A3 Review & IPC

Lecture 23
Class 25 of 28 | April 18th 2023 | COMP 211-002 | Joshua Bakita

Welcome!

Today:

- ➔ A3 Review

Logistics:

- ➔ A5 fully posted
- ➔ A4 late due date Thurs
- ➔ A3 grades tonight
- ➔ Final exam exceptions: https://eef.oasis.unc.edu/
- ➔ For regrade rqs., prefer Gradescope or Pizza
- ➔ Research opportunity if you get an A/A-

Fun fact…

You can include any sort of shell command in the commands section for a target in a Makefile.

Want to force people to specify a target rather than using the default? You could add a dummy target like:

```
dummy:
    aafire
```

at the top.

Assignment 3 Review

We plan to release style and functionality grades late tonight.

Assignment 3 Review

Style Feedback: Common Functional Issues

1. Underflow in comparison functions
2. Insufficiently large path buffers
3. Missing error checking on fopen(), fread(), malloc(), strdup(), and realloc(), etc.
4. No support for input from the console, rather than a redirected file
5. Allocating a temporary input-line buffer of size strlen(), leaving insufficient space for the terminating null-character
6. Count lines via # of '\n's, but this will skip last line if there's no trailing newline
    - Or count on the trailing '\n' to exist at location length - 1
7. Missing cleanup, particularly in cases of early termination
8. Uses int rather than unsigned int in internal struct

Assignment 3 Review

Style Feedback: Common Niceness Issues

9. Missing error or help messages to guide the user
10. Errors printed to stdout, rather than stderr
11. Only prints a generic error message, rather than checking errno or using perror()
12. Duplicate comparator functions. (Can eliminate via a primary and secondary metric field in your tracking struct.)
13. Duplicate code or outdated comments

Assignment 3 Review

Style Feedback: Common Efficiency Issues

14. Excessive number of allocations and copies (almost everyone)
15. fgets() into a temporary buffer, then copied to the permanent one
    - Why not read directly into the permanent buffer?
16. Growing arrays via realloc() only one entry at a time
    - realloc() may require copying the whole array every time
17. Read character-by-character via fgetc(), incurring significant syscall overhead
18. Duplicate string traversals (taking strlen()/strcspn() rather than using length from an API that provides it, like getline())

Looking closer at memory efficiency

My Solution

ex_game_list.txt =

jonas_the_unbeatable.bin

alex_the_best.bin

bob_the_novice.bin

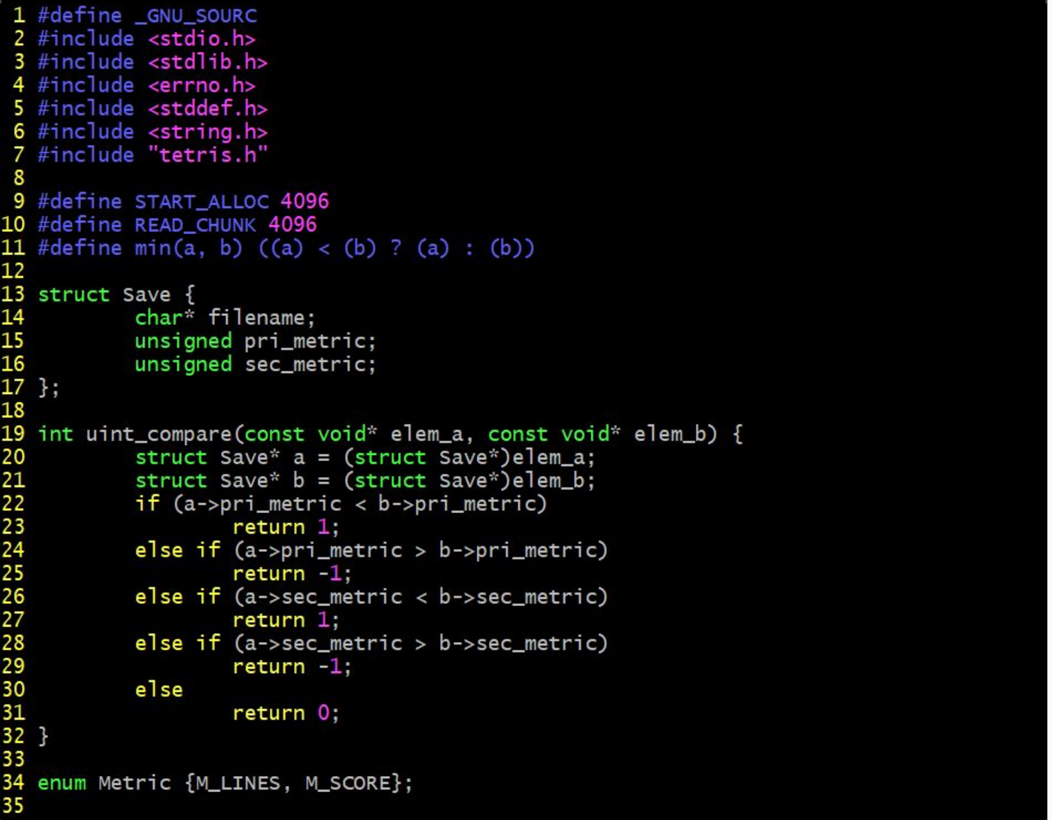
./rank score 2 < ex_game_list.txt

Let the contents of ex_game_list.txt be

```
jonas_the_unbeatable.bin
alex_the_best.bin
bob_the_novice.bin
```

```
./rank score 2 < ex_game_list.txt
```

ON LINE 1

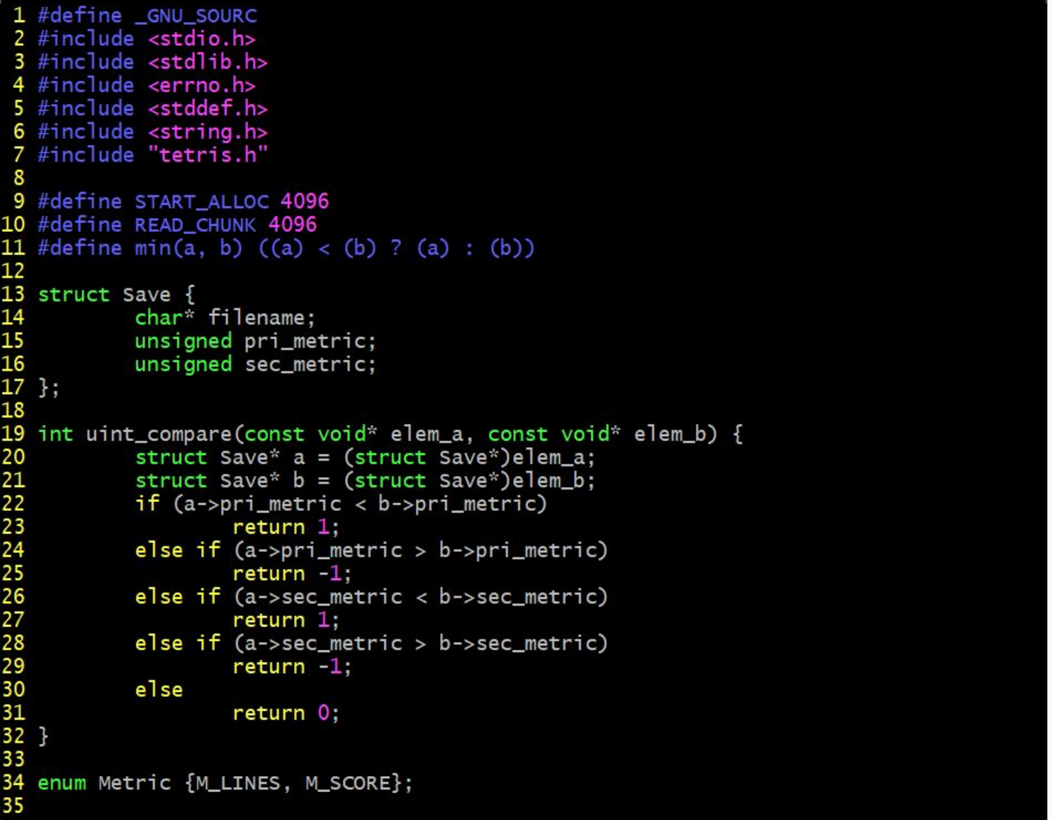


```c
 1 #define _GNU_SOURC
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <errno.h>
 5 #include <stddef.h>
 6 #include <string.h>
 7 #include "tetris.h"
 8
 9 #define START_ALLOC 4096
10 #define READ_CHUNK 4096
11 #define min(a, b) ((a) < (b) ? (a) : (b))
12
13 struct Save {
14         char* filename;
15         unsigned pri_metric;
16         unsigned sec_metric;
17 };
18
19 int uint_compare(const void* elem_a, const void* elem_b) {
20         struct Save* a = (struct Save*)elem_a;
21         struct Save* b = (struct Save*)elem_b;
22         if (a->pri_metric < b->pri_metric)
23                 return 1;
24         else if (a->pri_metric > b->pri_metric)
25                 return -1;
26         else if (a->sec_metric < b->sec_metric)
27                 return 1;
28         else if (a->sec_metric > b->sec_metric)
29                 return -1;
30         else
31                 return 0;
32 }
33
34 enum Metric {M_LINES, M_SCORE};
35
```

Stack

Heap

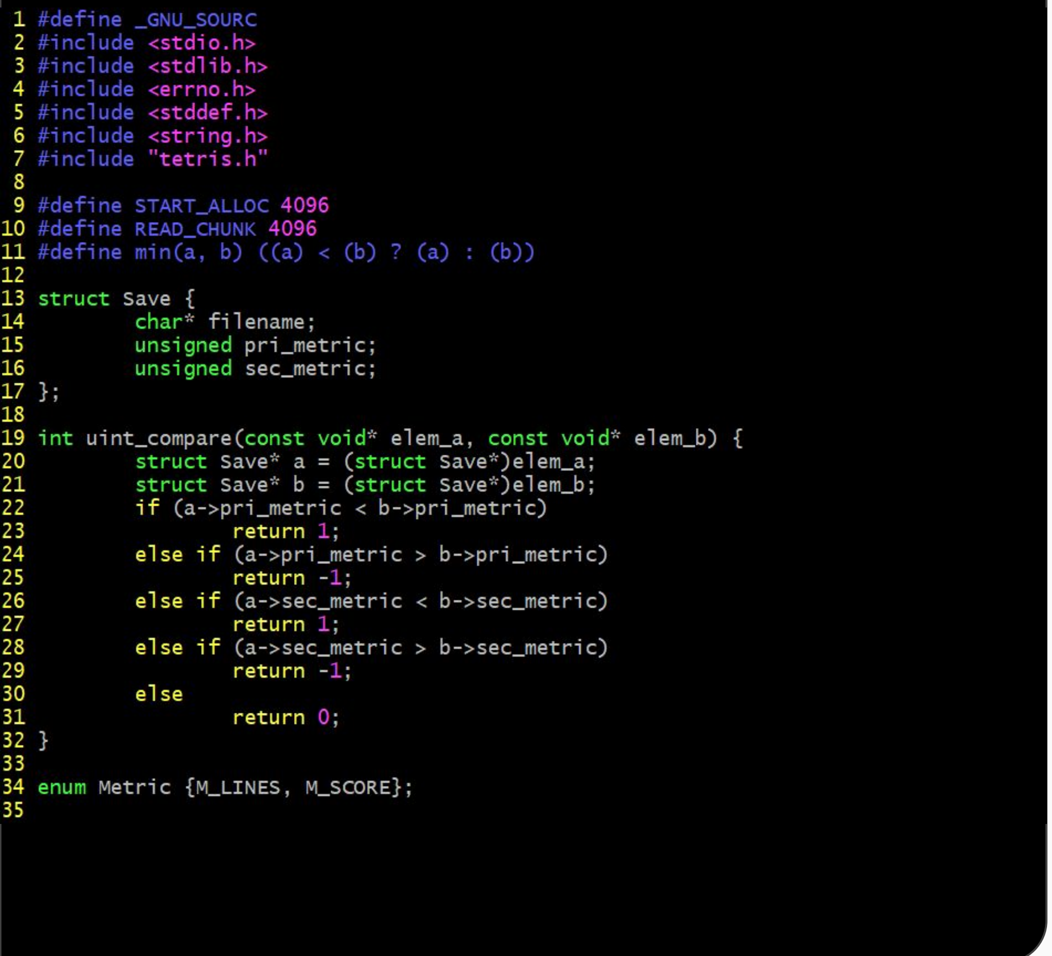
ON LINE 35



```
1 #define _GNU_SOURC
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <stddef.h>
6 #include <string.h>
7 #include "tetris.h"
8
9 #define START_ALLOC 4096
10 #define READ_CHUNK 4096
11 #define min(a, b) ((a) < (b) ? (a) : (b))
12
13 struct Save {
14         char* filename;
15         unsigned pri_metric;
16         unsigned sec_metric;
17 };
18
19 int uint_compare(const void* elem_a, const void* elem_b) {
20         struct Save* a = (struct Save*)elem_a;
21         struct Save* b = (struct Save*)elem_b;
22         if (a->pri_metric < b->pri_metric)
23                 return 1;
24         else if (a->pri_metric > b->pri_metric)
25                 return -1;
26         else if (a->sec_metric < b->sec_metric)
27                 return 1;
28         else if (a->sec_metric > b->sec_metric)
29                 return -1;
30         else
31                 return 0;
32 }
33
34 enum Metric {M_LINES, M_SCORE};
35
```

Stack	Heap

ON LINE 35



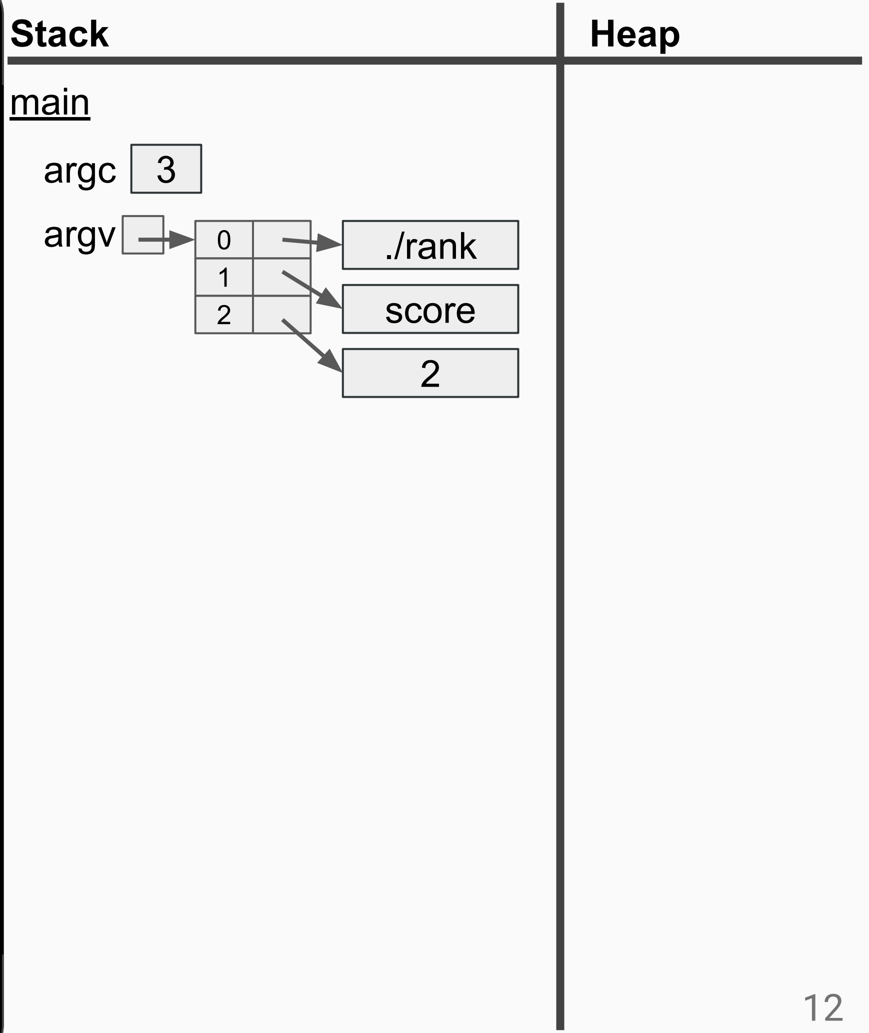
```
 1 #define _GNU_SOURC
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <errno.h>
 5 #include <stddef.h>
 6 #include <string.h>
 7 #include "tetris.h"
 8
 9 #define START_ALLOC 4096
10 #define READ_CHUNK 4096
11 #define min(a, b) ((a) < (b) ? (a) : (b))
12
13 struct Save {
14         char* filename;
15         unsigned pri_metric;
16         unsigned sec_metric;
17 };
18
19 int uint_compare(const void* elem_a, const void* elem_b) {
20         struct Save* a = (struct Save*)elem_a;
21         struct Save* b = (struct Save*)elem_b;
22         if (a->pri_metric < b->pri_metric)
23                 return 1;
24         else if (a->pri_metric > b->pri_metric)
25                 return -1;
26         else if (a->sec_metric < b->sec_metric)
27                 return 1;
28         else if (a->sec_metric > b->sec_metric)
29                 return -1;
30         else
31                 return 0;
32 }
33
34 enum Metric {M_LINES, M_SCORE};
35
```

Stack	Heap

Remember that function definitions and static variables defined outside of main are stored in static memory which is why stack and heap are still empty before main.

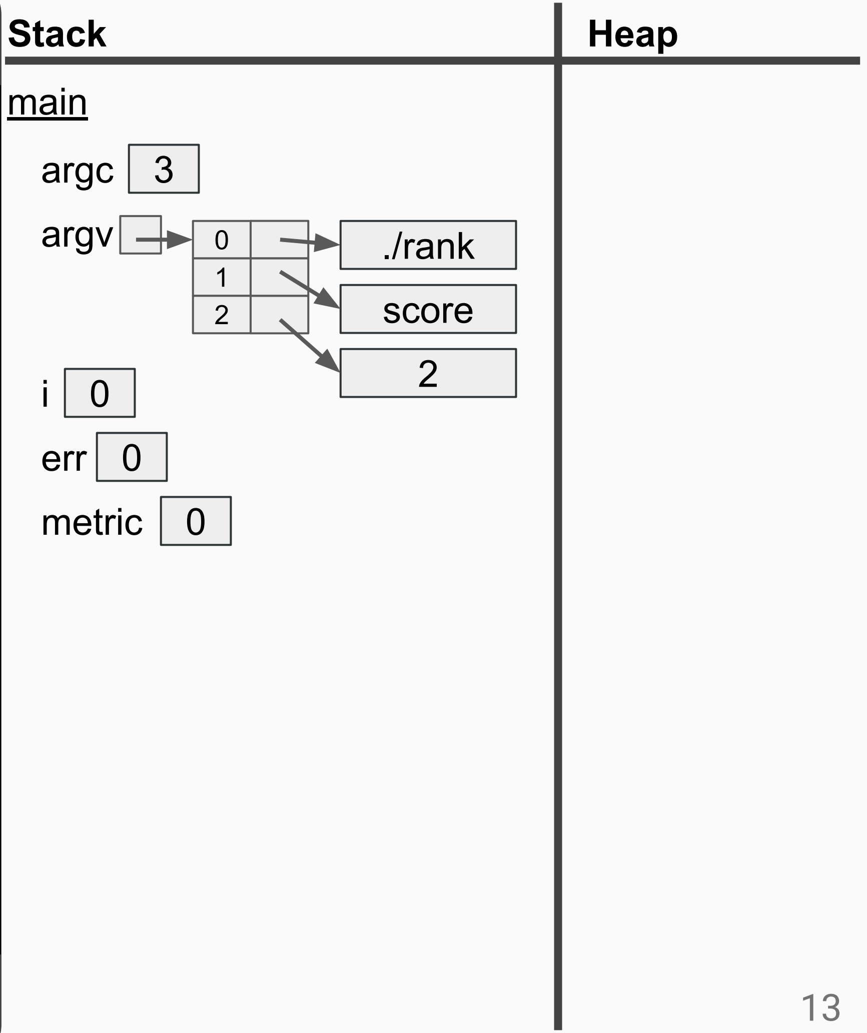
ON LINE 36

```c
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



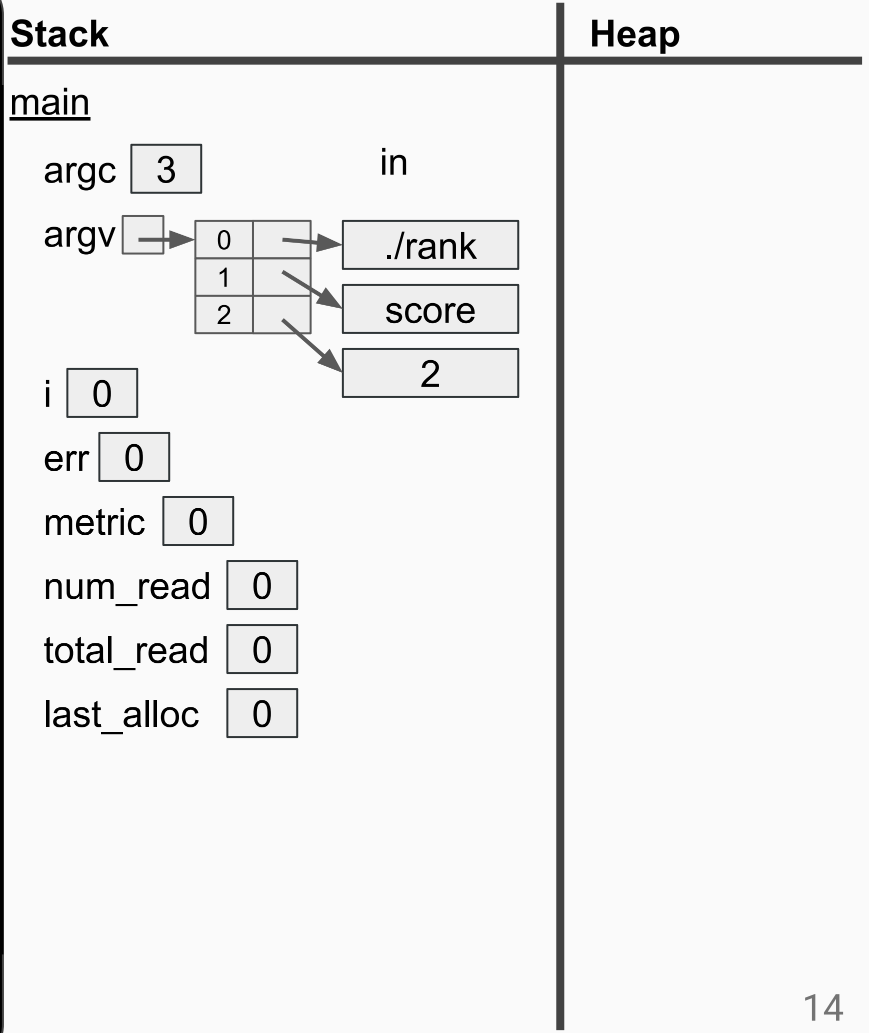
ON LINE 52

```
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



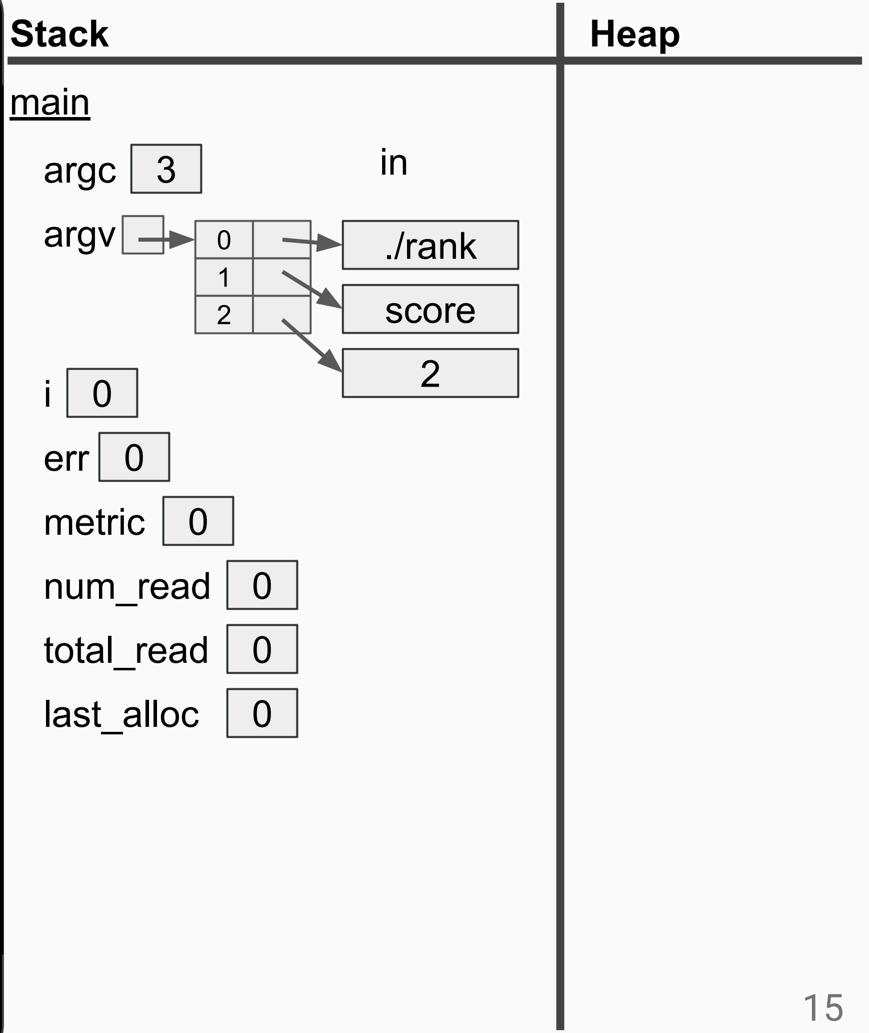
ON LINE 55

```c
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



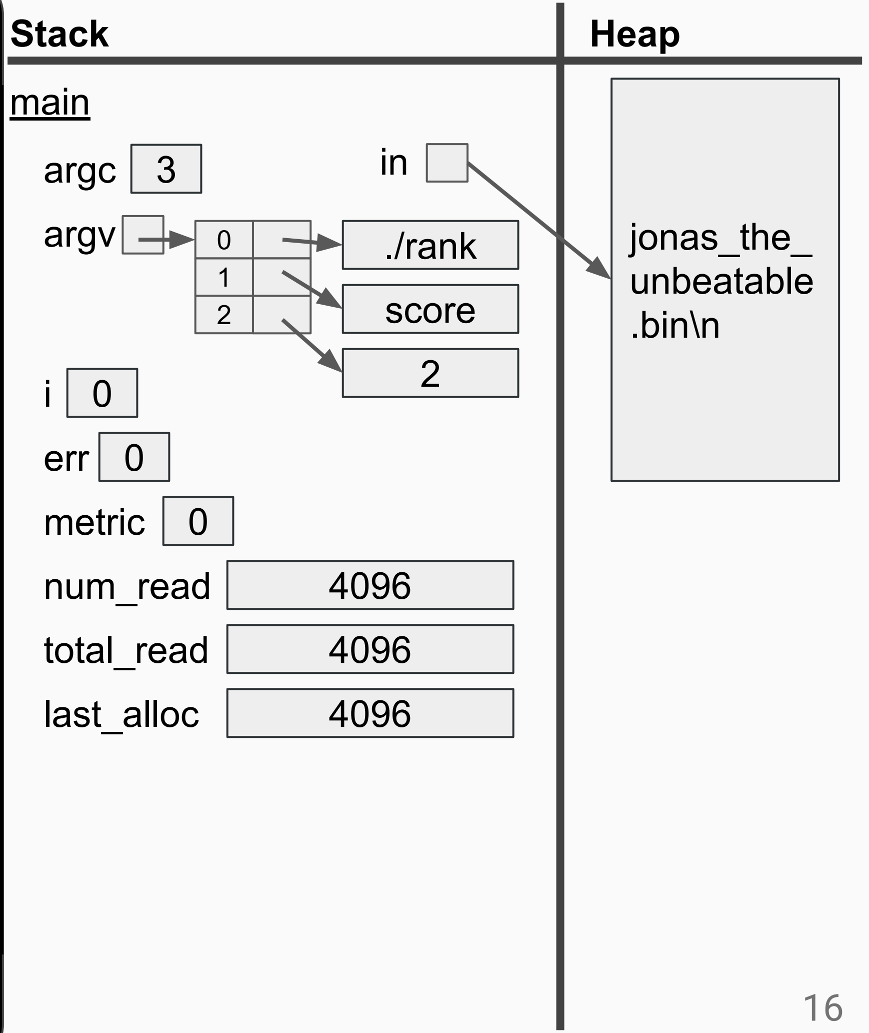
ON LINE 55

```c
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



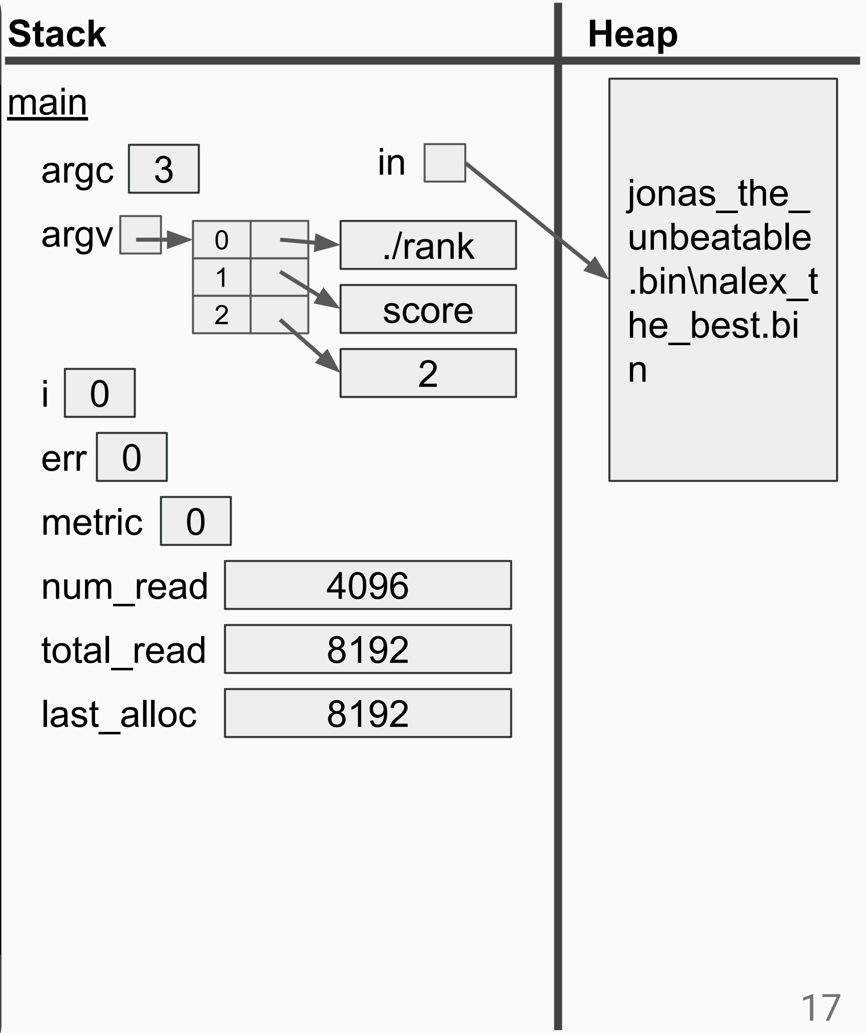
ON LINE 68

```c
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



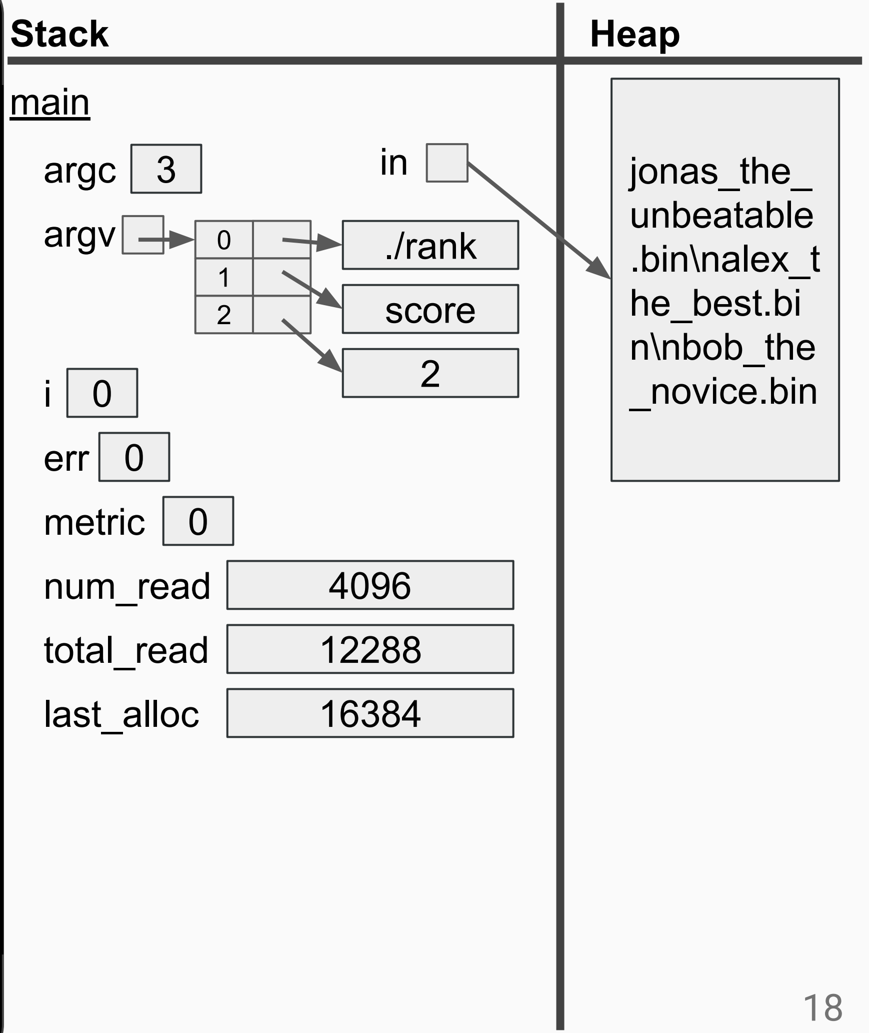
ON LINE 68

```
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



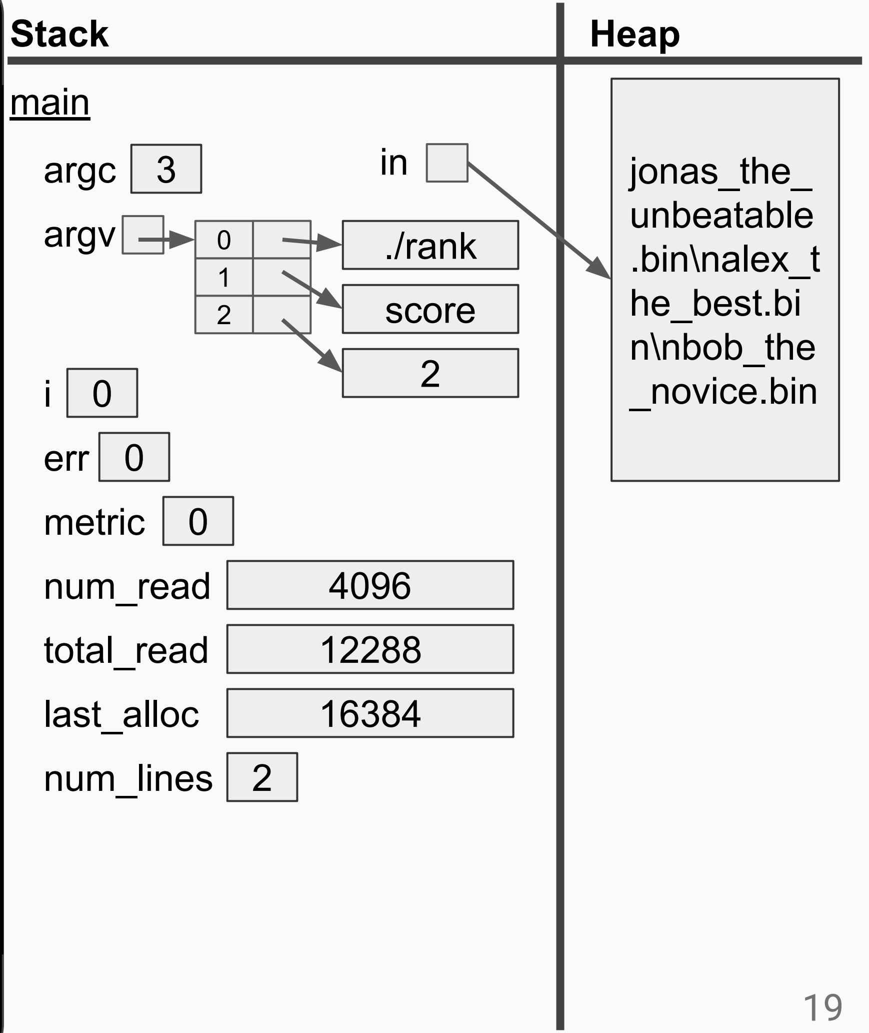
ON LINE 68

```c
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```

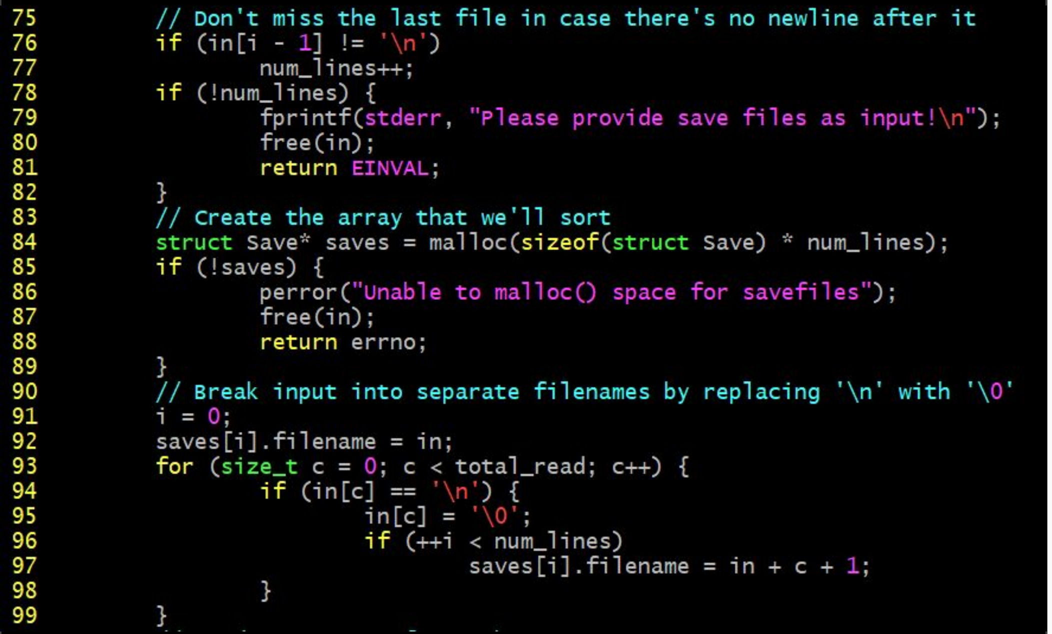


ON LINE 75

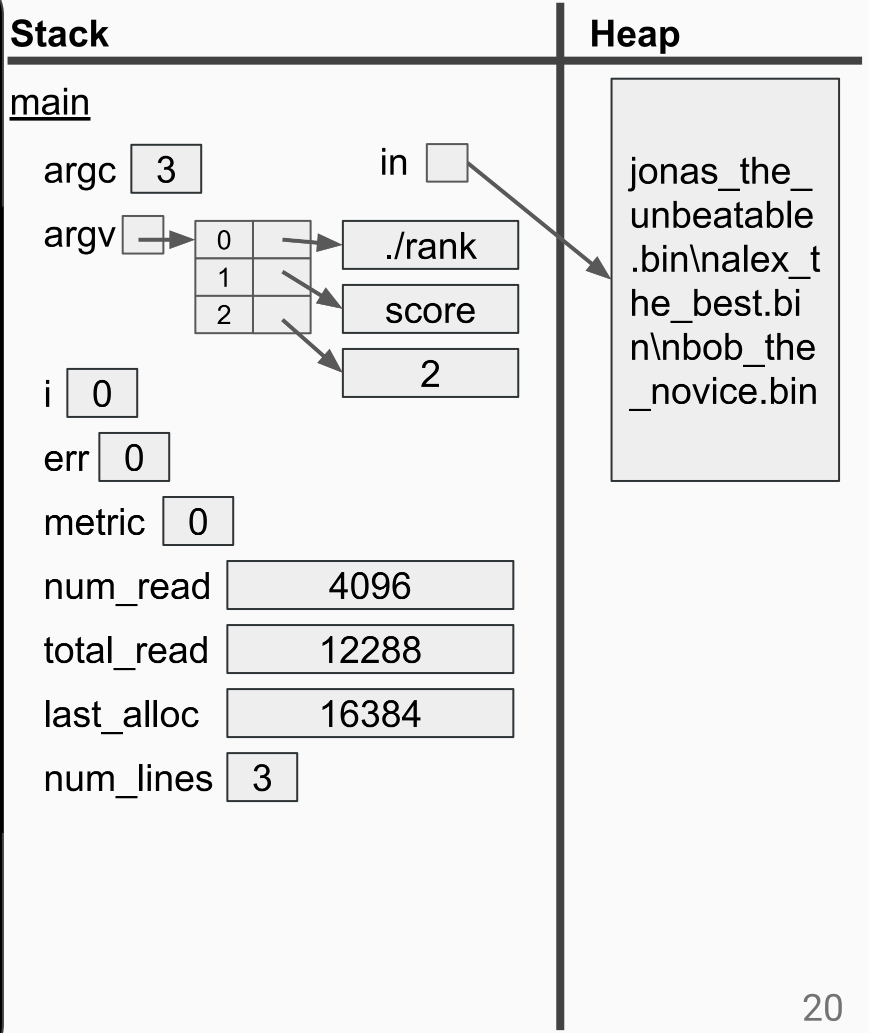
```
36 int main(int argc, char** argv) {
37         size_t i = 0;
38         int err = 0;
39         // Validate arguments
40         if (argc != 3) {
41                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
42                 return EINVAL;
43         }
44         enum Metric metric;
45         if (strcmp(argv[1], "lines") == 0)
46                 metric = M_LINES;
47         else if (strcmp(argv[1], "score") == 0)
48                 metric = M_SCORE;
49         else {
50                 fprintf(stderr, "Usage: %s [score|lines] [num top]\n", argv[0]);
51                 return EINVAL;
52         }
53         // Read all input lines
54         size_t num_read = 0, total_read = 0, last_alloc = 0;
55         char* in = NULL;
56         do {
57                 // Double available space if insufficient space for next read
58                 if (total_read + READ_CHUNK >= last_alloc) {
59                         last_alloc = last_alloc ? last_alloc * 2 : READ_CHUNK;
60                         in = realloc(in, last_alloc + 1);
61                         if (!in) {
62                                 perror("Unable to realloc() space for input");
63                                 free(in);
64                                 return errno;
65                         }
66                 }
67         } while ((num_read = fread(in + total_read, 1, READ_CHUNK, stdin))
68                  && (total_read += num_read) && !feof(stdin));
69         // Null-terminate input (this is safe due to +1 in realloc())
70         in[total_read] = '\0';
71         // Count number of lines
72         unsigned int num_lines = 0;
73         for (i = 0; i < total_read; i++)
74                 num_lines += (in[i] == '\n');
```



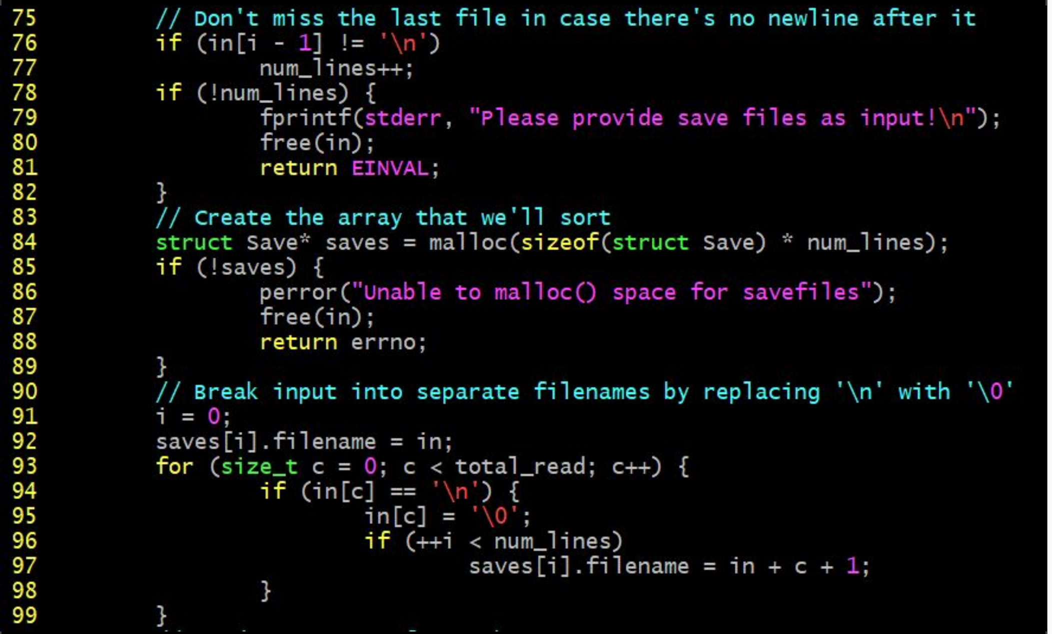
ON LINE 83



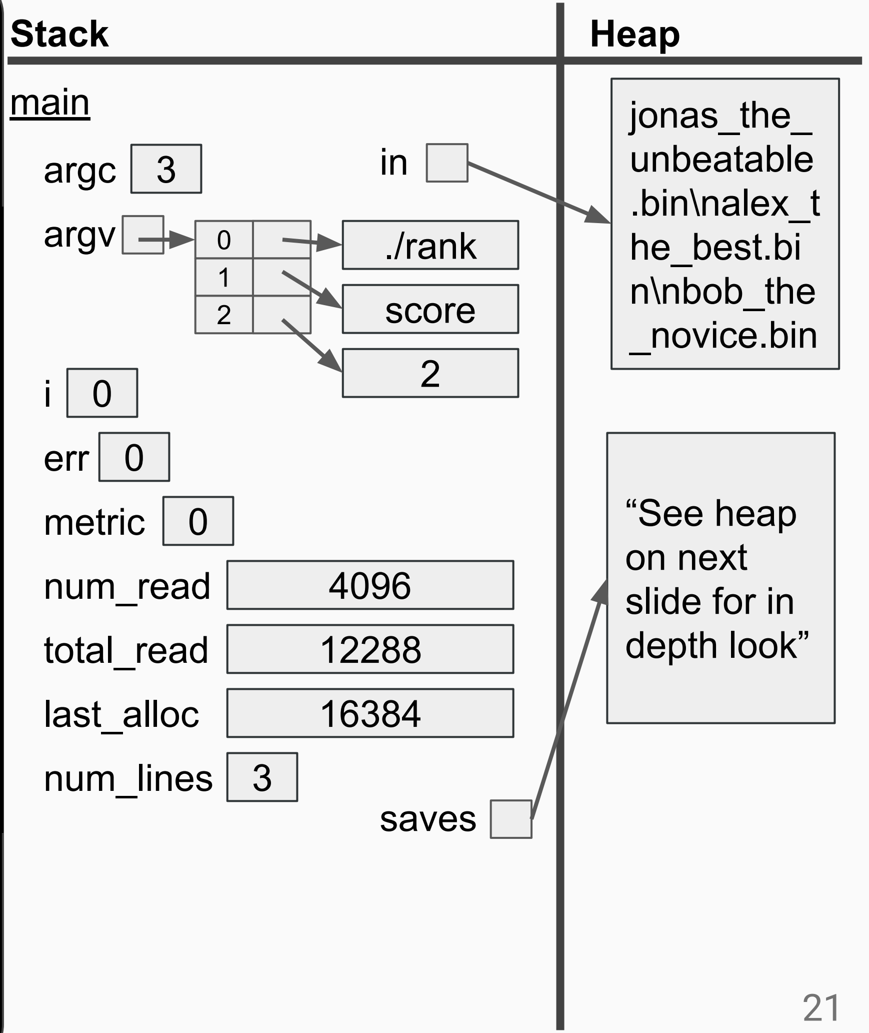
```
75      // Don't miss the last file in case there's no newline after it
76      if (in[i - 1] != '\n')
77          num_lines++;
78      if (!num_lines) {
79          fprintf(stderr, "Please provide save files as input!\n");
80          free(in);
81          return EINVAL;
82      }
83      // Create the array that we'll sort
84      struct Save* saves = malloc(sizeof(struct Save) * num_lines);
85      if (!saves) {
86          perror("Unable to malloc() space for savefiles");
87          free(in);
88          return errno;
89      }
90      // Break input into separate filenames by replacing '\n' with '\0'
91      i = 0;
92      saves[i].filename = in;
93      for (size_t c = 0; c < total_read; c++) {
94          if (in[c] == '\n') {
95              in[c] = '\0';
96              if (++i < num_lines)
97                  saves[i].filename = in + c + 1;
98          }
99      }
```



ON LINE 91

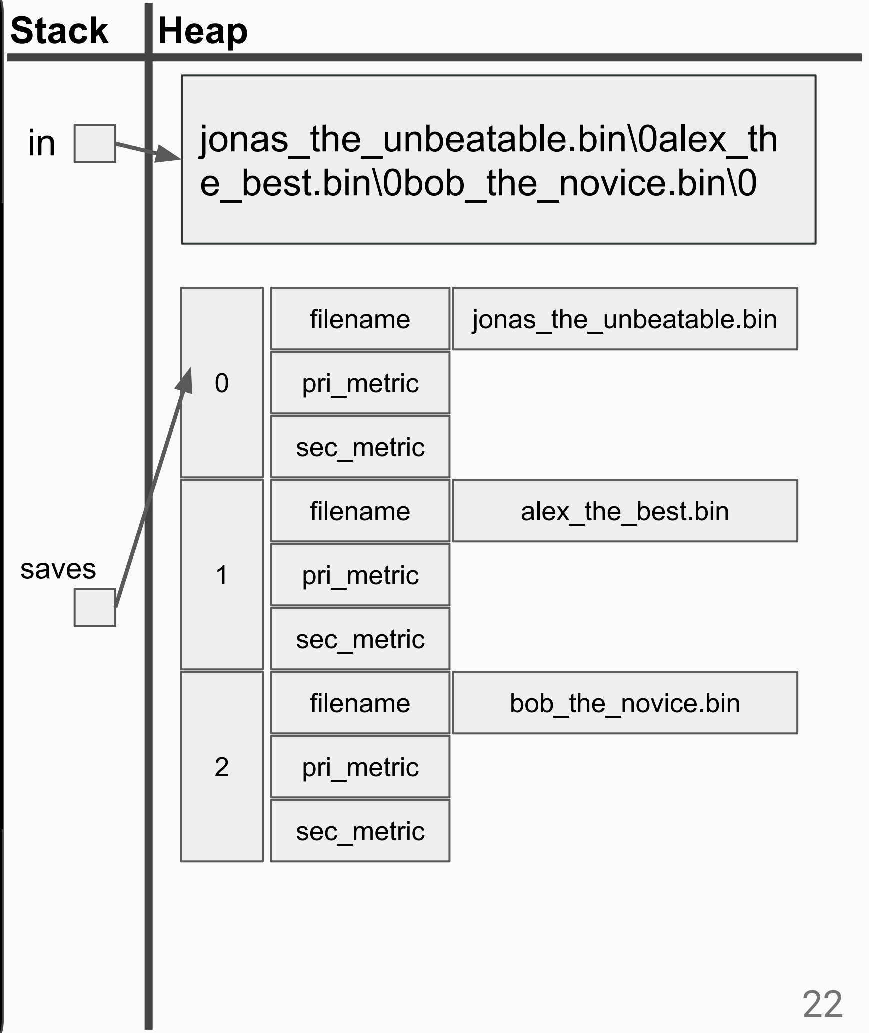


```
75          // Don't miss the last file in case there's no newline after it
76          if (in[i - 1] != '\n')
77                  num_lines++;
78          if (!num_lines) {
79                  fprintf(stderr, "Please provide save files as input!\n");
80                  free(in);
81                  return EINVAL;
82          }
83          // Create the array that we'll sort
84          struct Save* saves = malloc(sizeof(struct Save) * num_lines);
85          if (!saves) {
86                  perror("Unable to malloc() space for savefiles");
87                  free(in);
88                  return errno;
89          }
90          // Break input into separate filenames by replacing '\n' with '\0'
91          i = 0;
92          saves[i].filename = in;
93          for (size_t c = 0; c < total_read; c++) {
94                  if (in[c] == '\n') {
95                          in[c] = '\0';
96                          if (++i < num_lines)
97                                  saves[i].filename = in + c + 1;
98                  }
99          }
```

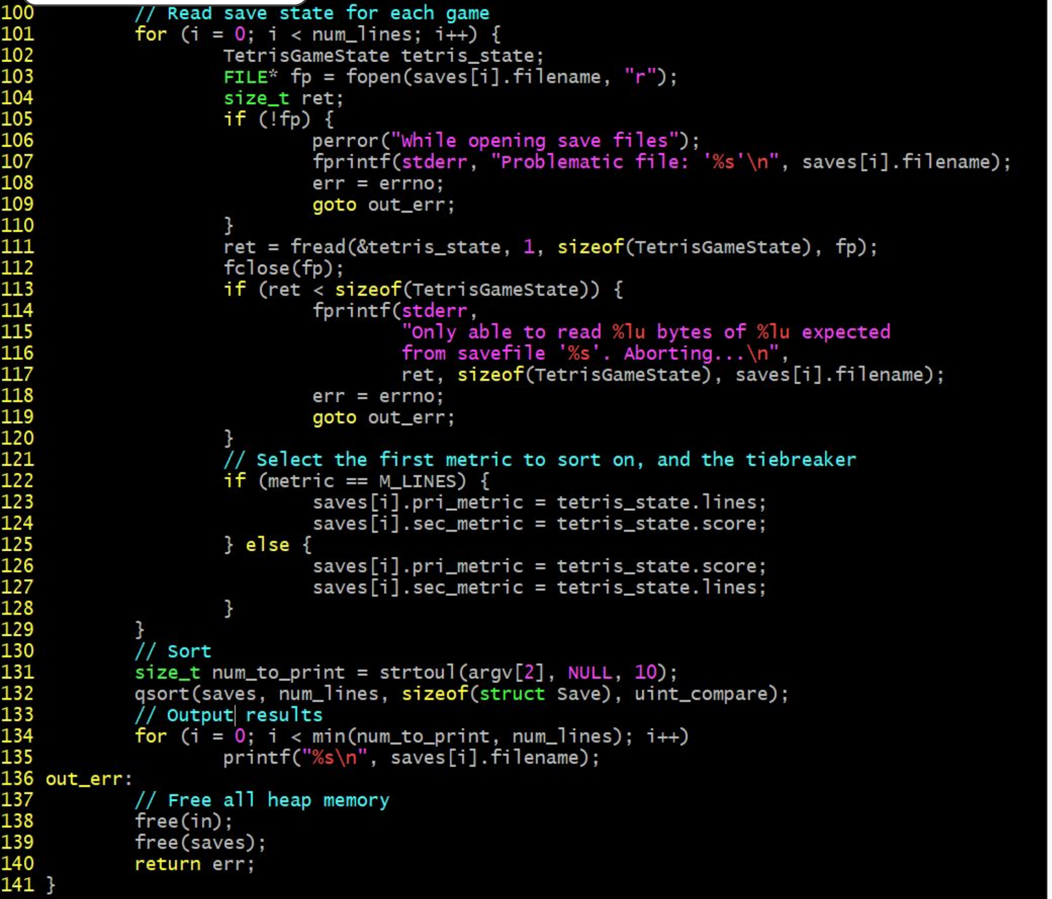


ON LINE 99

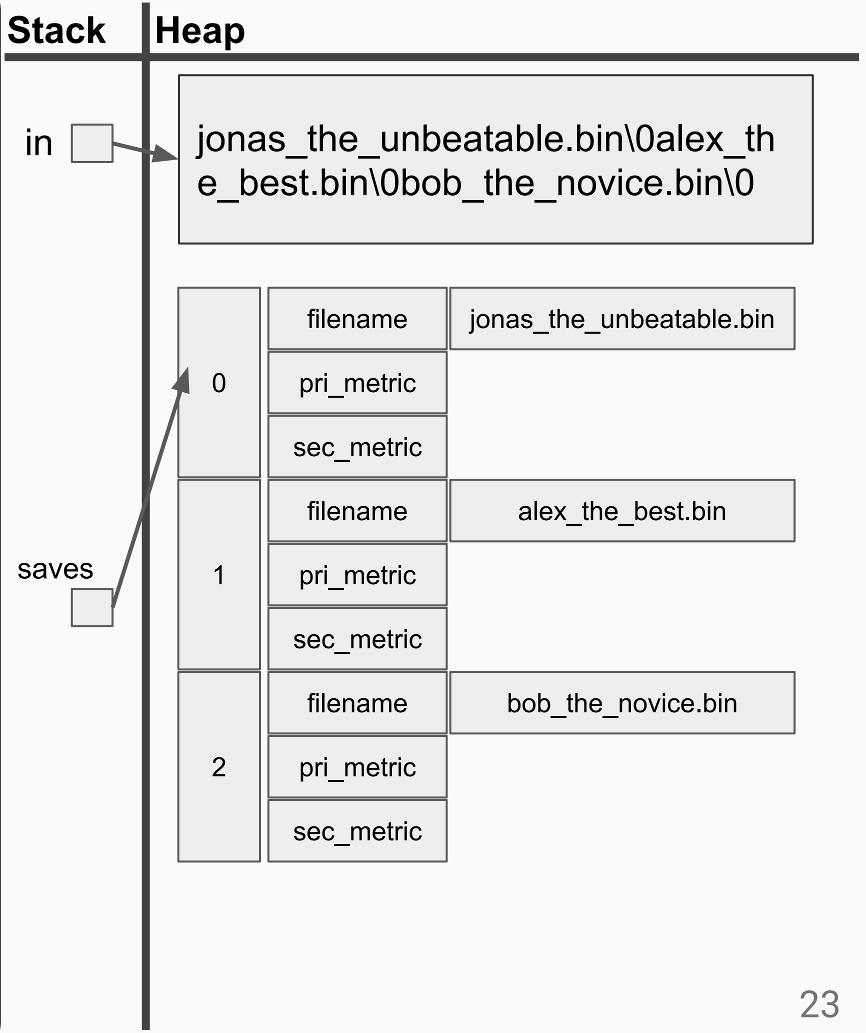
```
75          // Don't miss the last file in case there's no newline after it
76          if (in[i - 1] != '\n')
77                  num_lines++;
78          if (!num_lines) {
79                  fprintf(stderr, "Please provide save files as input!\n");
80                  free(in);
81                  return EINVAL;
82          }
83          // Create the array that we'll sort
84          struct Save* saves = malloc(sizeof(struct Save) * num_lines);
85          if (!saves) {
86                  perror("Unable to malloc() space for savefiles");
87                  free(in);
88                  return errno;
89          }
90          // Break input into separate filenames by replacing '\n' with '\0'
91          i = 0;
92          saves[i].filename = in;
93          for (size_t c = 0; c < total_read; c++) {
94                  if (in[c] == '\n') {
95                          in[c] = '\0';
96                          if (++i < num_lines)
97                                  saves[i].filename = in + c + 1;
98                  }
99          }
```



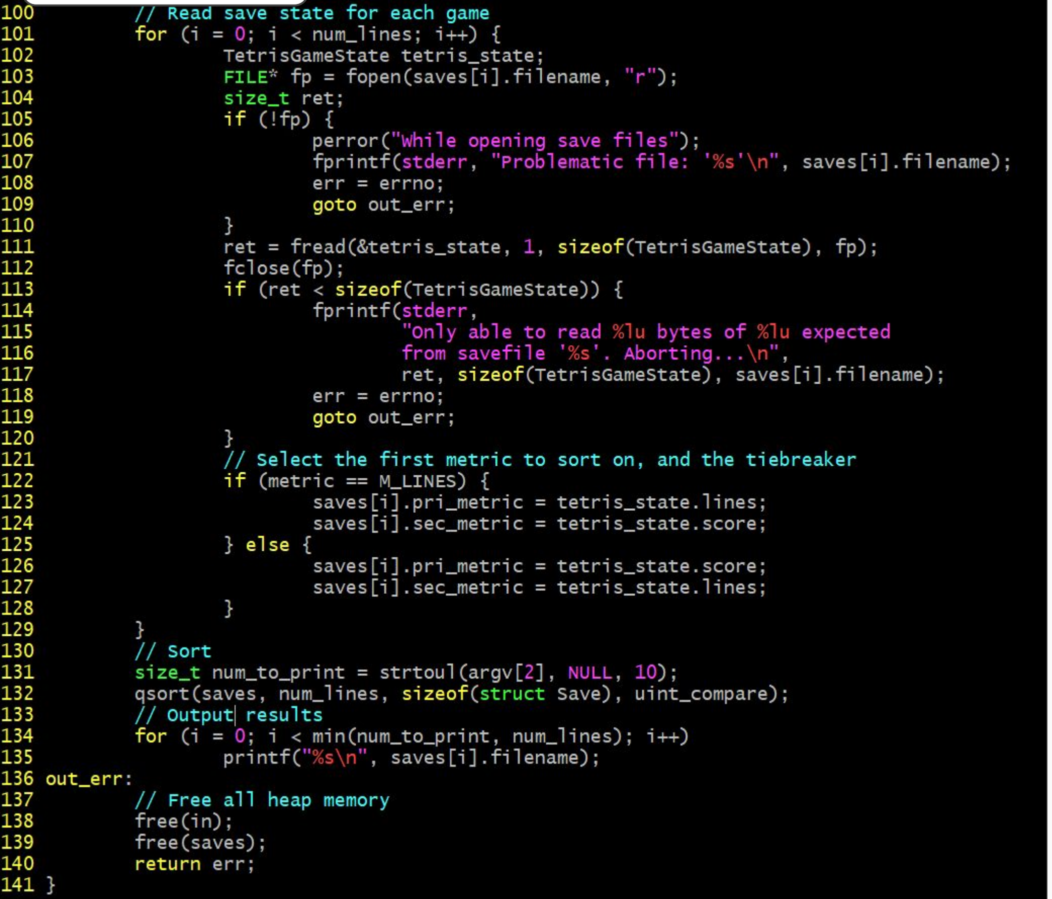
ON LINE 100



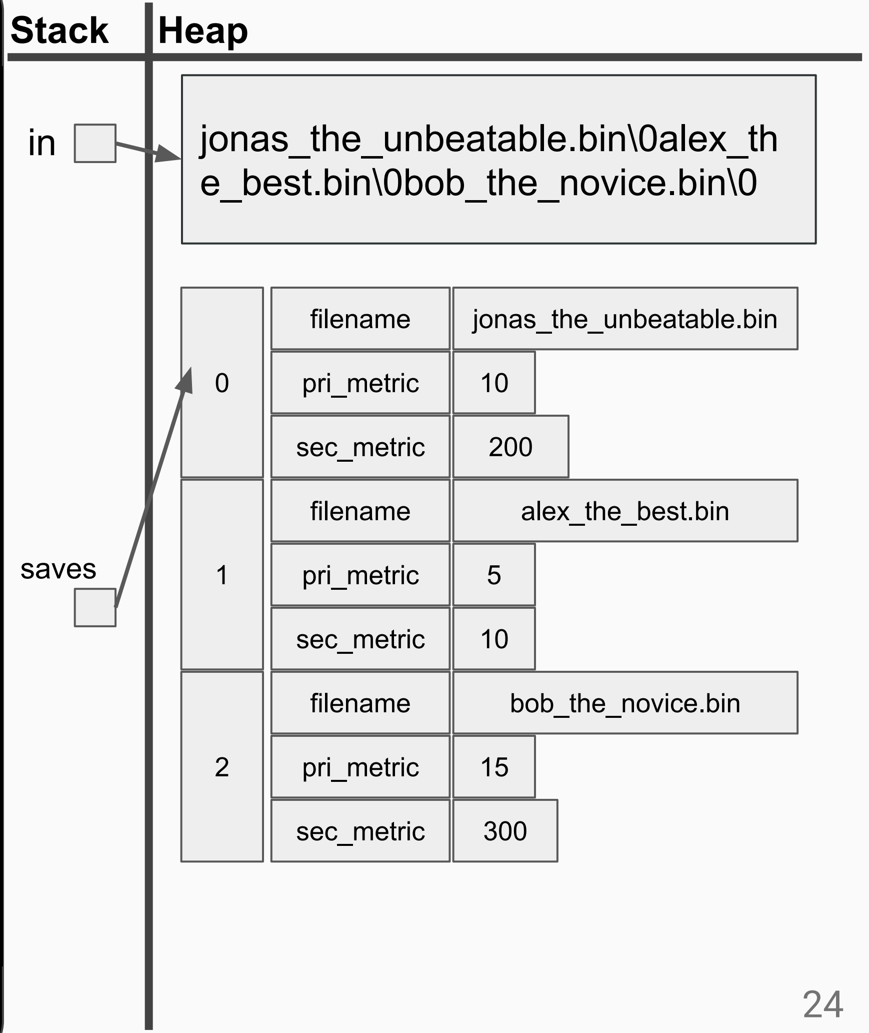
```c
100         // Read save state for each game
101         for (i = 0; i < num_lines; i++) {
102                 TetrisGameState tetris_state;
103                 FILE* fp = fopen(saves[i].filename, "r");
104                 size_t ret;
105                 if (!fp) {
106                         perror("While opening save files");
107                         fprintf(stderr, "Problematic file: '%s'\n", saves[i].filename);
108                         err = errno;
109                         goto out_err;
110                 }
111                 ret = fread(&tetris_state, 1, sizeof(TetrisGameState), fp);
112                 fclose(fp);
113                 if (ret < sizeof(TetrisGameState)) {
114                         fprintf(stderr,
115                                 "Only able to read %lu bytes of %lu expected
116                                 from savefile '%s'. Aborting...\n",
117                                 ret, sizeof(TetrisGameState), saves[i].filename);
118                         err = errno;
119                         goto out_err;
120                 }
121                 // Select the first metric to sort on, and the tiebreaker
122                 if (metric == M_LINES) {
123                         saves[i].pri_metric = tetris_state.lines;
124                         saves[i].sec_metric = tetris_state.score;
125                 } else {
126                         saves[i].pri_metric = tetris_state.score;
127                         saves[i].sec_metric = tetris_state.lines;
128                 }
129         }
130         // Sort
131         size_t num_to_print = strtoul(argv[2], NULL, 10);
132         qsort(saves, num_lines, sizeof(struct Save), uint_compare);
133         // Output results
134         for (i = 0; i < min(num_to_print, num_lines); i++)
135                 printf("%s\n", saves[i].filename);
136 out_err:
137         // Free all heap memory
138         free(in);
139         free(saves);
140         return err;
141 }
```



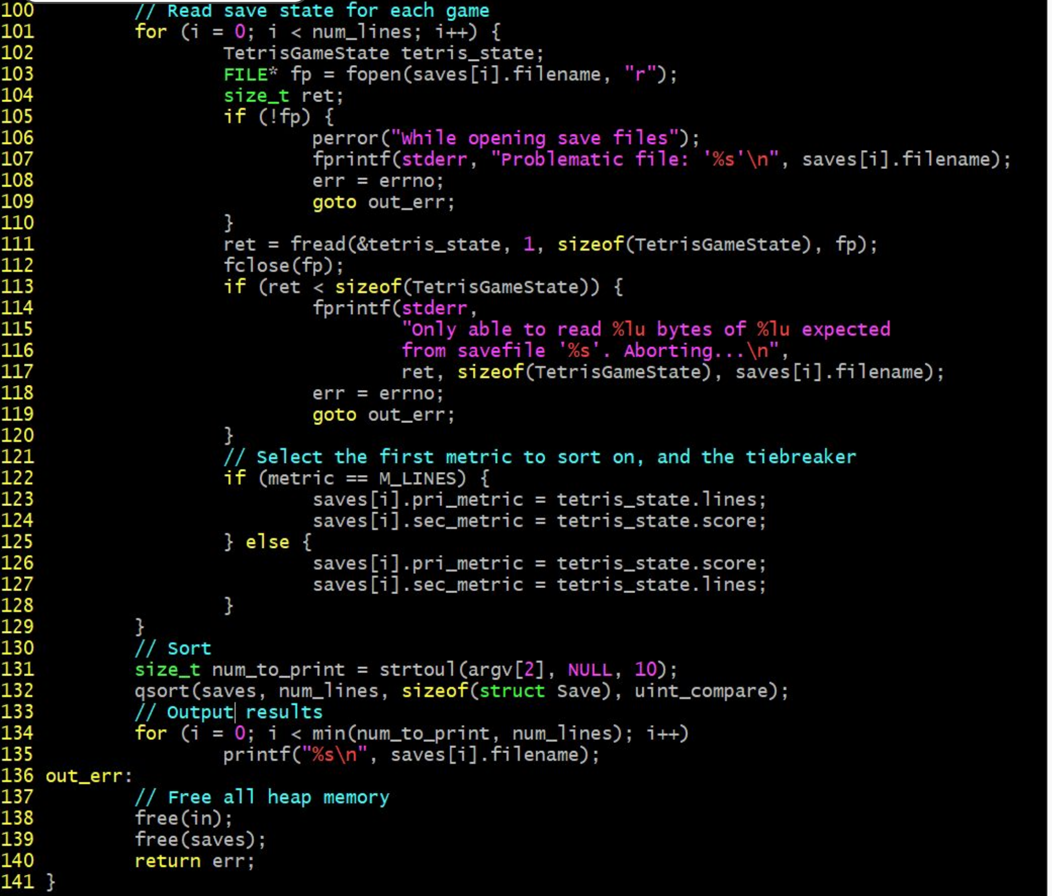
ON LINE 129



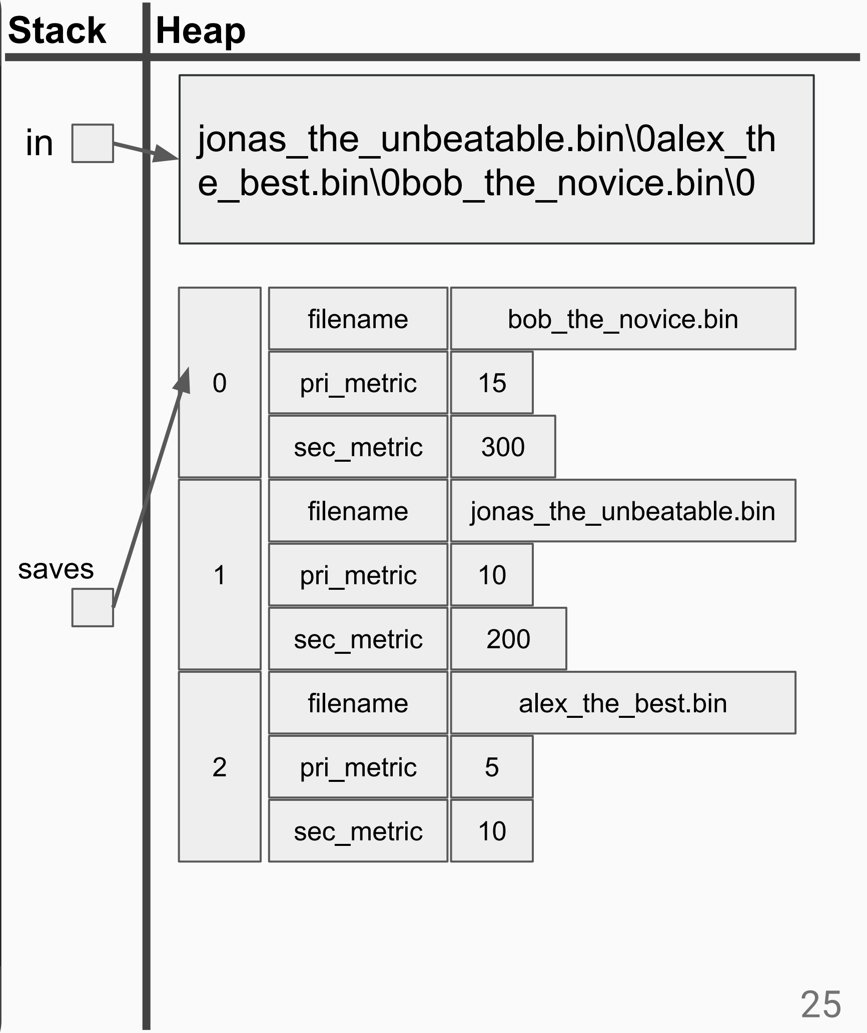
```
100         // Read save state for each game
101         for (i = 0; i < num_lines; i++) {
102                 TetrisGameState tetris_state;
103                 FILE* fp = fopen(saves[i].filename, "r");
104                 size_t ret;
105                 if (!fp) {
106                         perror("While opening save files");
107                         fprintf(stderr, "Problematic file: '%s'\n", saves[i].filename);
108                         err = errno;
109                         goto out_err;
110                 }
111                 ret = fread(&tetris_state, 1, sizeof(TetrisGameState), fp);
112                 fclose(fp);
113                 if (ret < sizeof(TetrisGameState)) {
114                         fprintf(stderr,
115                                 "Only able to read %lu bytes of %lu expected
116                                 from savefile '%s'. Aborting...\n",
117                                 ret, sizeof(TetrisGameState), saves[i].filename);
118                         err = errno;
119                         goto out_err;
120                 }
121                 // Select the first metric to sort on, and the tiebreaker
122                 if (metric == M_LINES) {
123                         saves[i].pri_metric = tetris_state.lines;
124                         saves[i].sec_metric = tetris_state.score;
125                 } else {
126                         saves[i].pri_metric = tetris_state.score;
127                         saves[i].sec_metric = tetris_state.lines;
128                 }
129         }
130         // Sort
131         size_t num_to_print = strtoul(argv[2], NULL, 10);
132         qsort(saves, num_lines, sizeof(struct Save), uint_compare);
133         // Output results
134         for (i = 0; i < min(num_to_print, num_lines); i++)
135                 printf("%s\n", saves[i].filename);
136 out_err:
137         // Free all heap memory
138         free(in);
139         free(saves);
140         return err;
141 }
```



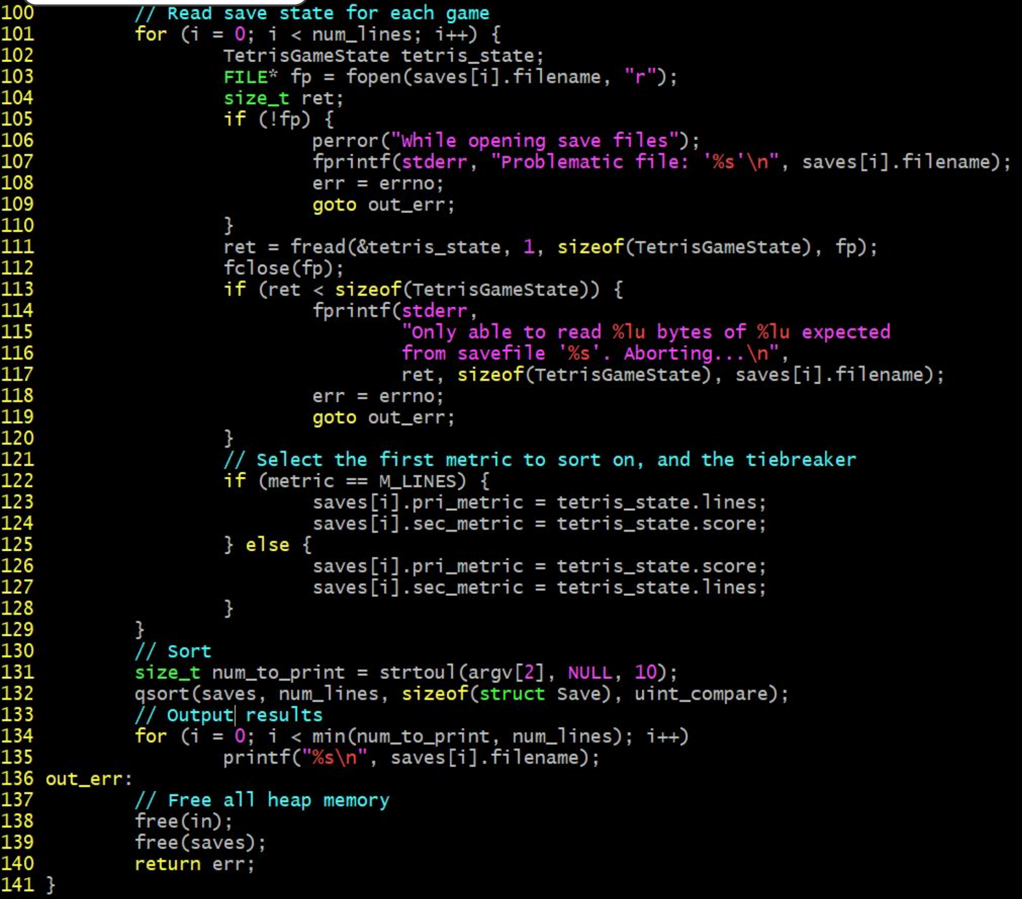
ON LINE 133



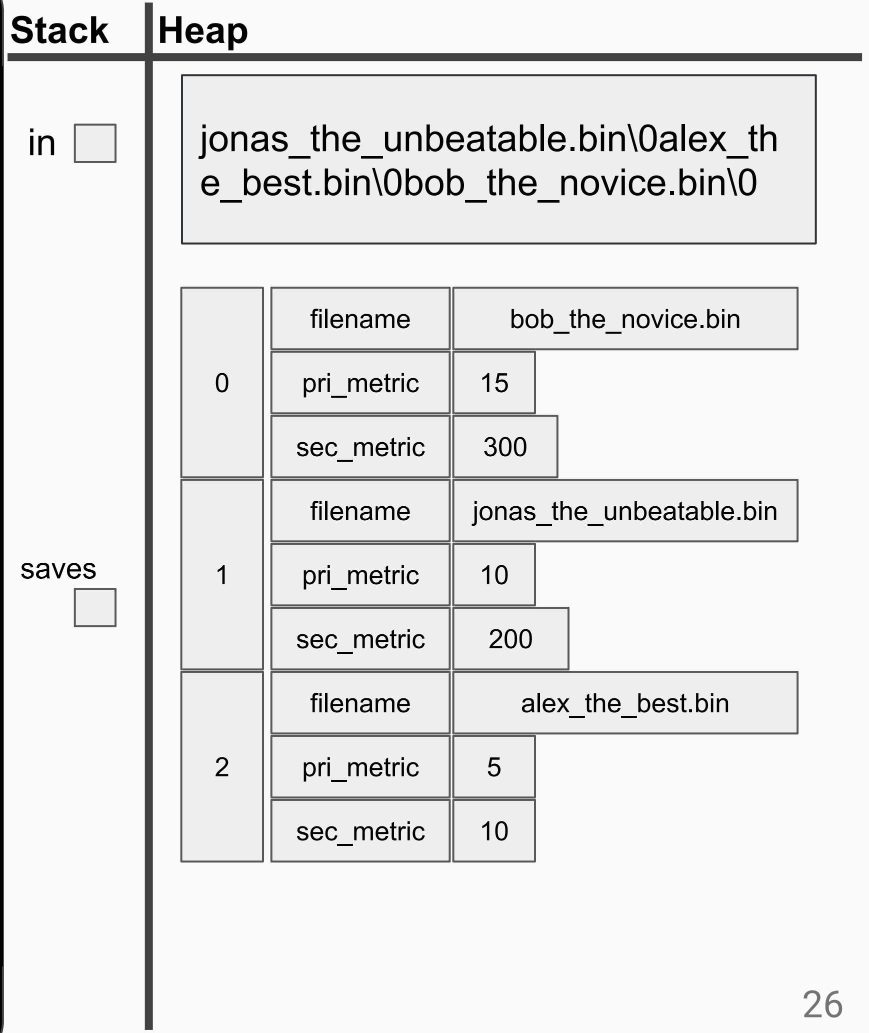
```
100         // Read save state for each game
101         for (i = 0; i < num_lines; i++) {
102                 TetrisGameState tetris_state;
103                 FILE* fp = fopen(saves[i].filename, "r");
104                 size_t ret;
105                 if (!fp) {
106                         perror("While opening save files");
107                         fprintf(stderr, "Problematic file: '%s'\n", saves[i].filename);
108                         err = errno;
109                         goto out_err;
110                 }
111                 ret = fread(&tetris_state, 1, sizeof(TetrisGameState), fp);
112                 fclose(fp);
113                 if (ret < sizeof(TetrisGameState)) {
114                         fprintf(stderr,
115                                 "Only able to read %lu bytes of %lu expected
116                                 from savefile '%s'. Aborting...\n",
117                                 ret, sizeof(TetrisGameState), saves[i].filename);
118                         err = errno;
119                         goto out_err;
120                 }
121                 // Select the first metric to sort on, and the tiebreaker
122                 if (metric == M_LINES) {
123                         saves[i].pri_metric = tetris_state.lines;
124                         saves[i].sec_metric = tetris_state.score;
125                 } else {
126                         saves[i].pri_metric = tetris_state.score;
127                         saves[i].sec_metric = tetris_state.lines;
128                 }
129         }
130         // Sort
131         size_t num_to_print = strtoul(argv[2], NULL, 10);
132         qsort(saves, num_lines, sizeof(struct Save), uint_compare);
133         // Output results
134         for (i = 0; i < min(num_to_print, num_lines); i++)
135                 printf("%s\n", saves[i].filename);
136 out_err:
137         // Free all heap memory
138         free(in);
139         free(saves);
140         return err;
141 }
```

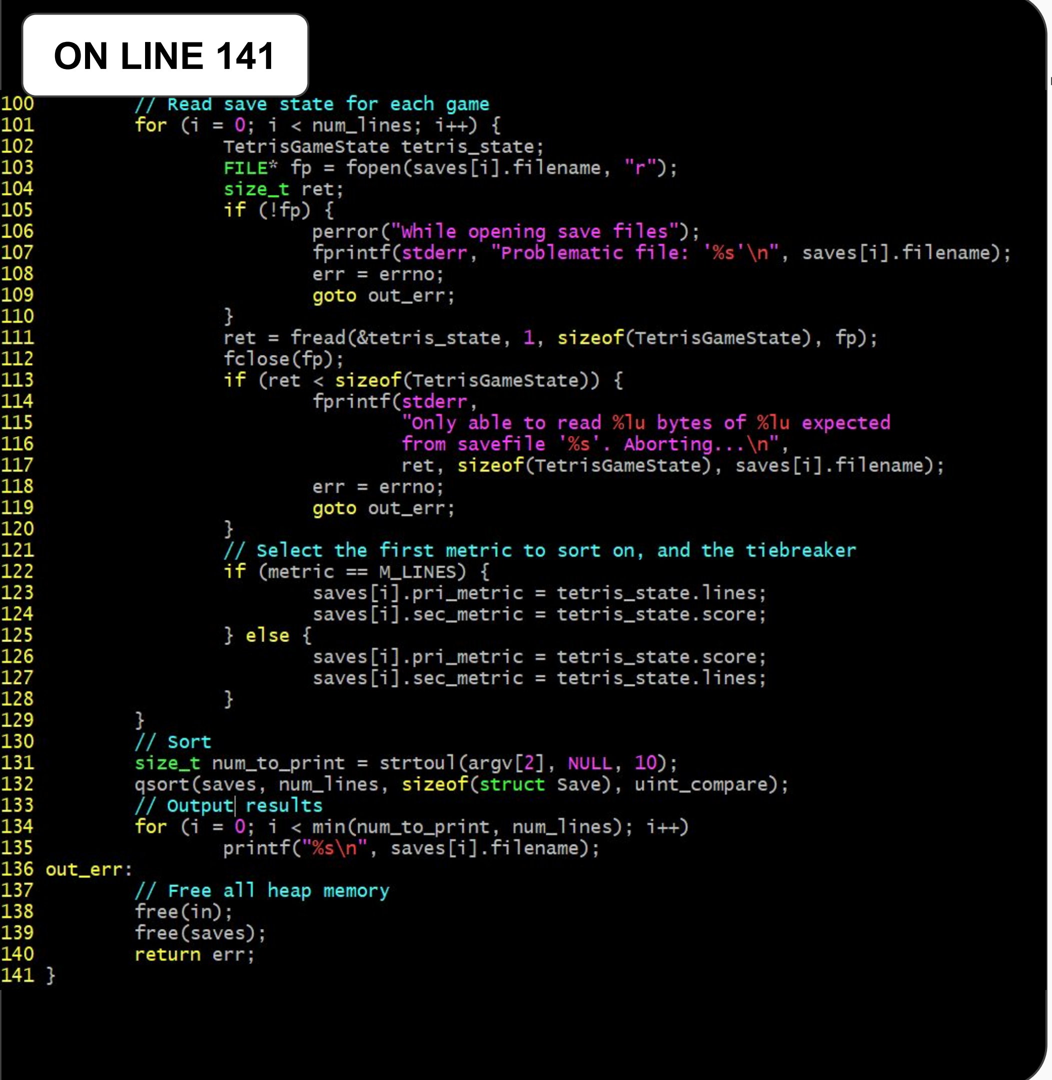


ON LINE 139



```c
100         // Read save state for each game
101         for (i = 0; i < num_lines; i++) {
102                 TetrisGameState tetris_state;
103                 FILE* fp = fopen(saves[i].filename, "r");
104                 size_t ret;
105                 if (!fp) {
106                         perror("While opening save files");
107                         fprintf(stderr, "Problematic file: '%s'\n", saves[i].filename);
108                         err = errno;
109                         goto out_err;
110                 }
111                 ret = fread(&tetris_state, 1, sizeof(TetrisGameState), fp);
112                 fclose(fp);
113                 if (ret < sizeof(TetrisGameState)) {
114                         fprintf(stderr,
115                                 "Only able to read %lu bytes of %lu expected
116                                 from savefile '%s'. Aborting...\n",
117                                 ret, sizeof(TetrisGameState), saves[i].filename);
118                         err = errno;
119                         goto out_err;
120                 }
121                 // Select the first metric to sort on, and the tiebreaker
122                 if (metric == M_LINES) {
123                         saves[i].pri_metric = tetris_state.lines;
124                         saves[i].sec_metric = tetris_state.score;
125                 } else {
126                         saves[i].pri_metric = tetris_state.score;
127                         saves[i].sec_metric = tetris_state.lines;
128                 }
129         }
130         // Sort
131         size_t num_to_print = strtoul(argv[2], NULL, 10);
132         qsort(saves, num_lines, sizeof(struct Save), uint_compare);
133         // Output results
134         for (i = 0; i < min(num_to_print, num_lines); i++)
135                 printf("%s\n", saves[i].filename);
136 out_err:
137         // Free all heap memory
138         free(in);
139         free(saves);
140         return err;
141 }
```





ON LINE 141

```c
100         // Read save state for each game
101         for (i = 0; i < num_lines; i++) {
102                 TetrisGameState tetris_state;
103                 FILE* fp = fopen(saves[i].filename, "r");
104                 size_t ret;
105                 if (!fp) {
106                         perror("While opening save files");
107                         fprintf(stderr, "Problematic file: '%s'\n", saves[i].filename);
108                         err = errno;
109                         goto out_err;
110                 }
111                 ret = fread(&tetris_state, 1, sizeof(TetrisGameState), fp);
112                 fclose(fp);
113                 if (ret < sizeof(TetrisGameState)) {
114                         fprintf(stderr,
115                                 "Only able to read %lu bytes of %lu expected
116                                 from savefile '%s'. Aborting...\n",
117                                 ret, sizeof(TetrisGameState), saves[i].filename);
118                         err = errno;
119                         goto out_err;
120                 }
121                 // Select the first metric to sort on, and the tiebreaker
122                 if (metric == M_LINES) {
123                         saves[i].pri_metric = tetris_state.lines;
124                         saves[i].sec_metric = tetris_state.score;
125                 } else {
126                         saves[i].pri_metric = tetris_state.score;
127                         saves[i].sec_metric = tetris_state.lines;
128                 }
129         }
130         // Sort
131         size_t num_to_print = strtoul(argv[2], NULL, 10);
132         qsort(saves, num_lines, sizeof(struct Save), uint_compare);
133         // Output results
134         for (i = 0; i < min(num_to_print, num_lines); i++)
135                 printf("%s\n", saves[i].filename);
136 out_err:
137         // Free all heap memory
138         free(in);
139         free(saves);
140         return err;
141 }
```

Stack

Heap

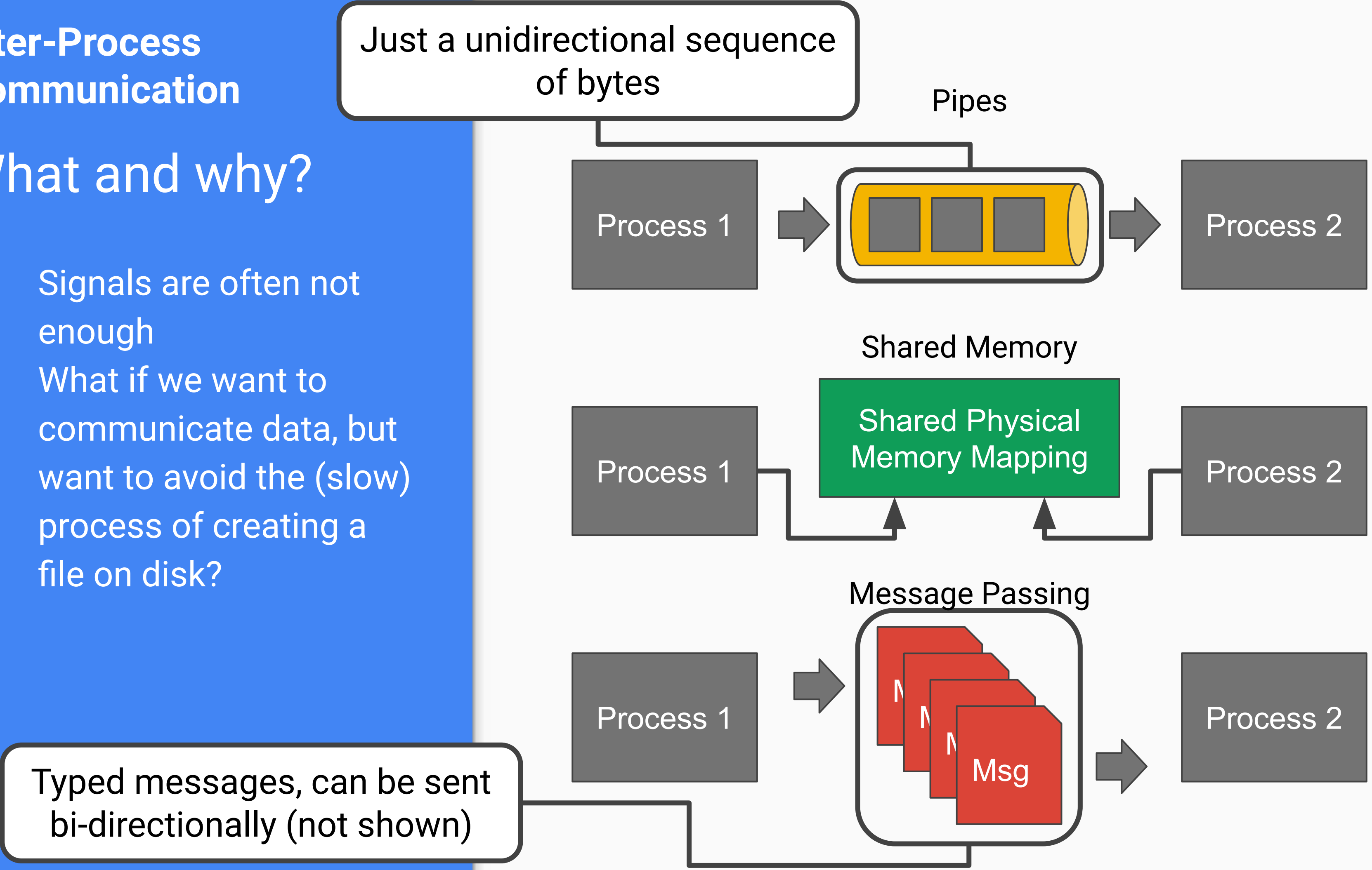
Inter-Process Communication (IPC)

Beyond Signals

Inter-Process Communication

What and why?

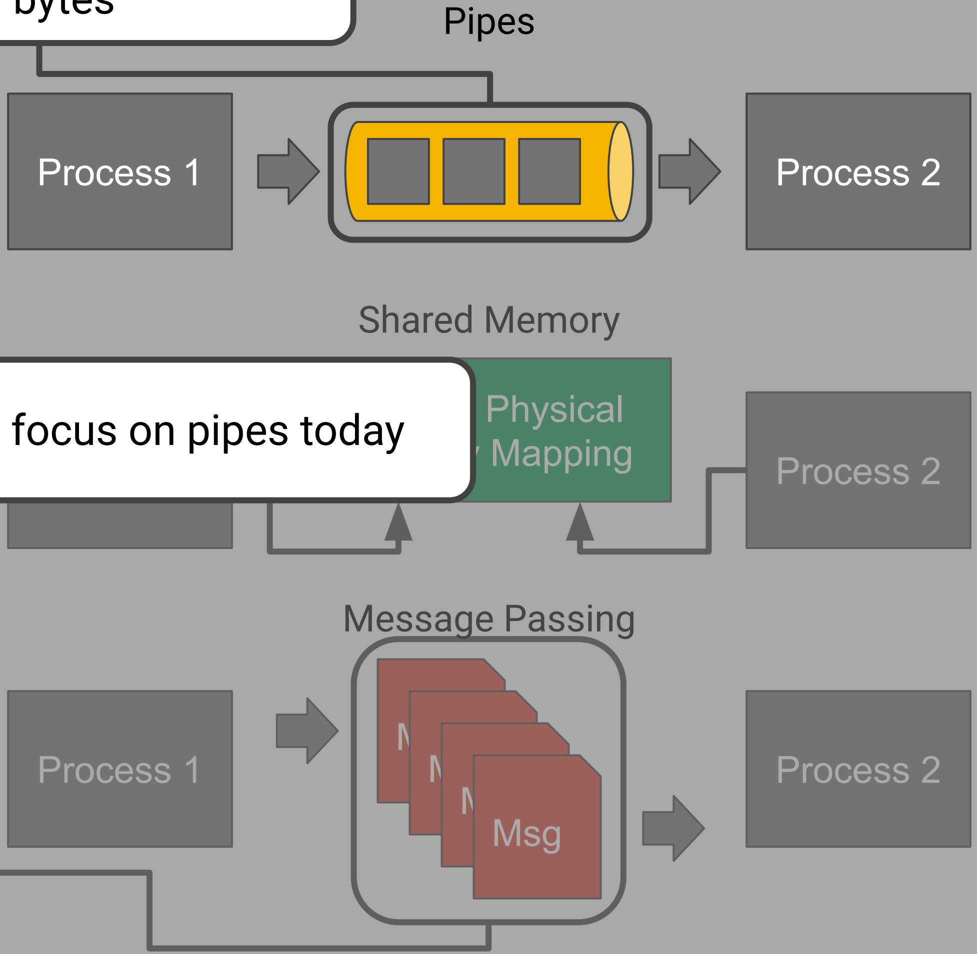
- ➔ Signals are often not enough
- ➔ What if we want to communicate data, but want to avoid the (slow) process of creating a file on disk?


Just a unidirectional sequence of bytes
Pipes
Process 1
Process 2
Shared Memory
Process 1
Shared Physical Memory Mapping
Process 2
Message Passing
Process 1
Msg
Process 2
Typed messages, can be sent bi-directionally (not shown)

Inter-Process Communication

What and why?

- ➔ Signals are often not enough
- ➔ What if we want to communicate data, but want to avoid the (slow) process of creating a file on disk?



Just a unidirectional sequence of bytes

We'll focus on pipes today

Typed messages, can be sent bi-directionally (not shown)

Questions?

Contact:
Email: hacker@unc.edu
Twitter: @JJBakita
Web: https://cs.unc.edu/~jbakita

Old Well, University of North Carolina at Chapel Hill, Winter 2017