Pointer arithmetic & Strings

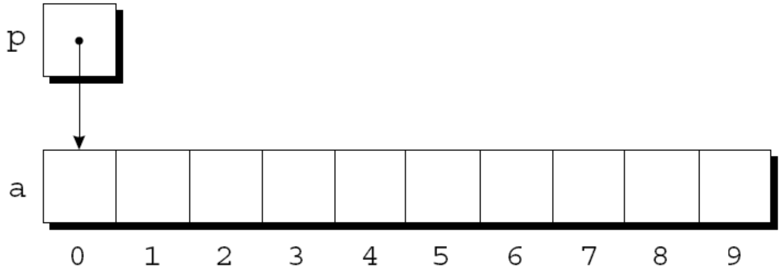
CS 210 Fall 2023

Vasiliki Kalavri

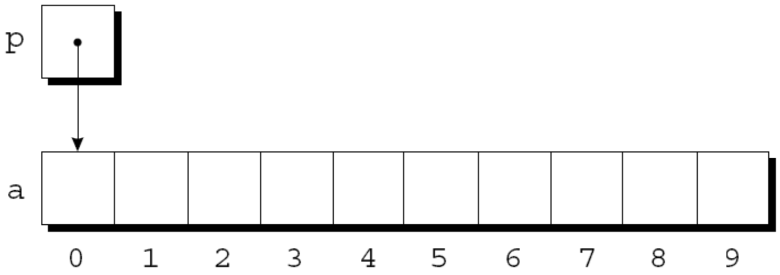
Pointers and Arrays are closely related in C

- Any operation that can be performed using array subscripting can also be performed using pointer arithmetic

```
int a[10], *p;
p = &a[0];
```

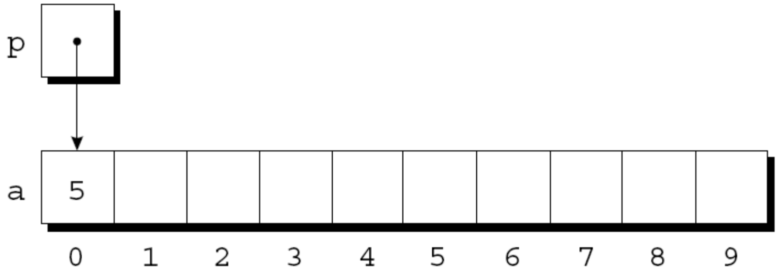


```
int a[10], *p;
p = &a[0];
```



We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```



Pointer arithmetic

- If p, q point to elements of the same array, then we can compare them using ==, !=, <, >= etc.
- `p < q` is true when p points to an array element before the one q points to.

Comparing pointers that don't point to elements of the same array has undefined behavior!

Adding/Subtracting an integer to/from a pointer

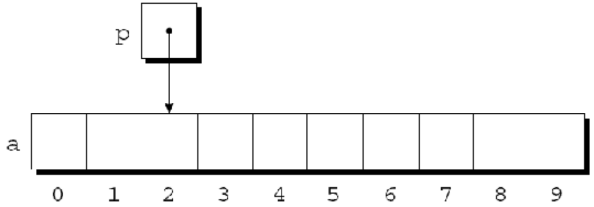
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.

- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.

- The computed value is scaled according to the size of the data type referenced by the pointer.

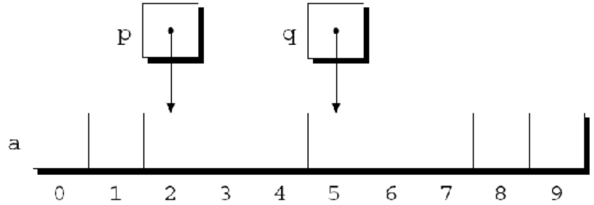
```
int a[10], *p, *q, i;
```

```
p = &a[2];
```



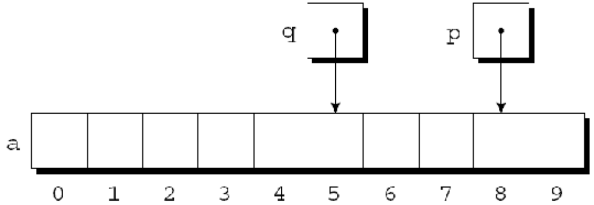
```
q = p + 3;
```

2+3=5

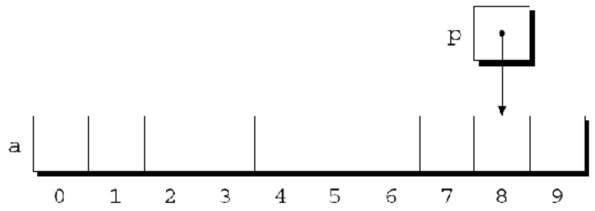


```
p += 6;
```

8

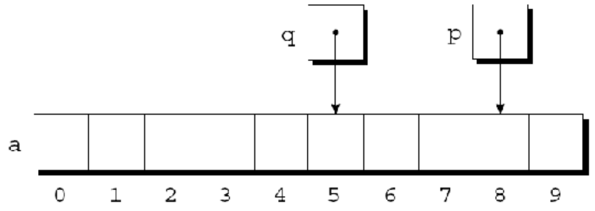


```
p = &a[8];
```



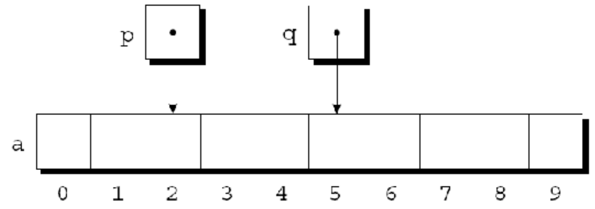
```
q = p - 3;
```

a[5]



```
p -= 6;
```

p= a[2]



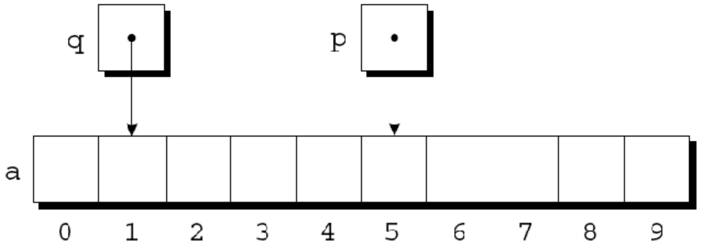
Subtracting a pointer from another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.

i - j

```
p = &a[5];

q = &a[1];
```

i = 4



```
i = p - q;    /* i is 4 */

i = q - p;    /* i is -4 */
```

-4

Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array `a`:

10

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
   sum += *p;
```

It’s legal to apply the address operator to the first element after the end of the array.

10

&a[N-1]

legal only this

before the array or after the array

Not

Using an array name as a pointer

The name of an array can be used as a pointer to the first element in the array.

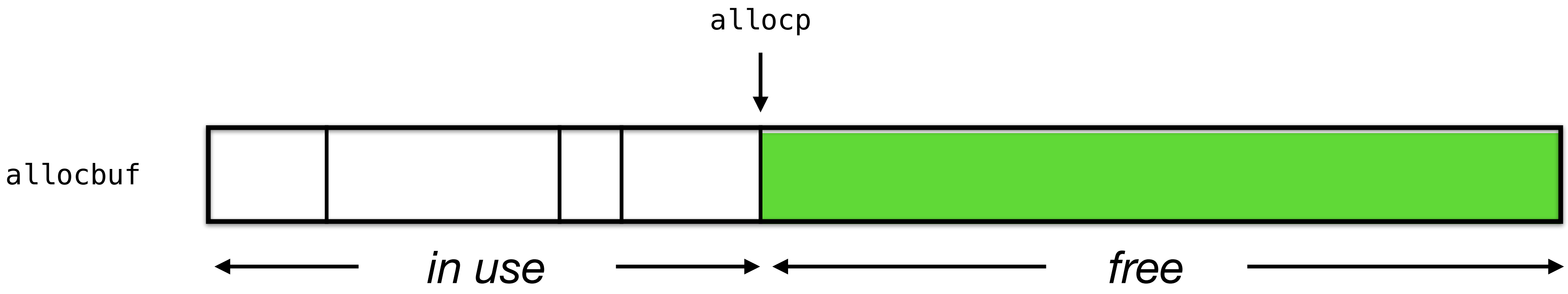
```
int a[10];
*a = 7;     /* stores 7 in a[0] */
*(a+1) = 12;    /* stores 12 in a[1] */
```

array는 그자체 pointer

- In general, `a + i` is the same as `&a[i]`.
  - Both represent a pointer to element `i` of `a`.
- Also, `*(a+i)` is equivalent to `a[i]`.
  - Both represent element `i` itself.

A simple memory manager

- We will implement two functions
  - `alloc(n)` returns a pointer `p` for `n` sequential `char` locations which can be used by `alloc`'s caller to store `char`s
  - `afree(p)` frees up memory allocated with `alloc(n)` and makes it available for use again
- For this simple version, we will assume that available memory is organized as a stack (Last-In-First-Out).

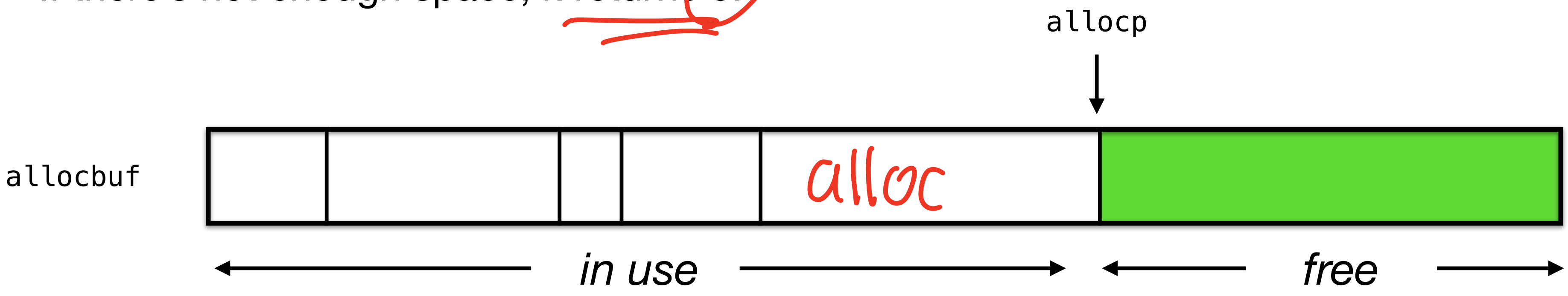


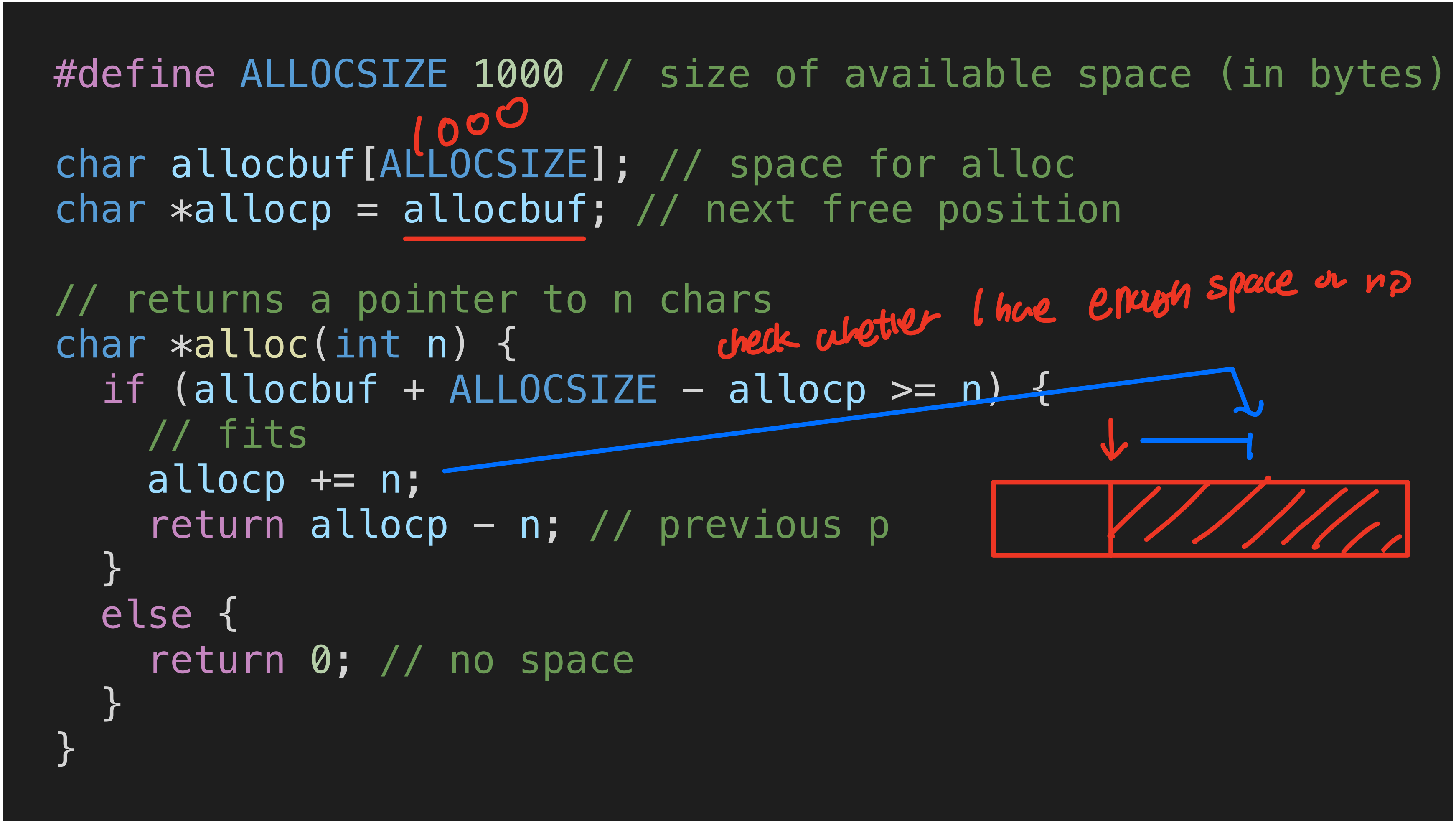
alloc(n) checks whether allocbuf has enough free space.

If it does, it returns the current value of allocp (start of free block) and then increases it to point to the next free block.

address

If there's not enough space, it returns 0.

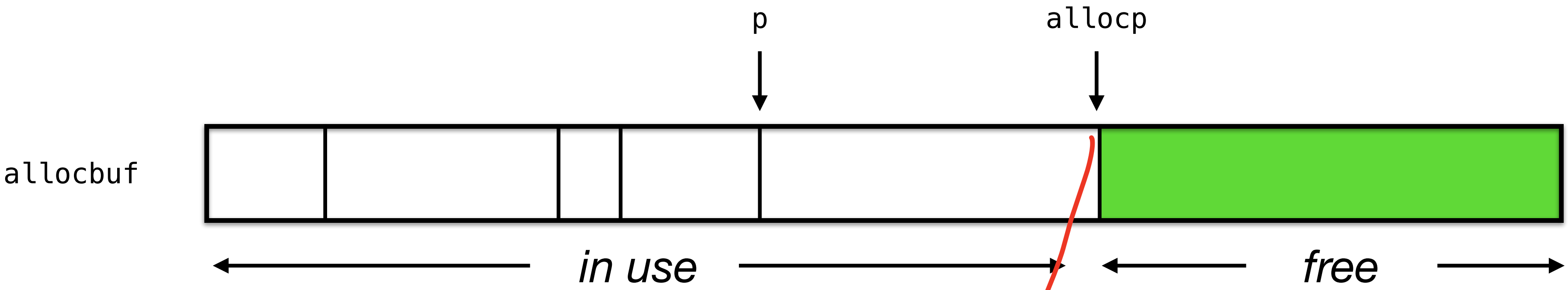




```
#define ALLOCSIZE 1000 // size of available space (in bytes)

char allocbuf[ALLOCSIZE]; // space for alloc
char *allocp = allocbuf; // next free position

// returns a pointer to n chars
char *alloc(int n) {
  if (allocbuf + ALLOCSIZE - allocp >= n) {
    // fits
    allocp += n;
    return allocp - n; // previous p
  }
  else {
    return 0; // no space
  }
}
```

1000

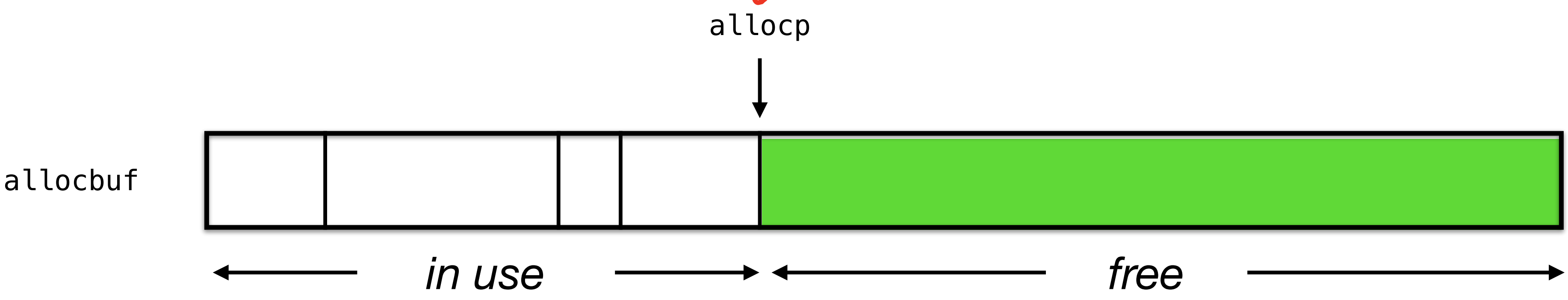
check whether I have enough space or no

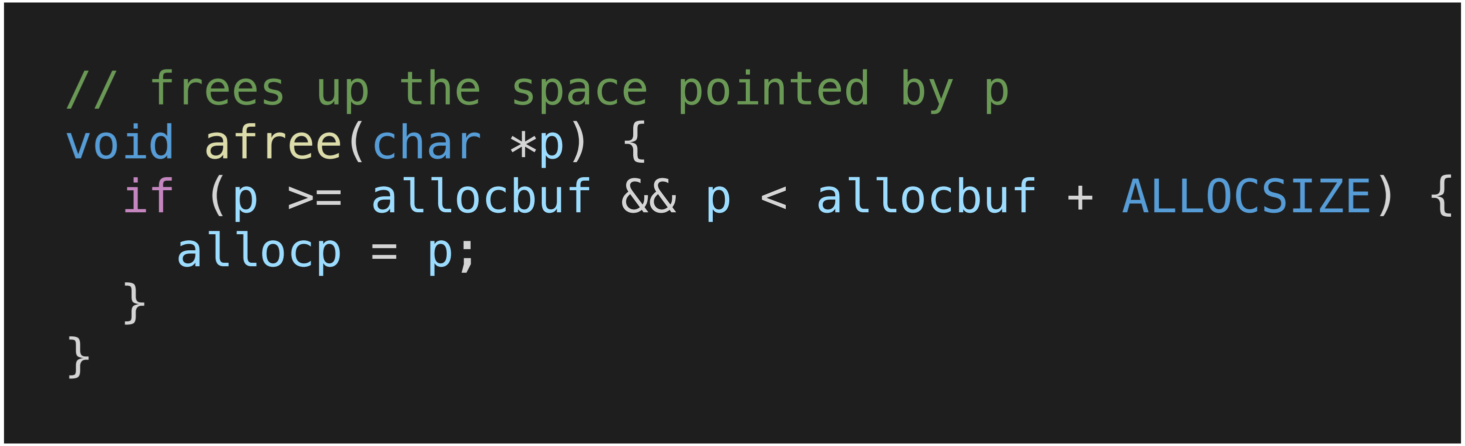


itfree(p) the mae allocp to p position

afree(p) checks whether p is pointing inside allocbuf.

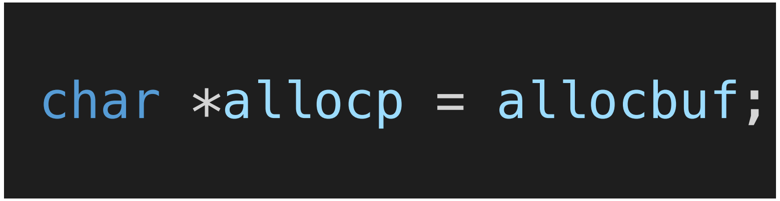
If it does, it assigns the allocp the value of p.





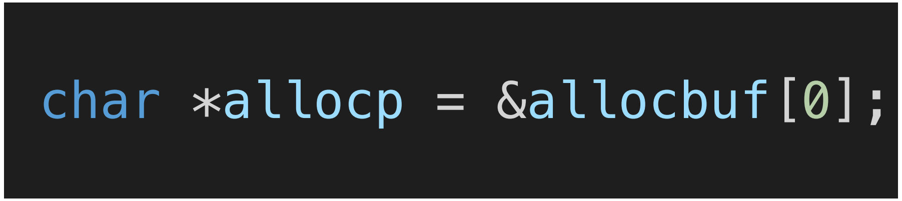
```c
// frees up the space pointed by p
void afree(char *p) {
  if (p >= allocbuf && p < allocbuf + ALLOCSIZE) {
    allocp = p;
  }
}
```

Initialization



```
char *allocp = allocbuf;
```

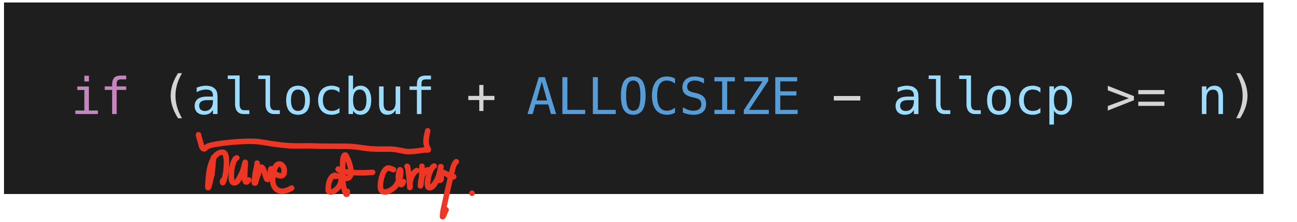
같은것.



```
char *allocp = &allocbuf[0];
```

Recall that the address of an array is the address of its first element

Pointer arithmetic



```
if (allocbuf + ALLOCSIZE - allocp >= n)
```

name of array.

If p, q point to elements of the same array, then we can compare them using ==, !=, <, >= etc.

p < q is true when p points to an array element before the one q points to.

Comparing pointers that don’t point to elements of the same array has undefined behavior!

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory by mistake
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

C operators

Operators	*Associativity*
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

- **`->`, `()`, and `[]` have high precedence, with `*` and `&` just below**
- **Unary `+`, `-`, and `*` have higher precedence than binary forms**

C Pointer Declarations: Test Yourself!

pointer to int

point an array

같은거?

char a[13]

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int

Pointer to a !

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf(“%d”,&val);
```

takes a pointer as an arg

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

→ 초기화 해주지 않는 함

calloc

Overwriting Memory

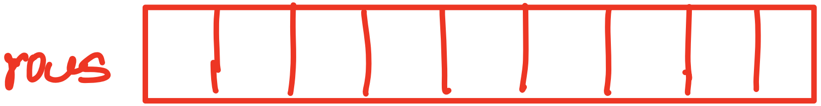
- Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

N

← row

← col



Overwriting Memory

- Off-by-one error

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Overwriting Memory

- Not checking the max string size

```
char s[8];
int i;
```

gets() does not check String is over the array or no.

```
gets(s);  /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

Overwriting Memory

- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Overwriting Memory

- **Referencing a pointer instead of the object it points to**

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

okay!

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

- **Nasty!**

Again

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

Referencing Freed Blocks

- Evil!

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

x= Null

Failing to Free Blocks (Memory Leaks)

- **Slow, long-term killer!**

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

No free

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```

나머지[illegible]

Dealing With Memory Bugs

- **Debugger: gdb**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- **Binary translator: `valgrind`** ☆
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- **glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`

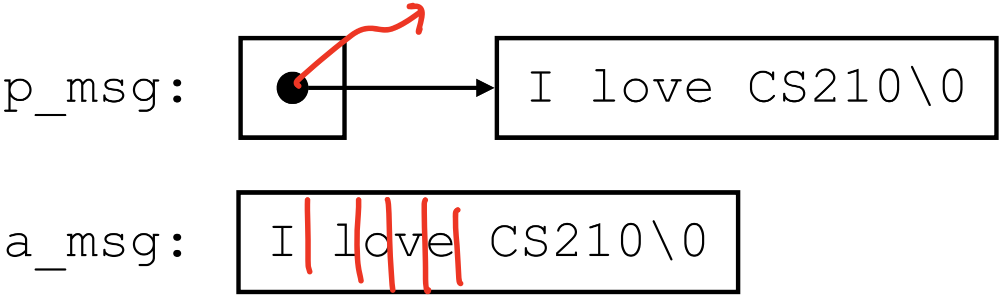
Lect.

Strings

String literals

A string literal, such as “I love CS210” is an array of chars, terminated by the null character `'\0'`. 까지.

```
char *p_msg = “I love CS210”; // pointer
char a_msg[] = “I love CS210”; // array const
```



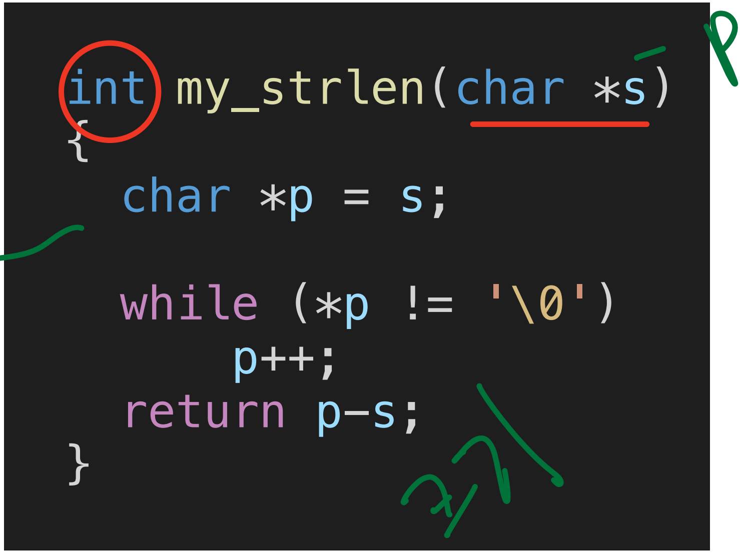
p_msg = 9;

Finding the length of a string

length of a string

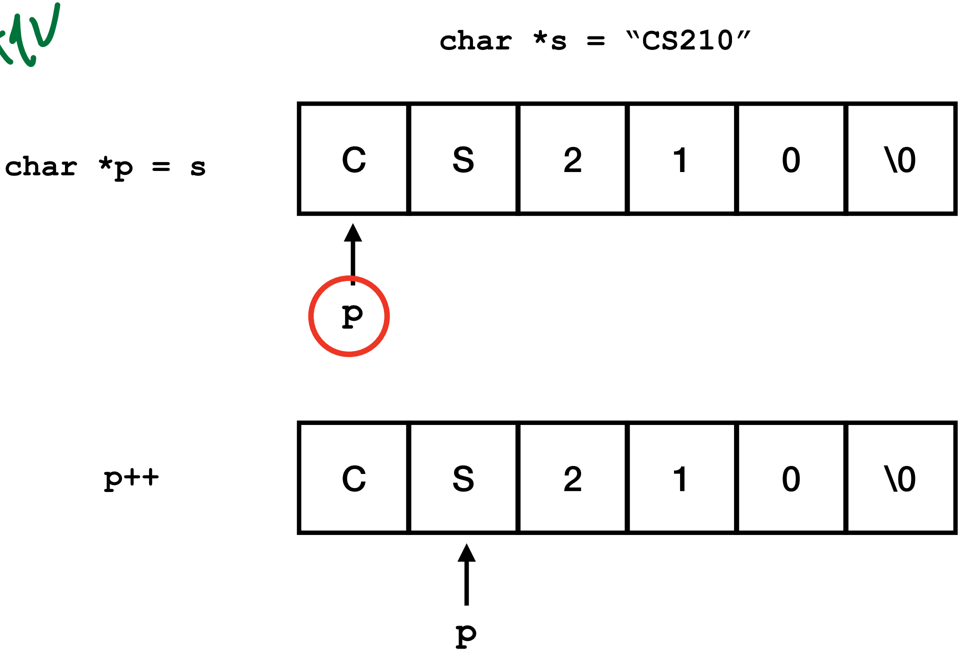
pointer

initial



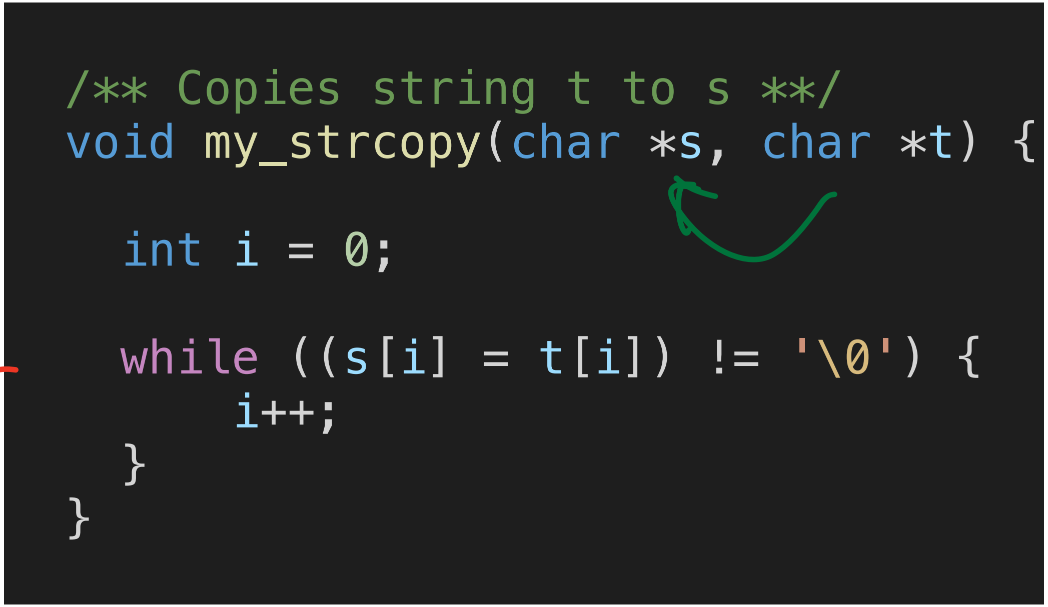
```
int my_strlen(char *s)
{
   char *p = s;

   while (*p != '\0')
       p++;
   return p-s;
}
```

거리



Copying strings

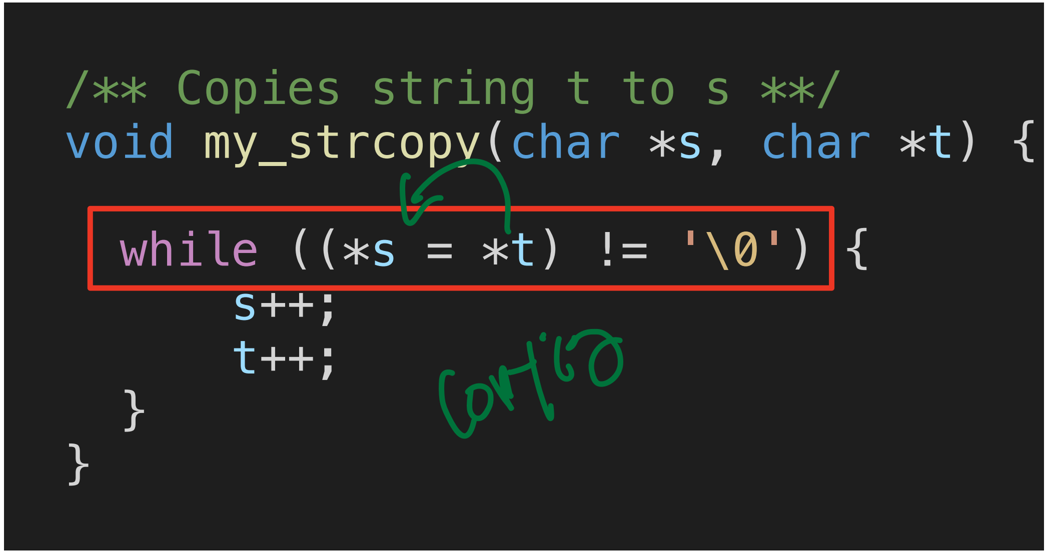
s=t;



```
/** Copies string t to s **/
void my_strcopy(char *s, char *t) {

  int i = 0;

  while ((s[i] = t[i]) != '\0') {
      i++;
  }
}
```

if t[i]!= '\0' ,
s[i] = t[i]

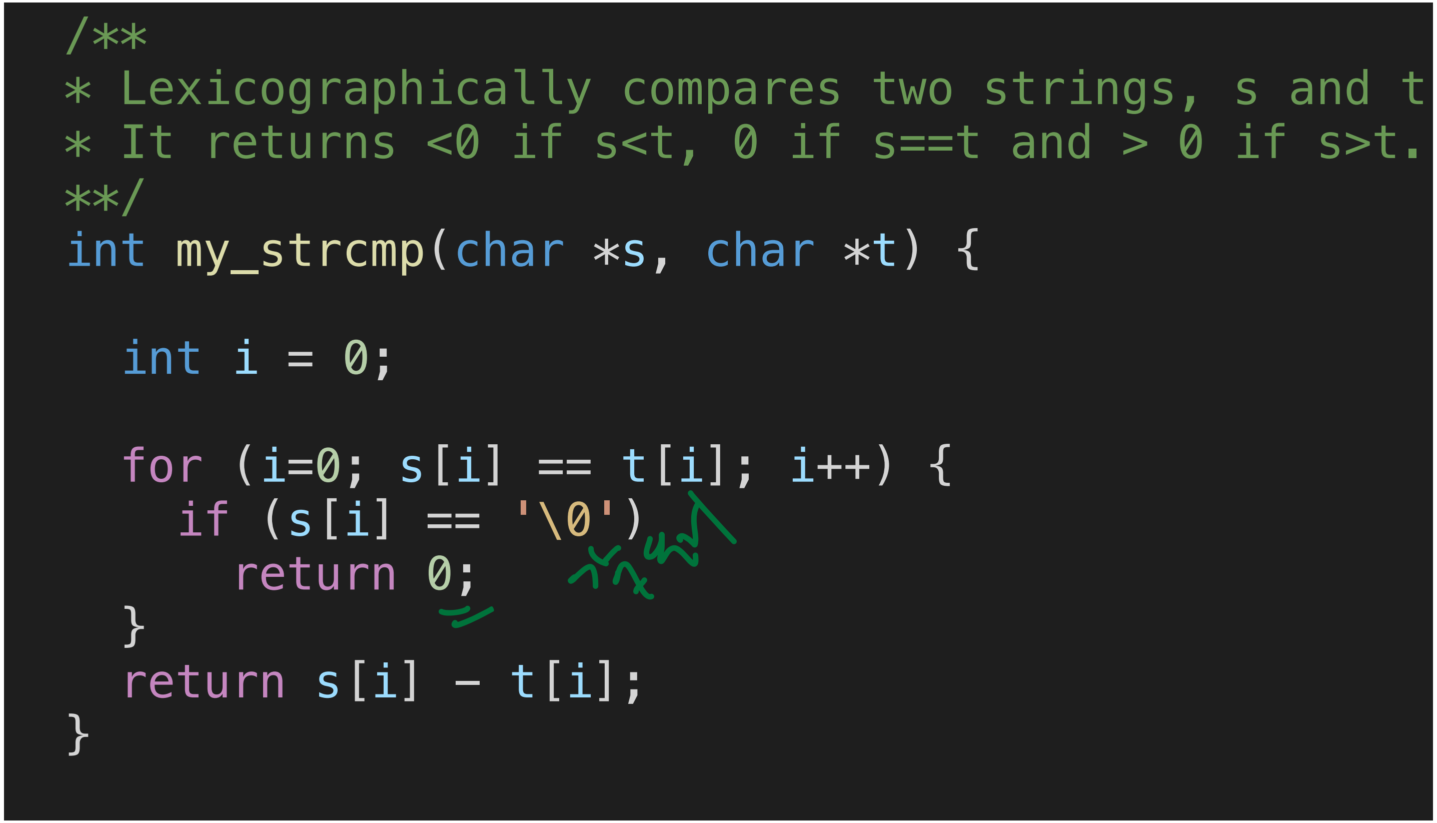
Copying strings (using pointers)



```
/** Copies string t to s **/
void my_strcopy(char *s, char *t) {

  while ((*s = *t) != '\0') {
        s++;
        t++;
  }
}
```

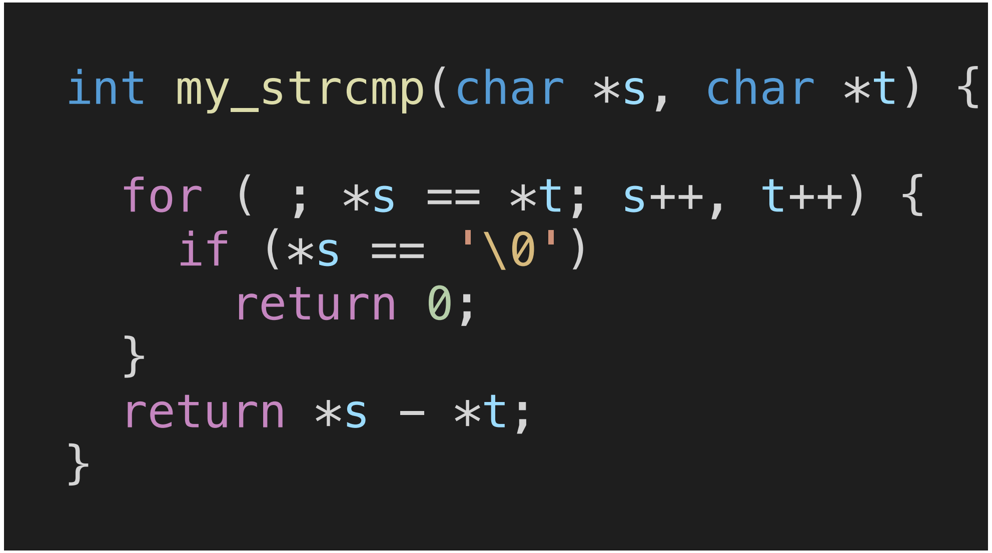
Contido

Comparing strings



```c
/**
* Lexicographically compares two strings, s and t
* It returns <0 if s<t, 0 if s==t and > 0 if s>t.
**/
int my_strcmp(char *s, char *t) {

  int i = 0;

  for (i=0; s[i] == t[i]; i++) {
    if (s[i] == '\0')
      return 0;
  }
  return s[i] - t[i];
}
```

Comparing strings (using pointers)



```c
int my_strcmp(char *s, char *t) {

  for ( ; *s == *t; s++, t++) {
    if (*s == '\0')
      return 0;
  }
  return *s - *t;
}
```

Using the C String Library

- The C library provides a set of functions for performing operations on strings.

- Programs that need string operations should contain the following line:

```
#include <string.h>
```

strig

stdio.h

The `strcpy` (String Copy) Function

- Prototype for the `strcpy` function:

```
char *strcpy(char *s1, const char *s2);
```

- `strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`.
- strcpy returns `s1` (a pointer to the destination string).
- The `strncpy` function is a safer way to copy a string:

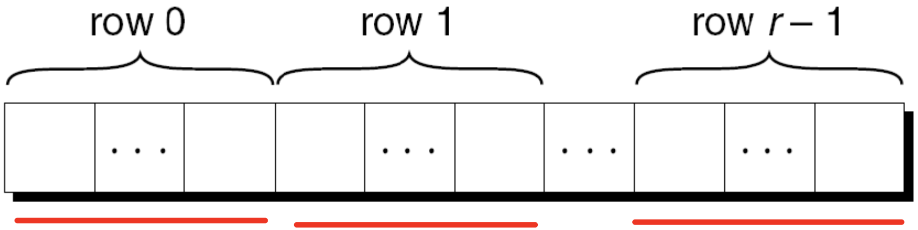
```
strncpy(str1, str2, sizeof(str1));
```

Check the book for other functions!

Multidimensional arrays and pointers

Multidimensional arrays and pointers

Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays.



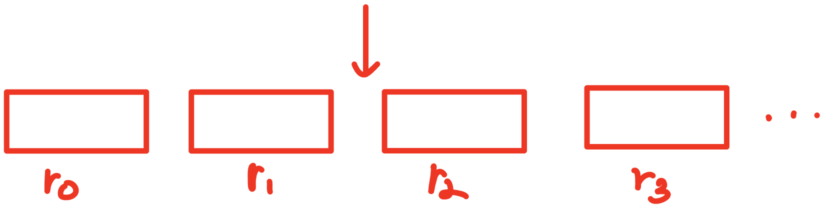
```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
   *p = 0;
```

Multidimensional arrays and pointers

- A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.

- To visit the elements of row `i`, we'd initialize `p` to point to element 0 in row `i` in the array `a`:

```
p = &a[i][0];
```



or we could simply write

```
p = a[i];
```

- For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first element in row `i`.

Multidimensional arrays and pointers

```
int a[NUM_ROWS][NUM_COLS];
```

p++

`a` is *not* a pointer to `a[0][0]`; instead, it’s a pointer to `a[0]`.

- C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.
- When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

To clear column `i` of the array `a`, we can write:

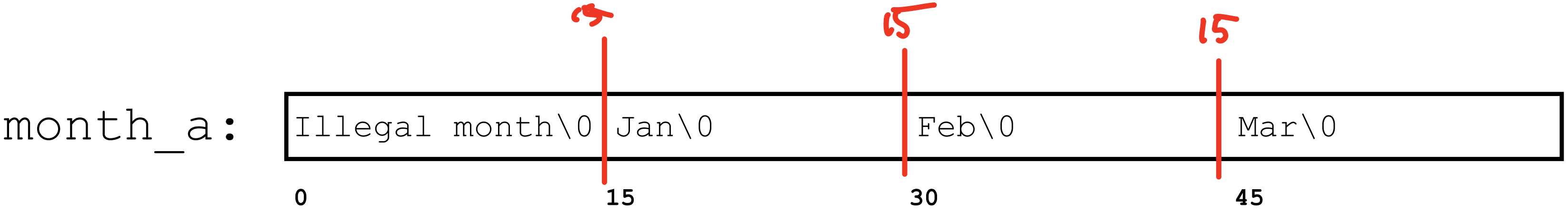
```
for (p = a; p < a + NUM_ROWS; p++)
   (*p)[i] = 0;
```

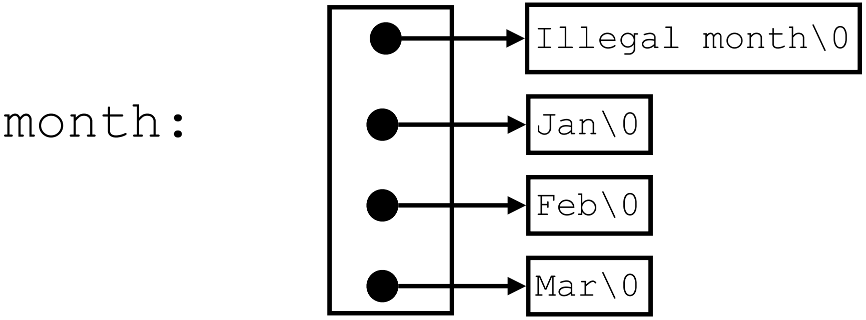
Multidimensional arrays and pointers

Are these declarations equivalent?

each array size

```
char month_a[][15] = {“Illegal month”,
                      “Jan”, “Feb”, “Mar”};
char *month[] = {“Illegal month”,
                 “Jan”, “Feb”, “Mar”};
```





Command-Line Arguments

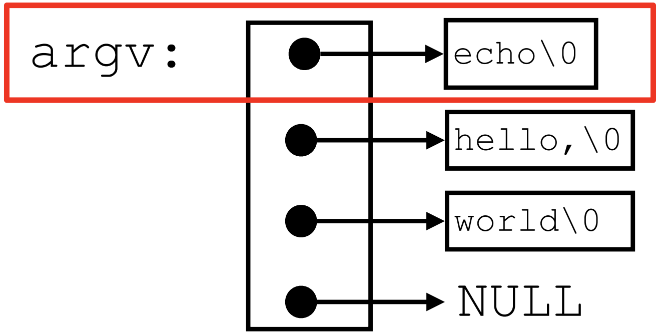
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[]){…}
```

number of arguments

An array of pointers to strings that contain the arguments

```
echo hello, world
```



Command-Line Arguments

- Accessing command-line arguments:

```
int i;
for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

- Or set up a pointer to `argv[1]`, then increment the pointer repeatedly:

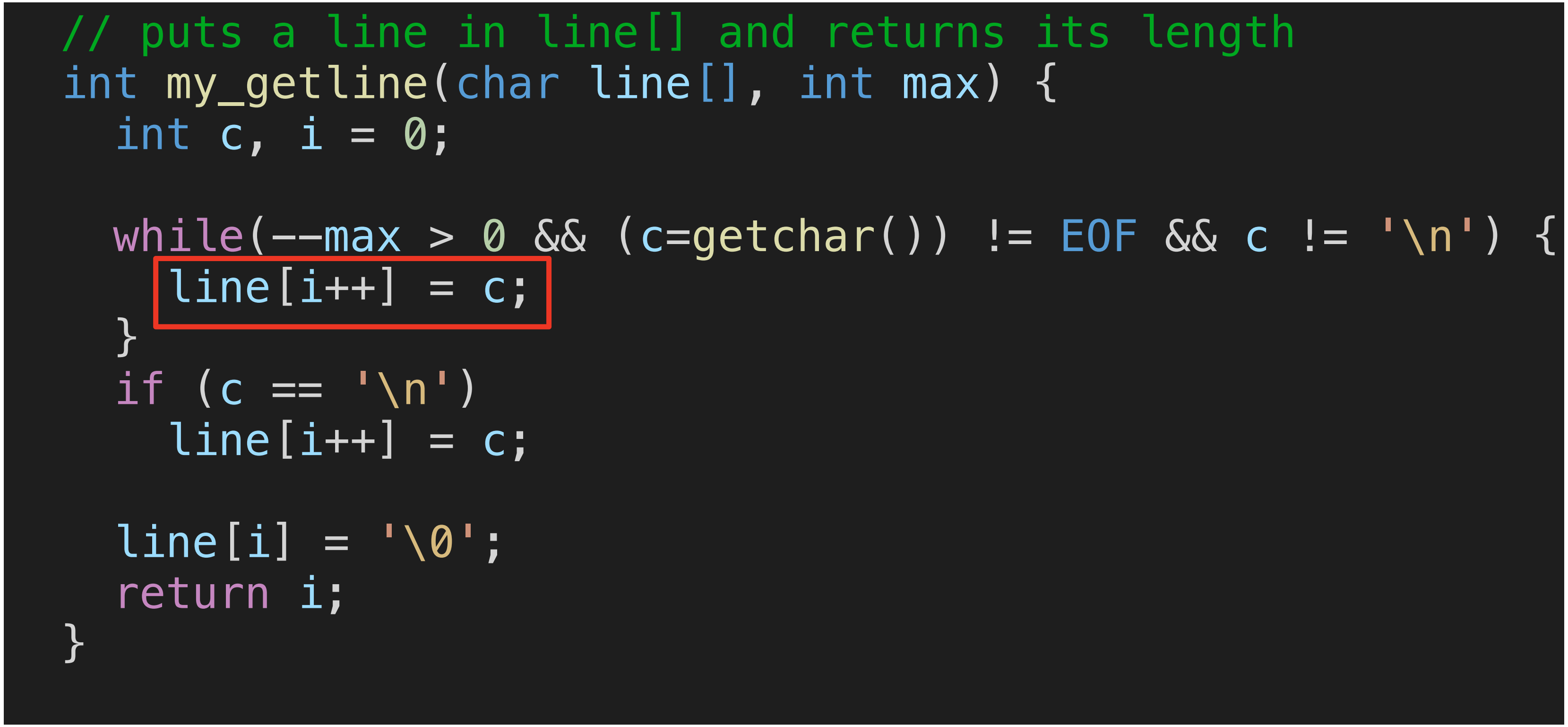
```
char **p;
for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

find: show the lines containing a pattern

```
vkalavri$ ./find char < allocate.c
```

```
3 static char allocbuf[ALLOCSIZE]; // space for alloc

4 static char *allocp = &allocbuf[0]; // next free position

6 // returns a pointer to n chars

7 char *alloc(int n) {

19 void afree(char *p) {
```

```c
#include <stdio.h>
#include <string.h>
#define MAXLINE 100

int my_getline(char line[], int max);

int main(int argc, char *argv[]) {
  char line[MAXLINE];
  int found = 0;
  long line_no = 0;

  if (argc != 2)
    printf("Usage: find <pattern>\n");
  else {
    while (my_getline(line, MAXLINE) > 0) {
      line_no++;
      if (strstr(line, argv[1]) != NULL) {
        printf("%ld %s", line_no, line);
        found++;
      }
    }
  }
  return found;
}
```



```c
// puts a line in line[] and returns its length
int my_getline(char line[], int max) {
  int c, i = 0;

  while(--max > 0 && (c=getchar()) != EOF && c != '\n') {
    line[i++] = c;
  }
  if (c == '\n')
    line[i++] = c;

  line[i] = '\0';
  return i;
}
```