Week 8

Kyle Dewey

Overview

- Exam #2
- Multidimensional arrays
- Command line arguments
- `void*`
- Dynamic memory allocation
- Project #3 will be released tonight

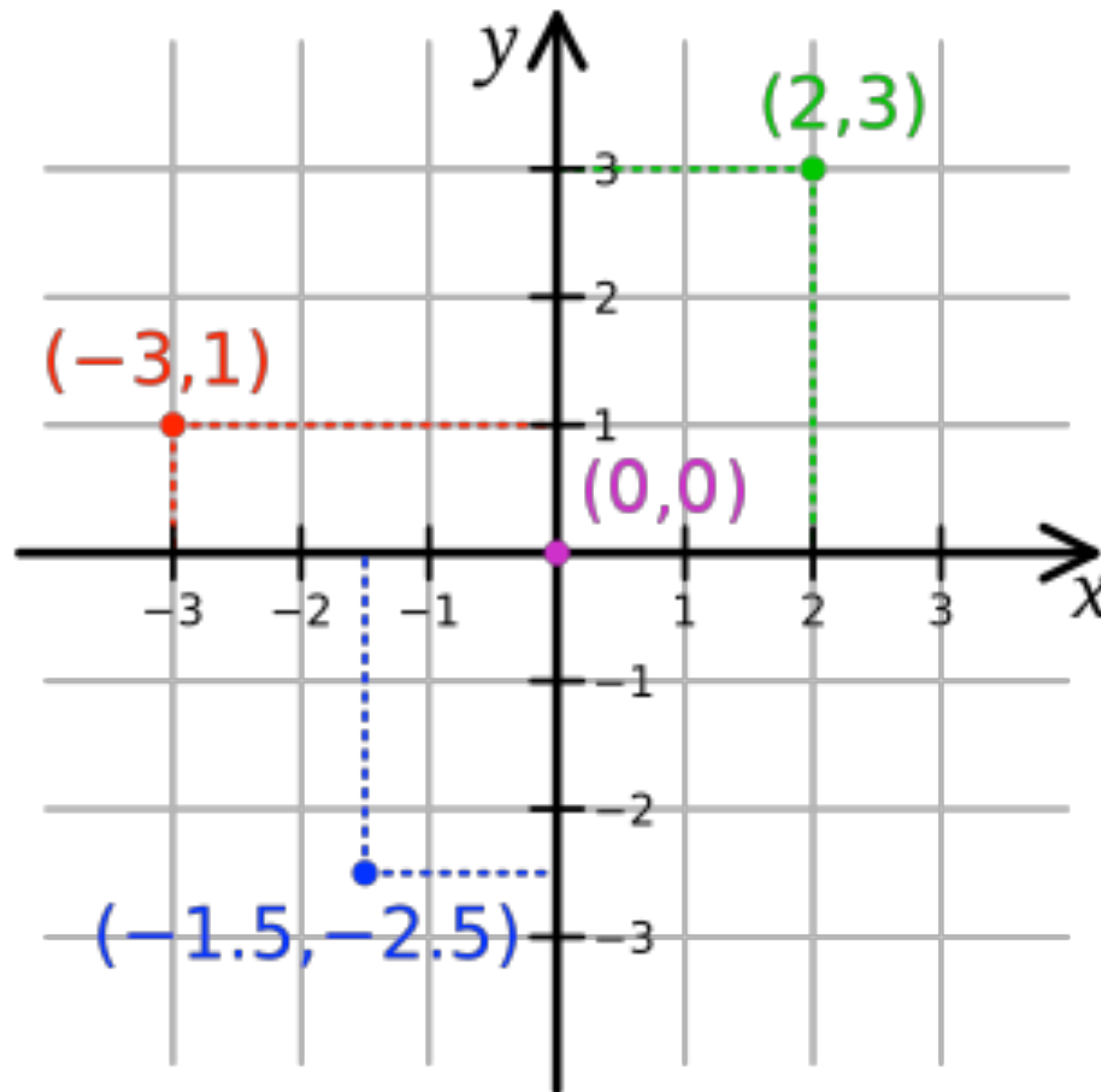
Exam #2

- Difficulty
  - Too hard? Too easy? Just right?
- Content
  - On topic? Irrelevant?
- Review
  - Helpful? Not helpful?

Multidimensional Arrays

Motivation

- Cartesian coordinate system



Coordinates

- Points can be addressed by X and Y (and possibly Z for a three dimensional grid)
  - If only one point, could have `x`, `y`, and `z` variables for that point
- There can be an arbitrary number of points

What is Wanted

```
// `grid` is a two dimensional grid
// of integers. `0` means there isn't
// a point there, and anything else
// indicates a point

grid[ 2 ][ 3 ] = 1; // put a point
grid[ 1 ][ 4 ] = 0; // remove a point

// if there is a point here
if ( grid[ 0 ][ 2 ] ) { ... }
```

What this Is

- Looks like array access...
- C lets us make arrays of arrays
  - A mechanism for representing grids

```
grid[ 2 ][ 3 ] = 1; // put a point
grid[ 1 ][ 4 ] = 0; // remove a point

// if there is a point here
if ( grid[ 0 ][ 2 ] ) { ... }
```

Multidimensional Arrays

- Can be declared like so:

```
int grid[5][5];
int grid2[2][3];
int grid3[3][2];
```

- Initial values are undefined

Multidimensional Arrays

- Can be initialized like so:

```
int grid[2][3] =
  { { 1, 2, 3 },
    { 4, 5, 6 } };

int grid2[3][2] =
  { { 7, 8 },
    { 9, 10 },
    { 11, 12 } };
```

Initialization

- Same rules as typical arrays
  - ...however, can omit only the outermost size

```
// valid
int grid[][3] =
  { { 1, 2, 3 },
    { 4, 5, 6 } };
```

Initialization

- Same rules as typical arrays
  - ...however, can omit only the outermost size

```
// invalid - omitted inner
// size
int grid[][] =
  { { 1, 2, 3 },
    { 4, 5, 6 } };
```

Initialization

- Same rules as typical arrays
  - ...however, can omit only the outermost size

```
// invalid - omitted inner
// size
int grid[2][] =
  { { 1, 2, 3 },
    { 4, 5, 6 } };
```

Usage

- Use `[]` just as with normal arrays
- Except now there is another level

```
int grid[][3] =
  { { 1, 2, 3 },
    { 4, 5, 6 } };
grid[ 0 ][ 0 ]; // gets 1
grid[ 1 ][ 1 ] = 10; // 5 is replaced
                     // with 10
```

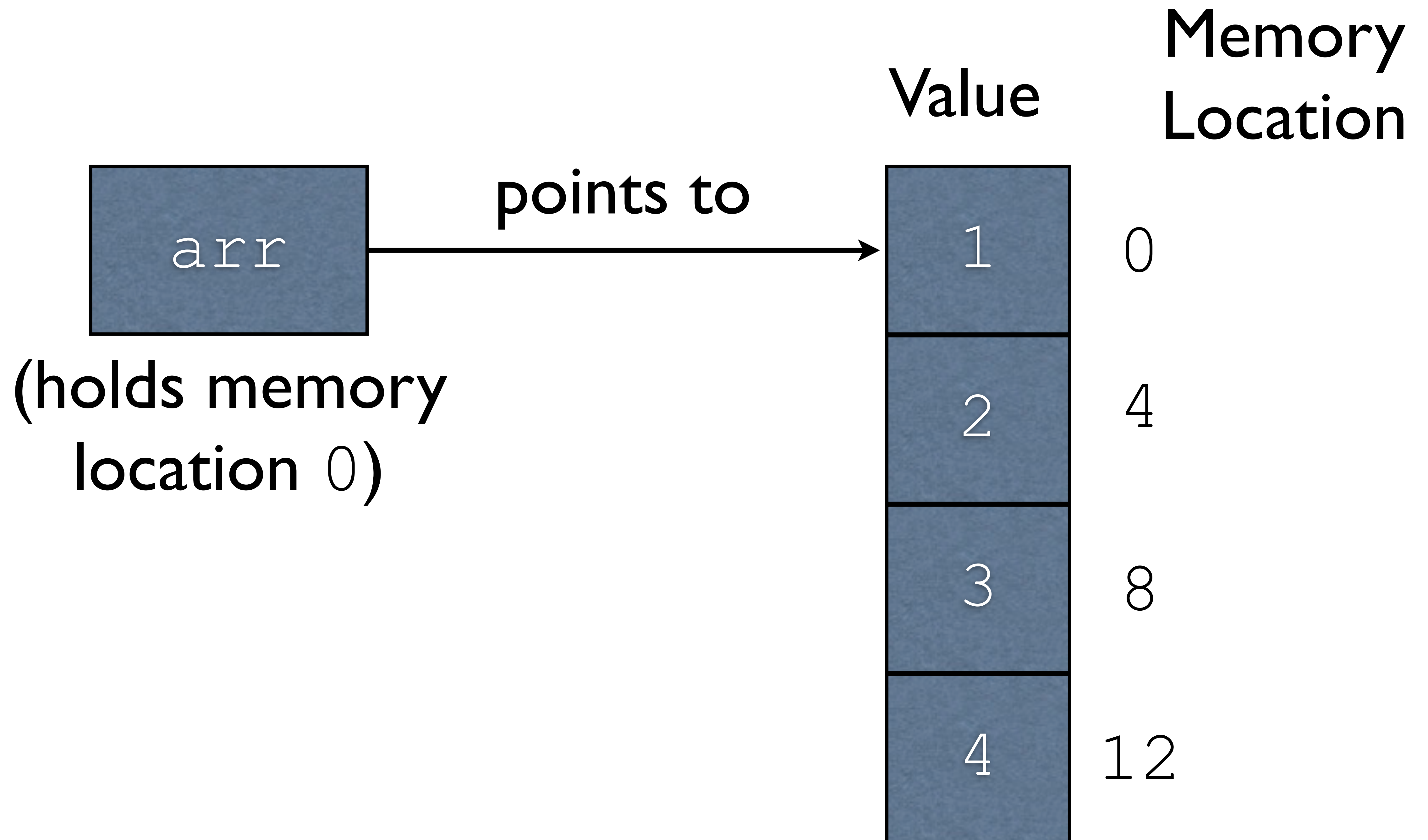
Recall Representation

- Type of a one-dimensional array of integers: `int[]`
  - Decays nicely to an `int*`

```
int arr[] = { 1, 2, 3 };
int* arr2 = arr;
```

Recall...

```
int arr[] = { 1, 2, 3, 4 };
```



Representation

- Do not decay nicely
  - `gcc`: fails to compile
  - `ch`: crashes if you try to use `arr2`

```
int arr[2][3];
int** arr2 = arr;
```

Why?

- Internal representation is flat

Logically		
0	1	2
3	4	5
6	7	8

Actually

0	1	2	3	4	5	6	7	8

Significance

- There are no actual pointers to pointers here (as `int**` would imply)
- This could be either of the two:

```
int arr[] = {0,1,2,3,4,5,6,7,8};
int arr2[][3] = {{0,1,2},
                 {3,4,5},
                 {6,7,8}};
```

0	1	2	3	4	5	6	7	8

Arrays and Functions

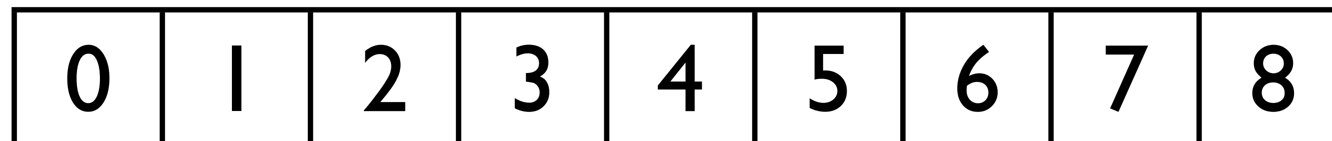
- Need to specify all but the innermost dimension
- These dimensions must be specified as constants

```
int arr2[][3] = {{0,1,2},
                 {3,4,5},
                 {6,7,8}};
void blah( int arr[][3] );
...
int main() {
  blah( arr2 );
}
```

Aside: Trick to Treat it Like a Pointer

- Recall the internal representation:

```
int arr[][3] = {{0,1,2},
                {3,4,5},
                {6,7,8}};
```




```
arr[ROW][COL] ==
((int*)arr)[ROW*COLS + COL]
```

Another Representation

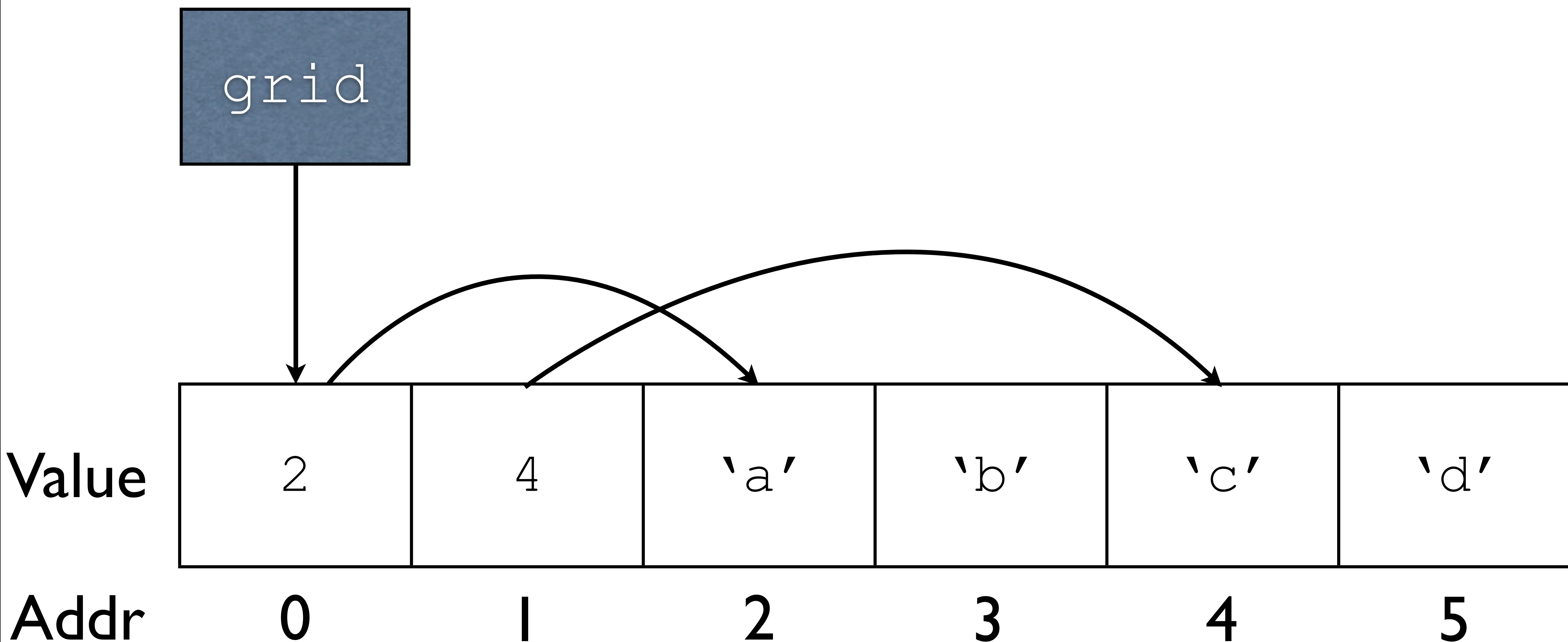
- Multidimensional arrays can be represented as arrays of pointers to arrays
- This means we have true pointers to pointers

Example

```
int firstRow[] = { 0, 1, 2 };
int secondRow[] = { 3, 4, 5 };
int thirdRow[] = { 6, 7, 8 };
int* temp[] = { firstRow,
                secondRow,
                thirdRow };
int** full = temp;
```

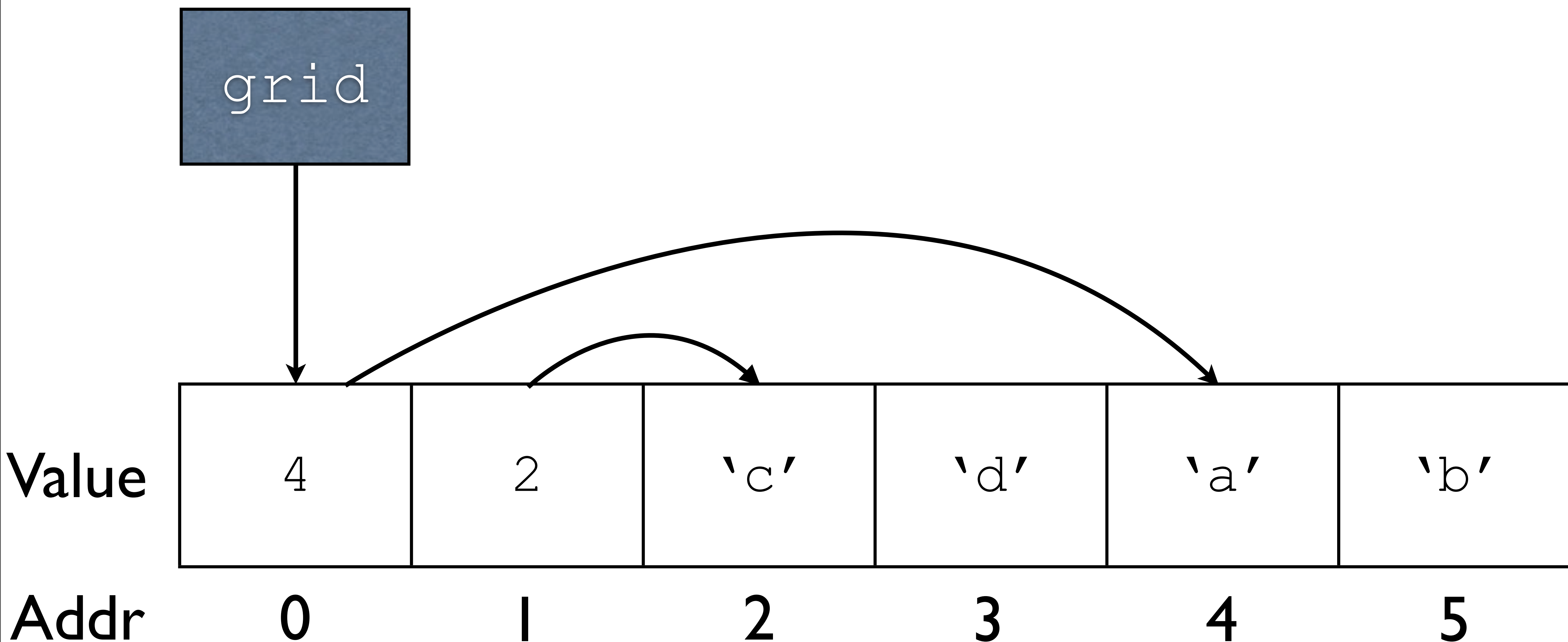
Possible Representation

```
char first[] = { 'a', 'b' };
char second[] = { 'c', 'd' };
char* grid[] = { first, second };
```



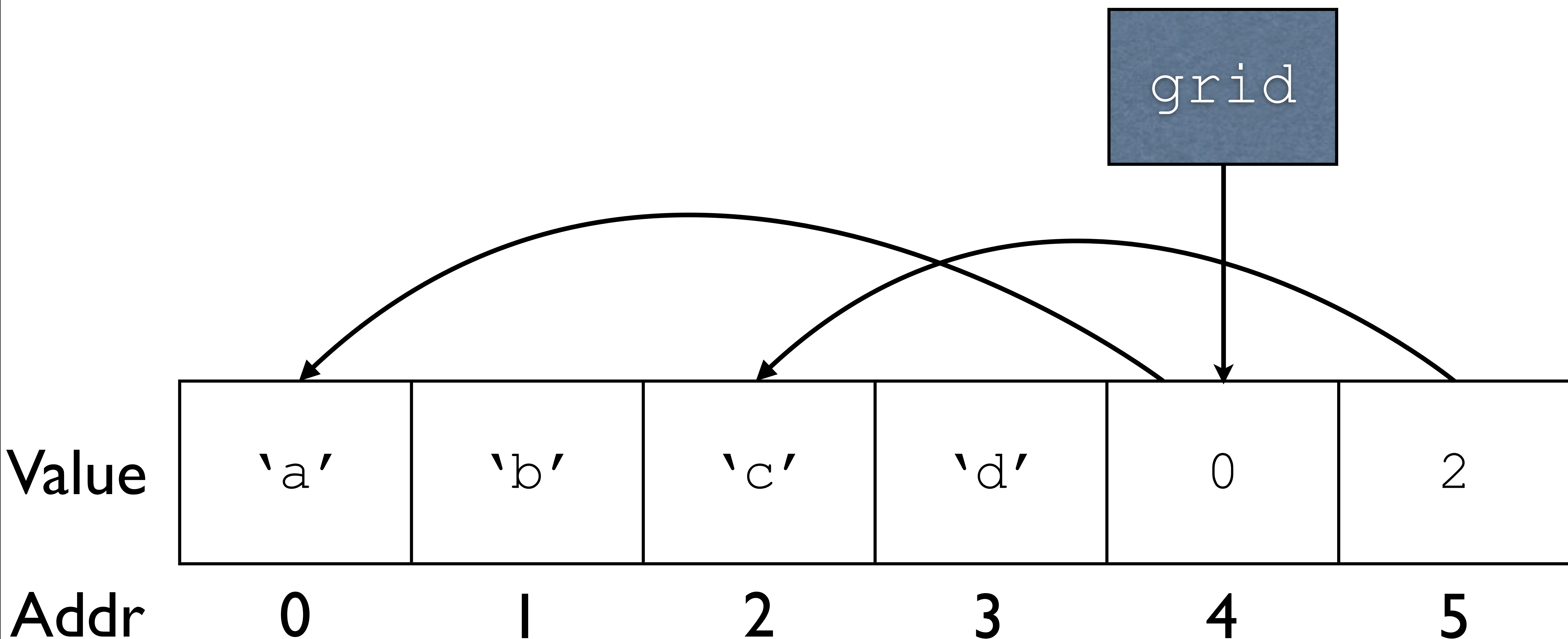
Possible Representation

```
char first[] = { 'a', 'b' };
char second[] = { 'c', 'd' };
char* grid[] = { first, second };
```



Possible Representation

```
char first[] = { 'a', 'b' };
char second[] = { 'c', 'd' };
char* grid[] = { first, second };
```



The Point

- Each individual array must be contiguous
  - Why?
- Arrays themselves can be all over in memory

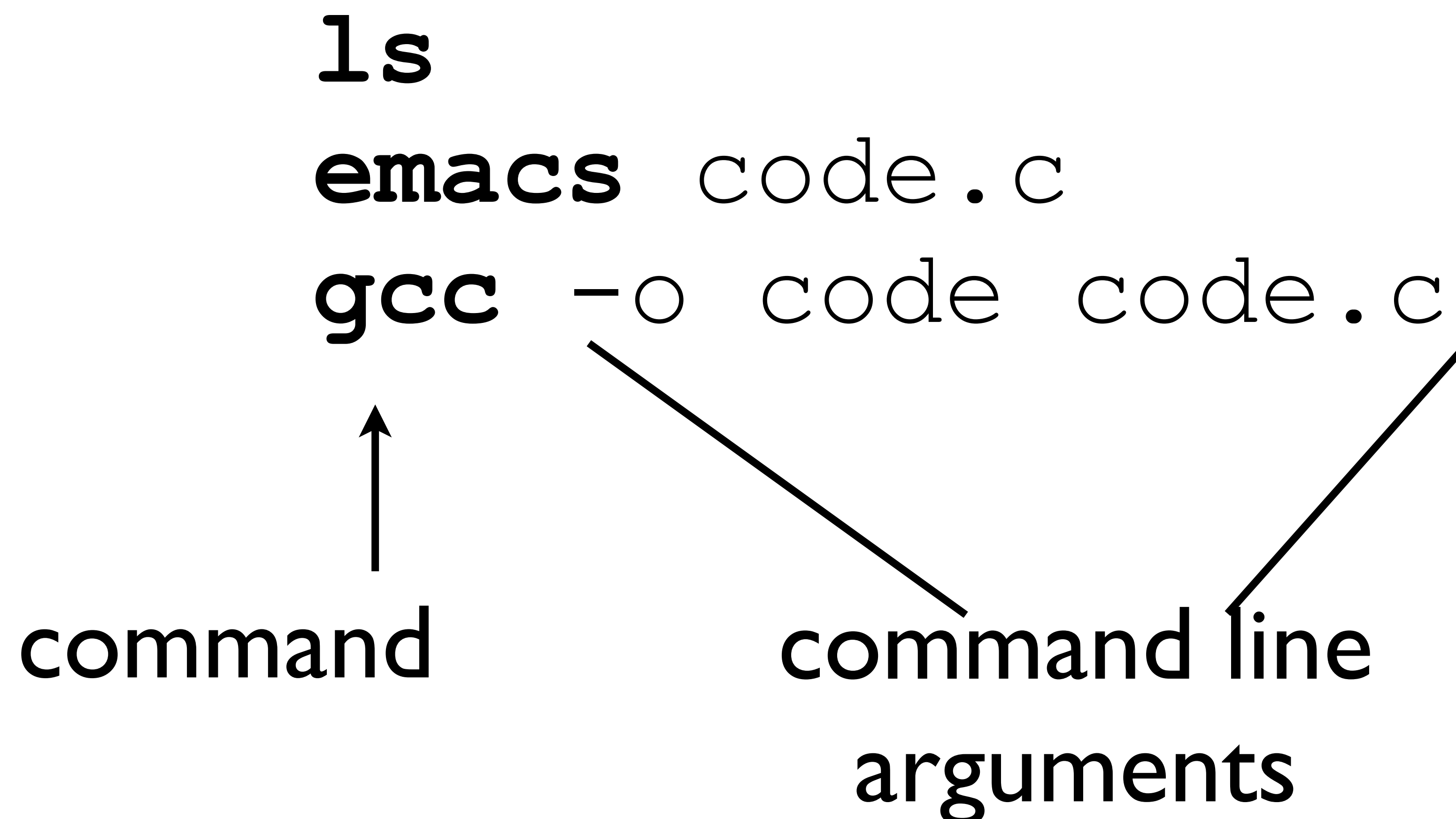
Example

- Print out a two-dimensional cartesian plane
- Represent points with periods, and everything else with spaces
- `cartesian.c`

Command Line Arguments

Command Line Arguments

- Recall how UNIX commands work



Command Line Arguments

- There is nothing special about **`ls`**, **`emacs`**, and **`gcc`**
- These are simply programs
- Any program (including your own) can take command line arguments

main()

- We usually define `main` like so:

```
int main() {
   ...
}
```

- However, it can also be defined like so:

```
int main( int argc, char** argv ) {
   ...
}
```

main()

- When defined like this, it is set up to read command line arguments
  - `argc`: The number of arguments passed
  - `argv`: The arguments themselves
- Note that `argv[ 0 ]` is always the name of the program

```
int main( int argc, char** argv ) {
   ...
}
```

command_line.c

Interpreting Arguments

- Each argument is just a string
- You'll need to interpret this however your program needs
  - i.e. use `atoi()` to convert a string to an integer, etc.

`void*`

`void*`

- Like any other pointer, it refers to some memory address
- However, it has no associated type, and cannot be dereferenced directly
  - Question: why can’t it be dereferenced?

No Dereferencing

```
void* p = 2;
*p; // get what's at p
```

Value	0x21	0x00	0x01	0x52	0xF0	0xAB	0x2C
Address	0	1	2	3	4	5	6

- `void*` is a value without context
- Without context, there is no way to know how to interpret the value (`int`? `char`? `double`?)

How to Use a `void*`

- A `void*` cannot be dereferenced directly
- However, it is possible to cast a `void*` to another type

```
char* str = "moo";
void* p = str;
printf( "%s\n", (char*)p );
```

How to Use a `void*`

- A `void*` also coerces into other pointer types

```
char* str = “moo”;
void* p = str;
char* str2 = p; // no errors
```

Caveat

- A `void*` also coerces into other pointer types
- The compiler will trust you blindly

```
char* str = “moo”;
void* p = str;

// no compiler errors, but
// this is probably not what
// is desired
int* nums = p;
```

Why a `void*`?

- Makes data structures generic (you'll have to trust me on this...)
- Can refer to some block of memory without context
  - Up next: why anyone would want to do that

Dynamic Memory Allocation

Motivation

- We want to read in a dictionary of words
- Before reading it in:
  - We don’t know how many words there are
  - We don’t know how big each word is

```
apple
banana        <<empty>>        aardvark
pear
```

Possible Solution

- Allocate the maximum amount you could ever need
- Question: why is this generally not a good solution? (2 reasons)

```
// 1000 words max with
// 100 characters max per word
char dictionary[1000][100];
```

Problems

- Most things do not have a good “maximum” you can get a grasp of
- Your program always needs the maximum amount of memory, and usually the vast majority is completely wasted

What is Desired

- A way to tell the computer to give a certain amount of memory to a program as it runs
- Only what is explicitly requested is allocated

Dynamic Memory Allocation

- Dynamic: as the program runs
- Memory allocation: set aside memory

malloc

- The most generic way to allocate memory
- Takes the number of bytes to allocate
- Returns a `void*` to the block of memory allocated

```
// size_t is an integral defined
// elsewhere
void* malloc( size_t numBytes );
```

Using `malloc`

- The `sizeof` operator comes in handy
  - Returns an integral size as a `size_t`
- For example: allocate room for 50 integers dynamically:

```
// dynamically
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );

int nums2[ 50 ]; // statically
```

Importance

- Static allocation can only be done with constants
- Dynamic allocation can be done with variables

```
int numToAllocate;
scanf( “%i”, &numToAllocate );
int* nums =
  malloc(sizeof( int ) * numToAllocate);
int nums2[ numToAllocate ]; // ERROR
```

Memory Contents

- The contents of the memory allocated by malloc is undefined
- You will need to initialize it yourself with a loop (or by using `memset`)

malloc1.c,
malloc2.c

calloc

- Very similar to malloc
- Takes the number of elements to allocate and the size of each element
  - Will do the multiplication itself
- Will also initialize the allocated portion to zero at the binary representation

```
void* calloc( size_t num, size_t size );
```

calloc

- Very similar to malloc
- Will also initialize the allocated portion to zero at the binary representation

```
int* nums1, nums2;
nums1 = malloc( sizeof( int ) * 50 );
nums2 = calloc( 50, sizeof( int ) );
```

`realloc`

- For resizing a block of memory that has already been allocated (with one of `malloc`, `calloc`, or `realloc`)
  - Except if given `NULL` - then it behaves like `malloc`
- Takes the previously allocated memory block, and the new size

```
void* realloc( void* ptr, size_t size );
```

realloc

- For resizing a block of memory that has already been allocated (with one of `malloc`, `calloc`, or `realloc`)
  - Except if given `NULL` - then it behaves like `malloc`

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// we want 5 more integers
nums = realloc( nums,
                sizeof( int ) * 55 );
```

realloc

- For resizing a block of memory that has already been allocated (with one of `malloc`, `calloc`, or `realloc`)

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// we want 5 more integers
nums = realloc( nums,
                sizeof( int ) * 55 );
```

free

- Once we are done using a block of memory, call free on it
- If a block is never freed, it is called a **memory leak**
  - Memory is still allocated but wasted

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// done with nums
free( nums );
```

Project #3

Basic Idea

- We have a dictionary of words
- We are given an unordered series of letters
  - i.e. the ordering does not matter
- Using these letters, which words in the dictionary can be made?

Basic Idea

Dictionary:

```
moo
cow
bull
steer
```

Letters:

```
mlublorts
```

Basic Idea

Dictionary:

`moo`

`cow`

`bull`

`steer`

Letters:

`m`**`lubl`**`orts`

Words:

`bull`

For Full Credit (4% of Final Grade)

- Dictionary is provided and is hardcoded
- Prompt user for some letters
- Print out which words in the given dictionary can be made
- Keep doing this until the user uses "exit" for the letters

Bonuses

- Six bonuses
- Can add 6% to your final grade
- Must be done in sequence if they are to be done
  - I.e. you cannot get credit for bonus #3 without doing bonus #2
- Don't need to do them all (or any)

Bonus #1 (2%)

- Read in the dictionary from a file named "`dictionary.txt`"
- Safe to assume the max number of words and the max word length are set constants
- Format:

```
moo
cow
bull
steer
```

Bonus #2 (0.5%)

- The maximum word length is not constant

Bonus #3 (0.5%)

- The maximum number of words in the dictionary is not constant

Bonus #4 (0.5%)

- Instead of prompting the user for input letters, read in the input from a file named "`input.txt`"
- Format:

```
bjkbkj
fbjb
wyuil
```

Bonus #5 (0.5%)

- Write the results to a file named "`output.txt`"
- Output format:

```
floom:
 moo
hwravwt:
 hat
 vat
m:
```

Bonus #6 (2%)

- Read in the input file, dictionary file, and output file from the command line
- Command line format:

```
./prog -d dict.txt -i in.txt -o out.txt
```

Bonus #6 (continued)

- The `-d`, `-i`, and `-o` parts can be in any order, or omitted
  - If no `-d` is provided, default to "`dictionary.txt`"
  - If no `-i` is provided, instead prompt the user for letters interactively
  - If no `-o` is provided, instead print out the results to the user