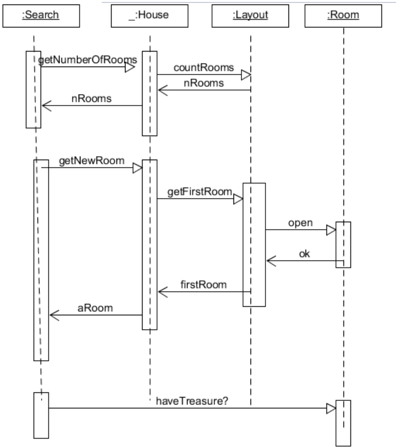
Outline

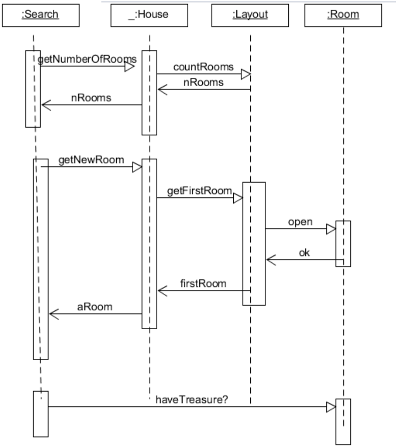
- How to do the homework, and
- because the homework is really easy,
- how to process command line arguments

We can do this homework. I



- Figure out which functions are implied by the sequence diagram.

We can do part one. I

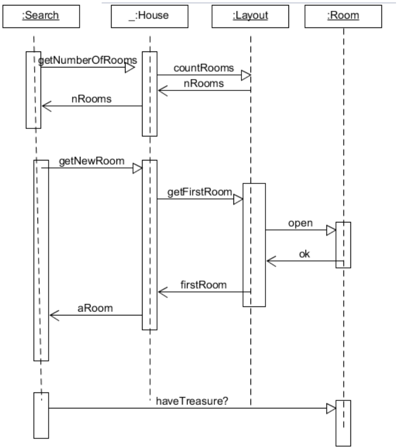


- To infer a function the sequence diagram, we use a message that is not a return.

We can do part one. II

- The first example is “getNumberOfRooms”.
- There is a return under it, which conveys the number of rooms (probably an integer).
- The return is associated with “getNumberOfRooms”, so it does not imply an additional function.
- The second example is “countRooms”.
- After we collect all the messages from the sequence diagram,
- we further reflect that every production function implies a test function.
- While we’re on a roll here, every test function implies at least two test cases.

We can do part two. I

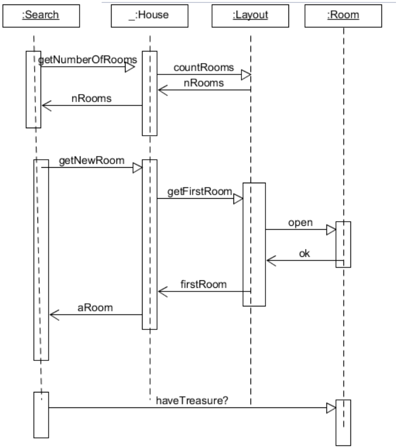


- We can make a .h file,
- and a .c file,

We can do part two. II

- for each block that we see in the sequence diagram.
- (Not just one, even though the homework uses singular.)
- The first block is Search, so we expect to create Search.c and Search.h.
- We have have some function prototypes for functions identified in part one.
- They need to occur in some .h file.
- The first function, “getNumberOfRooms”, is invoked by Search on House.
- From this we can conclude that the .h file containing the function prototype for “getNumberOfRooms” needs to appear in an include directive in both Search.c and House.c
- Moreover, we would write that prototype into House.h, because the function implementation will be found in House.c

We can do part three. I



- We can write function prototypes.

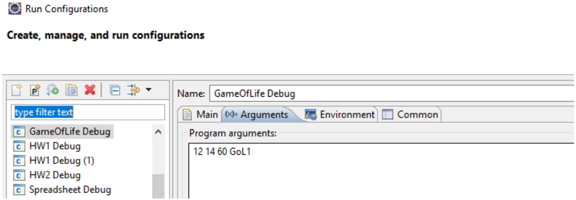
We can do part three. II

- From the quiz, you've demonstrated you know what goes into a function prototype.
- For the homework, you need to create function prototypes, into the appropriate .h file,
- which is House.h in the first case, because the message arrow ends on that lifeline.

Now we can consider processing command line arguments.

- Command line arguments are grouped in two variables, argc and argv.
- The variable argc is a count of the arguments,
- The first one is the program name.
- So, if argc is one, and if we don't care what executable name was called,
- we probably don't care about the command line anymore.
- When argc > 1, there is information we probably want to get.
- The command line information is suitable for typing with a keyboard, so
- it is taken as a vector of strings.
- The count argc tells us how many strings there are.

We can transmit “command line” arguments.



We can receive “command line” arguments.

```
int main(int argc, char* argv[])
{
    puts("!!!Let’s do HW1!!!");
    if(tests())
    {
          production(argc, argv);
```

We can interpret "command line" arguments. I

```
#include "production.h"
#include <string.h> //for strcpy

bool production(int argc, char* argv[])
{
bool answer = false;
if(argc <=1) //no interesting information
{
     puts("Didn't find any arguments.");
    fflush(stdout);
}
else //there is interesting information
{
    printf("Found %d arguments.\n", argc);
    fflush(stdout);
    long rows_L;
```

We can interpret “command line” arguments. II

```
int rows =0;
long cols_L;
int cols = 0;
long gens_L;
int gens = 0;
bool print = false;
bool pause = false;
char filename[FILENAMELENGTHALLOWANCE];
for(int i = 1; i<argc; i++) //don't want to read argv[0
{//argv[i] is a string
//in this program our arguments are NR, NC, gens, filen
//because pause is optional, argc could be 6 or 7
//because print is optional (if print is not present, n
    switch(i)
     {
    case 1:
    //this is NR
```

We can interpret “command line” arguments. III

```
        rows_L = strtol(argv[i]);
        rows = (int) rows_L;
        break;
    case 2:
    //this is NC
        cols_L = strtol(argv[i]);
        cols = (int) cols_L;
        break;
    case 3:
    //this is gens
        gens_L = strtol(argv[i]);
        gens = (int) gens_L;
        break;
    case 4:
        //this is filename
        if(strlen(argv[i]>FILENAMELENGTHALLOWANCE))
        {
```

We can interpret “command line” arguments. IV

```
            puts("Filename is too long.");
        }
        else
        {
            strcpy(filename, argv[i]);
        }
        break;
    case 5:
    //this is the optional print
        print= true;
        break;
    case 6:
    //this is the optional pause
        pause = true;
        break;
    default:
        puts("Unexpected argument count.");
```

We can interpret “command line” arguments. V

```
                break;
            }
        }
    }
    return answer;
}
```

Perhaps of interest: representing a graph as an adjacency matrix.

- Let's say we have a graph, with nodes numbered 0 through N-1.
- We are concerned about representing the edges.
- Recall, these are pairs of nodes.
- So with an N by N array,
- we can place in cell (r,c) a true,
- when node r appears in an edge pair followed by node c.
- For undirected graphs, a true in (r,c) implies a true in (c,r).
- All cells on the main diagonal contain true.

An adjacency matrix can be written into, and read from a file.

- We have seen an example of reading from a file.
- One way to store an array is one row per line.

We could represent a house as a graph.

- Each room can be a node.