

COMP201

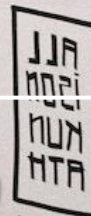
Computer Systems & Programming

Lecture #07 – Chars and Strings in C



KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020



**COMP201 Topic 3: How can a
computer represent and
manipulate more complex data
like text?**

Plan for Today

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

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

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

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

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>	0	1	2	3	4	5
	<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  
}
```

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

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

Lecture Plan

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

Common string.h Functions

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

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

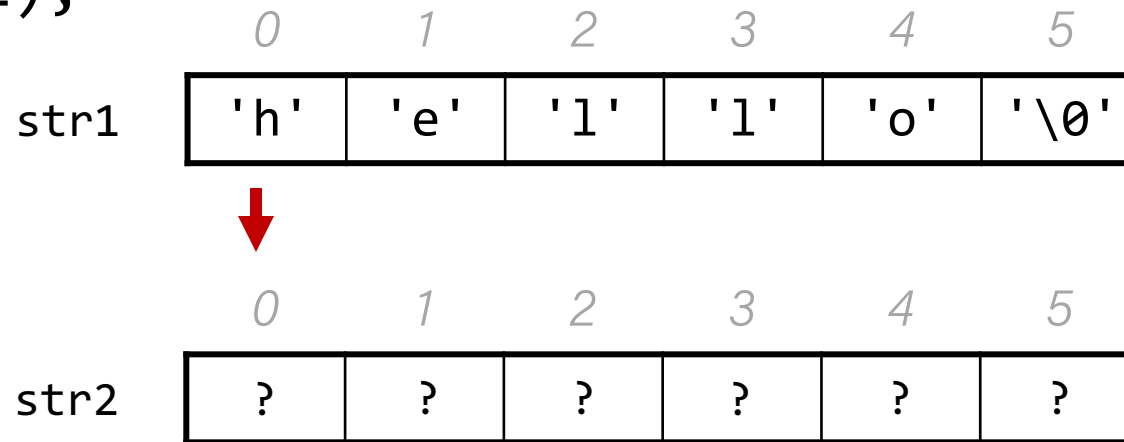
```
char str2[6];  
strcpy(str2, str1);  
str2[0] = 'c';
```

```
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

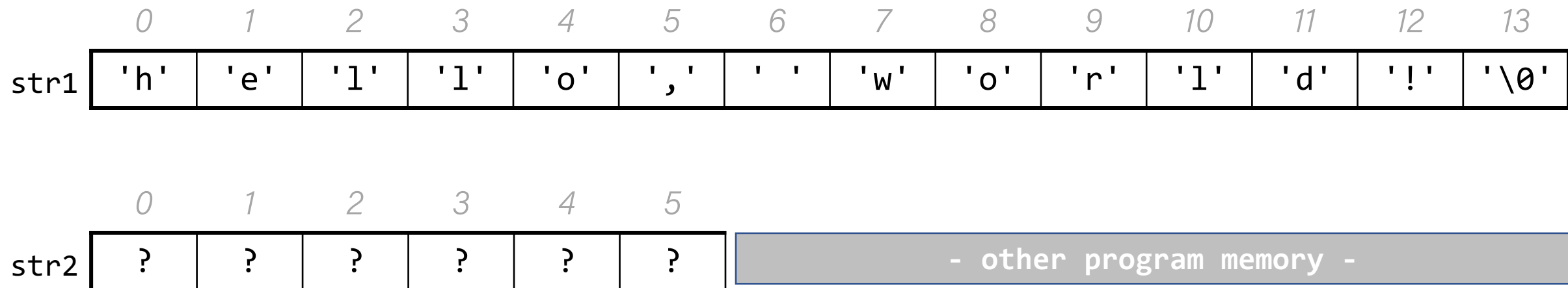
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
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!

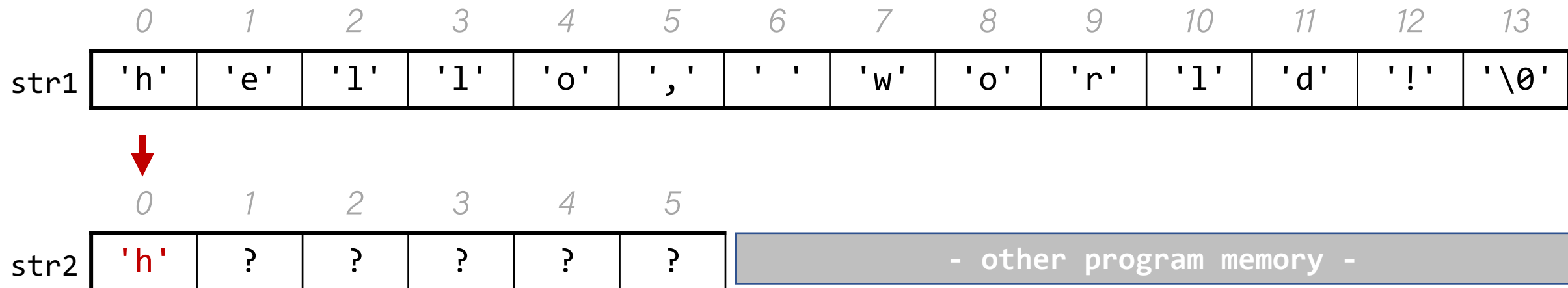
Copying Strings – Buffer Overflows

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



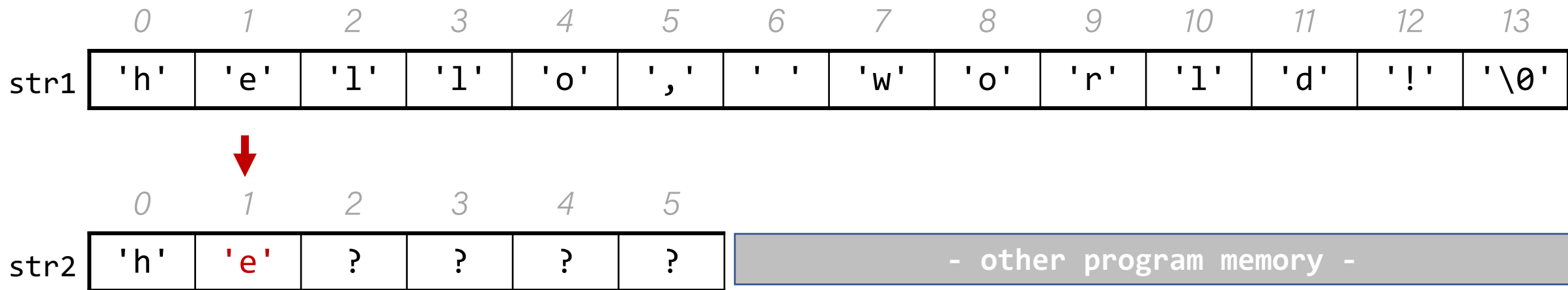
Copying Strings – Buffer Overflows

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



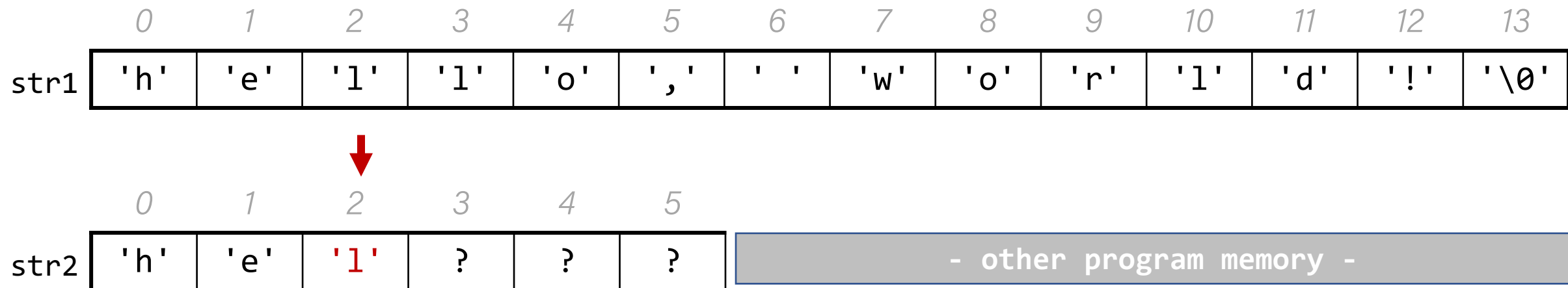
Copying Strings – Buffer Overflows

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



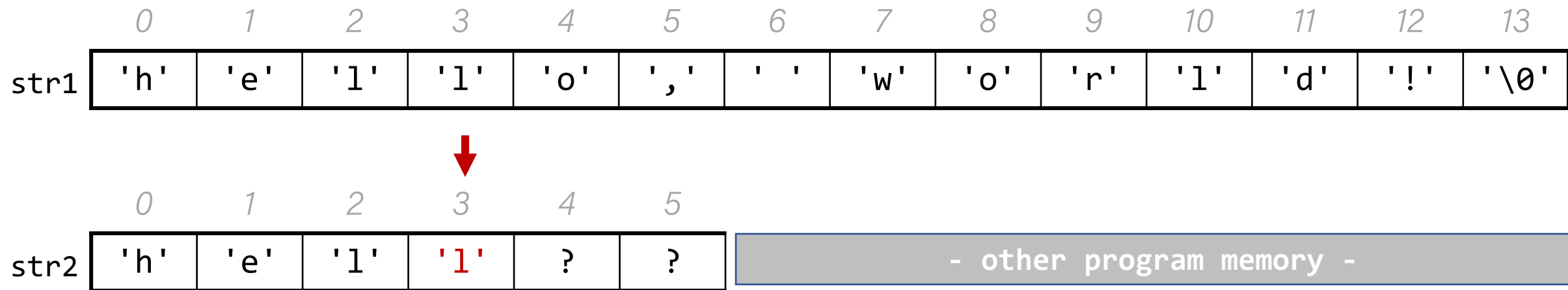
Copying Strings – Buffer Overflows

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



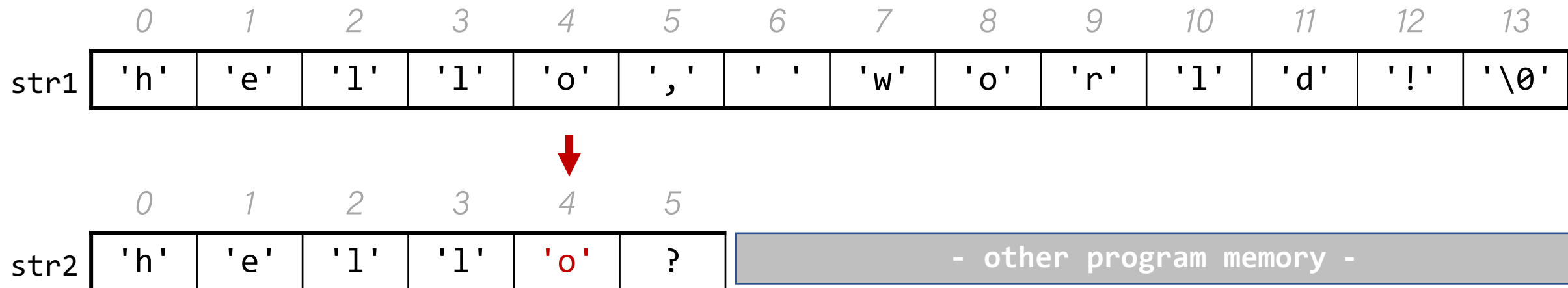
Copying Strings – Buffer Overflows

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



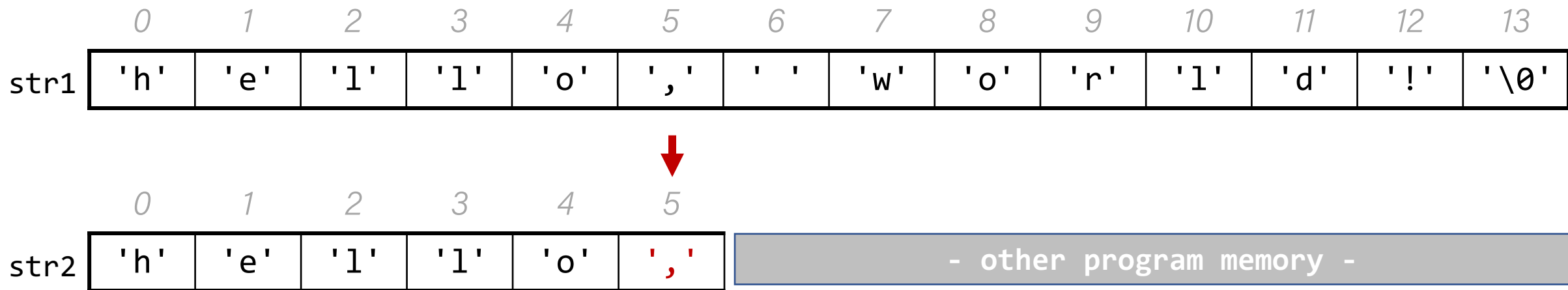
Copying Strings – Buffer Overflows

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



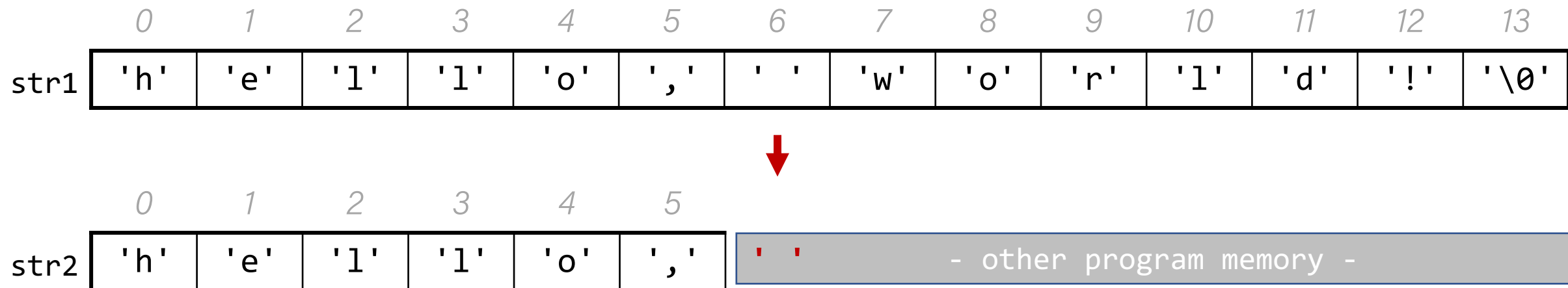
Copying Strings – Buffer Overflows

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



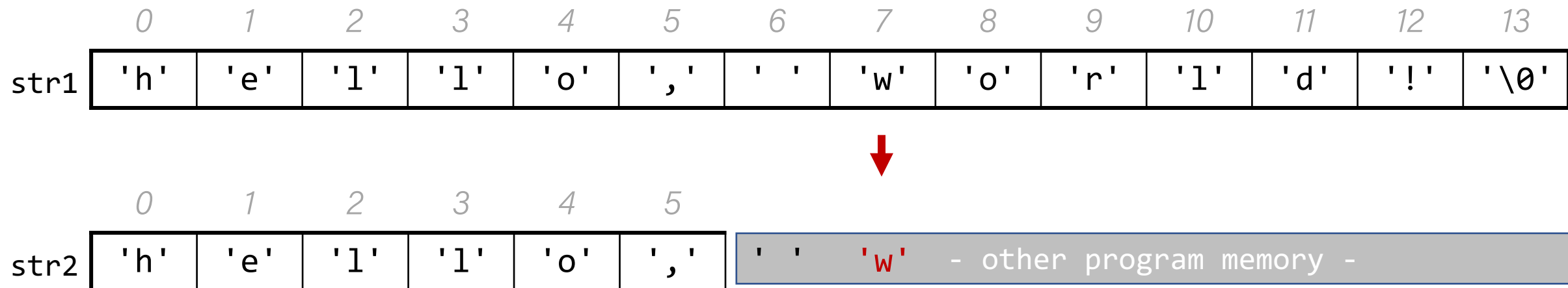
Copying Strings – Buffer Overflows

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



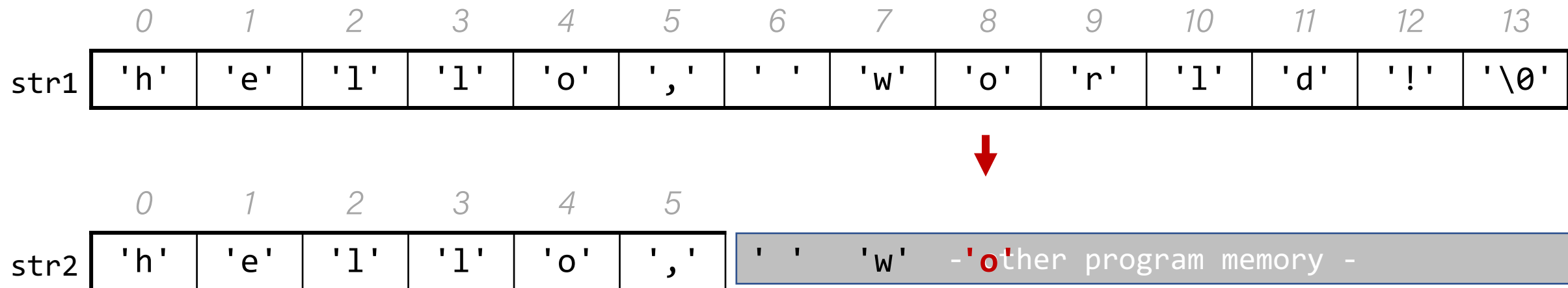
Copying Strings – Buffer Overflows

```
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!
```



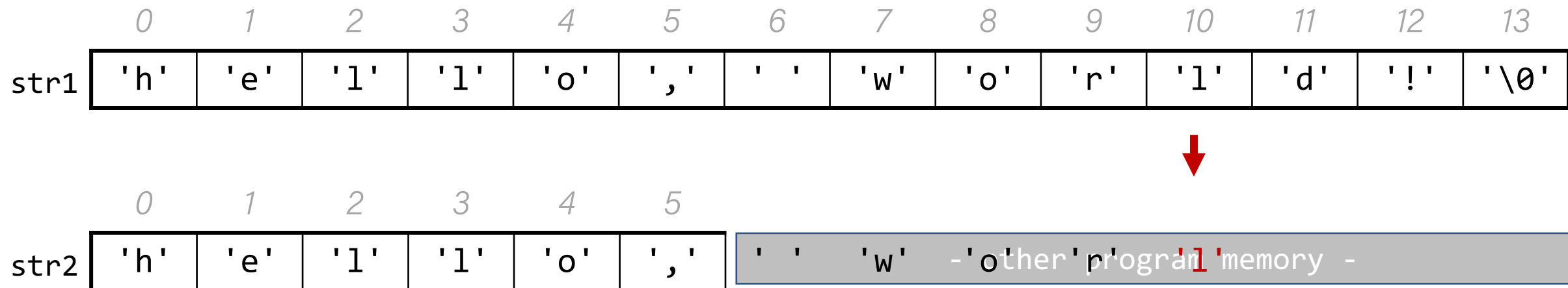
Copying Strings – Buffer Overflows

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



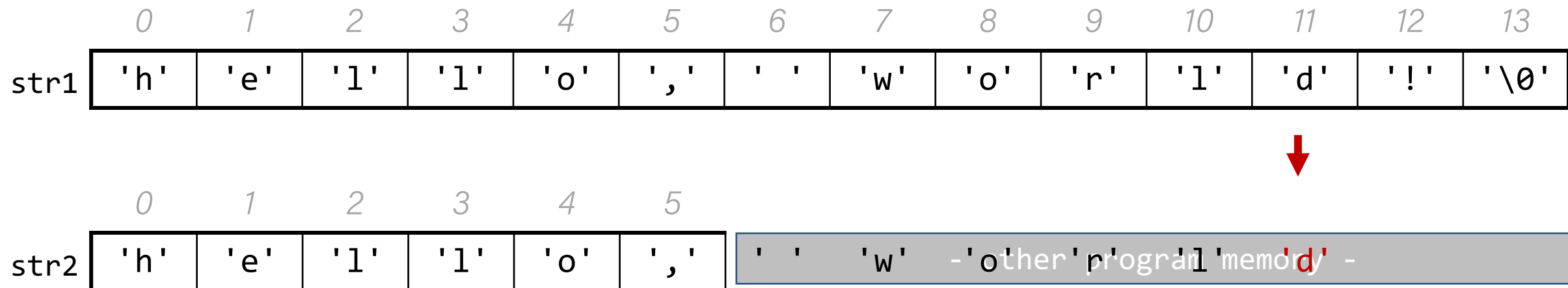
Copying Strings – Buffer Overflows

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



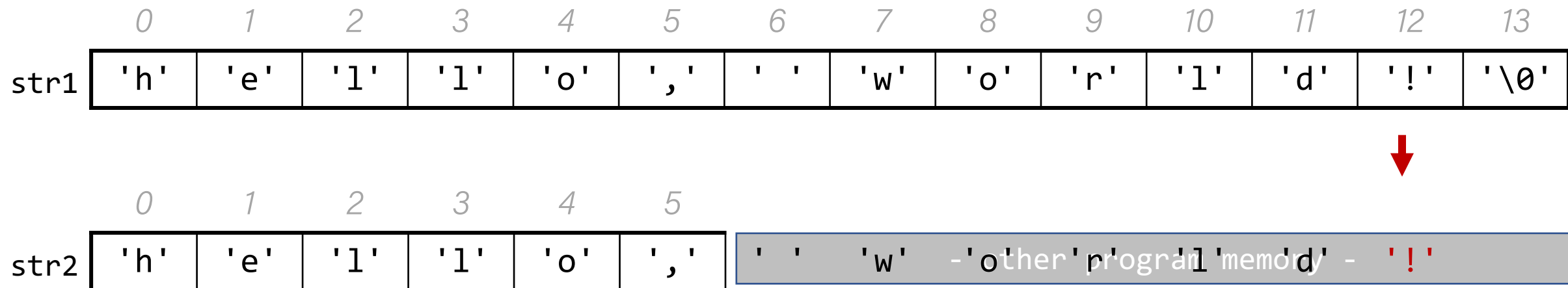
Copying Strings – Buffer Overflows

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



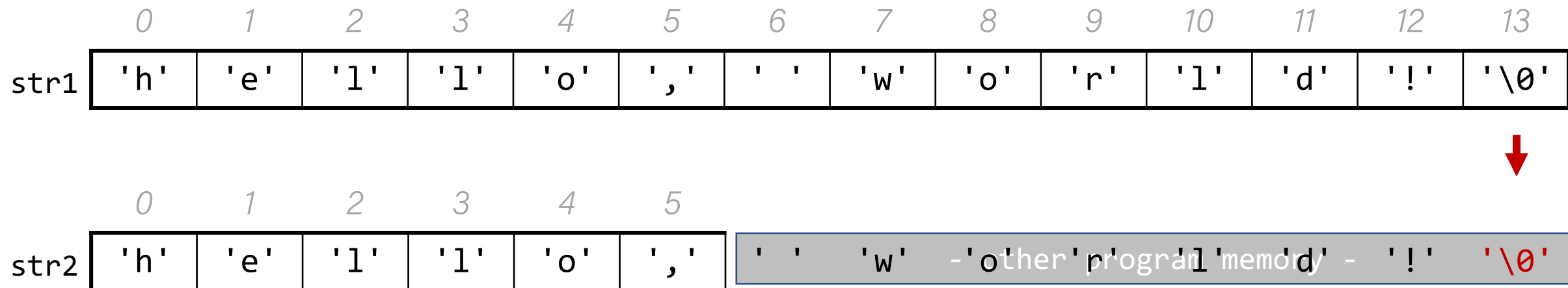
Copying Strings – Buffer Overflows

```
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

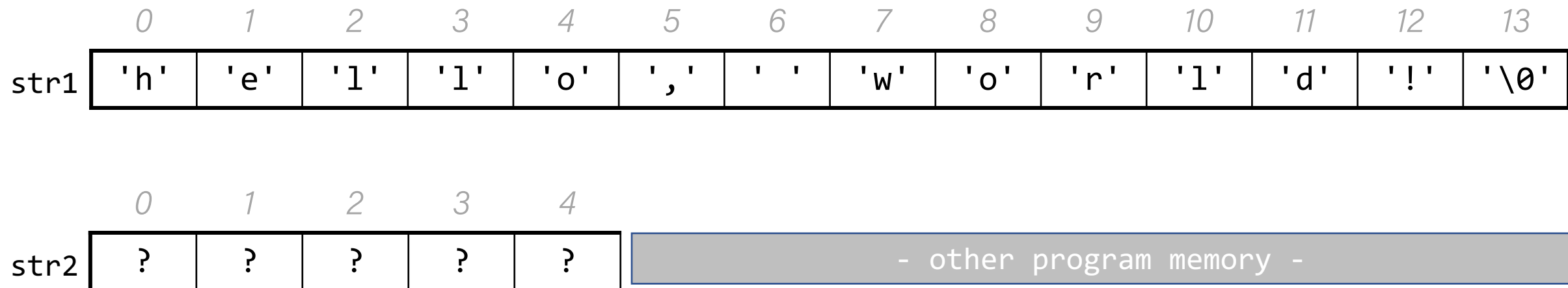
strncpy(dst, src, n): copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **dst** will *not be null terminated*!

```
// 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'`!

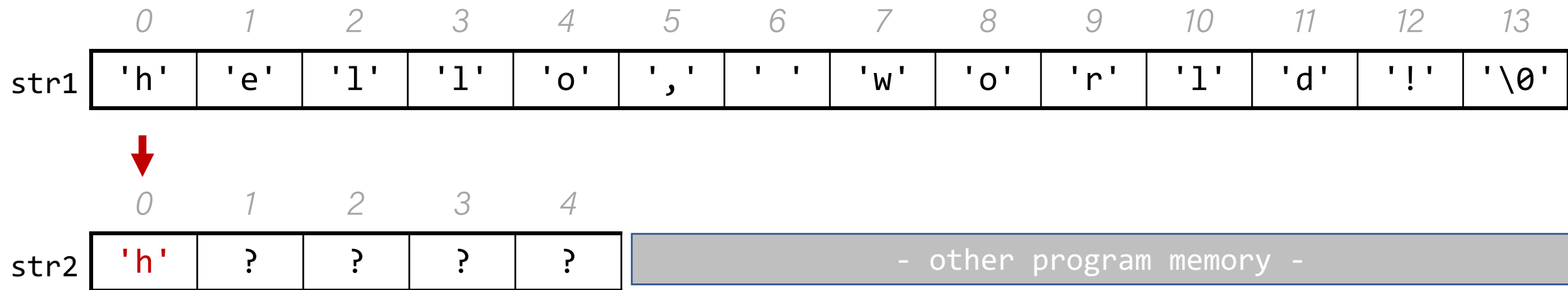
Copying Strings – strncpy

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



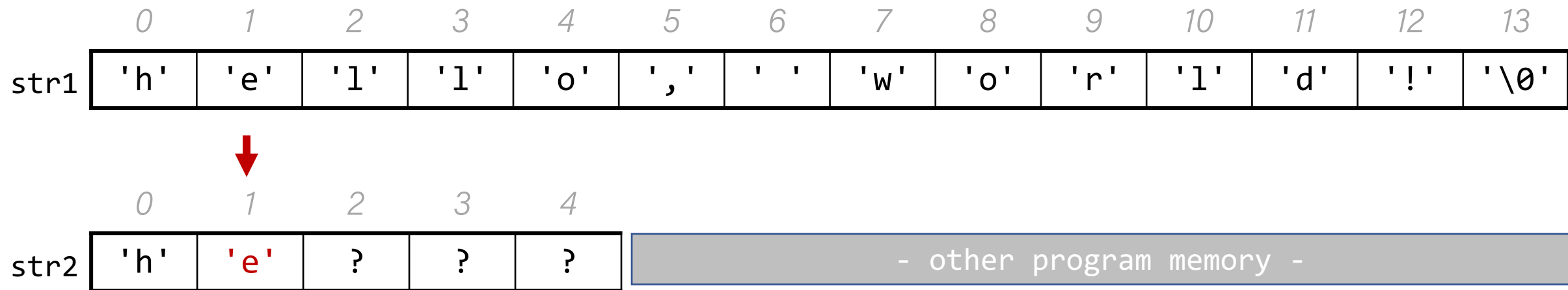
Copying Strings – strncpy

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



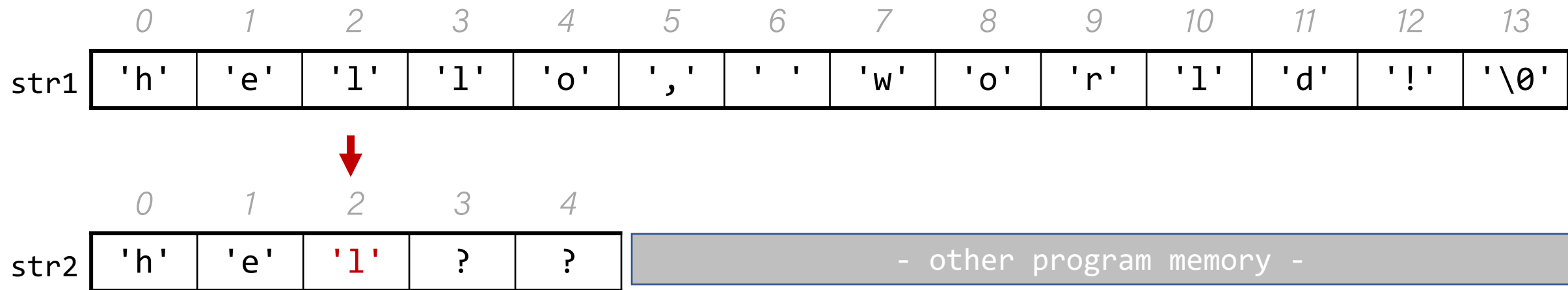
Copying Strings – strncpy

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



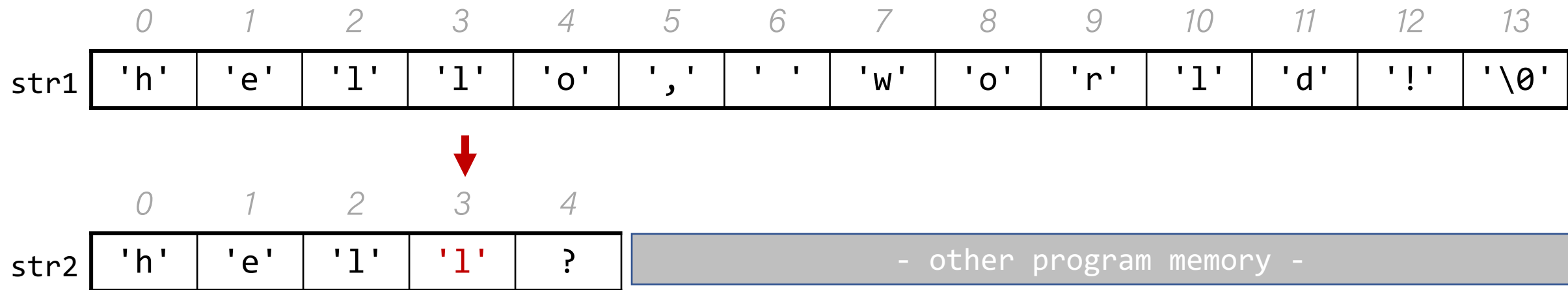
Copying Strings – strncpy

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



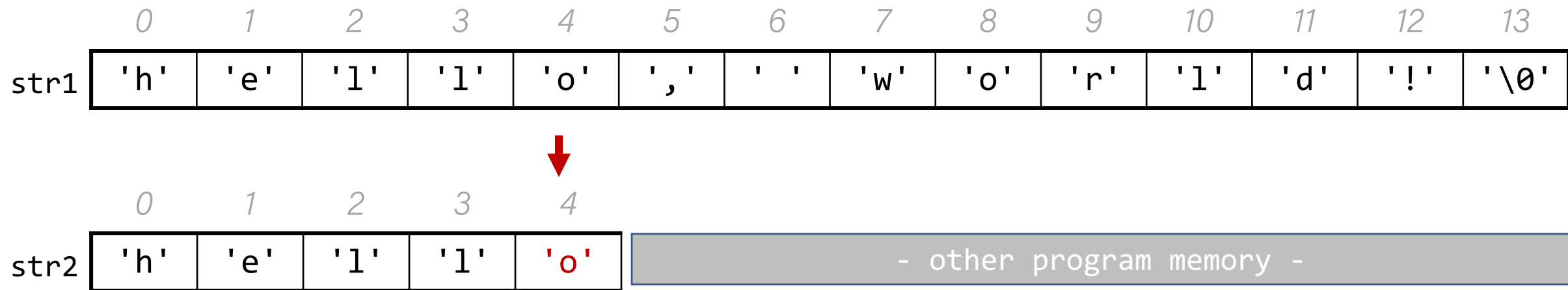
Copying Strings – strncpy

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



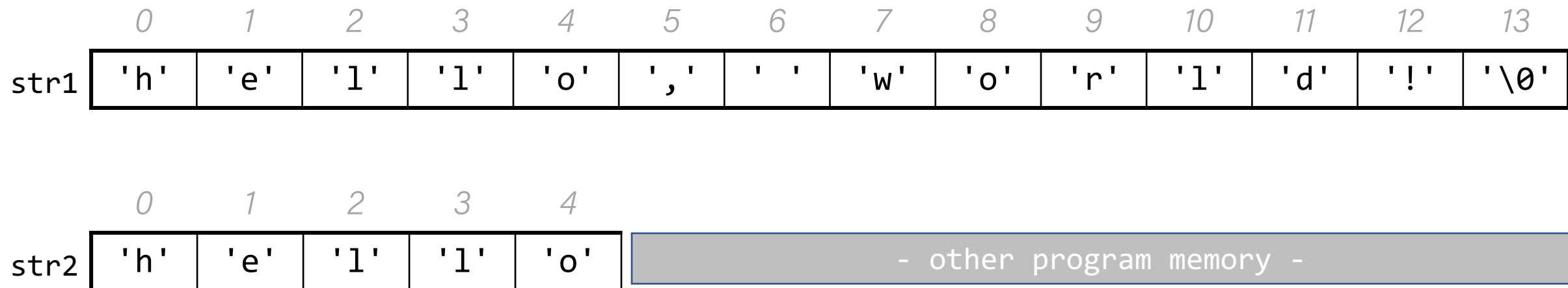
Copying Strings – strncpy

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



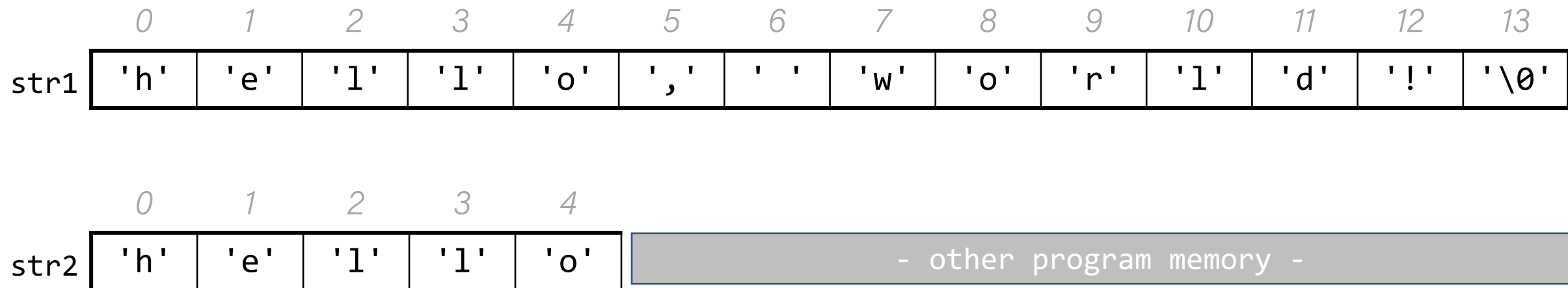
Copying Strings – strncpy

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



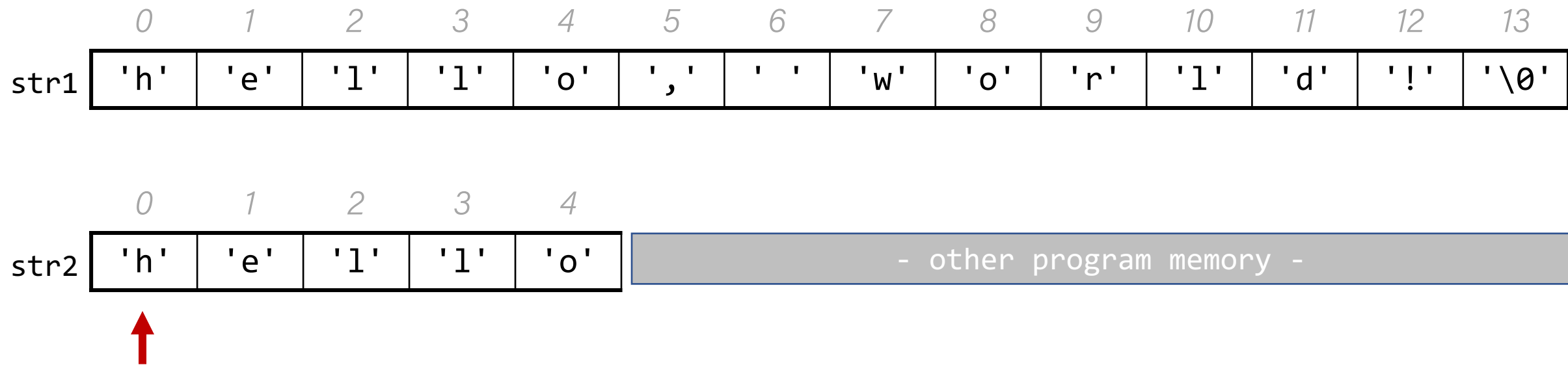
Copying Strings – strncpy

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



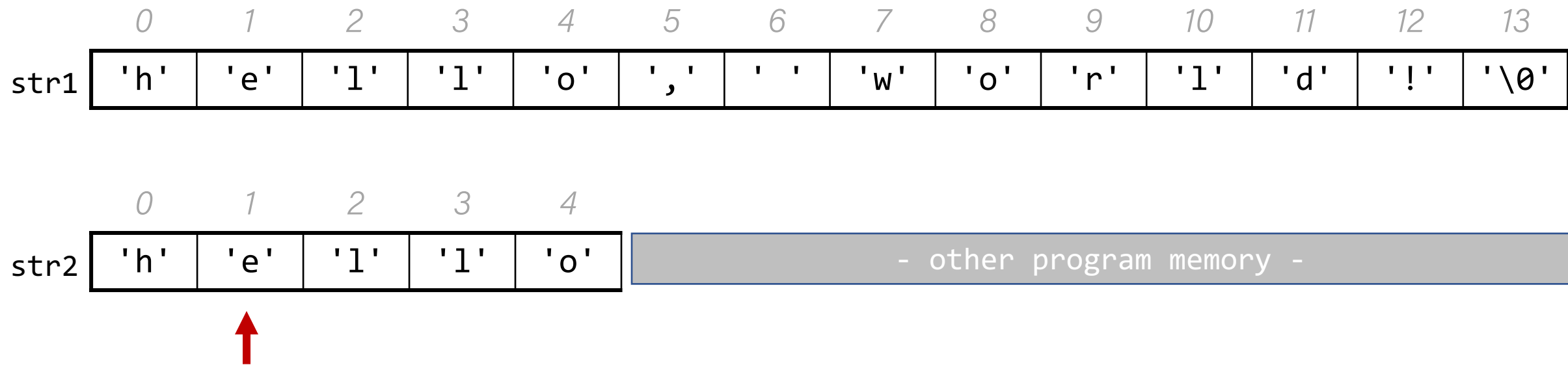
Copying Strings – strncpy

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



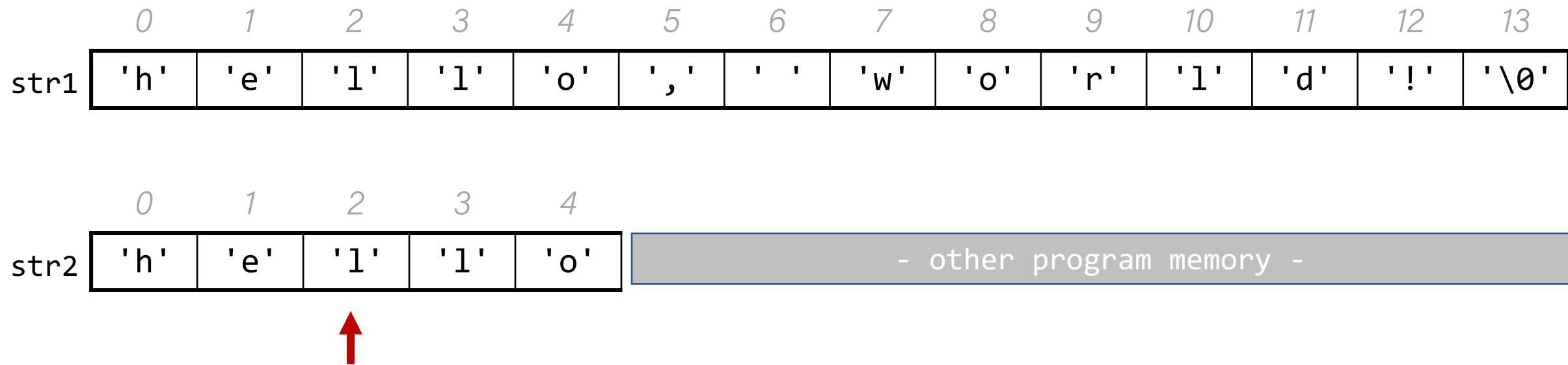
Copying Strings – strncpy

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



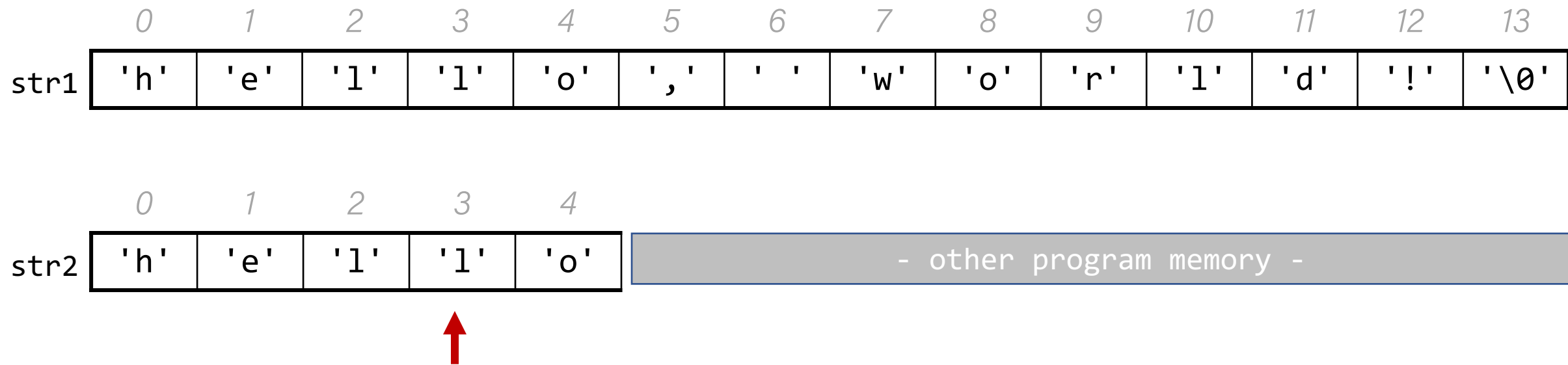
Copying Strings – strncpy

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



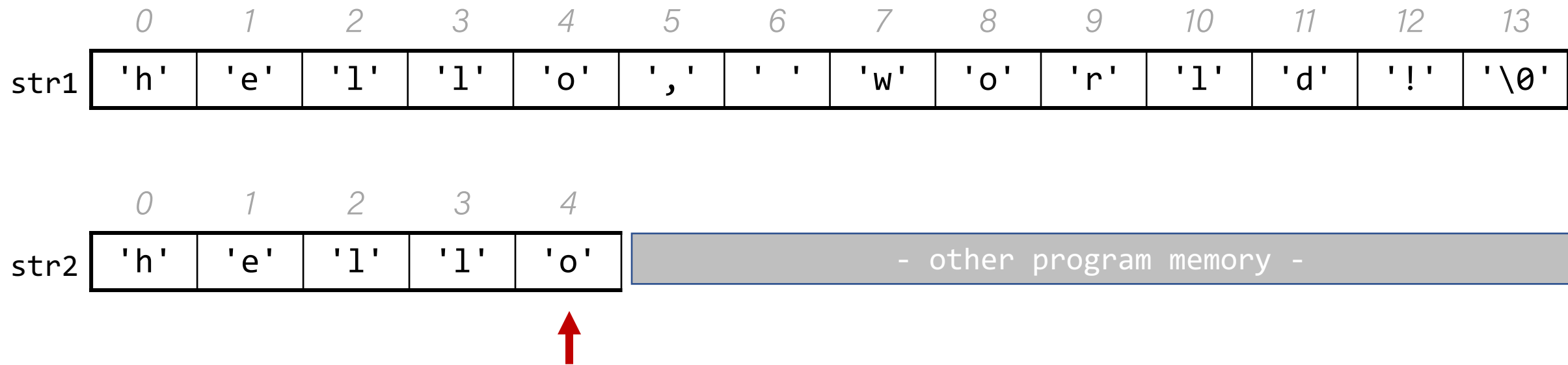
Copying Strings – strncpy

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



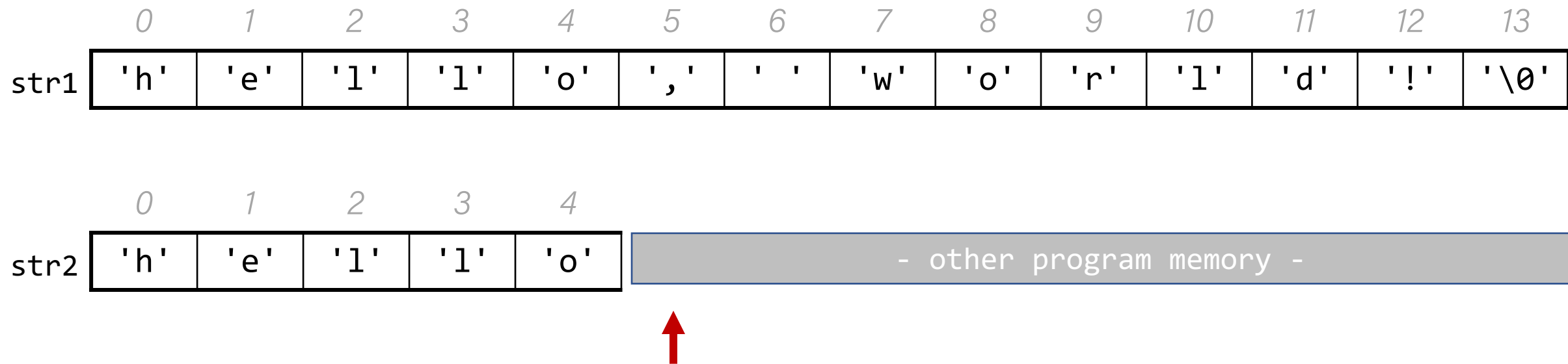
Copying Strings – strncpy

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



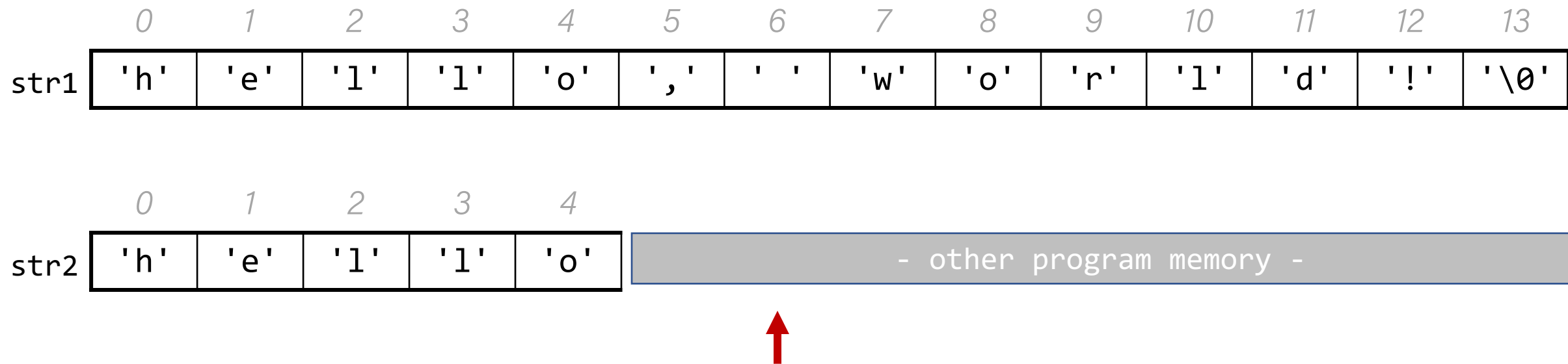
Copying Strings – strncpy

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



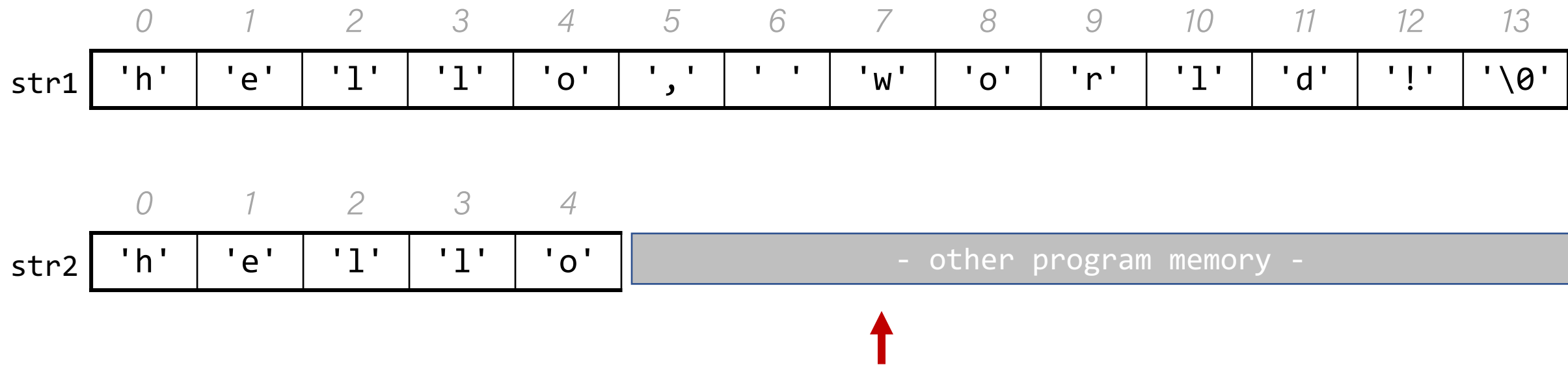
Copying Strings – strncpy

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



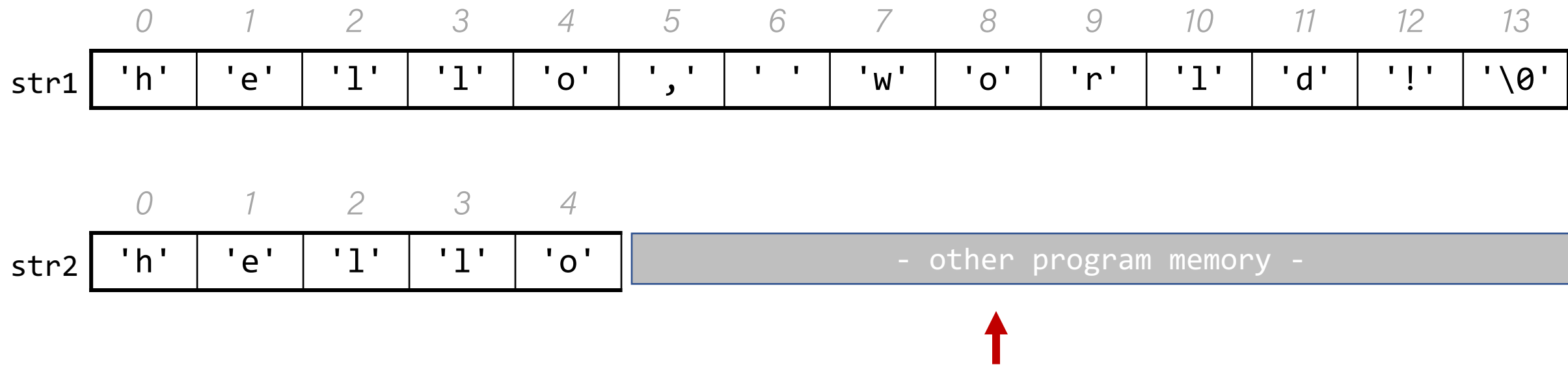
Copying Strings – strncpy

```
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);
```



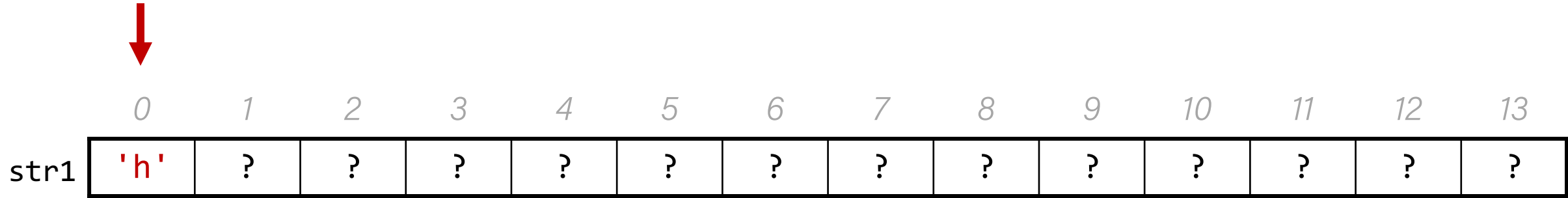
Copying Strings – strncpy

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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	?	?	?	?	?	?	?	?	?	?	?	?	?	?

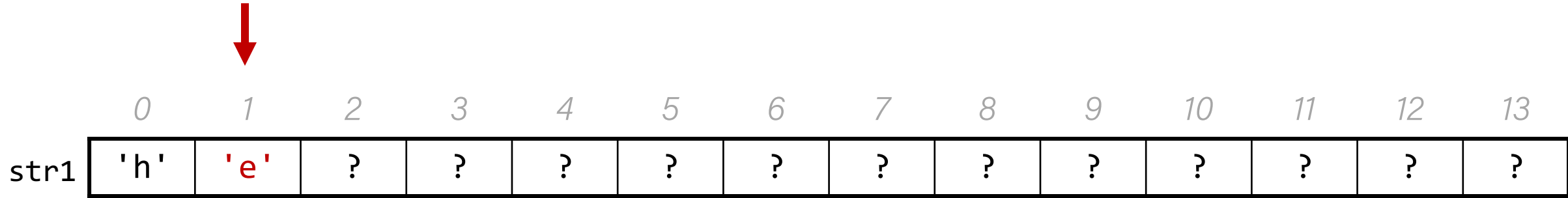
Copying Strings – strncpy

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



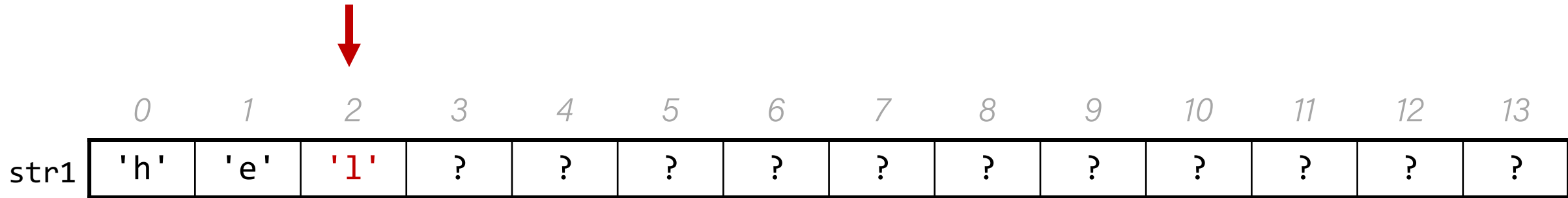
Copying Strings – strncpy

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



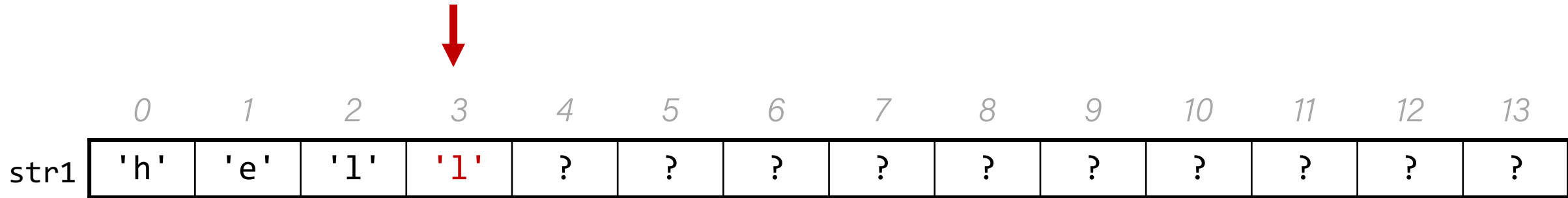
Copying Strings – strncpy

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



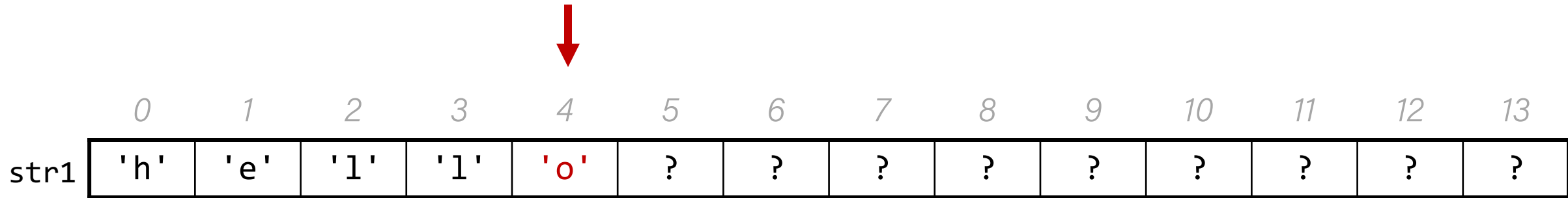
Copying Strings – strncpy

```
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);
```

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

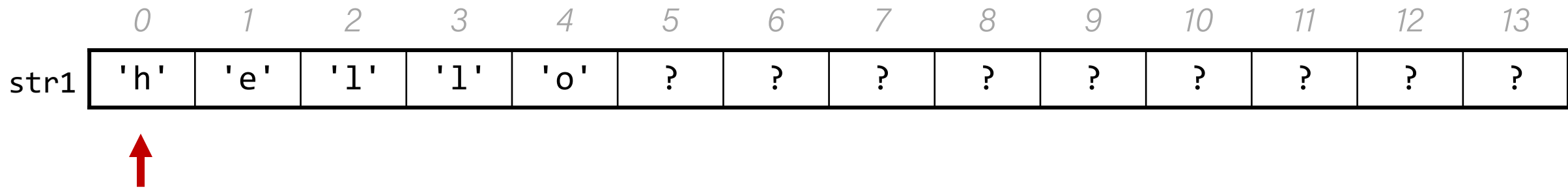
Copying Strings – strncpy

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

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

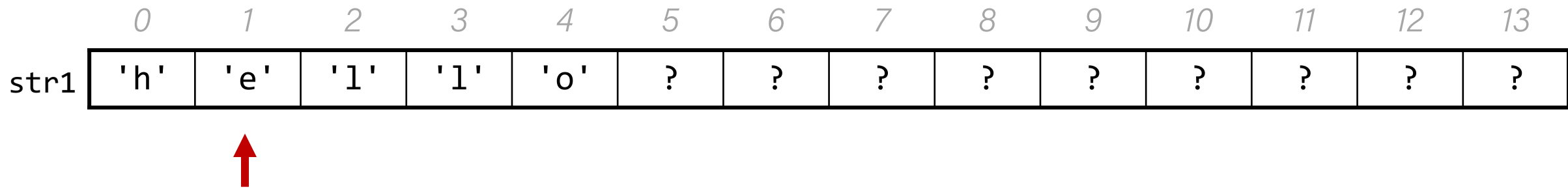
Copying Strings – strncpy

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



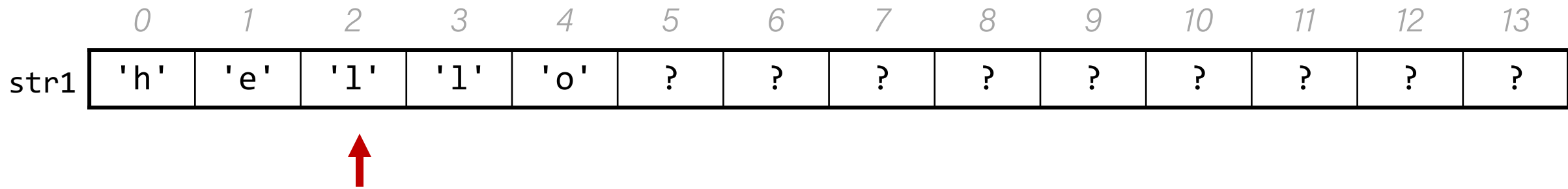
Copying Strings – strncpy

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



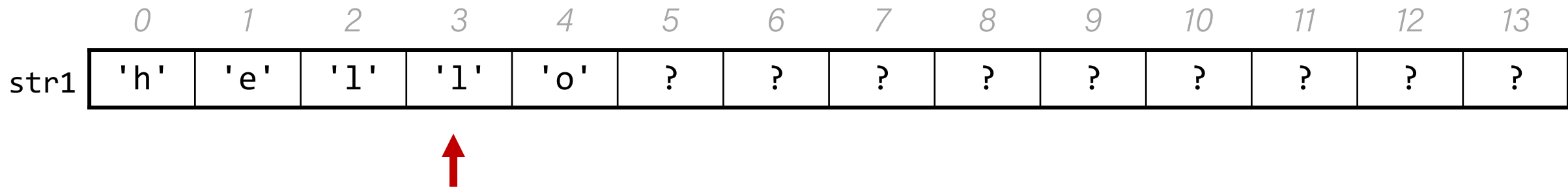
Copying Strings – strncpy

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



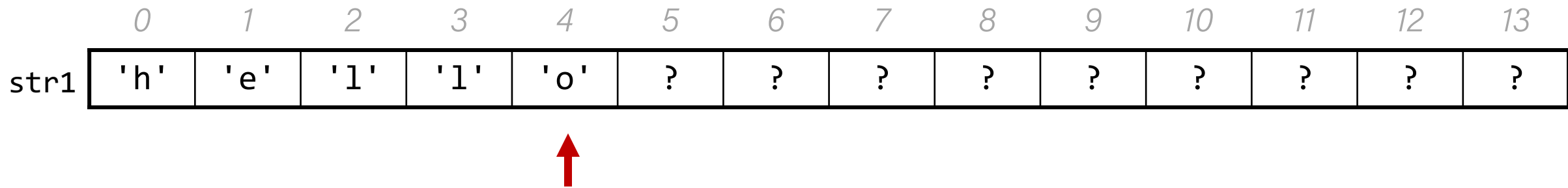
Copying Strings – strncpy

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



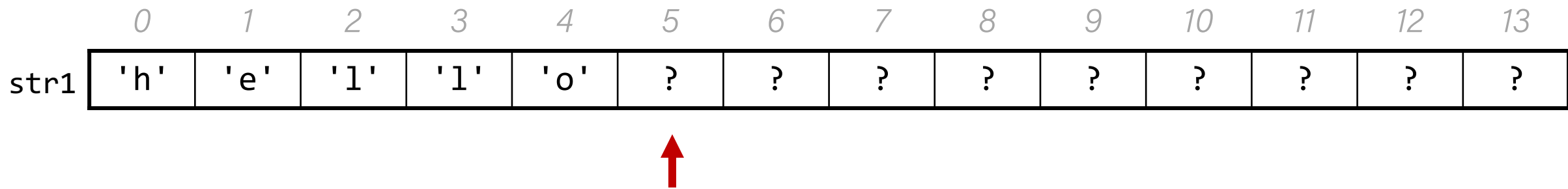
Copying Strings – strncpy

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



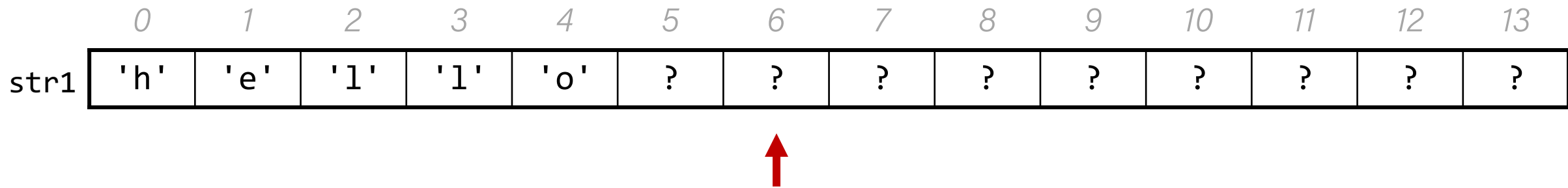
Copying Strings – strncpy

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



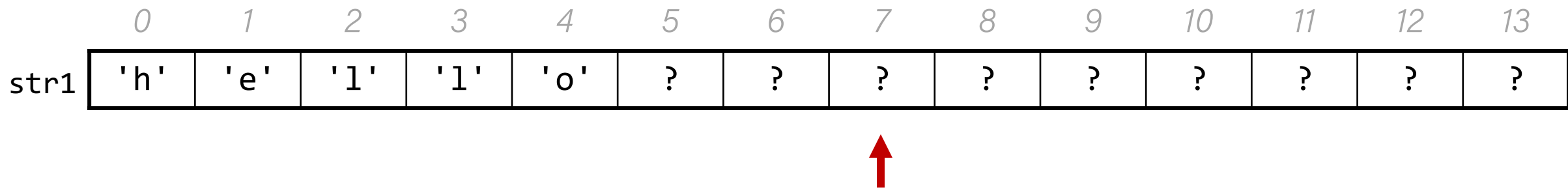
Copying Strings – strncpy

```
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 = %lu\\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



Recap

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

Next time: more strings