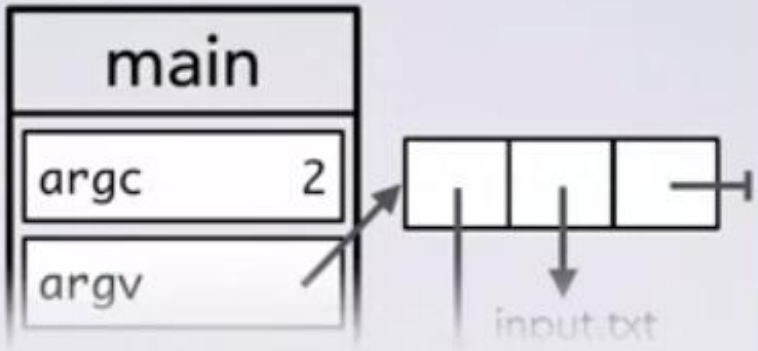
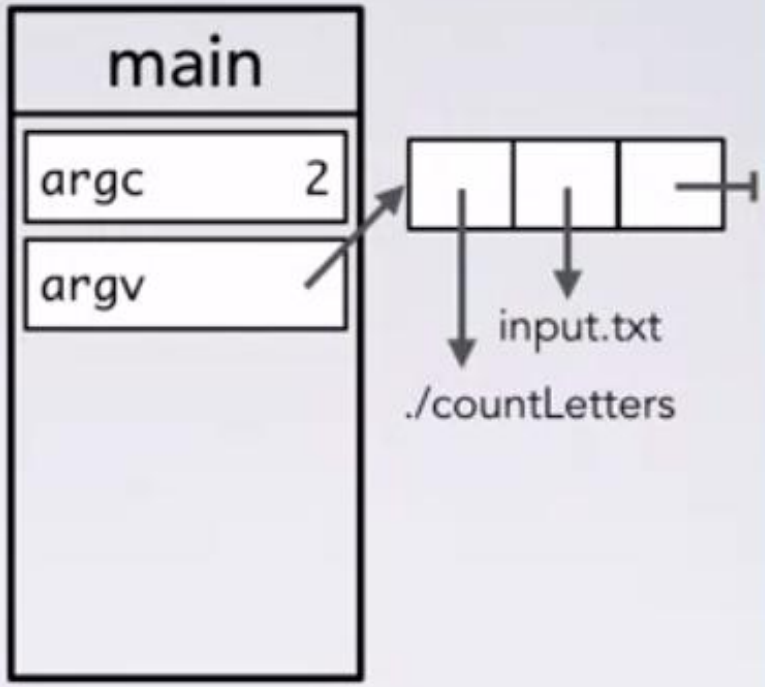
```
int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  int c;
  int letters = 0;
  while ( (c = fgetc(f)) != EOF ) {
    if (isalpha(c)) {
      letters++;
    }
  }
  printf("%s has %d letters in it\n", argv[1], letters);
  return EXIT_SUCCESS;
}
```



Output

As always, our execution arrow
begins at the start of main.

```
int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  int c;
  int letters = 0;
  while ( (c = fgetc(f)) != EOF ) {
    if (isalpha(c)) {
      letters++;
    }
  }
  printf("%s has %d letters in it\n", argv[1], letters);
  return EXIT_SUCCESS;
}
```

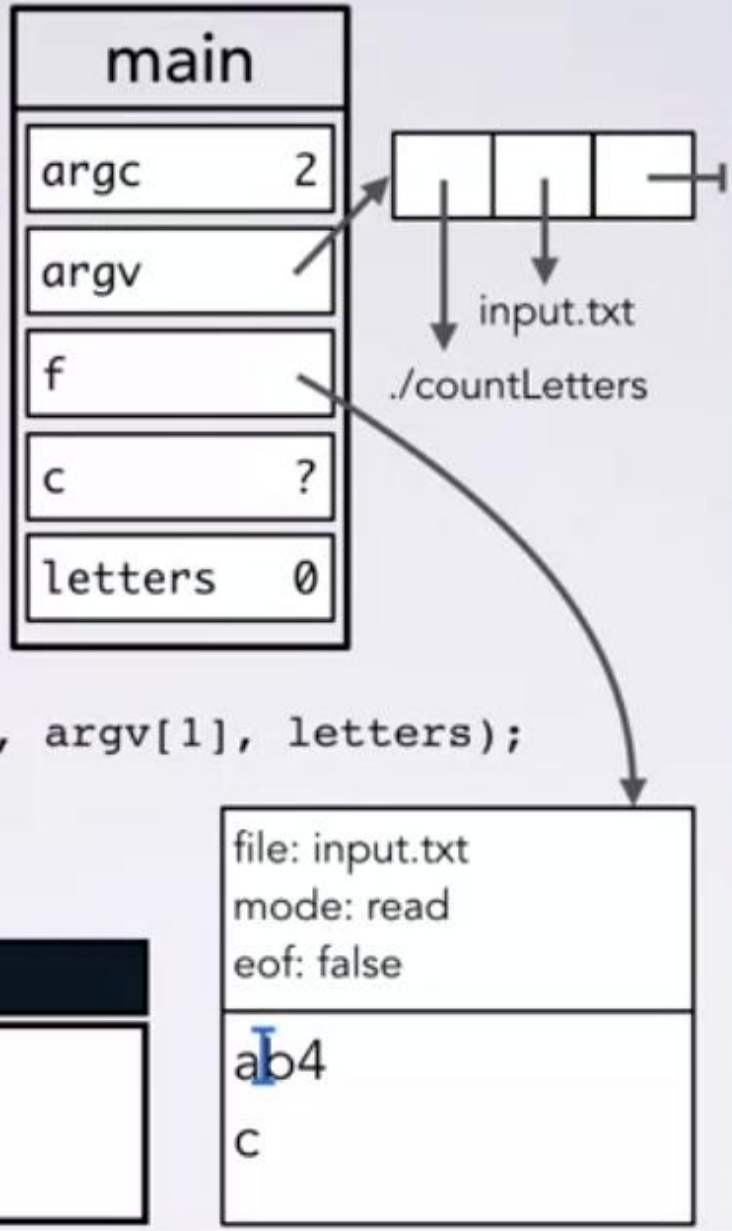


Output

Now that we have command line arg
main's frame begins with argc

```
int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  int c;
  int letters = 0;
  while ( (c = fgetc(f)) != EOF ) {
    if (isalpha(c)) {
      letters++;
    }
  }
  printf("%s has %d letters in it\n", argv[1], letters);
  return EXIT_SUCCESS;
}
```

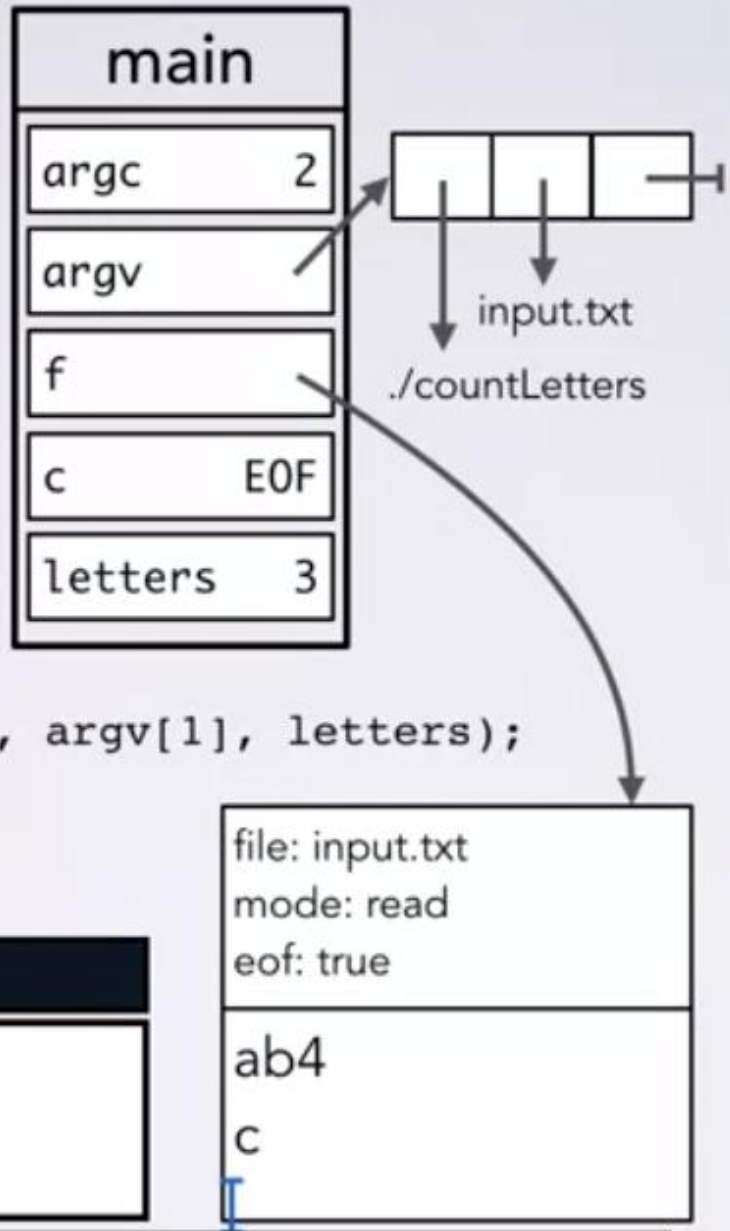
a



Output

We then do this assignment

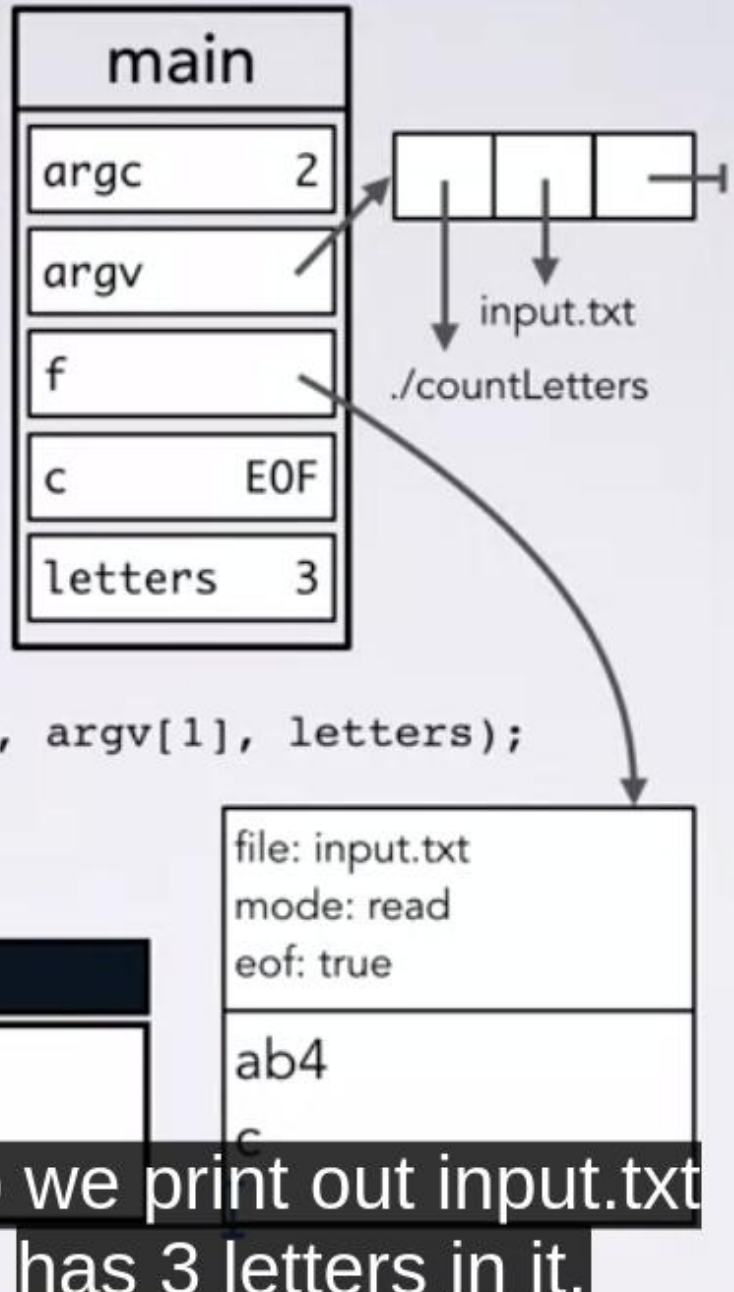
```
int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  int c;
  int letters = 0;
  while ( (c = fgetc(f)) != EOF ) {
    if (isalpha(c)) {
      letters++;
    }
  }
  printf("%s has %d letters in it\n", argv[1], letters);
  return EXIT_SUCCESS;
}
```



Output

If we were to call feof,

```
int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  int c;
  int letters = 0;
  while ( (c = fgetc(f)) != EOF ) {
    if (isalpha(c)) {
      letters++;
    }
  }
  printf("%s has %d letters in it\n", argv[1], letters);
  return EXIT_SUCCESS;
}
```



Output

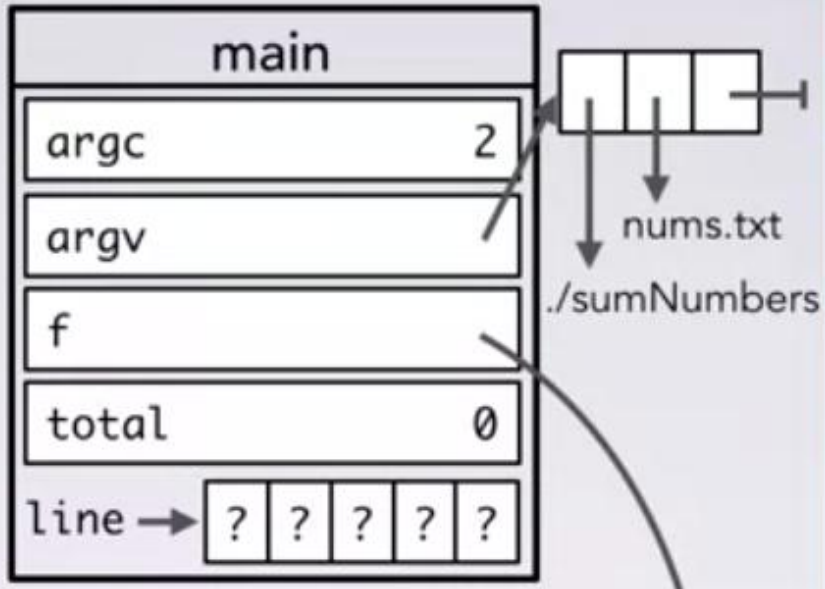
input.txt has 3 letters in it

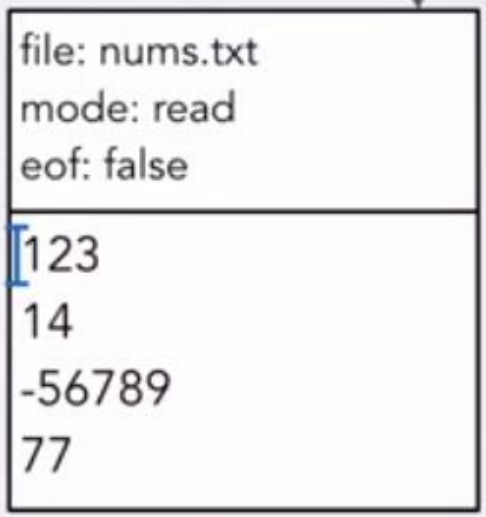
So we print out input.txt has 3 letters in it.

```c
#define LINE_SIZE 5

int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  long total = 0;
  char line[LINE_SIZE];
  while (fgets(line, LINE_SIZE, f) != NULL) {
    if (strchr(line, '\n') == NULL) {
      printf("Line is too long!\n");
      return EXIT_FAILURE;
    }
    total += atoi(line);
  }
  printf("The total is %ld\n", total);
  return EXIT_SUCCESS;
}
```

Output

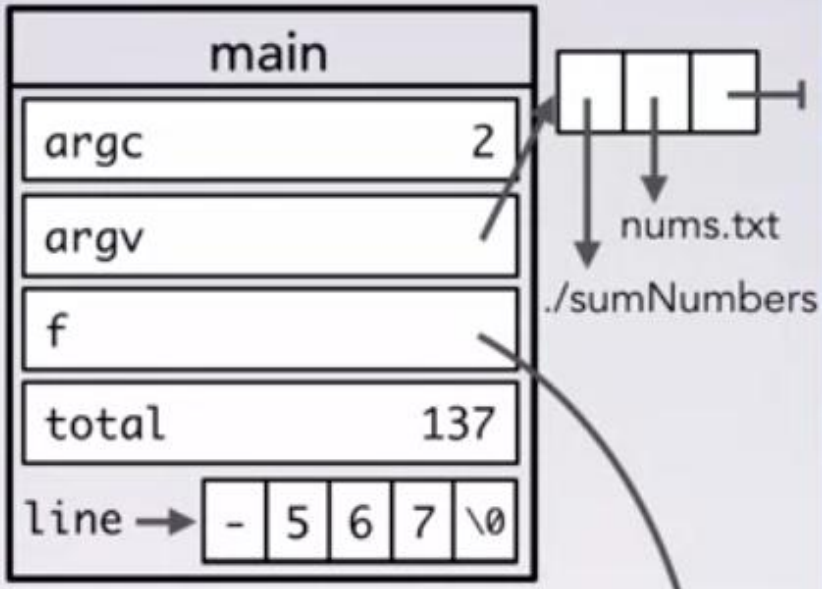
```c
#define LINE_SIZE 5

int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  long total = 0;
  char line[LINE_SIZE];
  while (fgets(line, LINE_SIZE, f) != NULL) {
    if (strchr(line, '\n') == NULL) {
      printf("Line is too long!\n");
      return EXIT_FAILURE;
    }
    total += atoi(line);
  }
  printf("The total is %ld\n", total);
  return EXIT_SUCCESS;
}
```

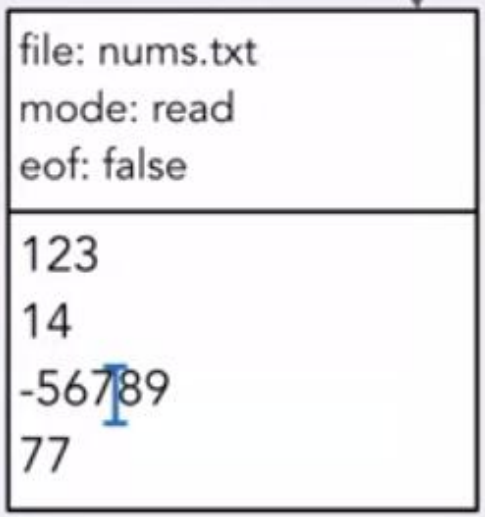




Output

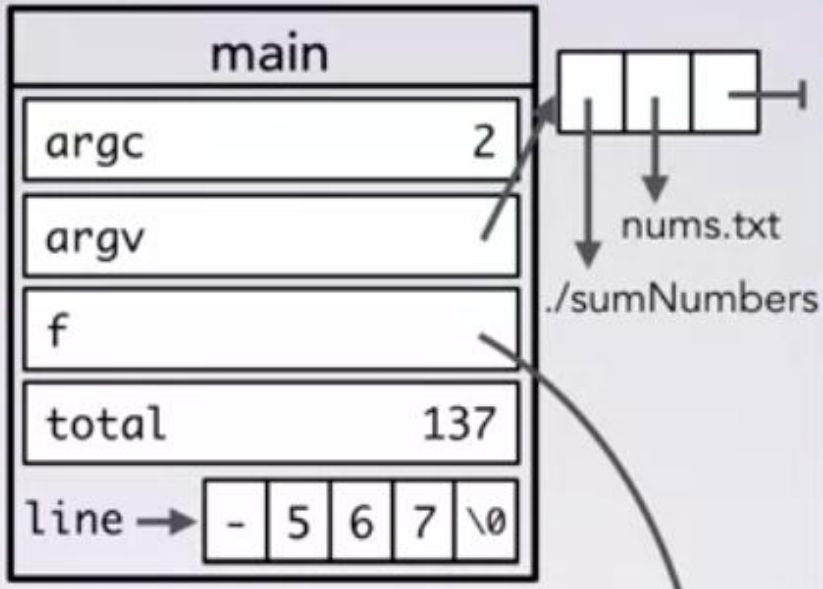
```c
#define LINE_SIZE 5

int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  long total = 0;
  char line[LINE_SIZE];
  while (fgets(line, LINE_SIZE, f) != NULL) {
    if (strchr(line, '\n') == NULL) {
      printf("Line is too long!\n");
      return EXIT_FAILURE;
    }
    total += atoi(line);
  }
  printf("The total is %ld\n", total);
  return EXIT_SUCCESS;
}
```





Output

```c
#define LINE_SIZE 5

int main (int argc, char ** argv) {
  if (argc != 2) { /* omitted */ }
  FILE * f = fopen(argv[1], "r");
  if (f == NULL) { /* omitted */ }
  long total = 0;
  char line[LINE_SIZE];
  while (fgets(line, LINE_SIZE, f) != NULL) {
    if (strchr(line, '\n') == NULL) {
      printf("Line is too long!\n");
      return EXIT_FAILURE;
    }
    total += atoi(line);
  }
  printf("The total is %ld\n", total);
  return EXIT_SUCCESS;
}
```

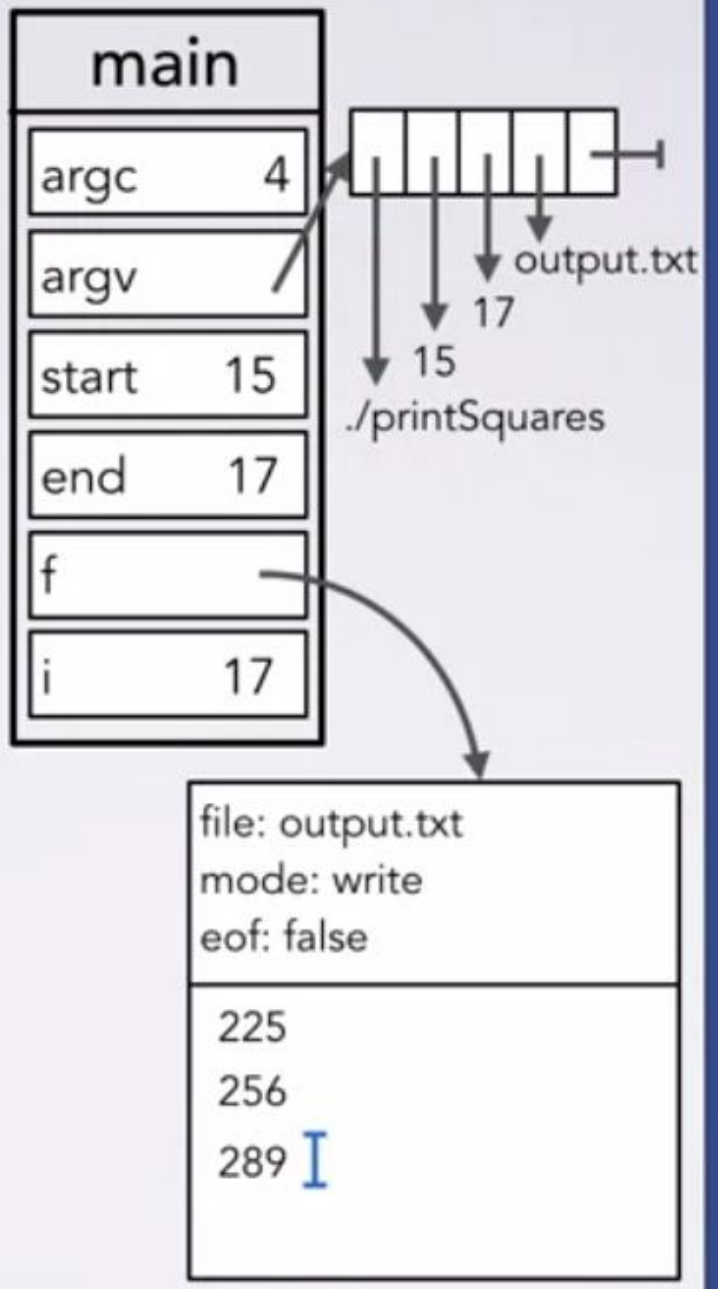


file: nums.txt
mode: read
eof: false

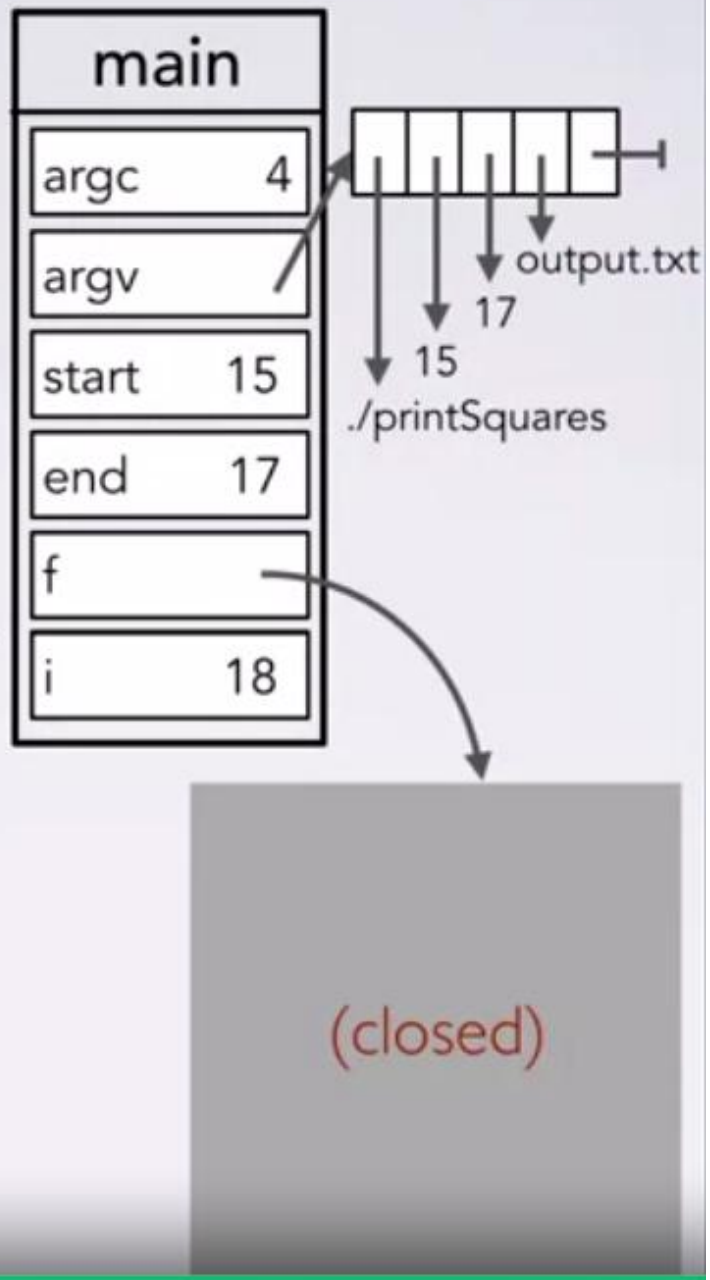
123
14
-56789
77

Output

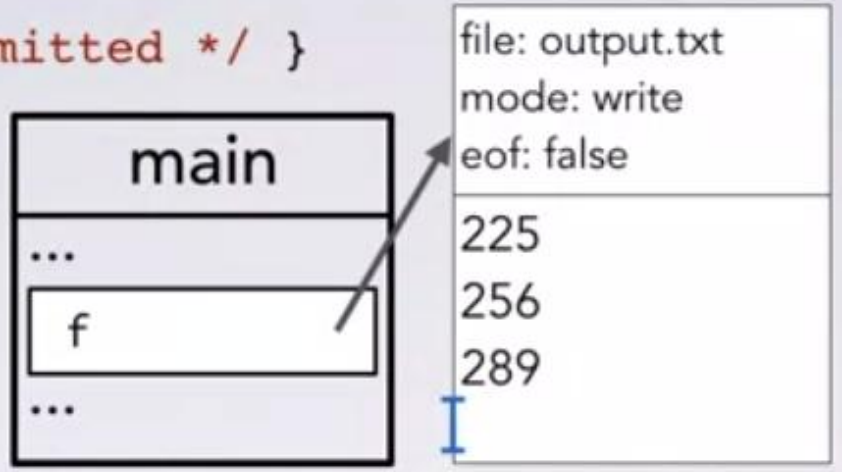
```
int main (int argc, char ** argv) {
  if (argc != 4) { /* omitted */ }
  int start = atoi(argv[1]);
  int end = atoi(argv[2]);
  FILE * f = fopen(argv[3], "w");
  if (f == NULL) { /* omitted */ }
  for (int i = start; i <= end; i++) {
    fprintf(f, "%d\n", i*i);
  }
  //fclose discussed in next section
  if (fclose(f) != 0) { /* omitted */ }
  return EXIT_SUCCESS;
}
```



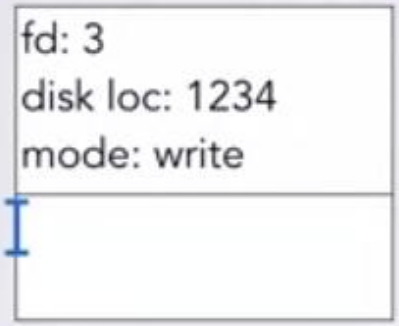
```
int main (int argc, char ** argv) {
  if (argc != 4) { /* omitted */ }
  int start = atoi(argv[1]);
  int end = atoi(argv[2]);
  FILE * f = fopen(argv[3], "w");
  if (f == NULL) { /* omitted */ }
  for (int i = start; i <= end; i++) {
    fprintf(f, "%d\n", i*i);
  }
  //fclose discussed in next section
  if (fclose(f) != 0) { /* omitted */ }
  return EXIT_SUCCESS;
}
```



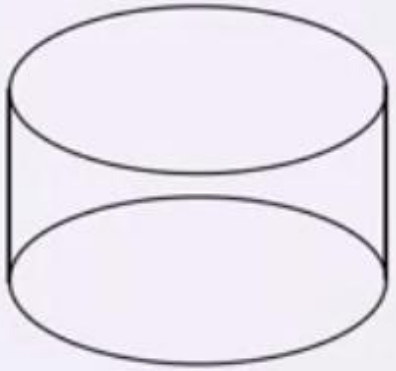
```
...
if (fclose(f) != 0) { /* omitted */ }
...
```

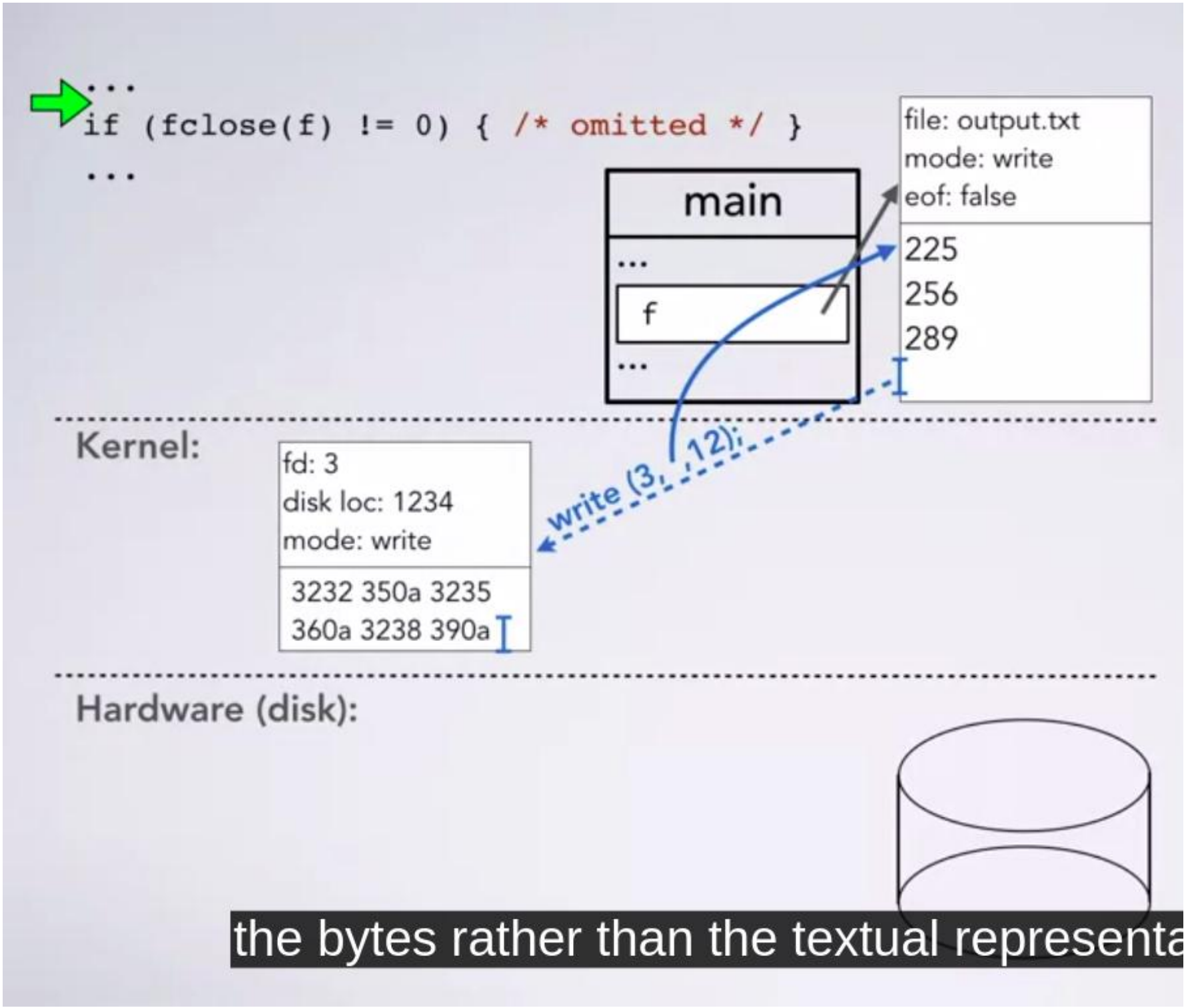


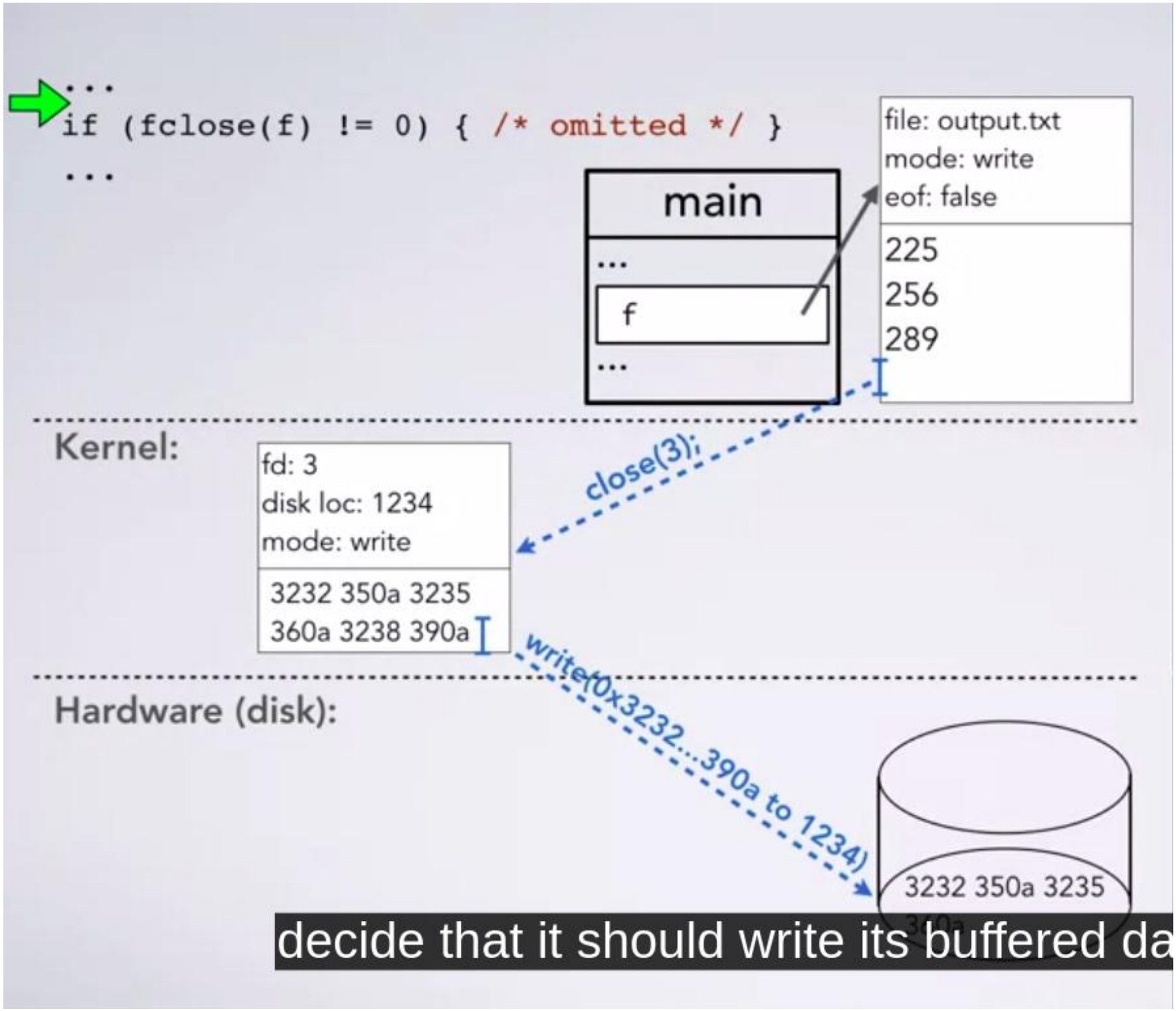
Kernel:

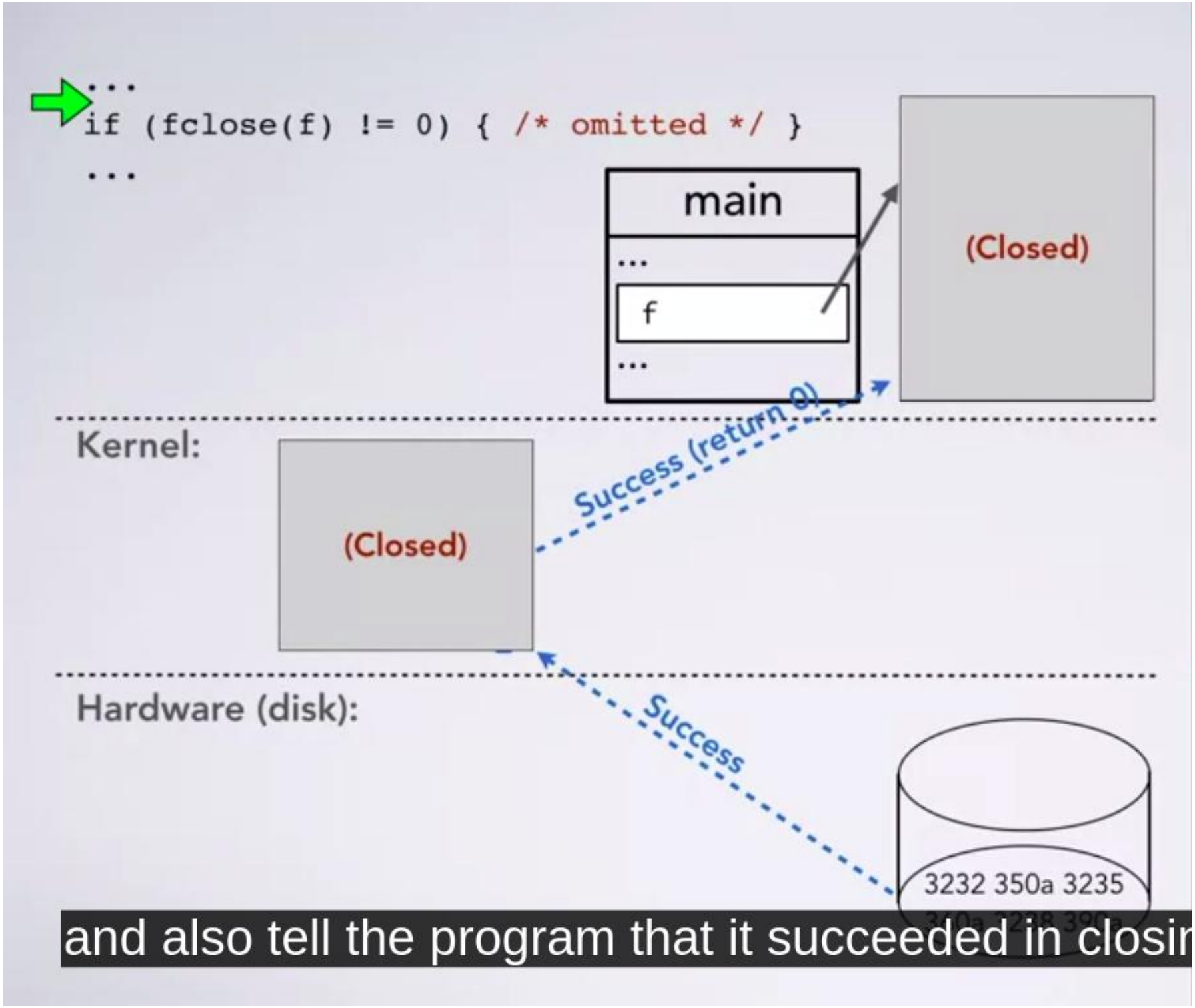


Hardware (disk):

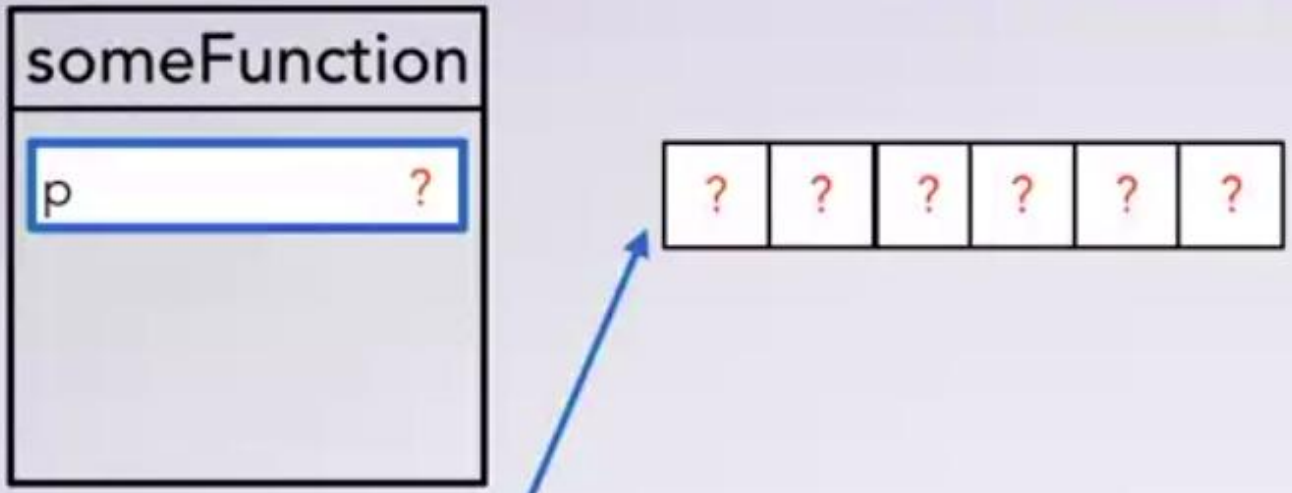



...
if (fclose(f) != 0) { /* omitted */ }
...
main
...
f
...
file: output.txt
mode: write
eof: false
225
256
289
Kernel:
fd: 3
disk loc: 1234
mode: write
3232 350a 3235
360a 3238 390a
write (3, , 12);
Hardware (disk):
the bytes rather than the textual representa


...
if (fclose(f) != 0) { /* omitted */ }
...
main
...
f
...
file: output.txt
mode: write
eof: false
225
256
289
Kernel:
fd: 3
disk loc: 1234
mode: write
3232 350a 3235
360a 3238 390a
close(3);
write(0x3232...390a to 1234)
Hardware (disk):
3232 350a 3235
decide that it should write its buffered da



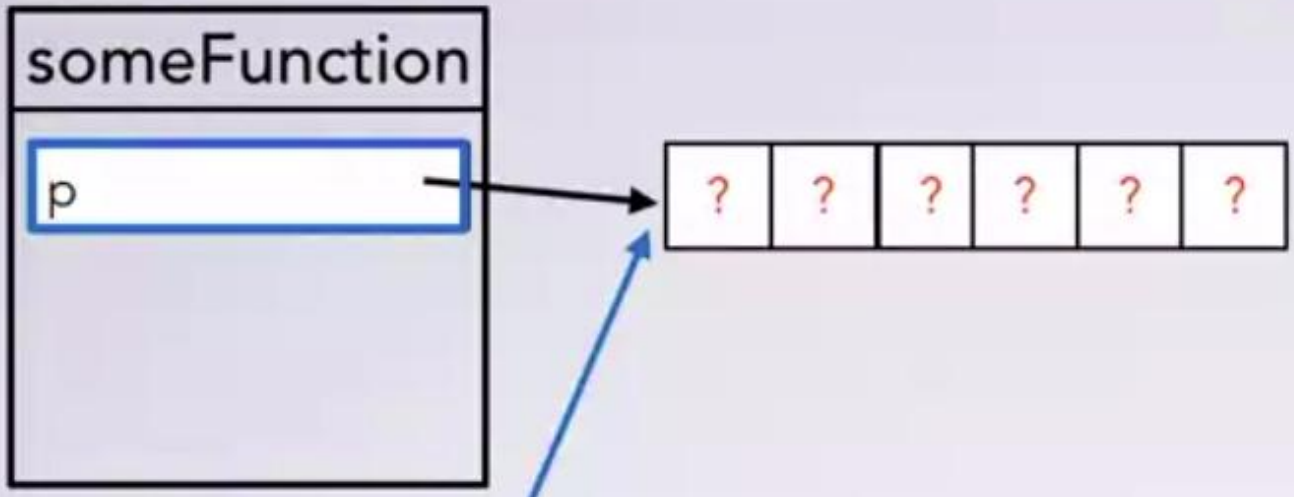
and also tell the program that it succeeded in closi



return value of malloc

```
int * p;
p = malloc (6 * sizeof(*p));

.....
return;
```

So, let's draw that box.

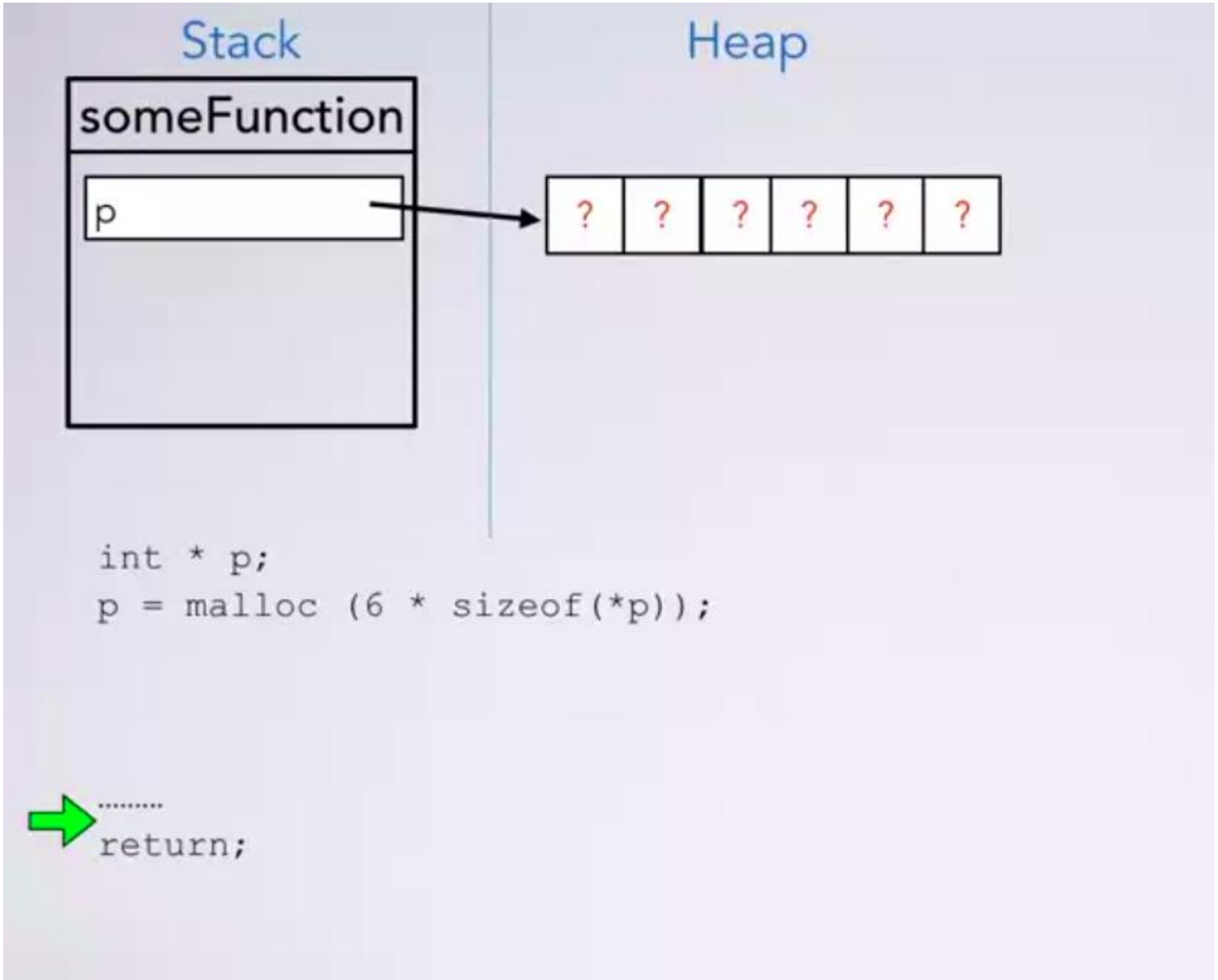



```
int * p;
p = malloc (6 * sizeof(*p));
```

return value of malloc

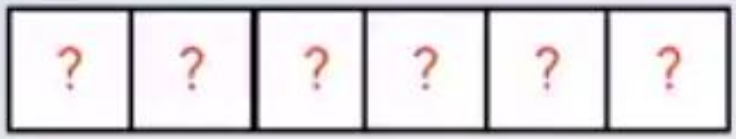
```
.....
return;
```

which is the value of the function call i


Stack
Heap
someFunction
p
? ? ? ? ? ?
int * p;
p = malloc (6 * sizeof(*p));
.....
return;

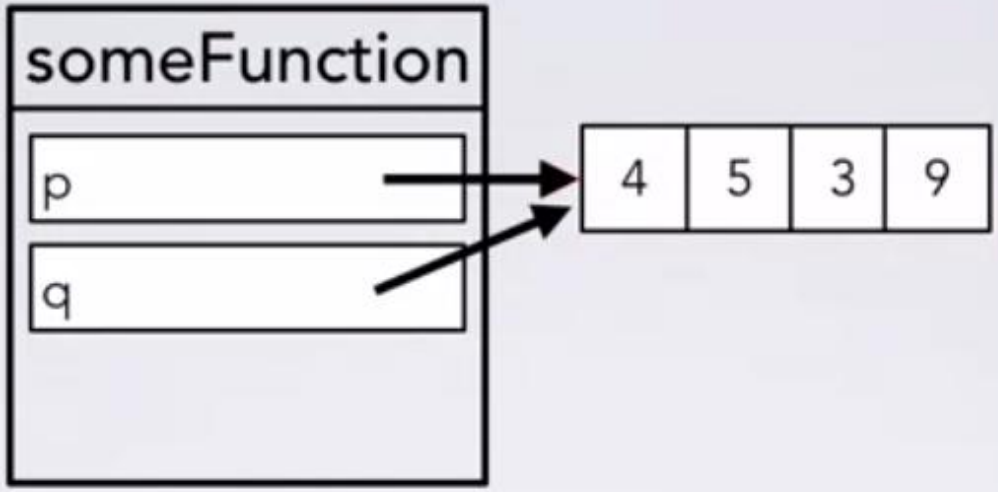
Stack

Heap



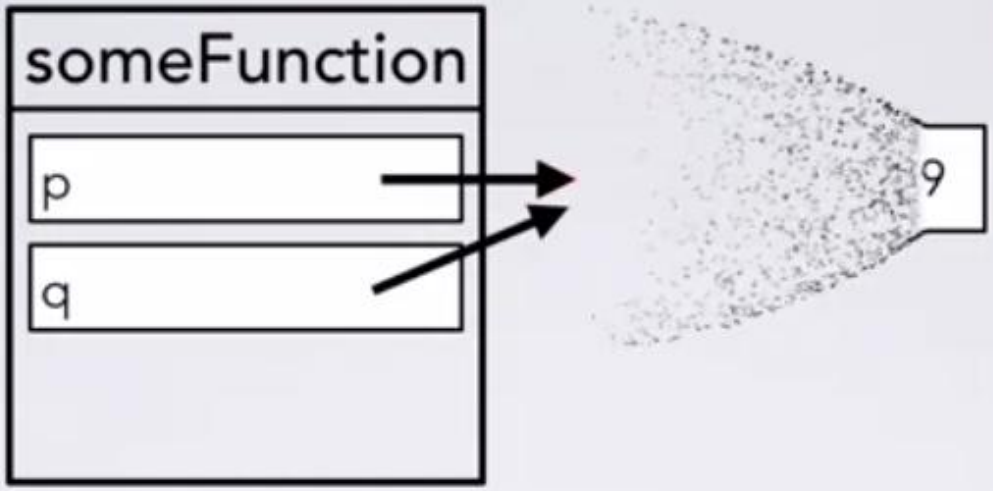

```
int * p;
p = malloc (6 * sizeof(*p));


.....
return;
```

```
int * p = malloc(4 * sizeof(*p));
p[0] = 4;
p[1] = 5;
p[2] = 3;
p[3] = 9;
int * q = p;

//.....

free(p);
```



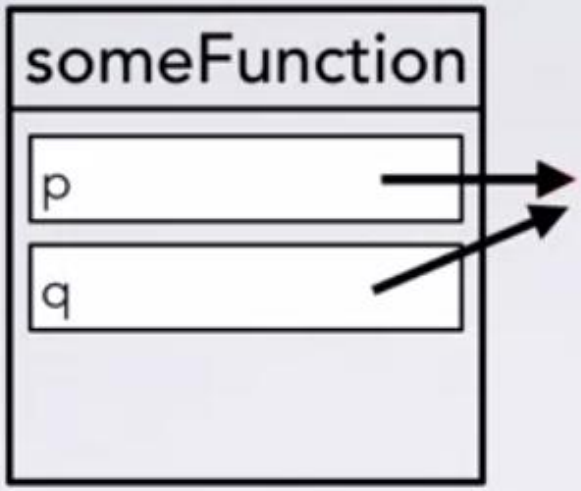
Free p does not actually affect th
but rather the memory that p poin

```
int * p = malloc(4 * sizeof(*p));
p[0] = 4;
p[1] = 5;
p[2] = 3;
p[3] = 9;
int * q = p;

//.....

free(p);
```



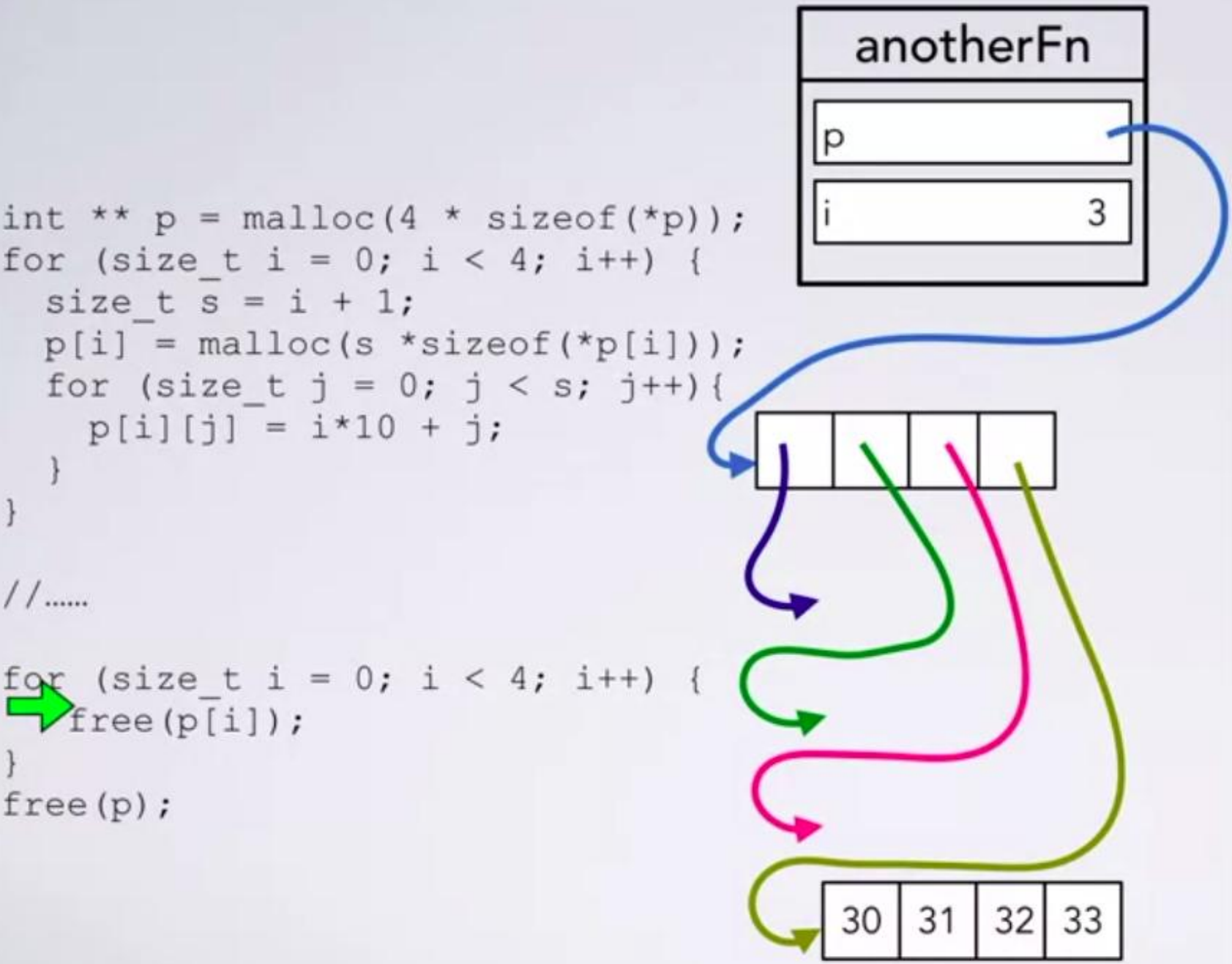
Freeing this memory destroys that
leaves p dangling.

```
int * p = malloc(4 * sizeof(*p));
p[0] = 4;
p[1] = 5;
p[2] = 3;
p[3] = 9;
int * q = p;

//.....

free(p);
```

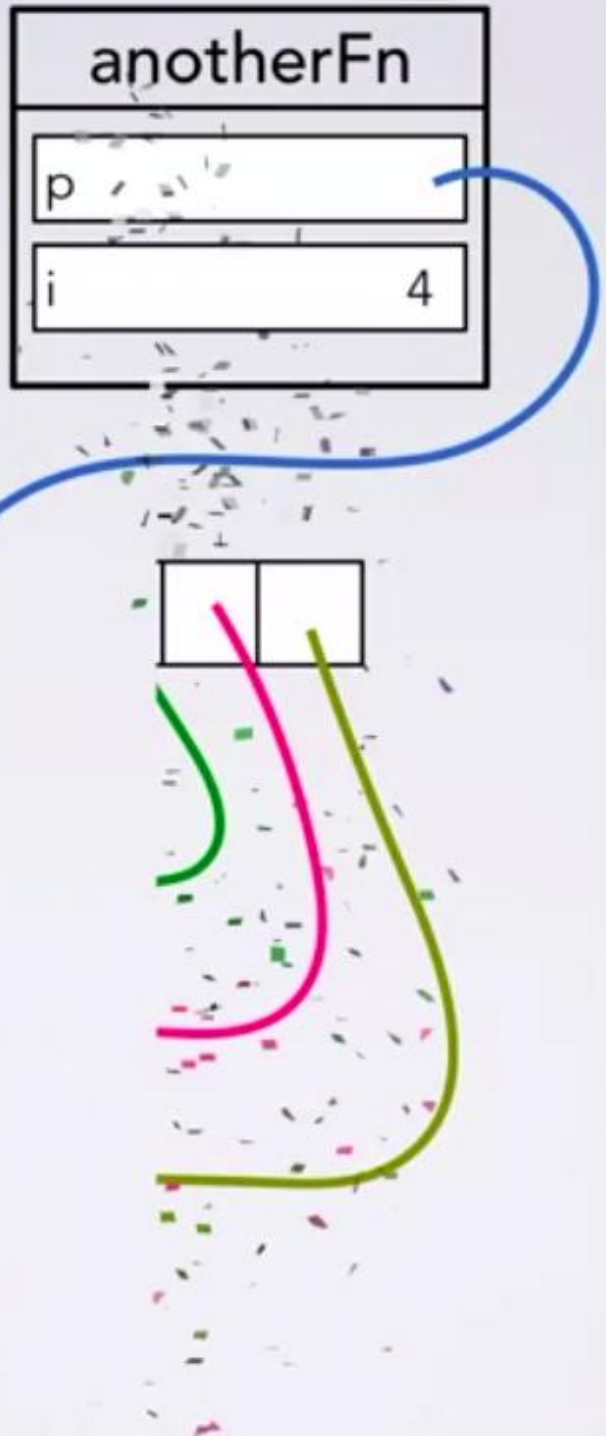


as with any dangling pointer.

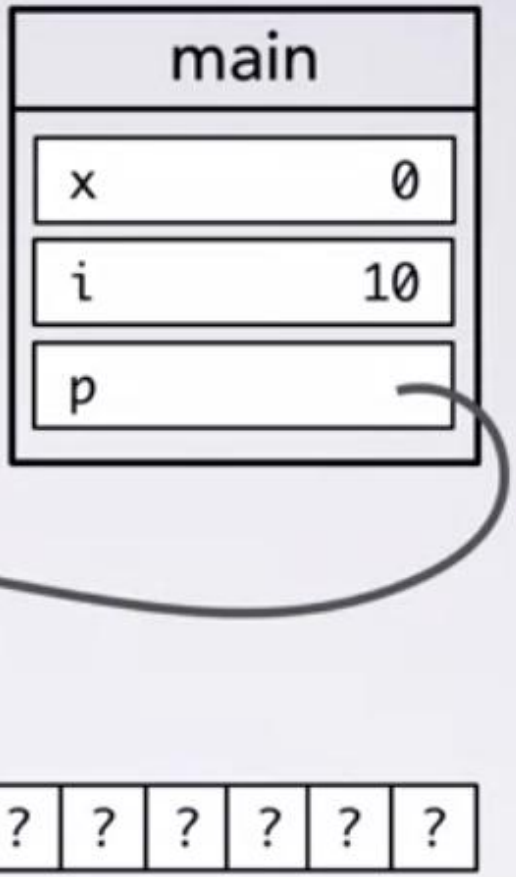
```
int ** p = malloc(4 * sizeof(*p));
for (size_t i = 0; i < 4; i++) {
  size_t s = i + 1;
  p[i] = malloc(s *sizeof(*p[i]));
  for (size_t j = 0; j < s; j++){
    p[i][j] = i*10 + j;
  }
}

//……

for (size_t i = 0; i < 4; i++) {
  free(p[i]);
}
free(p);
```



```
int ** p = malloc(4 * sizeof(*p));
for (size_t i = 0; i < 4; i++) {
  size_t s = i + 1;
  p[i] = malloc(s *sizeof(*p[i]));
  for (size_t j = 0; j < s; j++){
    p[i][j] = i*10 + j;
  }
}

//……

for (size_t i = 0; i < 4; i++) {
   free(p[i]);
}
free(p);
```

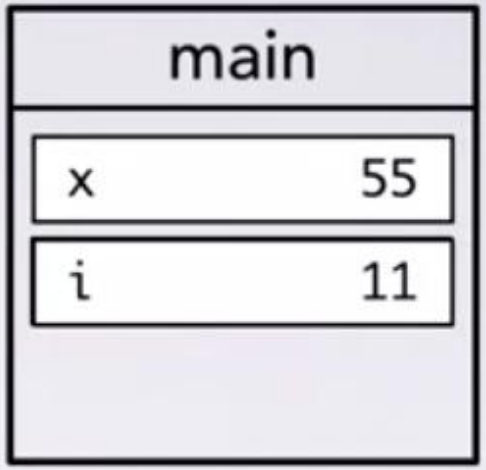


```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```



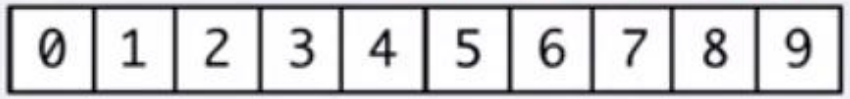
Example without free

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```

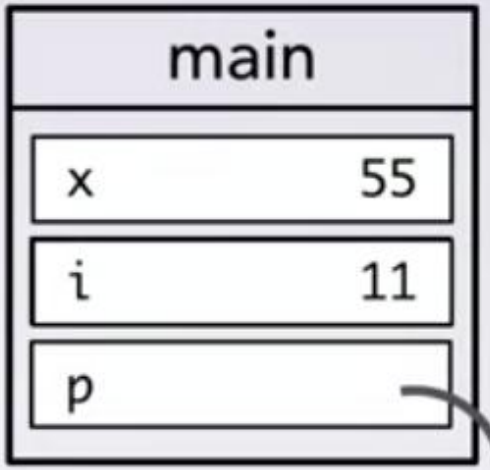


Example without free

Leaked memory!

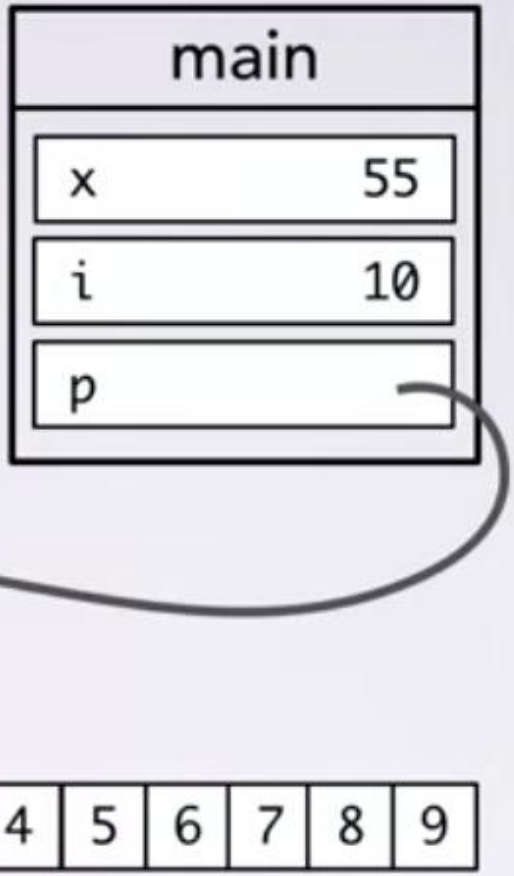


```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```



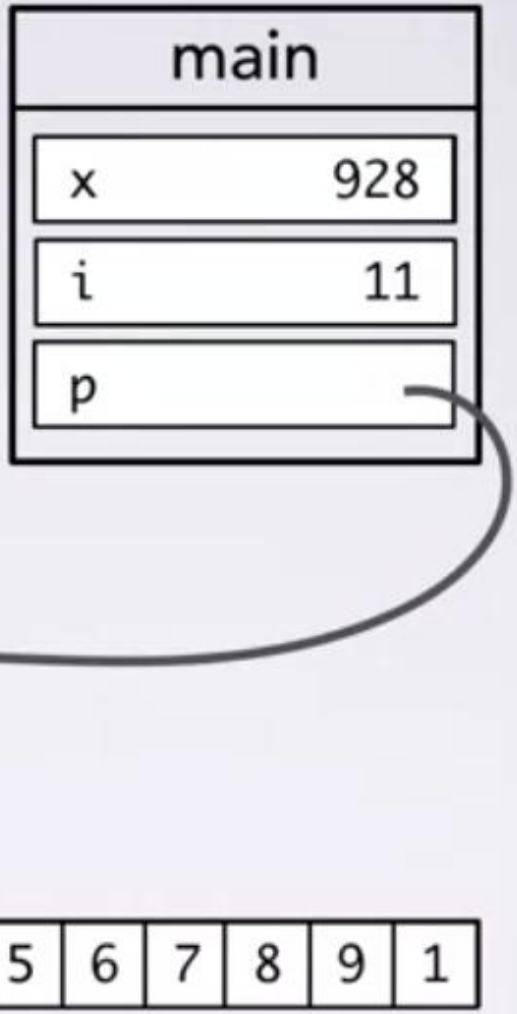
Example without free

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
    free(p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```



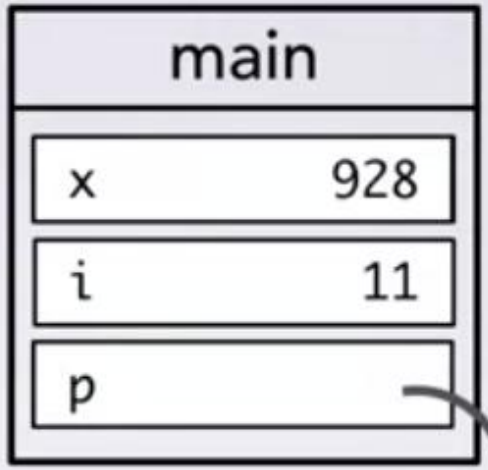
Example with free

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
    free(p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```



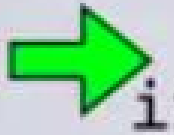
Example with free

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int * p = malloc(i * sizeof(*p));
    x = doSomeComputation(x, i, p);
    free(p);
  }
  printf("Answer %d\n", x);
  return EXIT_SUCCESS;
}
```



Example with free

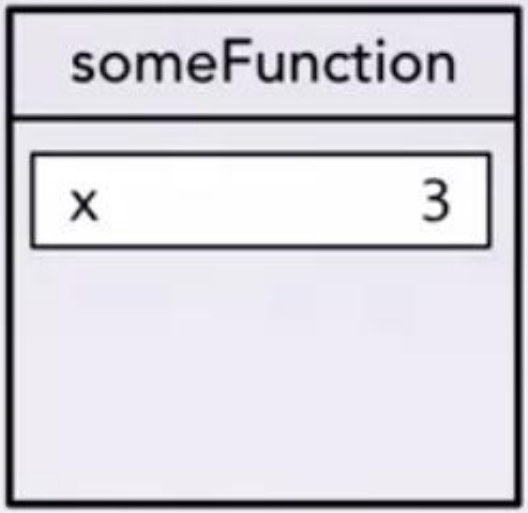
Double Free



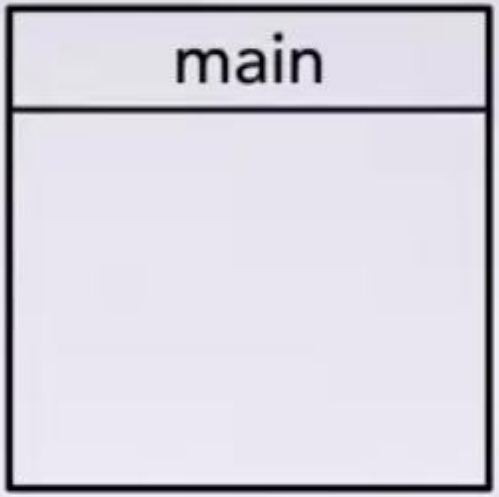
```
int * p = malloc(4 * sizeof(*p));
int * q = p;
…
free(q);
…
free(p);
```

Free Memory Not in the Heap

```
int x = 3;
int * p = &x;
…
free(p);
```



```
int main (void) {
  int * p = malloc(10 * sizeof(*p));
  initValues(p);
  p = realloc(p, 14 * sizeof(*p));
  initMoreValues(p);
  p = realloc(p, 4 * sizeof(*p));
  ...
  return EXIT_SUCCESS;
}
```

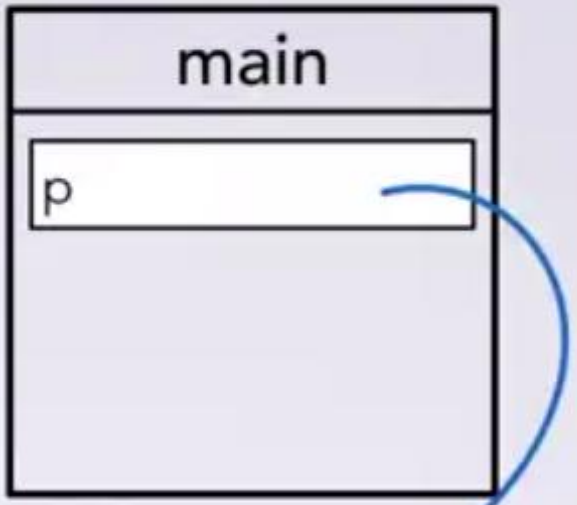


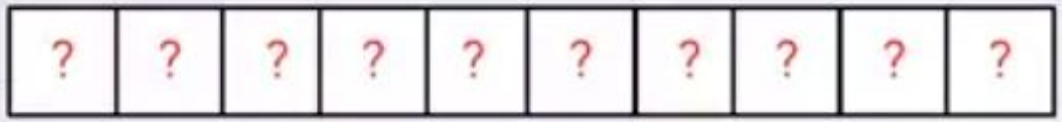
realloc = malloc + copy+ free

One Possible Execution

You can think of realloc as mallocing

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```



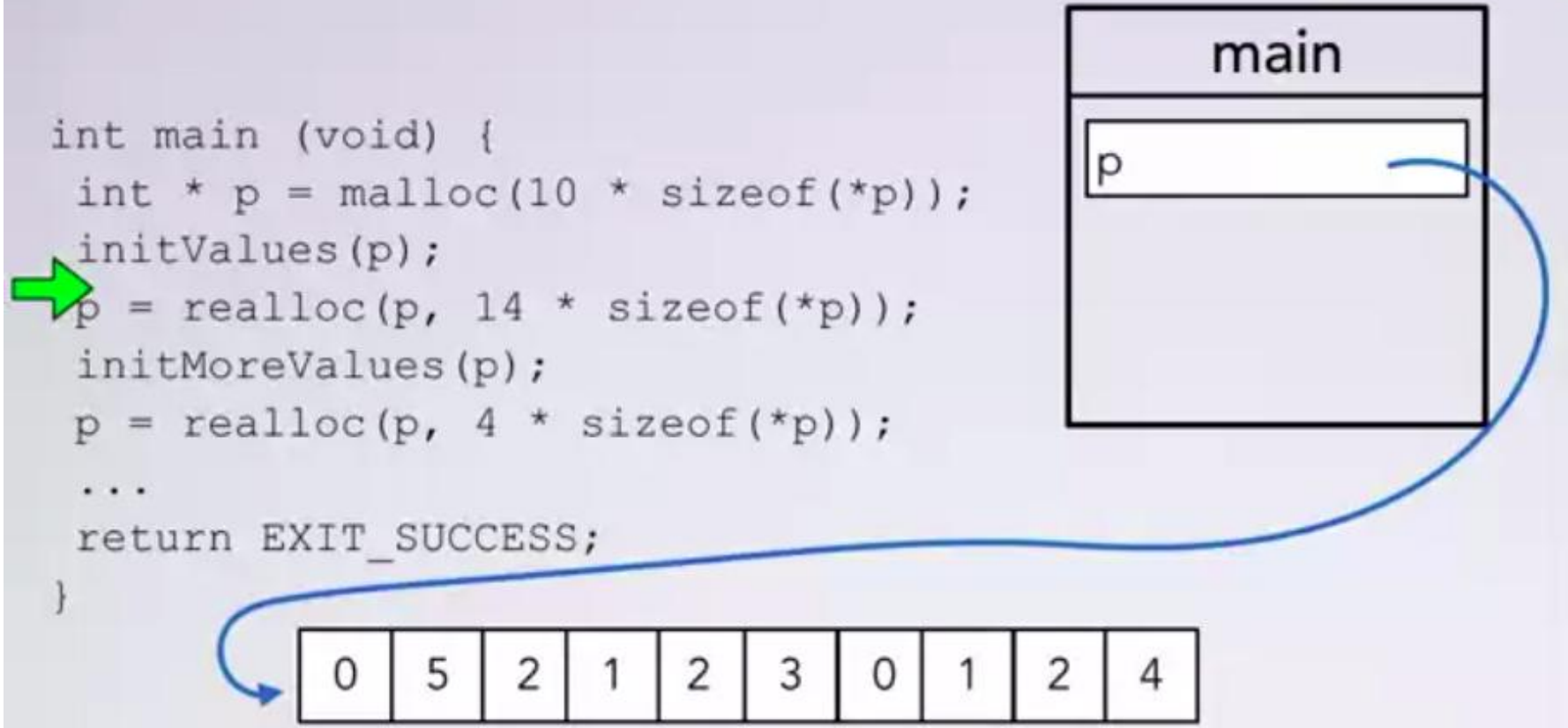


realloc = malloc + copy+ free

One Possible Execution

Next, we are going to fill these ints with

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```

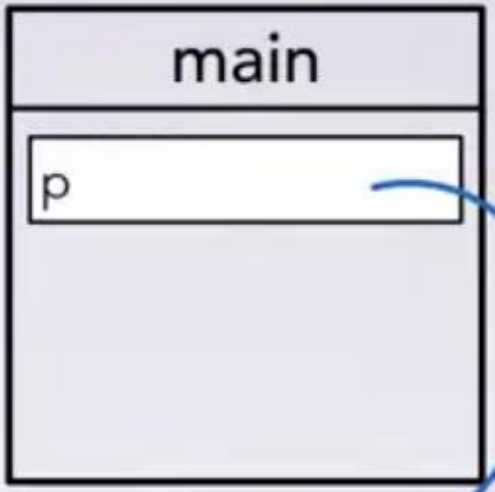


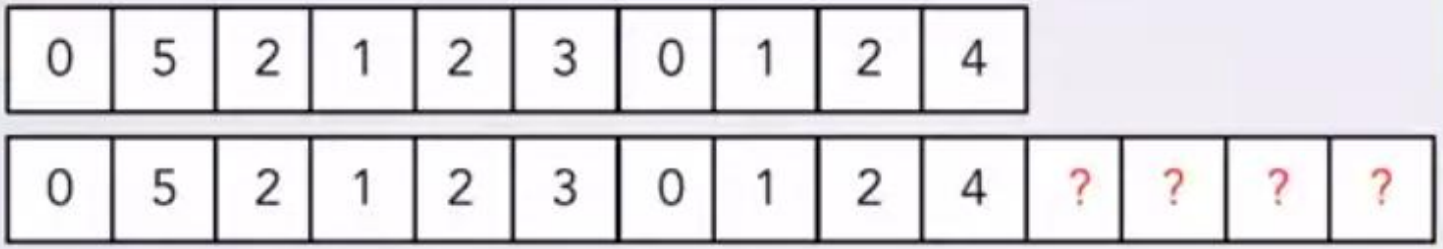
realloc = malloc + copy+ free

One Possible Execution

14 ints instead of just 10 so we cal

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
  = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```



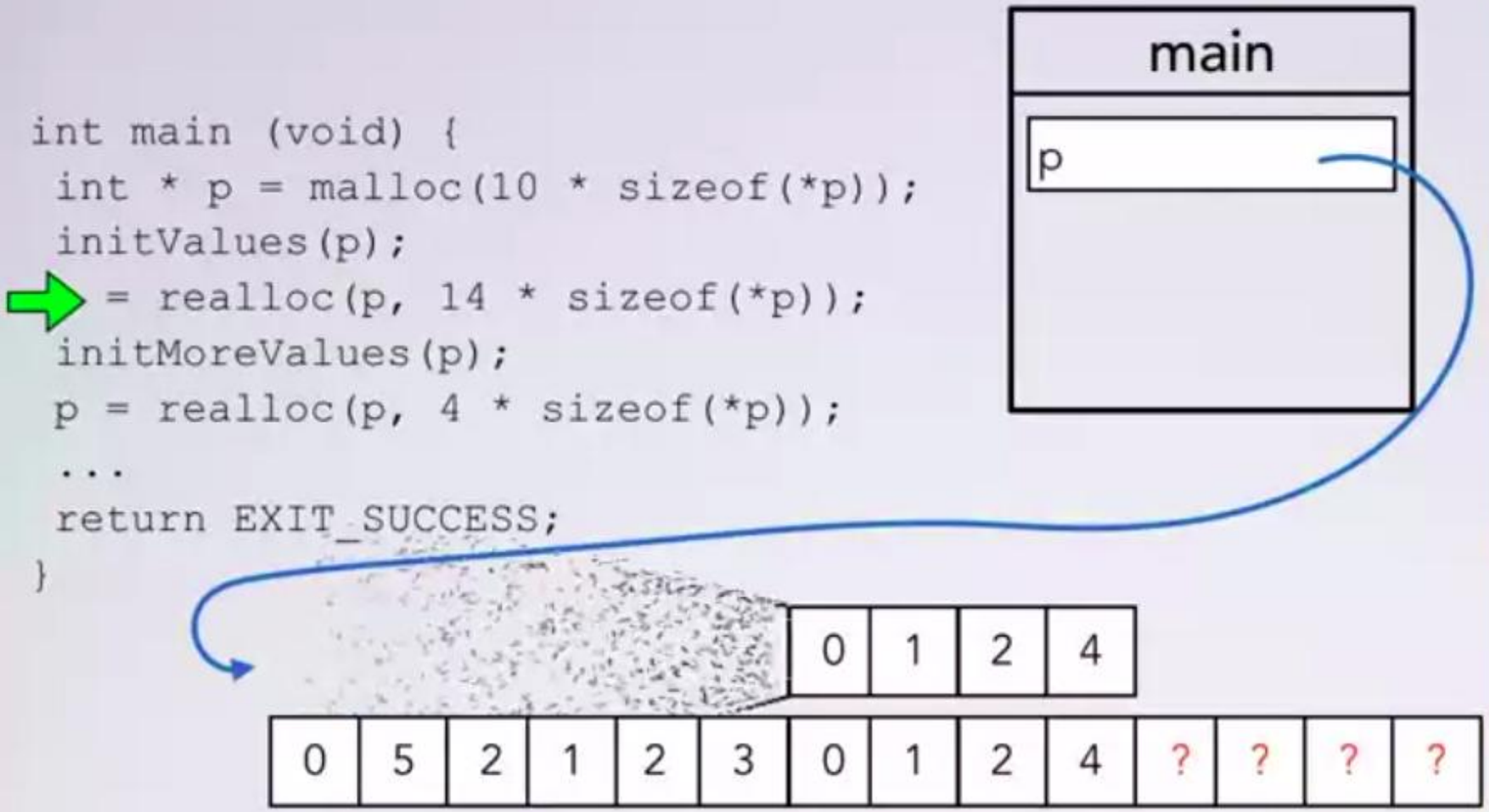


realloc = malloc + copy+ free

One Possible Execution

Notice that it only copied 10 elem

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```

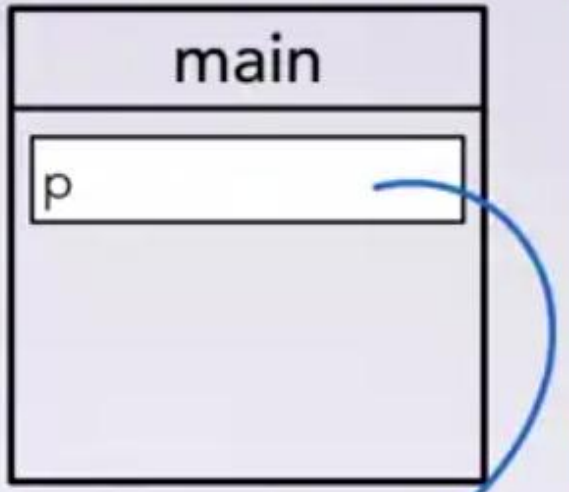


realloc = malloc + copy+ free

One Possible Execution

Last, realloc is going to free the original space whic

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```



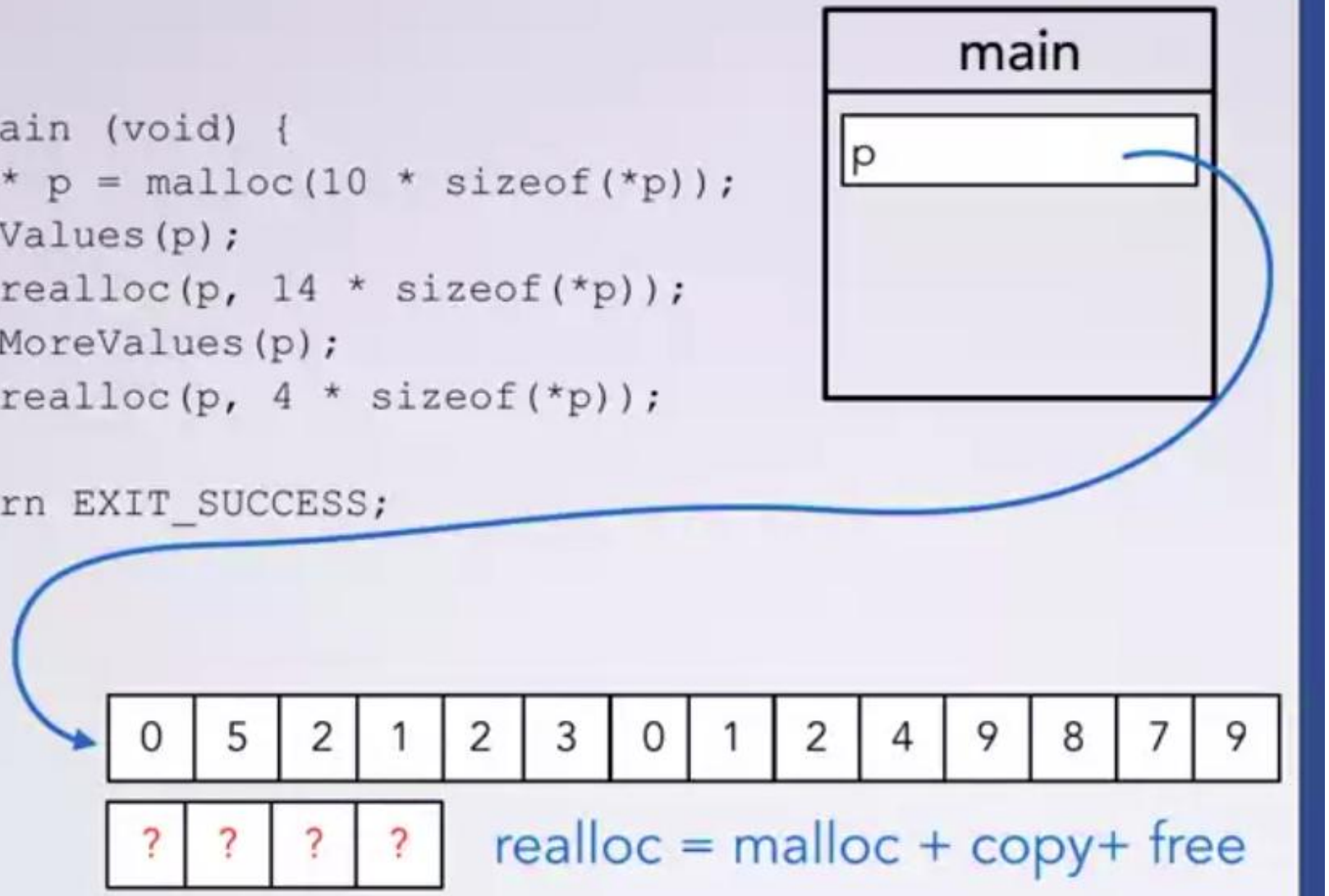
return value of realloc

realloc = malloc + copy+ free

One Possible Execution

some more initialization to fill in these othe

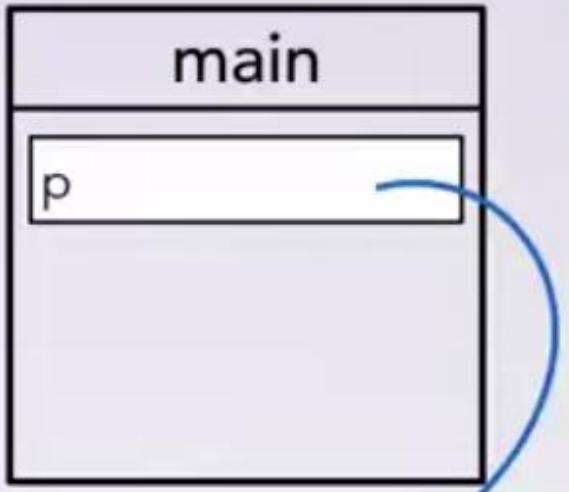
```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```

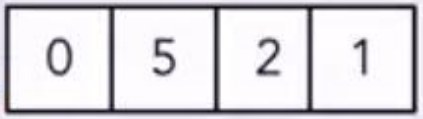


One Possible Execution

Then, you copy the first four ints into the

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```





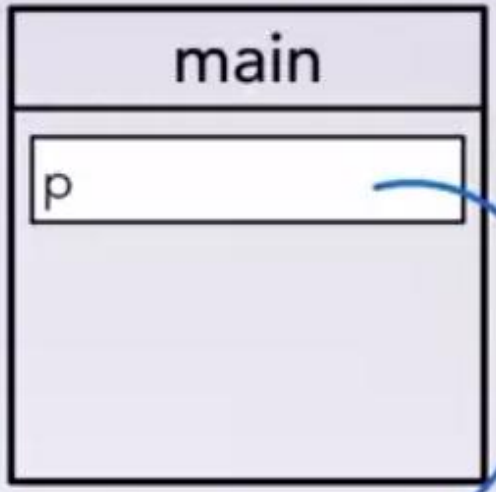
realloc = malloc + copy+ free

return value of realloc

One Possible Execution

the assignment statement updates P to point at this

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
   = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```



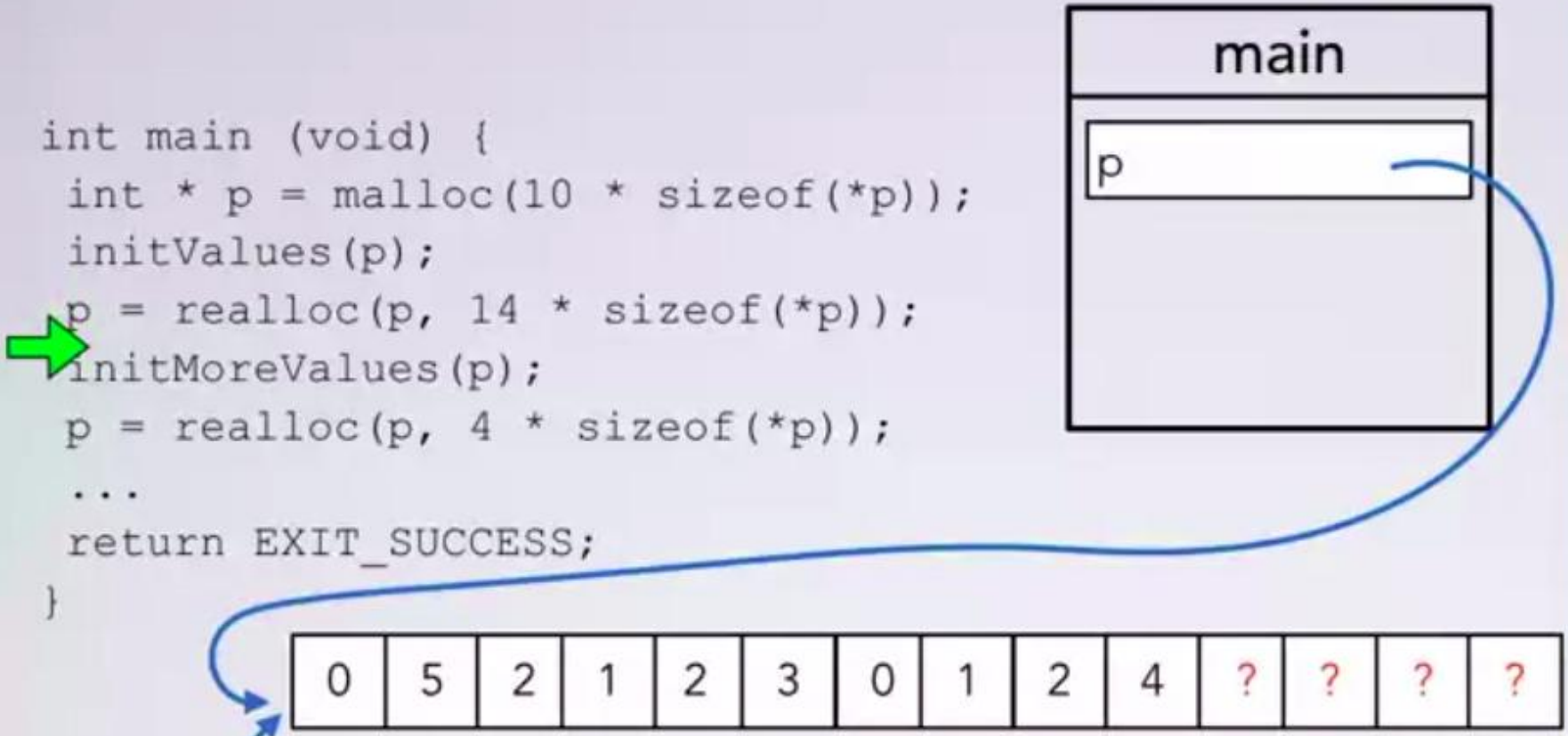
return value of realloc

realloc ***may*** re-use same space

Another Possible Execution

As before, realloc returns a pointer to the

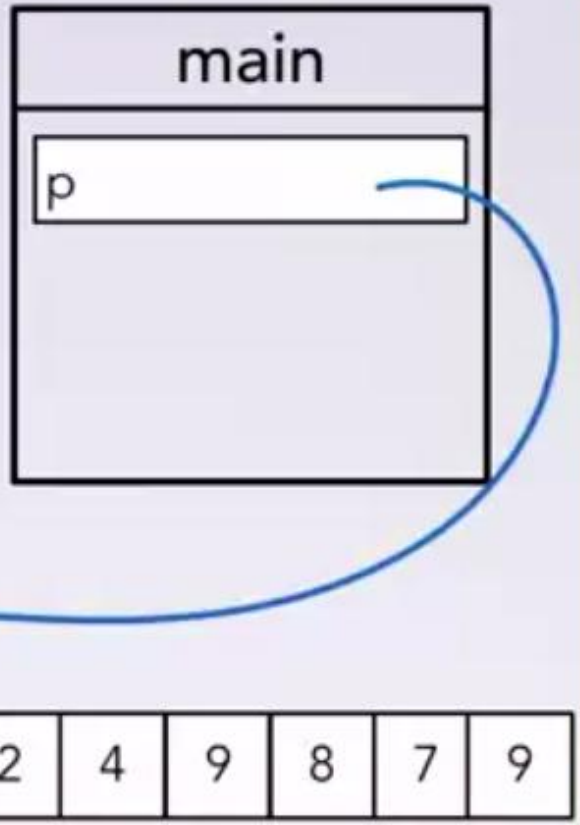
```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```



realloc ***may*** re-use same space

Our assignment statement will then update P to point as before.

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```

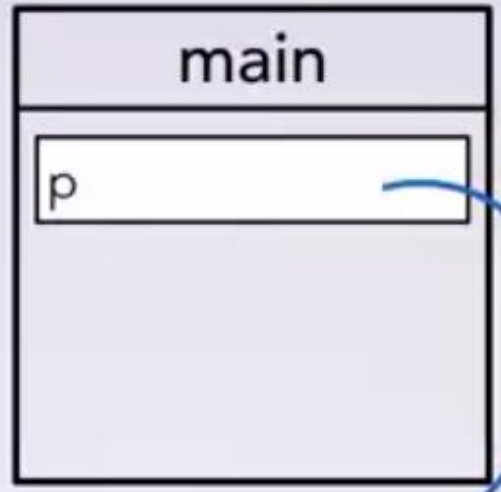


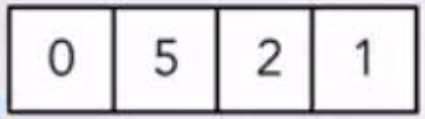
realloc ***may*** re-use same space

Another Possible Execution

reducing the size of this allocation to the newl

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```





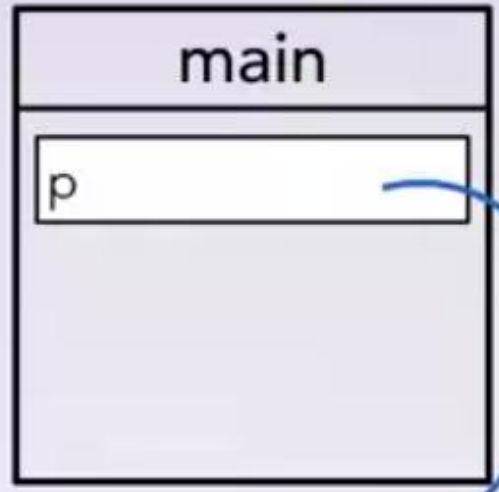
return value of realloc

realloc ***may*** re-use same space

Another Possible Execution

the particular implementation of realloc as

```
int main (void) {
 int * p = malloc(10 * sizeof(*p));
 initValues(p);
 p = realloc(p, 14 * sizeof(*p));
 initMoreValues(p);
 p = realloc(p, 4 * sizeof(*p));
 ...
 return EXIT_SUCCESS;
}
```

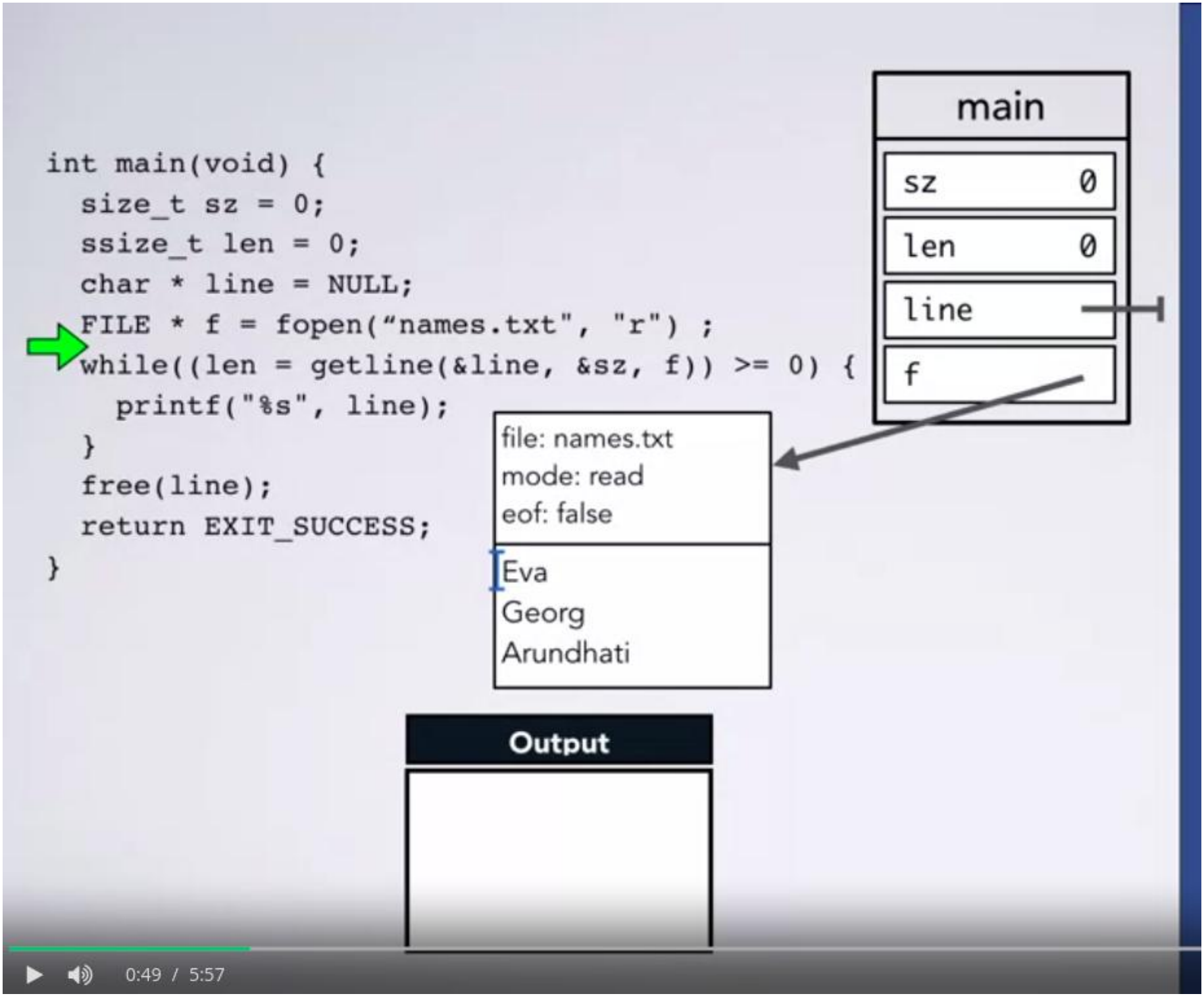


return value of realloc

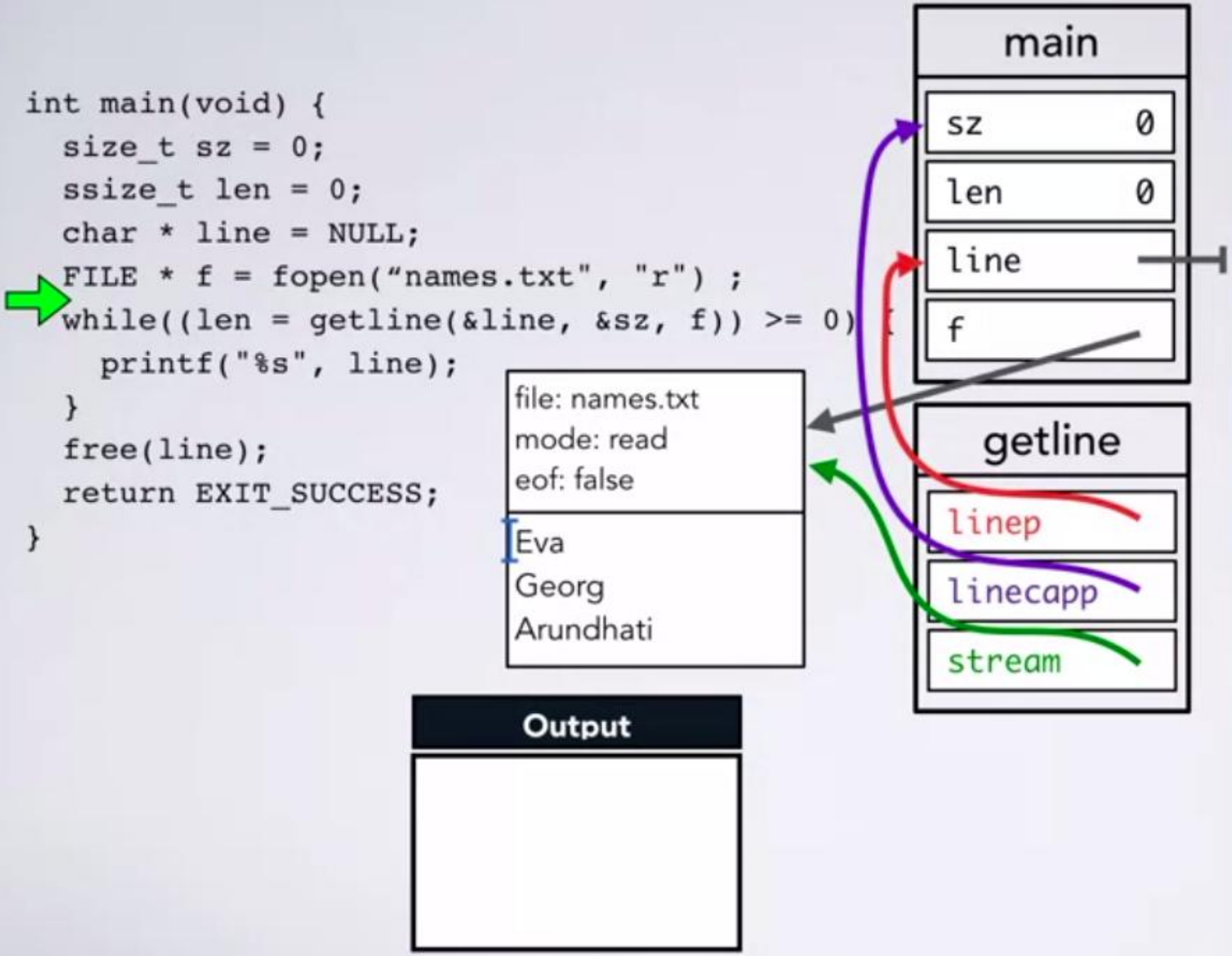
realloc ***may*** re-use same space

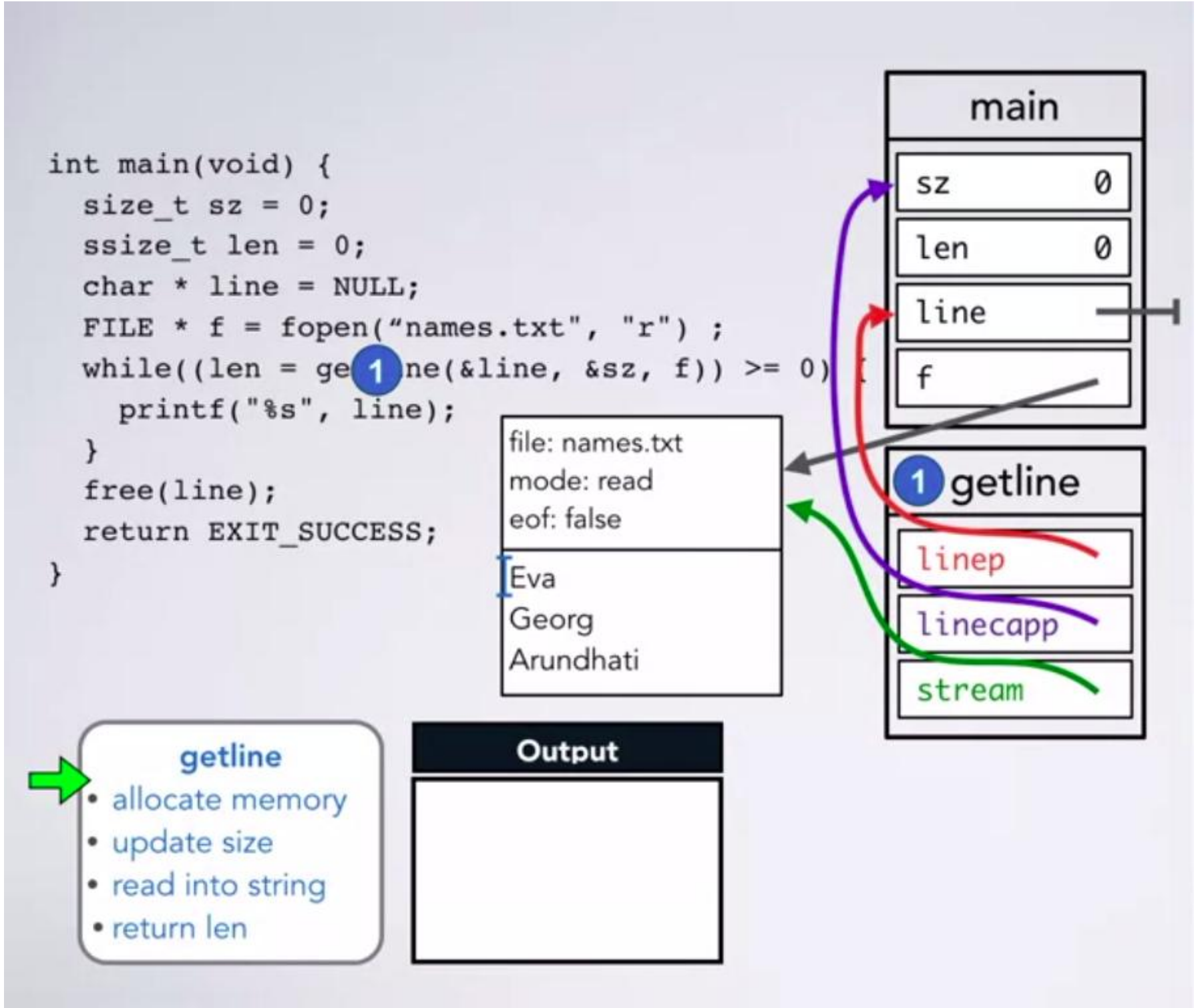
Another Possible Execution

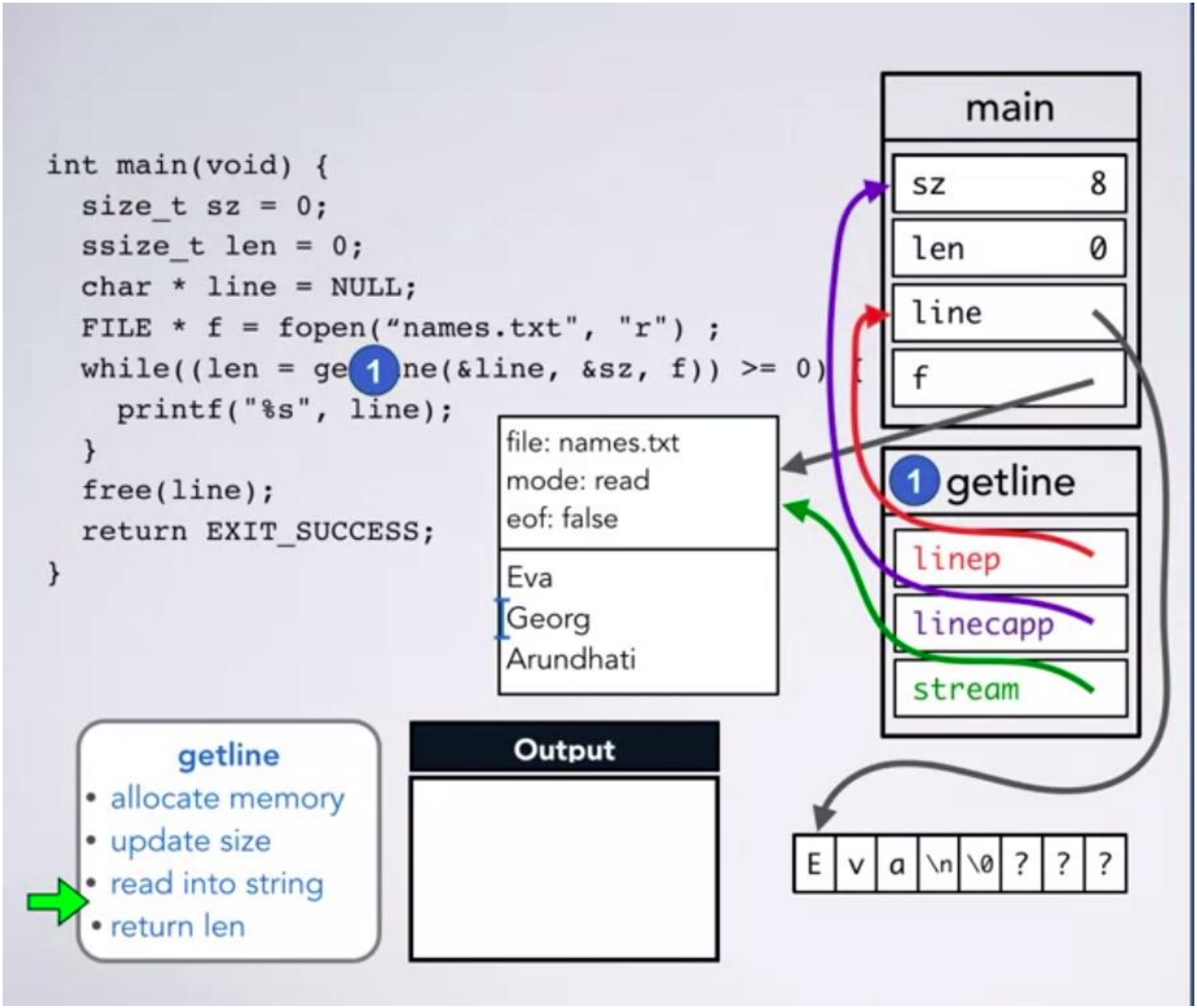
must always write code that expects realloc t


int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen(“names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
main
sz 0
len 0
line
f
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
Output
0:49 / 5:57

```c
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen("names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

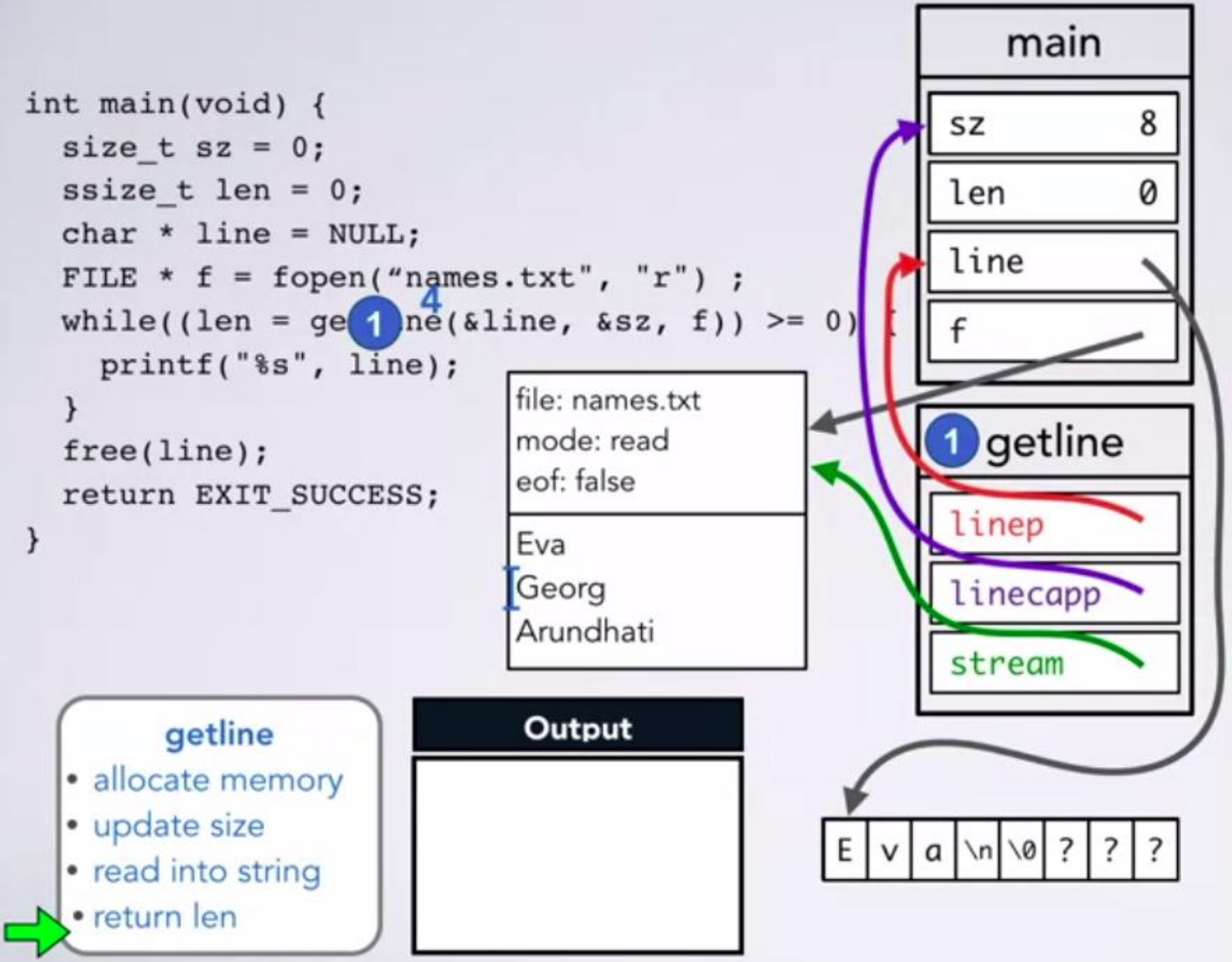



int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen(“names.txt", "r") ;
while((len = ge 1 ne(&line, &sz, f)) >= 0)
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
main
sz 0
len 0
line
f
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
1 getline
linep
linecapp
stream
getline
• allocate memory
• update size
• read into string
• return len
Output

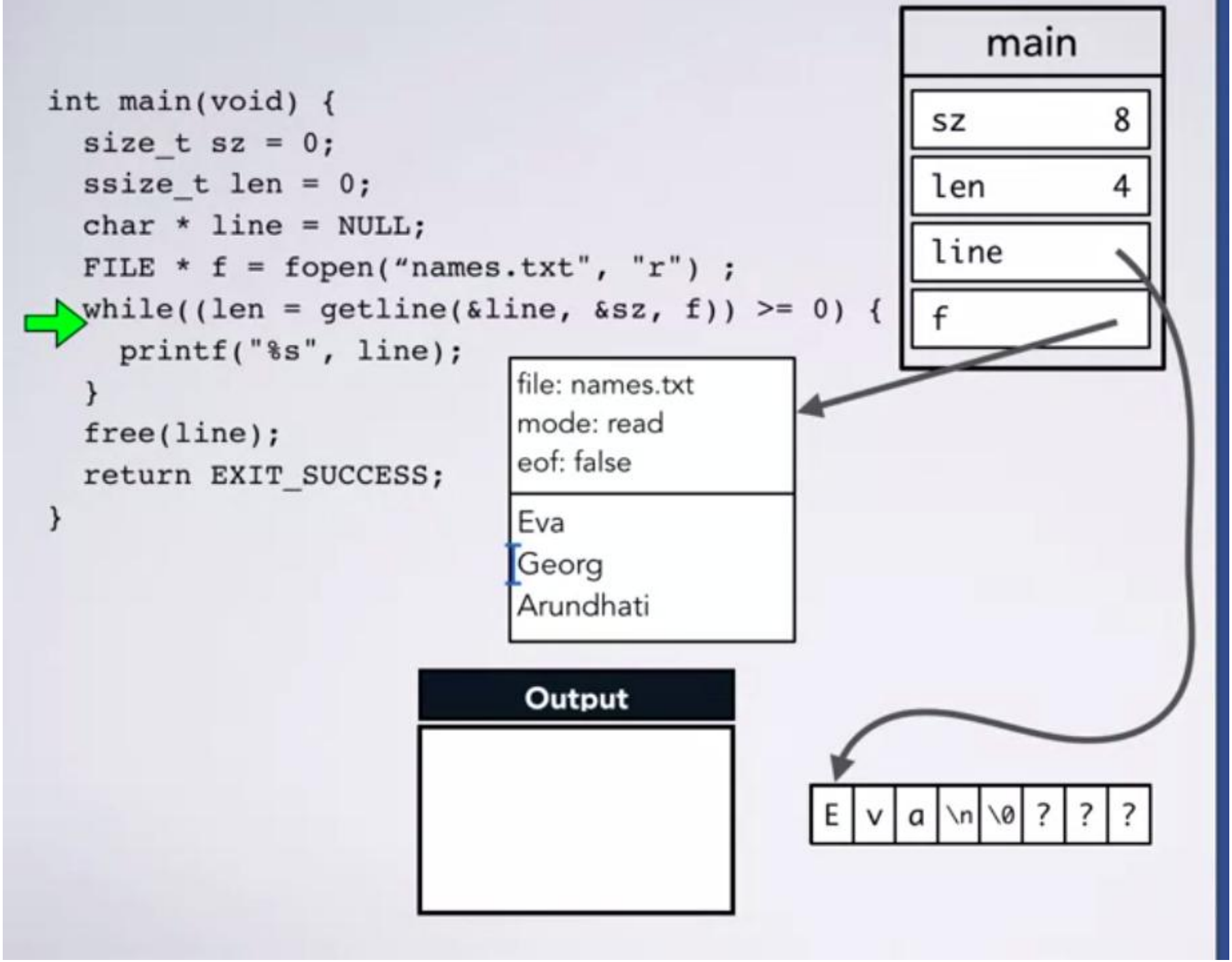

int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen("names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
1
main
sz 8
len 0
line
f
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
1 getline
linep
linecapp
stream
E v a \n \0 ? ? ?
getline
• allocate memory
• update size
• read into string
• return len
Output

```
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen("names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

1 4

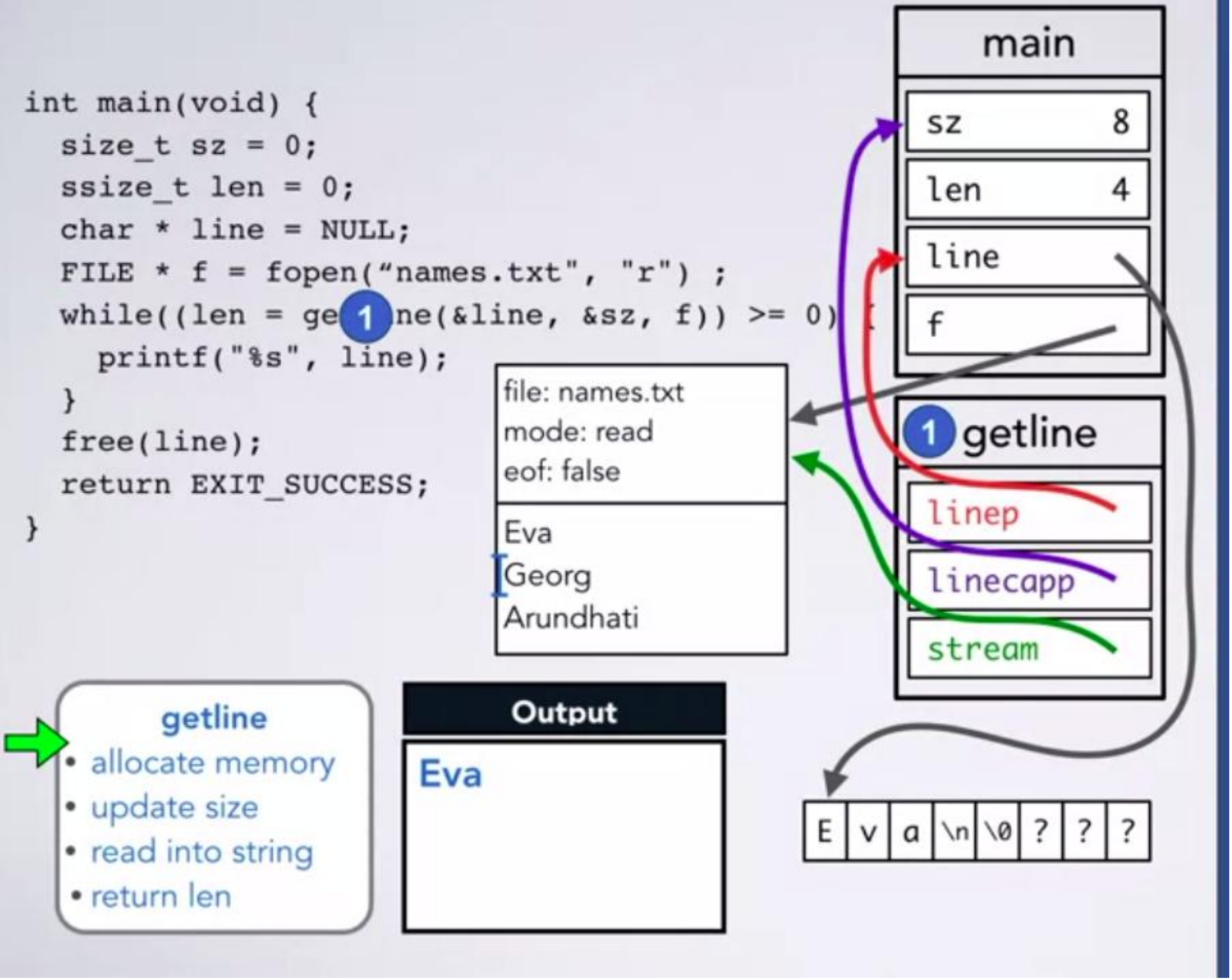


```
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen(“names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

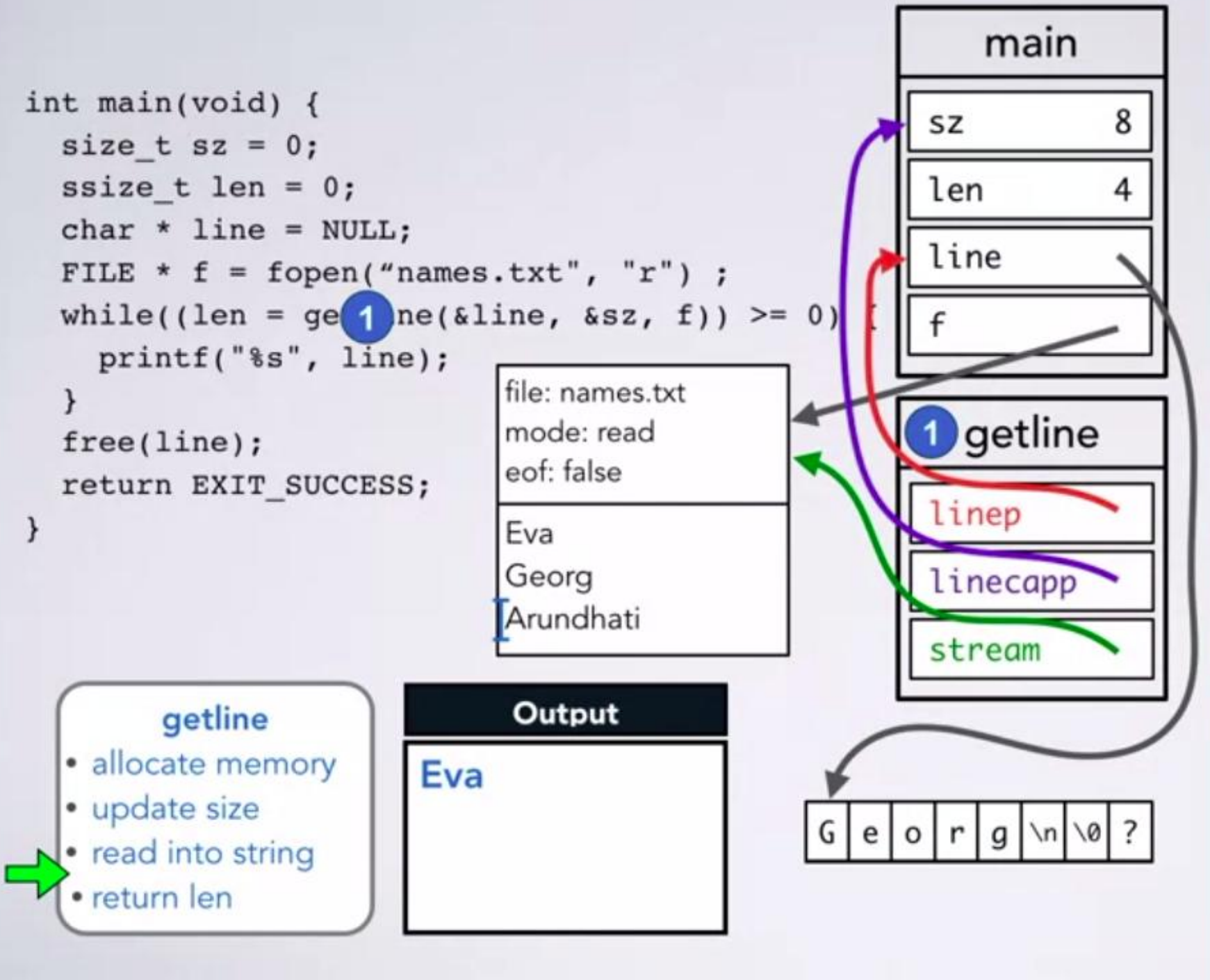


```
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen("names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

1



```
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen("names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

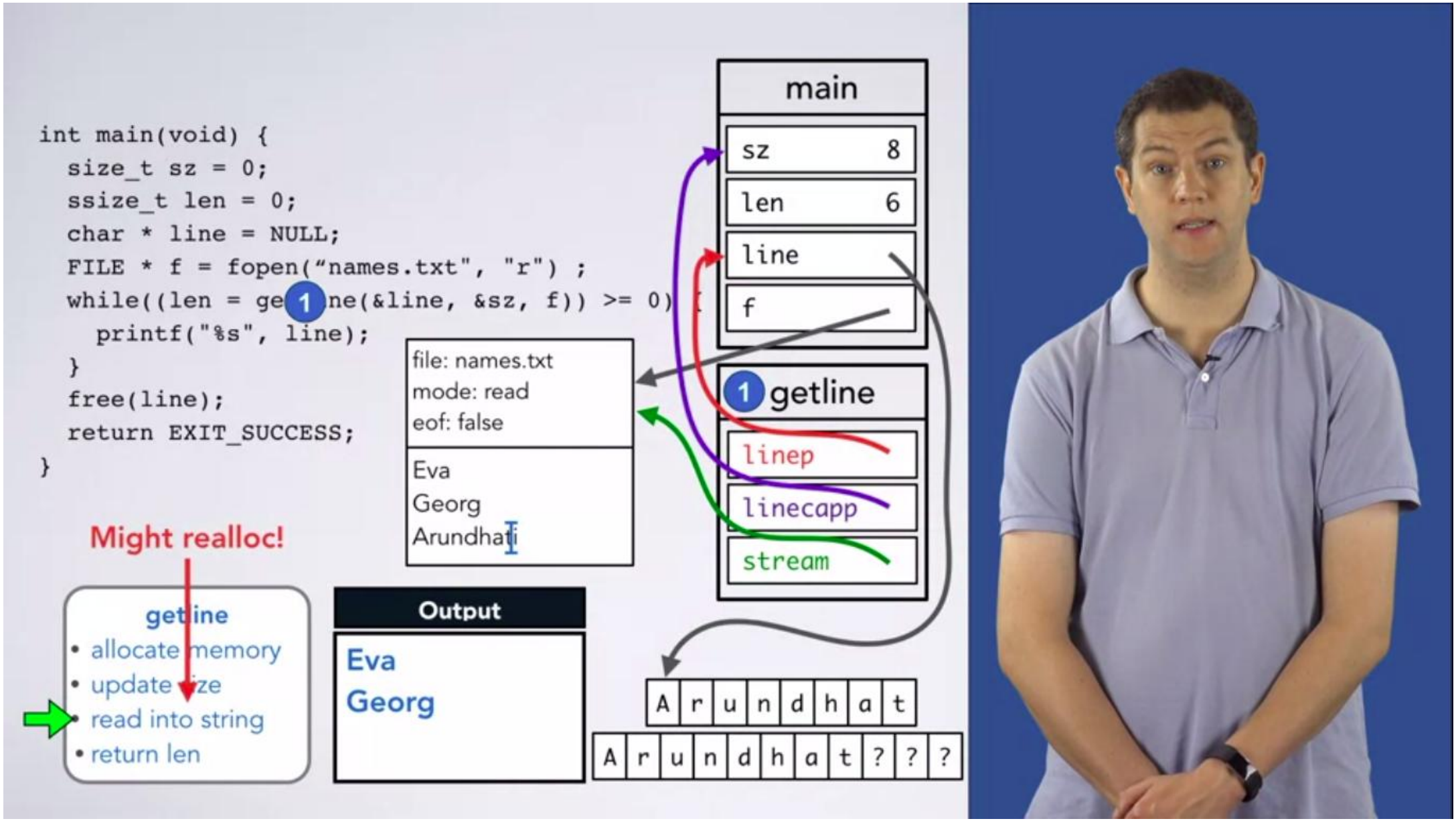


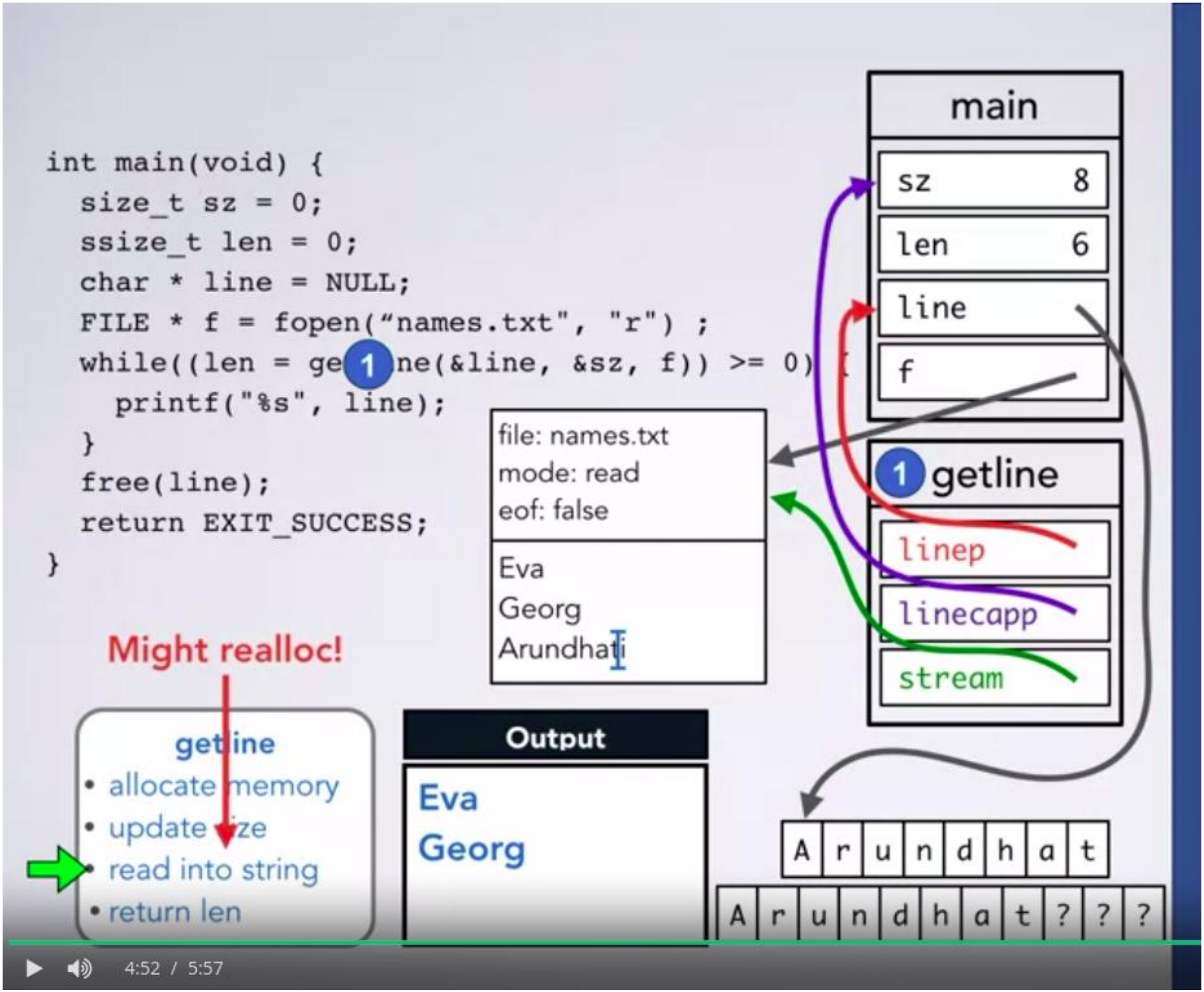
getline

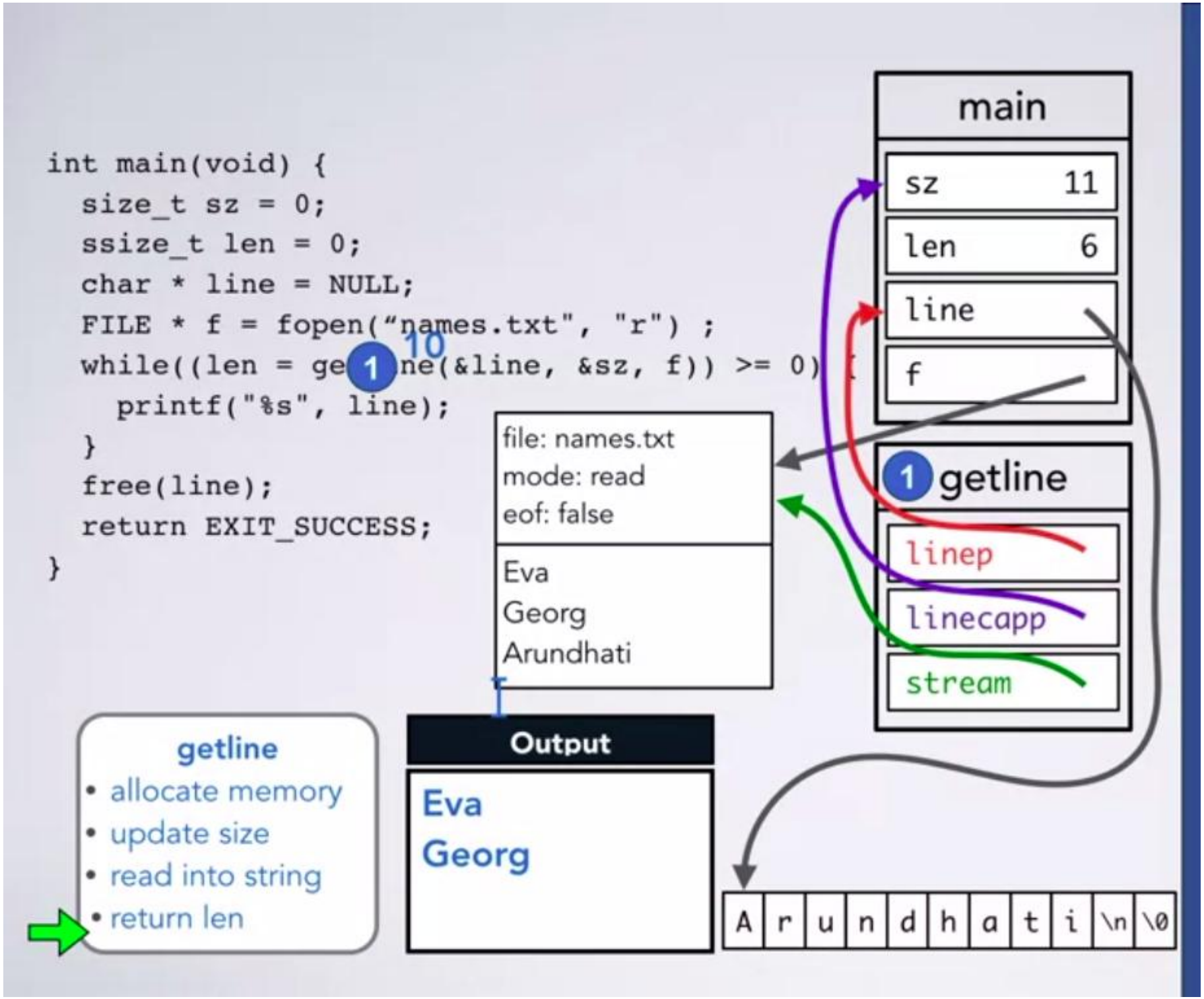
- allocate memory
- update size
- read into string
- return len

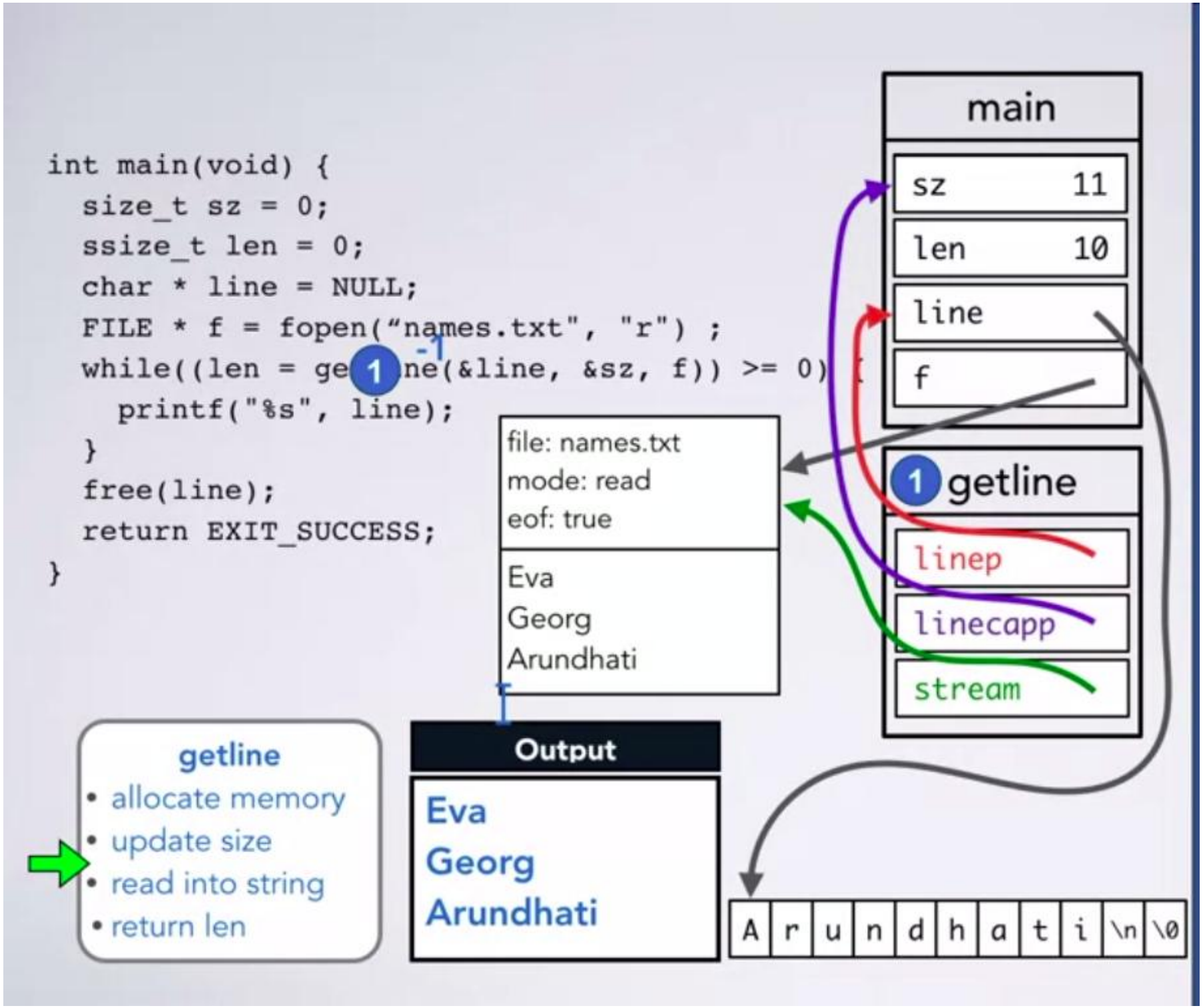
Output

Eva


int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen("names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
1
main
sz 8
len 6
line
f
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
1 getline
linep
linecapp
stream
Might realloc!
getline
• allocate memory
• update size
• read into string
• return len
Output
Eva
Georg
A r u n d h a t
A r u n d h a t ? ? ?


int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen("names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
1
main
sz 8
len 6
line
f
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
1 getline
linep
linecapp
stream
Might realloc!
getline
• allocate memory
• update size
• read into string
• return len
Output
Eva
Georg
A r u n d h a t
A r u n d h a t ? ? ?
4:52 / 5:57


int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen("names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
main
sz 11
len 6
line
f
1 getline
linep
linecapp
stream
file: names.txt
mode: read
eof: false
Eva
Georg
Arundhati
Output
Eva
Georg
getline
• allocate memory
• update size
• read into string
• return len
A r u n d h a t i \n \0


int main(void) {
size_t sz = 0;
ssize_t len = 0;
char * line = NULL;
FILE * f = fopen("names.txt", "r") ;
while((len = getline(&line, &sz, f)) >= 0) {
printf("%s", line);
}
free(line);
return EXIT_SUCCESS;
}
1
main
sz 11
len 10
line
f
1 getline
linep
linecapp
stream
file: names.txt
mode: read
eof: true
Eva
Georg
Arundhati
getline
• allocate memory
• update size
• read into string
• return len
Output
Eva
Georg
Arundhati
A r u n d h a t i \n \0

```
int main(void) {
  size_t sz = 0;
  ssize_t len = 0;
  char * line = NULL;
  FILE * f = fopen("names.txt", "r") ;
  while((len = getline(&line, &sz, f)) >= 0) {
    printf("%s", line);
  }
  free(line);
  return EXIT_SUCCESS;
}
```

```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

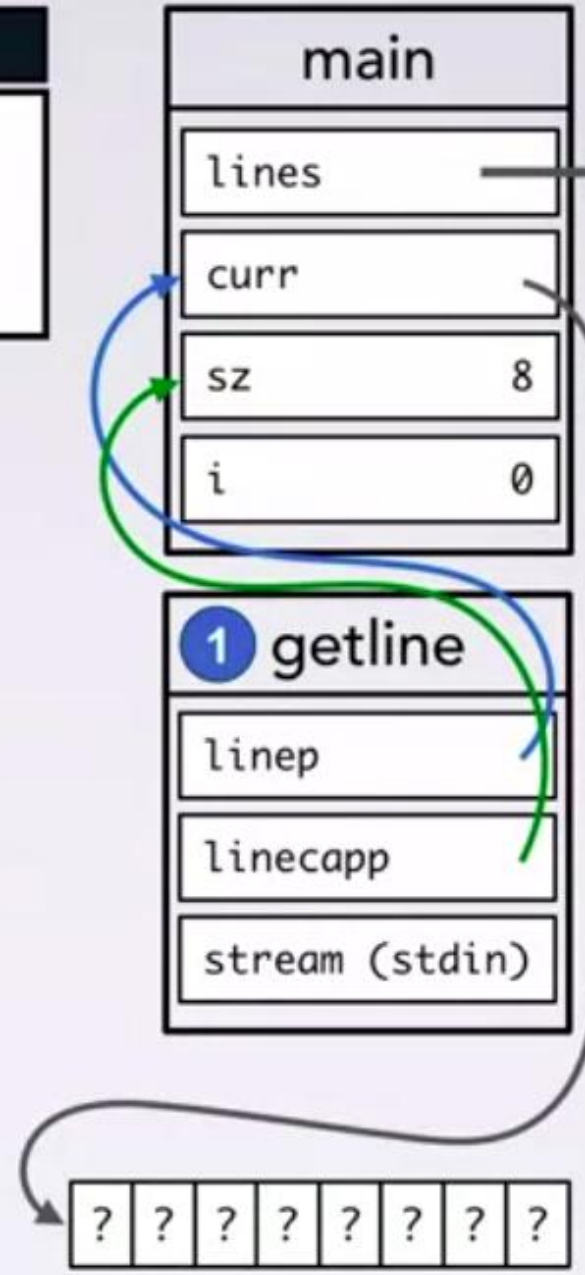
```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (ge 1 ne(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

somewhere in the C library

Input (stdin)

cat
ant
bread

Output

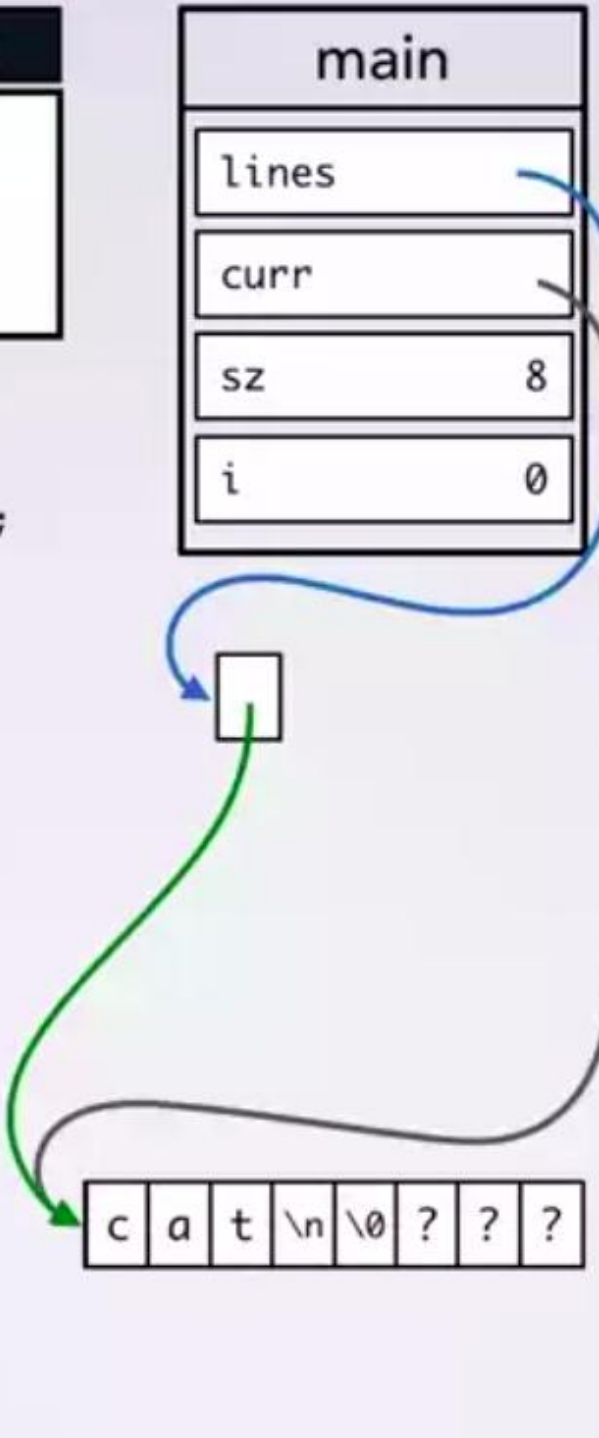


```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

Input (stdin)

cat
ant
bread

Output

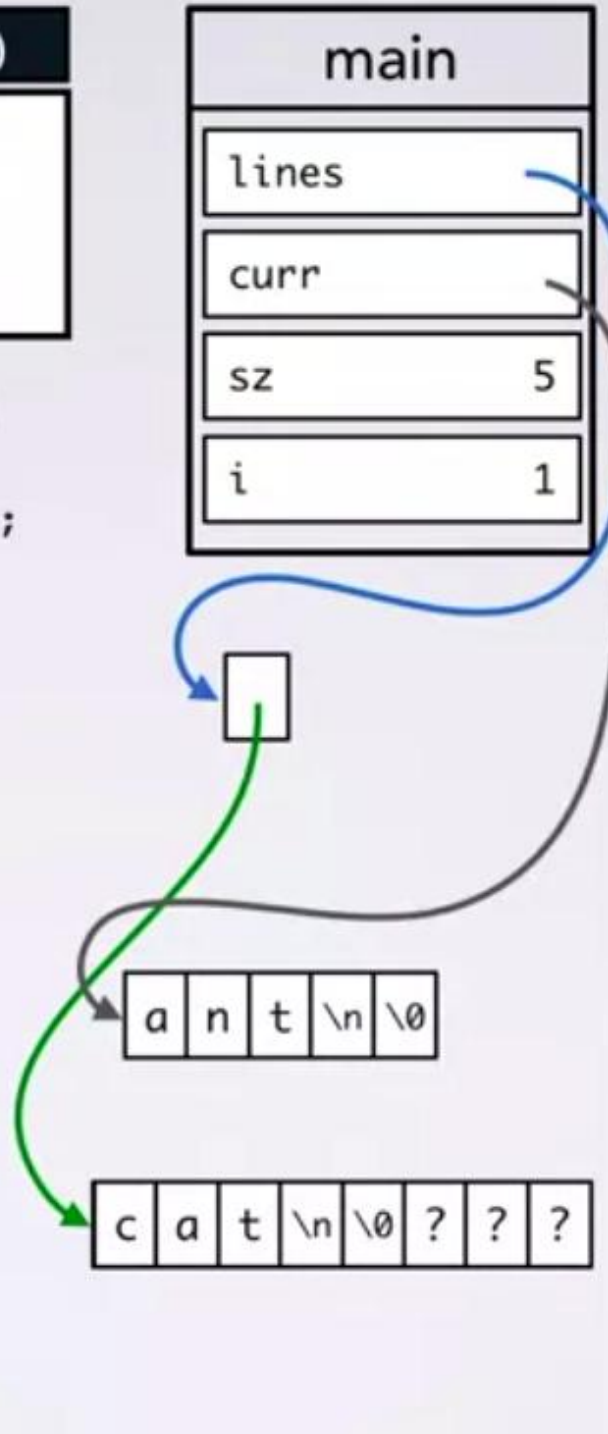


```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

Input (stdin)

cat
ant
bread

Output

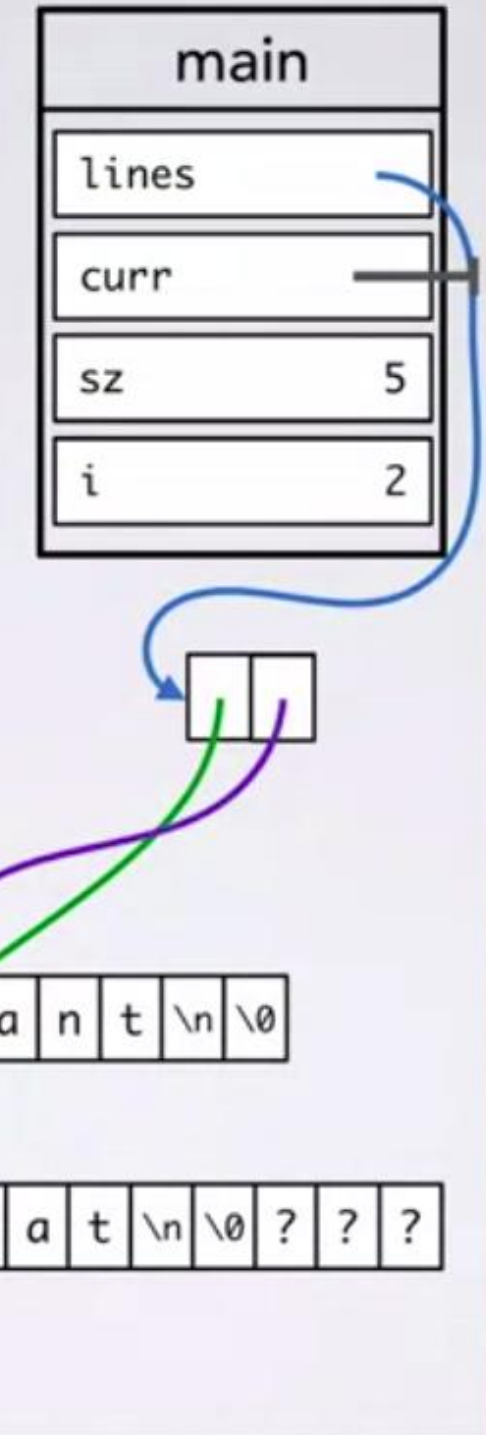


```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

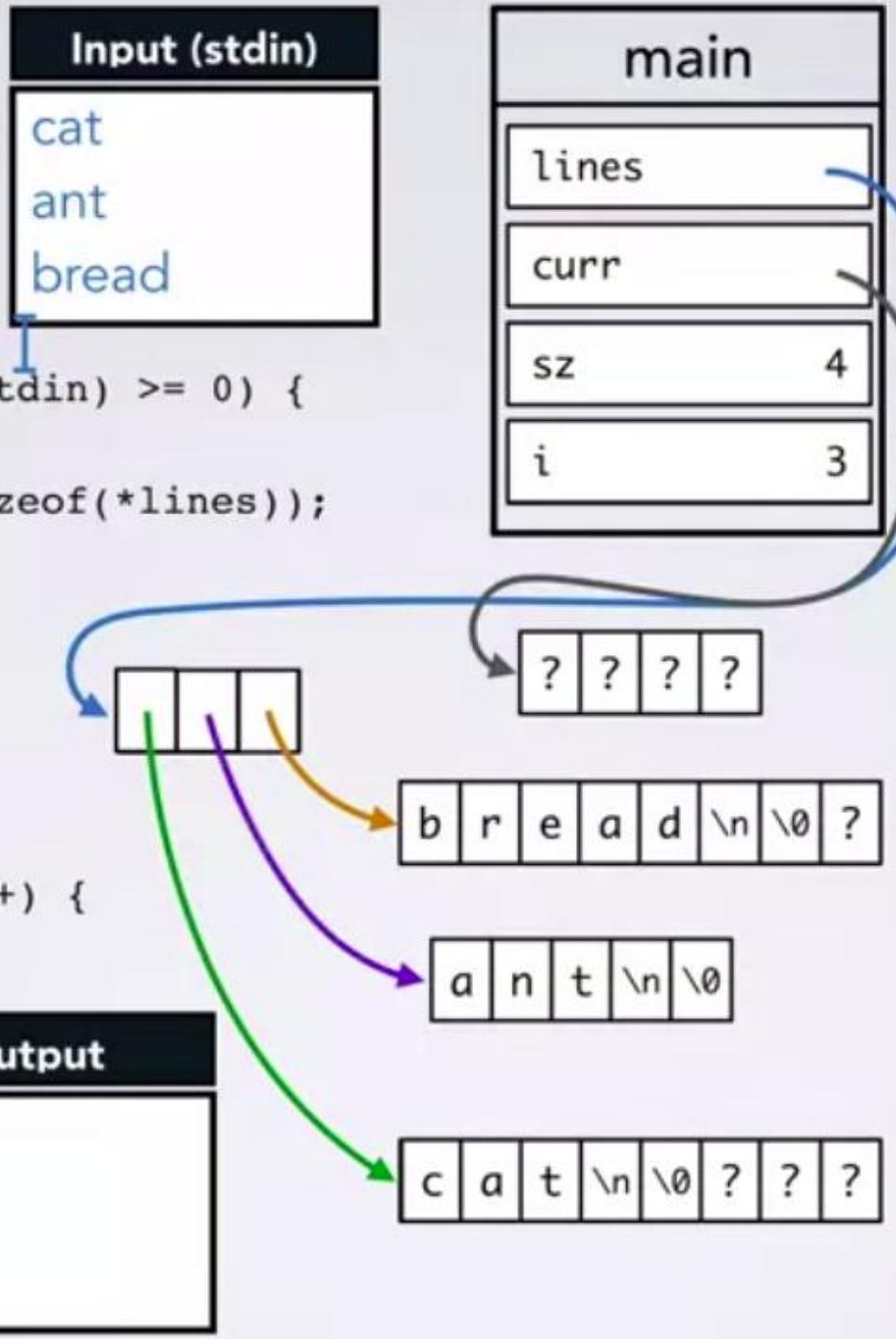
Input (stdin)

cat
ant
bread

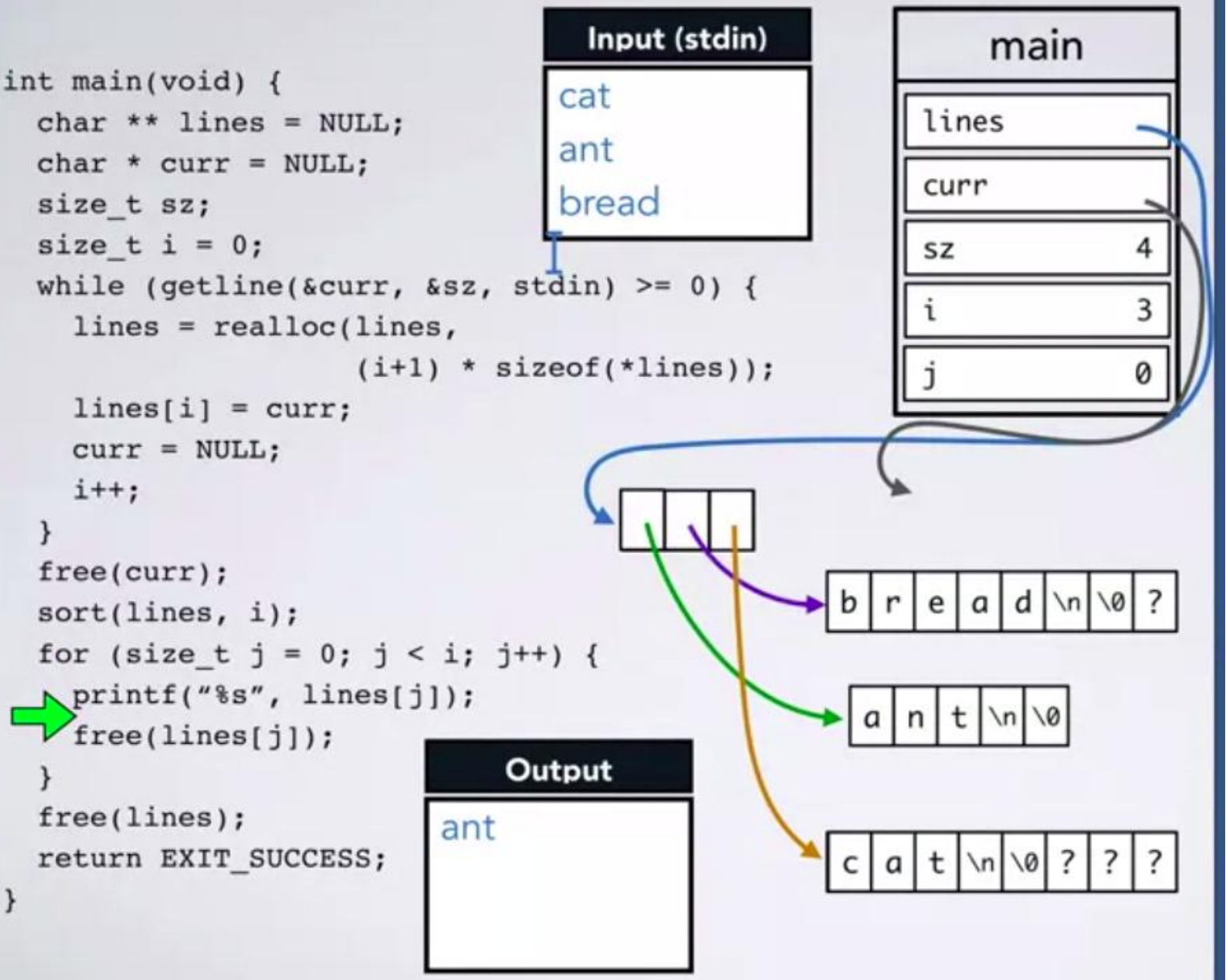
Output



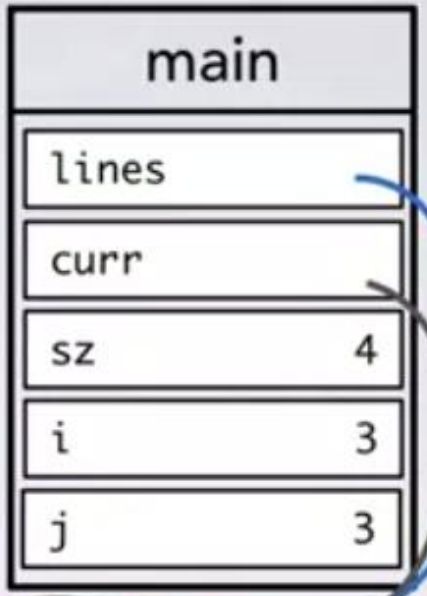
```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```

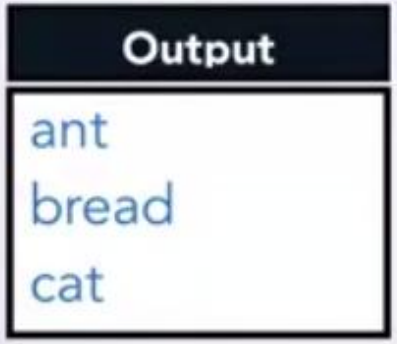


```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```



```
int main(void) {
  char ** lines = NULL;
  char * curr = NULL;
  size_t sz;
  size_t i = 0;
  while (getline(&curr, &sz, stdin) >= 0) {
    lines = realloc(lines,
                    (i+1) * sizeof(*lines));
    lines[i] = curr;
    curr = NULL;
    i++;
  }
  free(curr);
  sort(lines, i);
  for (size_t j = 0; j < i; j++) {
    printf("%s", lines[j]);
    free(lines[j]);
  }
  free(lines);
  return EXIT_SUCCESS;
}
```





Onwards to larger programs!

Critical principles for moderately sized programs

Starting point to continue on in Software Engineering

Step 1: Work an Example

Input:

```
3
Romeo
2
Duel300
Romance101
Juliet
2
Romance101
Poison352
Tybalt
1
Duel300
```

Students:

Romeo	Juliet	Tybalt
Duel300 Romance101	Romance101 Poison352	Duel300

All Classes:

Duel300 Romance101 Poison352

Step 1: Work an Example

Input:

```
3
Romeo
2
Duel300
Romance101
Juliet
2
Romance101
Poison352
Tybalt
1
Duel300
```

Students:

Romeo	Juliet	Tybalt
Duel300 Romance101	Romance101 Poison352	Duel300

All Classes:

Duel300 Romance101 Poison352

Rosters:

Duel300	Romance101	Poison352
Romeo Tybalt	Romeo Juliet	Juliet

Step 2: Write Down What You Did

Input:

```
3
Romeo
2
Duel300
Romance101
Juliet
2
Romance101
Poison352
Tybalt
1
Duel300
```

Students:

Romeo	Juliet	Tybalt
Duel300 Romance101	Romance101 Poison352	Duel300

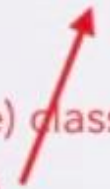
All Classes:

Duel300 Romance101 Poison352

Rosters:

Duel300	Romance101	Poison352
Romeo Tybalt	Romeo Juliet	Juliet

1. Read the input
2. Made a list of all of the (unique) class names
3. Wrote one output file per class



Step 3: Generalize

Input:

```
3
Romeo
2
Duel300
Romance101
Juliet
2
Romance101
Poison352
Tybalt
1
Duel300
```

Students:

Romeo	Juliet	Tybalt
Duel300 Romance101	Romance101 Poison352	Duel300

All Classes:

Duel300 Romance101 Poison352

Rosters:

Duel300	Romance101	Poison352
Romeo Tybalt	Romeo Juliet	Juliet

1. Read the input from the file named by argv[1] (call it `the_roster`)
2. Make a list of all of the (unique) class names (call it `unique_class_list`)
3. Write one output file per class (from `unique_class_list` and `the_roster`)

(Step 4: Test)

1. Read the input from the file named by argv[1] (call it `the_roster`)
2. Make a list of all of the (unique) class names (call it `unique_class_list`)
3. Write one output file per class (from `unique_class_list` and `the_roster`)

Step 5: Translate to Code

1. Read the input from the file named by argv[1] (call it `the_roster`)
2. Make a list of all of the (unique) class names (call it `unique_class_list`)
3. Write one output file per class (from `unique_class_list` and `the_roster`)

Each of these is a complex step ➤ **abstract into a function**!

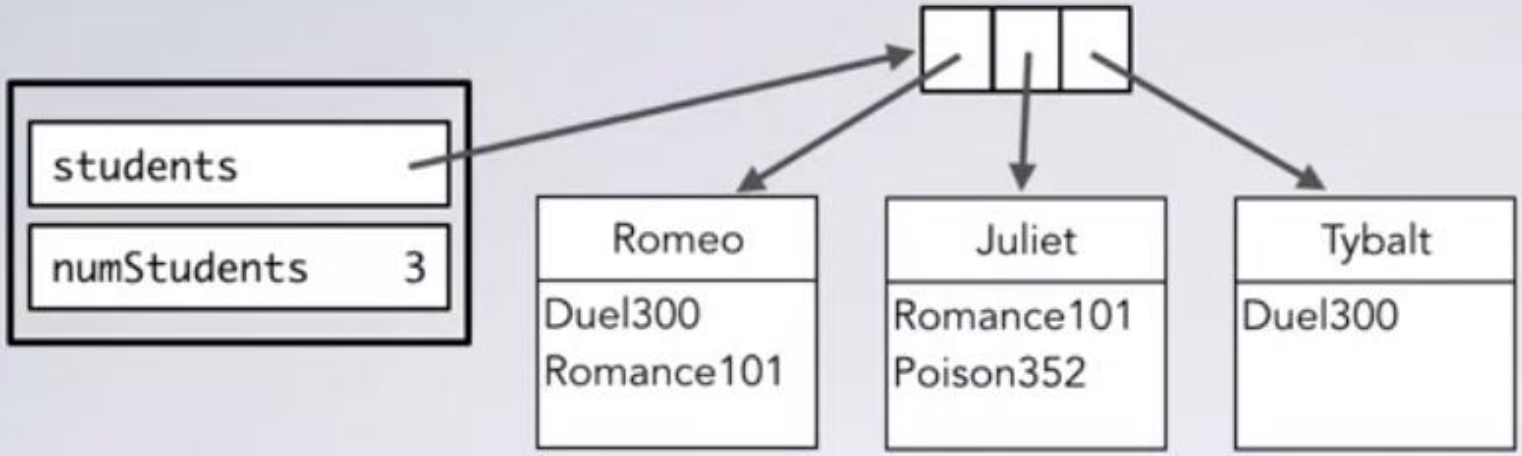
These few function calls (plus some error checking) make a good `main`.

Need to be a bit more precise with our drawing to see **types** that are needed.

E.g., what type does `readInput(argv[1])` return?

Conceptually, `readInput` returns a roster.

- Can name the type `roster_t`
- but what is the definition of `roster_t` ?

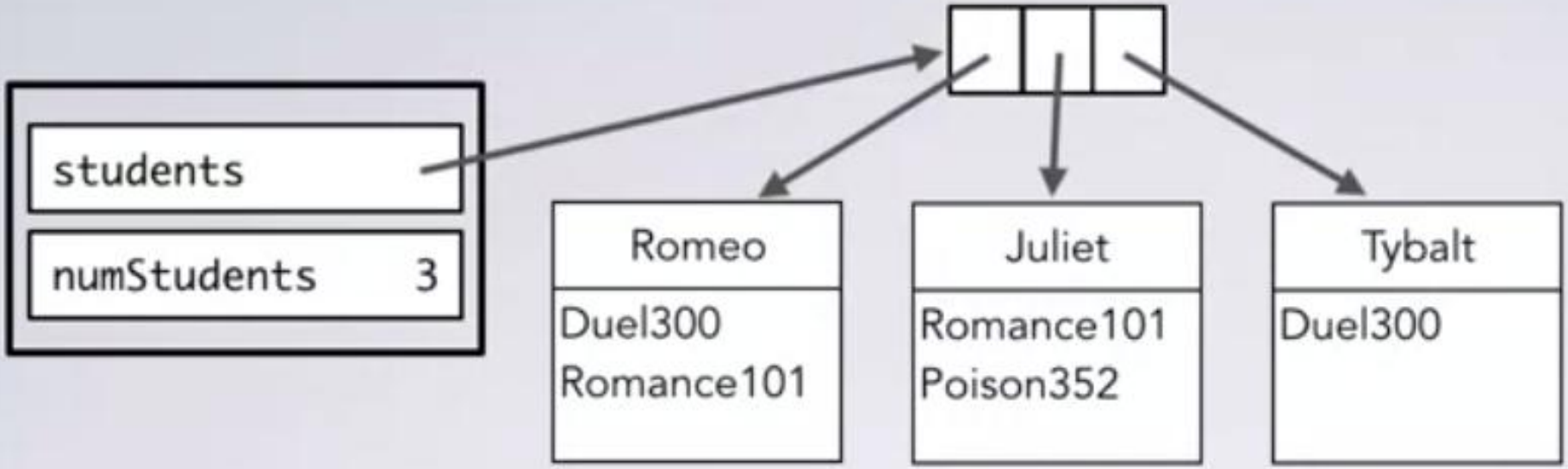


Multiple things (what kind of thing?) ➤ array
also need to keep track of how many

Need to think about what the "things" in the array are

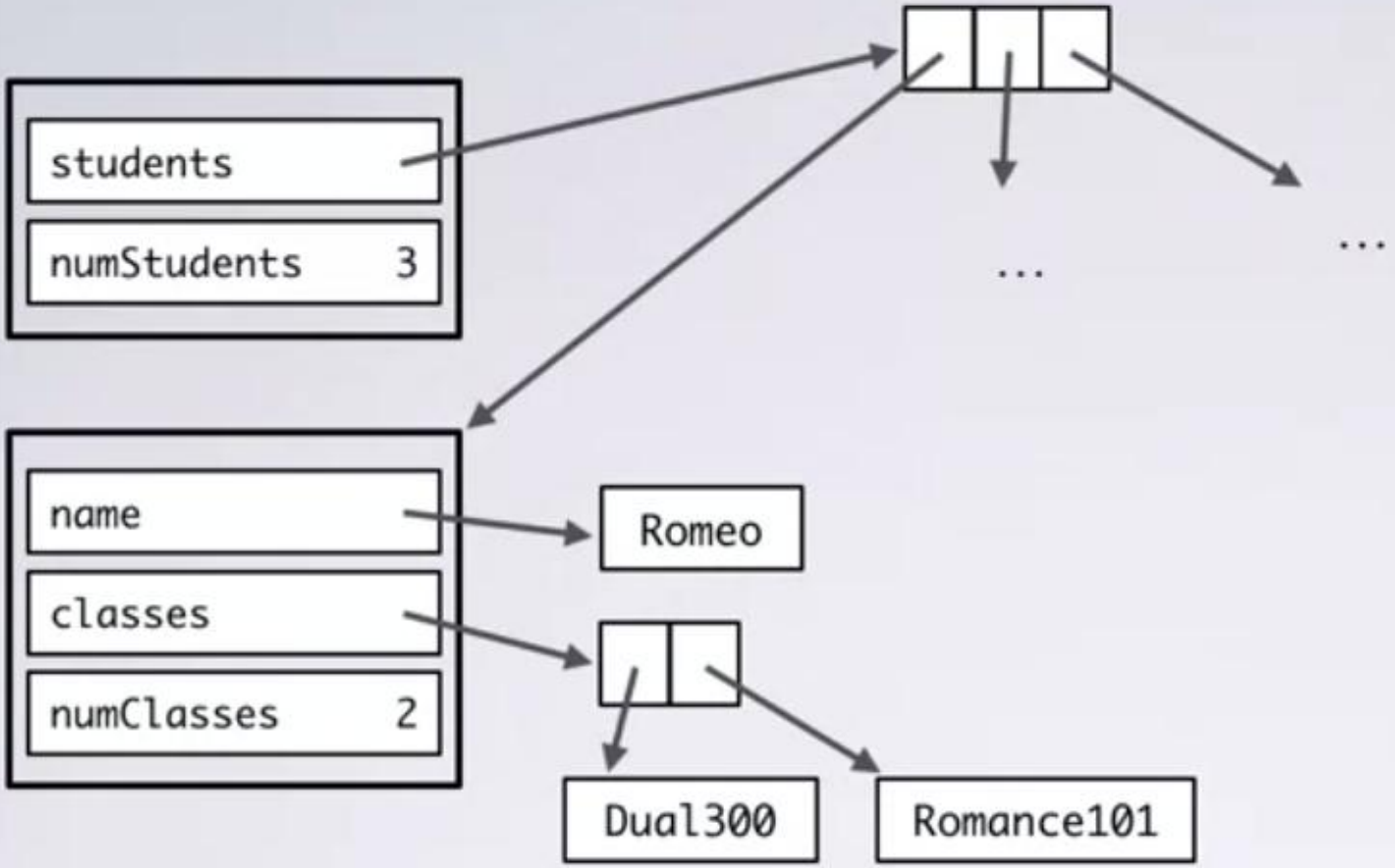
Each represents a student: call it a `student_t`

```
struct roster_tag {
  student_t ** students;
  int numStudents;
};
typedef struct roster_tag roster_t;
```



How about the details of a `student_t` ?

- has a name (string = `char *`)
- has classes (array of strings: `char **`, `int`)




```
struct student_tag {
  char * name;
  char ** classes;
  int numClasses;
};
typedef struct student_tag student_t;
```

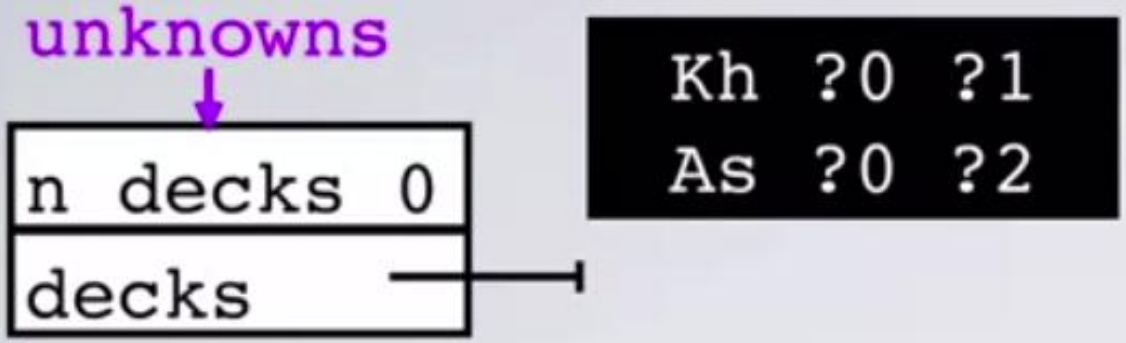
Poker Project: Final Steps

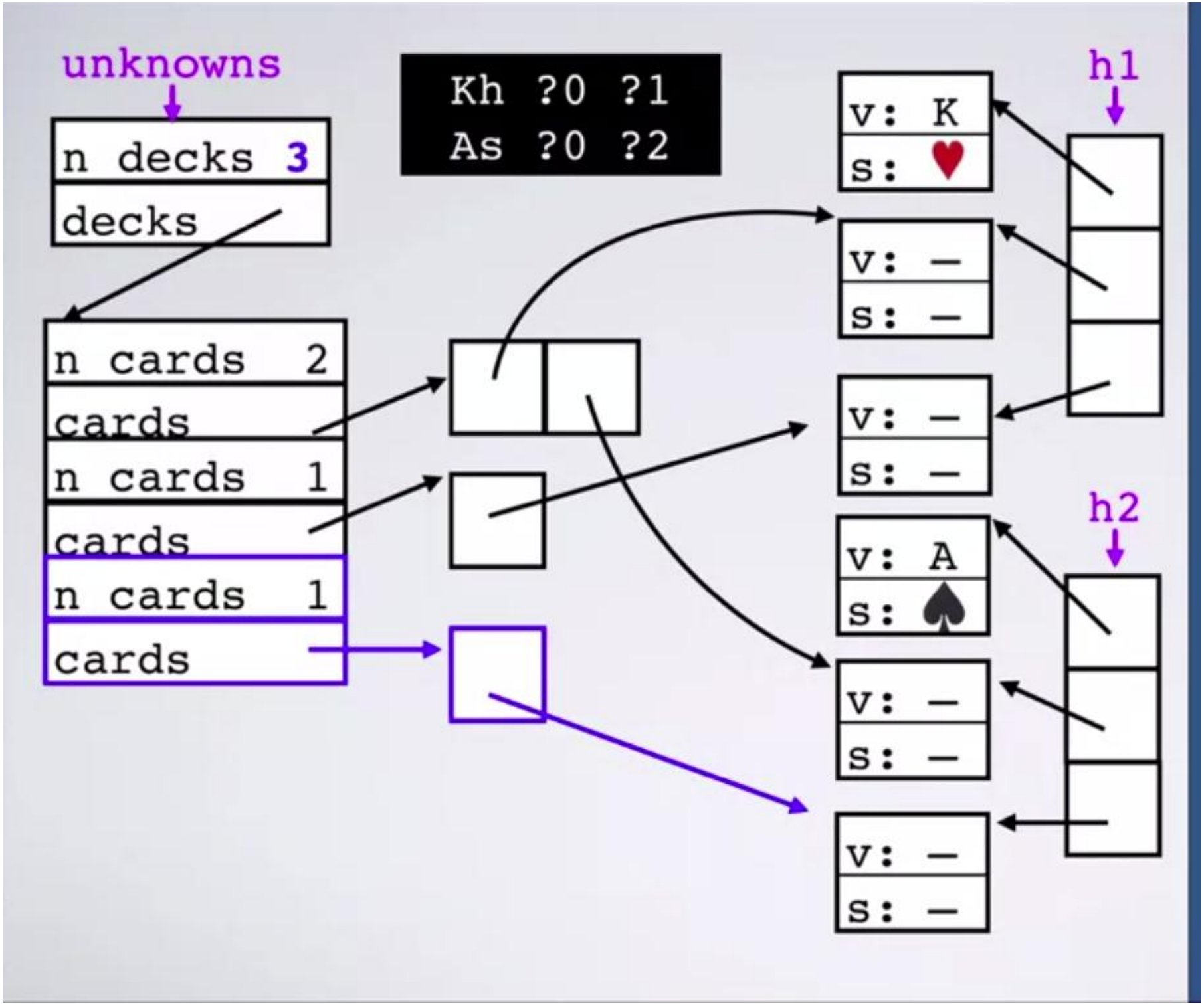
- Time to finish poker project!

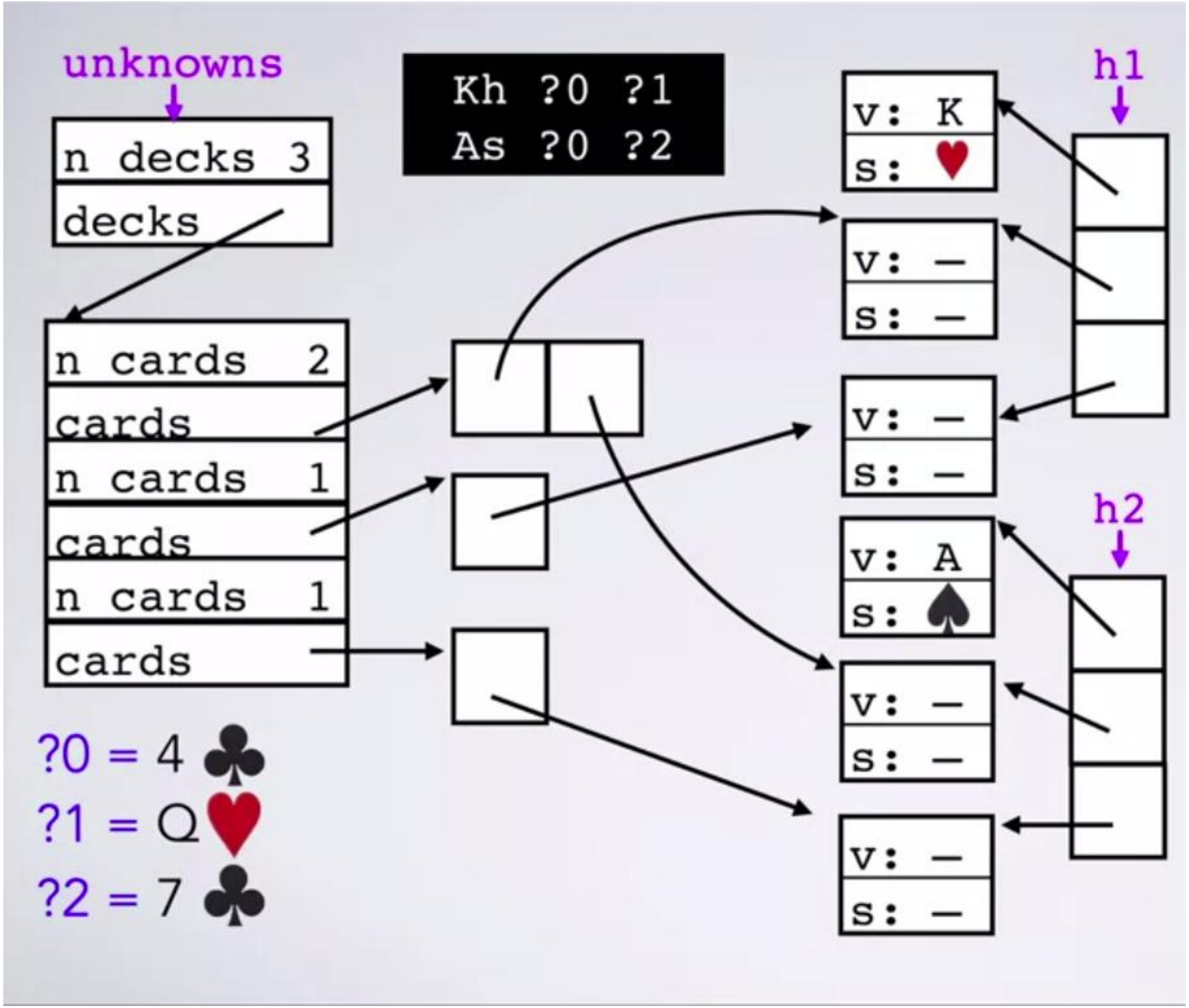
Unknown Cards

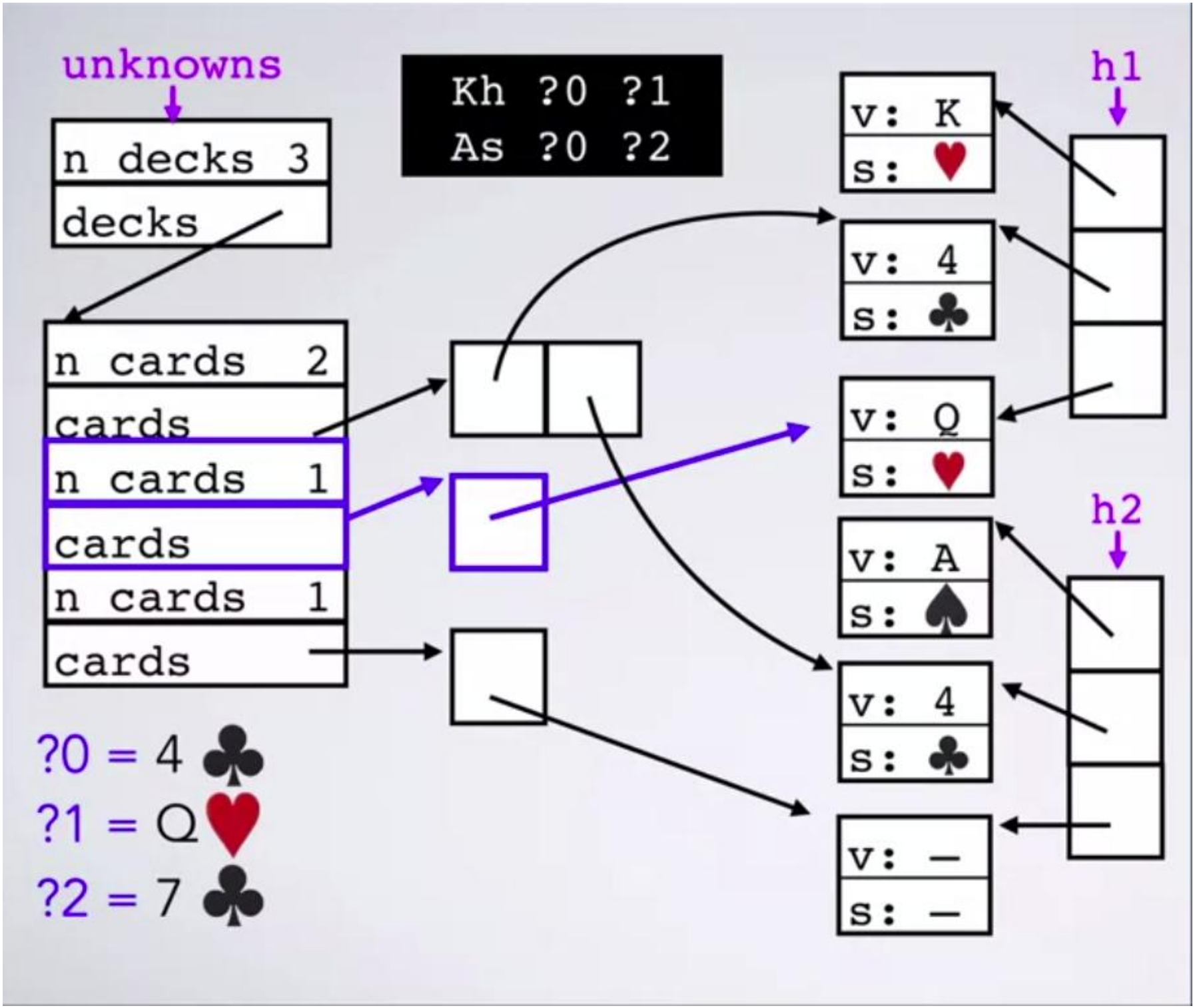
```
Kh ?0 ?1
As ?0 ?2
```

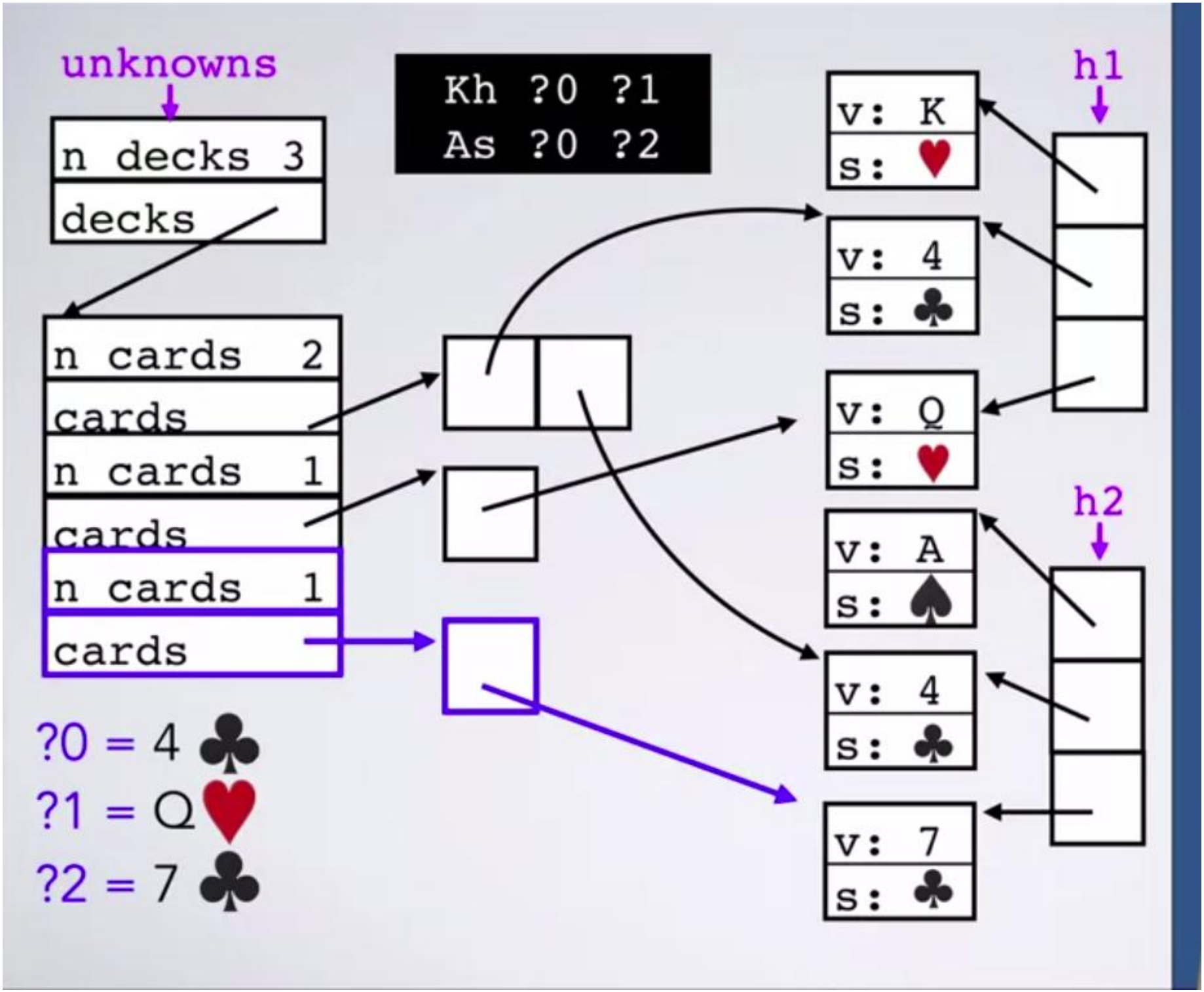
- How to handle unknown cards?


unknowns
n decks 0
decks
Kh ?0 ?1
As ?0 ?2


unknowns
n decks 3
decks
n cards 2
cards
n cards 1
cards
n cards 1
cards
Kh ?0 ?1
As ?0 ?2
h1
v: K
s:
v: –
s: –
v: –
s: –
h2
v: A
s:
v: –
s: –
v: –
s: –


unknowns
n decks 3
decks
n cards 2
cards
n cards 1
cards
n cards 1
cards
Kh ?0 ?1
As ?0 ?2
h1
v: K
s: ♥
v: –
s: –
v: –
s: –
h2
v: A
s: ♠
v: –
s: –
v: –
s: –
?0 = 4 ♣
?1 = Q ♥
?2 = 7 ♣


unknowns
n decks 3
decks
n cards 2
cards
n cards 1
cards
n cards 1
cards
Kh ?0 ?1
As ?0 ?2
?0 = 4
?1 = Q
?2 = 7
h1
v: K
s:
v: 4
s:
v: Q
s:
h2
v: A
s:
v: 4
s:
v: –
s: –


unknowns
n decks 3
decks
n cards 2
cards
n cards 1
cards
n cards 1
cards
Kh ?0 ?1
As ?0 ?2
?0 = 4 ♣
?1 = Q ♥
?2 = 7 ♣
h1
v: K
s: ♥
v: 4
s: ♣
v: Q
s: ♥
h2
v: A
s: ♠
v: 4
s: ♣
v: 7
s: ♣