

COMP201

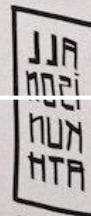
Computer Systems & Programming

Lecture #05 – Chars and Strings in C



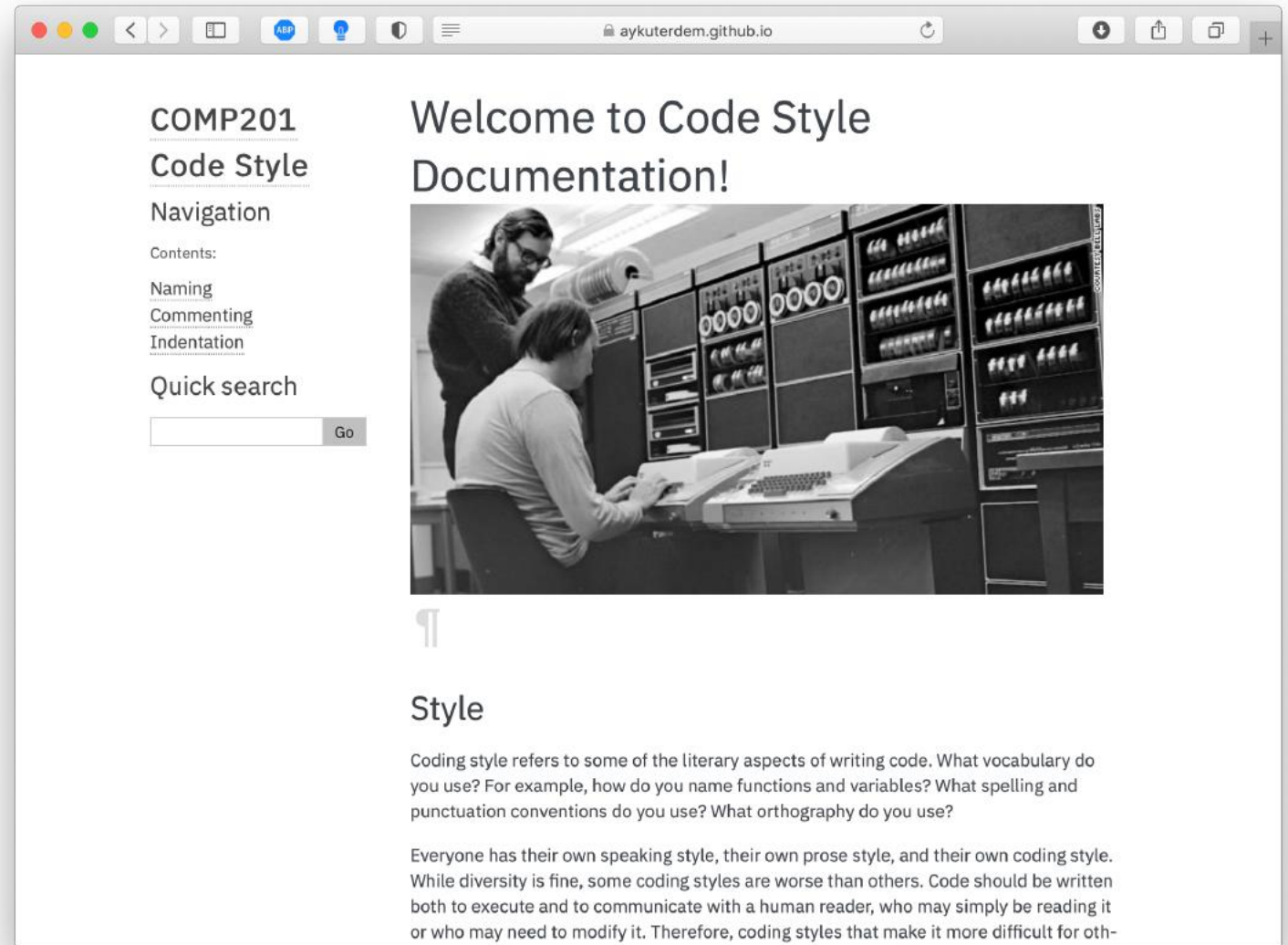
KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2021



COMP201 Coding Style Guide for C Programming

- Our guide serves as a brief introduction to C coding style.
- Following a formal style is very important to write a clean and easy to read code.
- There are many standards out there!



<https://aykuterdem.github.io/classes/comp201/code-style/html/index.html>

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

Plan for Today

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli and Lisa Yan's Stanford CS107 class

—Swami Iyer's Umass Boston CS110 class

Lecture Plan

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

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 8-bit *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!)

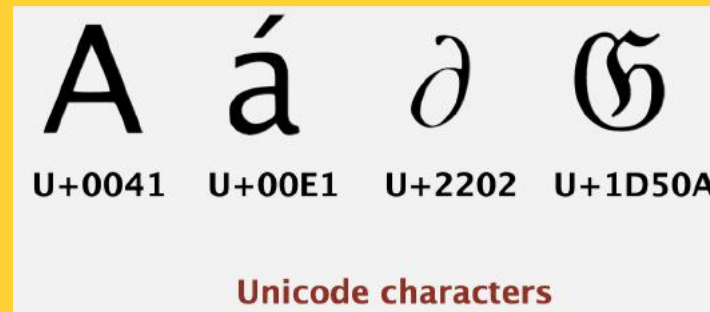
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

```
char uppercaseA = 'A';           // Actually 65
char lowercaseA = 'a';           // Actually 97
char zeroDigit = '0';            // Actually 48
```

















































































Unicode Transformation Formats

- The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more
- Unicode being somewhat wasteful of space for English documents, ISO also defined several "Unicode Transformation Formats" (UTF), the most popular being UTF-8



Emojis

- Emojis are just like characters, and they have a standard, too

Smileys & People															
face-positive															
No	Code	Browser	App!	Goog ^d	Twtr.	One	FB	FBM	Sams.	Wind.	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600														grinning face
2	U+1F601														beaming face with smiling eyes
3	U+1F602														face with tears of joy
4	U+1F923														rolling on the floor laughing
5	U+1F603														grinning face with big eyes
6	U+1F604														grinning face with smiling eyes
7	U+1F605														grinning face with sweat
8	U+1F606														grinning squinting face
9	U+1F609														winking face
⋮															

- Full Emoji List, v5.0

<https://unicode.org/emoji/charts/full-emoji-list.html>

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
- Practice: Diamonds

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
- Practice: Diamonds

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> . strcpy copies until 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> .

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");
```

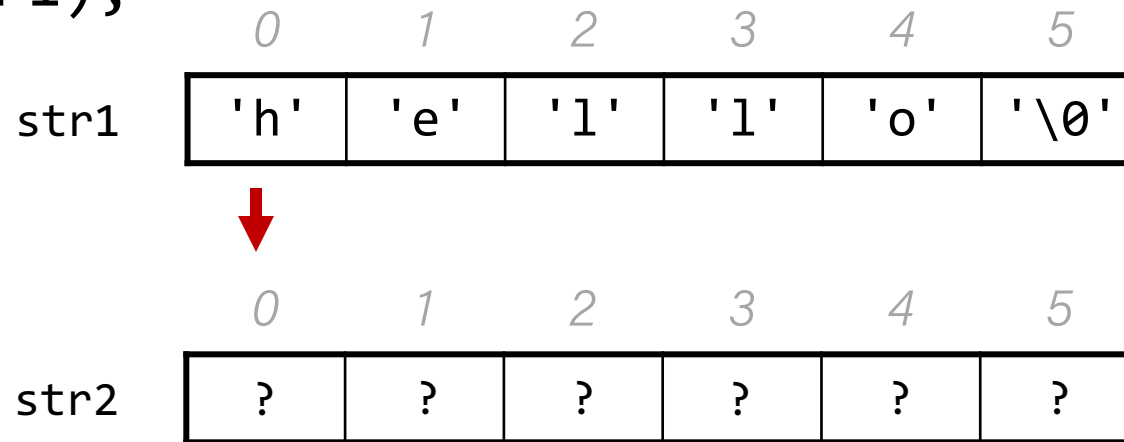
```
char str2[6];  
strcpy(str2, str1);  
str2[0] = 'c';
```

```
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

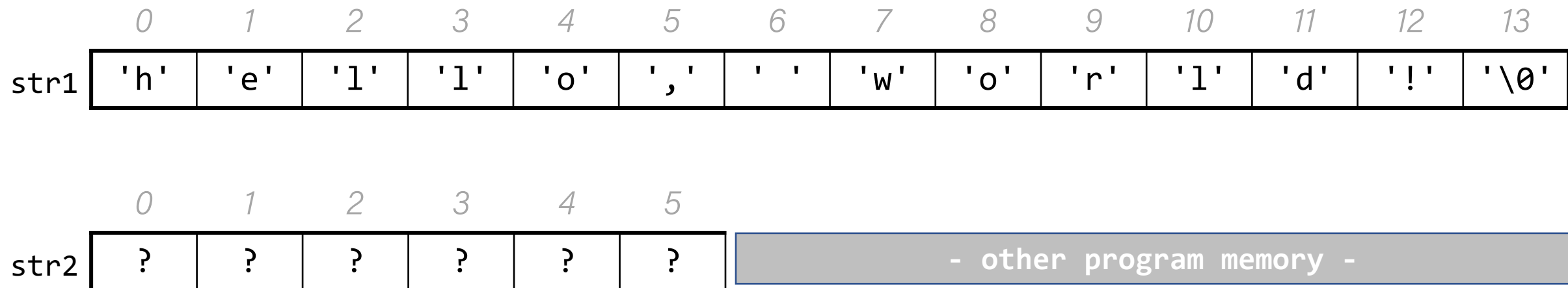
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
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!

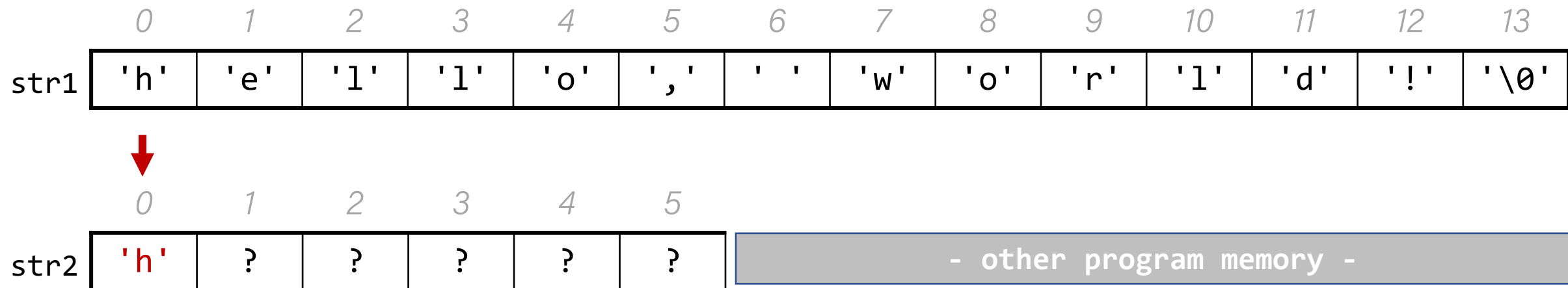
Copying Strings – Buffer Overflows

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



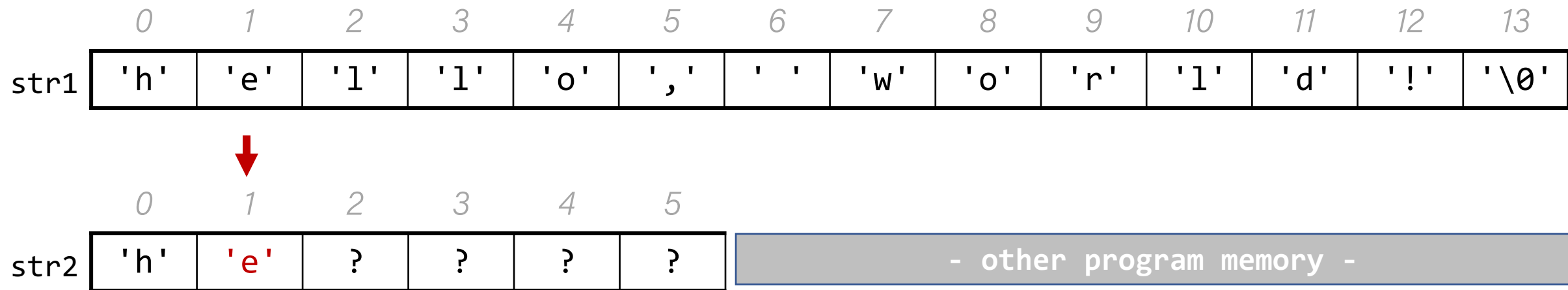
Copying Strings – Buffer Overflows

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



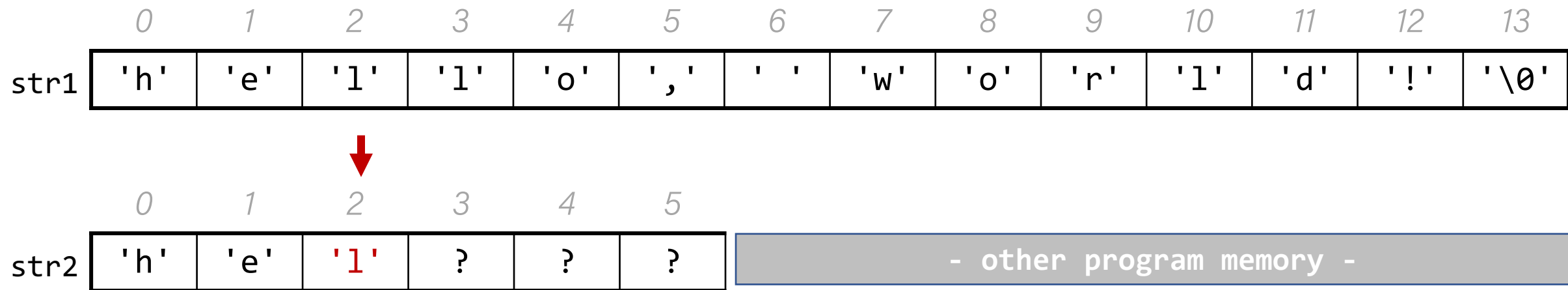
Copying Strings – Buffer Overflows

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



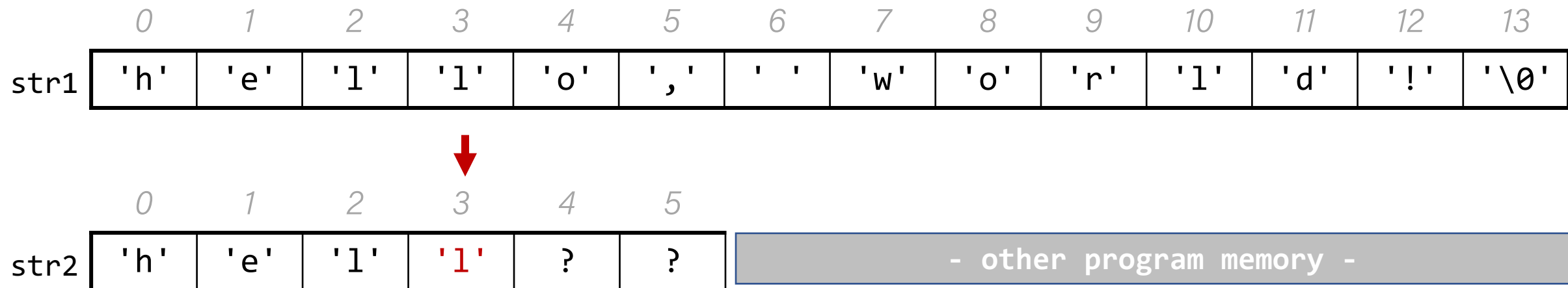
Copying Strings – Buffer Overflows

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



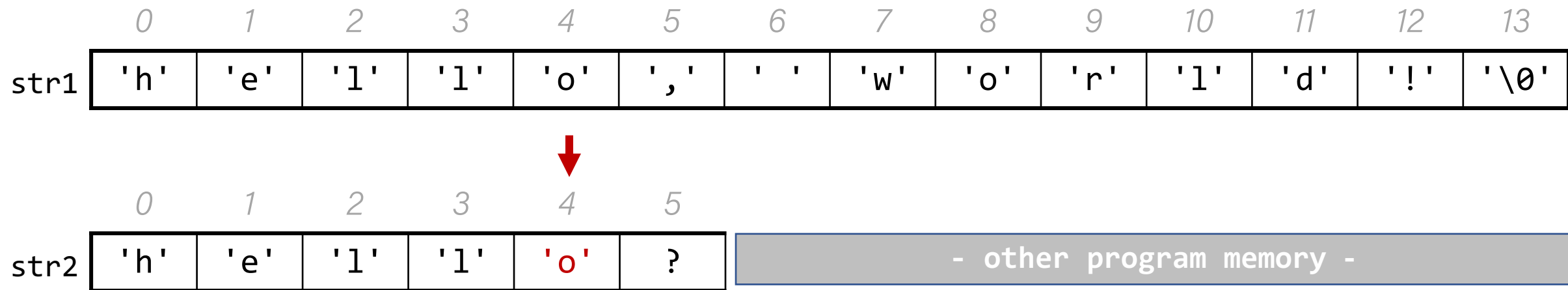
Copying Strings – Buffer Overflows

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



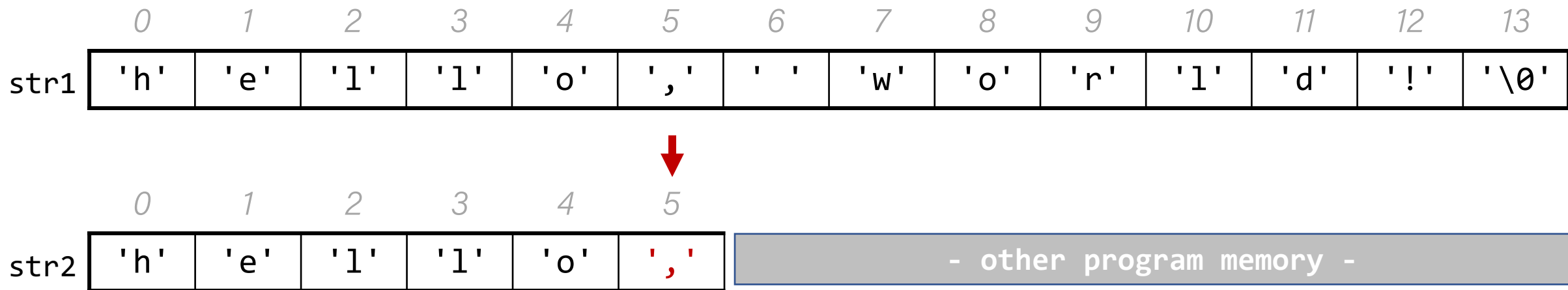
Copying Strings – Buffer Overflows

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



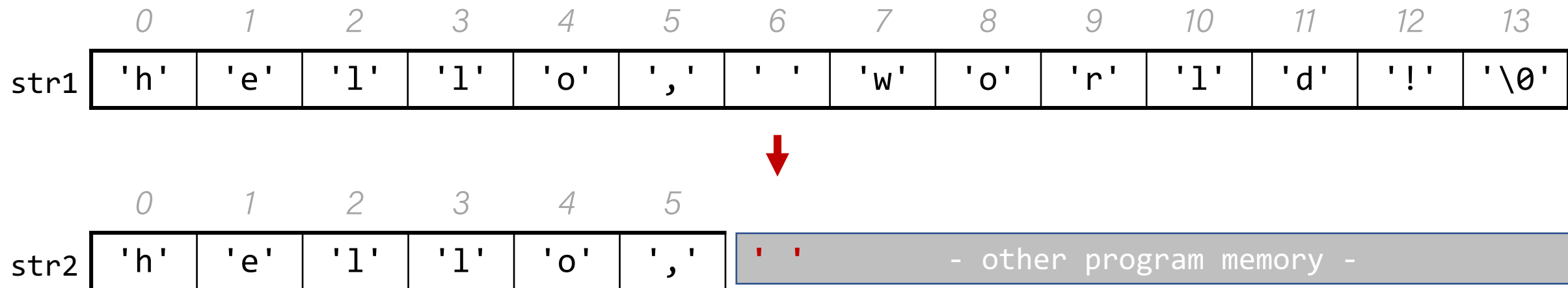
Copying Strings – Buffer Overflows

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



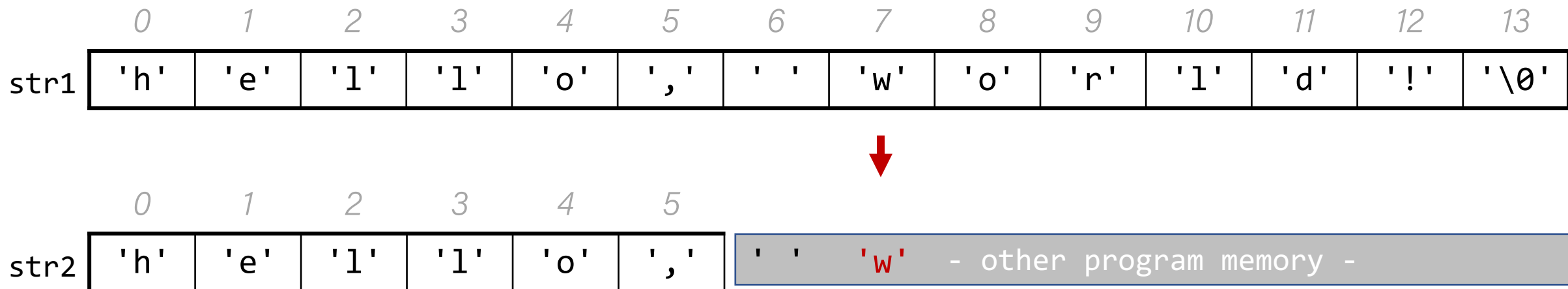
Copying Strings – Buffer Overflows

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



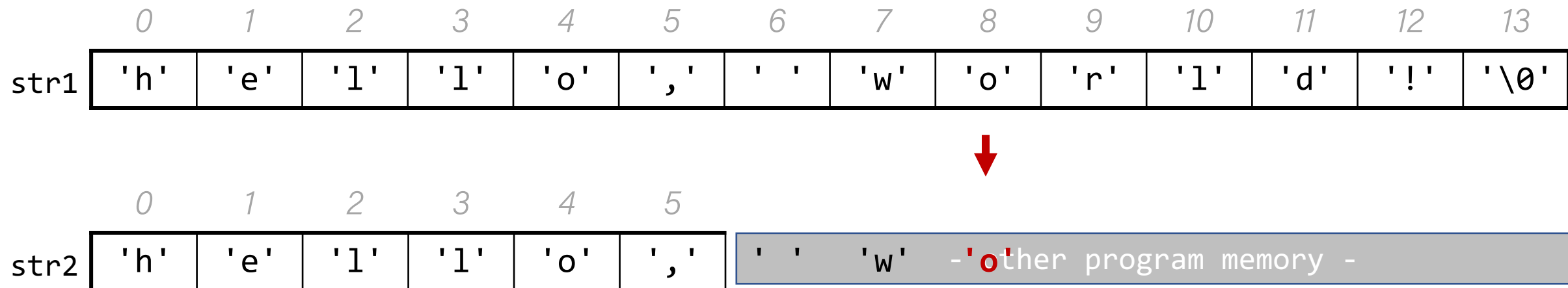
Copying Strings – Buffer Overflows

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



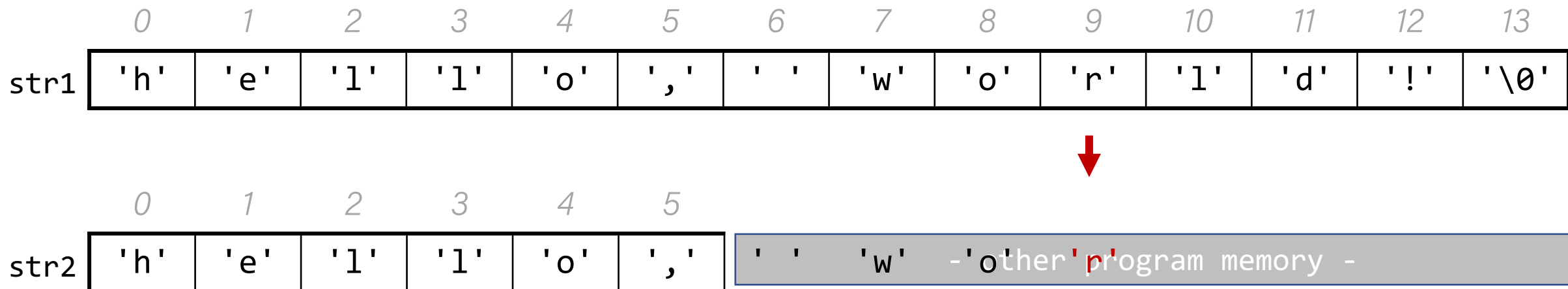
Copying Strings – Buffer Overflows

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



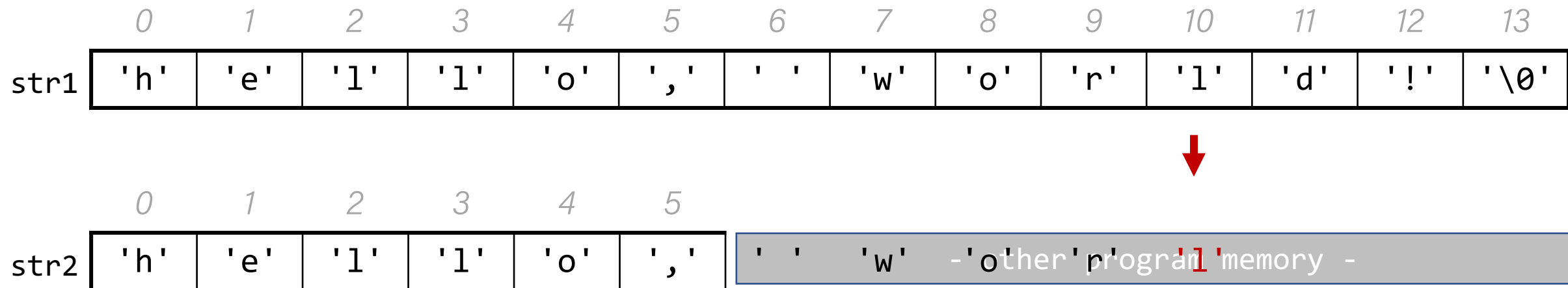
Copying Strings – Buffer Overflows

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



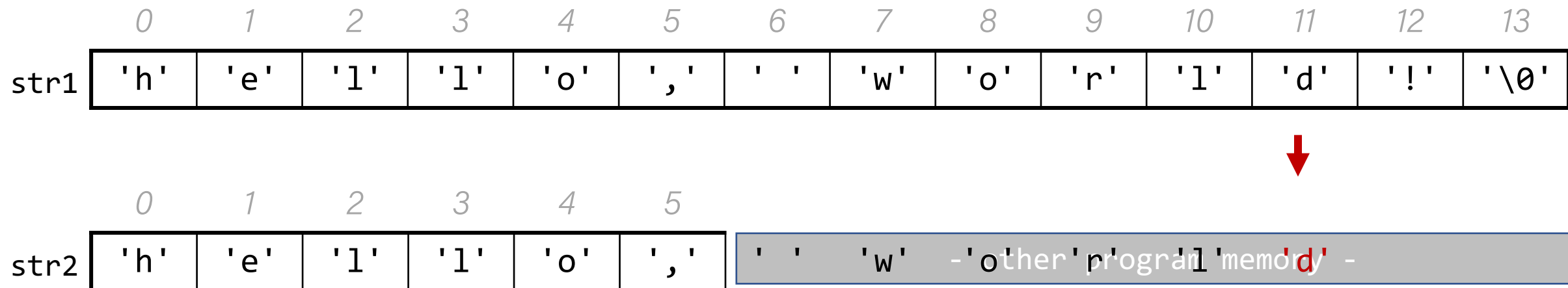
Copying Strings – Buffer Overflows

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



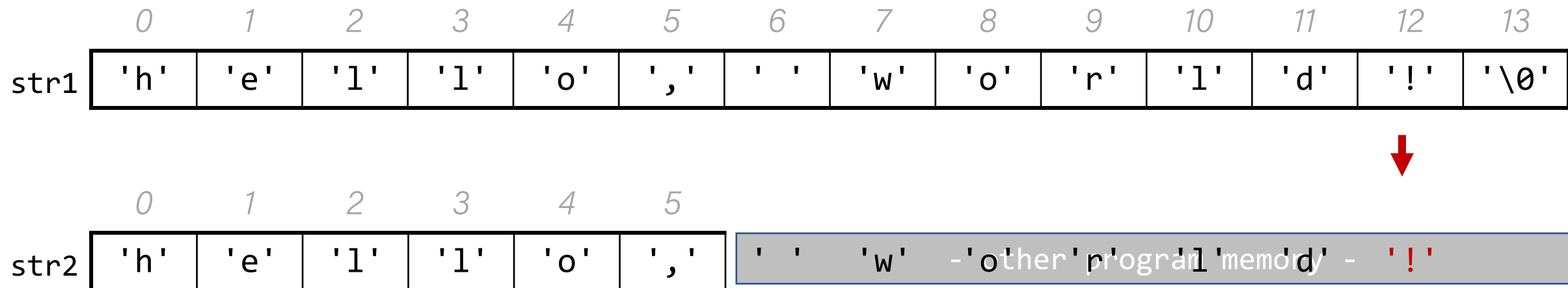
Copying Strings – Buffer Overflows

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



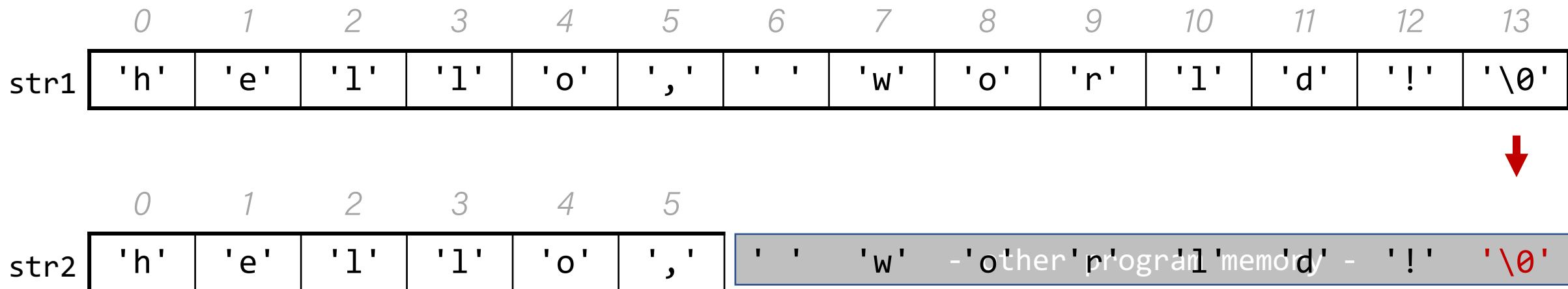
Copying Strings – Buffer Overflows

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



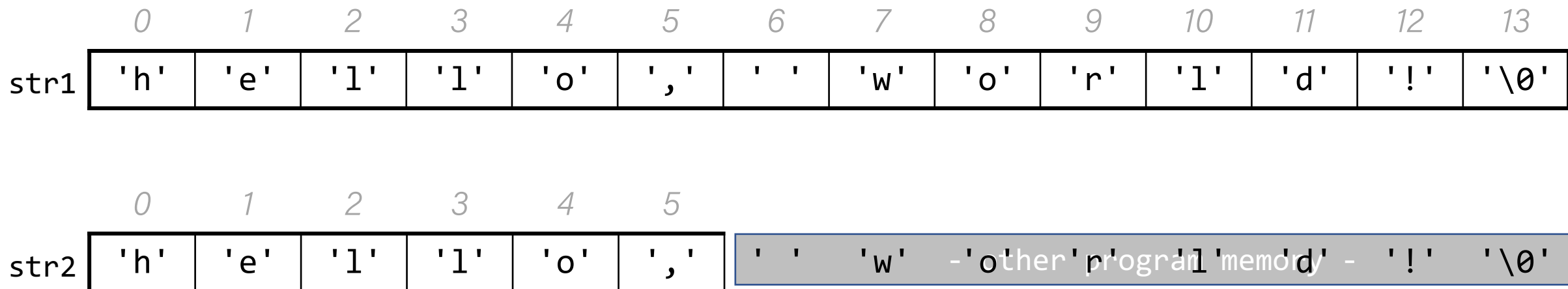
Copying Strings – Buffer Overflows

```
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!
```



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



Copying Strings – strncpy

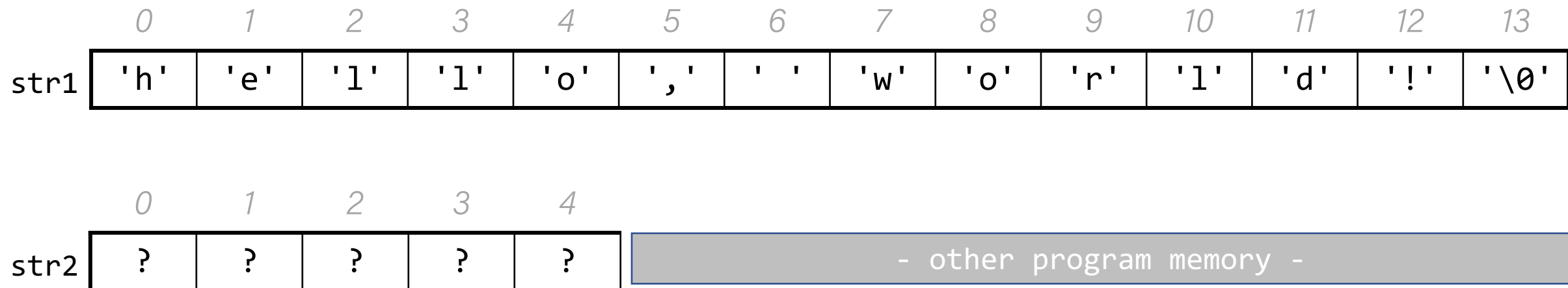
strncpy(dst, src, n): copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **dst** will *not be null terminated*!

```
// 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'`!

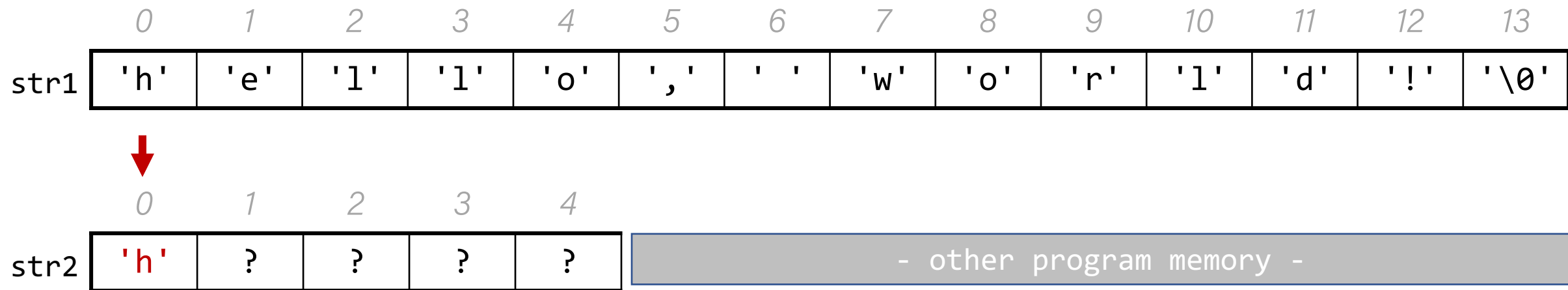
Copying Strings – strncpy

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



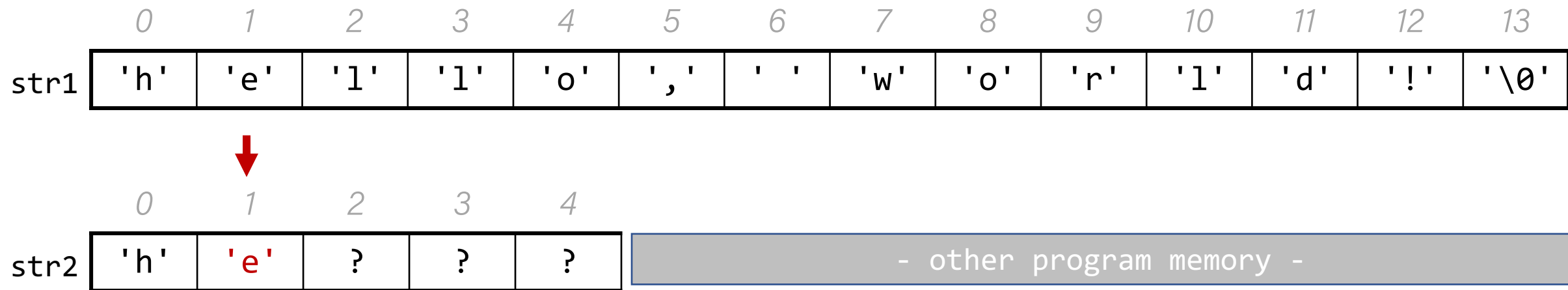
Copying Strings – strncpy

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



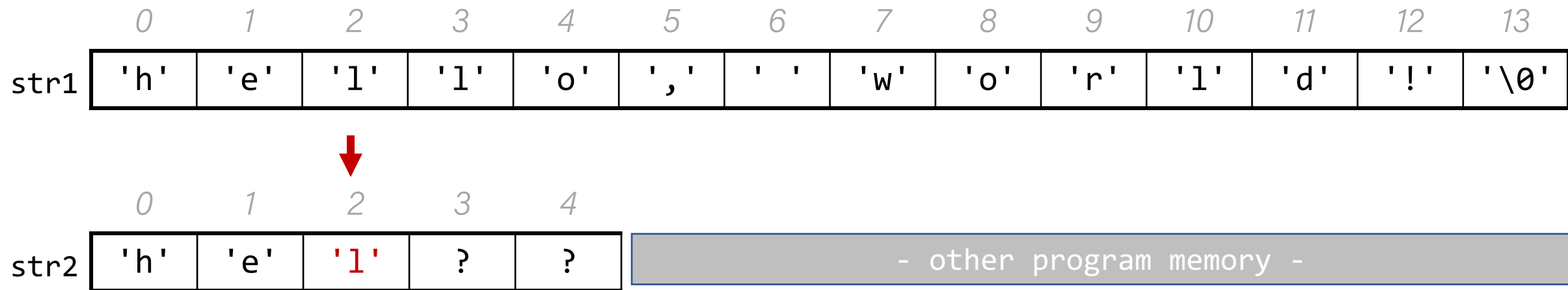
Copying Strings – strncpy

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



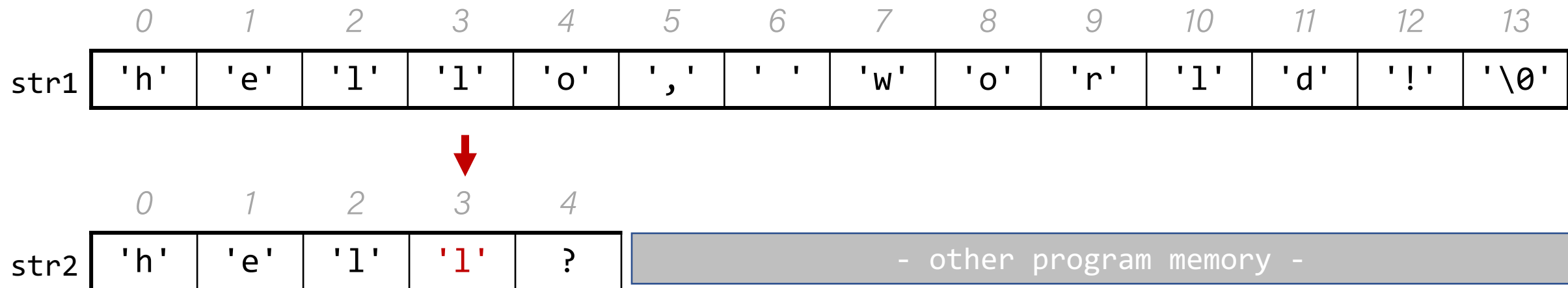
Copying Strings – strncpy

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



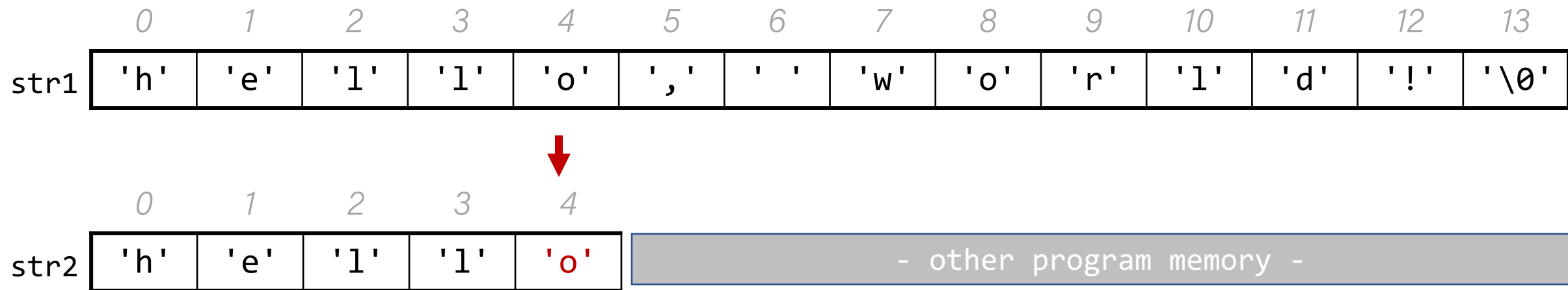
Copying Strings – strncpy

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



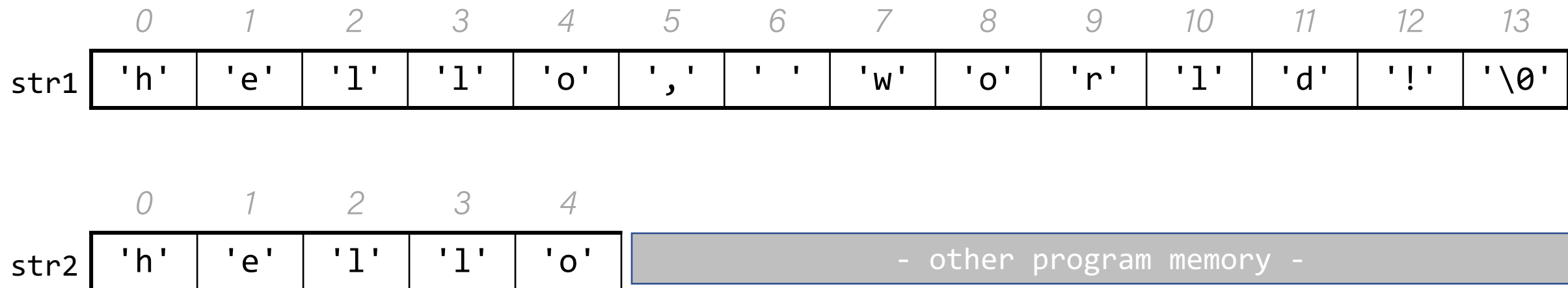
Copying Strings – strncpy

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



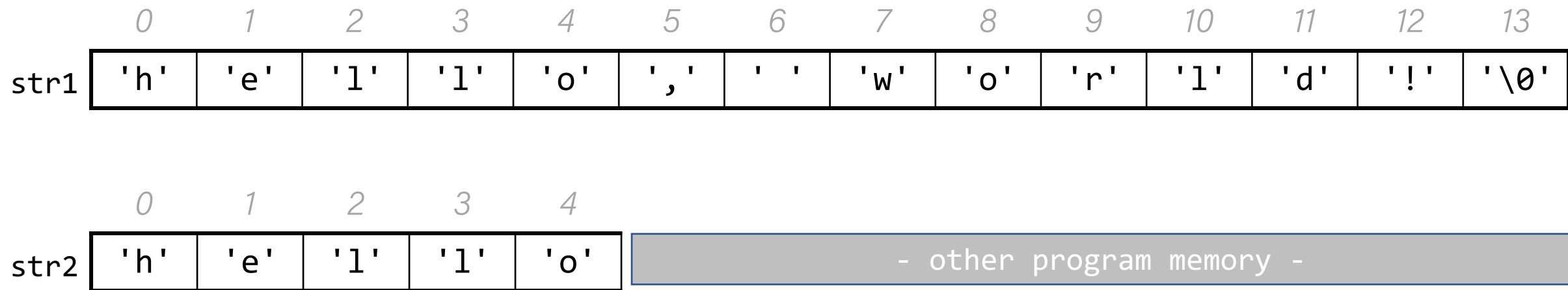
Copying Strings – strncpy

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



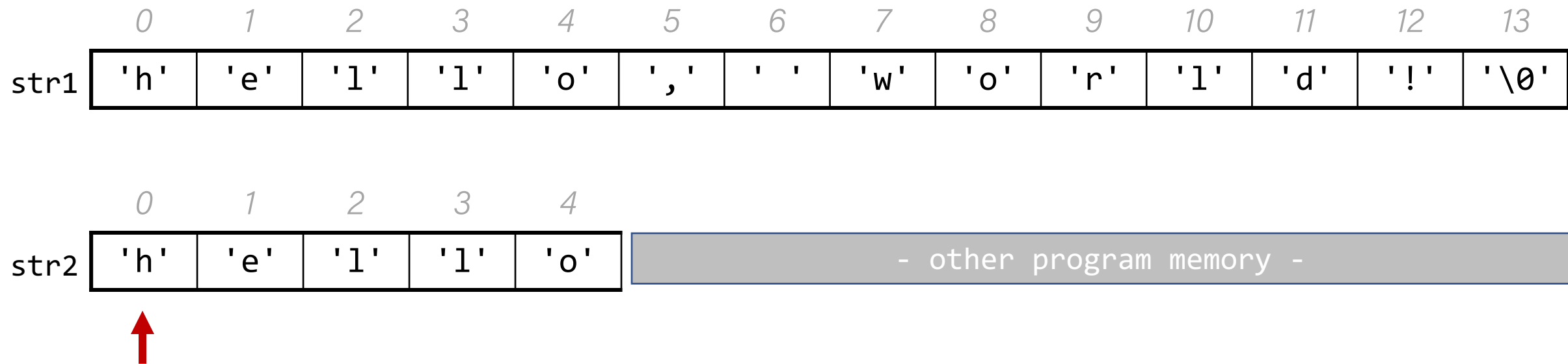
Copying Strings – strncpy

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



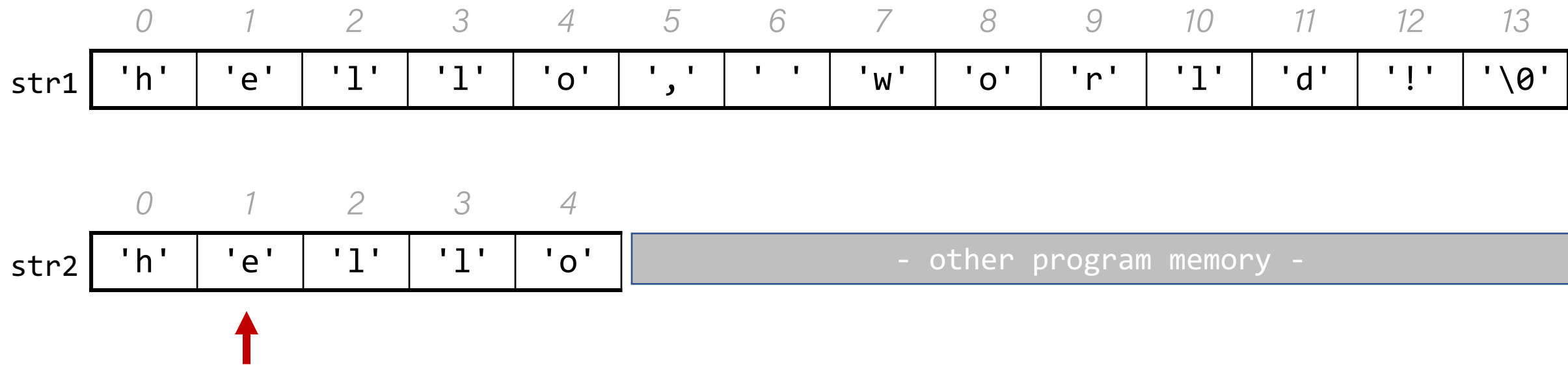
Copying Strings – strncpy

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



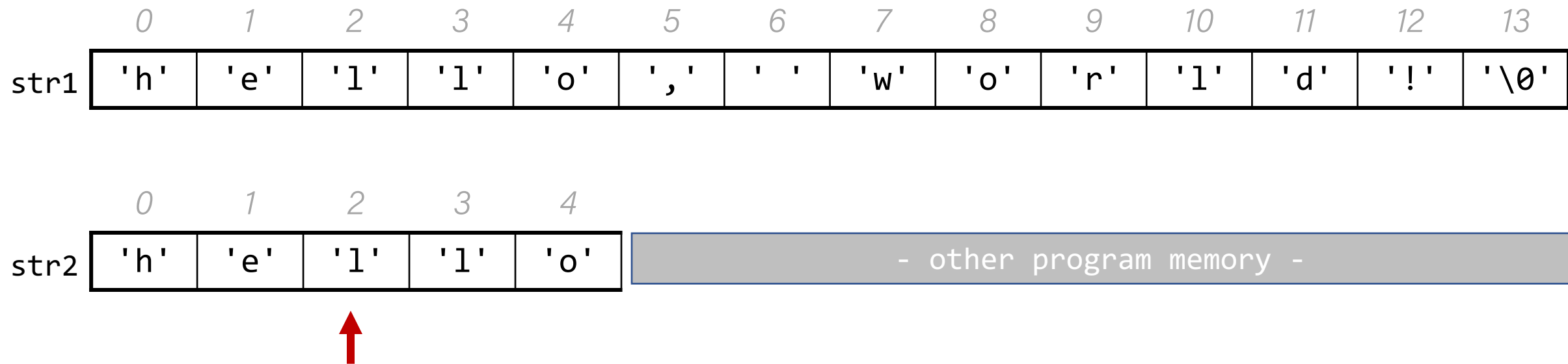
Copying Strings – strncpy

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



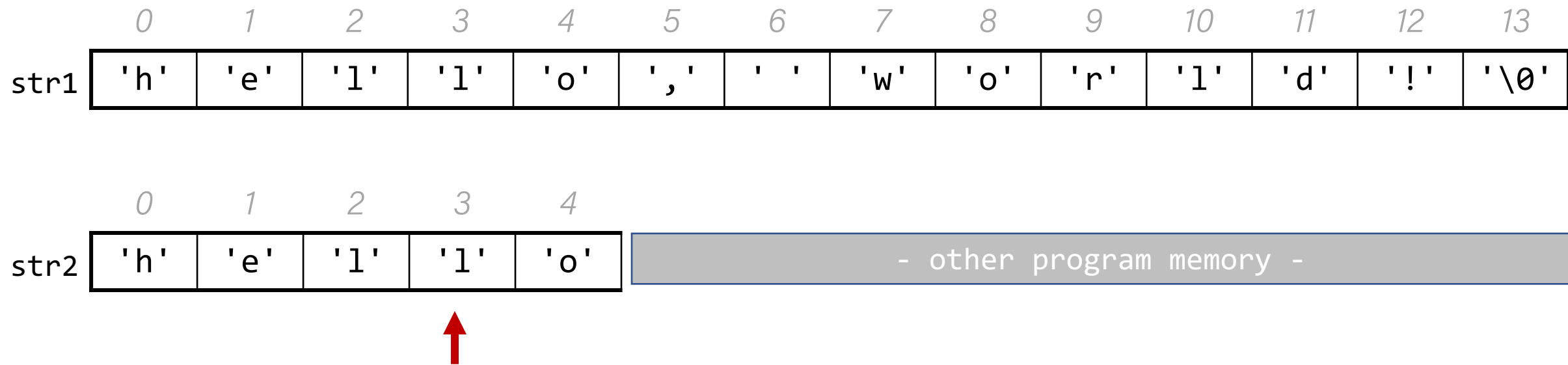
Copying Strings – strncpy

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



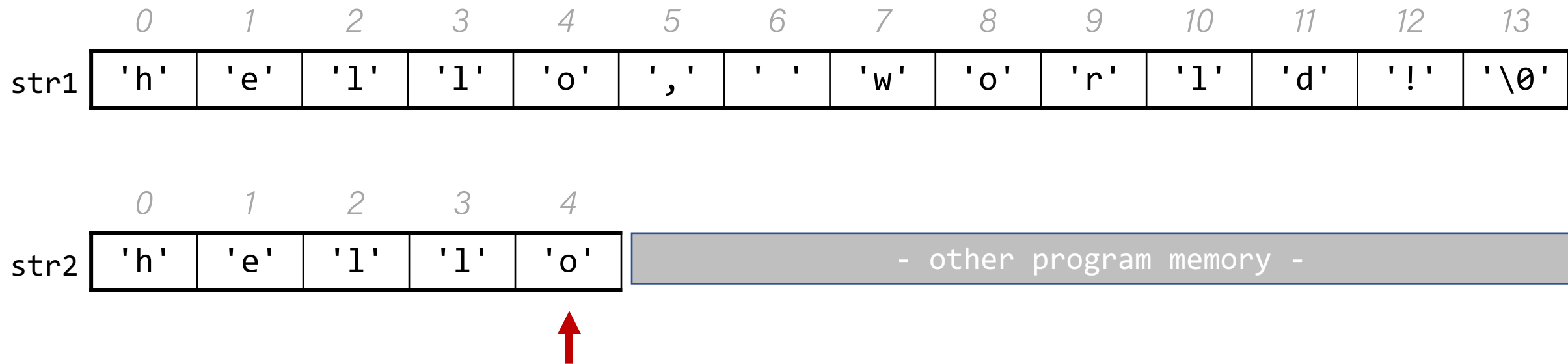
Copying Strings – strncpy

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



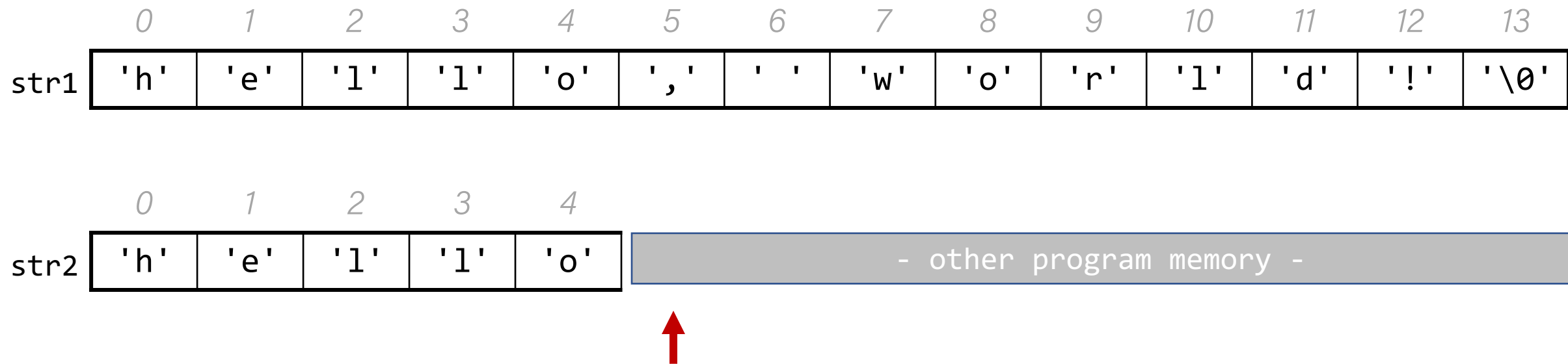
Copying Strings – strncpy

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



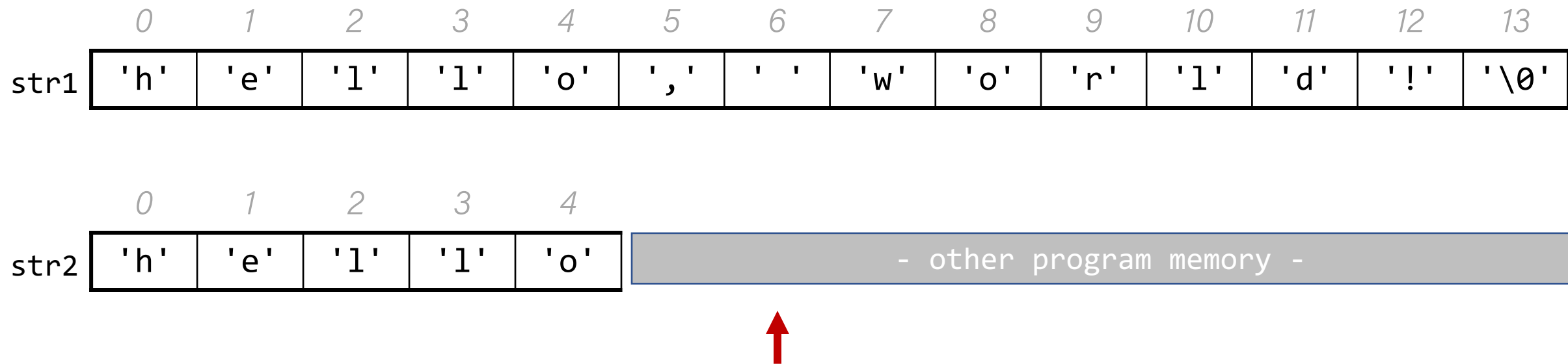
Copying Strings – strncpy

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



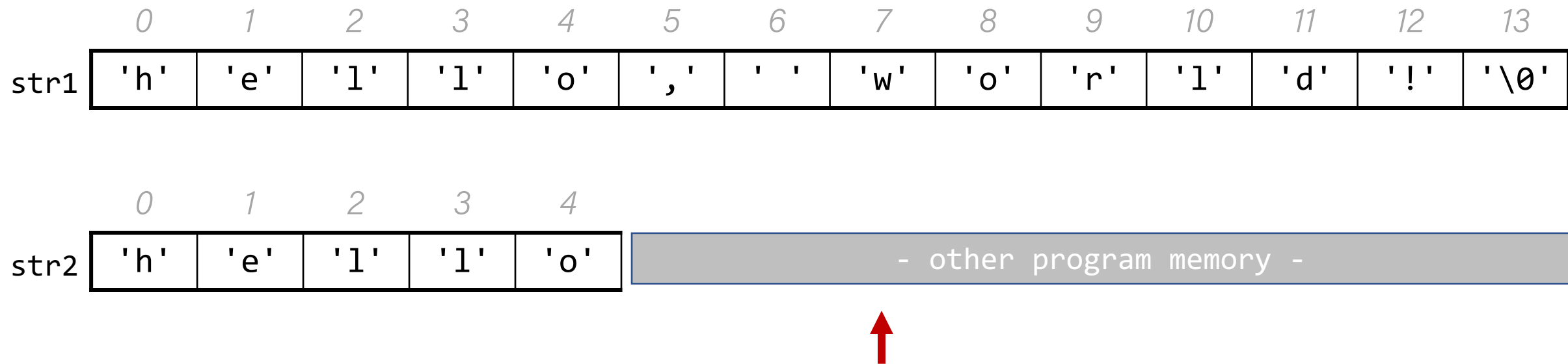
Copying Strings – strncpy

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



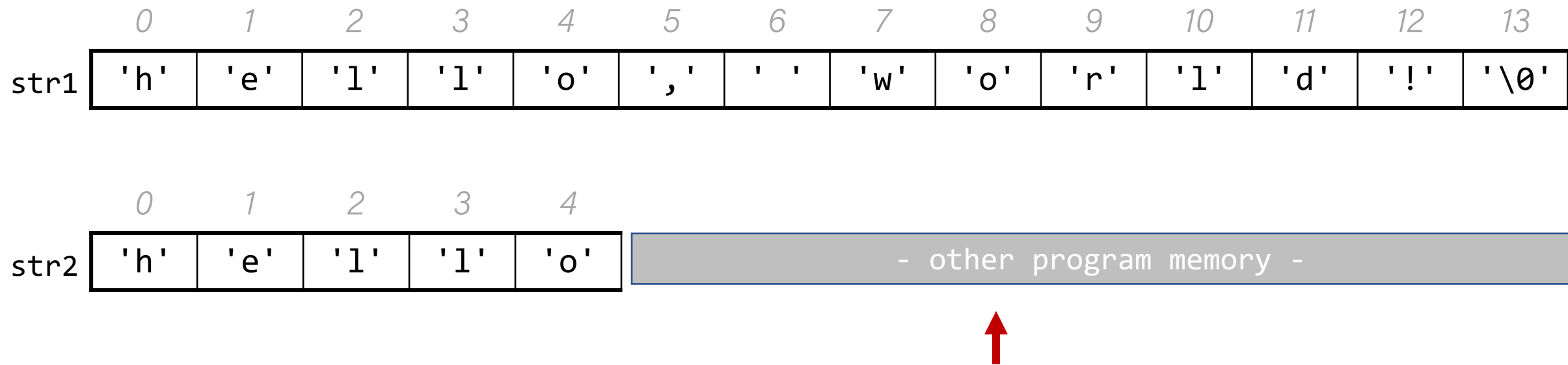
Copying Strings – strncpy

```
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);
```



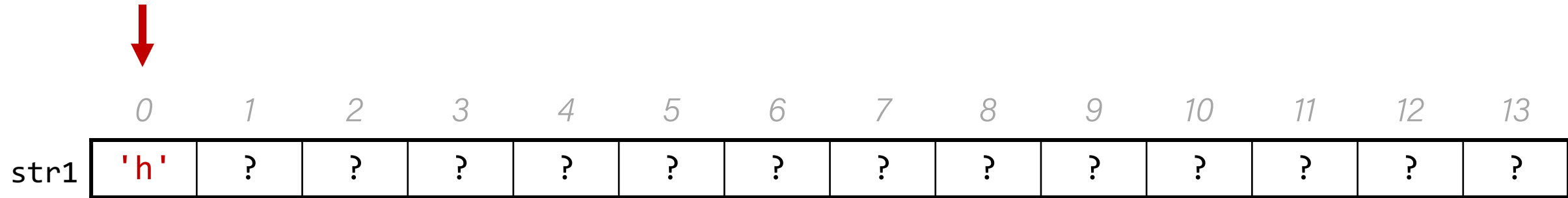
Copying Strings – strncpy

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

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

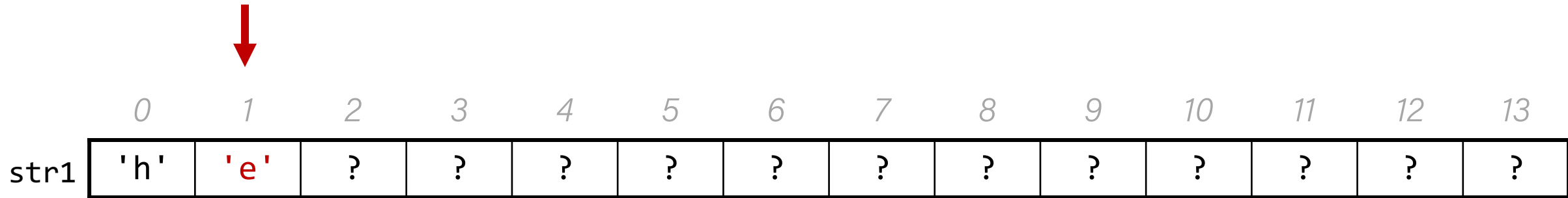
Copying Strings – strncpy

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



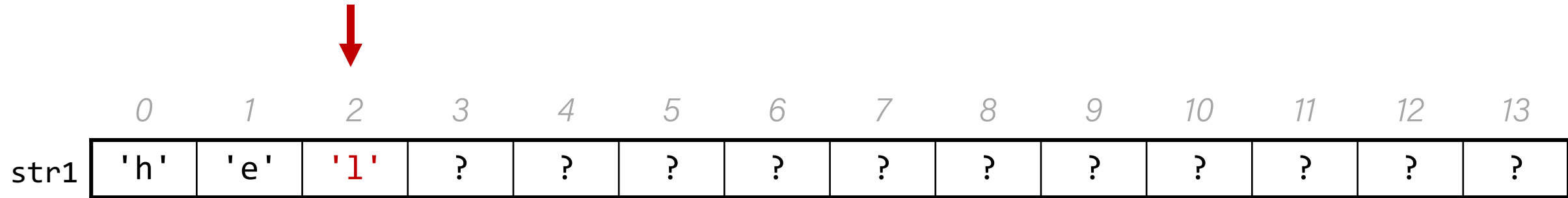
Copying Strings – strncpy

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



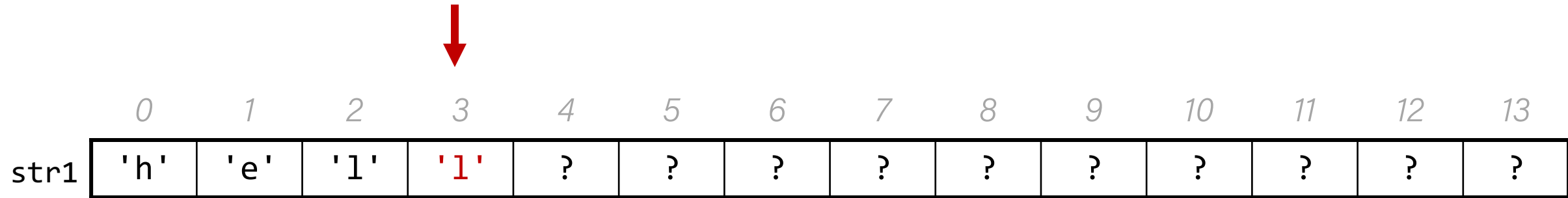
Copying Strings – strncpy

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



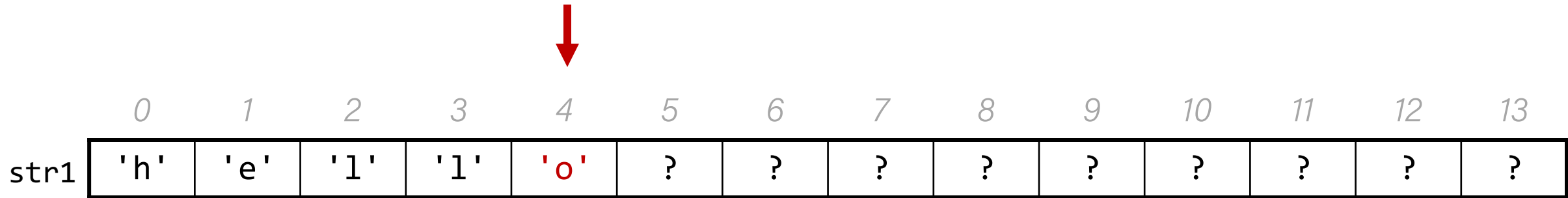
Copying Strings – strncpy

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

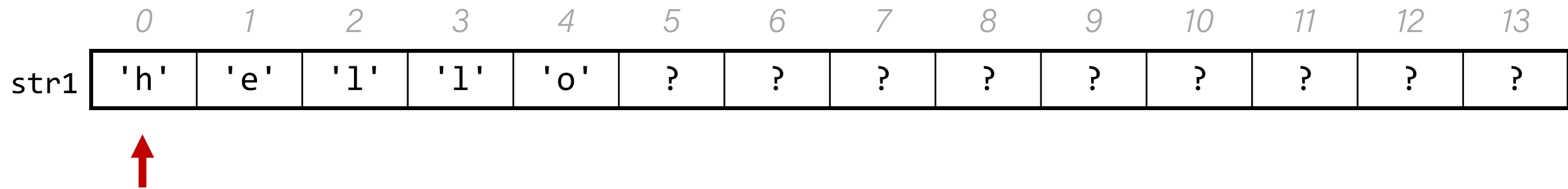
Copying Strings – strncpy

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

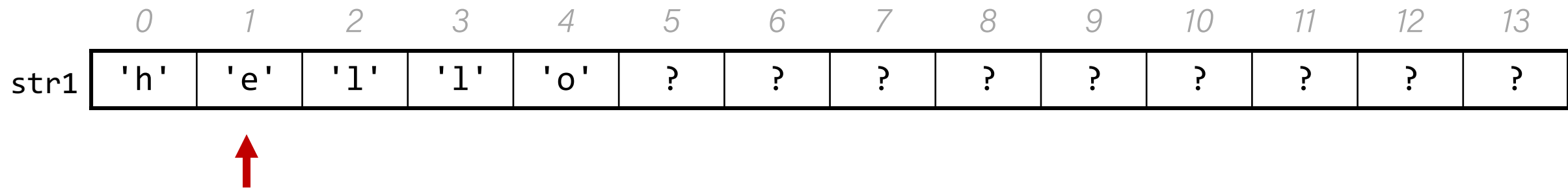
Copying Strings – strncpy

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



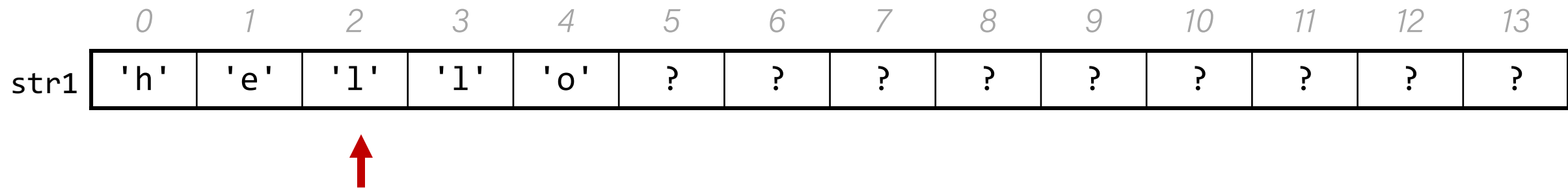
Copying Strings – strncpy

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



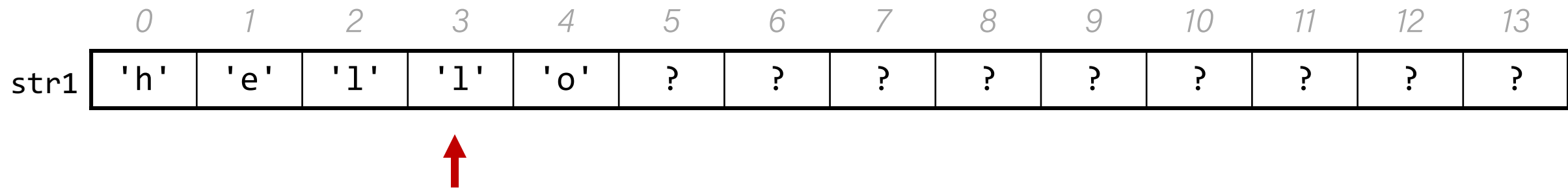
Copying Strings – strncpy

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



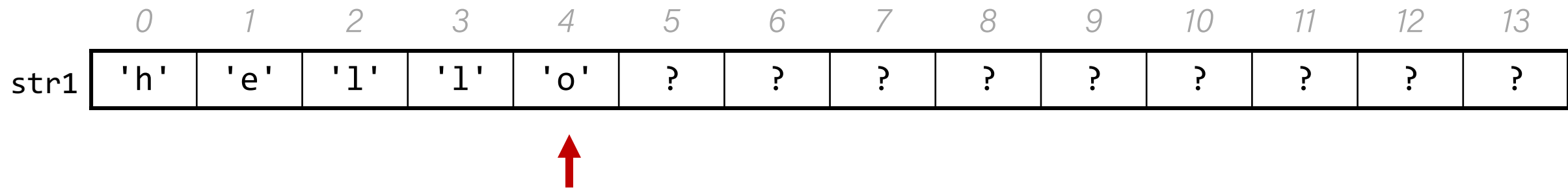
Copying Strings – strncpy

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



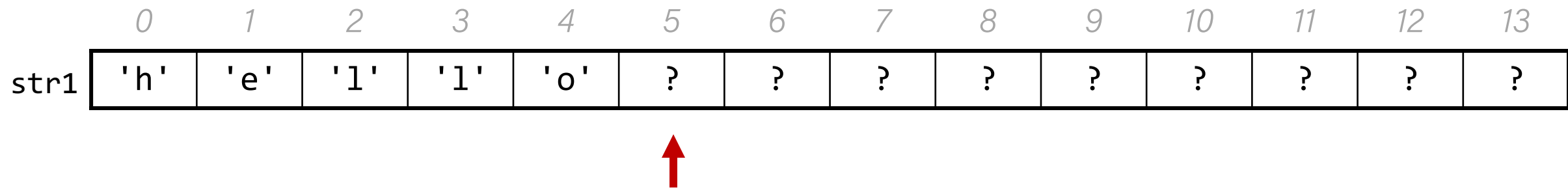
Copying Strings – strncpy

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



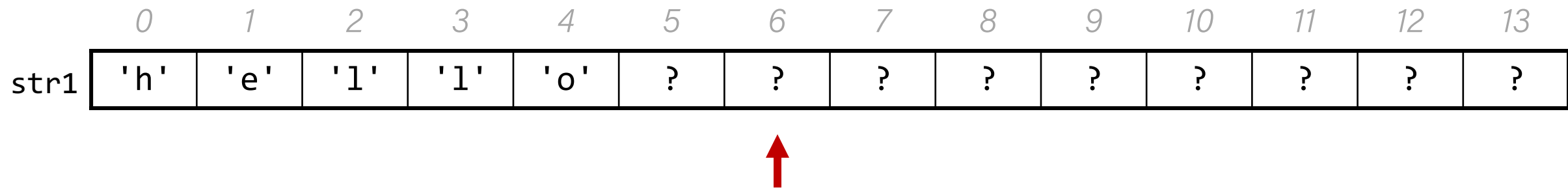
Copying Strings – strncpy

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



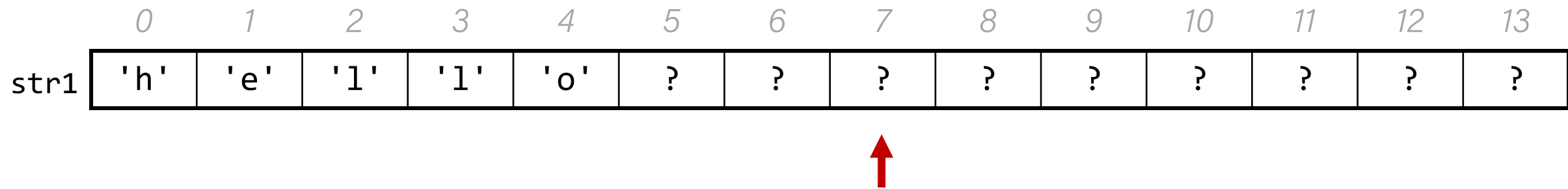
Copying Strings – strncpy

```
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
```

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);
```

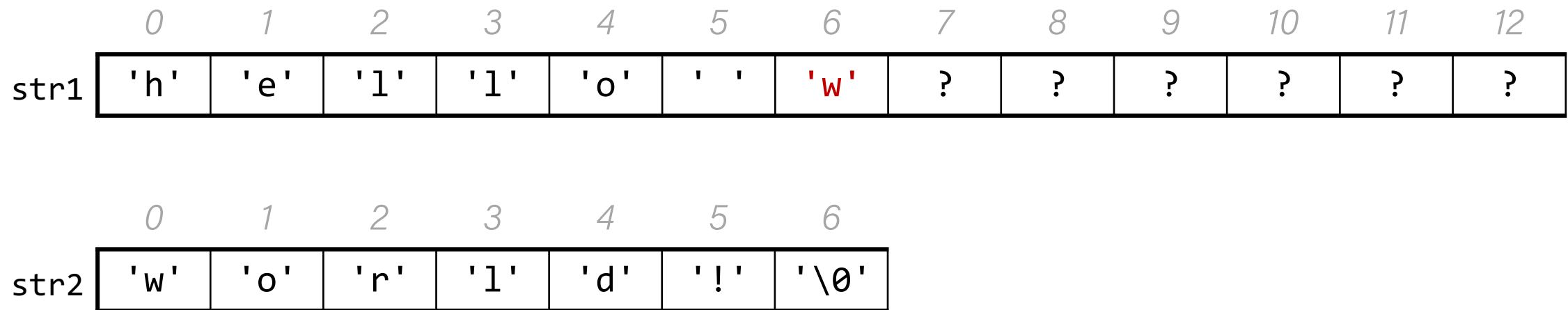
	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'\0'	'?	'?	'?	'?	'?	'?

	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!");
```

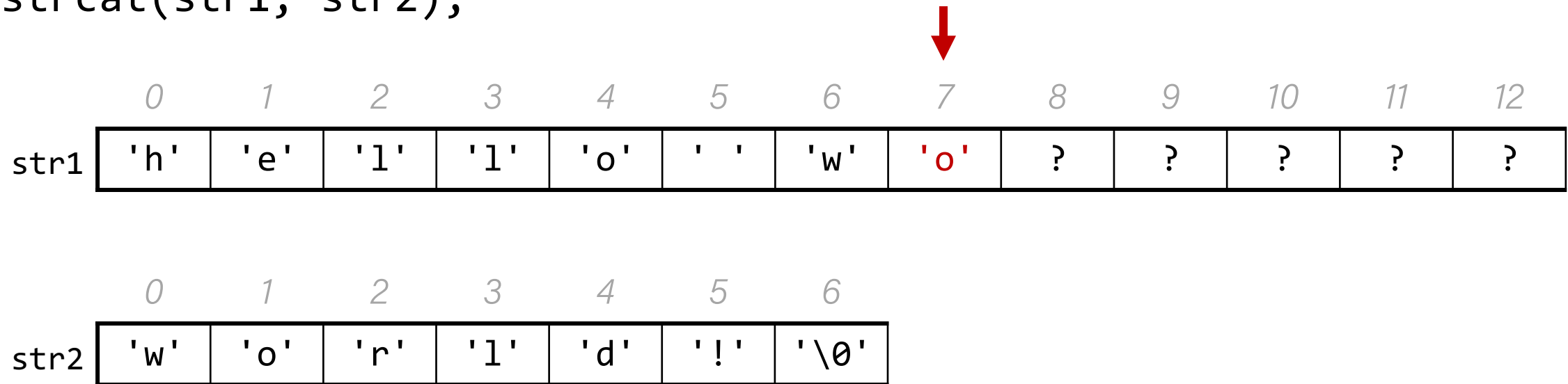
```
strcat(str1, str2);
```



Concatenating Strings

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

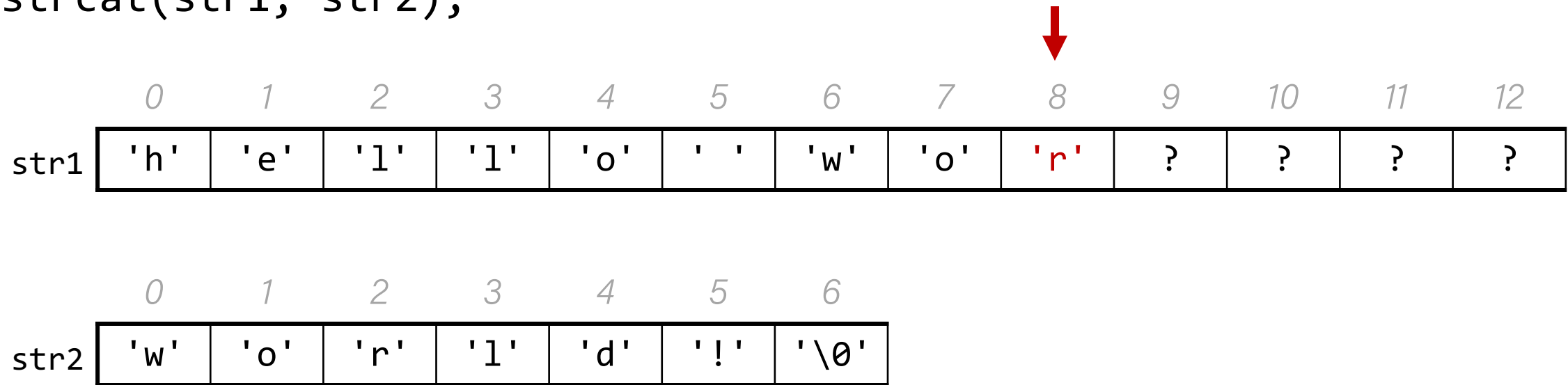
```
strcat(str1, str2);
```



Concatenating Strings

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

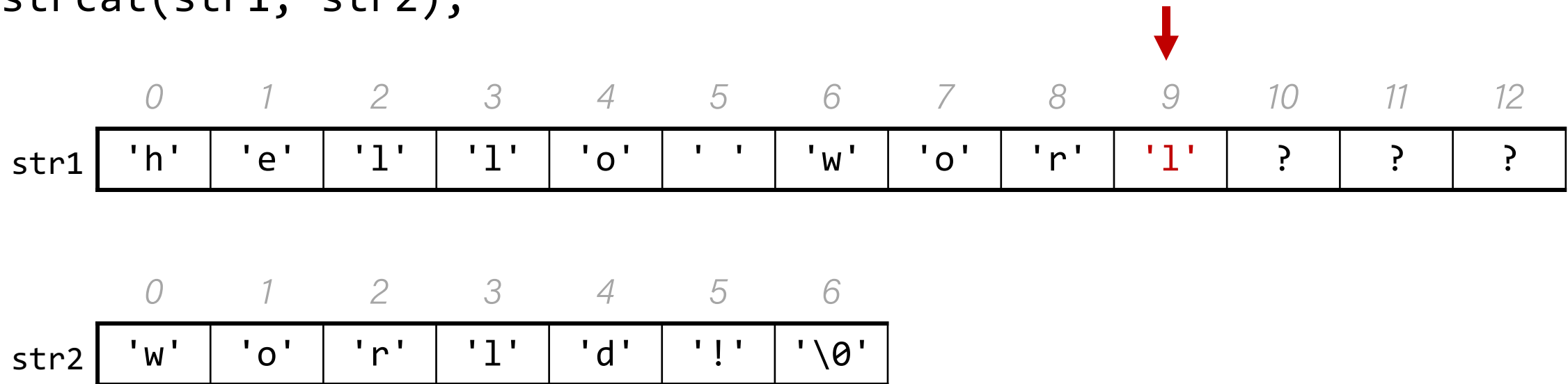
```
strcat(str1, str2);
```



Concatenating Strings

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

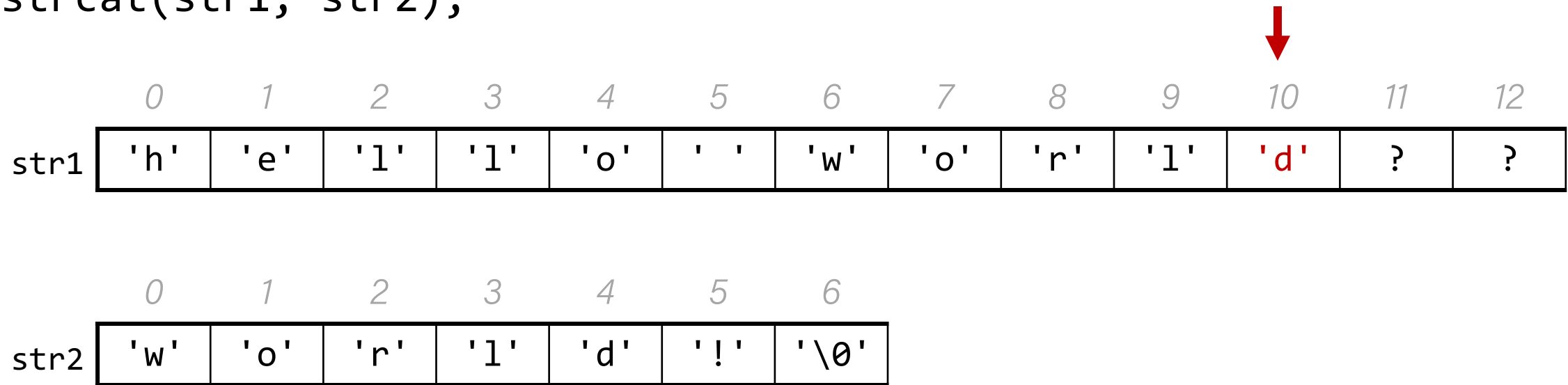
```
strcat(str1, str2);
```



Concatenating Strings

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

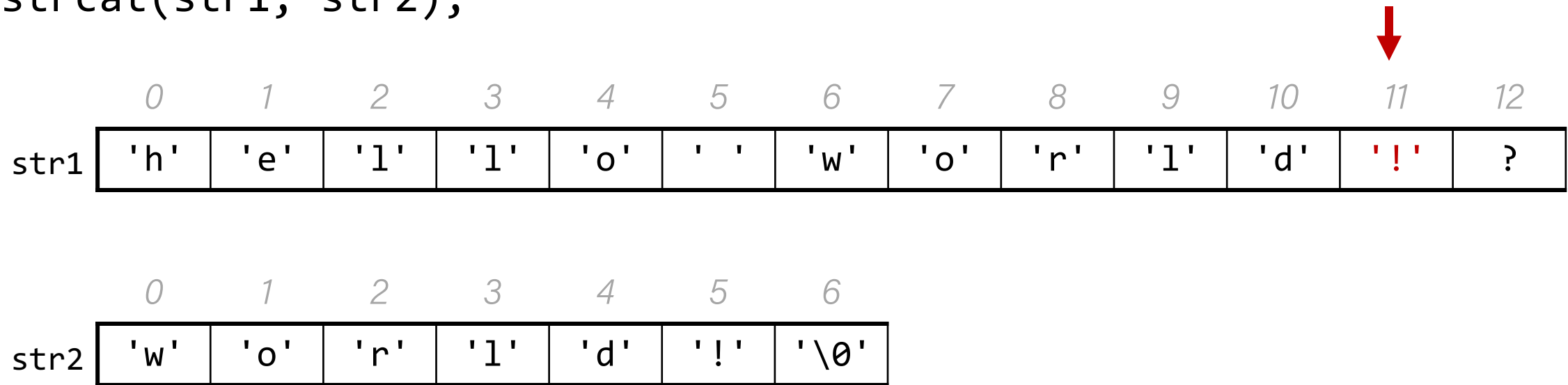
```
strcat(str1, str2);
```



Concatenating Strings

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

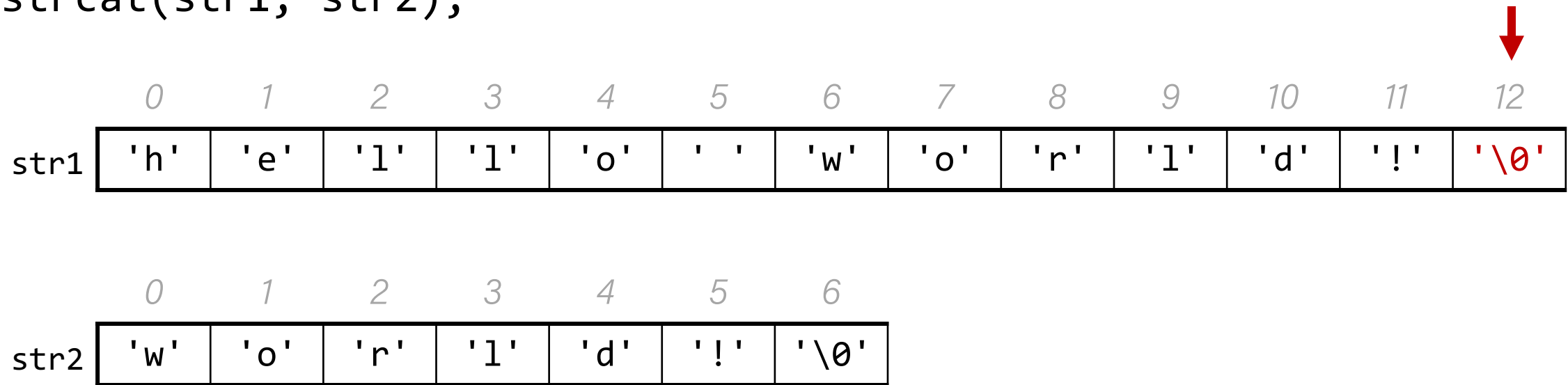
```
strcat(str1, str2);
```



Concatenating Strings

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

```
strcat(str1, str2);
```



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.

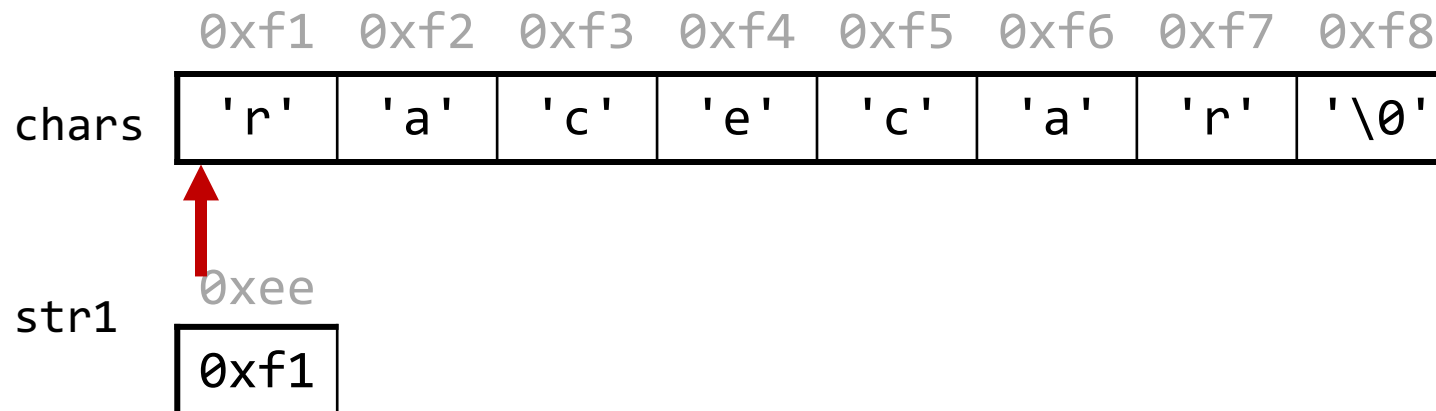
```
char myString[3];  
myString[0] = 'H';  
myString[1] = 'i';  
myString[2] = '\0';
```

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

Substrings

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

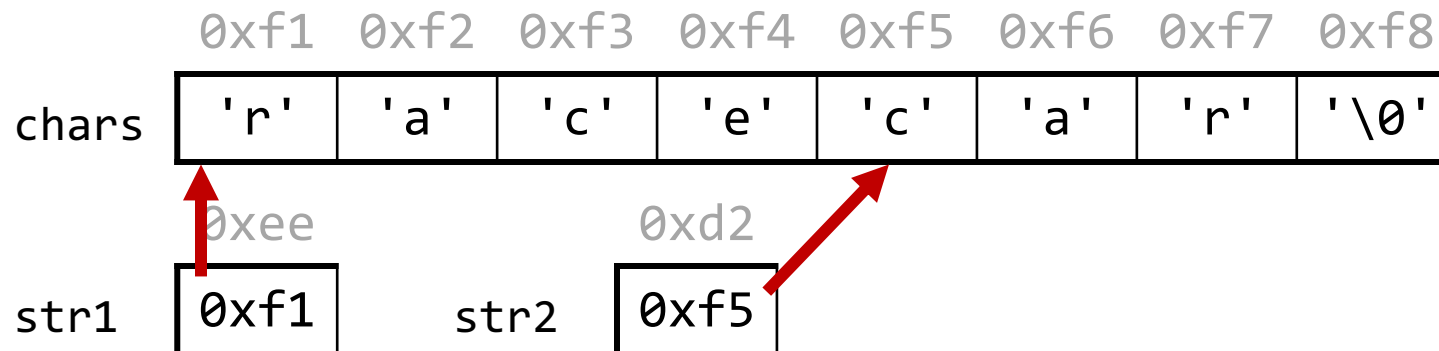
```
// Want just "car"  
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;
```



Substrings

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

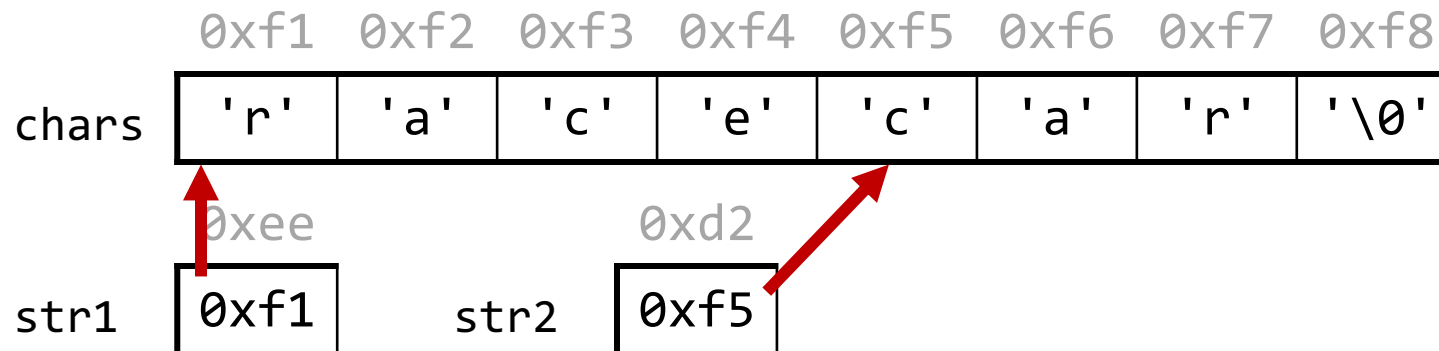
```
// Want just "car"  
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;
```



Substrings

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

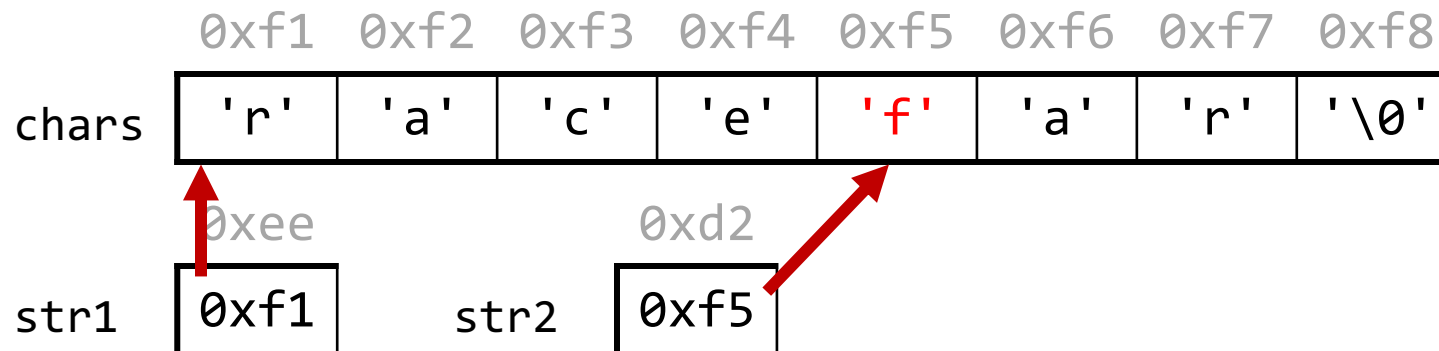
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
printf("%s\n", str1);           // racecar  
printf("%s\n", str2);           // car
```



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!

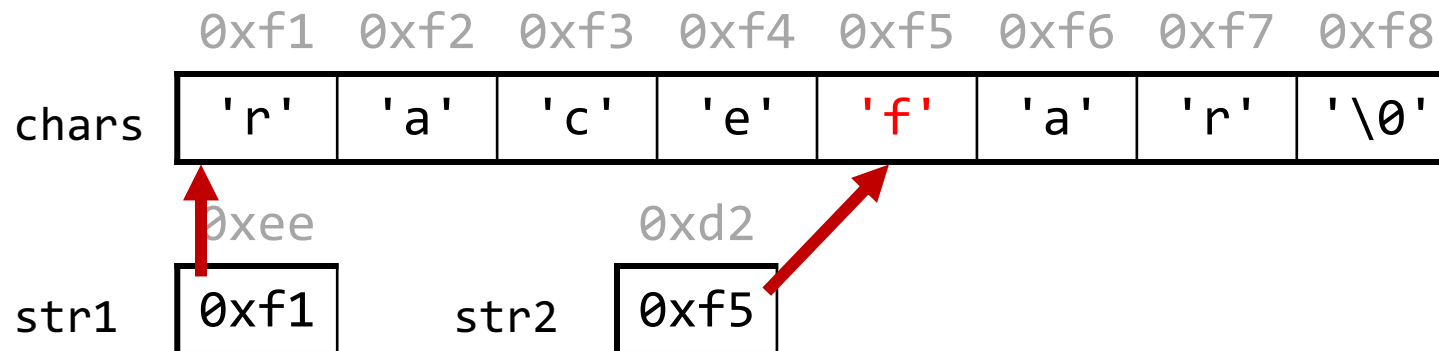
```
char chars[8];  
strcpy(chars, "racecar");  
char *str1 = chars;  
char *str2 = chars + 4;  
str2[0] = 'f';  
printf("%s %s\n", chars, str1);  
printf("%s\n", str2);
```



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
```



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")

Line 6: What is printed?

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



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";  
---  
char *myOtherString = myString;  
myOtherString = somethingElse;
```

// not allowed!

// 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
- Strings
- Common String Operations
- Practice: Diamonds

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("COMP201")` should print:

```
C
CO
COM
COMP
COMP2
COMP20
COMP201
OMP201
MP201
P201
201
01
1
```

Practice: Diamond



diamond.c

Key takeaways

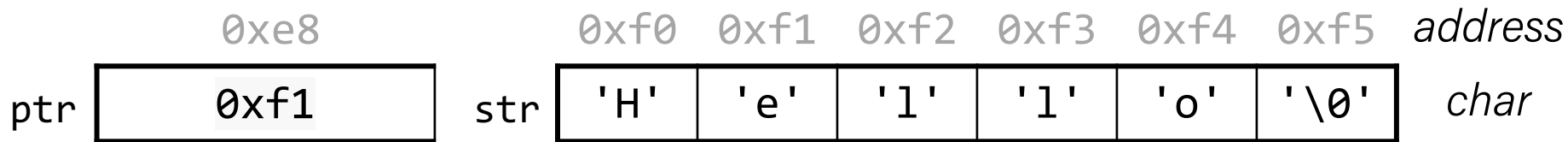
1. Valid strings are null-terminated.

	0xf0	0xf1	0xf2	0xf3	0xf4	0xf5	address
str	'H'	'e'	'l'	'l'	'o'	'\0'	char

```
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.



```
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

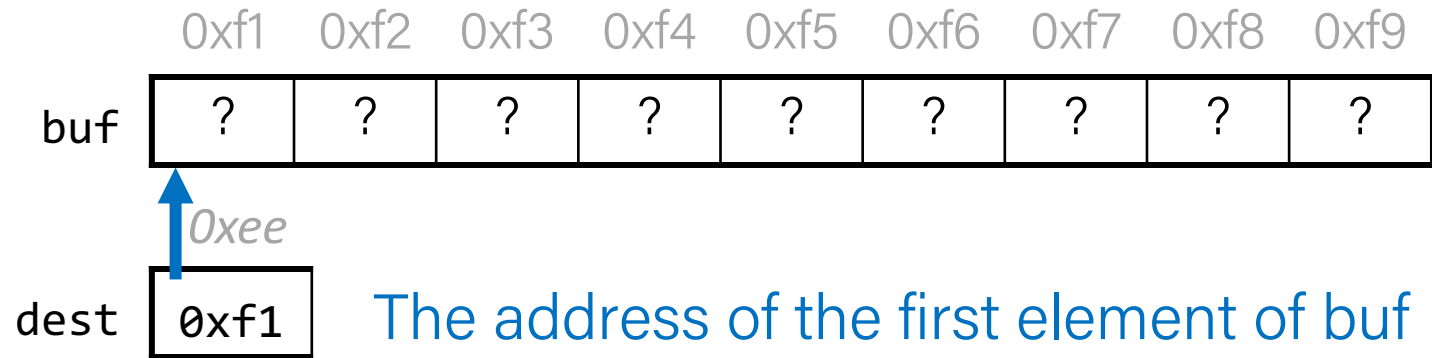
1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.
3. When you pass a `char[]` as a parameter, it is automatically passed as a `char *` (pointer to its first character)

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 6: where string constants like "hello" are stored.
- Lecture 11: what const means

Recap

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

Next time: More strings, pointers