CS107, Lecture 4
C Strings

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

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans
- **Practice:** Password Verification

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

```
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(`***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** the new char, they cannot modify an existing char!

Common ctype.h Functions

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

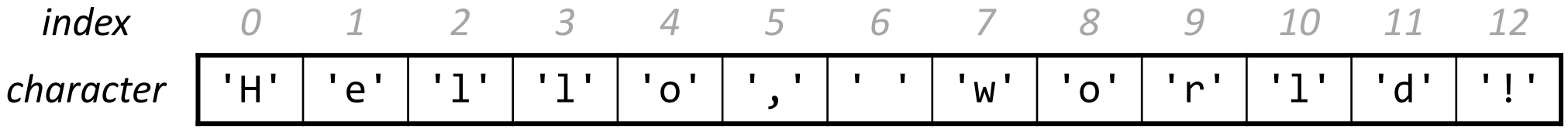
Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

C Strings

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

```
char text[] = "Hello, world!";
```



- Each character is assigned an *index*, going from 0 to length-1
- There is a **`char`** at each index

Creating Strings

```
char myString[] = "Hello, world!";
char empty[] = "";

myString[0] = 'h';
printf("%s", myString);              // hello, world!

char stringToFillIn[30];
stringToFillIn[0] = 'a';
...
```

Creating Strings

If you do not need to modify the string later, you can use this shorthand:

```
char *myString = "Hello, world!";
char *empty = "";

myString[0] = 'h';                  // crashes!
printf("%s", myString);             // Hello, world!
```

C Strings

```
char myString[]
```

vs

```
char *myString
```

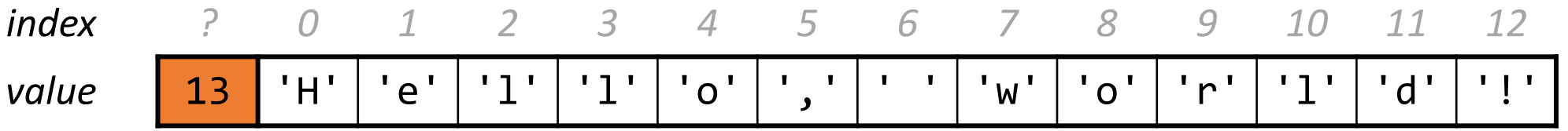
Both are essentially pointers to the first character in the string. However, you **cannot** reassign an array after you create it. You **can** reassign a pointer after you create it.

```
char myString[] = "Hello, world!";
myString = "Another string";                    // not allowed!
---
char *myString = "Hello, world!";
myString = "Another string";                    // ok
```

String Length

C strings are just arrays of characters. How do we determine string length?

Option 1: reserve the first byte to store the length

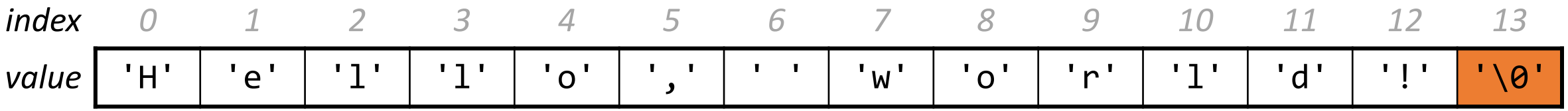


Pros	Cons
• Can get length in O(1) time!	• Length is limited by size of 1 byte, and longer lengths need more bytes. • Must compensate for indices (index 0 is length)

String Length

C strings are just arrays of characters. How do we determine string length?

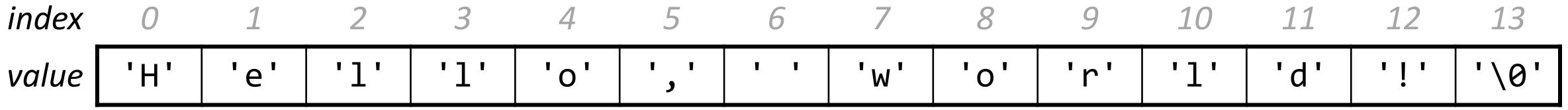
Option 2: terminate every string with a '\0' character.



Pros	Cons
• Always uses exactly 1 extra byte. • Doesn't change indices of other characters.	• Requires traversing the string to calculate its length.

String Length

C strings use Option 2 – they are arrays of characters, ending with a **null-terminating character** '\0'.



Use the provided `strlen` function to calculate string length. The null-terminating character does *not* count towards the length.

```
char *myStr = "Hello, world!";
int length = strlen(myStr); // 13
```

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

C Strings As Parameters

Regardless of how you created the string, when you pass a string as a parameter it is always passed as a **char *.**

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

char *myString = "Hello";
doSomething(myString);
```

C Strings As Parameters

Regardless of how you created the string, when you pass a string as a parameter it is always passed as a **char *.**

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

char myString[] = "Hello";
doSomething(myString);
```

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

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`***.

Comparing Strings

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

```
char *str1 = "hello";   // e.g. 0x7f42
char *str2 = "world";   // e.g. 0x654d
if (str1 > str2) { …    // compares 0x7f42 > 0x654d!
```

Instead, use `strcmp`:

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

Copying Strings

You cannot copy C strings using =. This copies character addresses!

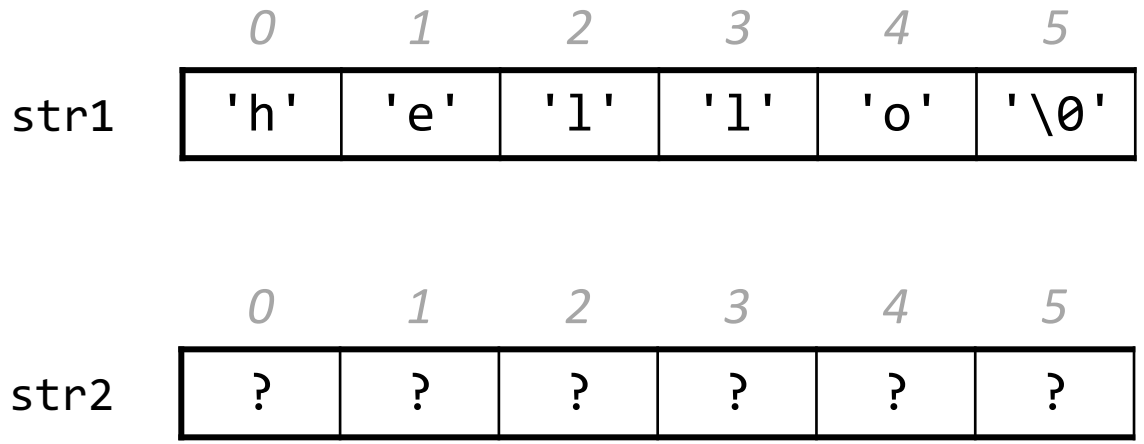
```
char str1[] = "hello";  // e.g. 0x7f42
char *str2 = str1;      // 0x7f42.  Points to same string!
str2[0] = 'H';
printf("%s", str1);     // Hello
printf("%s", str2);     // Hello
```

Instead, use `strcpy`:

```
char str1[] = "hello";  // e.g. 0x7f42
char str2[6];
strcpy(str2, str1);
str2[0] = 'H';
printf("%s", str1);     // hello
printf("%s", str2);     // Hello
```

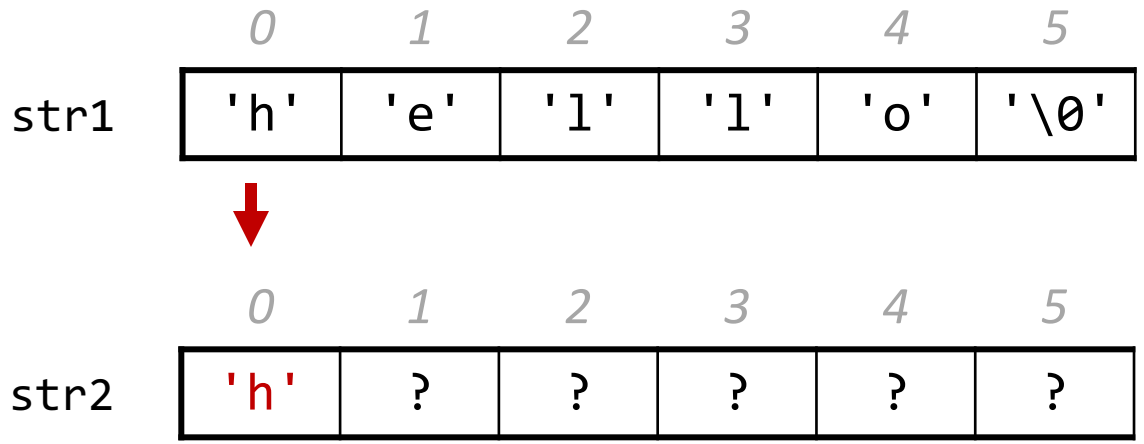
Copying Strings

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



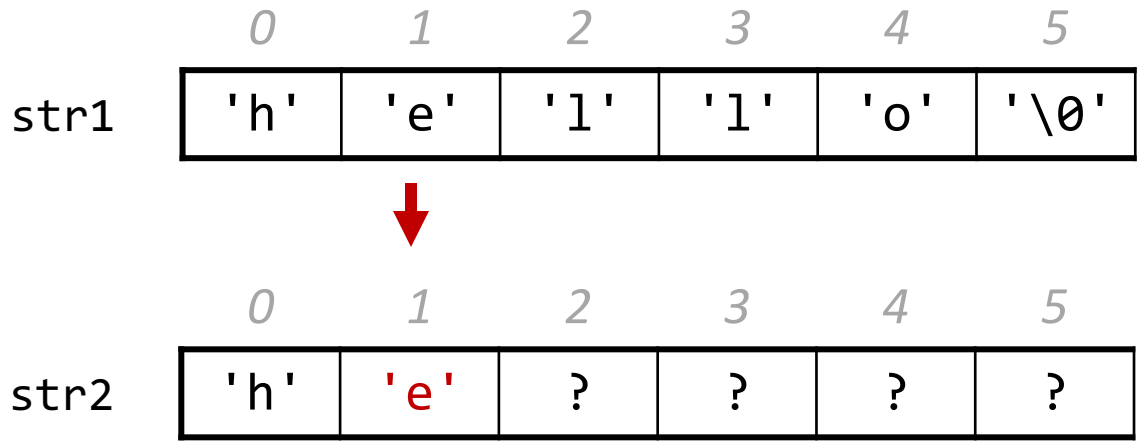
Copying Strings

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



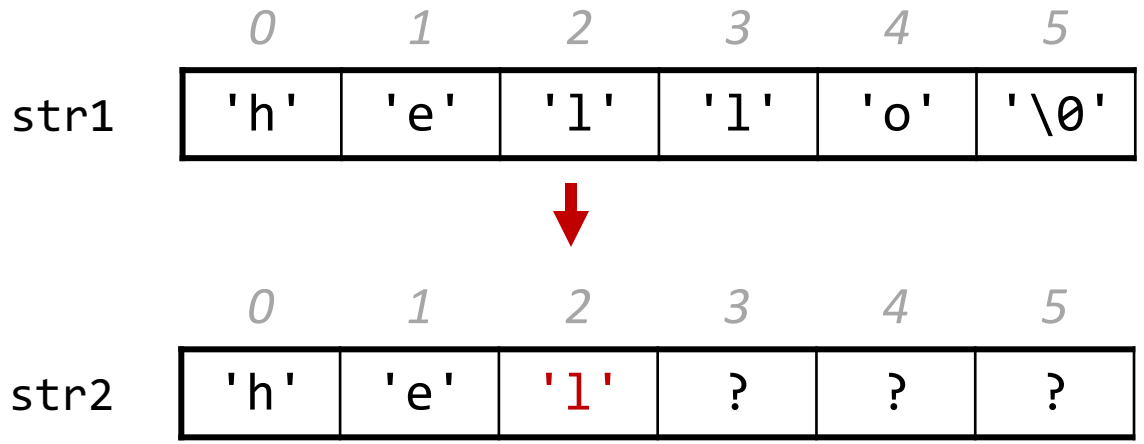
Copying Strings

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



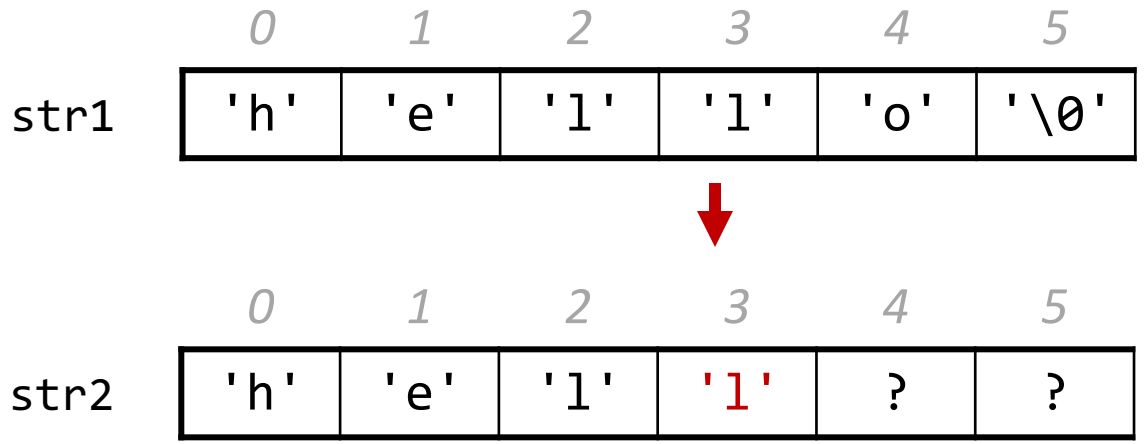
Copying Strings

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



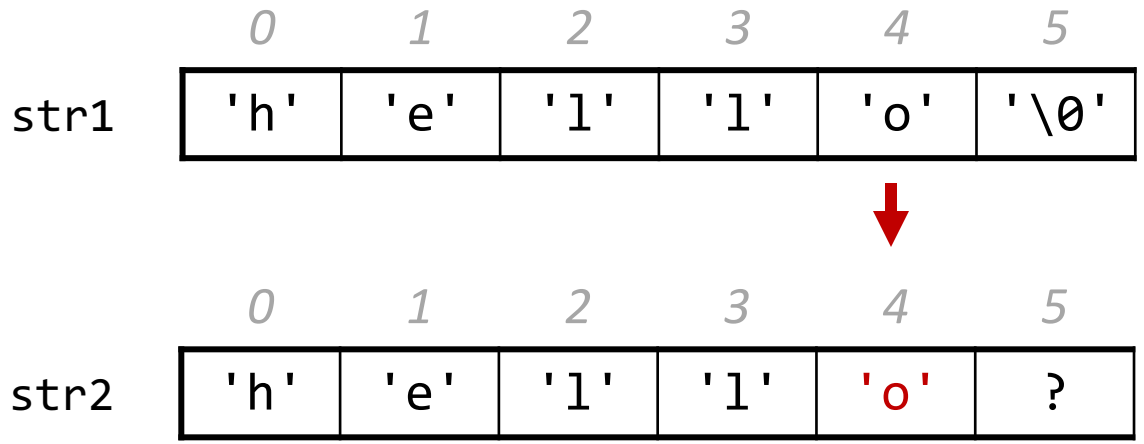
Copying Strings

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



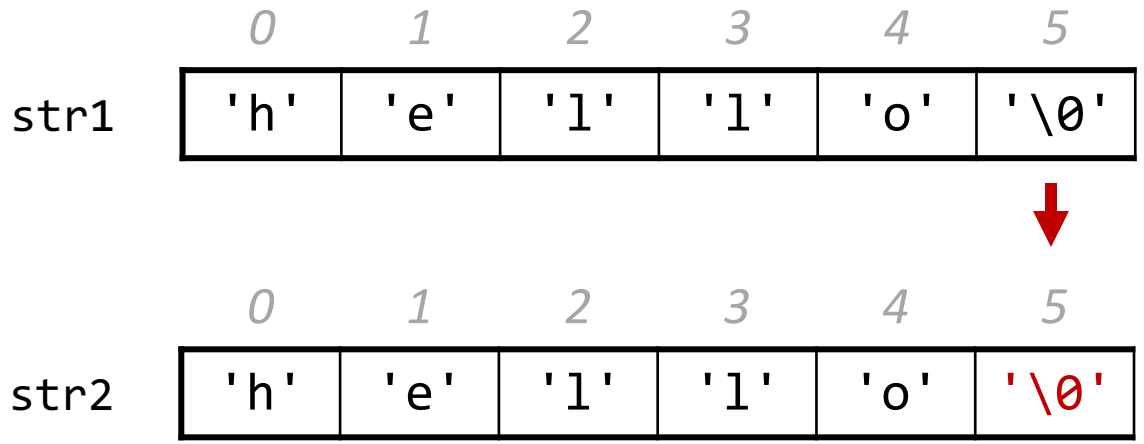
Copying Strings

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



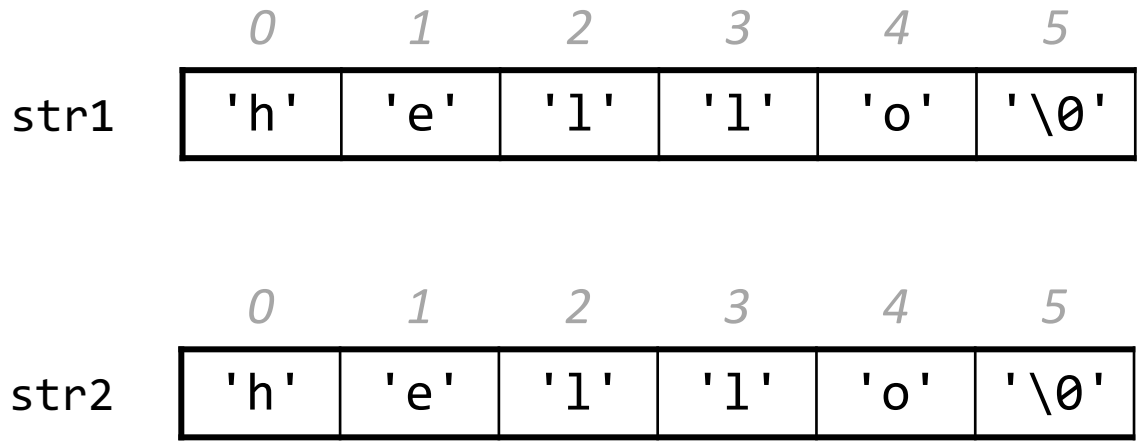
Copying Strings

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



Copying Strings

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



Copying Strings

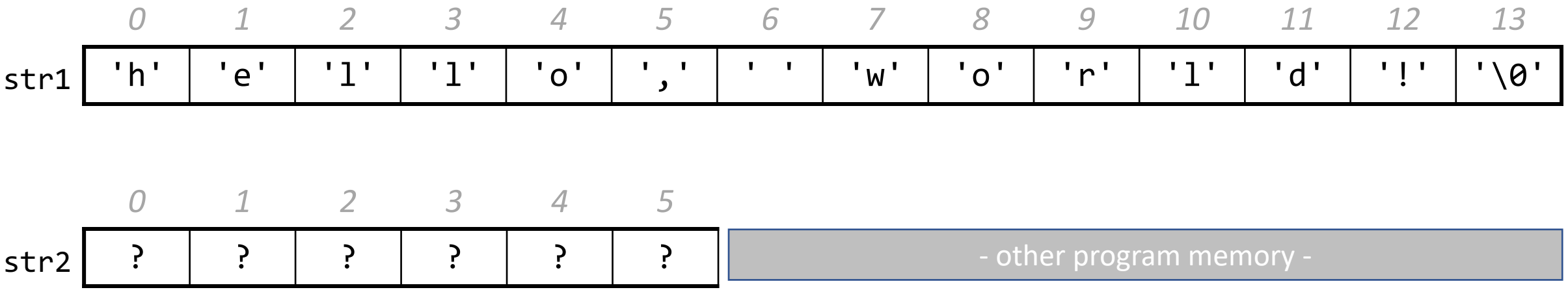
You are responsible for ensuring there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

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

Writing past your memory bounds is called a “buffer overflow”. It can allow for security vulnerabilities!

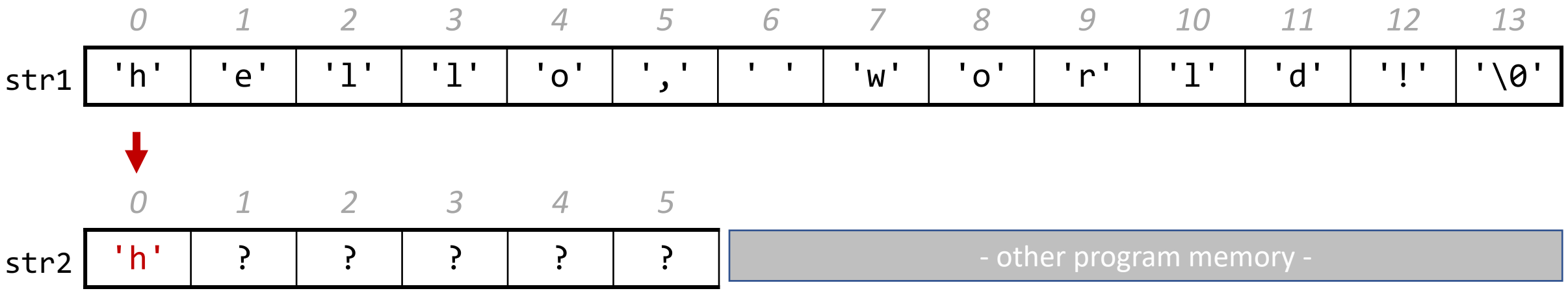
Copying Strings

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



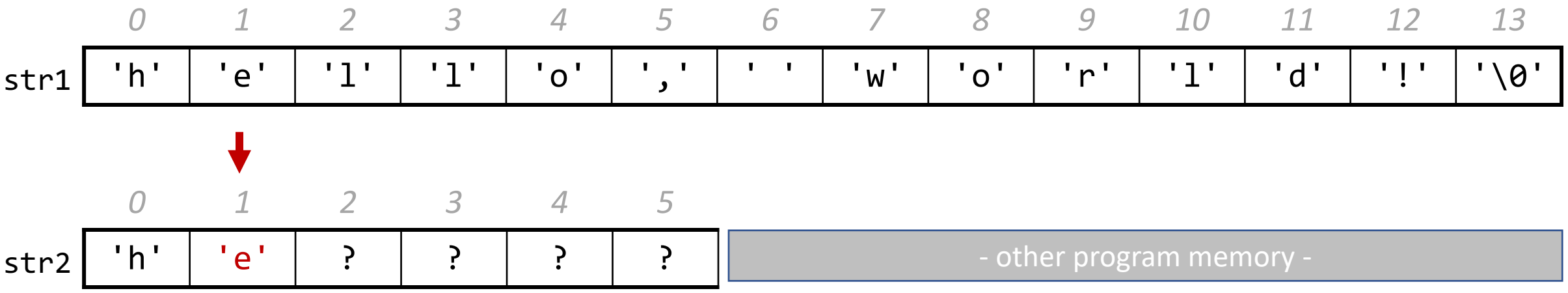
Copying Strings

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



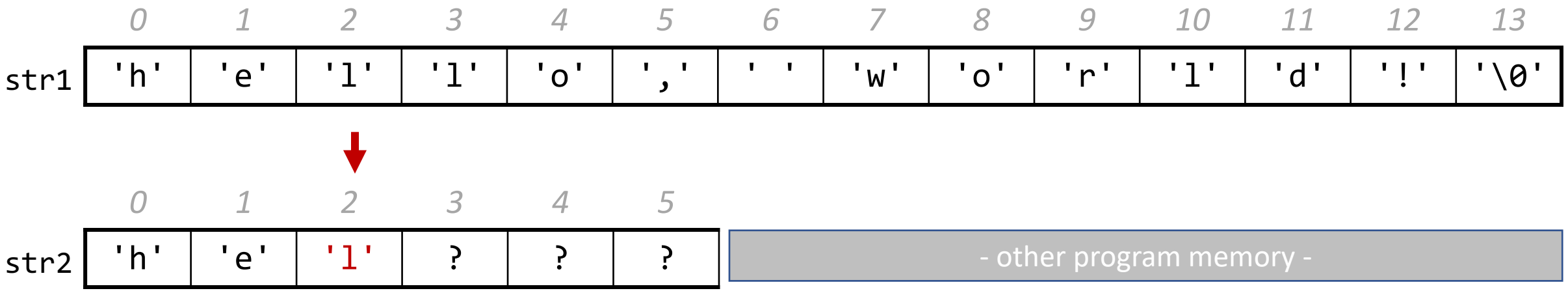
Copying Strings

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



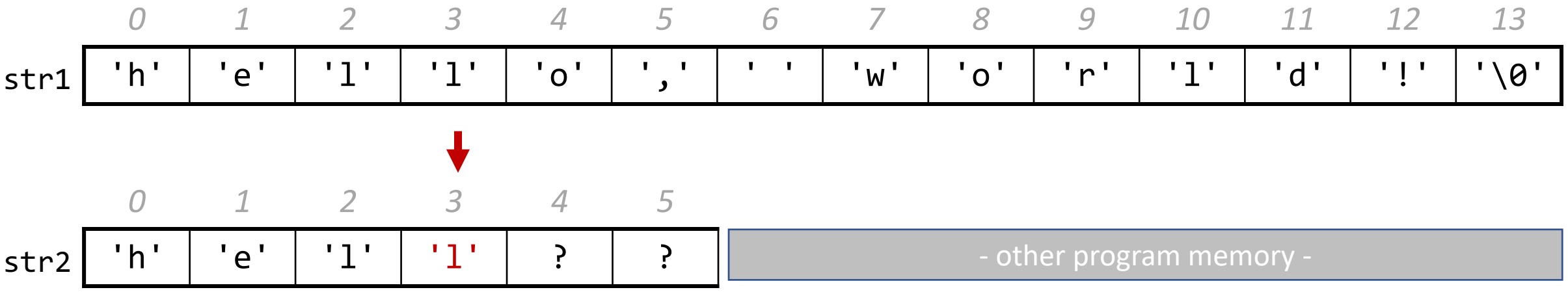
Copying Strings

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



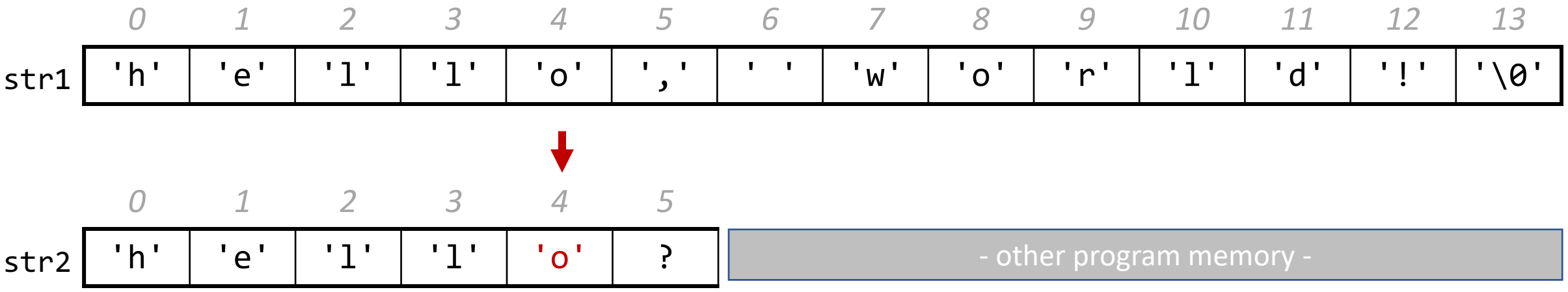
Copying Strings

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



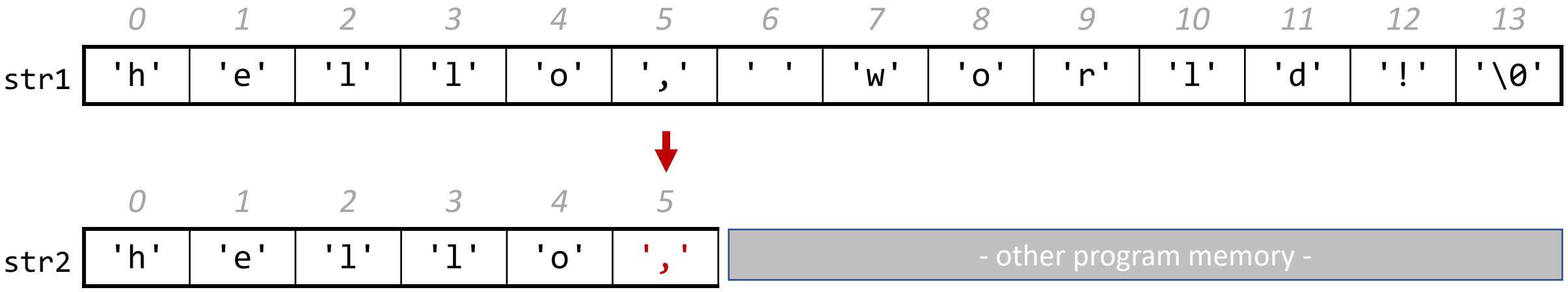
Copying Strings

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



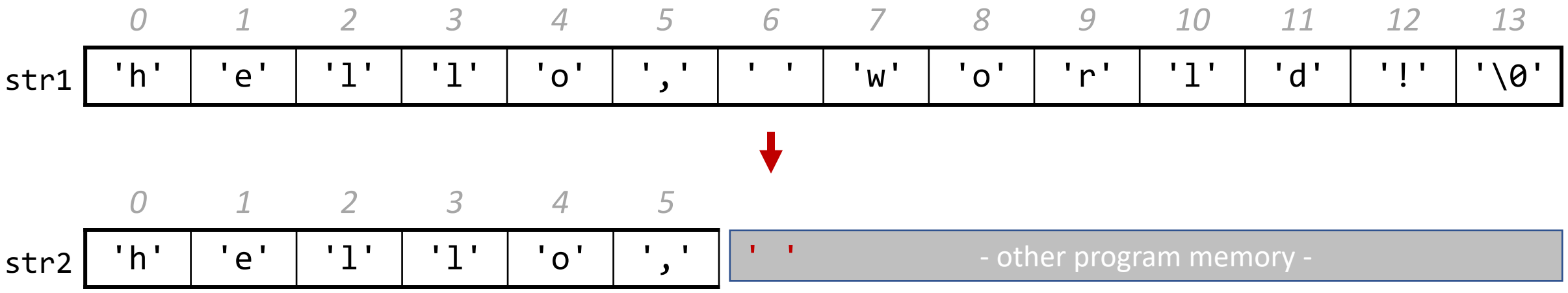
Copying Strings

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



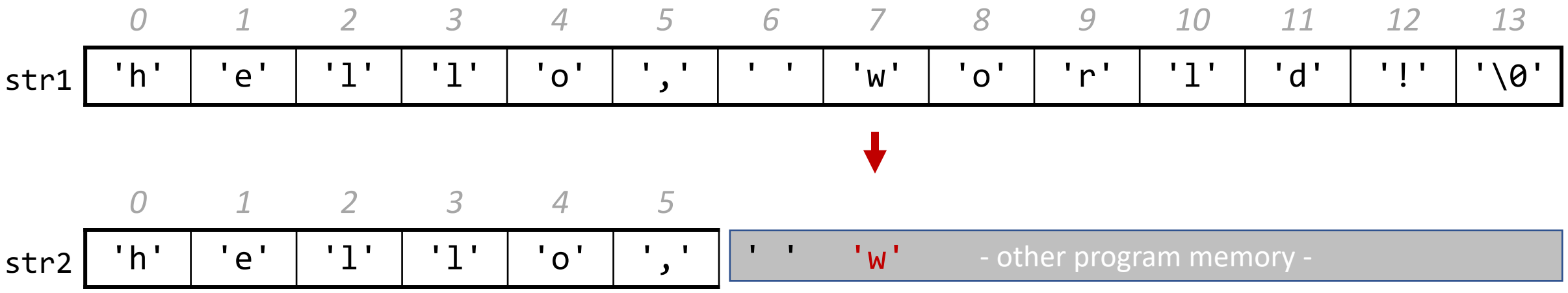
Copying Strings

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



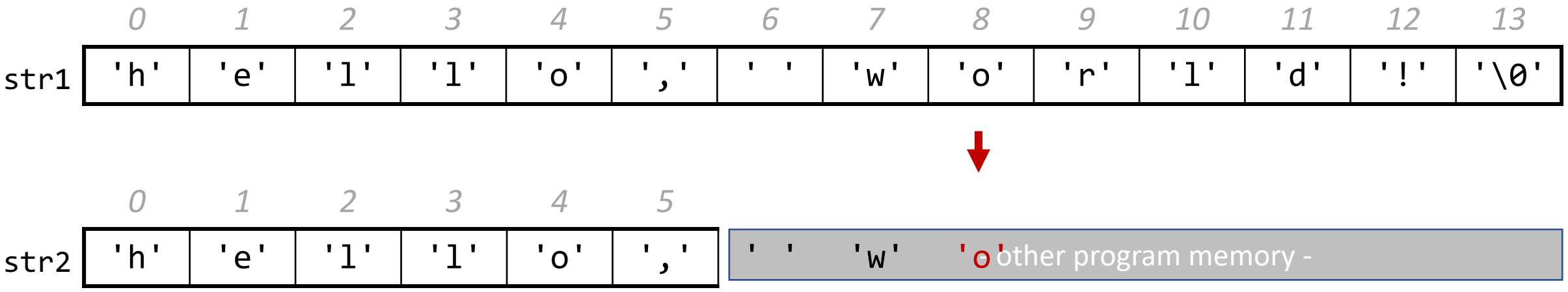
Copying Strings

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



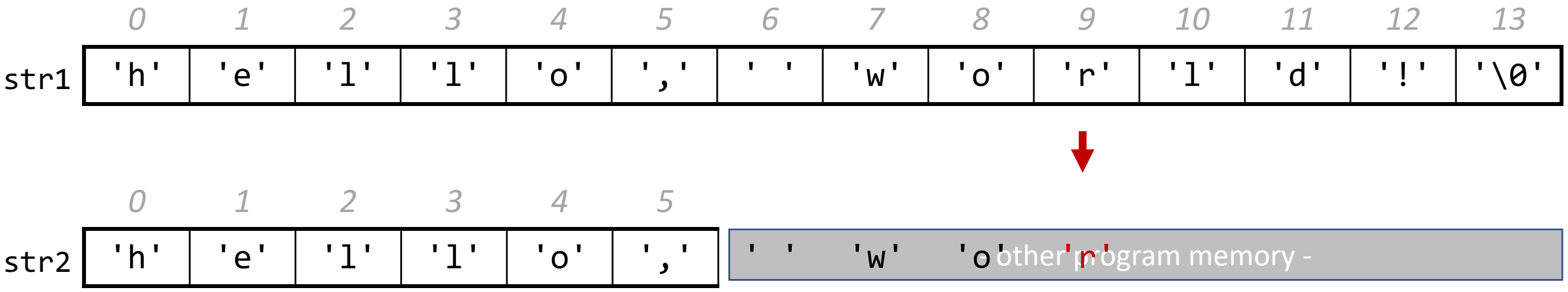
Copying Strings

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



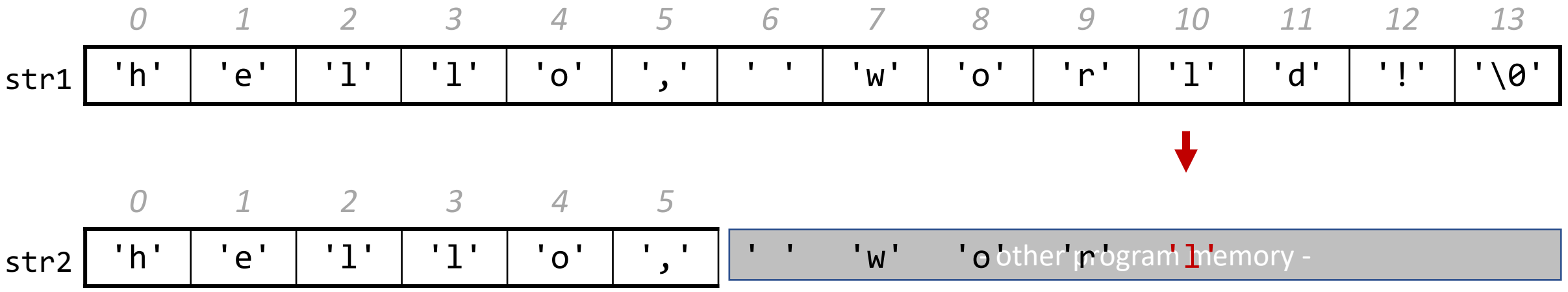
Copying Strings

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



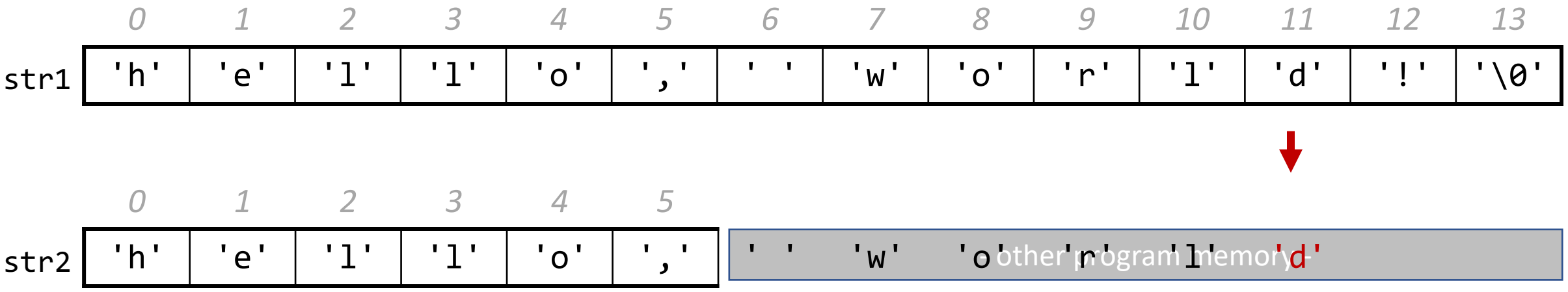
Copying Strings

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



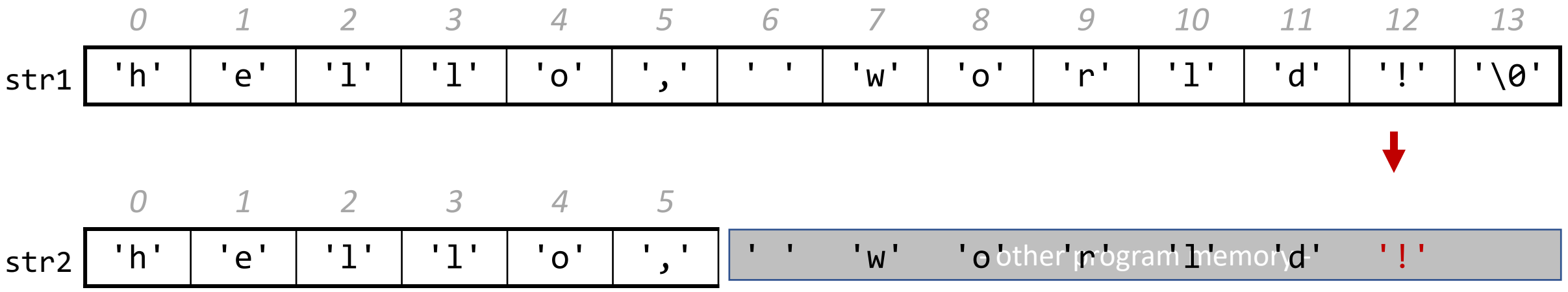
Copying Strings

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



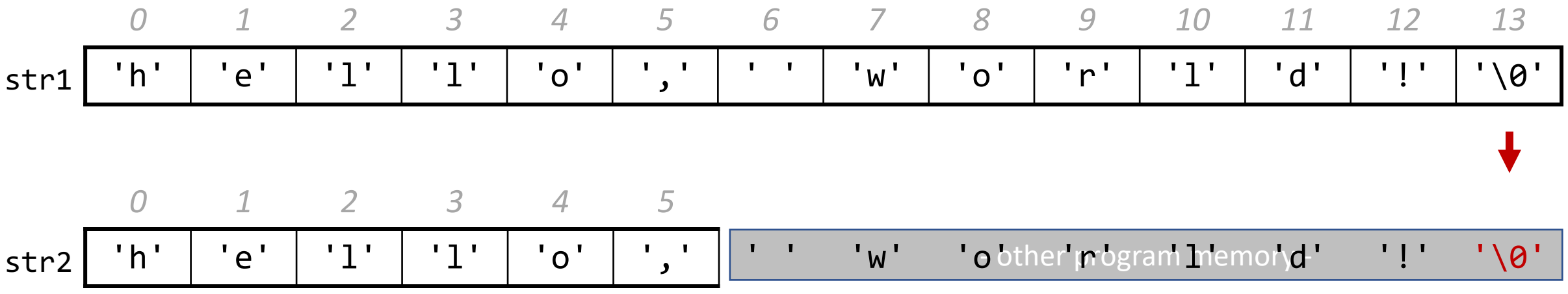
Copying Strings

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



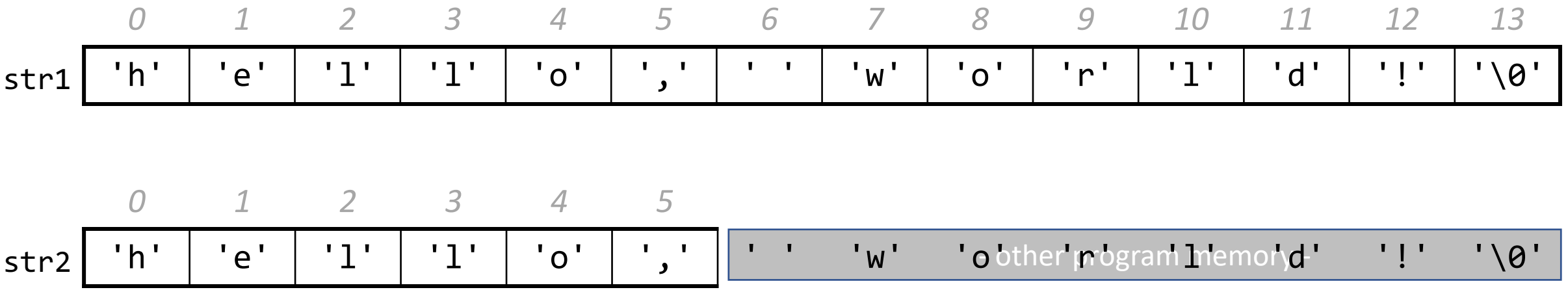
Copying Strings

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



Copying Strings

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



Copying Strings

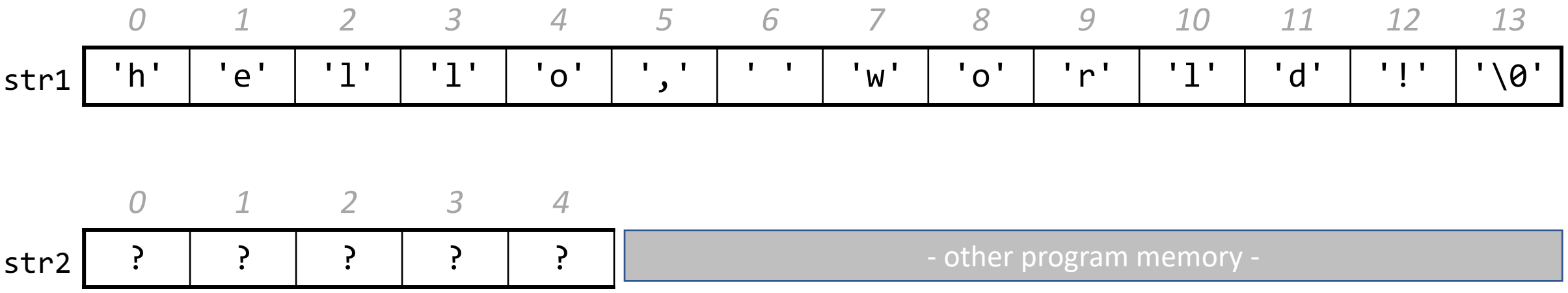
`strncpy` copies at most the first n bytes of **`src`** to **`dst`**. If there is no null-terminating character in these bytes, then **`dst`** will *not be null terminated*!

```
// copying "hello"
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 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'`!

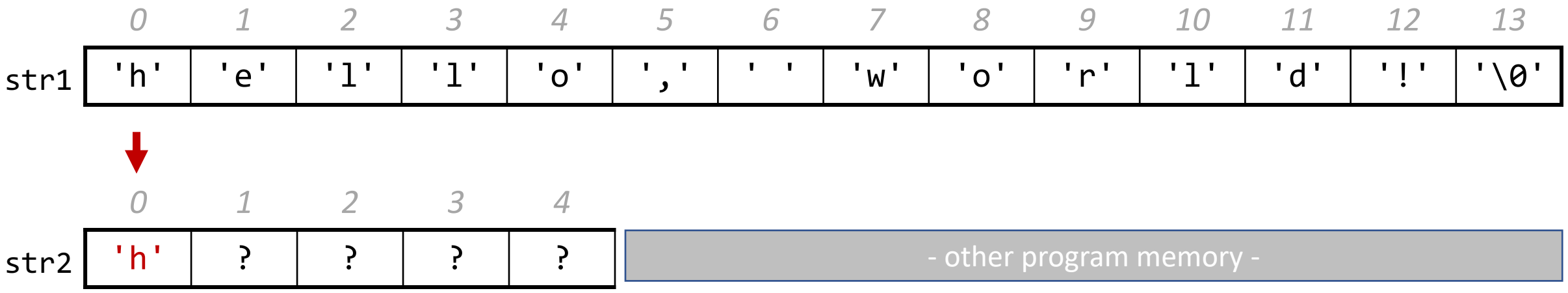
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



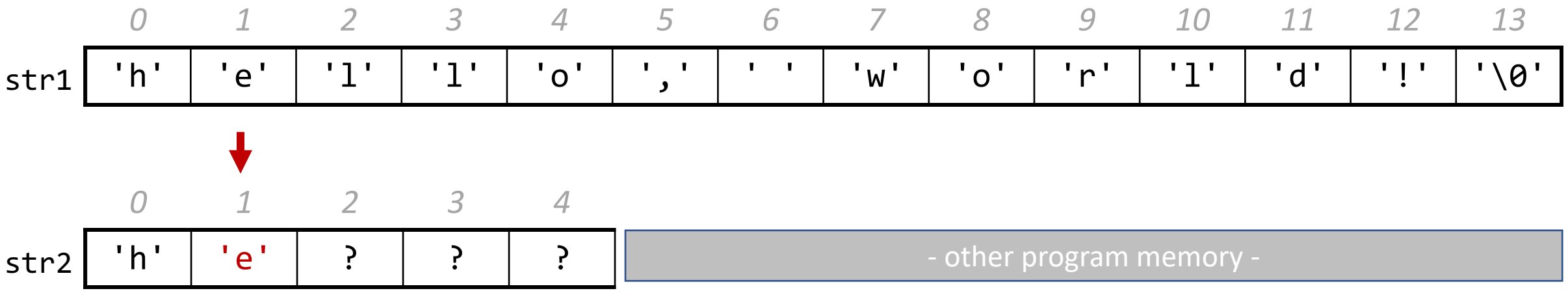
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



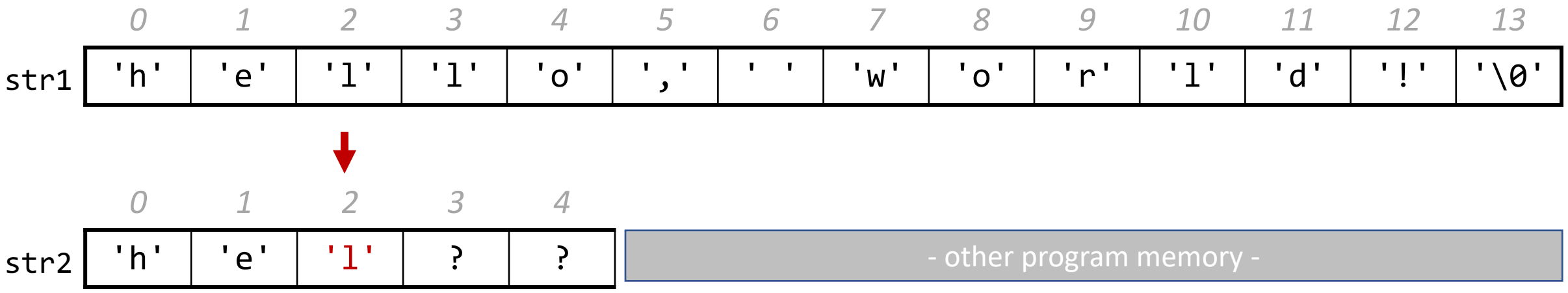
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



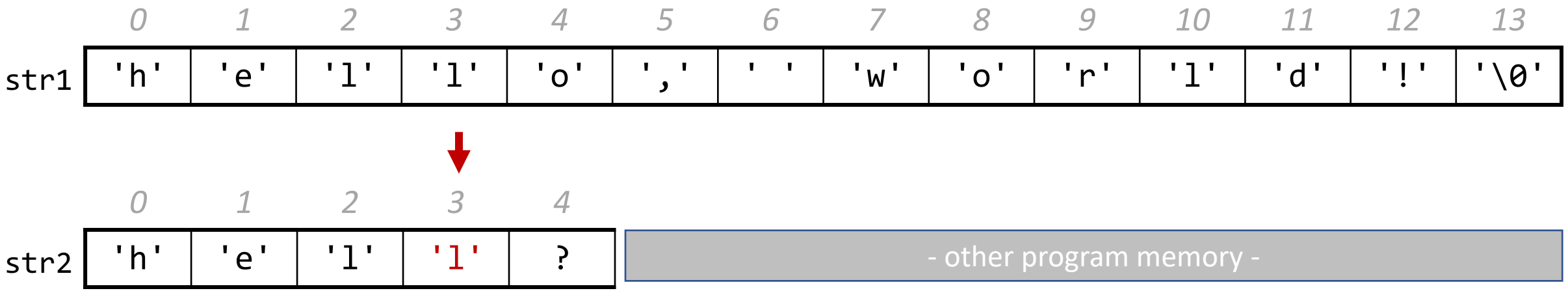
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



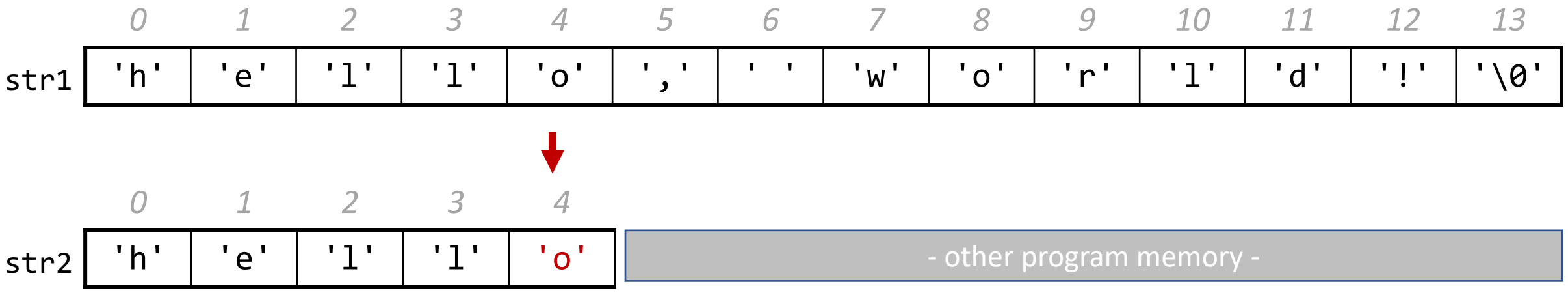
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



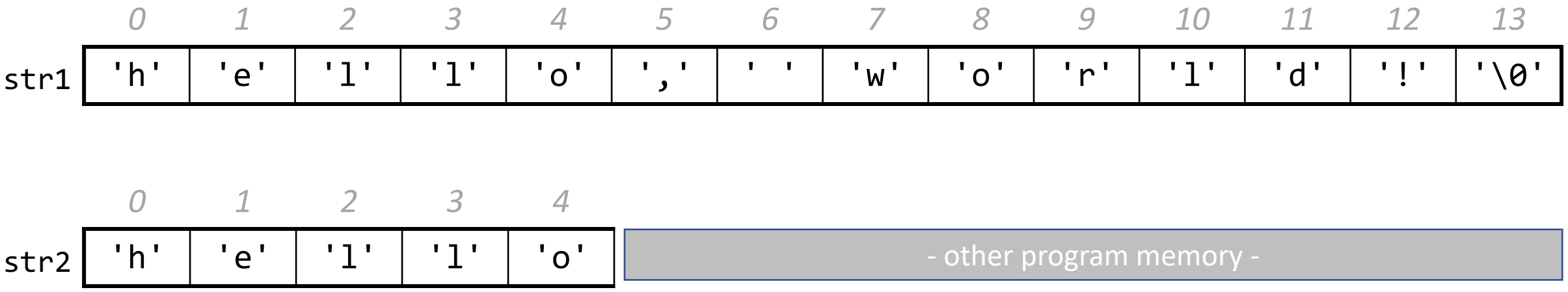
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



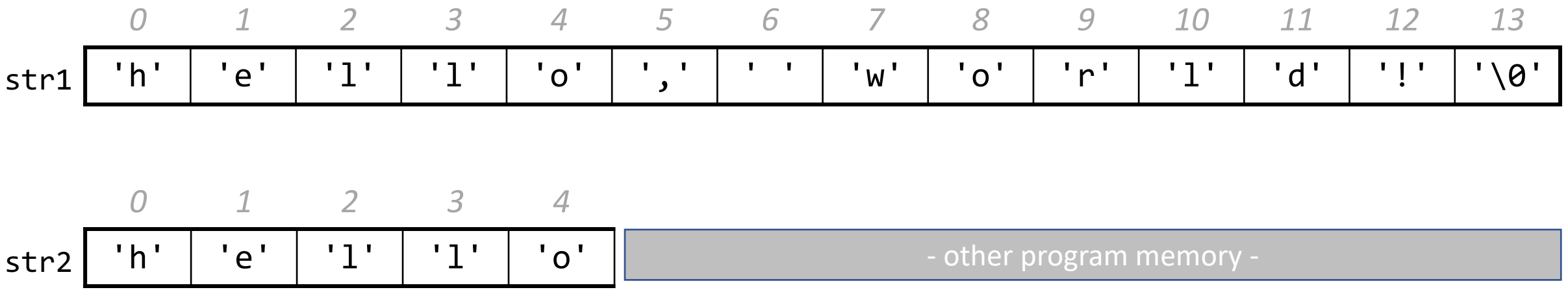
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



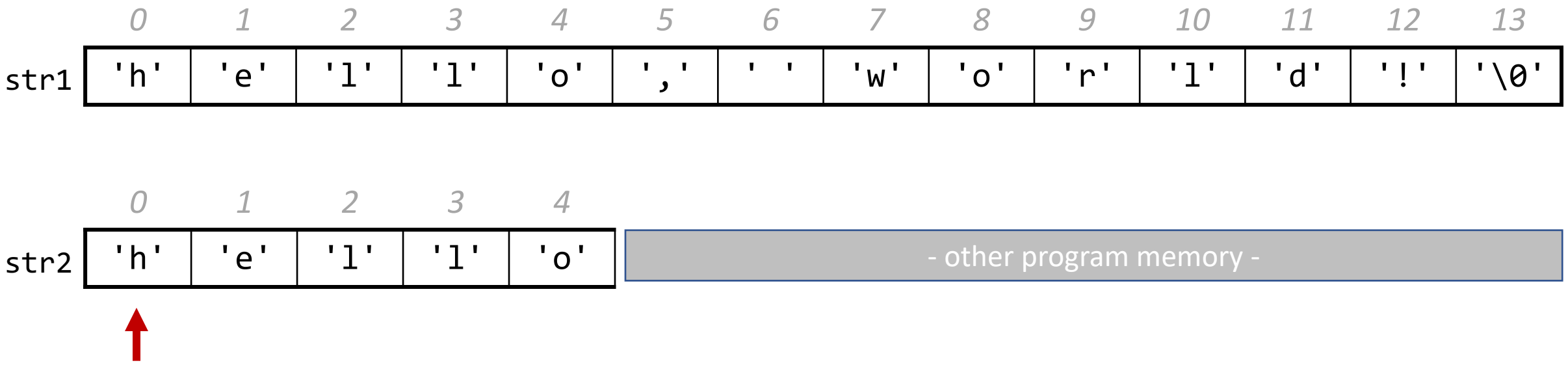
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



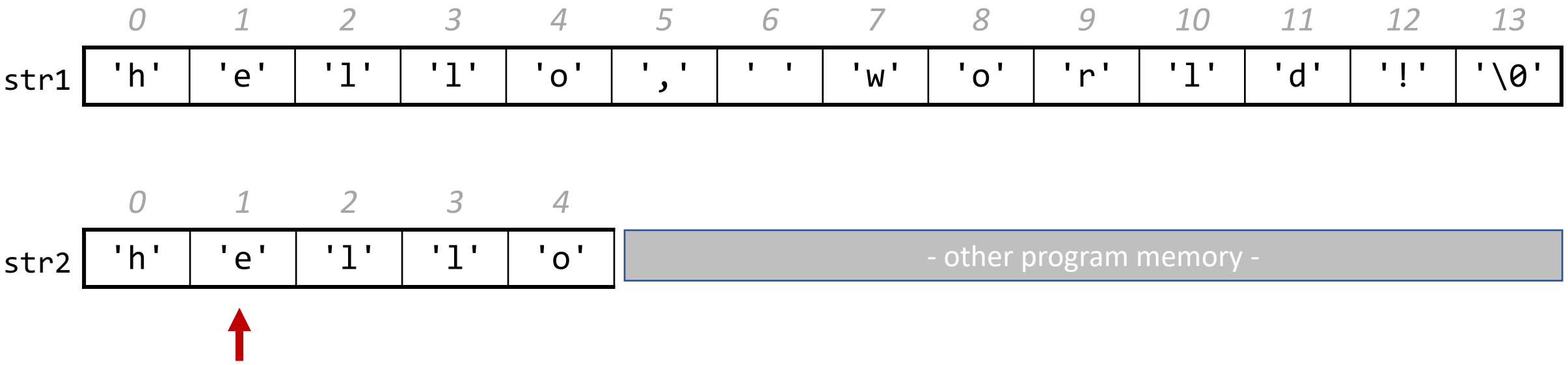
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



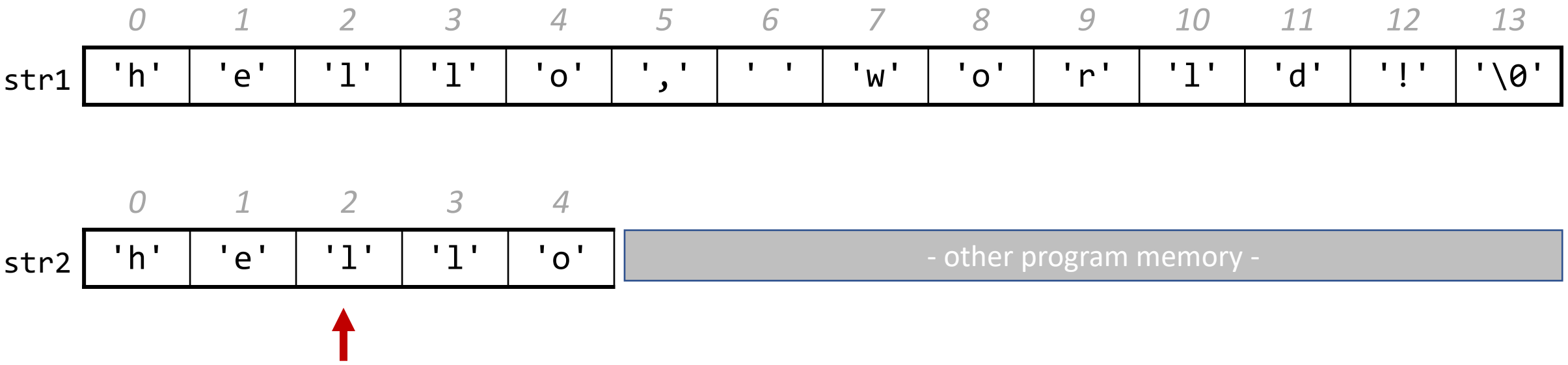
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



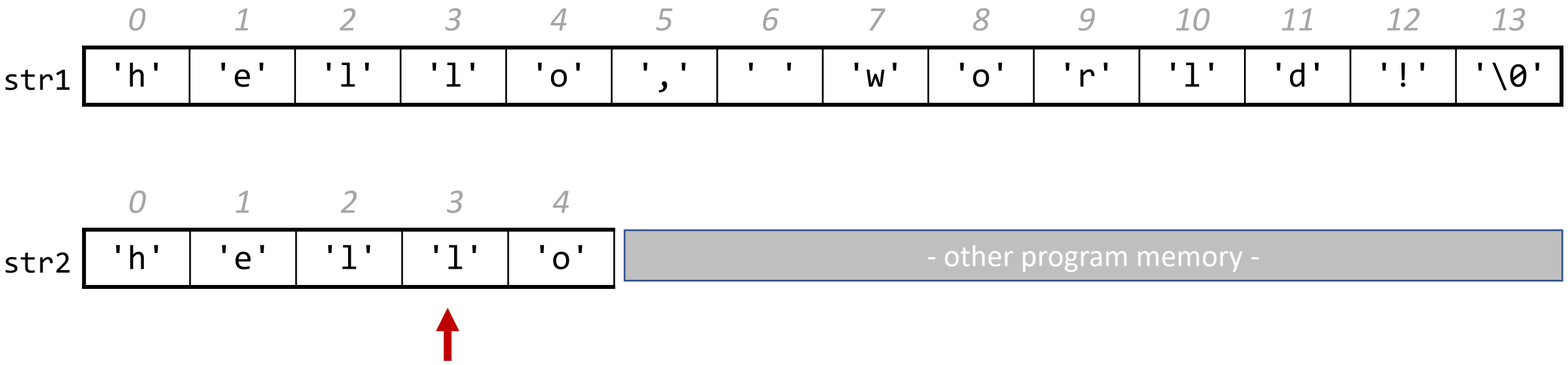
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



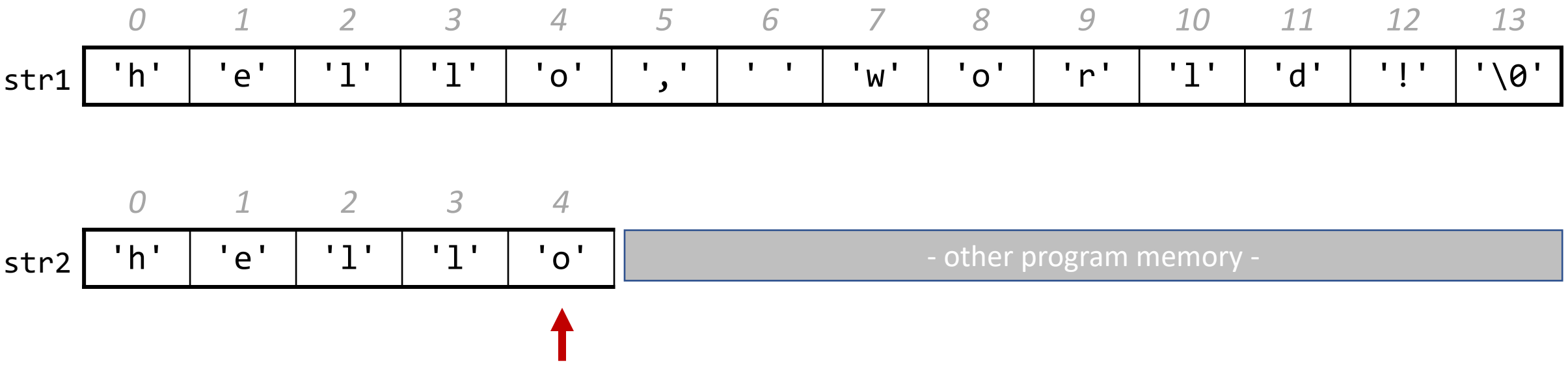
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



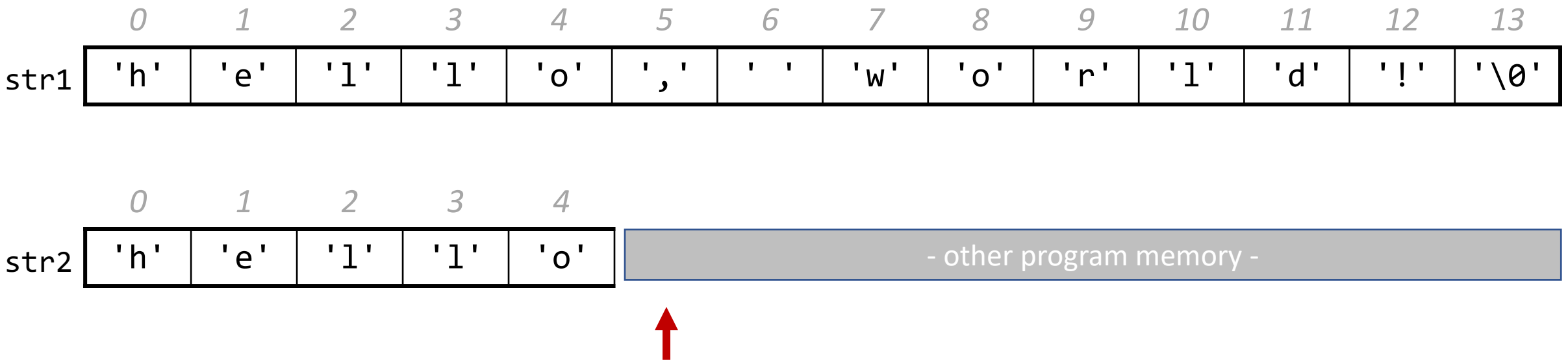
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



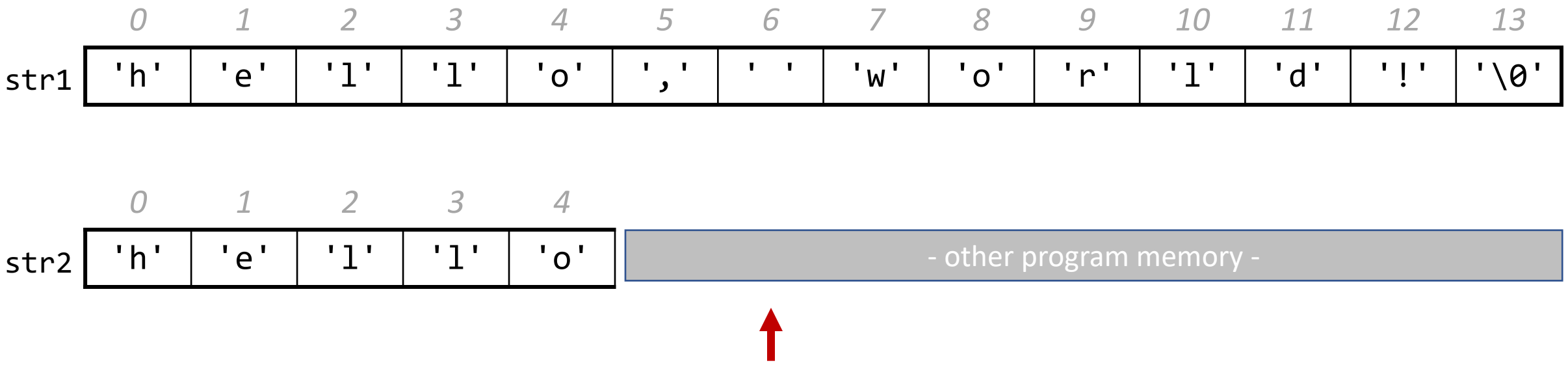
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



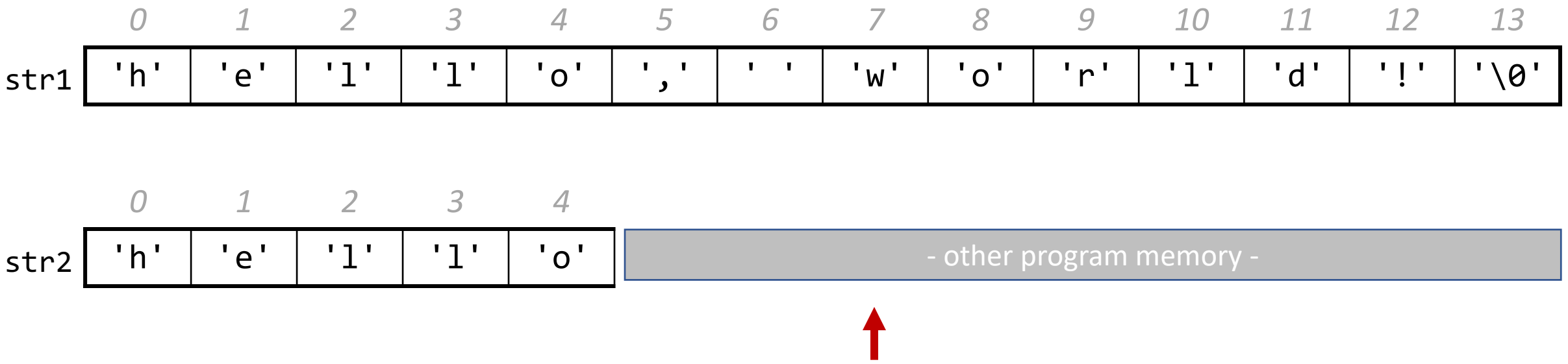
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



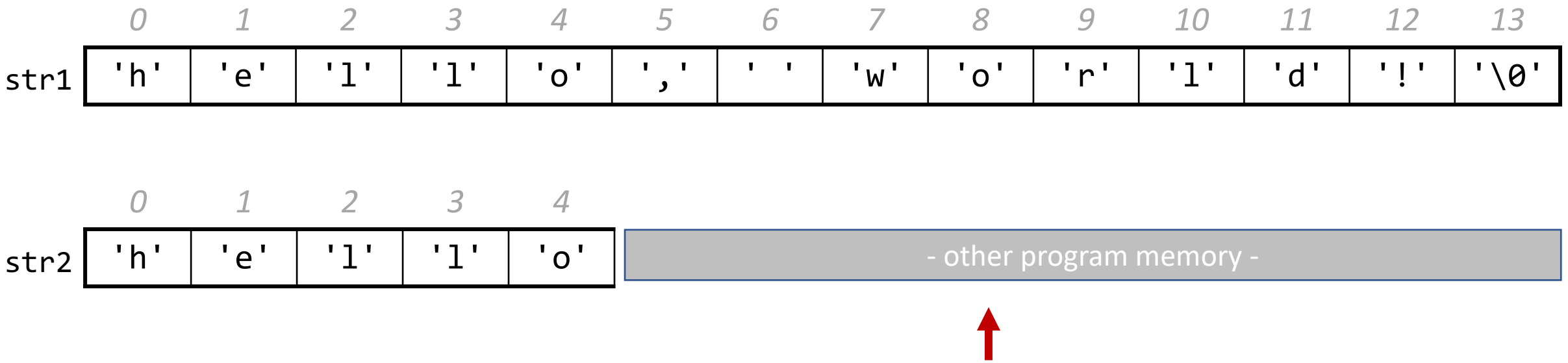
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



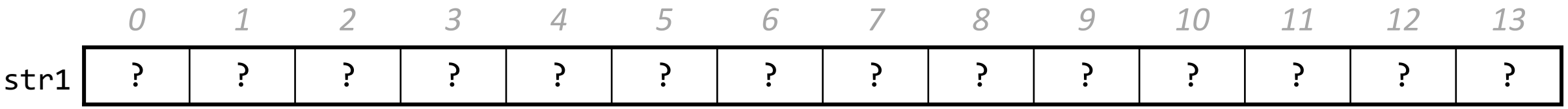
Copying Strings

```
char str1[] = "hello, world!";
char str2[5];
strncpy(str2, str1, 5);
int length = strlen(str2);
```



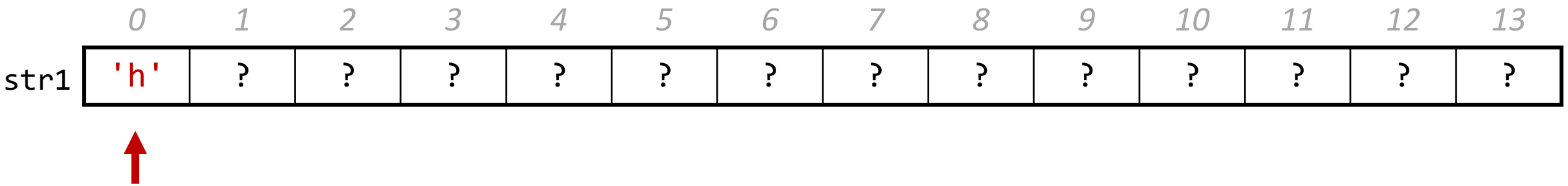
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



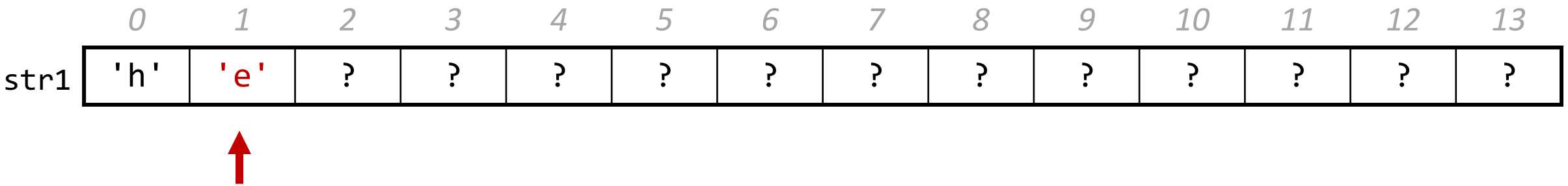
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



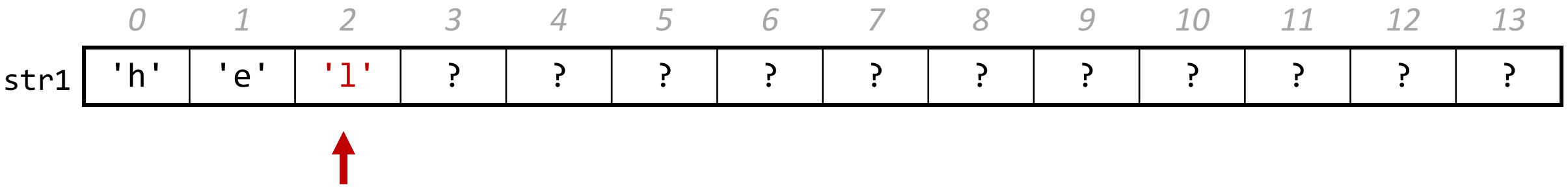
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



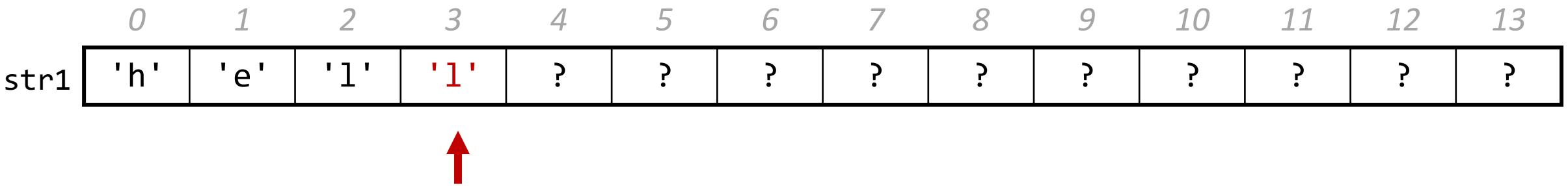
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



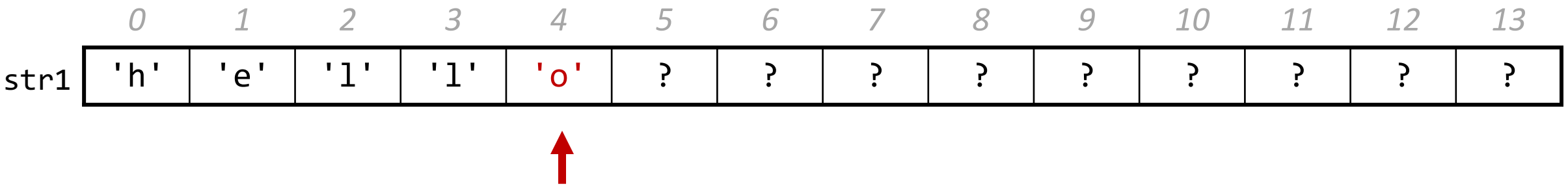
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



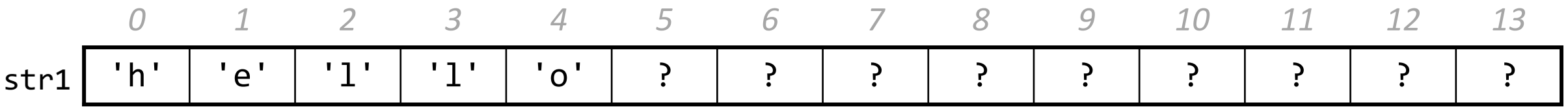
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



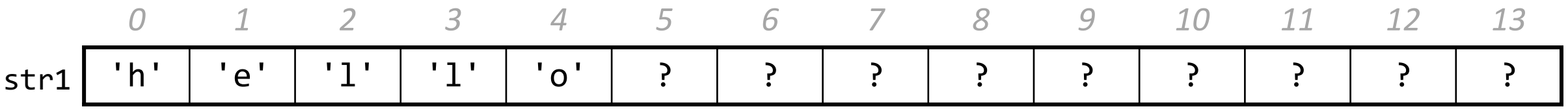
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
```



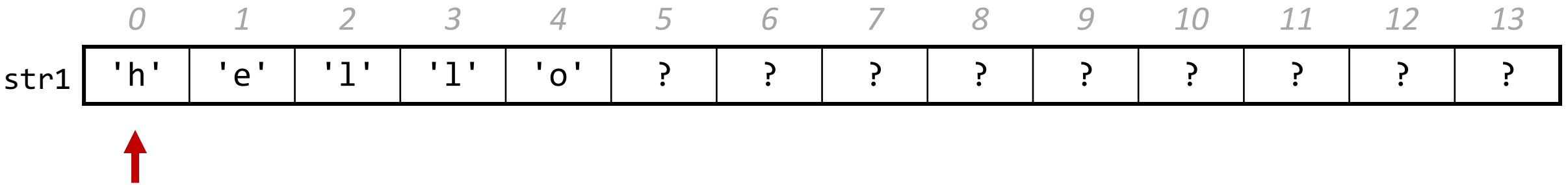
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



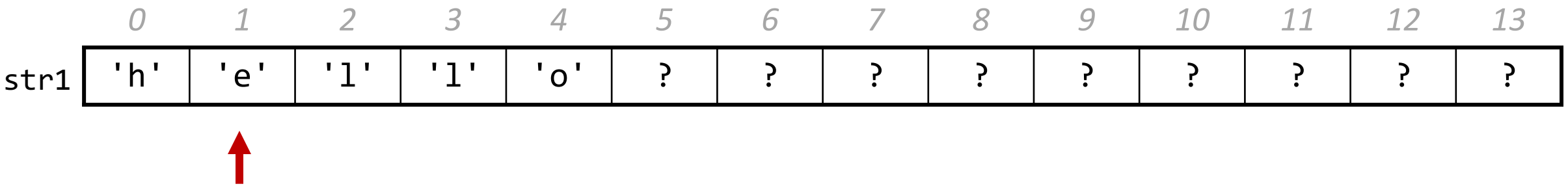
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



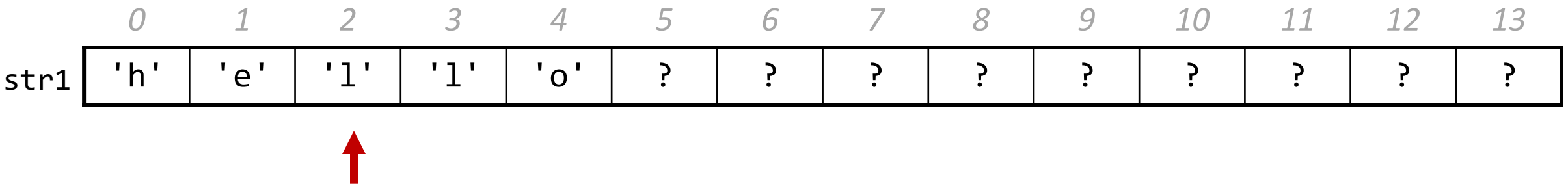
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



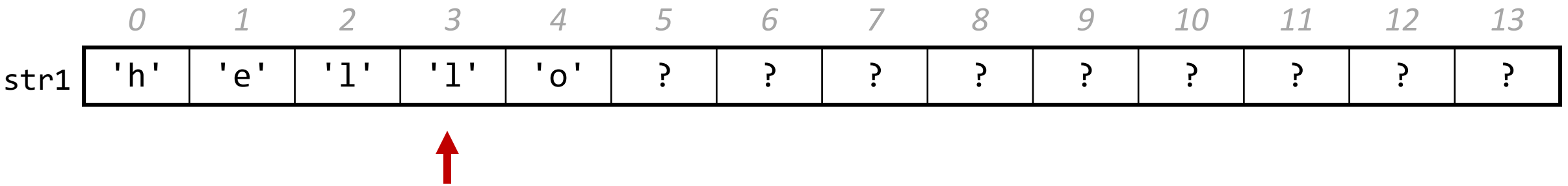
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



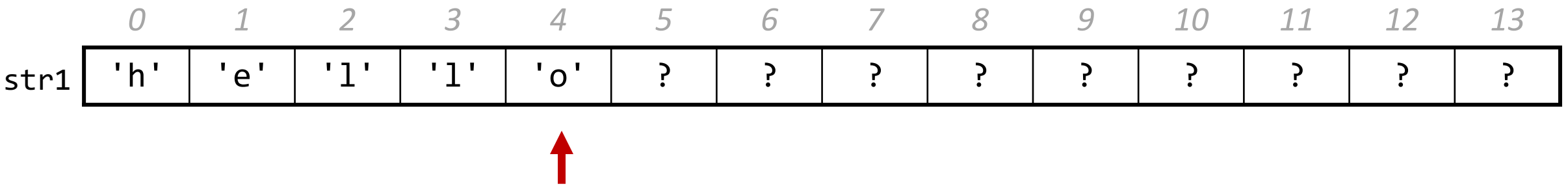
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



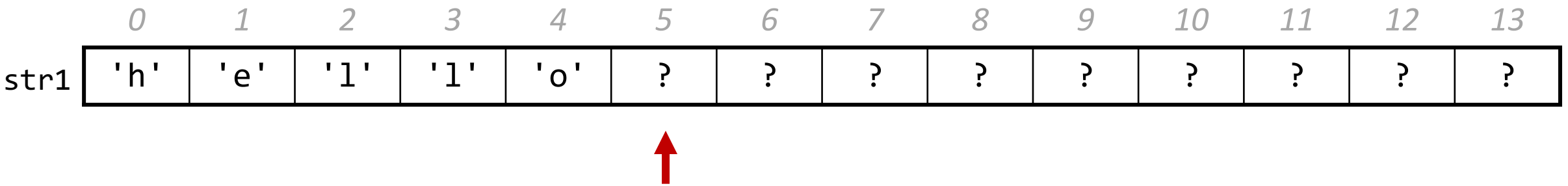
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



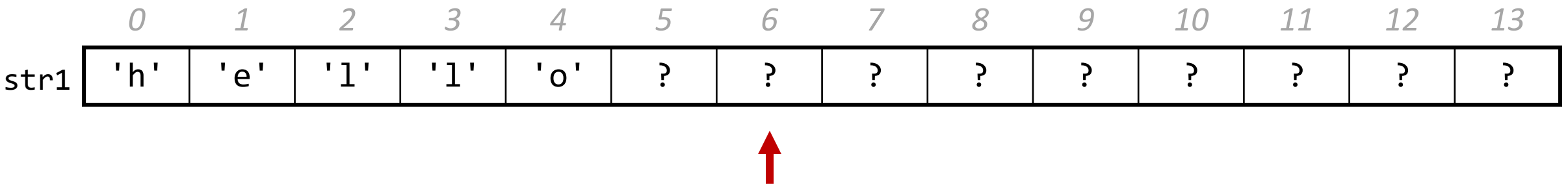
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



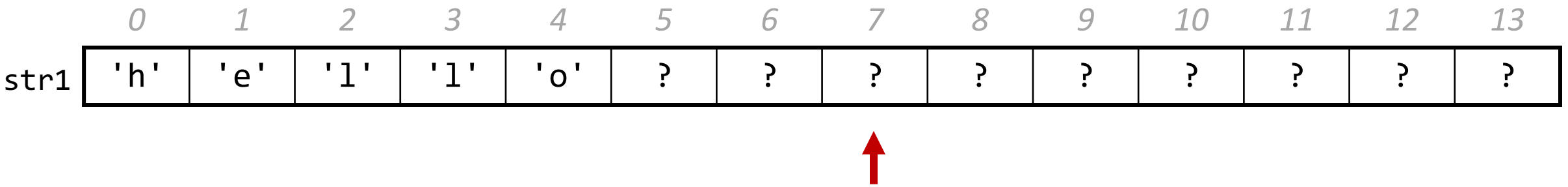
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



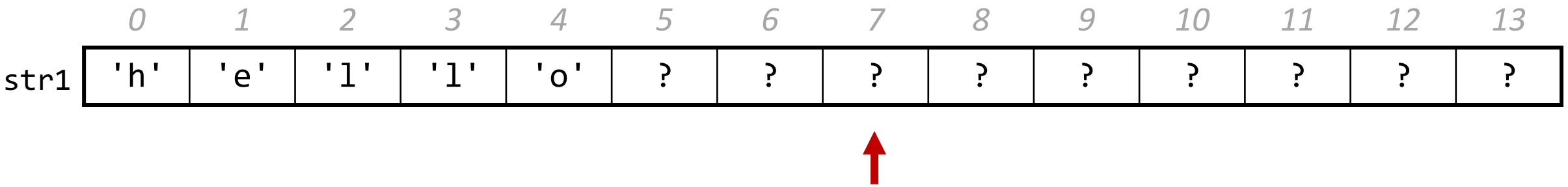
Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```



Copying Strings

```
char str1[14];
char *str2 = "hello there";
strncpy(str1, str2, 5);
printf("%s\n", str1);
```




```
$ ./strncpy_buggy wonderful
word: wonderful
wordcopy: wonde□□J□□□
```

Copying Strings

If necessary, make sure to add a null-terminating character yourself.

```
// copying "hello"
char str1[] = "hello, world!";
char str2[6];                       // room for string and '\0'
strncpy(str2, str1, 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 *hello = "hello";
char str[_____];
strcpy(str, hello);
```

Concatenating Strings

You cannot concatenate C strings using +. This adds character addresses!

```
char *str1 = "hello ";    // e.g. 0x7f42
char *str2 = "world!";    // e.g. 0x6541.
char *str3 = str1 + str2;// doesn't compile!
```

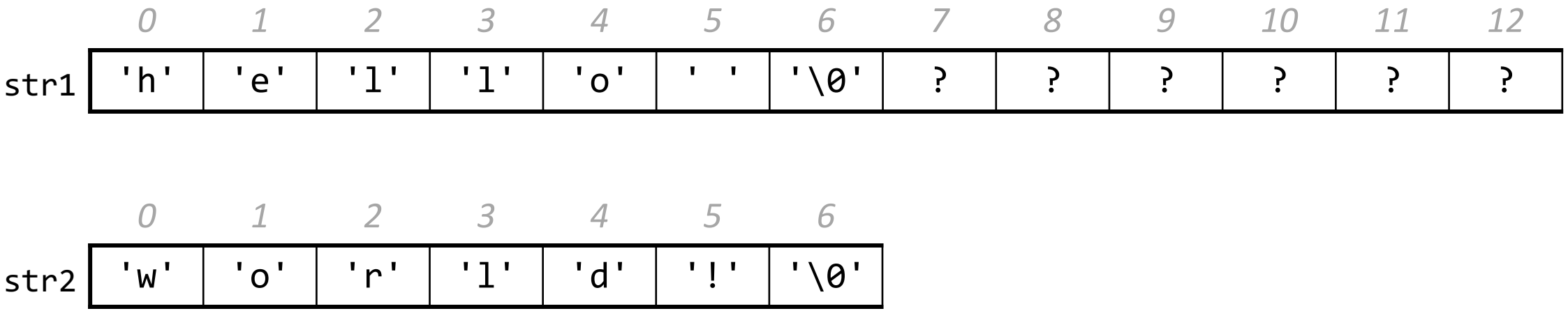
Instead, use `strcat`:

```
char str1[13] = "hello ";// enough space for strings + '\0'
char str2[] = "world!";  // e.g. 0x6541.
strcat(str1, str2);      // 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.

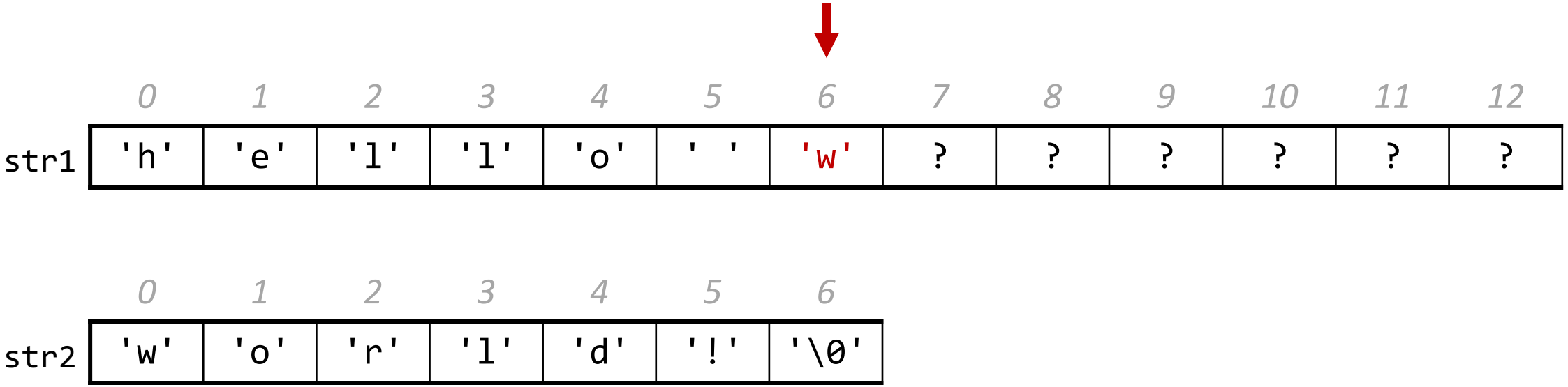
Concatenating Strings

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



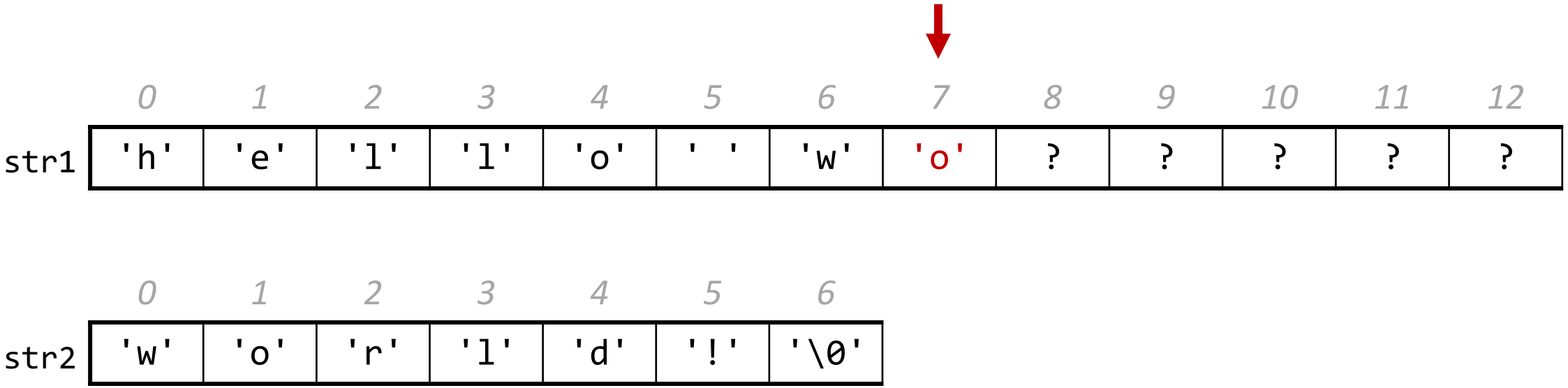
Concatenating Strings

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



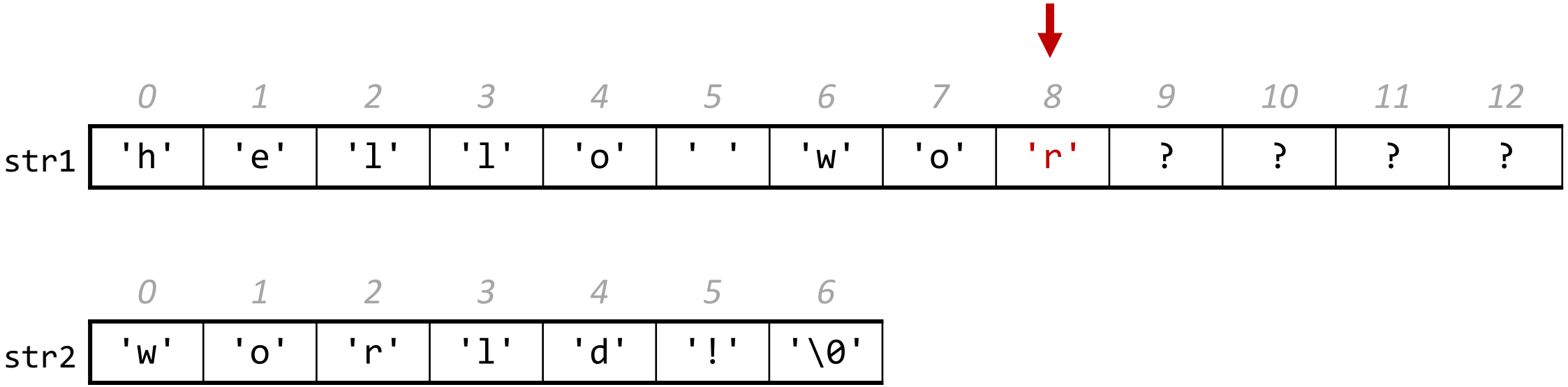
Concatenating Strings

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



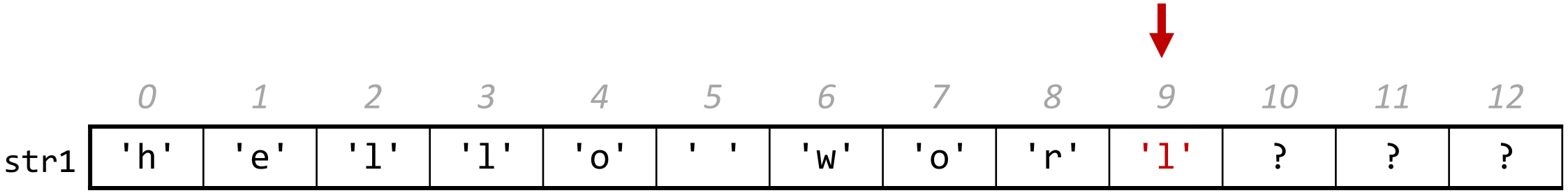
Concatenating Strings

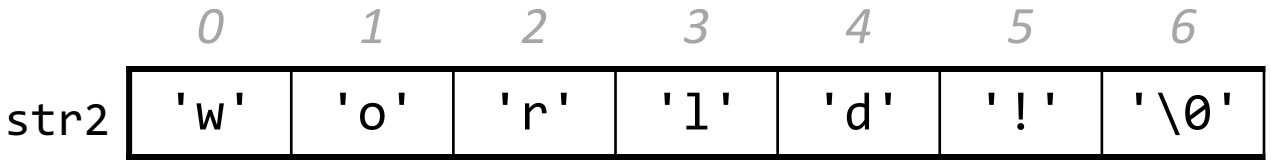
```
char str1[13] = "hello ";
char str2[] = "world!";
strcat(str1, str2);
```



Concatenating Strings

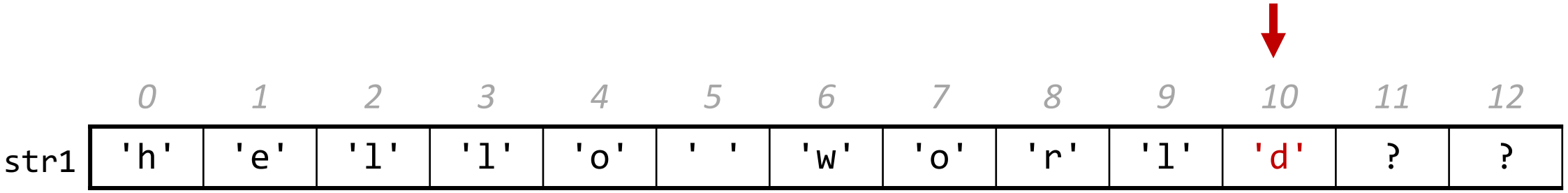
```
char str1[13] = "hello ";
char str2[] = "world!";
strcat(str1, str2);
```

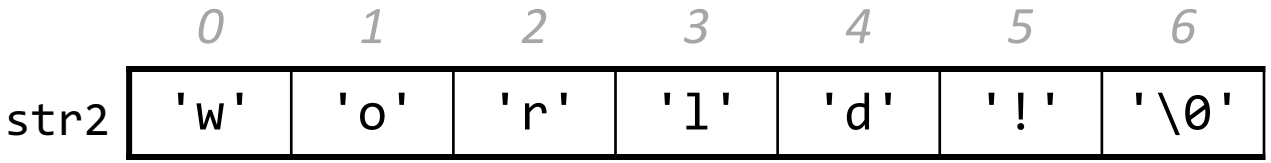




Concatenating Strings

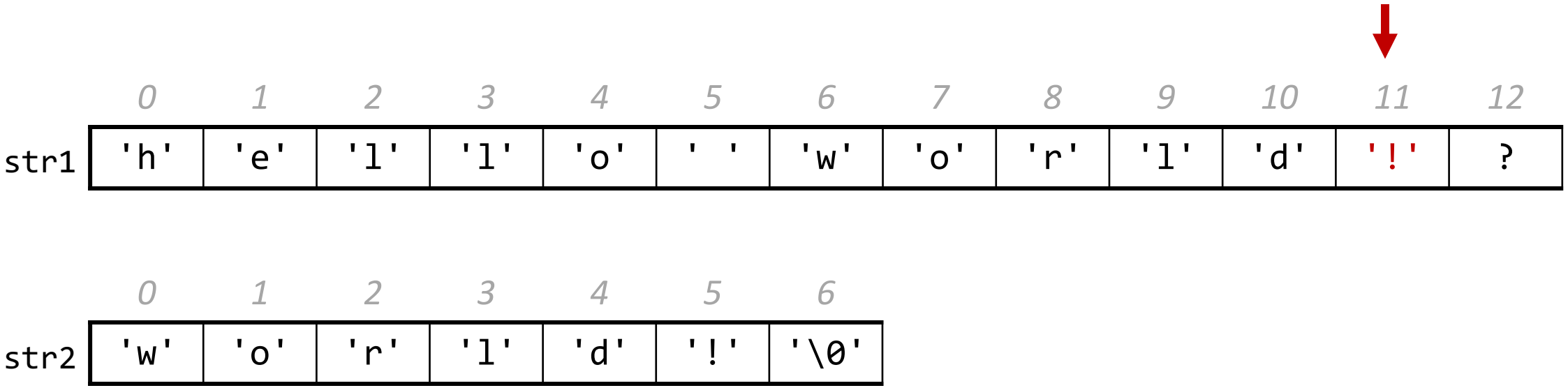
```
char str1[13] = "hello ";
char str2[] = "world!";
strcat(str1, str2);
```





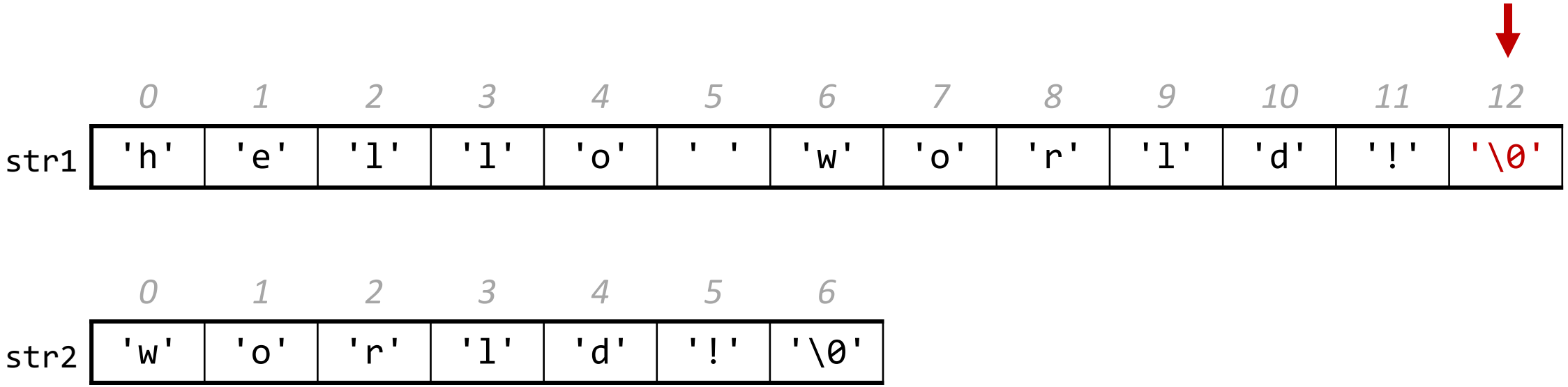
Concatenating Strings

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



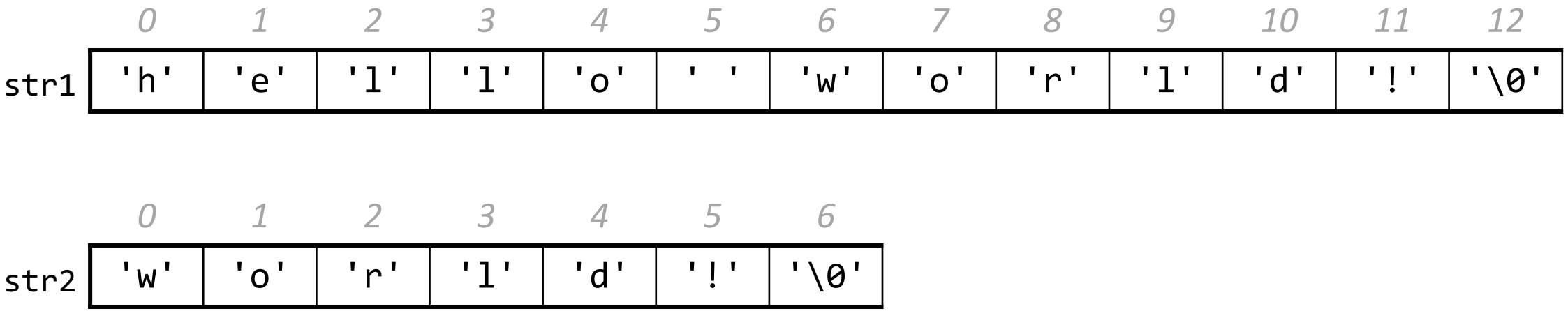
Concatenating Strings

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



Concatenating Strings

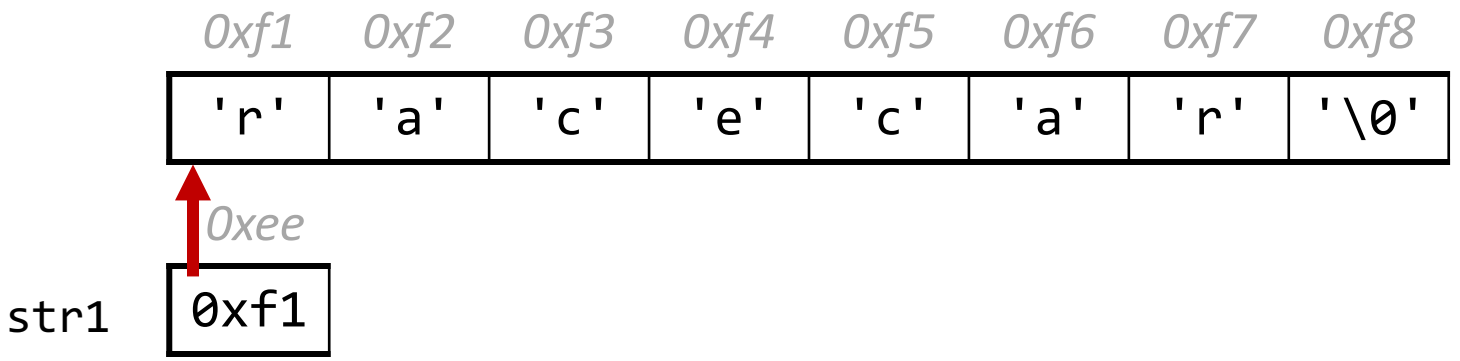
```
char str1[13] = "hello ";
char str2[] = "world!";
strcat(str1, str2);
```



Substrings

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

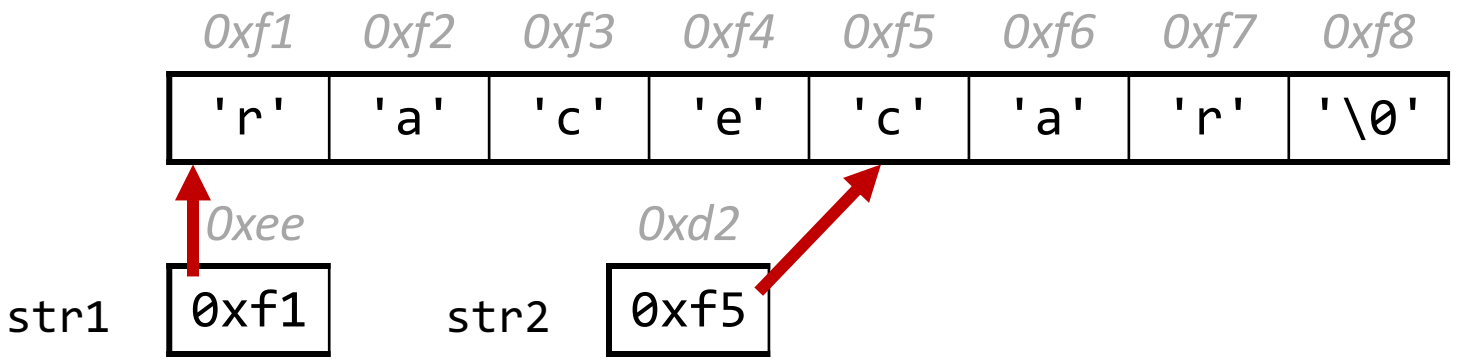
```
// Want just "car"
char *str1 = "racecar";
```



Substrings

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

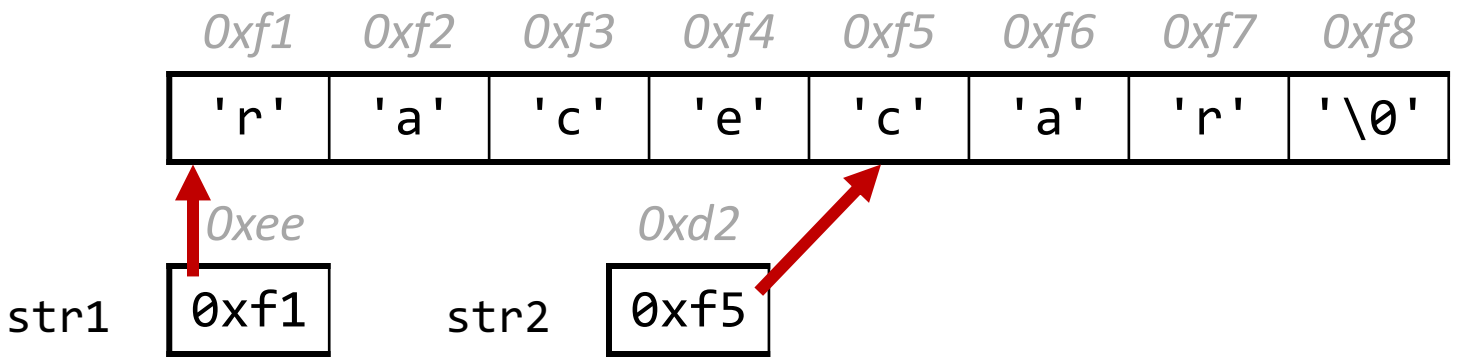
```
// Want just "car"
char *str1 = "racecar";
char *str2 = str1 + 4;
```



Substrings

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

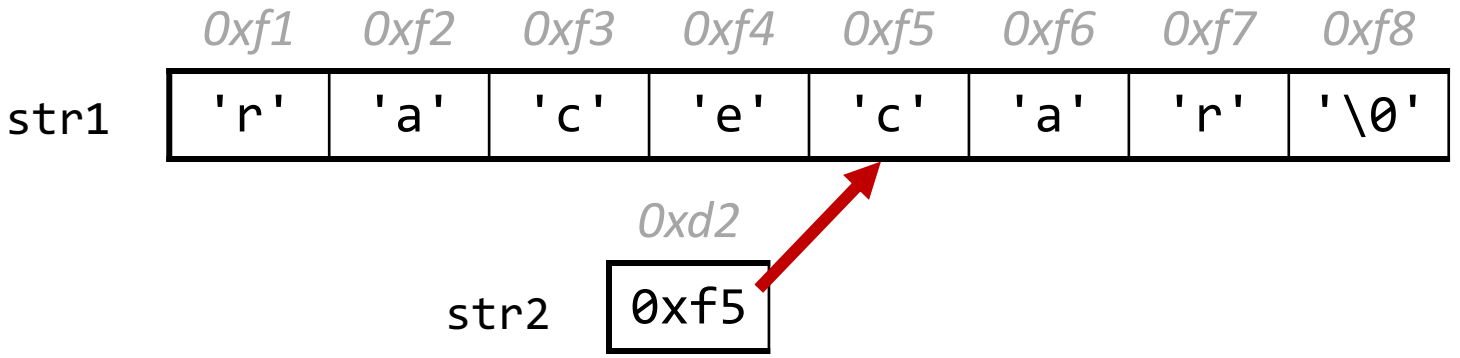
```
// Want just "car"
char *str1 = "racecar";
char *str2 = str1 + 4;
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // car
```



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!

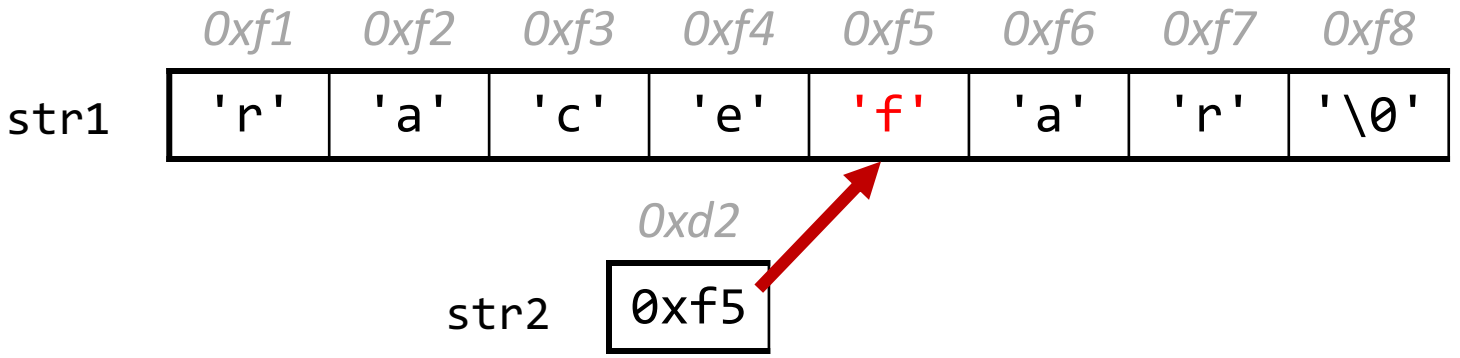
```
char str1[] = "racecar";
char *str2 = str1 + 4;
str2[0] = 'f';
printf("%s\n", str1);
printf("%s\n", str2);
```



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!

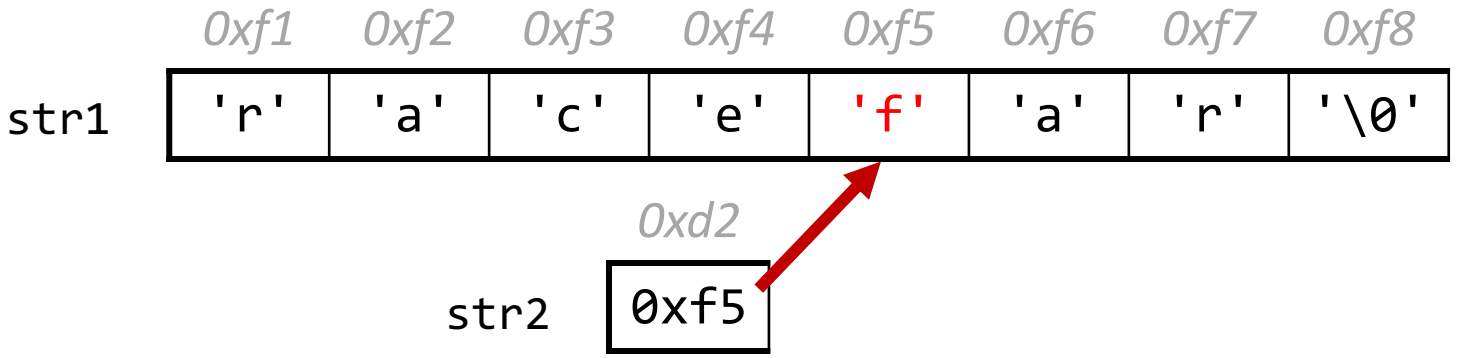
```
char str1[] = "racecar";
char *str2 = str1 + 4;
str2[0] = 'f';
printf("%s\n", str1);
printf("%s\n", str2);
```



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



Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char *str1 = "racecar";
```

Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char *str1 = "racecar";
char str2[5];
```

Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char *str1 = "racecar";
char str2[5];
strncpy(str2, str1, 4);
```

Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char *str1 = "racecar";
char str2[5];
strncpy(str2, str1, 4);
str2[4] = '\0';
```

Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char *str1 = "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 = "racecar";
```

Substrings

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

```
// Want just "ace"
char *str1 = "racecar";
char str2[4];
```

Substrings

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

```
// Want just "ace"
char *str1 = "racecar";
char str2[4];
strcpy(str2, str1 + 1, 3);
```

Substrings

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

```
// Want just "ace"
char *str1 = "racecar";
char str2[4];
strcpy(str2, str1 + 1, 3);
str2[4] = '\0';
```

Substrings

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

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

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

Announcements

- Clarification: reusing code from prior quarters you took CS107
- Emacs and GDB configuration files
  - cs107.stanford.edu/resources/emacs
  - cs107.stanford.edu/resources/gdb
- We hope you enjoyed your first lab!
- 3 minute break

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

String Diamond

- Write a function **`diamond`** that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  - For example, `diamond("DAISY")` should print:

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

String Diamond

- Write a function **`diamond`** that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  - For example, `diamond("DAISY")` should print:

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

Daisy!

Practice: Diamond

Plan For Today

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char *daisy = "Daisy";
char *letterA = strchr(daisy, 'a');
printf("%s\n", daisy);              // Daisy
printf("%s\n", letterA);            // aisy
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `strrchr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char *daisy = "Daisy Dog";
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy);              // Daisy Dog
printf("%s\n", substr);             // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char *daisy = "Daisy Dog";
int spanLength = strspn(daisy, "Daeoi");          // 3
```

String Spans

`strcspn` (c = “complement”) returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char *daisy = "Daisy Dog";
int spanLength = strcspn(daisy, "isdor");       // 2
```

Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- **Break**: Announcements
- **Practice**: Diamond
- More String Operations: Searching and Spans

Next time (video recording only): more strings