

# Step 11: Arrays and Pointers

---

This assignment is done in class on (monday)03-31-14 AND (friday)03-28-14.

This assignment is due for monday class 03-31-14 AND for the friday class 03-28-14.

## Getting Started

---

Create a new program called wumpus1.c with the following code:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

/*
 * Name : <insert name here>
 *
 * Simple Wumpus game in 1D
 */

/* Add any #defines here */

/* Add any function prototypes here */

int main()
{

    /* Seed the random number generator */
    srand(time(NULL));

}
```



Be sure it compiles and runs. Of course, it won't do anything, yet.

**Stop at this point until directed to proceed**

## 1. A 1D Wumpus Game

---

There once was a very popular computer game called [Hunt the Wumpus](#). We're going to experiment with variations on that game today.

The first version we are going to create is going to be a 1D Hunt the Wumpus game. Then we will create a 2D version. In this game, you are the Agent. You are in a cave system consisting of connected rooms all in a row. We'll use 20 rooms for the 1D version. You are trying to find the Wumpus and shoot it with your arrow. You kill the Wumpus to win. If you move into the room with the Wumpus, it eats you.

I am generally not going to indicate where you should compile and run your program. You should be figuring that out by now yourself. Try to at least compile often. Also, I will not always indicate that you need to declare a variable. You should also figure that out. Make any loop iterator variables (like the `i` in `for(i=0; i<10; i++)`) type `int`.

## Describing the World

We need to represent our cave system which has you, the Agent, and the Wumpus in it. We are also going to keep track of the ends of our cave, where we can no longer move. We'll keep track of our own location a different way, but for now we'll need to identify the Wumpus and ends. When we have things to represent in C, one choice is to assign

each of them a number, then use a `#define` to give that number a name. Add these `#defines` to your program to identify the things we will have in our cave:

```
/* Id's for things in our Cave */
#define Empty 0
#define Wumpus 1
#define End 2
```

Put the `#defines` after any includes, but before any function prototypes. Think of the beginning of your `.c` file as `#includes`, `#defines`, then function prototypes. I put comments in the code to indicate where these go.

To keep track of our cave, we'll use an array. Add this `#define` to indicate the cave size:

```
/* Number of rooms in our Cave */
#define CaveSize 20
```

Now we need to create the cave. The cave is a 1D array that is 2 larger than the number of rooms. The reason is that we're going to put two ends on our cave to make it easier to navigate later on. I find it useful to keep track of that as a separate `#define`. Add this one:

```
#define ArraySize (CaveSize + 2)
```

You can put some math in `#define` like this, but put it in parenthesis because that ensures the order of operations will be correct.

Add this array declaration to your main function:

```
int cave[ArraySize];
```

We need to keep track of what room we are in and what direction we are facing. Instead of an integer, I'm going to use a pointer. A pointer holds the address of something in memory. Our pointer will hold the address of the room we are currently in. Add this variable to your main function:

```
int *agentRoom;
```

To keep track of what direction I am facing, I'm going to use another `#define`. Add these `#defines` to your program for the directions I can face:

```
/* Directions I can face */
#define Left 0
#define Right 1
```

Then, add this variable to keep track of the direction you are facing:

```
int agentDirection;
```

## Creating the World

We have created a way to store our cave system, but we have not yet put anything into it. We will need to place ourselves and the Wumpus in the cave and make sure all rooms are initialized properly. When doing things like this, don't do them in your main function! Adding a function to do these tasks groups them together and makes your program easier to understand. Create this function:

```
void CreateWorld(int cave[])
{
}
```

Add a call to CreateWorld to your main function AFTER the random number generator has been seeded:

```
int main()
{
    int cave[CaveSize + 2];
    int *agentRoom;
    int agentDirection;

    /* Seed the random number generator */
    srand(time(NULL));

    CreateWorld(cave);
}
```

Now, what does CreateWorld need to do? It needs to:

1. Initialize the cave rooms to all empty.
2. Set the ends of the cave to End.
3. Randomly place the Wumpus.

We are now adding the code to CreateWorld. To initialize the cave rooms we need a loop:

```
/* Initialize cave to empty */
for(i = 0; i<ArraySize; i++)
{
    cave[i] = Empty;
}
```

Set the cave ends:

```
/* Set the ends */
cave[0] = End;
cave[ArraySize-1] = End;
```

We'll place the Wumpus in the cave randomly. The Wumpus can go into array locations 1 to CaveSize-2 inclusive. We are going to place other things in our cave, so let's make the simple rule that we can put the Wumpus in any empty room. Let's create a function to find a random empty room. I am going to have this function return a pointer to a room in the cave. Since the rooms are of type int, the function will return int \*, a pointer to an int. Add this function:

```
int *GetEmptyRoom(int cave[])
{
    int room;

    do
    {
        room = rand() % ArraySize;
    } while(cave[room] != Empty);

    return &cave[room];
}
```

Read this and be sure you understand how it works. It simply creates a random number from 0 to CaveSize - 1. The % operator is called modulo and means the remainder after integer division, so that will be a number from 0 to CaveSize - 1. If the array location is not empty, it tries again.

If you get an error like this:

```
wumpus1.c:63: error: conflicting types for 'GetEmptyRoom'
```

That's an indication you forgot to add the function declaration for the GetEmptyRoom function.

With that function we can put our Wumpus into a room in the CreateWorld function. First add this variable to CreateWorld:

```
int *room;
```

Then we can add the code to the end of CreateWorld to put the Wumpus into a random room:

```
/* Get a random empty room and put the Wumpus in it */  
room = GetEmptyRoom(cave);  
*room = Wumpus;
```

We have a cave system. One last thing we need to do is put the agent in the cave and give them a position. Add this code to main right after the call to CreateWorld:

```
agentRoom = GetEmptyRoom(cave);  
agentDirection = rand() % 2;
```

The second line just generates a random direction.

[Click here](#) to see what the CreateWorld function should look like.

## The Game Loop

We are going to have a loop that runs as long as our game continues. Add this loop to the end of your main function:

```
/* The game loop */  
while(true)  
{  
}
```

If you run your program right now it will get stuck in a continuous loop, so don't do so just yet. We are going to type in commands. Add this variable to main:

```
char command[20];
```

Then put this code inside the while loop to get the command: ([click if not sure where](#)).


```
/* Get the command */  
printf("Command: ");  
scanf("%20s", command);
```

We want to test for different commands that may be entered. To do so we will need to use our string library functions. Add this #include to the beginning of your program:

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

Then, after the scanf, add this code to test for the quit command: ([click if not sure where](#)).

```
if(strcmp(command, "quit") == 0)  
{  
    /* Exit, we are doing */  
    break;  
}  
else  
{  
    printf("I don't know what you are talking about\n");  
}
```

 This should run and you should be able to type commands that it gripes about and finally type quit to exit the program.

## DisplayWorld

I'm going to create a simple function to display the cave. We'll do that while we develop the code and turn it off later on. Add this function:

```
void DisplayWorld(int cave[], int *agent, int agentDir)
{
}
```

To display the status of the world, the function need to know the cave and where the agent is. Those are the three things I am passing. Add this call to the beginning of the game loop: ([click if not sure where](#)).

```
DisplayWorld(cave, agentRoom, agentDirection);
```

To display this, we're going to entirely use pointer arithmetic this time. Here's a loop that loops over every room of the cave from beginning to end. Add this to DisplayWorld:

```
int *room;

for(room = cave + 1; *room != End; room++)
{
}
```

The variable room is a pointer to an integer. The for loop initially sets it to point to the second room, since the first is an End. Then we loop over every room until the room pointed to by the variable room (\*room) is an end. Put this code in the for loop to output the room contents:

```
switch(*room)
{
case Wumpus:
    printf("-W- ");
    break;

default:
    printf(" . ");
    break;
}
```

Then add this printf statement to the end of DisplayWorld:

```
printf("\n");
```

When you run this, it will show the room the Wumpus is in:

```
cbowen@ubuntu:~/cse251$ ./wumpus1
. -W- . . . . . . . . . . . . . . . .
Command:
```

Now we want to indicate where the Agent currently is. To do that, we will test the value of room and see if it is equal to Agent. At the beginning of the for loop, before the switch, add this code: ([click if not sure where](#)).

```
if(room == agent)
{
    switch(agentDir)
    {
case Left:
```

```

        printf("<A ");
        break;

    case Right:
        printf(" A> ");
    }

    continue;
}

```

The keyword *continue* in C means we jump to the next loop iteration immediately. Think of it as jumping to the bottom of the for loop. After *continue*, the rest of the code in the for loop does not get executed.

When you run this you should see the world something like the following:

```

cbowen@ubuntu:~/cse251$ ./wumpus1
. . . . . A> . -W- . . .
Command: quit

```

Of course, every run will have things at a different place.

[Click here](#) to see what the DisplayWorld function should look like.

## Command: move

Earlier, we added code to handle the quit command. That's in the game loop. Now we'll add code to handle the move command. If we are moving left, we can decrease the agentRoom pointer by 1: `agentRoom -= 1`. Otherwise, we increase by 1: `agentRoom += 1`. I'm going to create a function that we can call that will convert the current direction into either a +1 or -1, depending on which way we are facing. Add this function to do that:

```

int DifferenceByDirection(int dir)
{
    if(dir == Left)
        return -1;
    else
        return 1;
}

```

But, we can't do that if we would move into an end room, so we need to check for that.

Here's the code you will add to the game loop: ([click if not sure where](#)).

```

    else if(strcmp(command, "move") == 0)
    {
        /* Move command */
        /* What way do we need to go? */
        direction = DifferenceByDirection(agentDirection);
        if( *(agentRoom + direction) != End)
            agentRoom += direction;
    }

```

You should be able to move, now.

## Command: turn

The turn command is pretty trivial, then:

```

    else if(strcmp(command, "turn") == 0)
    {
        agentDirection = !agentDirection;
    }

```

## DisplayStatus

What makes this interesting is to display the status. Add this function where we will display our status:

```
bool DisplayStatus(int cave[], int *agent)
{
    /* We will return true to indicate we are dead! */
    return false;
}
```

Add this call to the game loop right before the call to DisplayWorld:

```
if(DisplayStatus(cave, agentRoom))
    break;
```

We are using the idea that DisplayStatus will also test for a condition that ends the game. If it returns true, the game will be over.

Here are the conditions we need to check:

- If the Agent is in the room with the Wumpus, the Wumpus eats the Agent and the game ends.
- If the Wumpus is one room away from the Agent, the Agent smells the Wumpus.
- If there is no Wumpus at all, we have killed it and we win.

The first of these is easy to test for in DisplayStatus:

```
if(*agent == Wumpus)
{
    printf("You have been eaten by the Wumpus\n");
    return true;
}
```

The next is also easy to test for if we remember that the locations next to the agent are: agent - 1 and agent + 1. Of course, agent is a pointer, so agent + 1 is also a pointer. We can use \* to see what the pointer is pointing at. We used that before to test the room the agent is in for the Wumpus: \*agent. To test the rooms next to us, just do this: \*(agent+1) and \*(agent-1). Here is what the code looks like:

```
if(*(agent-1) == Wumpus || *(agent + 1) == Wumpus)
{
    printf("I smell a Wumpus\n");
}
```

This should work and you should be able to turn, move, and approach the Wumpus.

You should be able to move, now.

[Click here](#) to see what DisplayStatus should look like at this point.

As you likely have guessed, the call to DisplayWorld makes this game too easy. So, just *comment out* that call. The game is a bit more interesting that way:

```
/* DisplayWorld(cave, agentRoom, agentDirection); */
```

It is very common when writing code like this that we will create a function that is useful while we are developing and is removed later. In this case we wanted to see what was in the rooms. We created a function to do so, then removed that later to play the game.

## Your Task

1. Add a command: fire. When the fire command is selected, you fire your arrow. It travels up to 3 rooms in the direction you are looking. If it enters the room with the Wumpus, the Wumpus is killed. Set the room value to Empty to indicate he is gone. This should not take much code to get working.

You need to also add the ability for DisplayStatus to detect that we have won.

[Click here](#) if you would like some hints.

When this is done you should have a usable, if simple, game. **Turn in wumpus1.c** via <http://secure.cse.msu.edu/handin/>

## Preparing for 2D

Create a new program wumpus2.c with the following code:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

/*
 * Name : <insert name here>
 *
 * Simple Wumpus game in 2D
 */

/* Id's for things in our Cave */
#define Empty 0
#define Wumpus 1
#define End 2
#define Pit 3

/* Number of rooms in our Cave in each dimension */
#define CaveSize 10
#define ArraySize (CaveSize + 2)

/* Directions I can face */
#define Left 0
#define Up 1
#define Right 2
#define Down 3

int main()
{
    int *agentRoom;
    int agentDirection;
    char command[20];

    /* Seed the random number generator */
    srand(time(NULL));

    /* The game loop */
    while(true)
    {
        /* Get the command */
        printf("Command: ");
        scanf("%20s", command);
        if(strcmp(command, "quit") == 0)
        {
            /* Exit, we are doing */
        }
    }
}
```



```

        break;
    }
    else if(strcmp(command, "move") == 0)
    {
    }
    else if(strcmp(command, "turn") == 0)
    {
    }
    else if(strcmp(command, "fire") == 0)
    {
    }
    else
    {
        printf("I don't know what you are talking about\n");
    }
}
}

```

Stop at this point until directed to proceed

## 2. Wumpus in 2D

This is all well and good, but it will be more fun in 2D instead of 3D. I've given you some of the code to get you started. This should make things a bit easier. I also made the cave a bit smaller since it will have CaveSize \* CaveSize rooms (100) now.

### The Cave Array

Add this variable to create a 2D array in main:

```
int cave[ArraySize][ArraySize];
```

### CreateWorld

Again, we'll need a CreateWorld function. But, we can't pass a 2D array the same way we did a 1D array. Here's how we can pass the cave array to our CreateWorld function:

```
void CreateWorld(int cave[ArraySize][ArraySize])
{
}

```

As usual, we add the function declaration to the beginning of the program. But, in this case the function declaration is dependent on ArraySize, so it must be after that #define. Here is where I put it in my program:

```
/* Number of rooms in our Cave in each dimension */
#define CaveSize 10
#define ArraySize (CaveSize + 2)

void CreateWorld(int cave[ArraySize][ArraySize]);

```

Add the call to this function to main before the game loop, but after the srand call:

```
CreateWorld(cave);
```

We are now working with 2D arrays. So, we have to have two loops to iterate over all of the locations in the array. Here's the code for CreateWorld that will set all of the rooms to Empty:

```
int i, j;

for(i = 0; i < ArraySize; i++)

```

```

{
    for(j = 0; j<ArraySize; j++)
    {
        cave[i][j] = Empty;
    }
}

```

See how we work with a 2D array? We have two subscripts, so it works out to be `cave[i][j]` using two sets of brackets. This is different than many other languages, where this might be `[i, j]`. But, we are in C and this is the way it is done.

Here's an easy way to modify that loop to set the ends as well. Replace the previous version with this one. Do you understand how it works?

```

int i, j;

for(i = 0; i<ArraySize; i++)
{
    for(j = 0; j<ArraySize; j++)
    {
        if(i == 0 || j == 0 || i == ArraySize-1 || j == ArraySize-1)
        {
            cave[i][j] = End;
        }
        else
        {
            cave[i][j] = Empty;
        }
    }
}

```

Before, we generated a random location for the Wumpus right here. This time I am going to create a function to generate a random empty room that returns a pointer to the room instead of an integer. I think that's easier, especially in the 2D case. Here's the function to add:

```

int *GetEmptyRoom(int cave[ArraySize][ArraySize])
{
    int row, col;
    int *room;

    do
    {
        /* We need a random number in each dimension */
        row = rand() % ArraySize;
        col = rand() % ArraySize;

        room = &cave[row][col];
    } while(*room != Empty);

    return room;
}

```

Again, be sure to put the function declaration after `#define ArraySize`. Be sure you understand what this function is doing.

Now we can add code to `CreateWorld` to place the Wumpus in a random room. Add this pointer variable to `CreateWorld`:

```
int *room;
```

Then add this code to the end of `CreateWorld`:

```
room = GetEmptyRoom(cave);
*room = Wumpus;
```

## Place the Agent

Add this code to main right after the call to CreateWorld:

```
agentRoom = GetEmptyRoom(cave);
agentDirection = rand() % 4;
```

## DisplayWorld

I want to display the cave again. I'll just give you the code to display the cave, since I did a few interesting things in it:

```
void DisplayWorld(int cave[ArraySize][ArraySize], int *agent, int agentDir)
{
    int row, col;
    int *room;

    /* Loop over the rows of the cave */
    for(row=1; row <= CaveSize + 1; row++)
    {
        /*
         * This loop lets us print an up direction
         * above the agent or a v below the agent
         */

        for(col=1; col<=CaveSize; col++)
        {
            if(&cave[row][col] == agent && agentDir == Up)
            {
                printf(" ^ ");
            }
            else if(&cave[row-1][col] == agent && agentDir == Down)
            {
                printf(" v ");
            }
            else
            {
                printf("   ");
            }
        }

        printf("\n");
        if(row > CaveSize)
            break;

        /*
         * This loop prints the agent or the room contents
         */

        for(col=1; col<=CaveSize; col++)
        {
            room = &cave[row][col];
            if(room == agent)
            {
                switch(agentDir)
                {
                    case Left:
                        printf("<A ");
                        break;
```

```

        case Right:
            printf(" A> ");
            break;

        default:
            printf(" A  ");
            break;

    }
    continue;
}

switch(*room)
{
case Wumpus:
    printf("-W- ");
    break;

default:
    printf(" .  ");
    break;
}

}

printf("\n");
}
}

```

Yes, I know this is long. Just paste it in to use it to see what is going on. When you have to add things later, they will go in the bottom switch statement.

Add this code. Then add a call to this function in the game loop:

```
DisplayWorld(cave, agentRoom, agentDirection);
```

When you run the program you should now see the location of the agent and the Wumpus:

```
cbowen@ubuntu:~/cse251$ ./wumpus2
```

```

. . . . . -W- . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
.  A  . . . . . . .
  v
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
Command: quit

```

**Command: move**

To implement the move function in 2D, we need to do more than just `agentRoom++` or `agentRoom--`. That is fine for moving left or right. But, what about moving up? To move up, you can take advantage of the fact that C puts the rows of a 2D array one after the other in memory, something called *row major*. So, the location right above the current location is the same location in the previous row, which is `ArraySize` to the left. I found it helpful to make a function that tells what we add to a location given a direction we are looking. Add this function to your program:

```
int DifferenceByDirection(int dir)
{
    switch(dir)
    {
        case Up:
            return -ArraySize;

        case Down:
            return ArraySize;

        case Left:
            return -1;

        case Right:
            return 1;
    }
}
```

This makes the move command easy:

```
else if(strcmp(command, "move") == 0)
{
    d = DifferenceByDirection(agentDirection);

    if(*(agentRoom + d) != End)
        agentRoom += d;
}
```

I just determine the amount to move in memory based on the current value of Up, Down, Left, Right. If that location is not an End, we can move to it.

## Command: turn

The turn command is almost as easy as before.

```
else if(strcmp(command, "turn") == 0)
{
    agentDirection++;
    if(agentDirection > Down)
        agentDirection = Left;
}
```

## Final Tasks

1. Add a `DisplayStatus` function to your program just like we did in `wumpus1.c`. Note that you can smell a Wumpus if you are to the left, right, above, or below the Wumpus. That's locations `agent-1`, `agent+1`, `agent-ArraySize` and `agent+ArraySize`.

2. Add the fire command and make it work. You fire three rooms in the direction you are looking, just as before.

2. The real Wumpus game also has *pits*. A pit is a hole you will fall into if you walk into the room. Add support for pits. For this you will need:

- Add 10 random pits to the cave.
- If you are adjacent to a pit, print: "I feel a draft".
- Add code to `DisplayWorld` to show where the pits are.

- Add code to DisplayStatus that prints "You fell into a pit" and returns true if you are in the room that contains a pit.

When you turn in your game, please leave DisplayWorld enabled so Sorour can see where things are in your cave and does not have to search for them.

**Turn in wumpus2.c via <http://secure.cse.msu.edu/handin/>**

**CSE 251**