CS107, Lecture 4
C Strings

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

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

Lecture Plan

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

Lecture Plan

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

Char

A **`char`** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newLine = '\n';
char tab = '\t';
char singleQuote = '\'';
char backSlash = '\\';
```

ASCII

Under the hood, C represents each **`char`** as an *integer* (its "ASCII value").

- Uppercase letters are sequentially numbered
- Lowercase letters are sequentially numbered
- Digits are sequentially numbered
- Lowercase letters are 32 more than their uppercase equivalents (bit flip!)

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

ASCII

We can take advantage of C representing each **`char`** as an *integer:*

```
bool areEqual = 'A' == 'A';          // true
bool earlierLetter = 'f' < 'c';      // false
char uppercaseB = 'A' + 1;
int diff = 'c' - 'a';                // 2
int numLettersInAlphabet = 'z' – 'a' + 1;
// or
int numLettersInAlphabet = 'Z' – 'A' + 1;
```

ASCII

We can take advantage of C representing each **`char`** as an *integer:*

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

Common `ctype.h` Functions

Function	Description
`isalpha(`***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

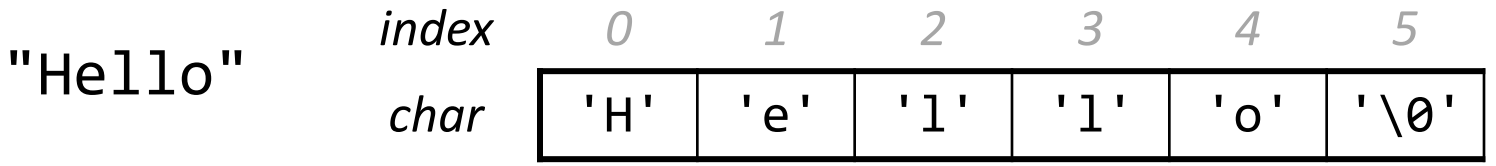
```
bool isLetter = isalpha('A');       // true
bool capital = isupper('f');        // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');       // true
```

Lecture Plan

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

C Strings

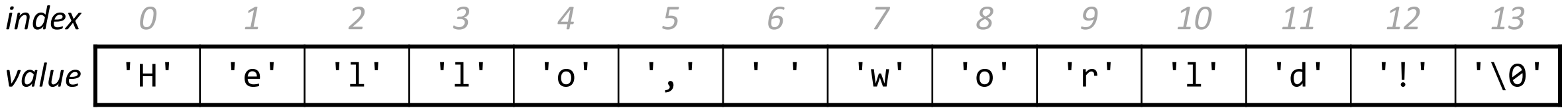
C has no dedicated variable type for strings. Instead, a string is represented as an **array of characters** with a special ending sentinel value.



`'\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!



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**: Diamond

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 does not 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. Always adds a null-terminating character.
`strspn(`***`str, accept`***`),` `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***`str`*** which contains only characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does not 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 does not 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. Always adds a null-terminating character.
`strspn(str, accept)`, `strcspn(str, reject)`	**strspn** returns the length of the initial part of ***str*** which contains only characters in ***accept***. **strcspn** returns the length of the initial part of ***str*** which does not 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`**.

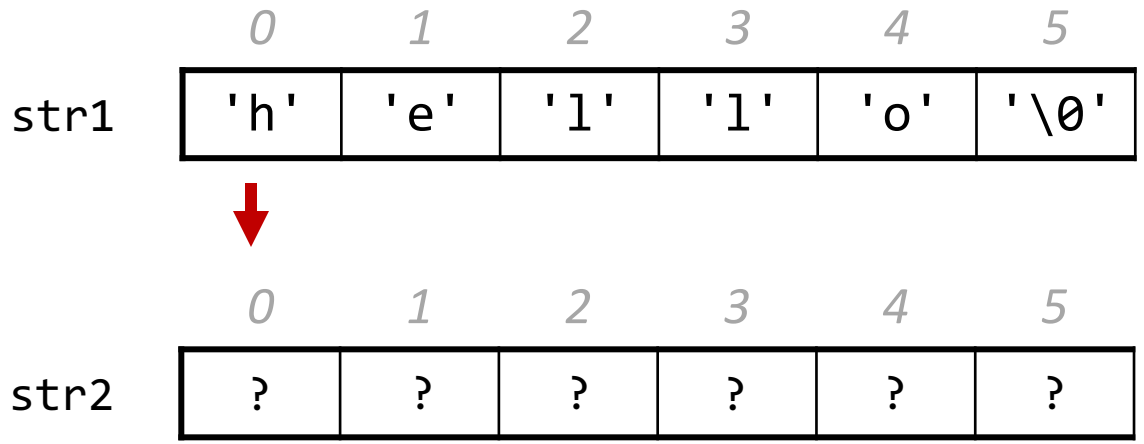
The string library: `strcpy`

`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`

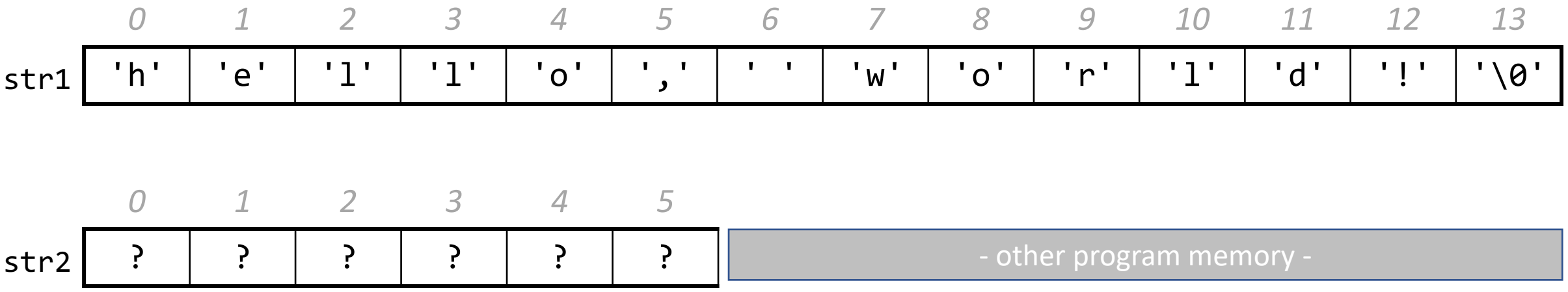
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
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!

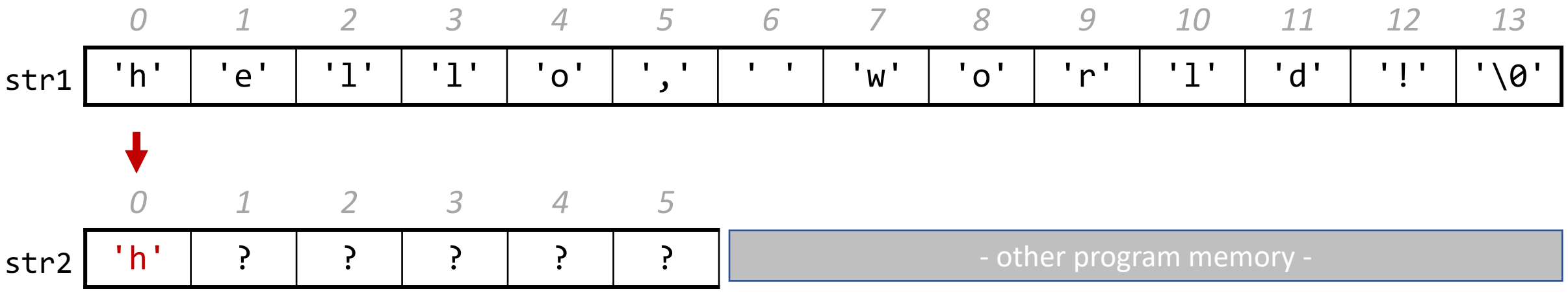
Copying Strings – Buffer Overflows

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



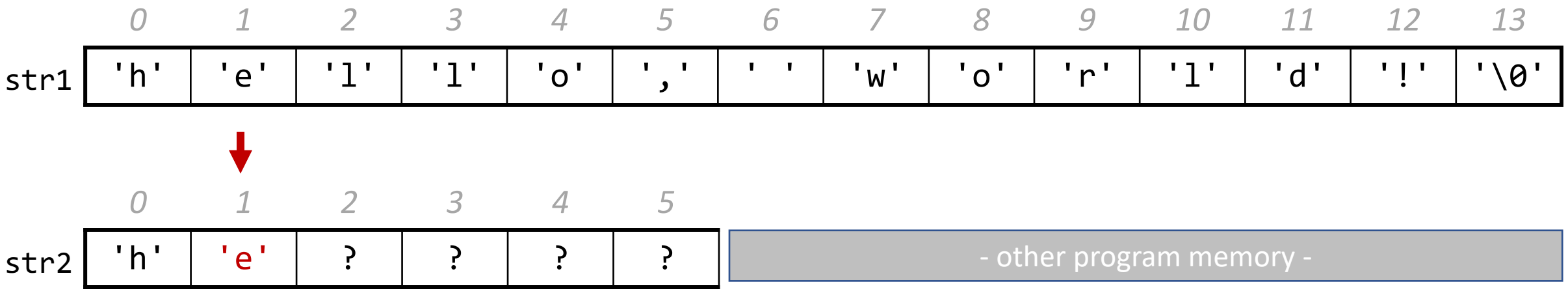
Copying Strings – Buffer Overflows

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



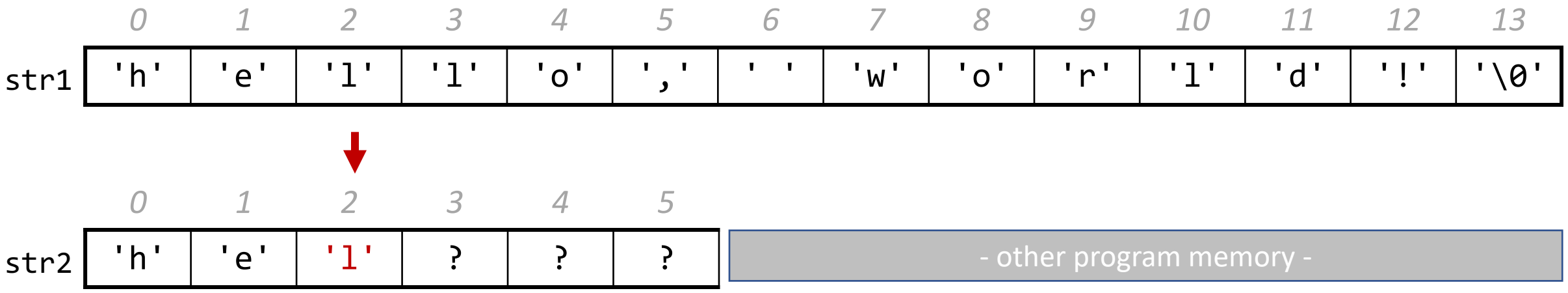
Copying Strings – Buffer Overflows

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



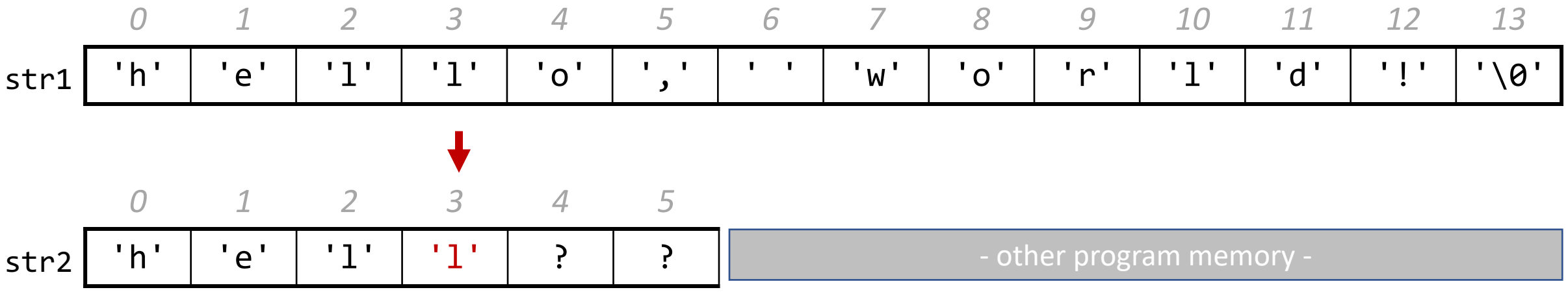
Copying Strings – Buffer Overflows

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



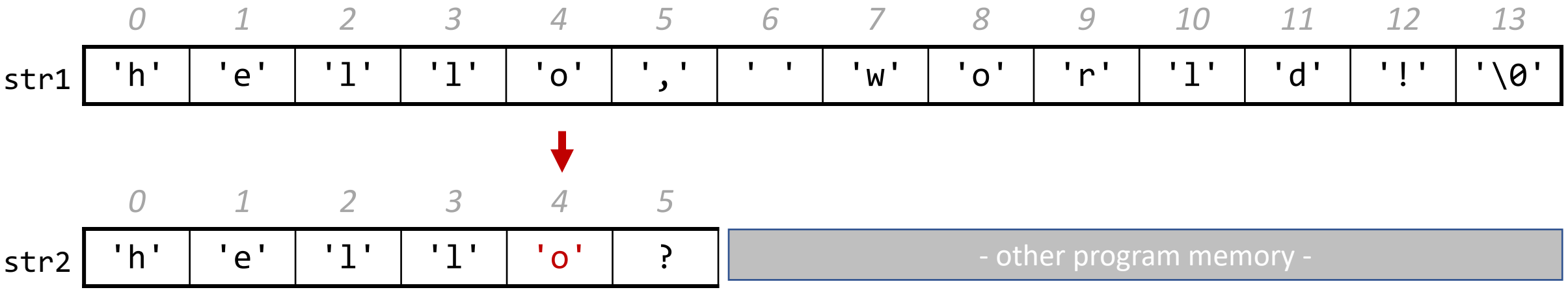
Copying Strings – Buffer Overflows

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



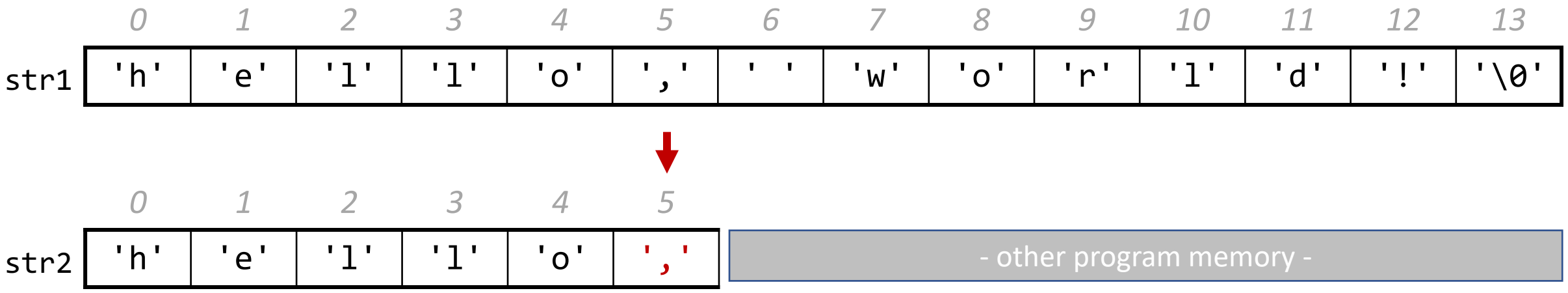
Copying Strings – Buffer Overflows

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



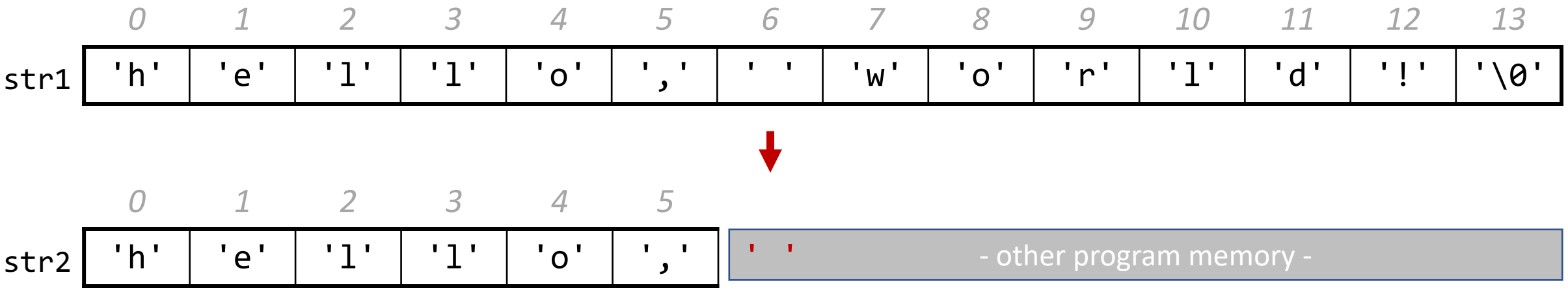
Copying Strings – Buffer Overflows

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



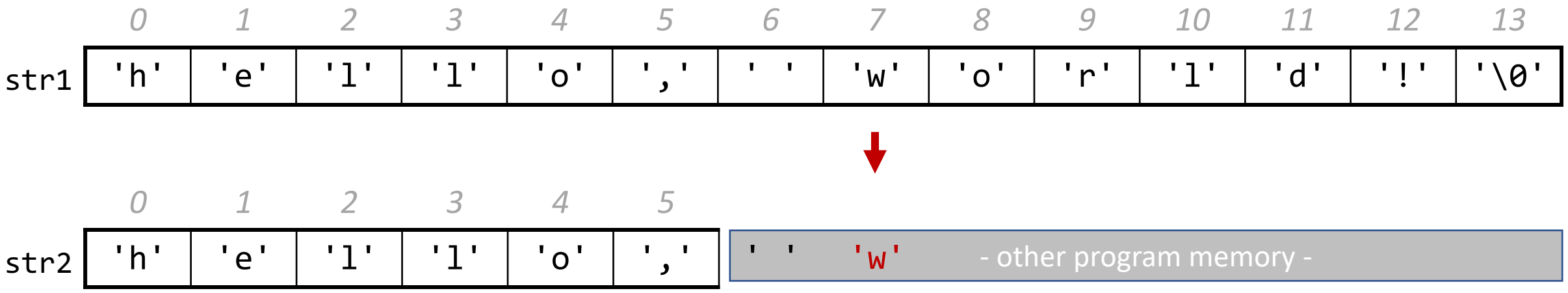
Copying Strings – Buffer Overflows

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



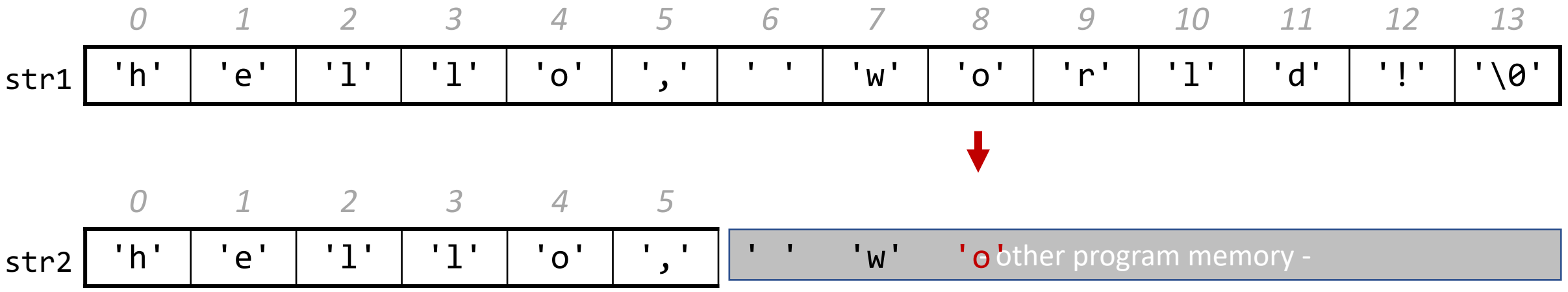
Copying Strings – Buffer Overflows

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



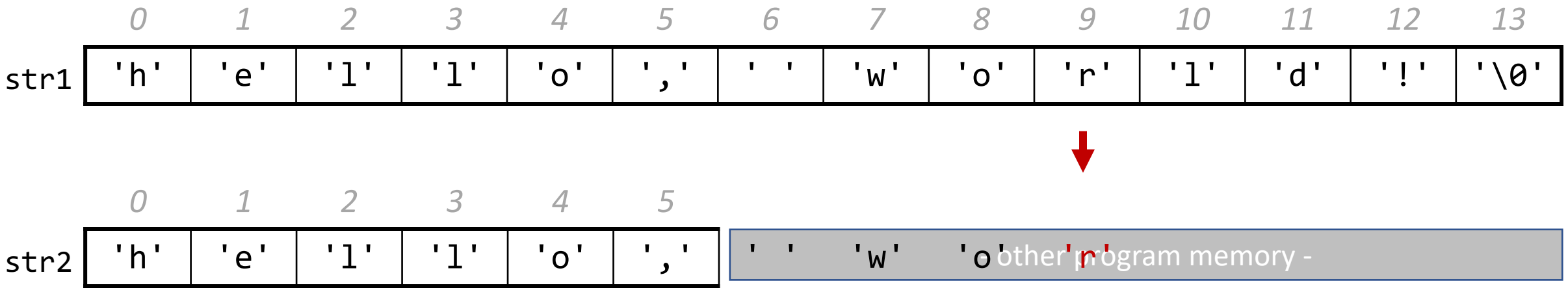
Copying Strings – Buffer Overflows

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



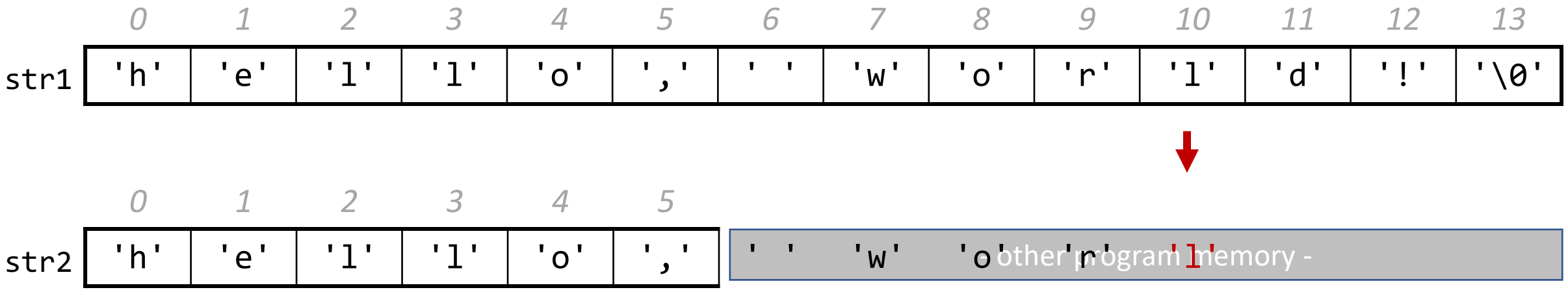
Copying Strings – Buffer Overflows

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



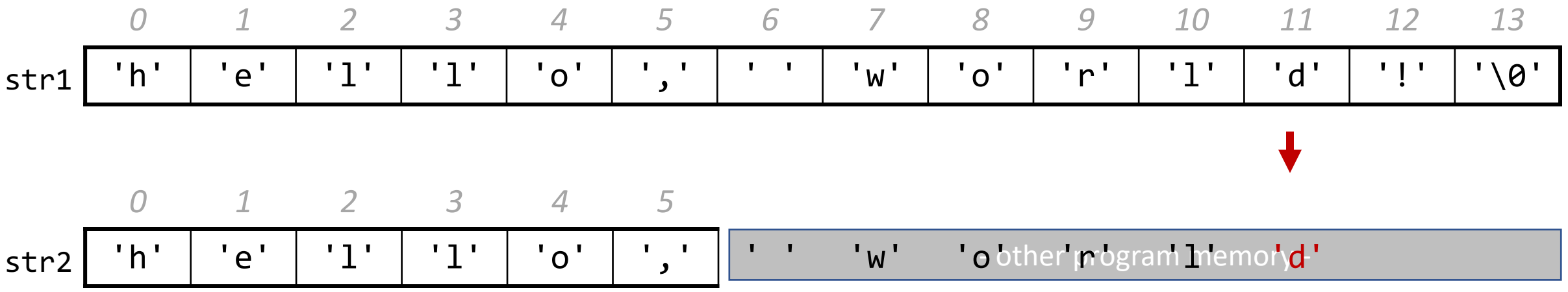
Copying Strings – Buffer Overflows

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



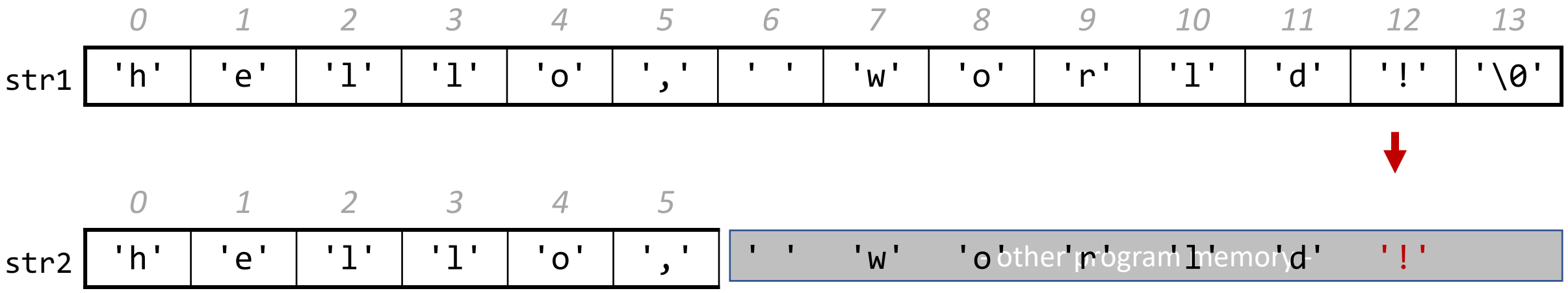
Copying Strings – Buffer Overflows

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



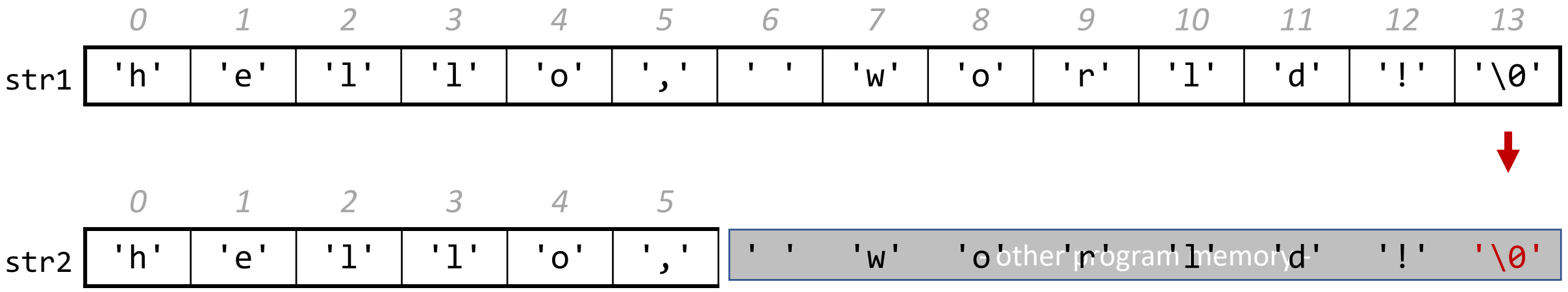
Copying Strings – Buffer Overflows

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



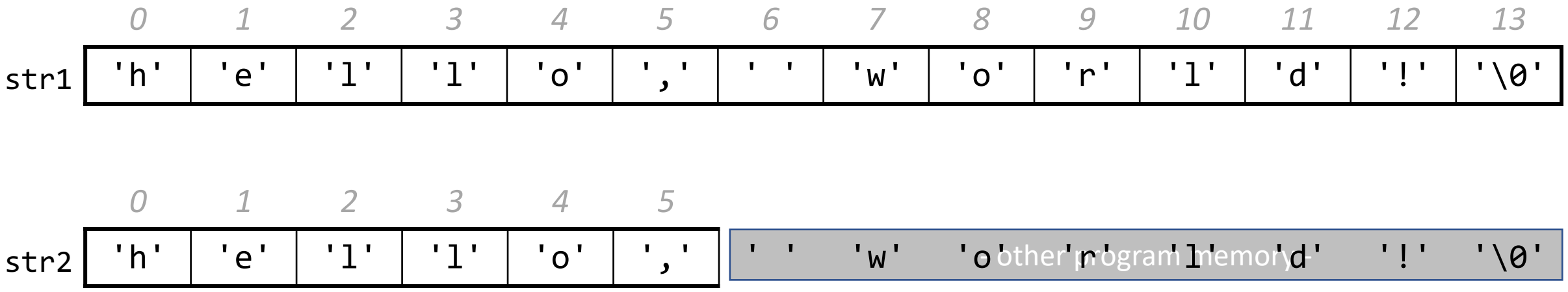
Copying Strings – Buffer Overflows

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



Copying Strings – Buffer Overflows

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



Copying Strings - `strncpy`

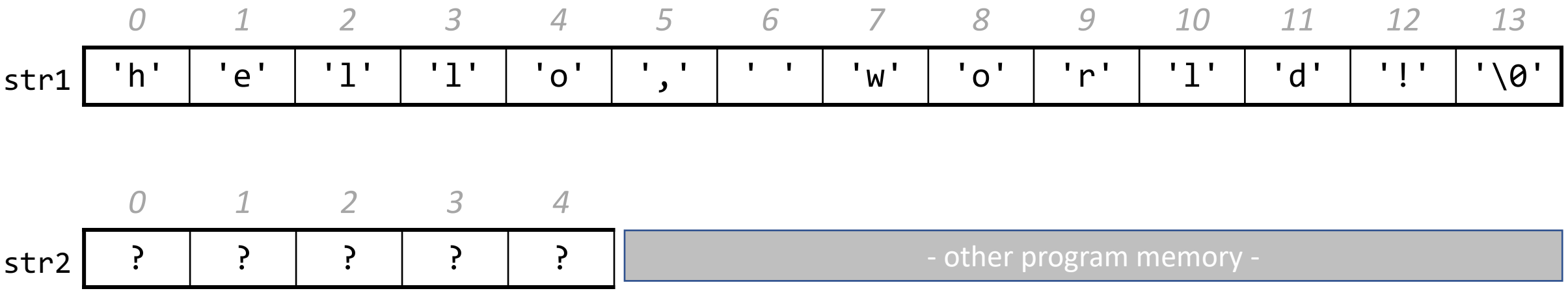
`strncpy(dst, src, n)`: copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **`dst`** will *not be null terminated*!

```
// copying "hello"
char str2[5];
strncpy(str2, "hello, world!", 5);     // doesn't copy '\0'!
```

If there is no null-terminating character, we may not be able to tell where the end of the string is anymore. E.g. `strlen` may continue reading into some other memory in search of `'\0'`!

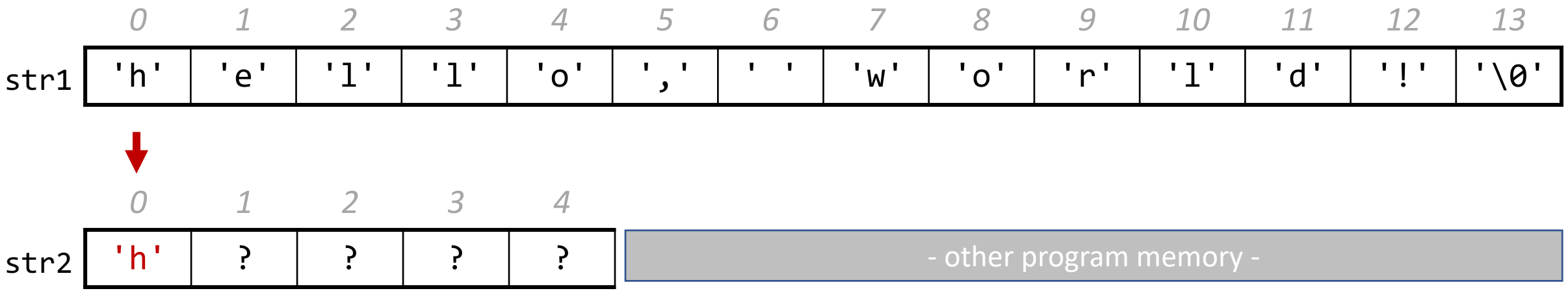
Copying Strings - strncpy

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



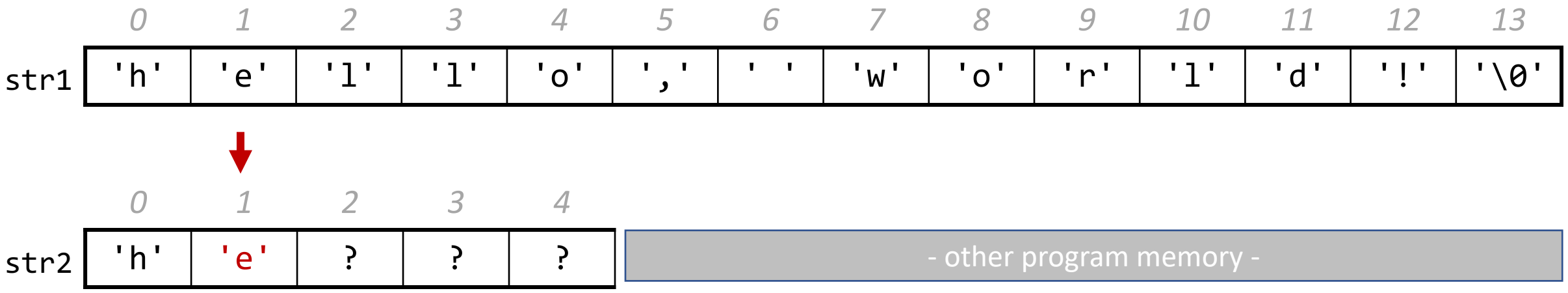
Copying Strings - strncpy

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



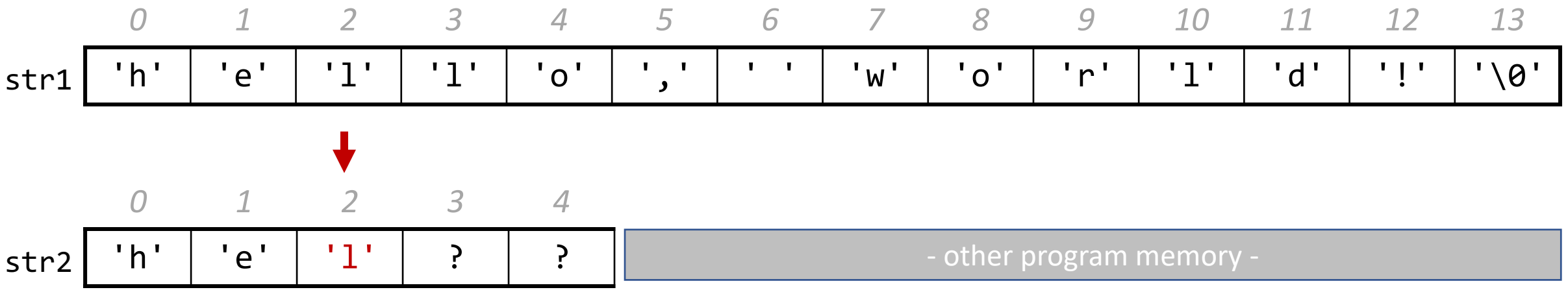
Copying Strings - strncpy

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



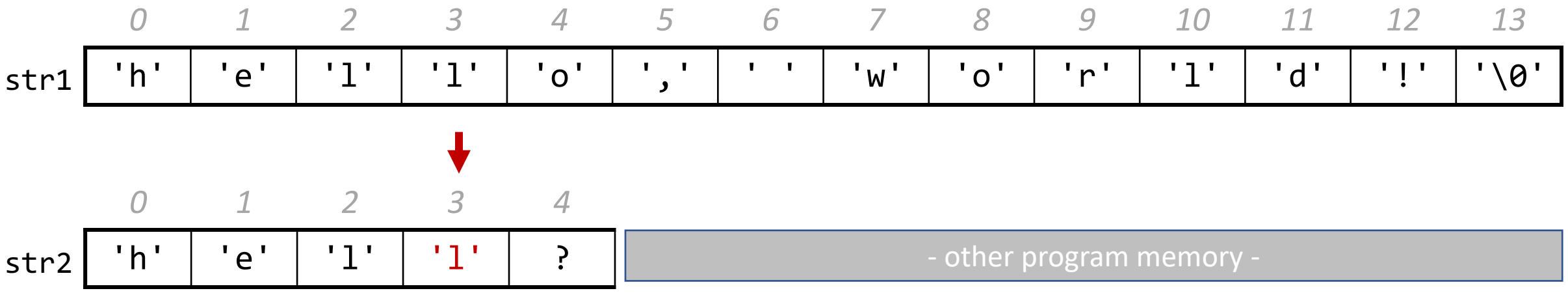
Copying Strings - strncpy

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



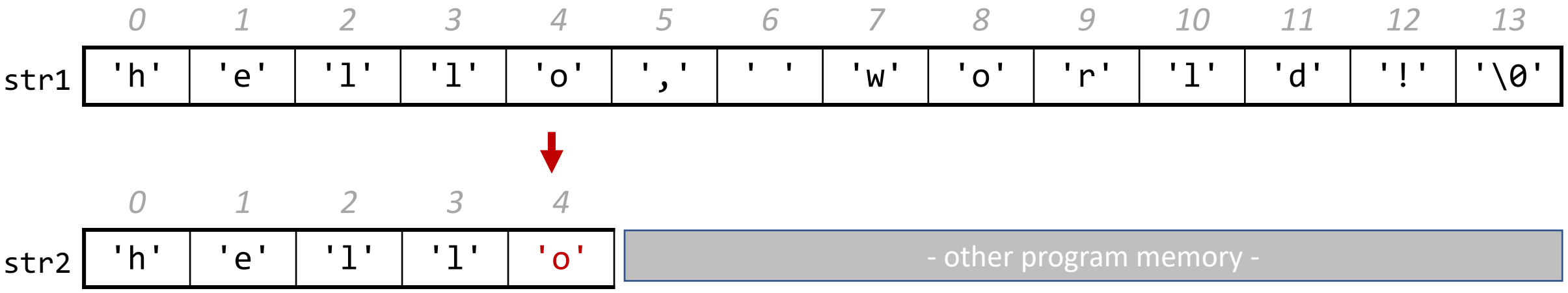
Copying Strings - strncpy

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



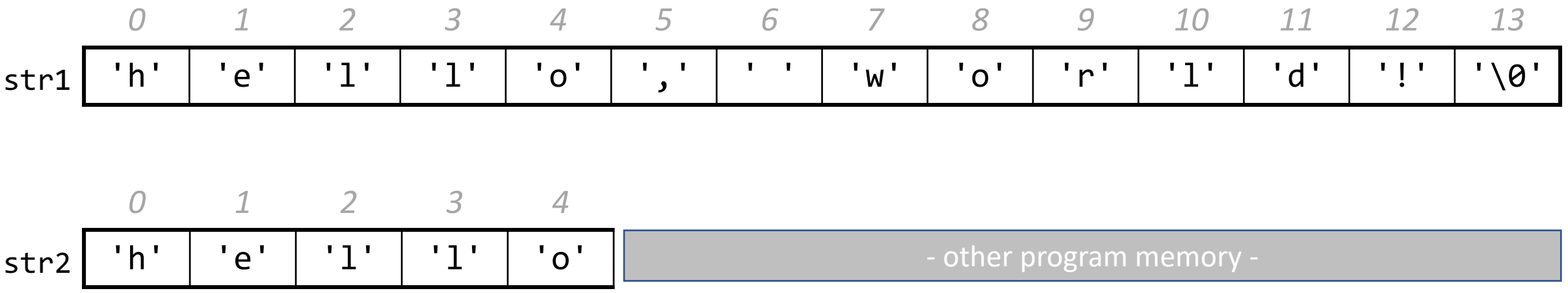
Copying Strings - strncpy

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



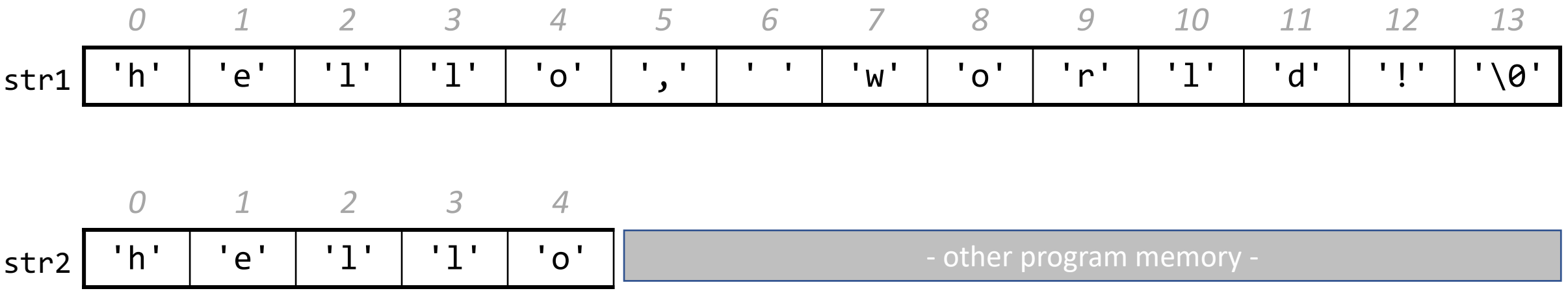
Copying Strings - strncpy

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



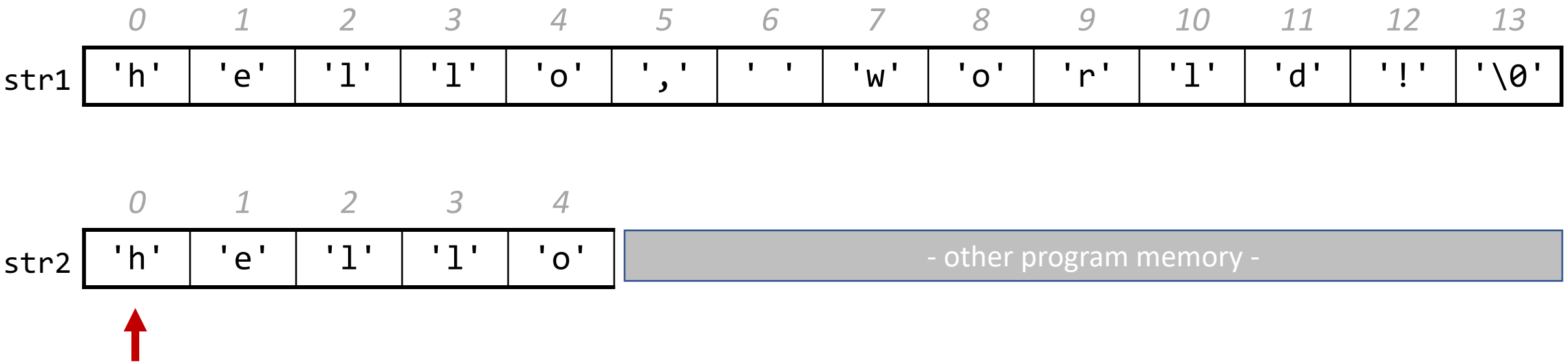
Copying Strings - strncpy

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



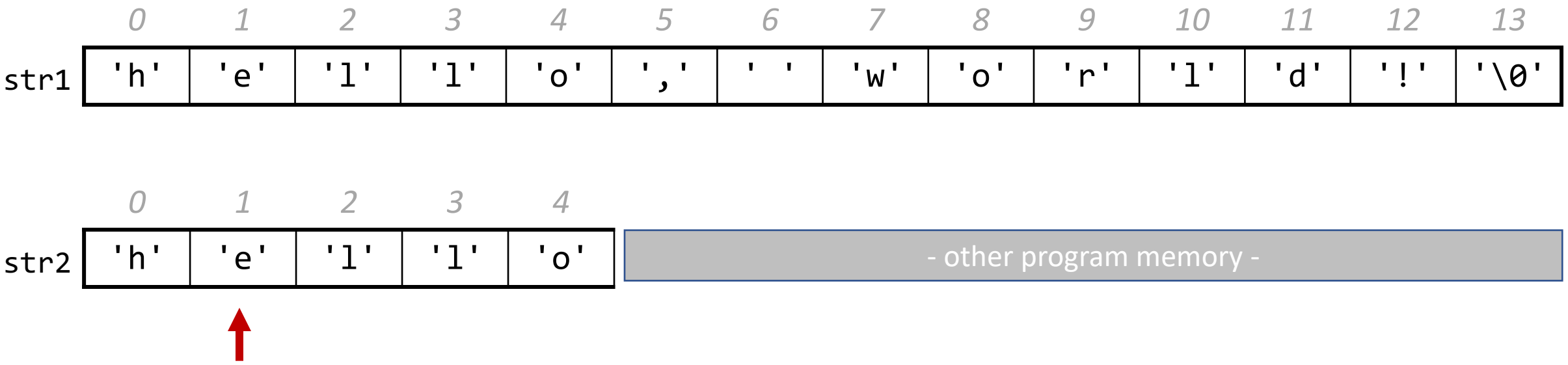
Copying Strings - strncpy

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



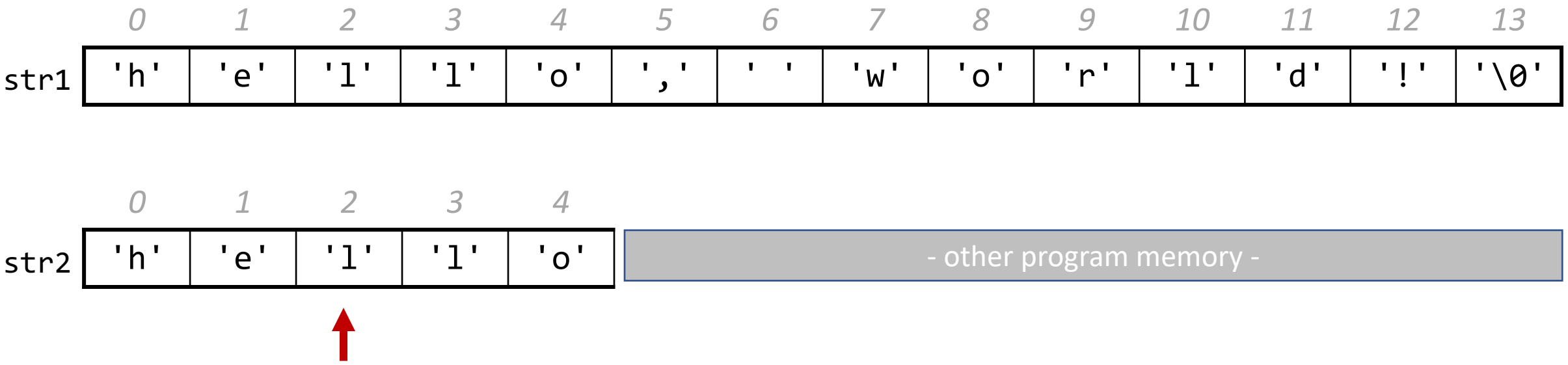
Copying Strings - strncpy

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



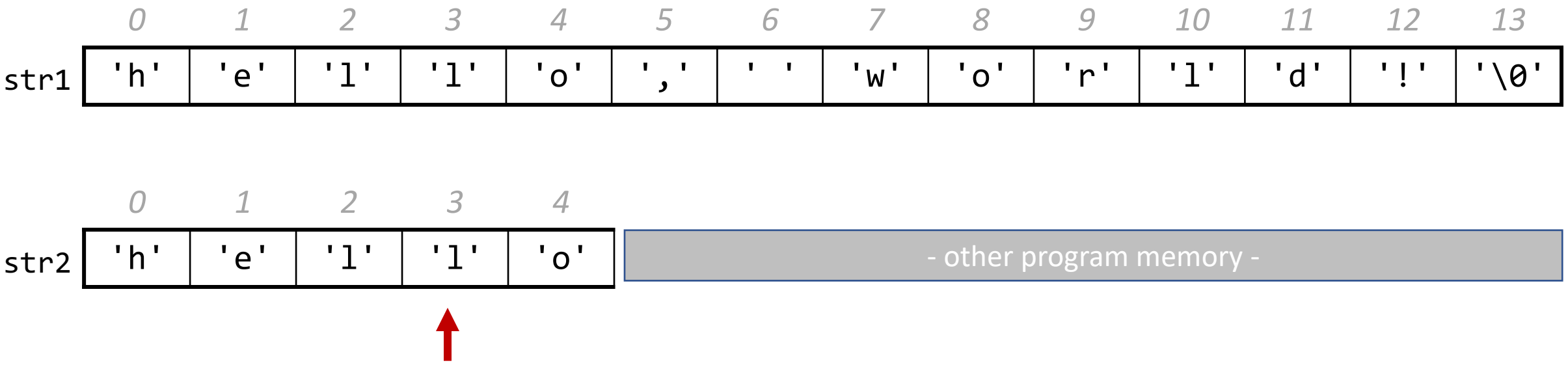
Copying Strings - strncpy

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



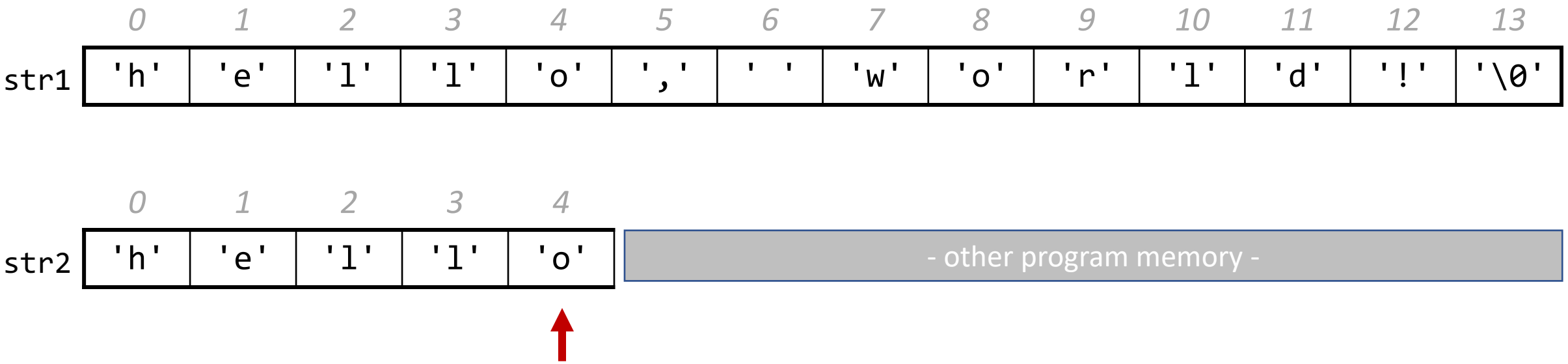
Copying Strings - strncpy

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



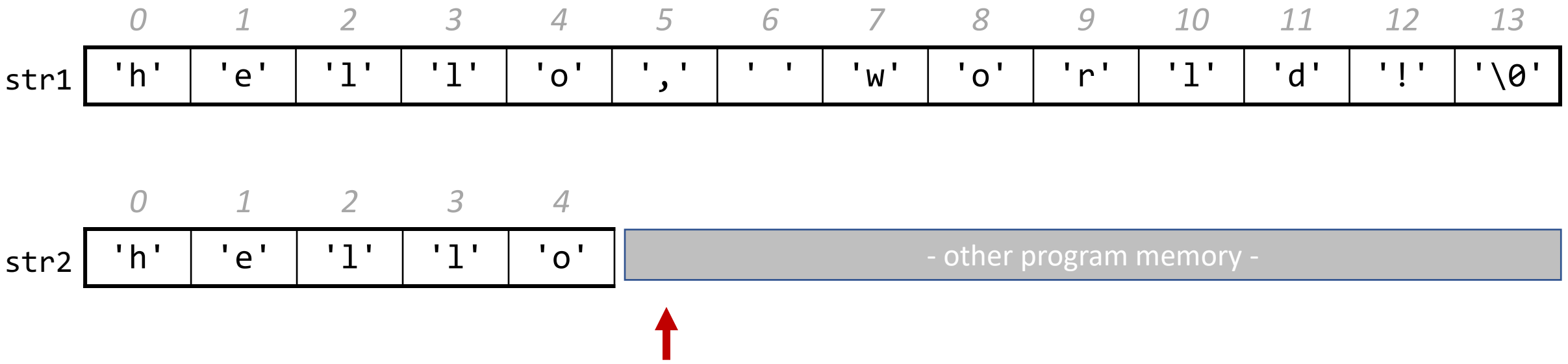
Copying Strings - strncpy

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



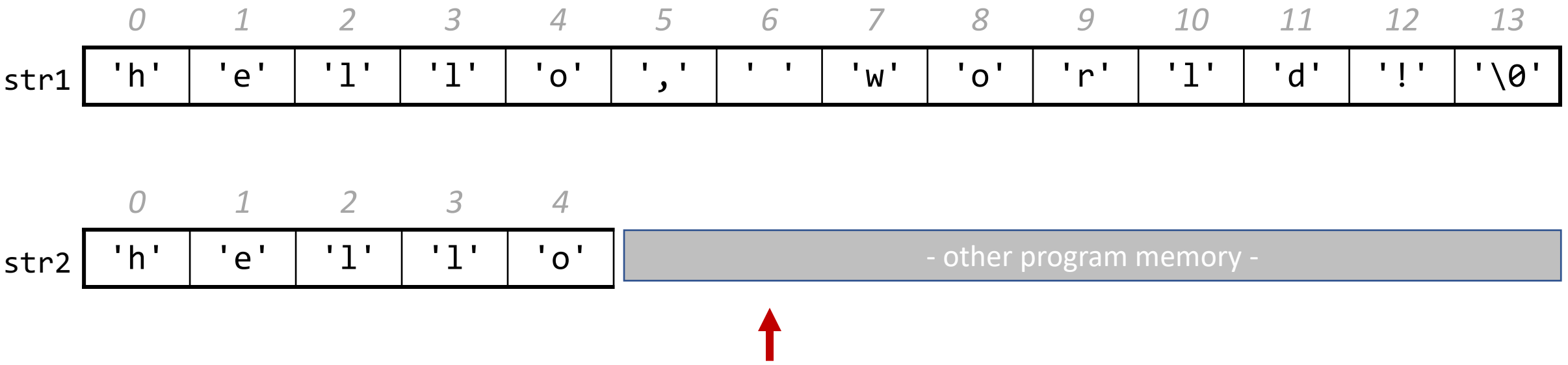
Copying Strings - strncpy

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



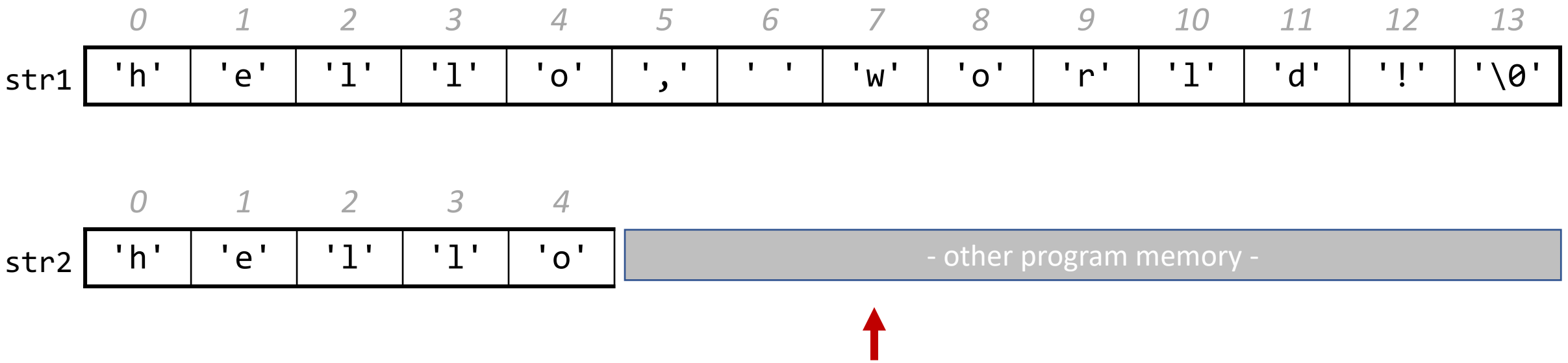
Copying Strings - strncpy

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



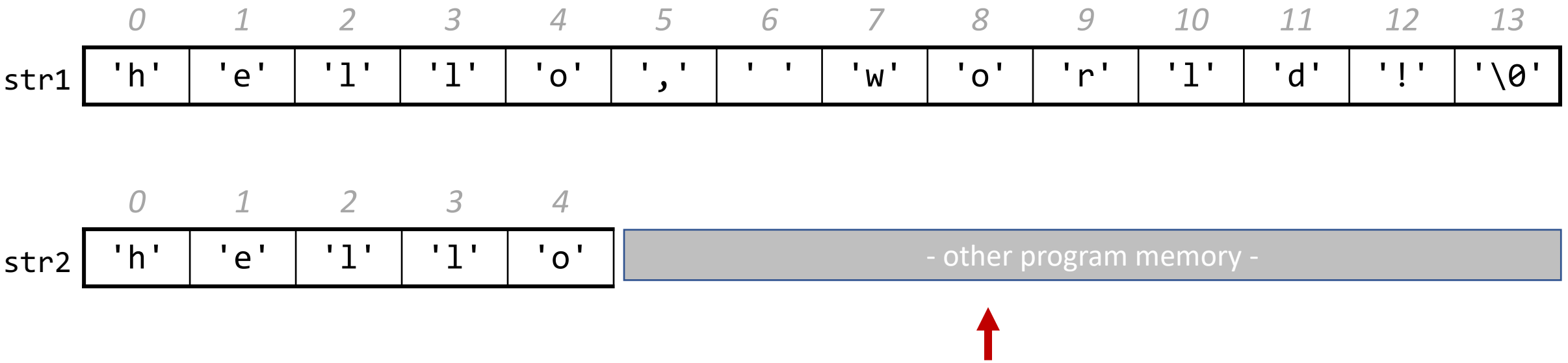
Copying Strings - strncpy

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



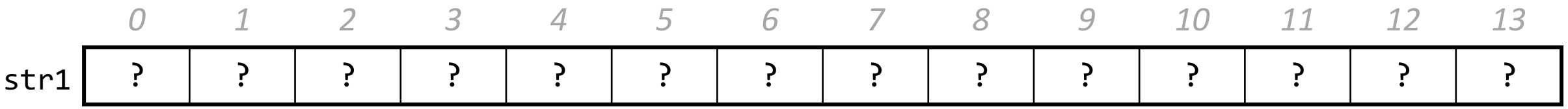
Copying Strings - strncpy

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



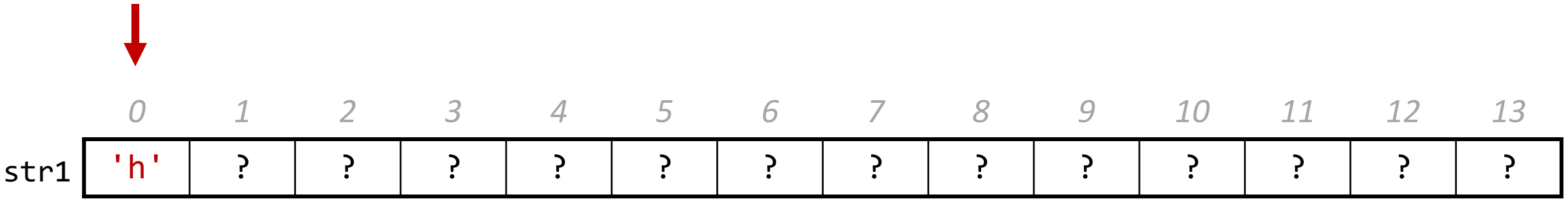
Copying Strings - strncpy

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



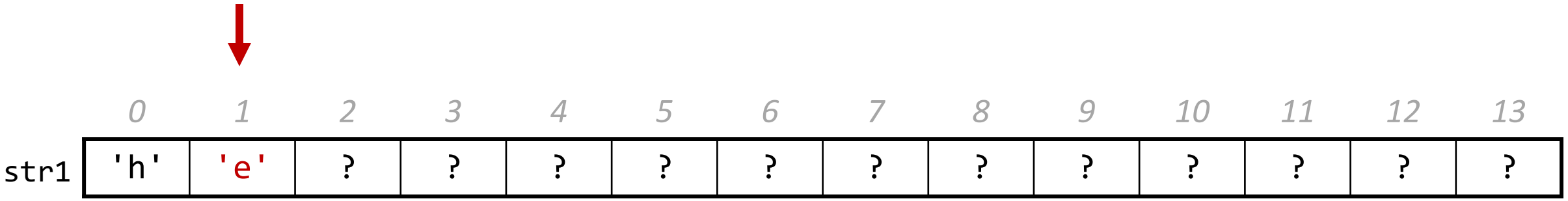
Copying Strings - strncpy

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



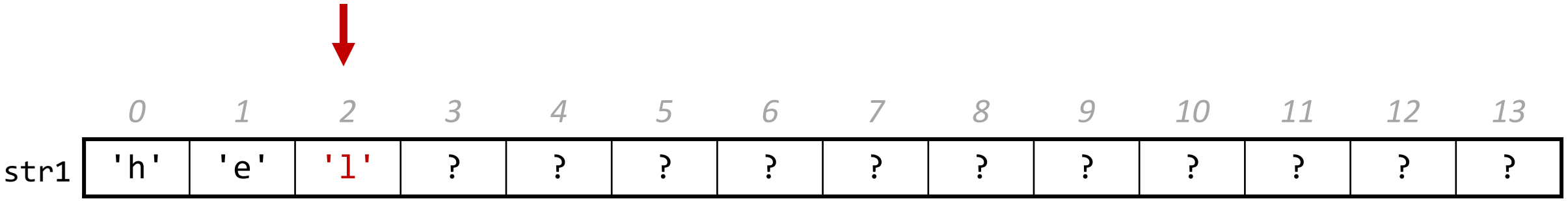
Copying Strings - strncpy

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



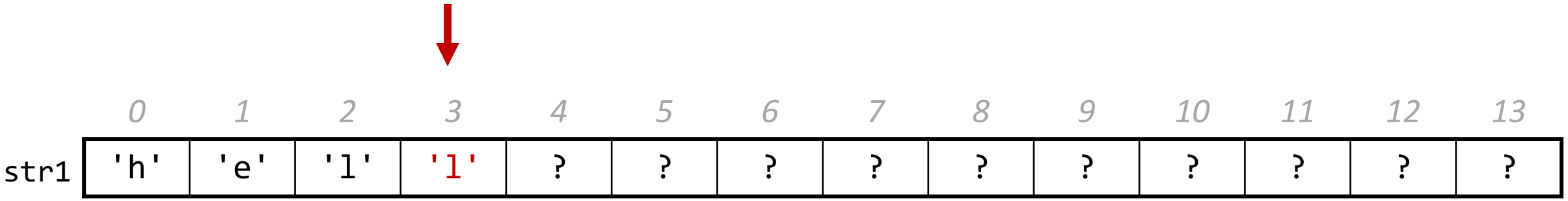
Copying Strings - strncpy

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



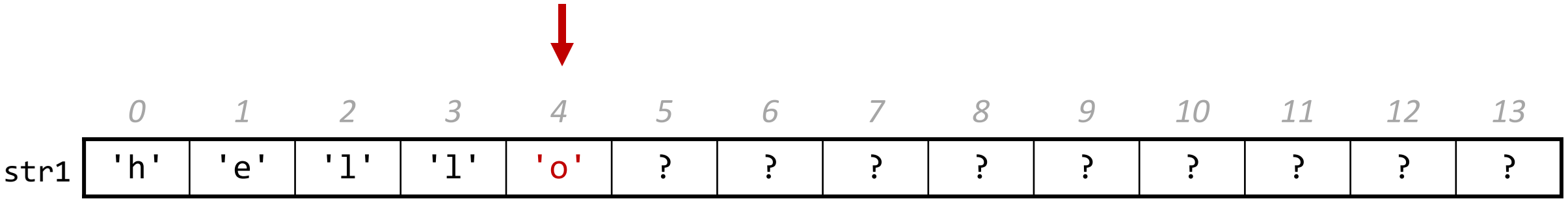
Copying Strings - strncpy

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



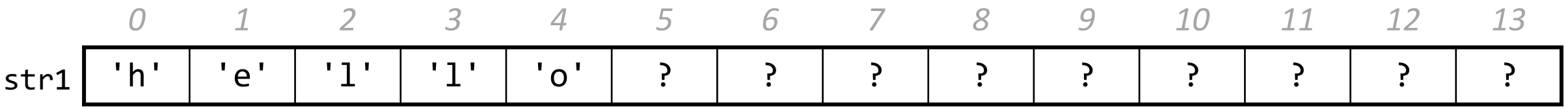
Copying Strings - strncpy

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



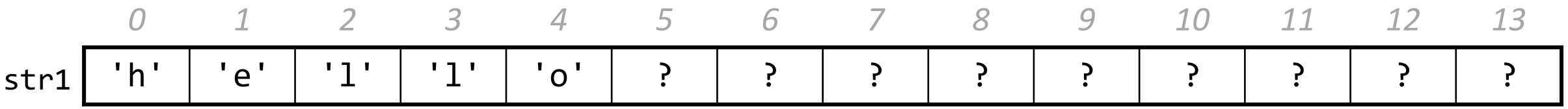
Copying Strings - strncpy

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



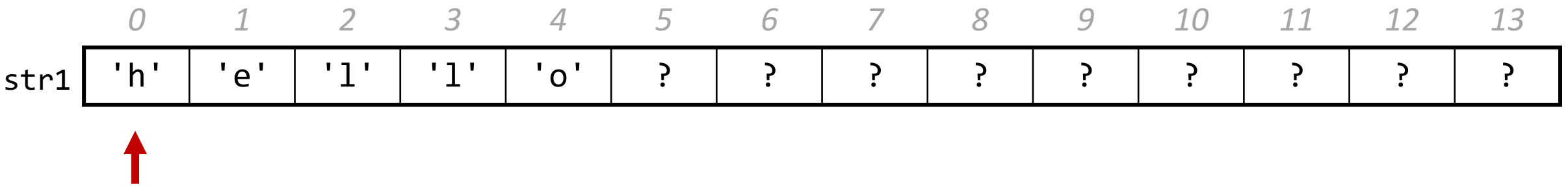
Copying Strings - strncpy

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



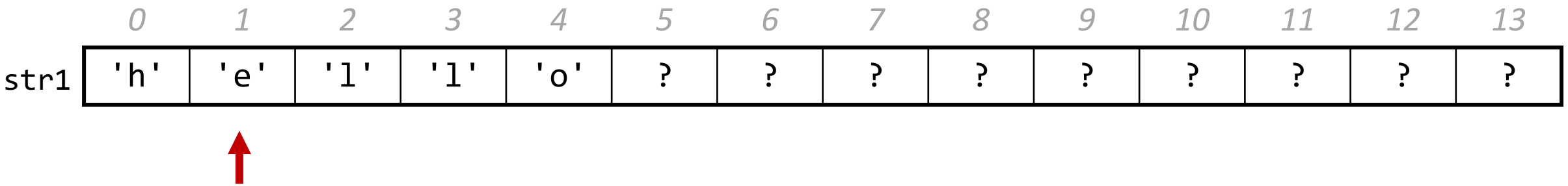
Copying Strings - strncpy

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



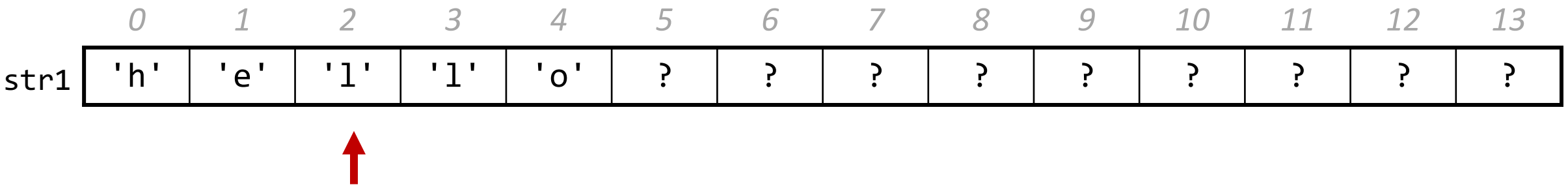
Copying Strings - strncpy

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



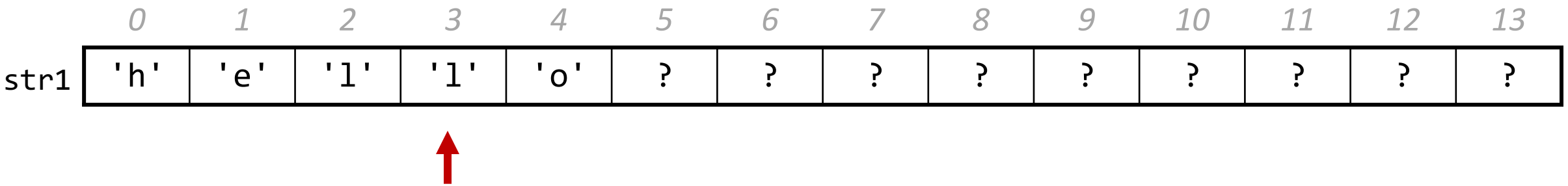
Copying Strings - strncpy

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



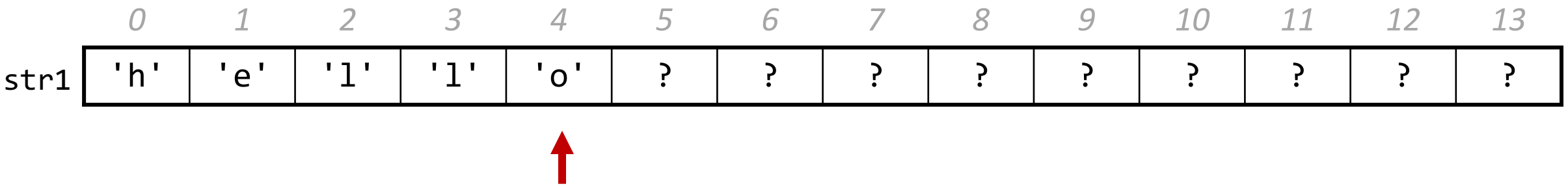
Copying Strings - strncpy

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



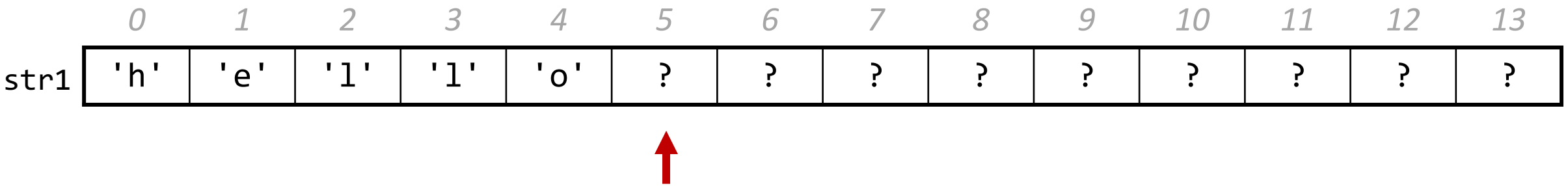
Copying Strings - strncpy

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



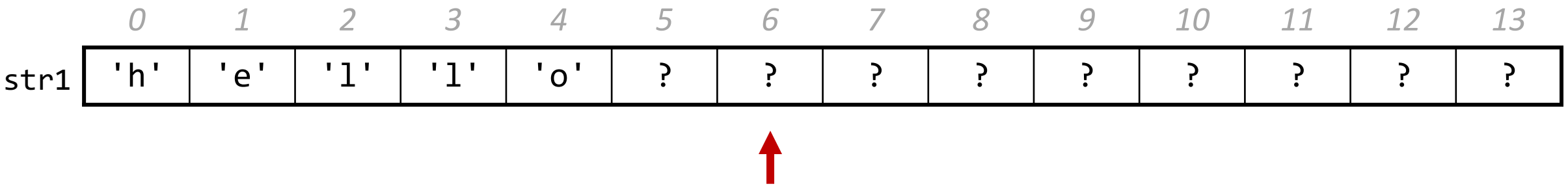
Copying Strings - strncpy

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



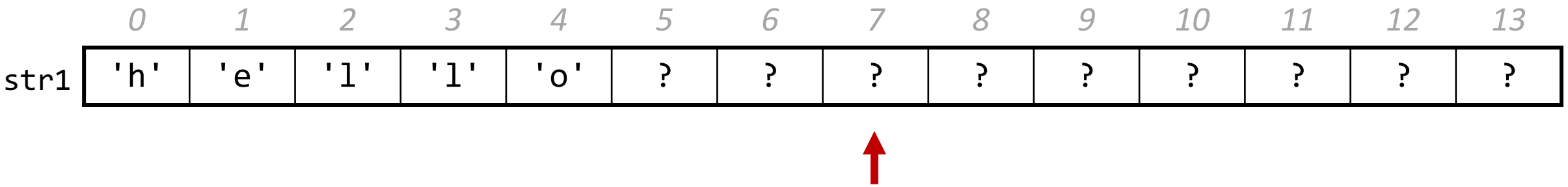
Copying Strings - strncpy

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



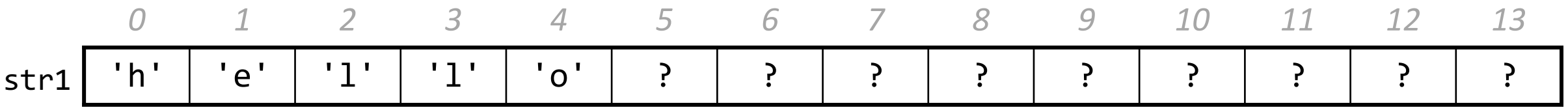
Copying Strings - strncpy

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



Copying Strings - strncpy

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




```
hello□□J□□□
```

Copying Strings - strncpy

If necessary, we can add a null-terminating character ourselves.

```
// copying "hello"
char str2[6];                          // room for string and '\0'
strncpy(str2, "hello, world!", 5);     // doesn't copy '\0'!
str2[5] = '\0';                        // add null-terminating char
```

String Copying Exercise

What value should go in the blank at right?

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

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

String Exercise

What is printed out by the following program?

```
1 int main(int argc, char *argv[]) {
2     char str[9];
3     strcpy(str, "Hi earth");
4     str[2] = '\0';
5     printf("str = %s, len = %lu\n",
6            str, strlen(str));
7     return 0;
8 }
```

A. `str = Hi, len = 8`
B. `str = Hi, len = 2`
C. `str = Hi earth, len = 8`
D. `str = Hi earth, len = 2`
E. None/other

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**.

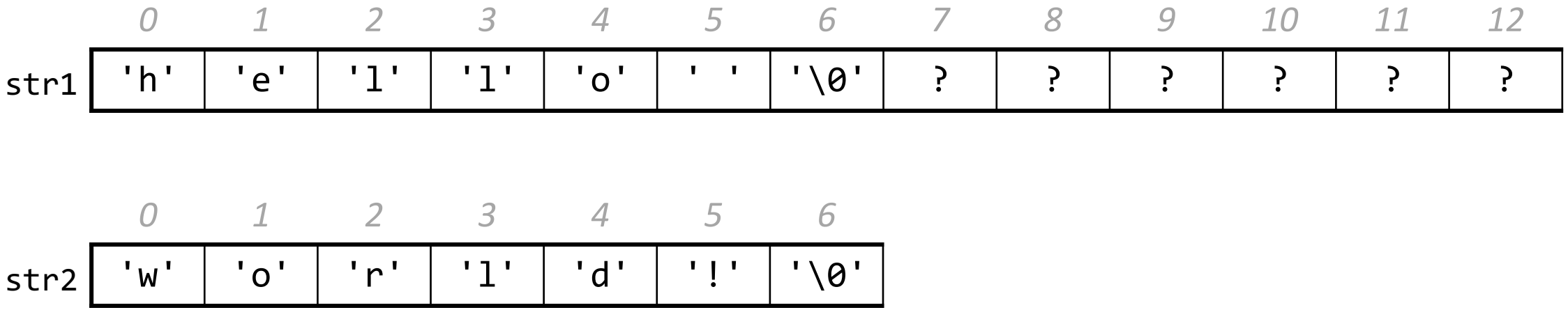
`strncat(dst, src, n)`: same, but concats at most n bytes from **src**.

```
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.

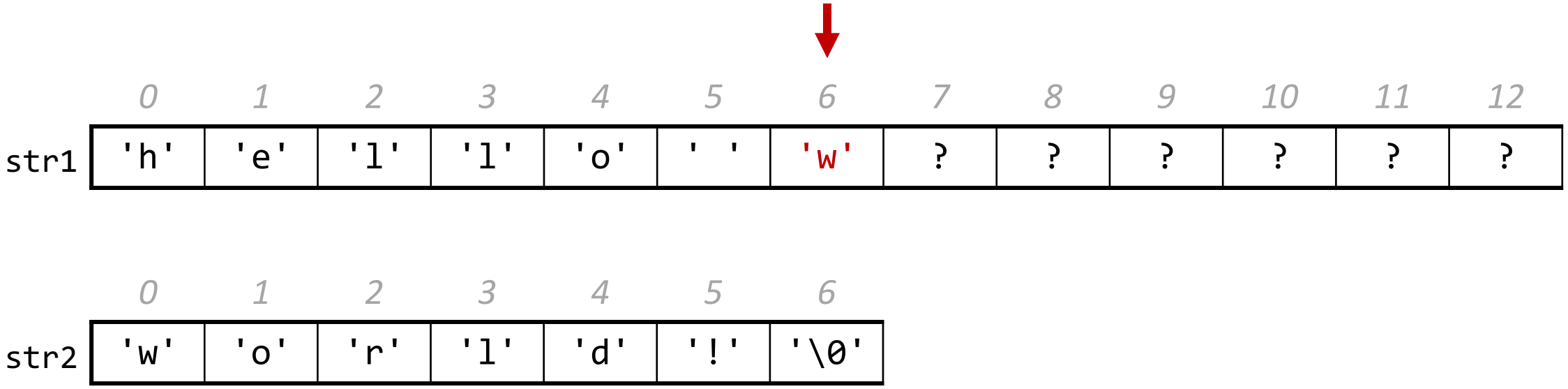
Concatenating Strings

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

strcat(str1, str2);
```



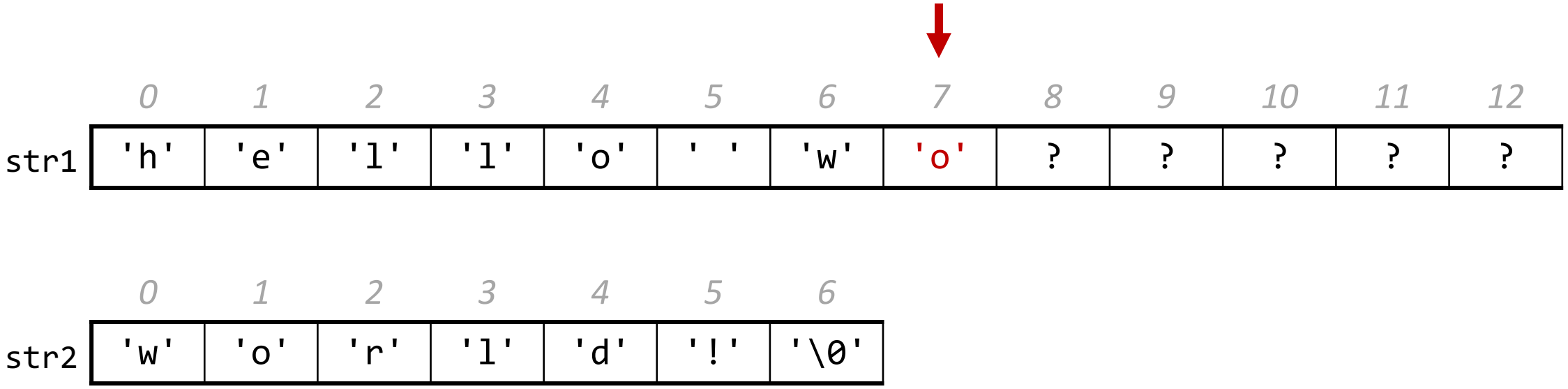
Concatenating Strings

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

strcat(str1, str2);
```



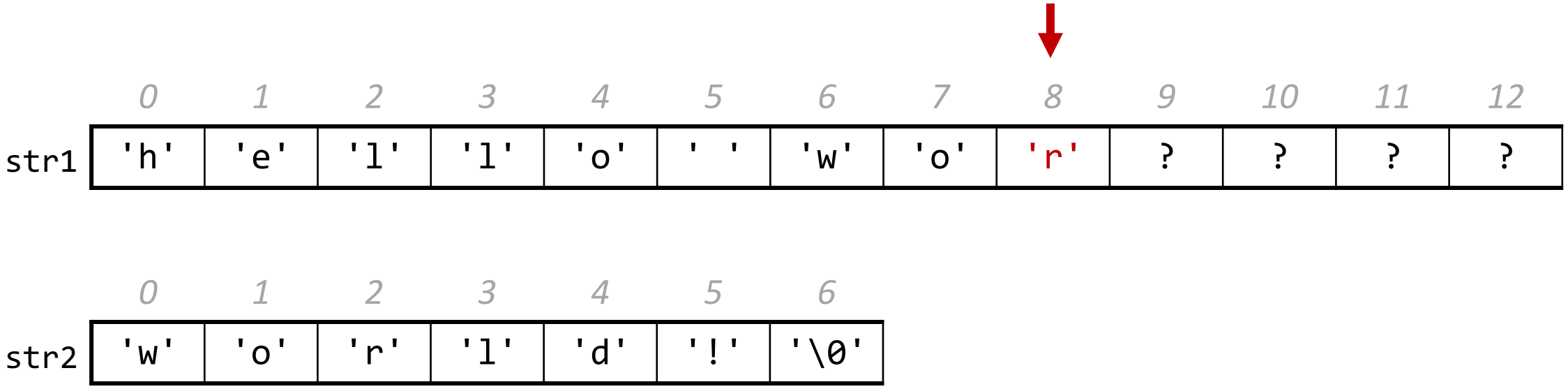
Concatenating Strings

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

strcat(str1, str2);
```



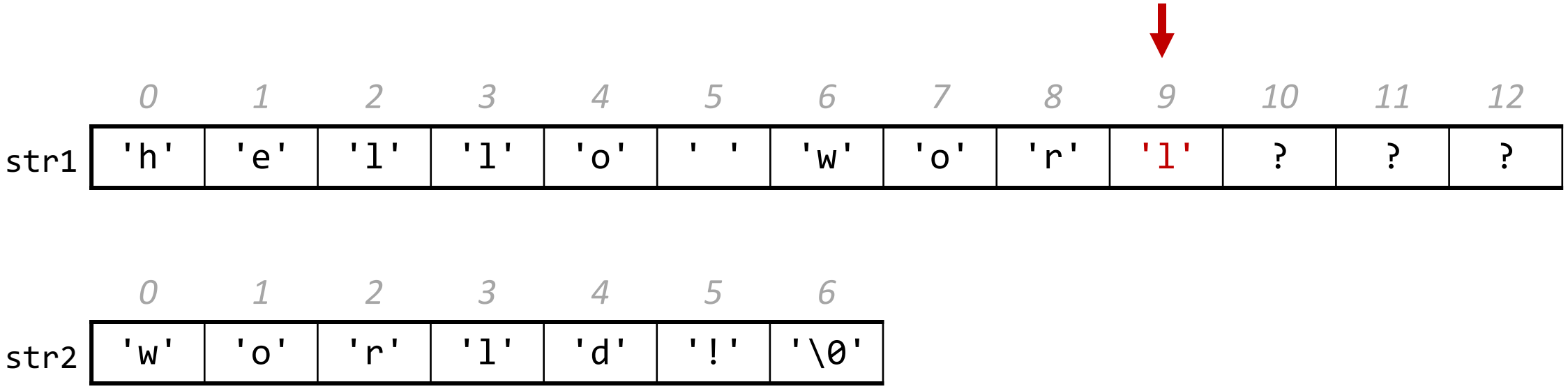
Concatenating Strings

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

strcat(str1, str2);
```



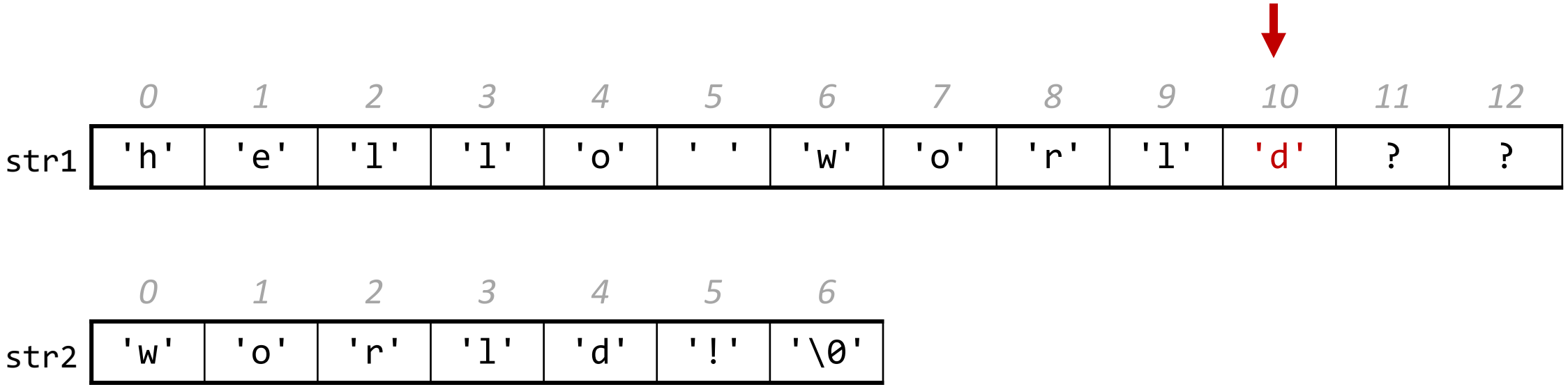
Concatenating Strings

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

strcat(str1, str2);
```



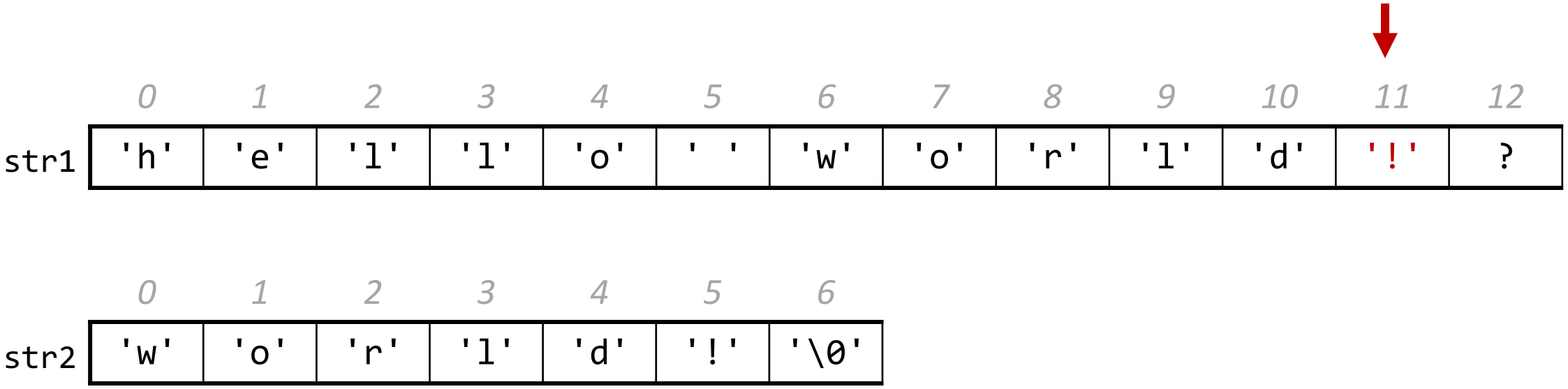
Concatenating Strings

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

strcat(str1, str2);
```



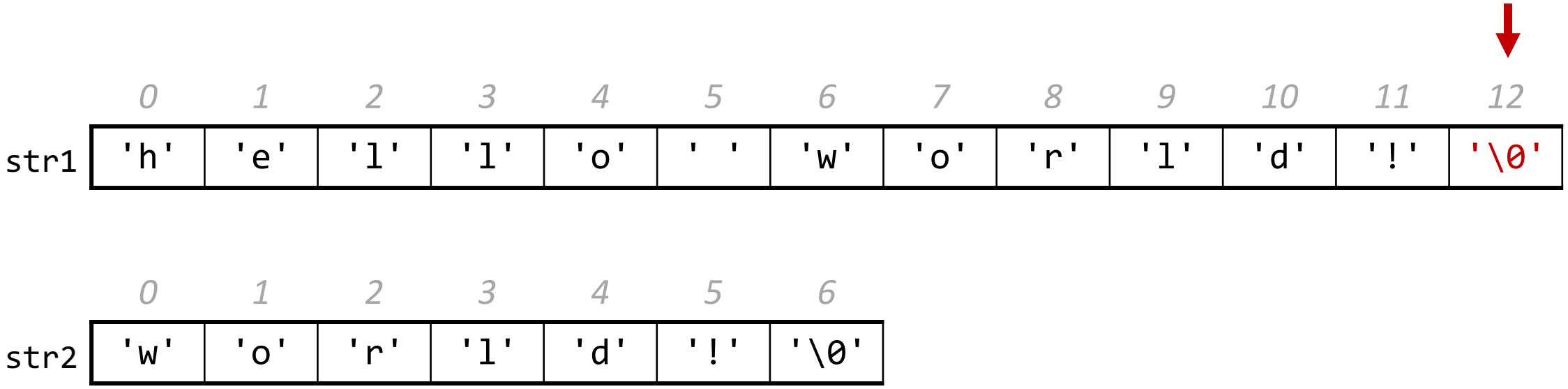
Concatenating Strings

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

strcat(str1, str2);
```



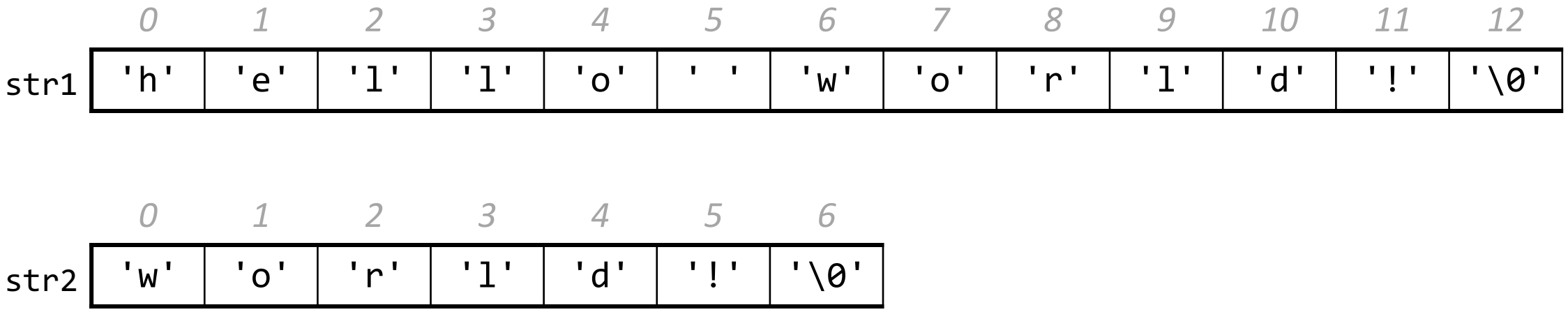
Concatenating Strings

```
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);
```



Substrings and char *

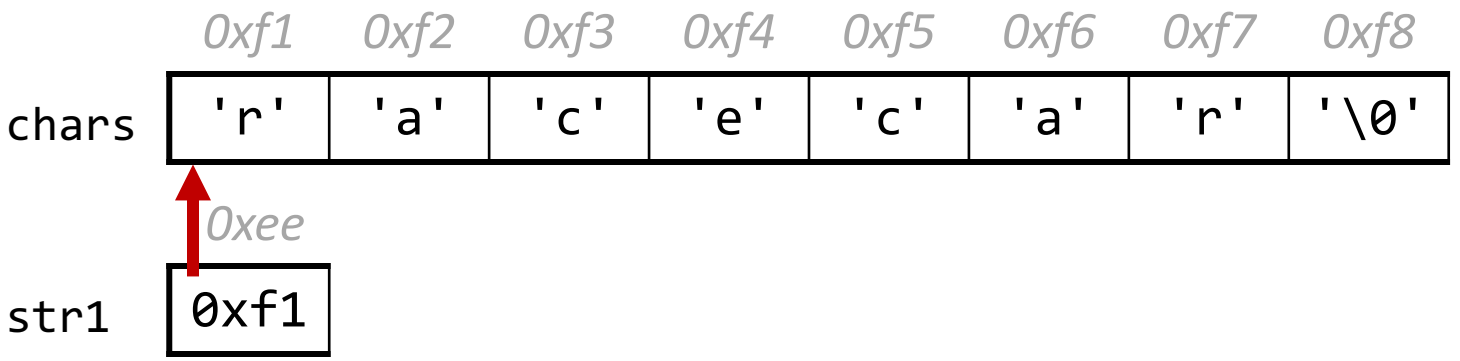
You can also create a char * variable yourself that points to an address within in an existing string.

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

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

Substrings

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

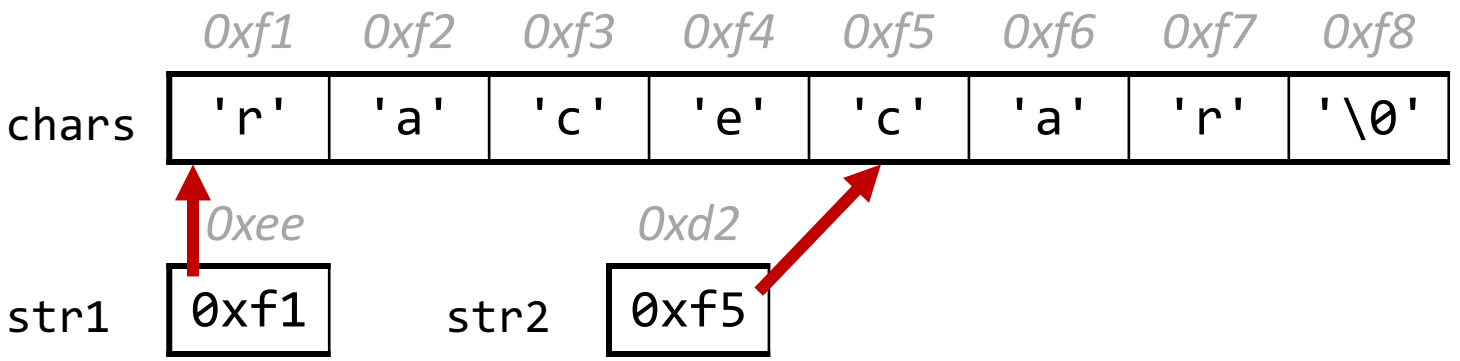
```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
```



Substrings

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

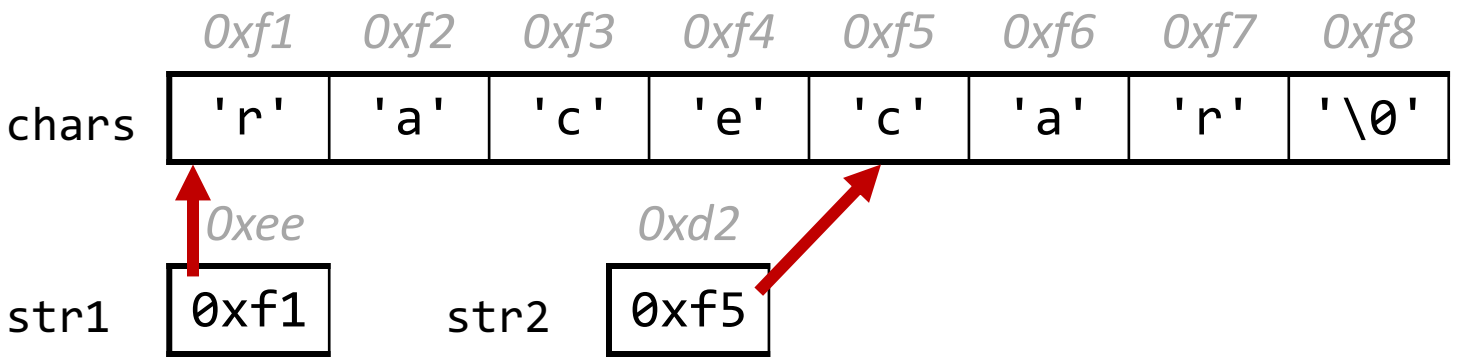
```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
```



Substrings

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

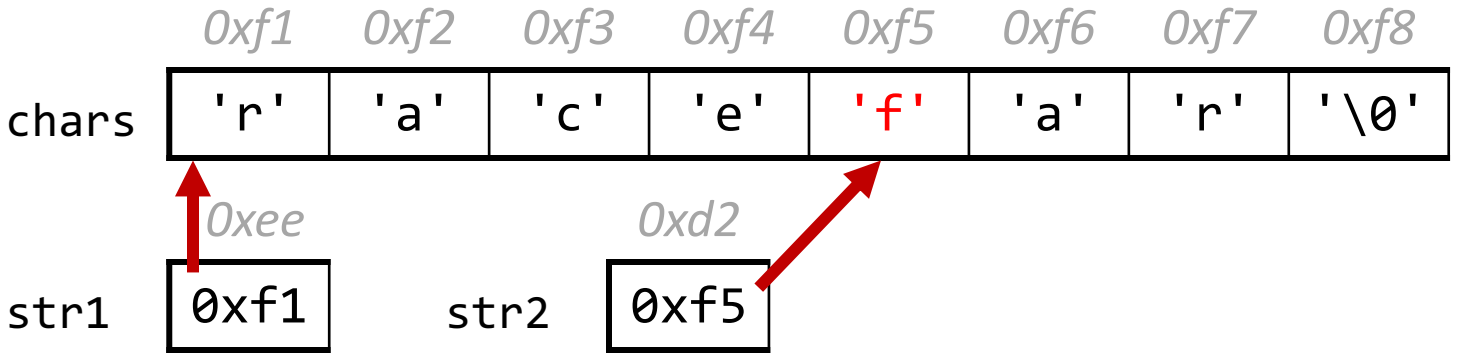
```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // car
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

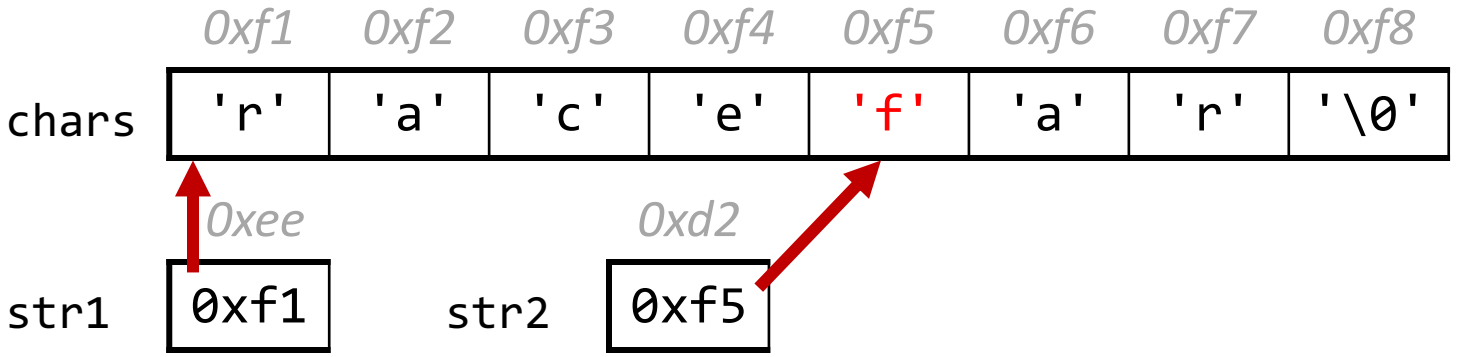
```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1);
printf("%s\n", str2);
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

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



char * vs. char[]

```
char myString[]
```

vs

```
char *myString
```

You can create char * pointers to point to any character in an existing string and reassign them since they are just pointer variables. You **cannot** reassign an array.

```
char myString[6];
strcpy(myString, "Hello");
myString = "Another string";                    // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;                  // ok
```

Substrings

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

```
// Want just "race"
char str1[8];
strcpy(str1, "racecar");

char str2[5];
strncpy(str2, str1, 4);
str2[4] = '\0';
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // race
```

Substrings

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

```
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

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

Lecture Plan

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

String Diamond

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

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

String Diamond

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

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

Daisy!

Practice: Diamond

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

Recap

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

Next time: more strings

Extra Practice

char * vs. char[]

- We'll talk more about char * vs char[] in lecture 5
- Some useful distinctions in the meantime:
  - 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)

String copying exercise

```
1 char buf[ ____ ];
2 strcpy(buf, "Chadwick");
3 printf("%s\n", buf);
4 char *word = buf + 2;
5 strncpy(word, "ris", 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("Chadwick")
F. Something else

Line 6: What is printed?

A. risick
B. risdwick
C. Chris
D. Chrisick
E. Something else
F. Compile error

String copying exercise

```
1 char buf[  9  ];
2 strcpy(buf, "Chadwick");
3 printf("%s\n", buf);
4 char *word = buf + 2;
5 strncpy(word, "ris", 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("Chadwick")`
F. Something else

Line 6: What is printed?

A. `risick`
B. `risdwick`
C. `Chris`
D. `Chrisick`
E. Something else
F. Compile error

Practice: Copycat

- How could we replace a call to **strcat** with a call to **strcpy** instead?
- **Challenge**: implement our own **mystrcat** using other string functions.

```
// assume enough space in dst
strcat(dst, src);

// same as
strcpy(dst + strlen(dst), src);
```