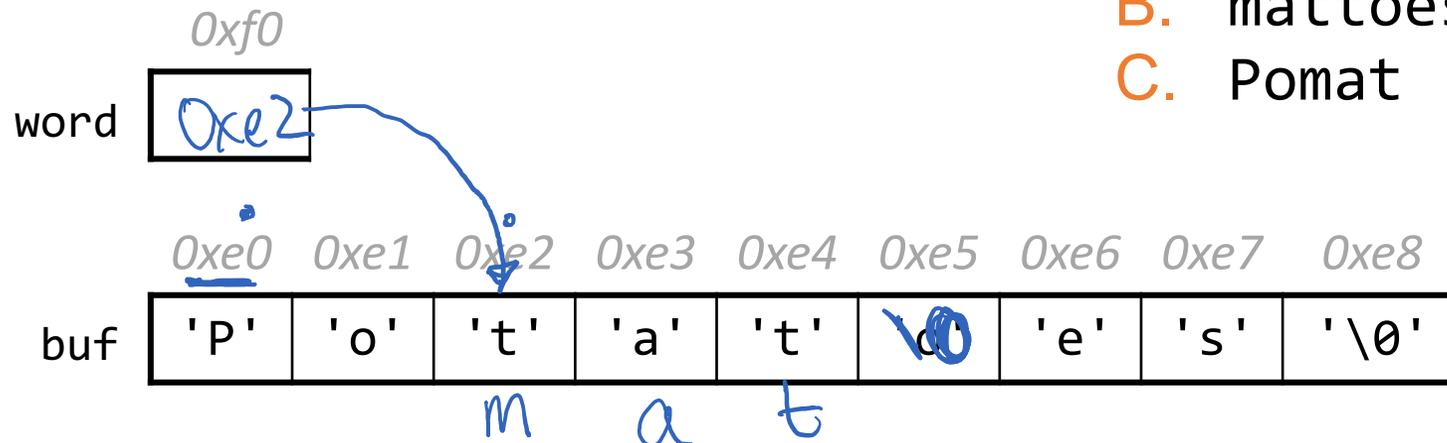
```
1 char buf[ 9 ];  
2 strcpy(buf, "Potatoes");  
3 printf("%s\n", buf);  
4 char *word = buf + 2; ←  
5 strncpy(word, "mat", 3);  
6 printf("%s\n", buf);
```

Line 1: What value should go in the blank?

- A. 7
- B. 8
- C. 9
- D. 12
- E. strlen("Potatoes")
- F. Something else

Line 6: What is printed?

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



# 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);
```

# Initializing strings

```
// create space for array first  
// then use string function to copy in content
```

```
char buf1[6];  
strcpy(buf1, "hello");
```

```
// initialize array to exactly the size that fits  
// string + null terminator
```

```
char buf2[] = "hello";
```

`{'h', 'e', 'l', 'l', 'o', '\0'};`

```
// will not work (why?)
```

```
char buf3[6];  
buf3 = "hello";
```

`char* ptr = "hello";`

`ptr = buf2;`

