CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 35 Apr 15, 2013

Bunch O' Stuff: How Memory Works, FSM Code, Game HW

Lecture Goals

1. Memory
2. Stack vs Heap
3. Scope
4. FSM Questions?
5. HW 11+12: A Game

Extra Credit Opportunity

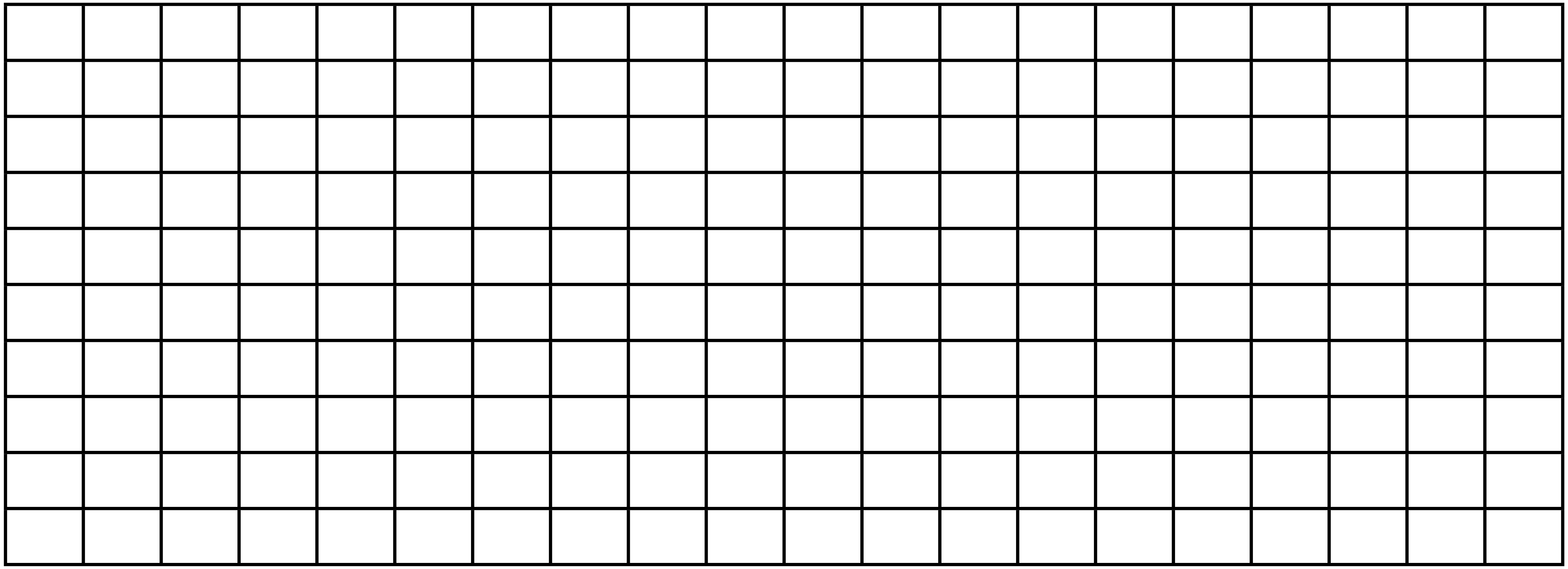
Go through my old slides and PDFs and find bugs. Email me with [cs2270 bug] in the subject (with the square brackets) with:

* The filename or lecture number
* The specific problem
* What the correct text should be

This is a good way to cash in on studying for the final. **I will cut this off on May 3.**

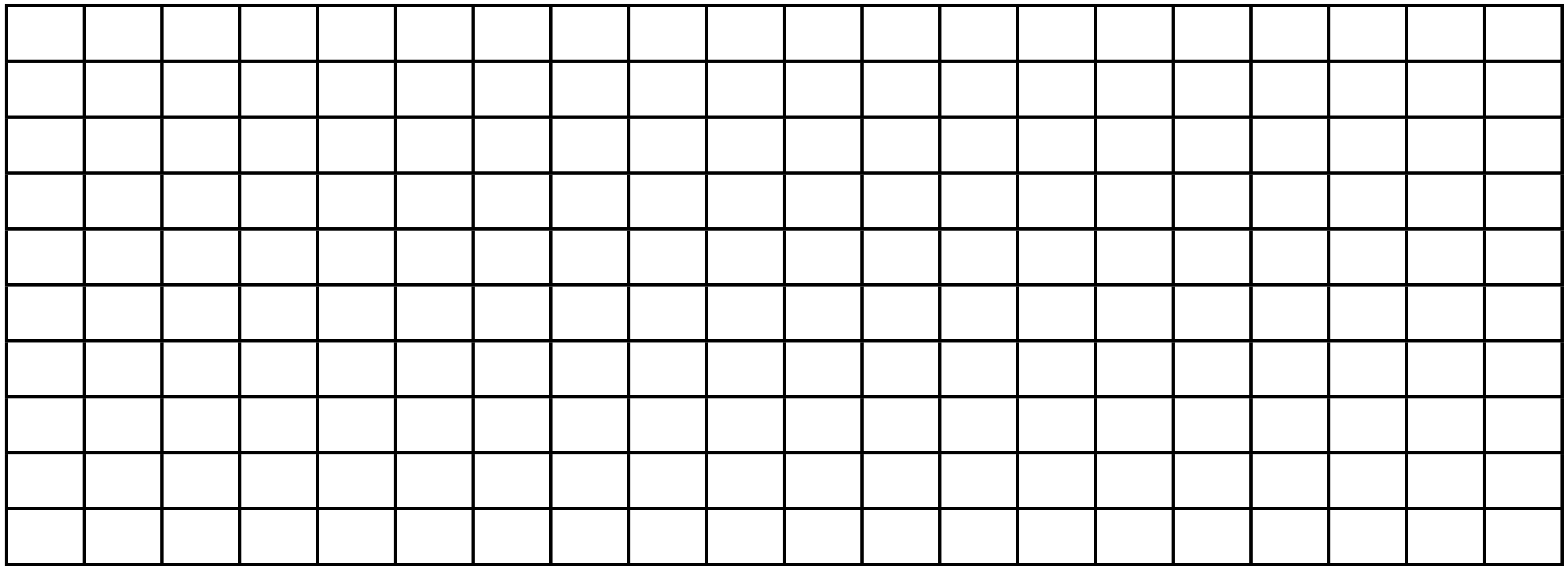
Memory

A CPU has to store data in *memory*. This is often called RAM (random access memory) but it could actually be something else. It is *not* a hard disk, though.



Memory

We can draw memory like this. This has 20 columns and 10 rows, so this memory has 20 * 10 = 200 cells. A real computer has BILLIONS of these cells.



Memory

Lets say we have a memory like below. There's 100 slots, going sequentially from left to right, top to bottom.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30										
40										
50										
60										
70										
80										
90										

Memory

When we create a new variable in C++, the computer has to allocate memory to store it. Let's make an int:

```
int x = 9001;
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30										
40										
50										
60										
70										
80										
90										

Memory

First the computer find some spot in memory to put the variable. I arbitrarily chose address 37 here. It is highlighted.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Memory

37 is that cell's *address*.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								🟥		
40										
50										
60										
70										
80										
90										

Memory

37 is that cell’s *address*. That’s variable x’s address.
9001 is that cell’s *value*. That’s variable x’s value.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Address Of Operator

```
&x
```

An ampersand in front of a variable like this says “I demand you give me the address of this variable!”

Memory

We can prove to ourselves this is true:

```
cout << x << endl; // outputs 9001
cout << &x << endl; // outputs 37
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Pointer Variables

```
int* xp = &x;
```

The address-of operator produces a *pointer.* We write pointers as <data_type>*. E.g.: int*, float*, string*, bool*, and so on.

Memory

Lets create a pointer variable to x called xp.

```
int* xp = &x;
```

I put it in a cell, since it has to be stored somewhere.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60			37							
70										
80										
90										

Dereference Operator

```
int z = *xp;
```

We can *dereference* a pointer by putting an asterisk in front of it. This gives us the *value* of the variable stored in the memory slot it refers to.

Memory

Lets create a pointer variable to x called xp.

```
  int z = *xp;
```

I put it in a cell, since it has to be stored somewhere.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60			37							
70							9001			
80										
90										

Pointer Syntax

```
values
pointers
addresses
```

Pointer Syntax

```
int x=42;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

Pointer Syntax

```
int x=42;
int* x_ptr;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

`x_ptr` is a *pointer to an int*. It is declared with the data type `int*`, but its value is unspecified.

Pointer Syntax

```
x_ptr = &x;
```

We know the value of an int is some number. The value of `x` is 42.

The *value* of a *pointer variable* is an *address.*

In the above code, we take the *address of `x`*, and store it in `x_ptr`.

Pointer Syntax

`x` ←	This is our int x. Value is 42.
`&x` ←	Memory address of x.
`x_ptr` ←	int pointer. Value is the address of int x.
`*x_ptr` ←	Dereference an int pointer. Value is 42.

Pointers in Memory

```
int x = 42; // given memory cell 23
int* x_ptr = &x; // given memory cell 30
int* other = &x; // cell 18
```

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										

var	type	addr
x	int	23
x_ptr	int*	30
other	int*	18

Pointers in Memory

Be aware that this is a simplification. Variables can take up several contiguous cells in memory; we know how many they take up based on their data type.

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										

Also, this is a tiny memory. A real computer has **billions** of bytes of memory that we can store things in.

C++ Arrays

```
int some_data[123];
```

An *Array* in C++ has (sort of) the same purpose as Java's arrays and Python's lists. In C++ it is a contiguous stretch of memory, which is divided into elements that contain some value.

In this case, `some_data` is an array of integers, and there are 123 elements reserved.

C++ Arrays

```
... more above
52: 0
53: 0
54: 0
55: 0
56: -1608234075
57: 1175298596
58: 50716776
59: 1
60: 1
61: 0
... more below
```

If we were to print out each element of `some_data`, we might see something like this.

Declaring an array (or anything else in C++ for that matter) does *not* initialize that memory. If we read from the variable without setting a value, we might get *garbage*.

Alternate initializers

```
int stack_arr[] = { 5, 6, 20, 3 };

int* dyna_arr = new int[6];
```

Here are two other ways of declaring and initializing arrays. The first gives the values of the array, and the length is derived by how many values are there.

The second is tricky and is discussed later.

C++ Arrays

Reading and writing to arrays should be familiar syntax. Here we **write**. Initialize each cell to 0, 1, 2, 3, and so on.

```
int ret[num];
for (int i=0; i < num; i++) {
  ret[i] = i;
}
```

C++ Arrays

Reading from arrays uses the same syntax with the square brackets. Note that uninitialized arrays will have garbage in them when we read.

```
for (int i=0; i < n; i++) {
  cout << i << ": "
       << ret[i] << endl;
}
```

C++ Arrays as pointers

Earlier I mentioned this one:

```
int* dyna_arr = new int[6];
```

... and pointed out that it was tricky. The *value* of an array (remember how variables *all contain values*?) is actually the address of some spot in memory.

The statement 'new int[6]' allocates memory for an array of 6 integers and returns a pointer to that array.

C++ Arrays as pointers

```
int numbers[] = { 5, 6, 20, 3 };
print_array(numbers, 4); // prints 5, 6, 20, 3
int* weird = numbers; // note, NOT &numbers
print_array(weird, 4); // prints 5, 6, 20, 3
```

Since the *value* of an array variable is the address of its first element, we can read that array without the square brackets to get the address where the array data lives.

Heap vs. Stack

When we ask for memory to store variables, the operating system will allocate some spots in memory for us to use. There's two kinds of memory you need to be aware of: *stack* and *heap* memory.

I'll start with *stack* memory since it is related to *scope*, another idea we've talked about before.

Heap vs. Stack

When we declare a variable inside some block of code, that variable can be used inside that block. This is that variable's *scope*. But it will be unavailable once we leave that scope.

Scope

```
int get_sum_times_two(int a, int b) {
  int sum = a + b;
  int twice = sum * 2;
  return twice;
}

int main() {
  int ten = get_sum_times_two(2, 3);
  cout << "sum: " << sum << endl;
}
```

```
foo.cpp:19:22: error: use of undeclared identifier 'sum'
  cout << "sum: " << sum << endl;
```

Scope

```
int get_sum_times_two(int a, int b) {
  int sum = a + b;
  int twice = sum * 2;
  return twice;
}
```

Inside this function we declare two variables: `sum` and `twice`. Once the `get_sum_times_two` function completes, the computer no longer needs them. So the memory that we were using to store them is recycled.

Scope

```
int get_sum_times_two(int a, int b) {
  int sum = a + b;
  int twice = sum * 2;
  return twice;
}
```

The variables we declare here are *stack variables*. These are normal variables that we expect to use only temporarily. The operating system is doing us a service by cleaning up our mess. *But what if we wanted them to stay in memory afterwards?*

Heap vs. Stack

The other kind of variable is one that lives in the *heap* section of memory. These are persistent—the operating system will not recycle that memory until we explicitly tell it to.

(And, if you forget to do this, you end up with memory leaks and you gradually run out of free memory.)

Heap vs. Stack

To create a heap variable in C++, use the ‘new’ keyword. This is not in plain C (though C has its own way of doing this, and we won’t go into it).

```
int* dyna_arr = new int[6];
```

We use ‘new’ to allocate memory. This process always returns a *pointer* to the memory it allocated. Here, **`dyna_arr`** is a pointer to the block of memory holding our array of 6 integers.

Heap vs. Stack

```
int* dyna_arr = new int[6];
```

When the dyna_arr variable is no longer in scope:

- The variable dyna_arr is no longer valid, *but:*
- The memory it points to is still safe!

Heap vs. Stack

Check out the file:

cs1300/code/cpp/
stack_vs_heap_array_pointers.cpp

This demonstrates how you can really screw things up by using a stack variable when you're going to need to refer to its memory later on with a pointer.

Heap vs. Stack

```
int* make_stack_array(int num) {
  int ret[num]; // stack variable ‘ret’
  for (int i=0; i < num; i++) {
    ret[i] = i;
  }
  return ret; // ret’s memory will be recycled
}

int* make_dynamic_array(int num) {
  int* ret = new int[num]; // heap variable ‘ret’
  for (int i=0; i < num; i++) {
    ret[i] = i;
  }
  return ret; // ret’s memory will be left alone
}
```

Heap vs. Stack

```
int main() {
  int* stack_arr = make_stack_array(10);
  cout << "make_stack_array(10):" << endl;
  print_array(stack_arr, 10);

  int* dynamic_arr = make_dynamic_array(10);
  cout << "make_dynamic_array(10):" << endl;
  print_array(dynamic_arr, 10);
}
```

stack_arr contains garbage at this point.
dynamic_arr contains the data we want.

FSM Code

Questions about the FSM assignment?

Object-orientation? If so we can do another thing on C++ classes and objects on Wednesday.

HW 11 and 12

Spring 2013 - CSCI 2270

	Sa	Su	**Mo**	**Tu**	**We**	**Th**	**Fr**	
Jan	12	13	14	15	16	17	18	
	19	20	~~21~~	22	23	24	**25**	**hw 1**
Feb	26	27	28	29	30	31	**1**	**hw 2**
	2	3	4	5	6	7	**8**	**h2 3**
	9	10	11	12	13	14	**15**	**test 1**
	16	17	18	19	20	21	**22**	**hw 4**
Mar	23	24	25	26	27	28	**1**	**hw 5**
	2	3	4	5	6	7	**8**	**hw 6**
	9	10	11	12	13	14	**15**	**hw 7**
	16	17	18	19	**20**	21	22	**test 2**
	23	24	~~25~~	~~26~~	~~27~~	~~28~~	~~29~~	
Apr	30	31	1	2	3	4	**5**	**hw 8**
	6	7	8	9	10	11	**12**	**hw 9**
	13	14	15	16	17	18	**19**	**hw 10**
	20	21	22	23	24	25	**26**	**hw 11**
May	27	28	29	30	1	2	3	
	4	5	6	7	8	9	10	**test 3**

Homework assignments 11 and 12 are related. Both are group projects. The first is for normal credit, the second is all extra credit. Both are out of 15 points.

HW 11 (Due Apr 26)

Design and implement a very simple game. It must allow user input (e.g. via ‘cin’ or other means). General pattern I want to see is:

* There is a game state
* There is a set of valid moves the user can make
* The user makes a move
* The game state updates based on that move
* There can be 1 player or (many more)

HW 12 (Due May 3)

Design and implement an AI that either plays against the human, or plays by itself, or plays against other AIs that you write.

We will talk about AI and planning and related in future lectures. This part of the assignment is entirely optional as it is for extra credit, and it is only given to the group members who contribute to the AI.

HW 11

Assignment:	Design a *very simple* game.
Language:	Whatever you want. Really!
Purpose:	1. To ***design*** something *from scratch* that uses some of your new skills from this class. 2. To see if you can apply your skills practically.
Deliverables:	1. An archive (zip or a tarball) of code and any other resources. 2. An 'introspection' document that discusses the *what* and *why* of your approach
Who:	Group of 3, 4, or 5. Everybody gets the same grade. Even if one of you does all of the work.

Deliverable: Code

Create a zip or tarball so it unpacks all files into its own directory. The directory name should be the name of your group. There must be a file called 'group.txt' that lists the name and student ID of all team members

If you want me to be able to build and run the program, provide instructions in a README.

I'm personally going to look through these.

Deliverable: Introspection

You also need to include a file ‘introspection.txt’ in the code archive. I will read this much more carefully than the code.

Your communication skills matter. You don’t need to be William Faulkner, but it should be (more or less) grammatically correct and spell checked. Most of all, it should be coherent and articulate.

Deliverable: Introspection

Sections for the introspection.txt file (2 slides):

1. Initial ideas: enumerate the list of crazy and not-so-crazy ideas that you initially had for your game.

2. Pivots: list the spots where you changed your mind because you wanted to make something easier (or harder) on your team. Explain why.

Deliverable: Introspection

3. Data structures: What data structures did you use? Why did you choose them? What alternate structures would be appropriate? Why didn't you choose those?

4. Describe the hardest aspect of this assignment.

What Kind of Game?

Ideas:

- Pong (one player per paddle)
- Pong (P1 plays both paddles, P2 is the ball)
- Checkers
- Chess
- Tic-Tac-Toe
- Go Fish
- Civilization Clone
- Text Adventure
- Roguelike