CS107, Lecture 4
C Strings

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

Bits and Bytes Revisited

Data is really stored in binary

```
int x = 5;     // really 0b00…0101 in memory!

char ch = 'A'; // really 0b01000001 in memory!
```

We know what that binary representation is for integers

```
int x = 5;     // really 0b00…0101 in memory!

char ch = 'A'; // really 0b01000001 in memory! (65)


int y = -5;    // two’s complement: 0b111...11011

unsigned long z = ULONG_MAX;  // 0b111...111
```

We can manipulate that binary representation with bitwise operators

```
int x = 5; // in binary it’s 0b0000....00101

if (x & 0x4 != 0) {
    printf("x's third bit from the right is on\n");
}

// turn on the 2nd bit from the right
x |= 0x2;

int y = 1;
y <<= x;  // y is shifted by 7 places
```

Bitwise OR (|)

| with 1 is useful for turning select bits on.

```
int x = 5; // 0b101

// Turn on the 2nd bit from the right
x |= 0x2; // 0b111

// Turn on the 9th bit from the right
x |= (1 << 8);
```

Bitwise OR (|)

| is useful for taking the union of bits.

```
int x = 5;      // 0b00101
int y = 26;     // 0b11010
int z = x | y; // 0b11111
printf("%d\n", z); // 31
```

Bitwise AND (&)

& with 0 is useful for turning select bits off.

```
int x = 5; // 0b101

// Turn off the 3rd bit from the right
x &= -5; // -5 is 0b111...1011
```

Bitwise AND (&)

& is useful for taking the intersection of bits.

```
int x = 21;     // 0b10101
int y = 27;     // 0b11011
int z = x & y; // 0b10001
printf("%d\n", z); // 17
```

Bitwise XOR (^)

^ with 1 is useful for flipping select bits.

```
int x = 5; // 0b101

// Flip the 2nd bit from the right
x ^= 2; // 0b111
```

Bitwise NOT (~)

~ is useful for flipping all bits.

```
int x = 5; // 0b101

// Flip all bits
x = ~x;  // 0b11111...1010, which is -6

// Take two's complement (same as negating)
int y = ~x + 1; // same as -x
```

Bitwise SHIFT (>> or <<)

>> and << are useful for moving bits.

```
int x = 5; // 0b00101
x <<= 2;   // 0b10100

// Check if 6th bit from the right is on
if (x & (1 << 5)) {...
// or
if ((x >> 5) & 1) {...
```

Number Literal Suffixes

U makes a literal unsigned, and **L** makes a literal a long.

```
int w = -5 >> 1;  // 0b1111...1101, -5
int x = -5U >> 1; // 0b0111...1101, 2147483645

int y = 1 << 32;  // 0! (technically undefined)
int z = 1L << 32; // 4294967296
```

A variable and its binary representation are one and the same

```
int x = 5; // in binary it’s 0b0000....00101

// turn on the 2nd bit from the right
x |= 0x2;

int y = 1;
y <<= x;  // y shifted by 7 places
```

We can better understand overflow and type conversion behavior

```
// Overflow
char x = 126;   // 0b01111110
x += 2; // -128!  0b10000000

// Expansion
long y = x;     // 0b1111...10000000

// Truncation helps us isolate the byte we want
int z = 0xaabbccdd;
char justCs = z >> 8;
```

We can inspect our program with GDB

- GDB is a **command-line debugger**, a text-based debugger with similar functionality to other debuggers you may have used, such as in Qt Creator
- It lets you put **breakpoints** at specific places in your program to pause there
- It lets you step through execution line by line
- It lets you print out values of variables in various ways (including binary)
- It lets you track down where your program crashed
- And much, much more!

GDB is essential to your success in CS107 this quarter! We'll be building our familiarity with GDB over the course of the quarter.

gdb on a program

• `gdb myprogram`	run gdb on executable
• `b`	Set breakpoint on a function (e.g., `b main`) or line (`b 42`)
• `r 82`	Run with provided args
• `n, s, continue`	control forward execution (next, step into, continue)
• `p`	print variable (`p varname`) or evaluated expression (`p 3L << 10`)
• `p/t, p/x`	binary and hex formats.
• `p/d, p/u, p/c`	
• `info`	`args, locals`

Important: gdb does not run the current line until you hit "next"

gdb: highly recommended

At this point, setting breakpoints/stepping in gdb may seem like overkill for what could otherwise be achieved by copious **printf** statements.

However, gdb is incredibly useful for assign1 (and all assignments):

- A fast “C interpreter”: p + <expression>
  - Sandbox/try out ideas around bitshift operators, signed/unsigned types, etc.
  - Can print values out in binary!
  - Once you’re happy, then make changes to your C file
- **Tip**: Open two terminal windows and SSH into myth in both
  - Keep one for emacs, the other for gdb/command-line
  - Easily reference C file line numbers and variables while accessing gdb
- **Tip**: Every time you update your C file, **make** and then rerun gdb.

Gdb takes practice! But the payoff is tremendous! ☺

<u>CS107 Topic 2</u>: How can a computer represent and manipulate more complex data like text?

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (this time)
- Helps us better understand buffer overflows, a common bug (this time)
- Introduces us to pointers, because strings can be pointers (next time)

assign2: implement 2 functions a 1 program using those functions to find the location of different built-in commands in the filesystem. You'll write functions to extract a list of possible locations and tokenize that list of locations.

Learning Goals

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

Lecture Plan

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

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:*

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

Common `ctype.h` Functions

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

index	*0*	*1*	*2*	*3*	*4*	*5*
char	`'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.

C Strings

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

String Length

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

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'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(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.
`strchr(`***`str, ch`***`)` `strrchr(`***`str, ch`***`)`	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.
`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***`needle`*** in ***`haystack`***, or ***`NULL`*** if ***`needle`*** was not found in ***`haystack`***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***`src`*** to ***`dst`***, including null-terminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and <u>does not</u> add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. <u>Always</u> adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***`str`*** which contains <u>only</u> characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does <u>not</u> contain any characters in ***`reject`***.

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).

`strcmp(`***`str1, str2`***`),`
`strncmp(`***`str1, str2, n`***`)`
	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.

`strchr(`***`str, ch`***`)`
`strrchr(`***`str, ch`***`)`
	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.

`strstr(`***`haystack, n`***	first occurrence of ot found in ***`haystack`***.

`strcpy(`***`dst, src`***`),`
`strncpy(`***`dst, src, n`***`)`
	rminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and <u>does not</u> add null-terminating char.

`strcat(`***`dst, src`***`),`
`strncat(`***`dst, src, n`***`)`
	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. <u>Always</u> adds a null-terminating character.

`strspn(`***`str, accept`***`),`
`strcspn(`***`str, reject`***`)`
	`strspn` returns the length of the initial part of ***`str`*** which contains <u>only</u> characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does <u>not</u> contain any characters in ***`reject`***.

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**.

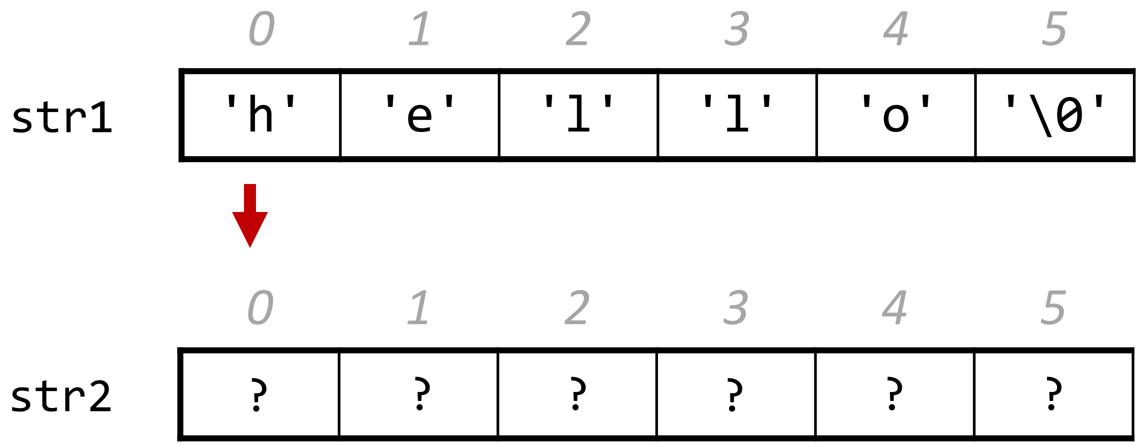
The string library: `strcpy`

`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`

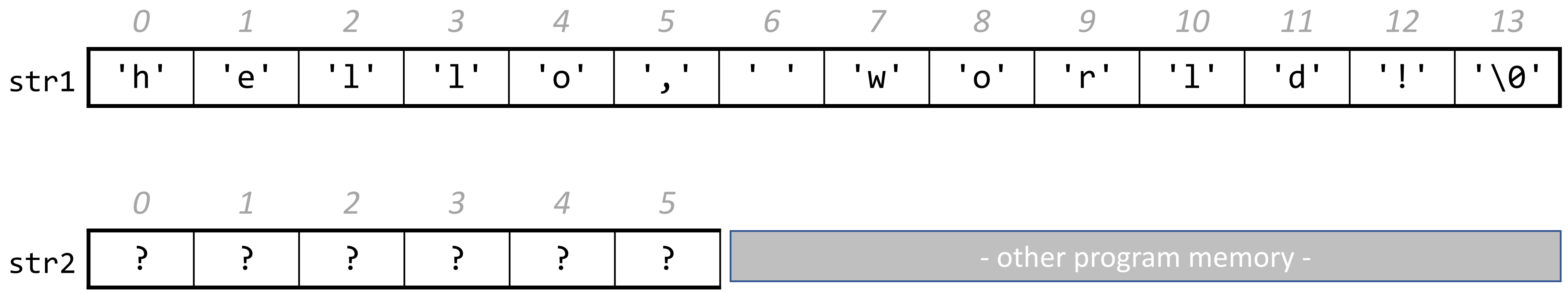
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
    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!

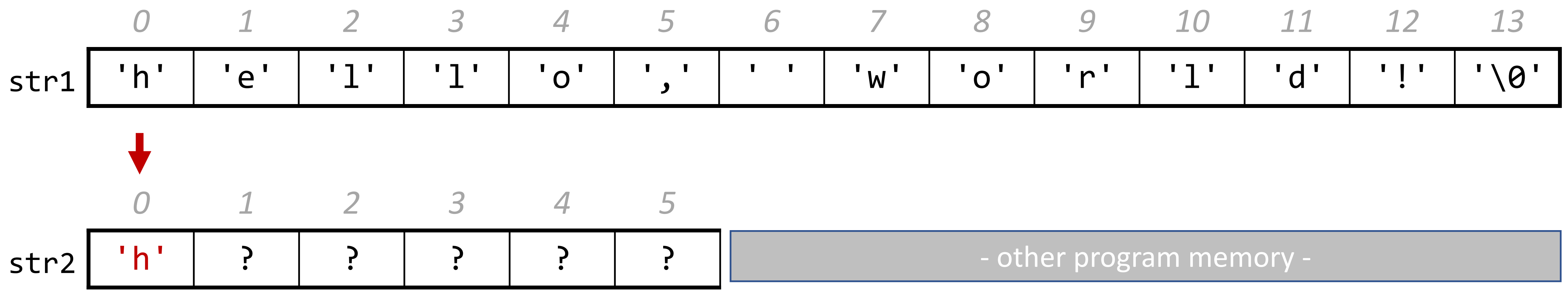
Copying Strings – Buffer Overflows

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



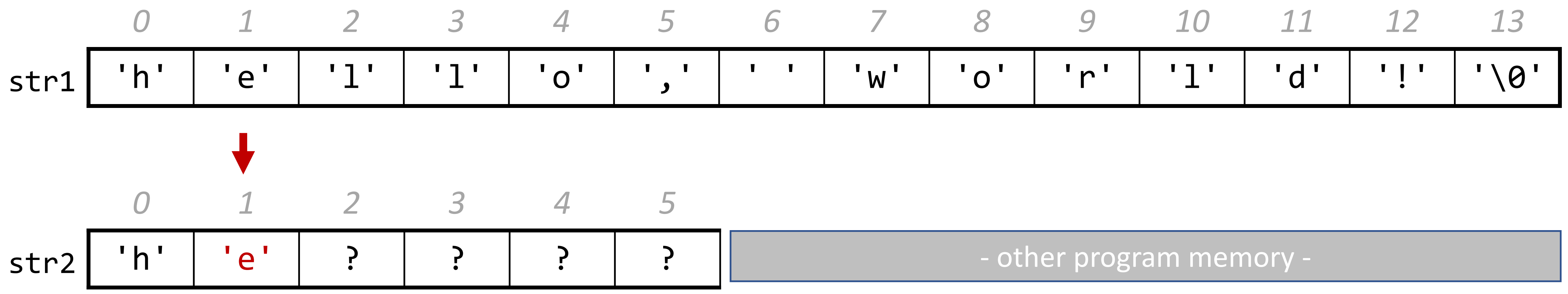
Copying Strings – Buffer Overflows

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



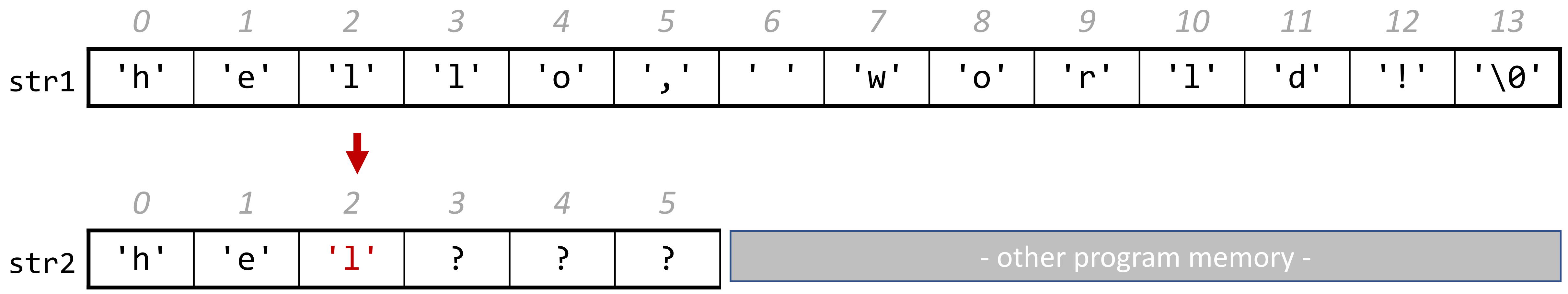
Copying Strings – Buffer Overflows

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



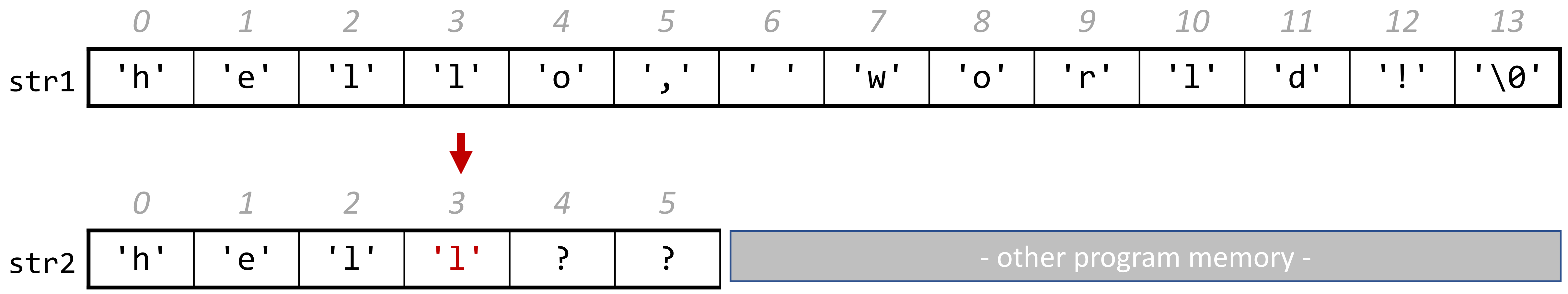
Copying Strings – Buffer Overflows

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



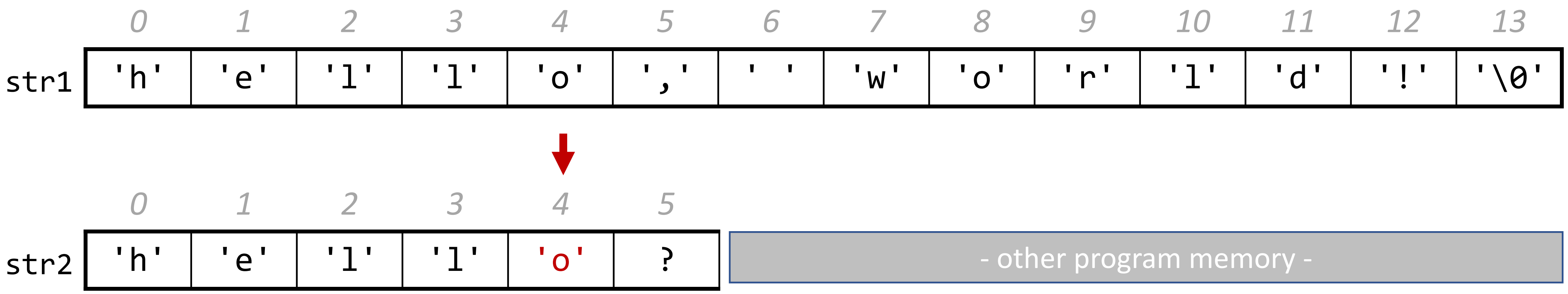
Copying Strings – Buffer Overflows

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



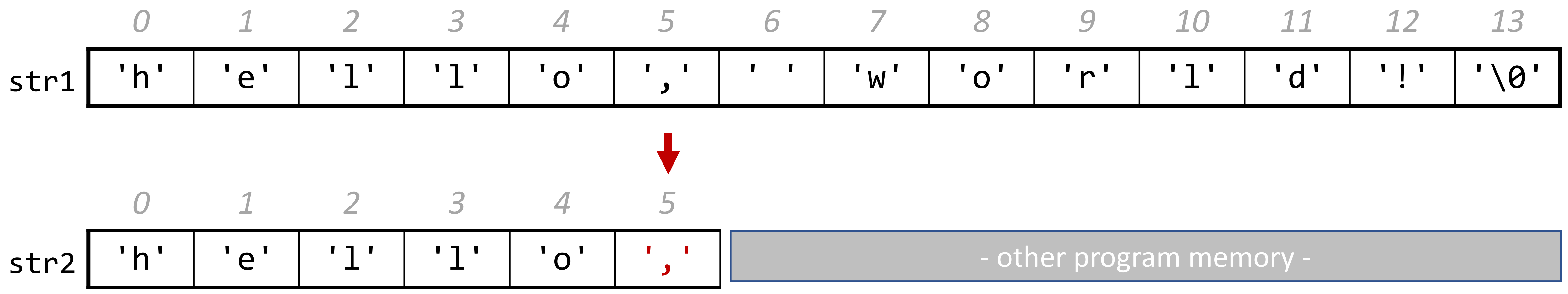
Copying Strings – Buffer Overflows

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



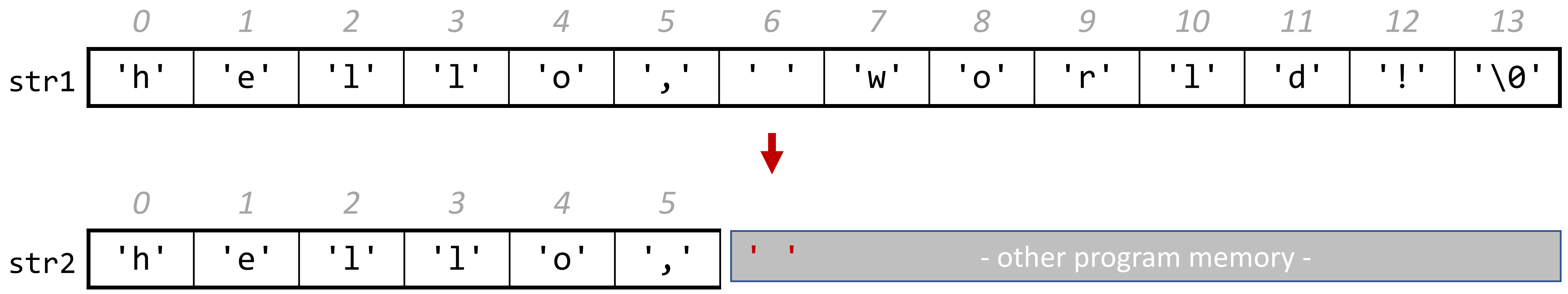
Copying Strings – Buffer Overflows

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



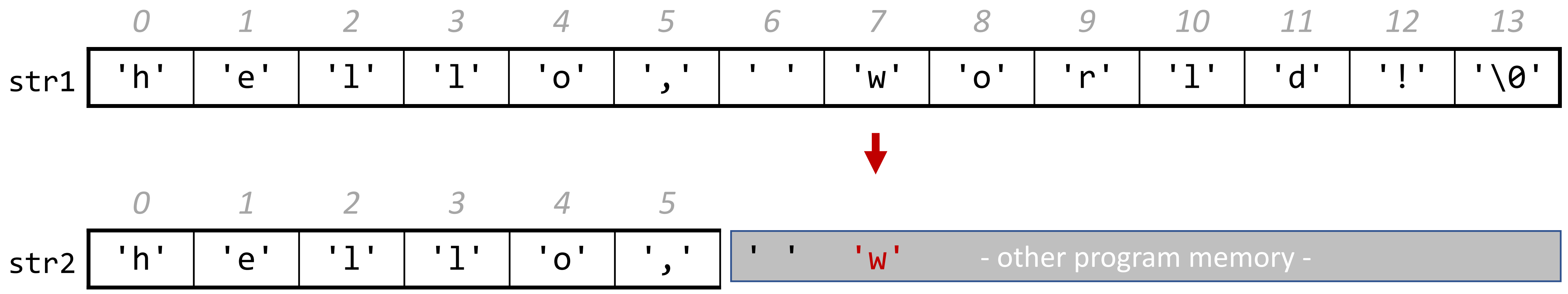
Copying Strings – Buffer Overflows

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



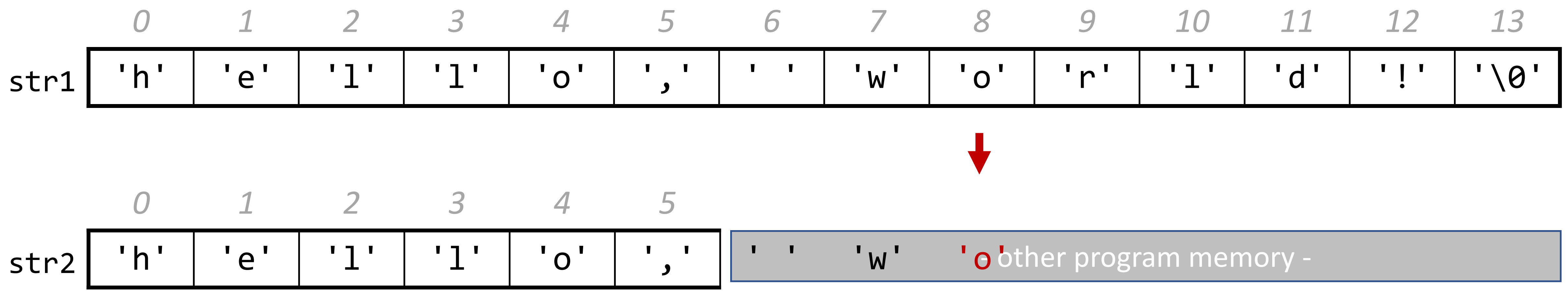
Copying Strings – Buffer Overflows

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



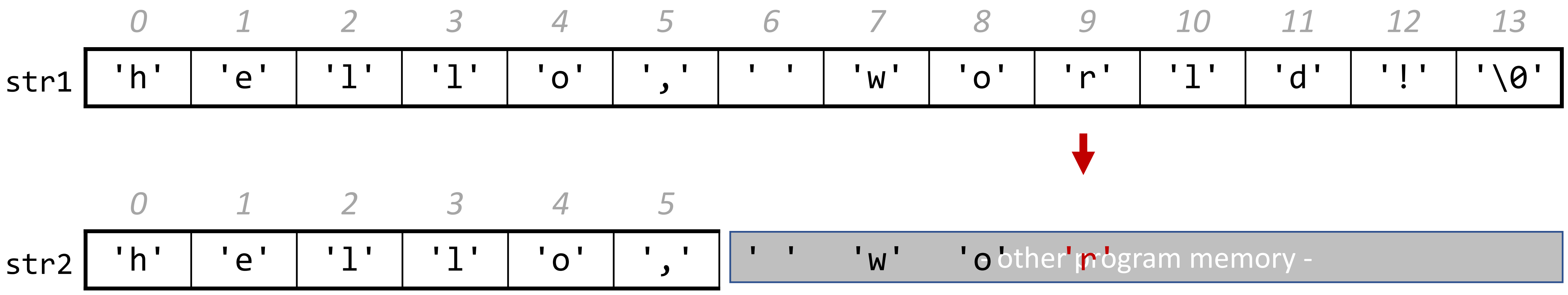
Copying Strings – Buffer Overflows

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



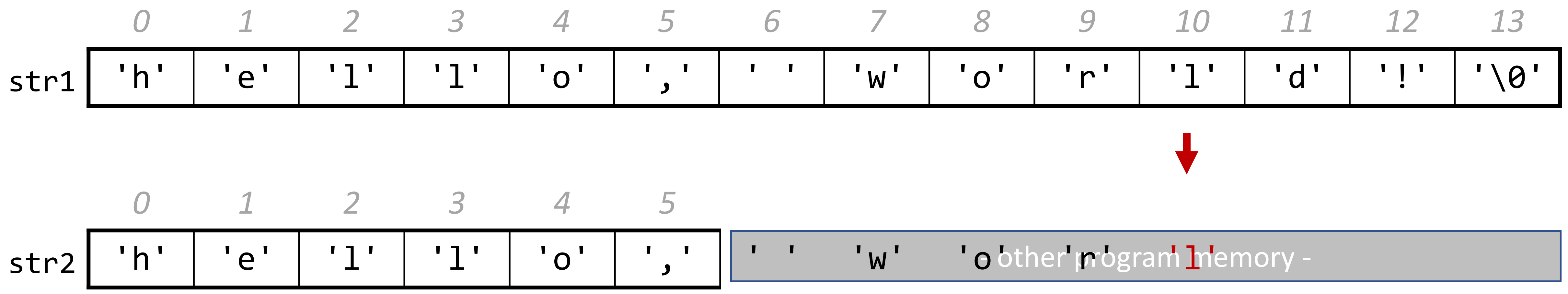
Copying Strings – Buffer Overflows

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



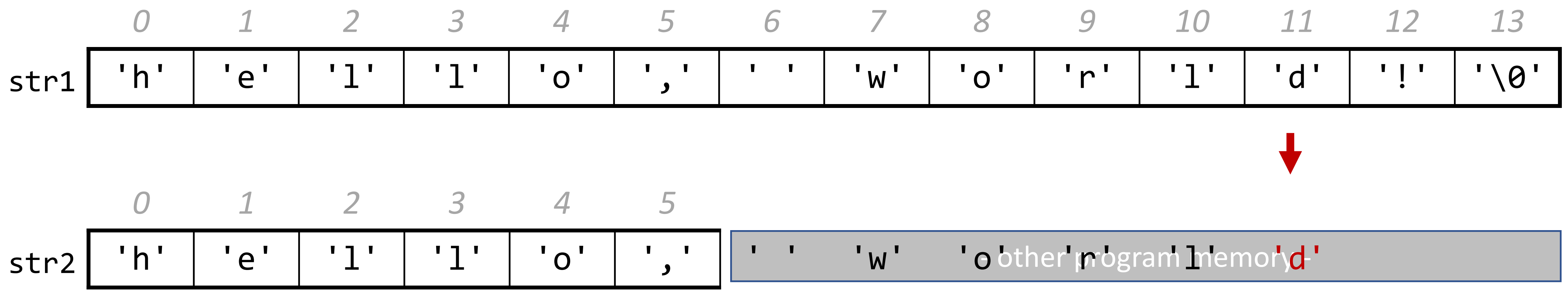
Copying Strings – Buffer Overflows

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



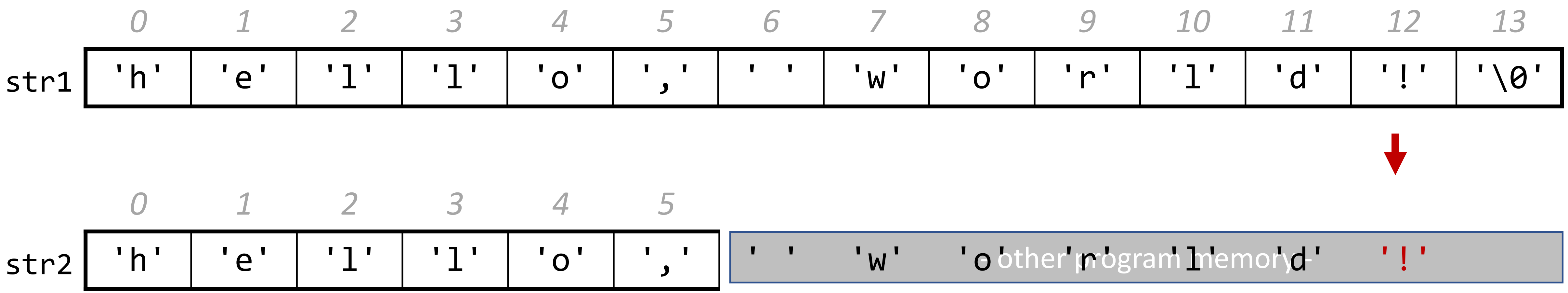
Copying Strings – Buffer Overflows

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



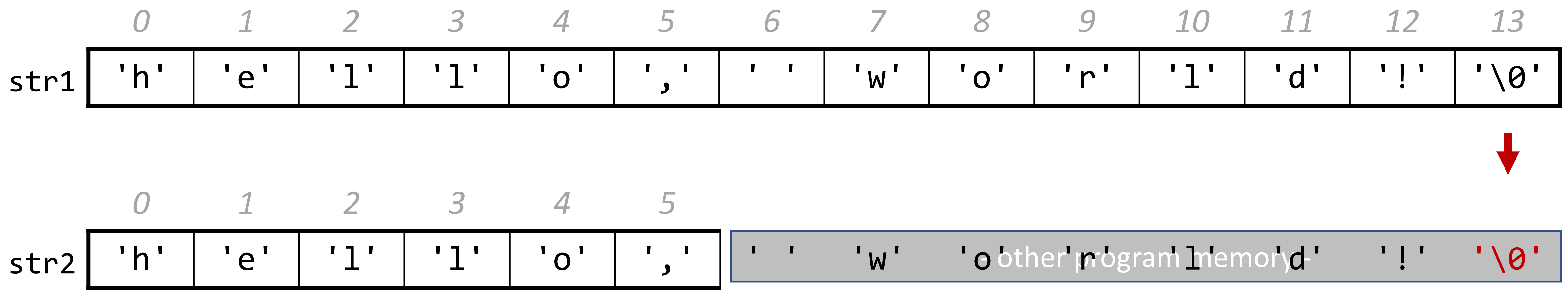
Copying Strings – Buffer Overflows

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



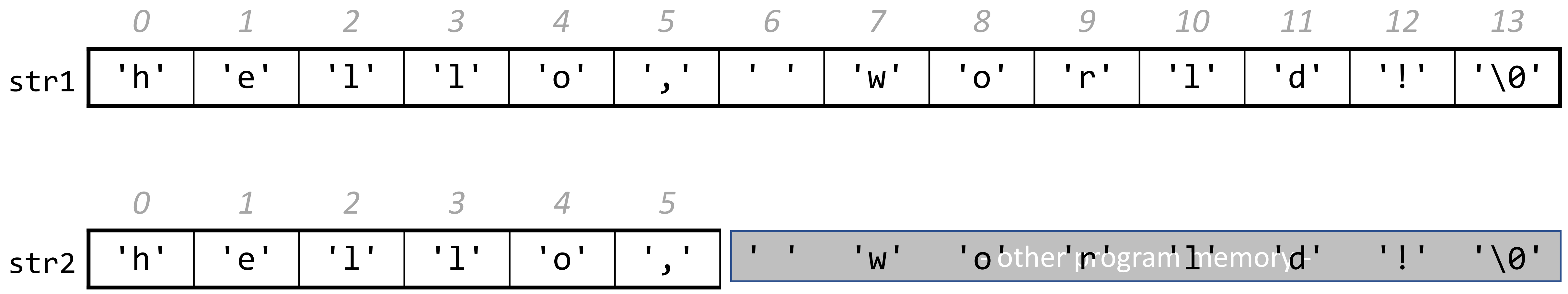
Copying Strings – Buffer Overflows

```
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`

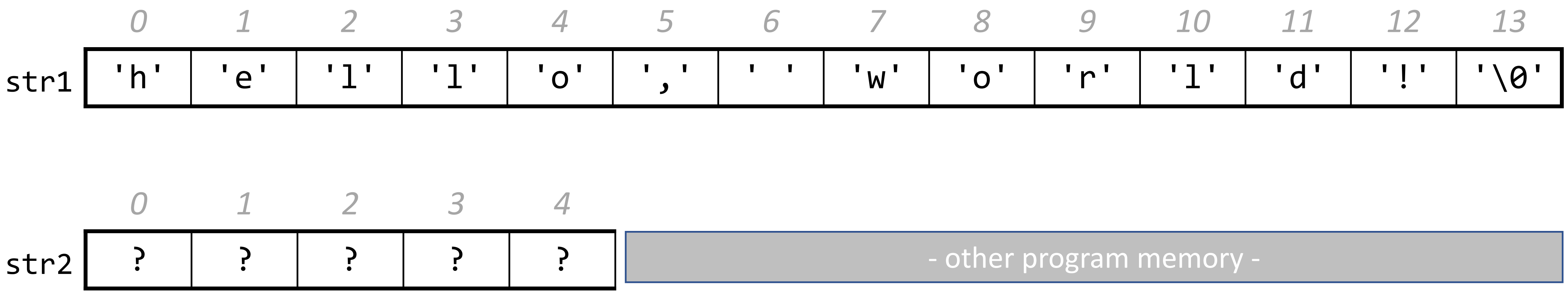
`strncpy(dst, src, n)`: copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **`dst`** will *not be null terminated*!

```
	// 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'`!

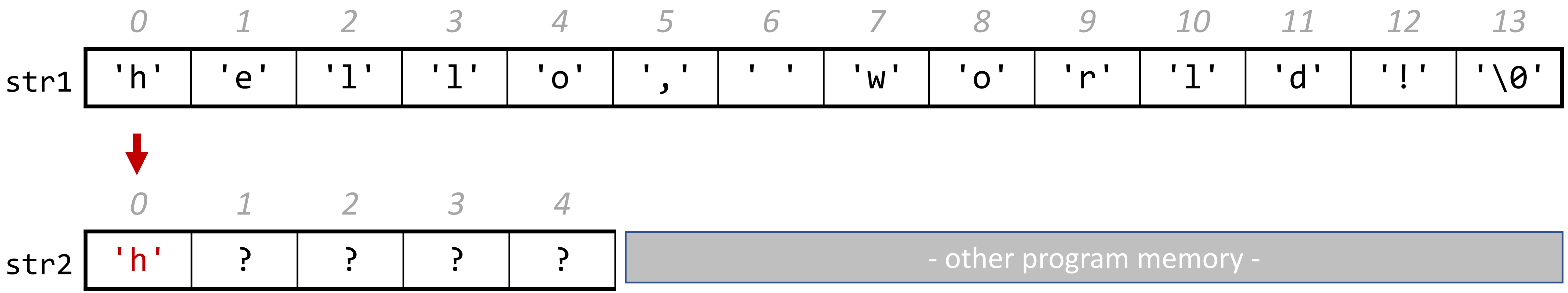
Copying Strings - strncpy

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



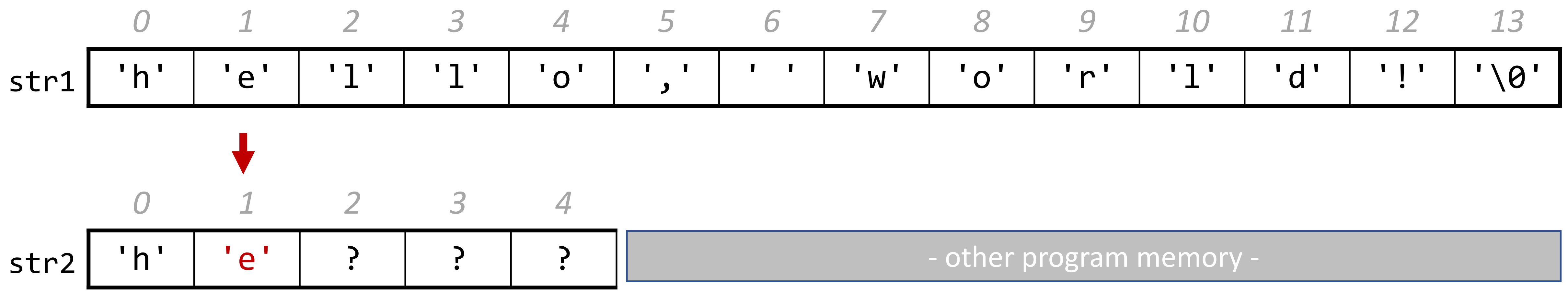
Copying Strings - strncpy

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



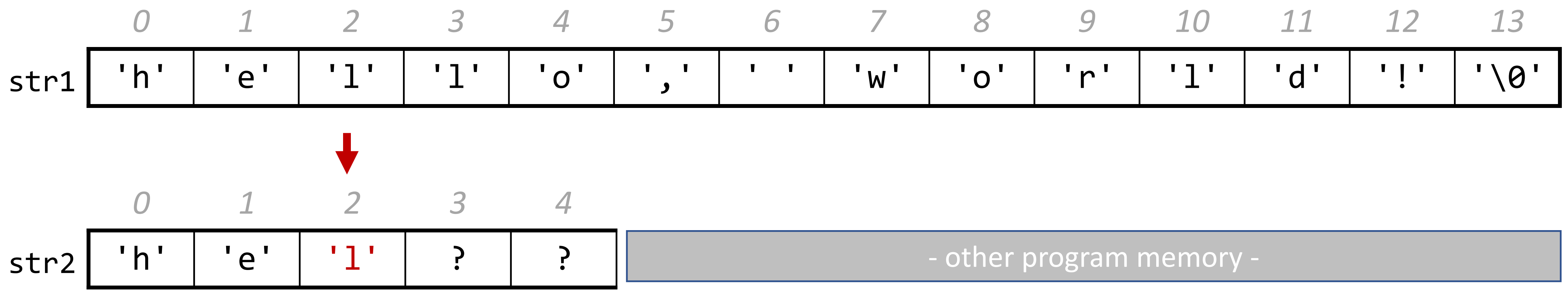
Copying Strings - strncpy

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



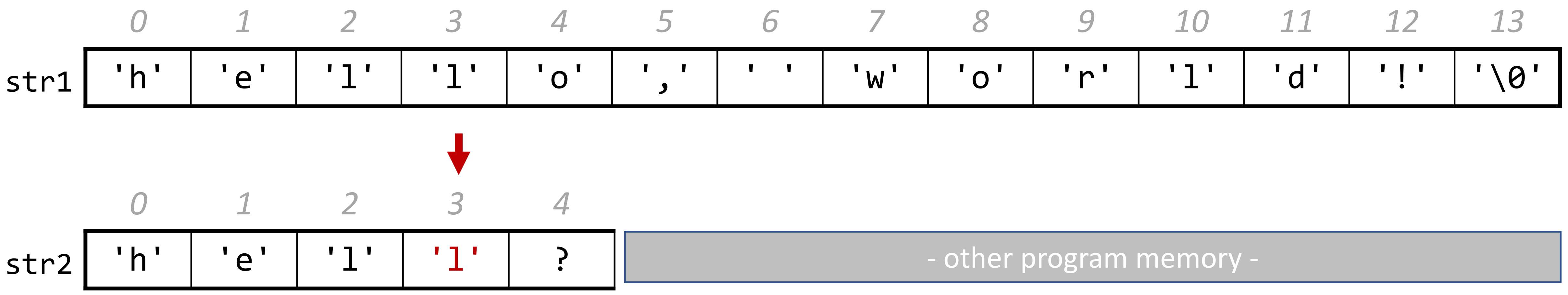
Copying Strings - strncpy

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



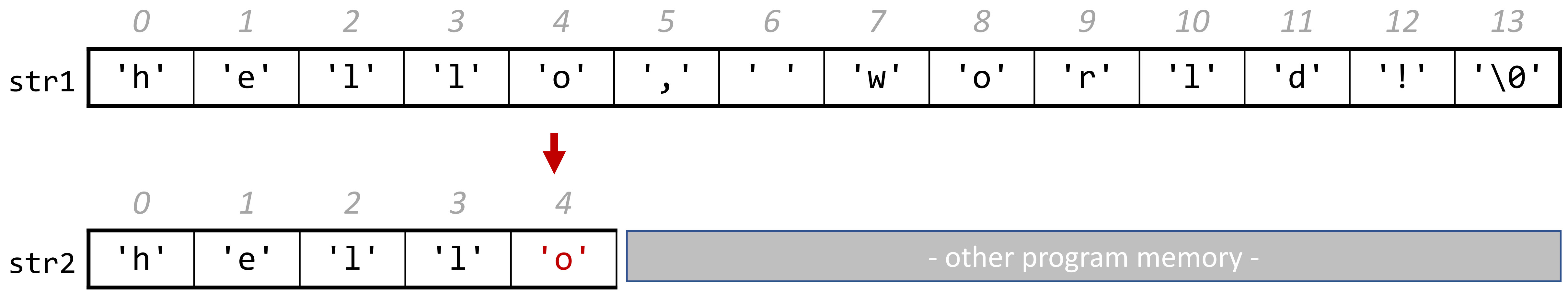
Copying Strings - strncpy

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



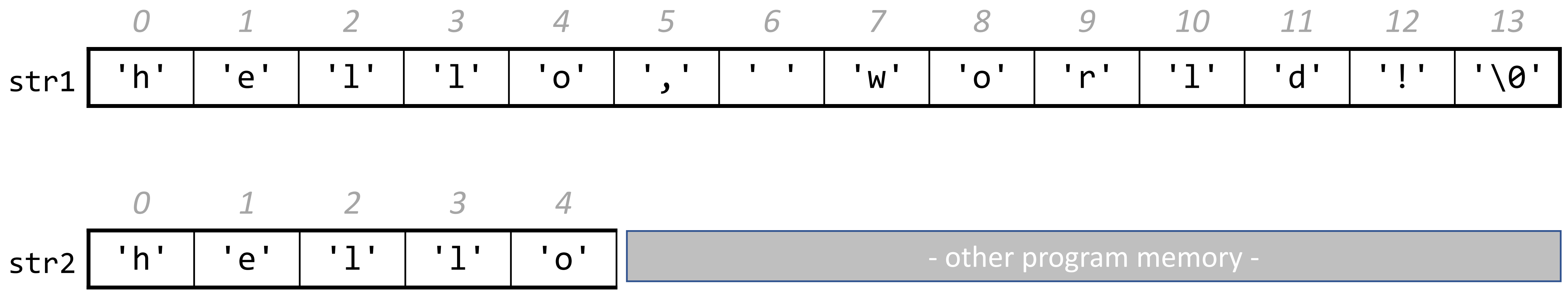
Copying Strings - strncpy

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



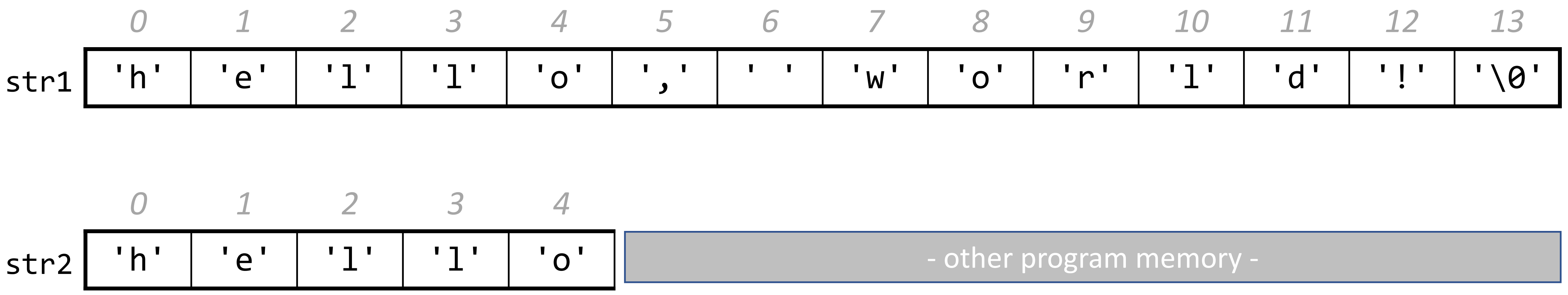
Copying Strings - strncpy

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



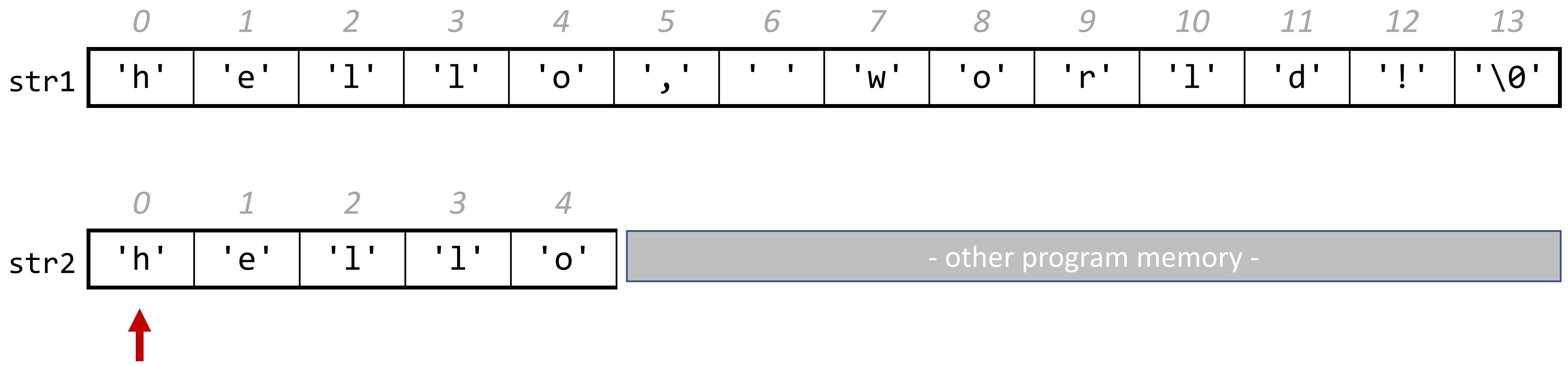
Copying Strings - strncpy

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



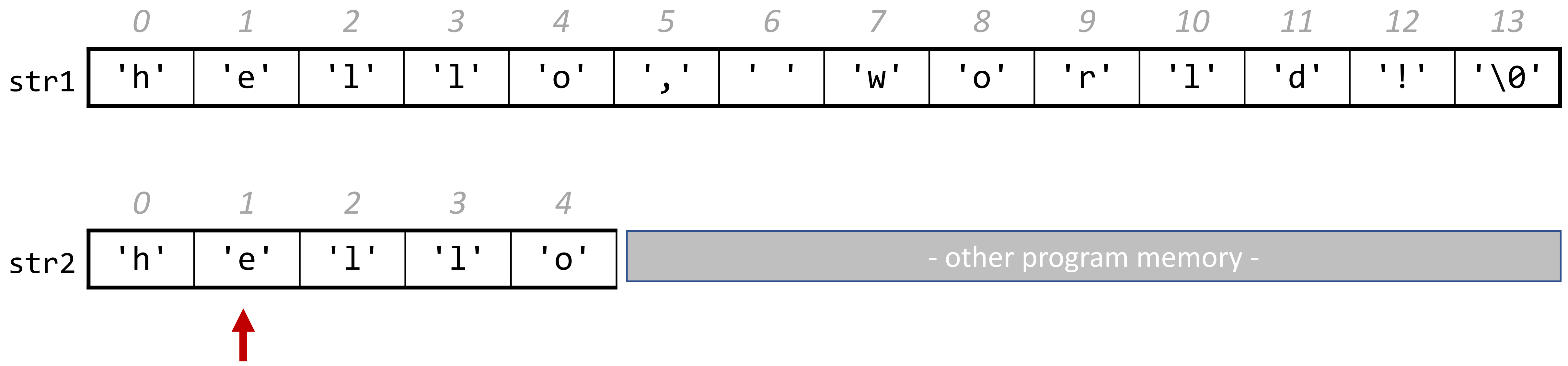
Copying Strings - strncpy

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



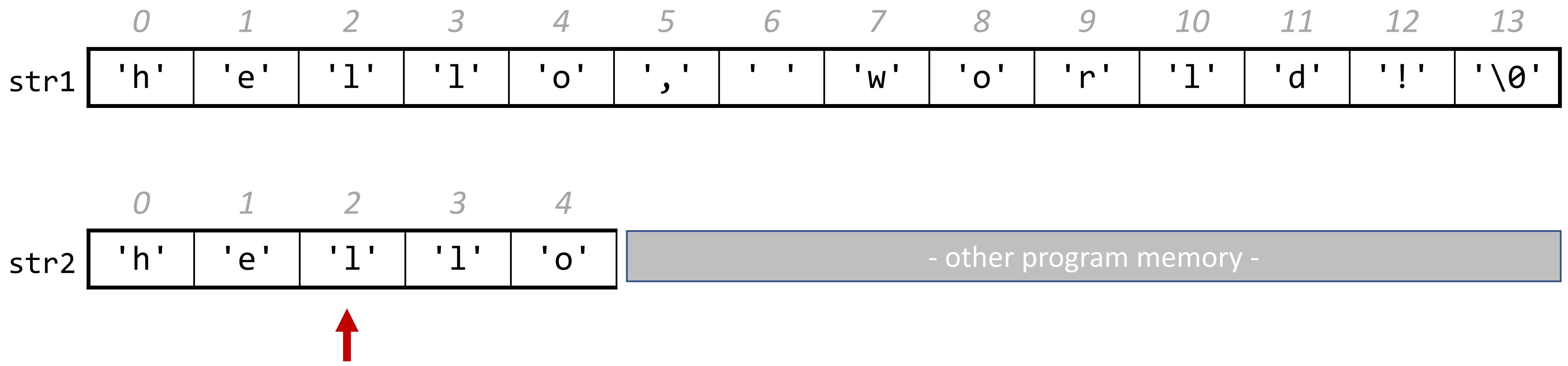
Copying Strings - strncpy

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



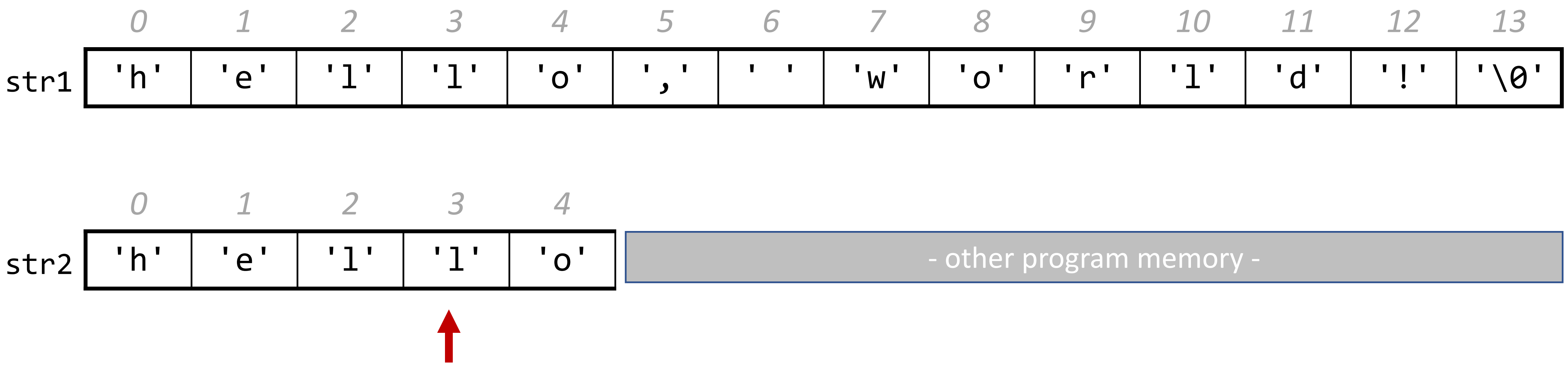
Copying Strings - strncpy

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



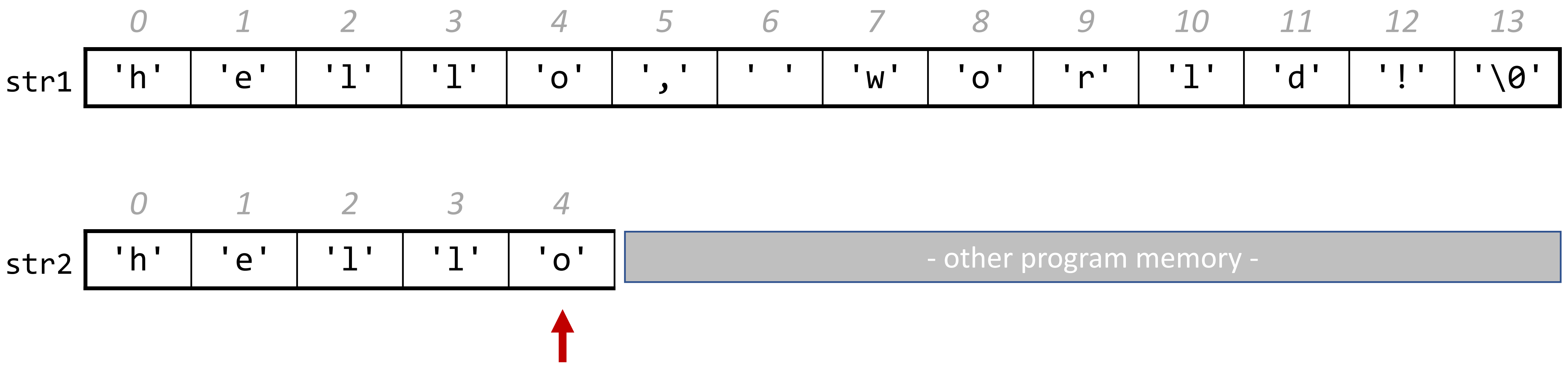
Copying Strings - strncpy

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



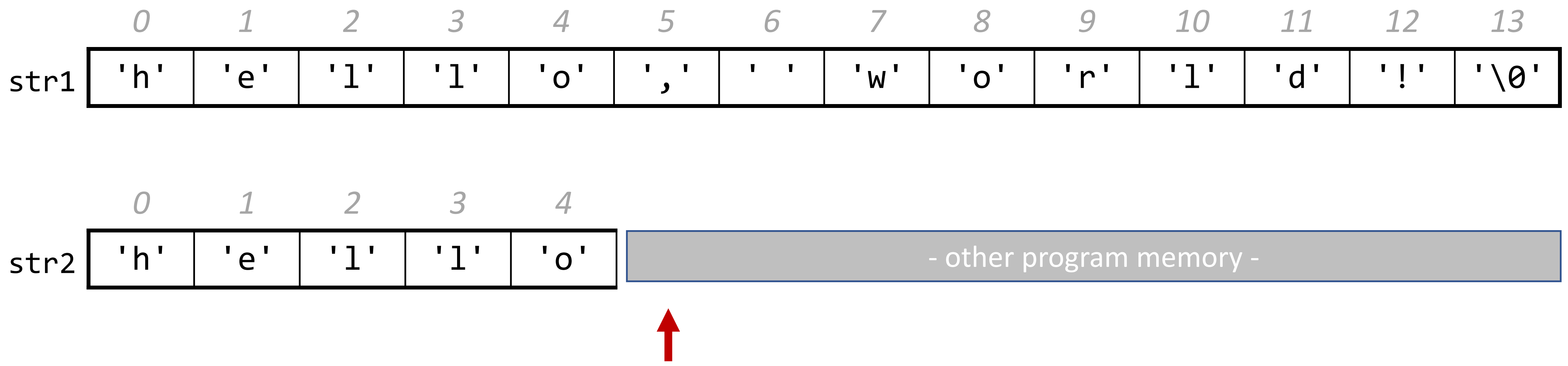
Copying Strings - strncpy

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



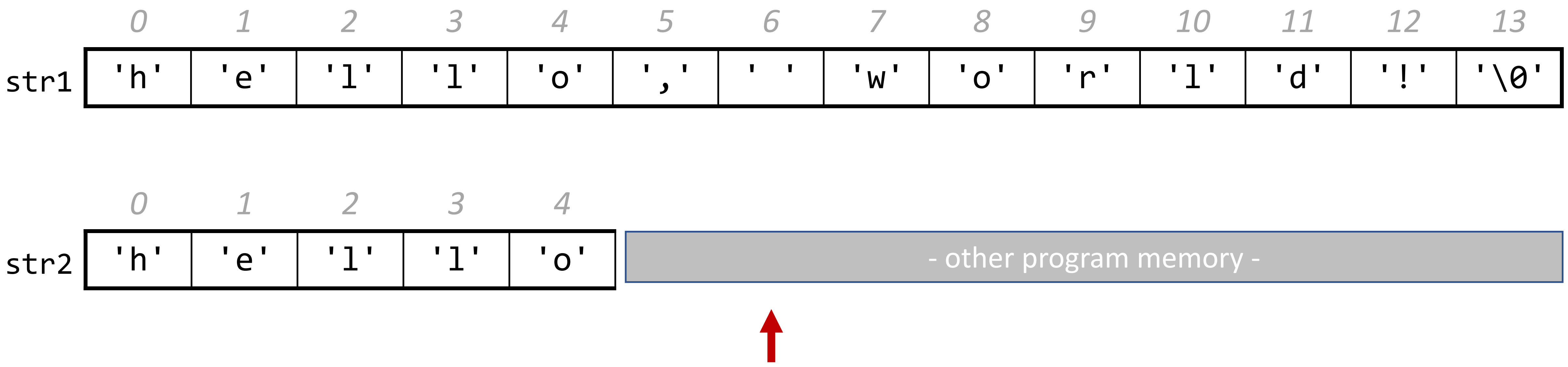
Copying Strings - strncpy

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



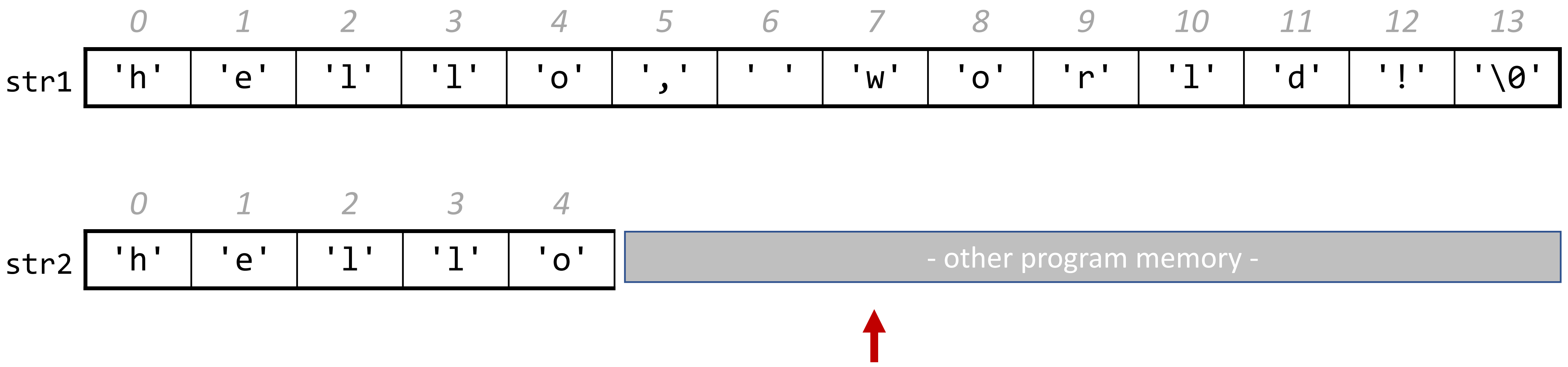
Copying Strings - strncpy

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



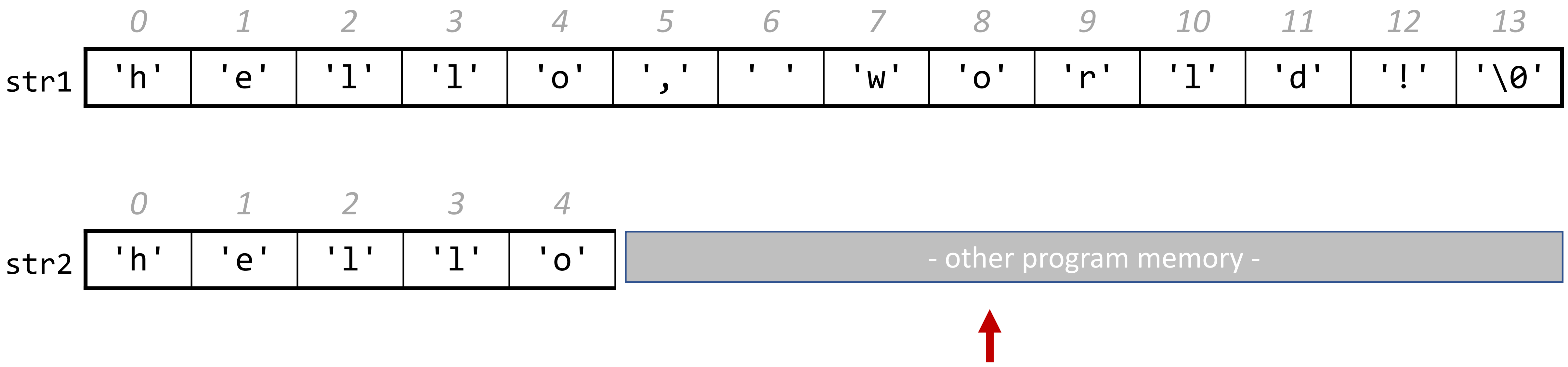
Copying Strings - strncpy

```
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);
```



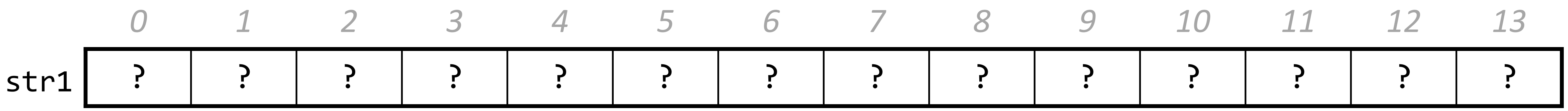
C Doesn’t Automatically Initialize

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

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

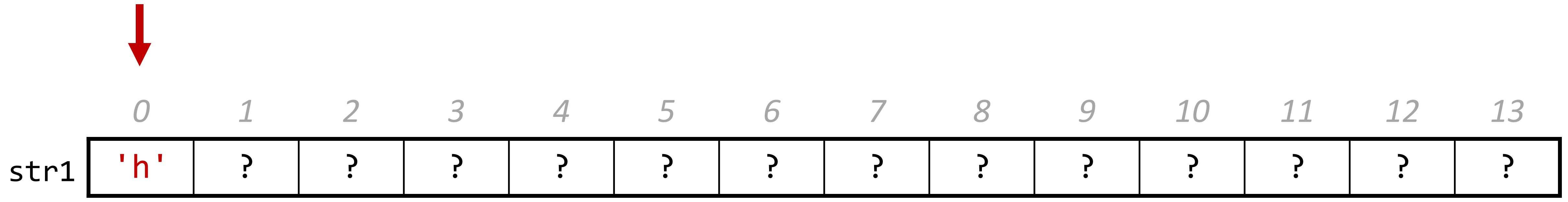
Copying Strings - `strncpy`

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



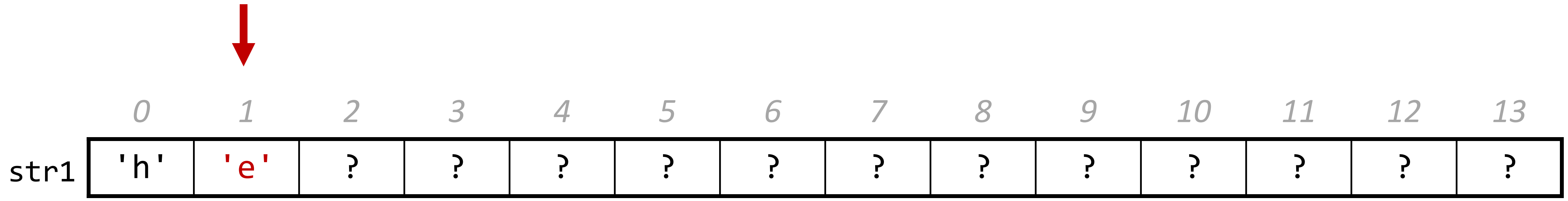
Copying Strings - strncpy

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



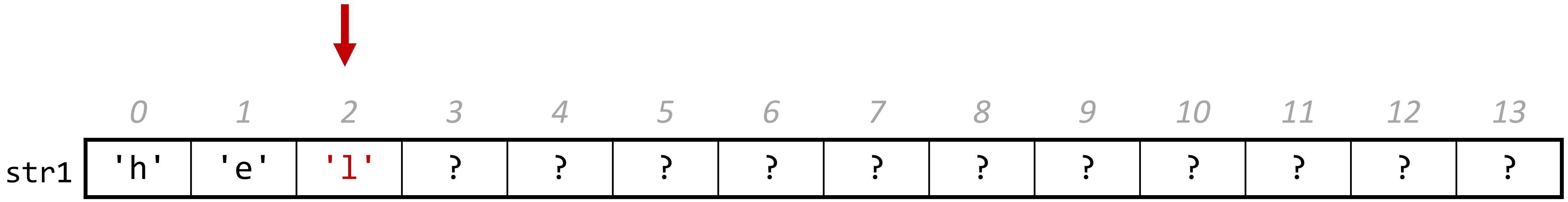
Copying Strings - strncpy

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



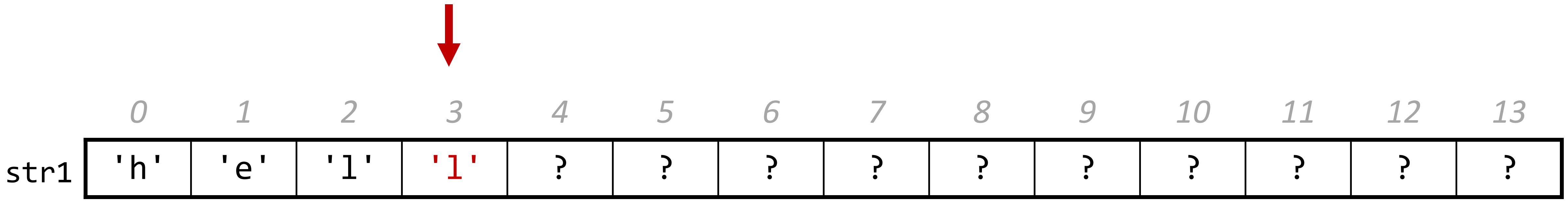
Copying Strings - strncpy

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



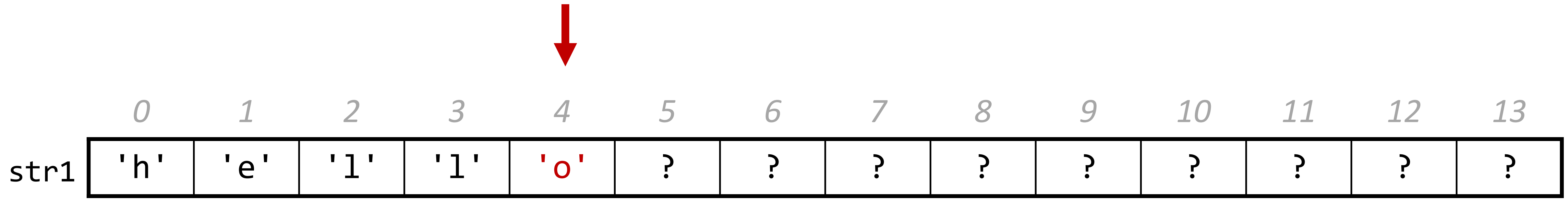
Copying Strings - strncpy

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



Copying Strings - strncpy

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



Copying Strings - strncpy

```c
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'	?	?	?	?	?	?	?	?	?

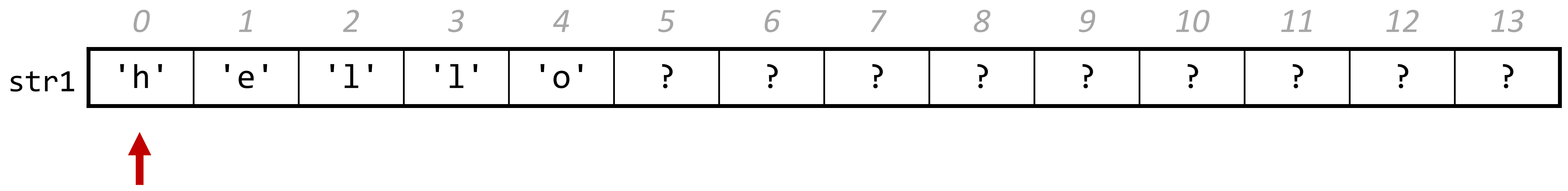
Copying Strings - strncpy

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

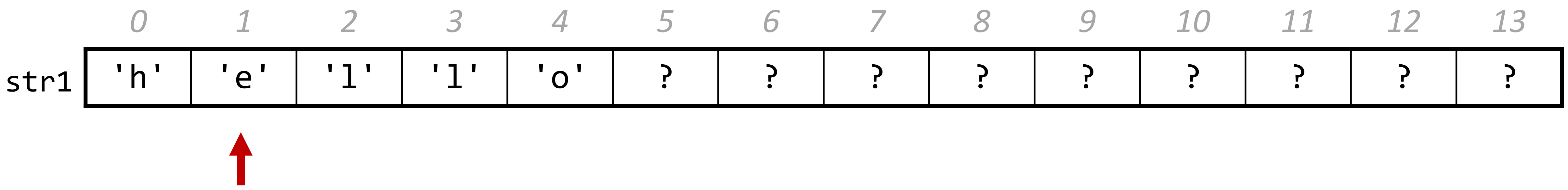
Copying Strings - strncpy

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



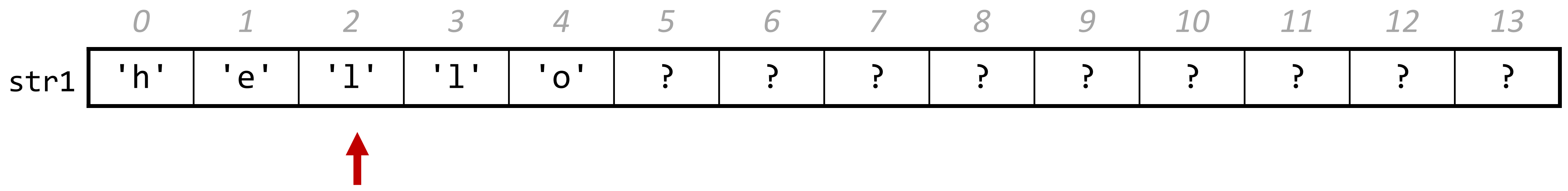
Copying Strings - strncpy

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



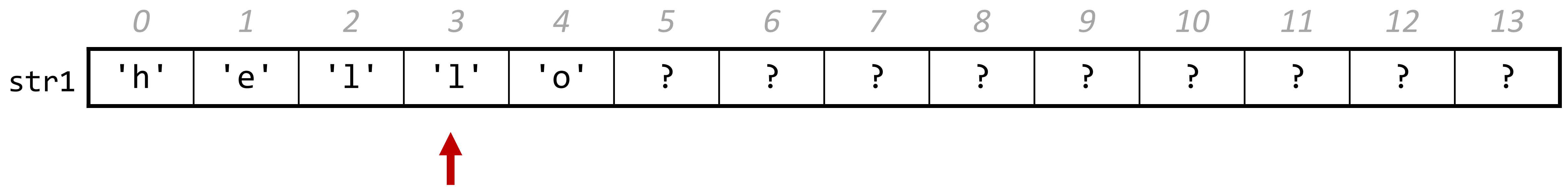
Copying Strings - strncpy

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



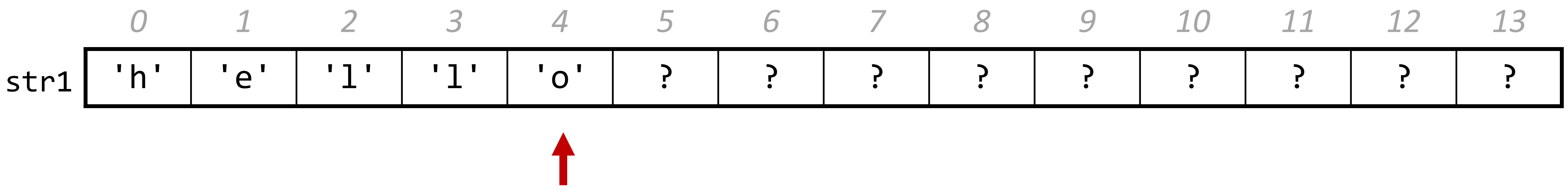
Copying Strings - strncpy

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



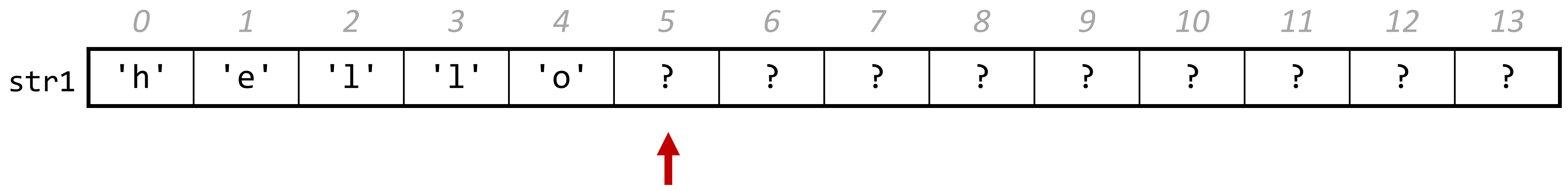
Copying Strings - strncpy

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



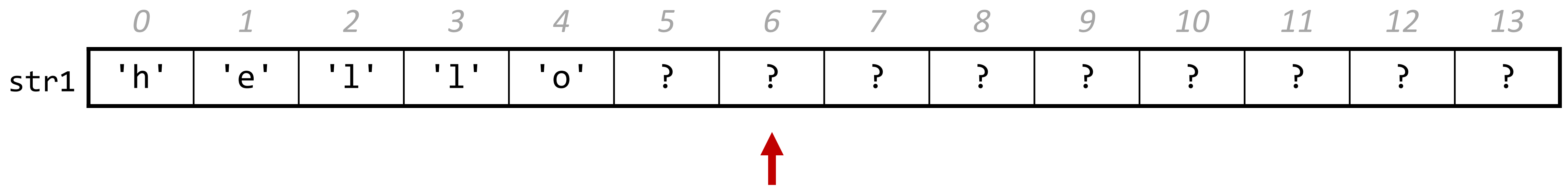
Copying Strings - strncpy

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



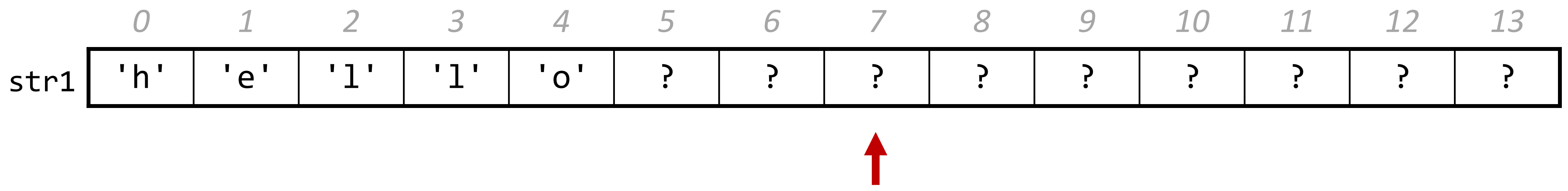
Copying Strings - strncpy

```
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?

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

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 <u>cannot</u> 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

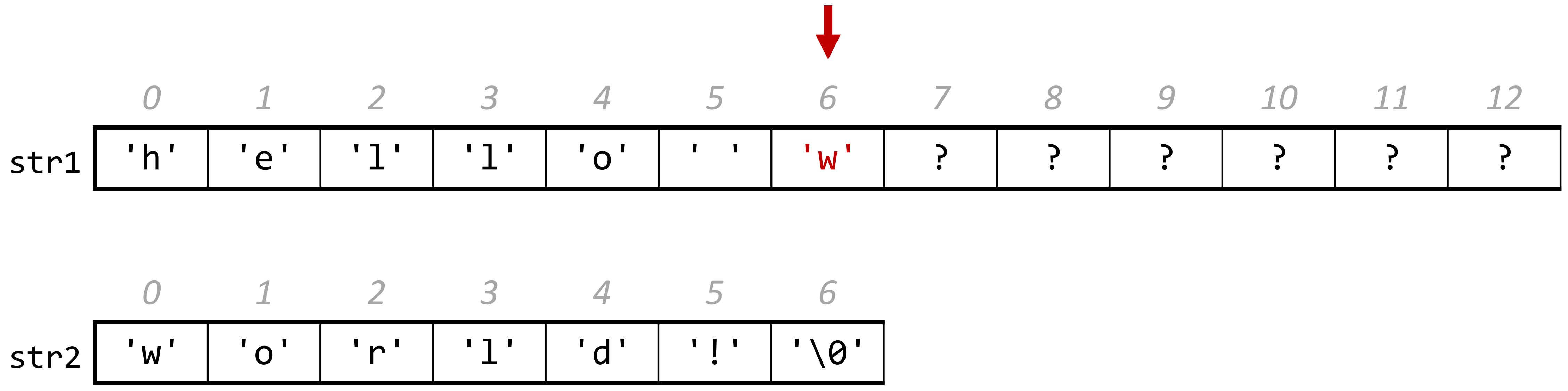
```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

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'

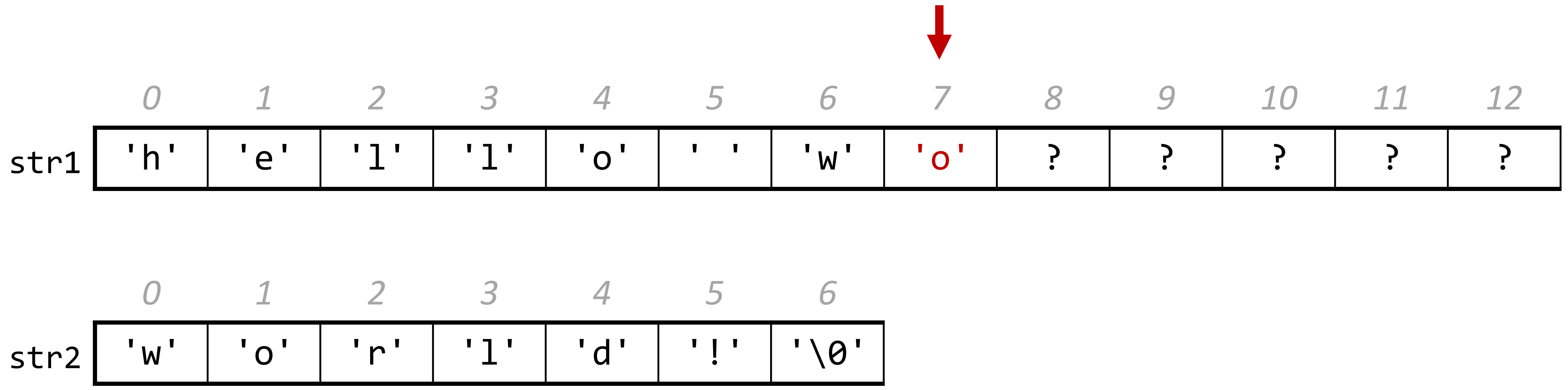
Concatenating Strings

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

strcat(str1, str2);
```



Concatenating Strings

```c
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'	?	?	?	?	?

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

Concatenating Strings

```c
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'	?	?	?	?

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

Concatenating Strings

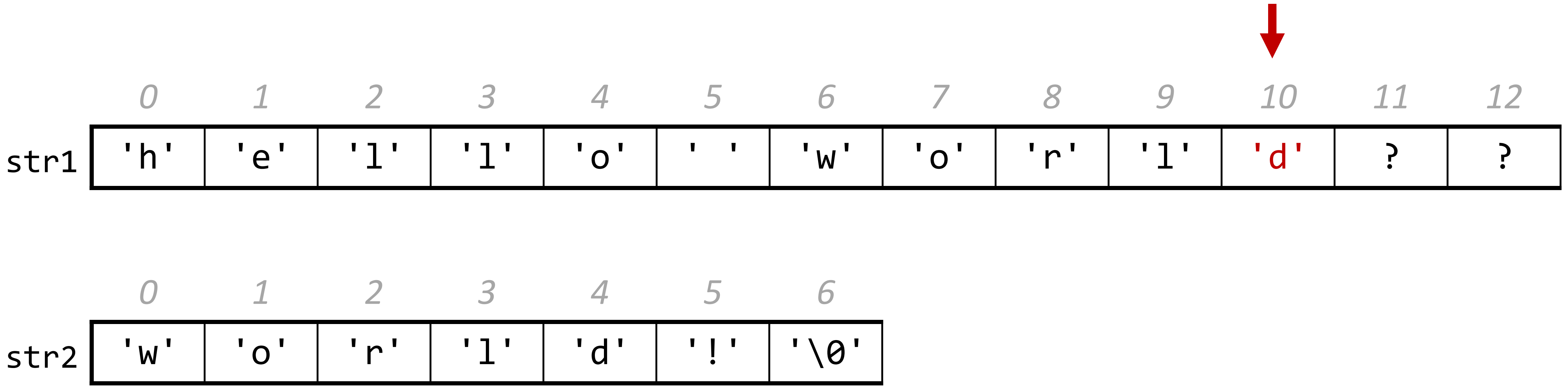
```c
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

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'	?	?	?

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

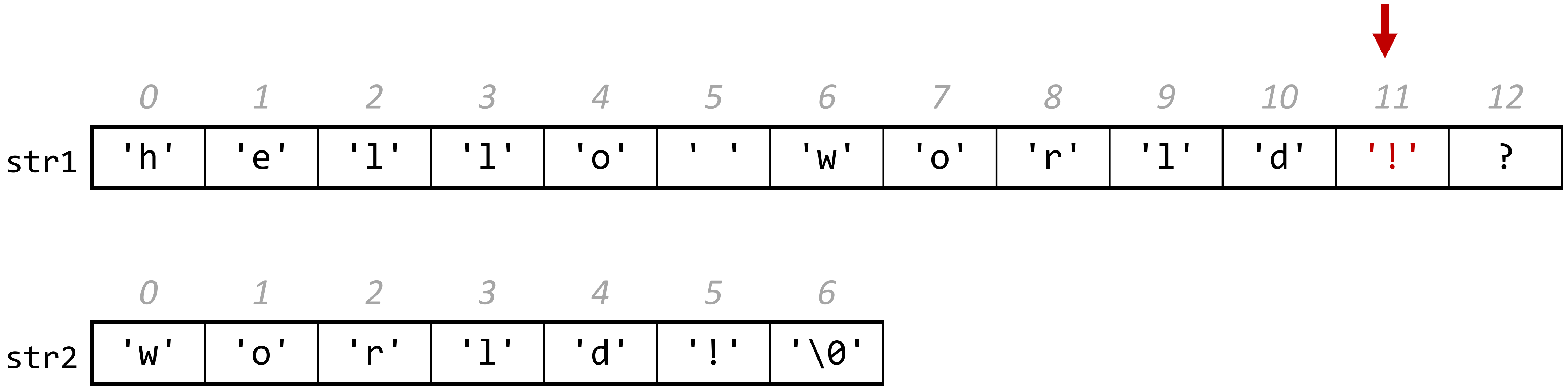
Concatenating Strings

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

strcat(str1, str2);
```



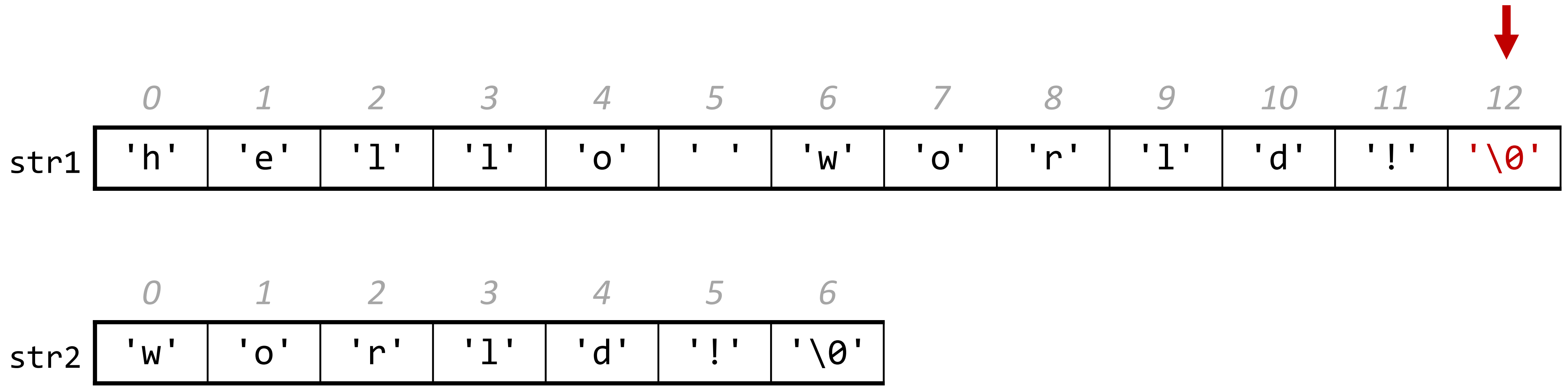
Concatenating Strings

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

strcat(str1, str2);
```



Concatenating Strings

```c
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'

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 *

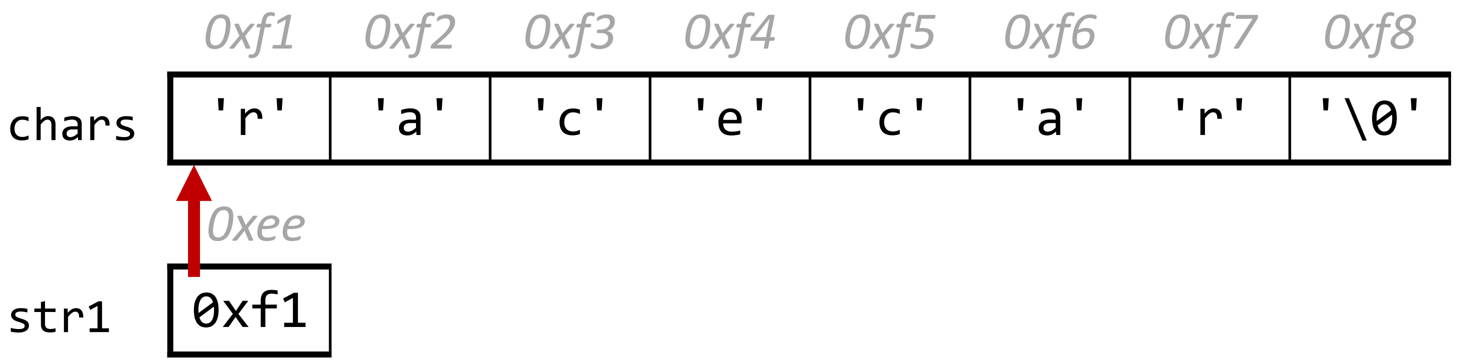
You can also create a char * variable yourself that points to an address within in an existing string.

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

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

Substrings

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

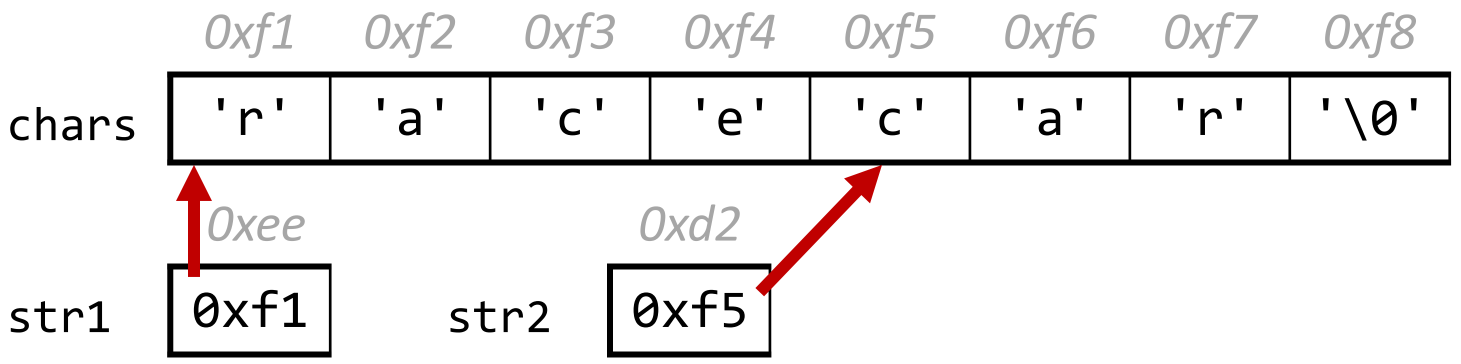
```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
```



Substrings

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

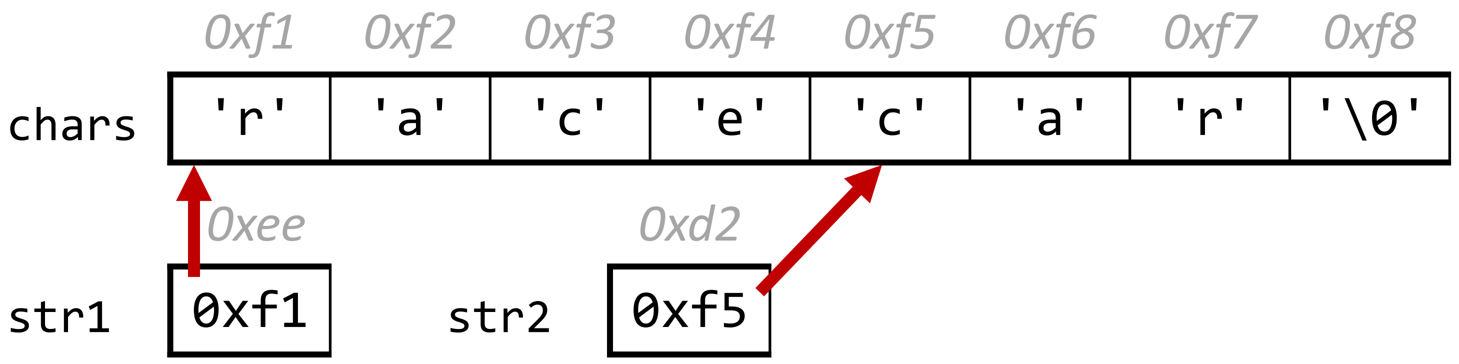
```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
```



Substrings

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

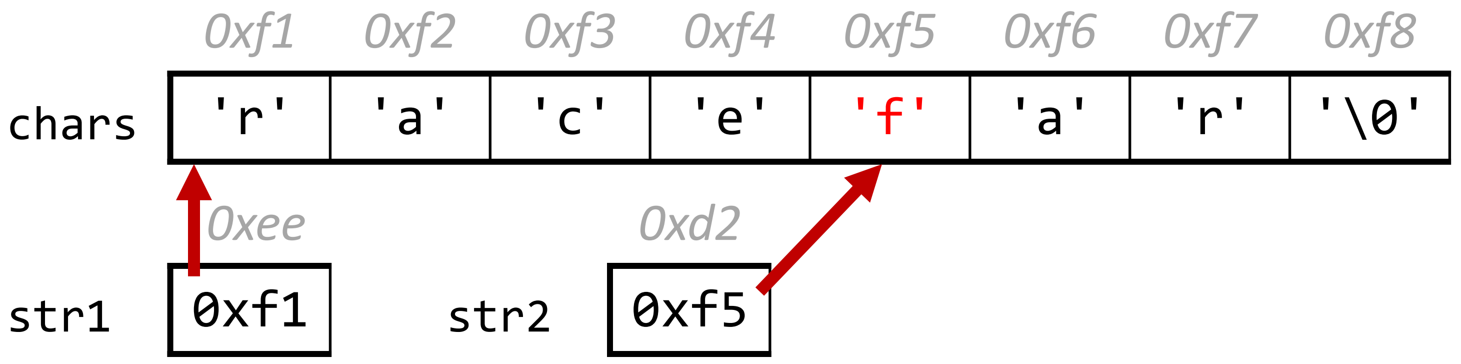
```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // car
```



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!

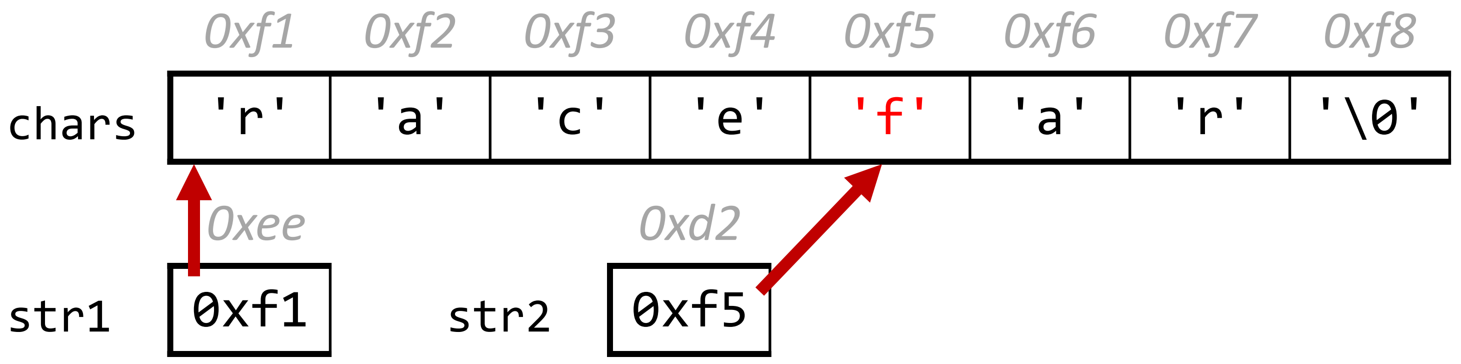
```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1);
printf("%s\n", str2);
```



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
```

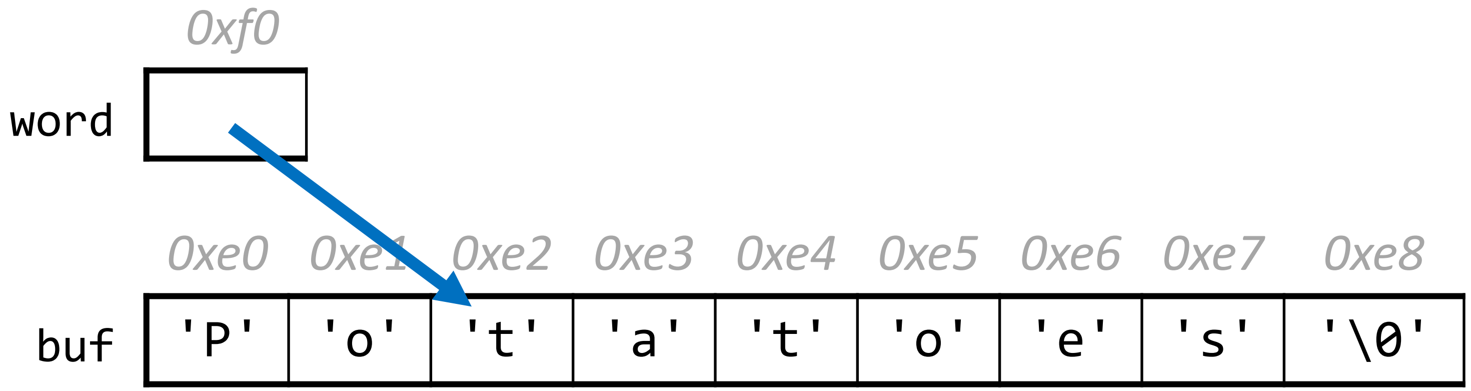


Strings Practice

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

Line 6: What is printed?

A. matoes
B. mattoes
C. Pomat
[✓ D. Pomatoes]
E. Something else
F. Compile error



char * vs. char[]

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

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
```

Recap

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

Lecture 4 takeaway: C strings are null-terminated arrays of characters. We can manipulate them using string and pointer operations.

Next time: more strings

Extra Practice

Copycat exercise

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

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

strcat(dst, src);
```

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

Copycat exercise

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

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

```c
strcat(dst, src);
```

$\xleftrightarrow{\text{equivalent}}$

```c
strcpy(dst + strlen(dst), src);
```