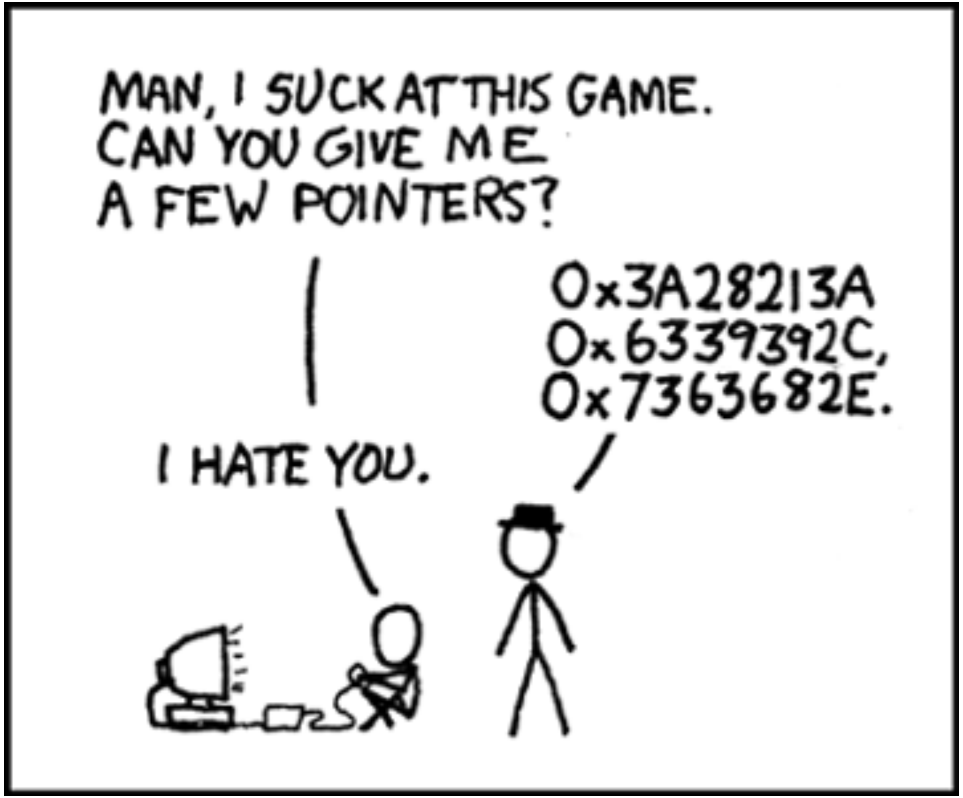
Pointers

Chapter 10



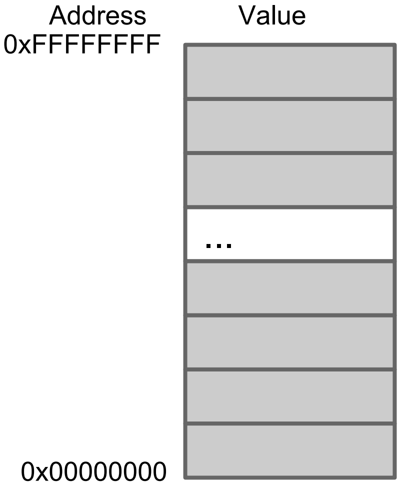
http://xkcd.com/138/

Memory

- To understand pointers, it helps to have an understanding of memory
- How is memory structured?

Memory

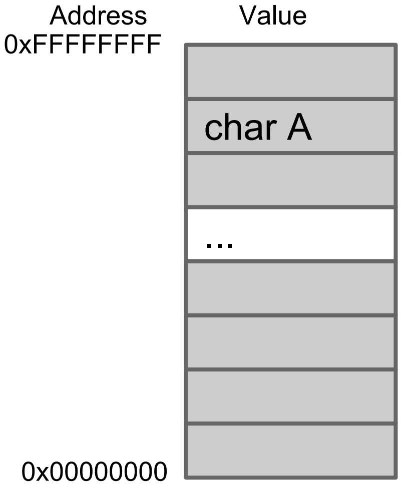
- Memory is structured similar to an array



Memory

- Every variable is assigned a memory location

char A;



Address

Value

0xFFFFFFFF

char A

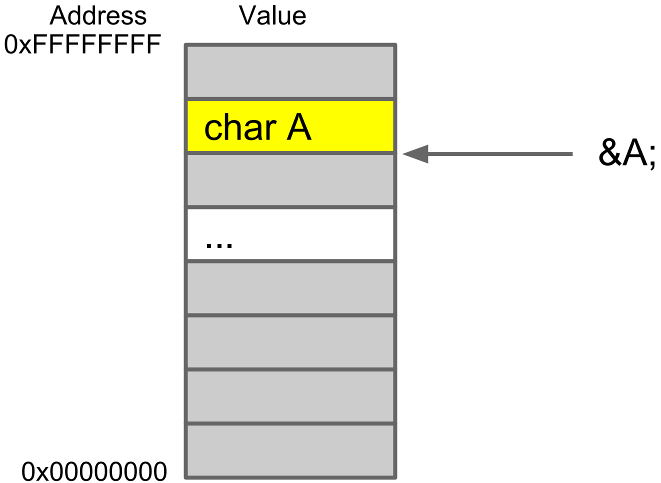
...

0x00000000

Memory

- The address can be retrieved using the address operator ‘&’

```
char A;
```



Address

Value

0xFFFFFFFF

char A

...

&A;

0x00000000

Memory

- The size of a variable in memory is determined by the system and type of variable
    - char - 1 byte
    - int - 1 , 2, 4, or 8 bytes
        - int8_t
        - int16_t
        - int32_t
        - int64_t

Memory

- The size of a variable in memory is determined by the system and type of variable
  - float - usually 2 or 4 bytes
  - double - usually 4 or 8 bytes
  - ...

Memory

- Regardless of the variable size, the address operator will always give you the beginning of the memory array

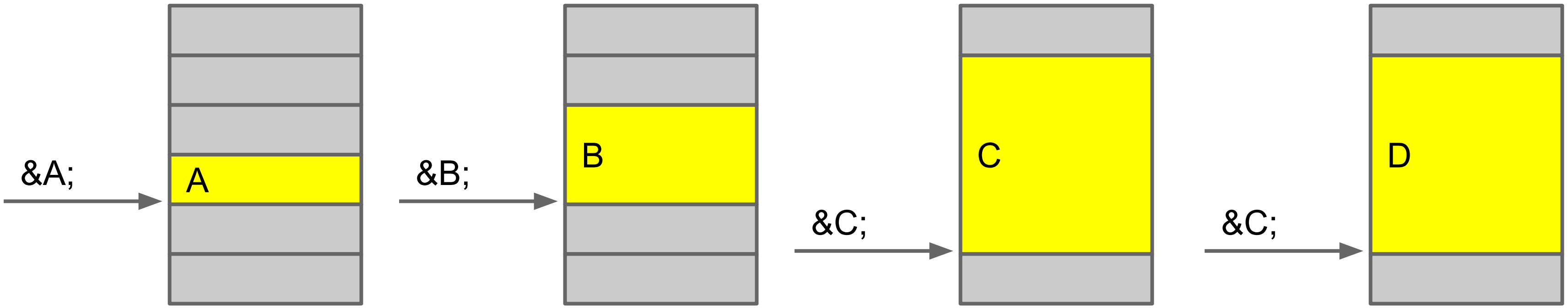
Memory

char A;

int16_t B;

double C;

uint32_t D;



Pointers Variables

- Pointer variables are similar to normal variables, except they hold memory locations (addresses)

Pointers Variables

- Pointer variables are similar to normal variables, except they hold memory locations (addresses)

```
int* pMyPointer;
```

Pointers Variables

- Pointer variables are similar to normal variables, except they hold memory locations (addresses)

```
int* pMyPointer;
```

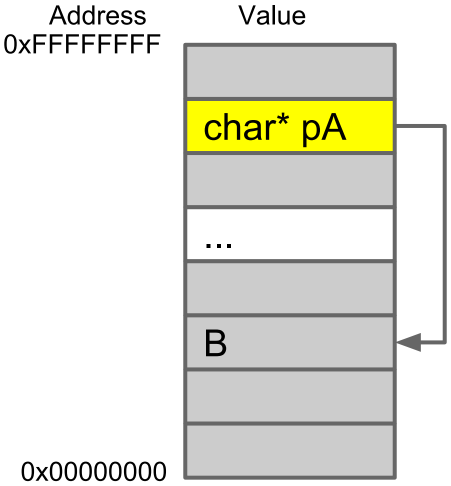
Notice the order here. It does not matter whether you declare:

```
int* pMyPointer; // I prefer this
int * pMyPointer;
int *pMyPointer;
```

Pointer Variable

- In memory, they look a bit different
  - A pointer variable points at another section of memory

char* pA;



Pointer Variable

- Can we use the pointer variable to access what it points at?

Pointer Variable

- Can we use the pointer variable to access what it points at?

- if pA == &b, then *pA == b

- This is called ‘dereferencing’ pA
    - *pA

Arrays and Pointers

- Arrays and pointers are very similiar
    - An array without the type and brackets acts just like a pointer

```
byte luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}
```

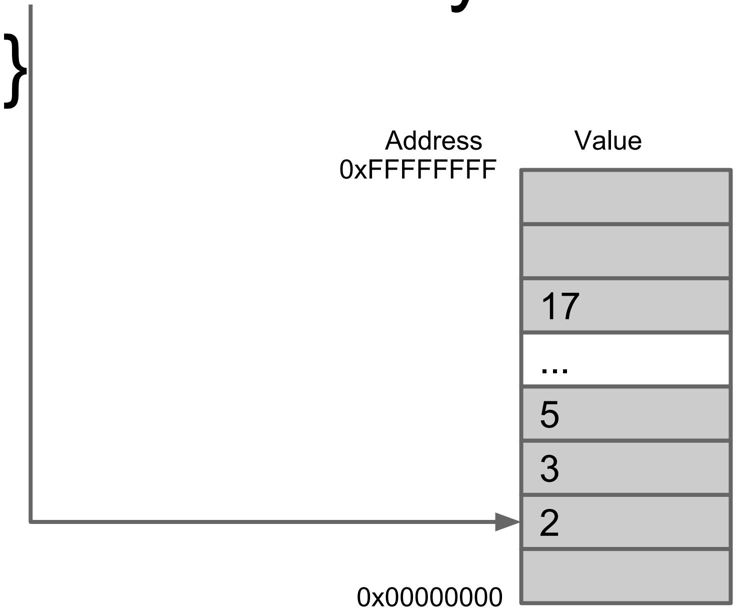
Arrays and Pointers

- Arrays and pointers are very similiar
  - An array without the type and brackets acts just like a pointer
  - The array name points at the first item in the list

byte luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}

Arrays and Pointers

unsigned char luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}



Pointer Arithmetic

- We can use some mathematical operations to move the pointer around

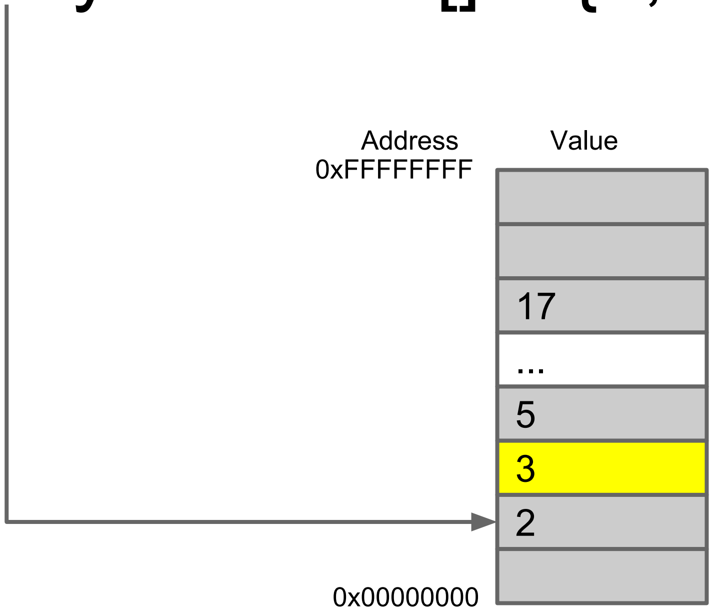
Pointer Arithmetic

- We can use some mathematical operations to move the pointer around

- For example: What if we wanted to access the 2nd element in our luckyNumber array?

Pointer Arithmetic

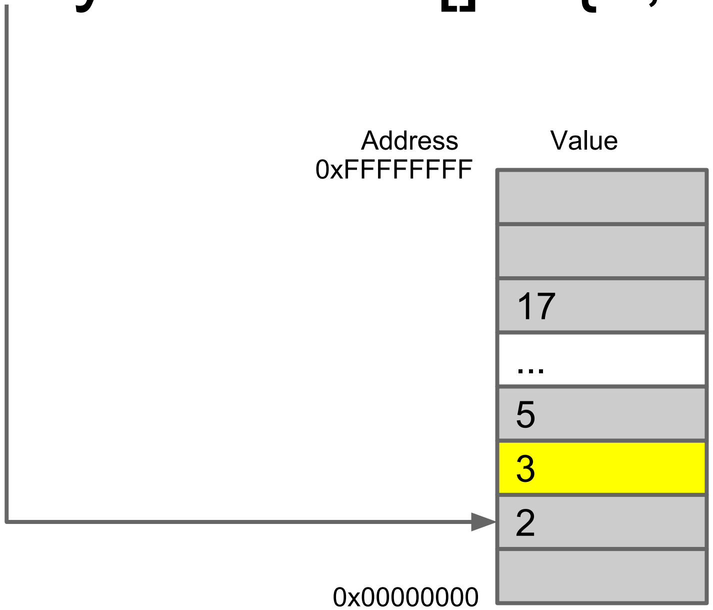
byte luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}



We want to point at 3

Pointer Arithmetic

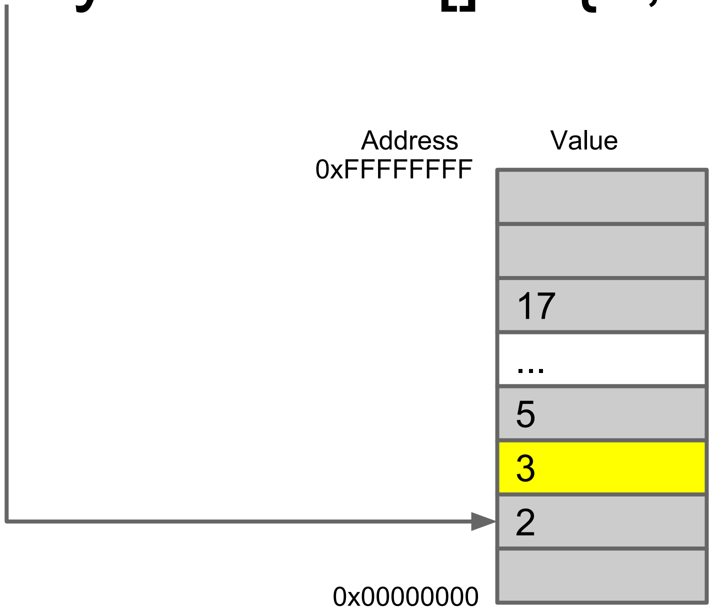
byte luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}



We want to point at 3

Pointer Arithmetic

byte luckyNumbers[] = {2, 3, 5, 7, 11, 13, 17}



Address

Value

0xFFFFFFFF

17

...

5

3

2

0x00000000

We want to point at 3 --->
*(luckyNumbers + 1)

Pointer Arithmetic

- During pointer arithmetic, the pointer moves according to the size of the variable type

- Check out ‘sizeof()’

Pointer Arithmetic

- What if we had an array of integers and wanted to access the 3rd element?

Pointer Arithmetic

- What if we had an array of integers and wanted to access the 3rd element?
  - Array of floats?
  - Array of doubles?
  - …

- Example

Pointer Initialization

- There is a special value in memory, where no variable can or should be stored

Pointer Initialization

- How would be initialize a pointer variable to another variables address?

Pointer Initialization

- There is a special value in memory, where no variable can or should be stored
    - NULL or 0

Pointer Comparison

- Comparing pointers can be tricky

Pointer Comparison

- Comparing pointers can be tricky
  - Comparing two pointers is not the same as comparing the values the pointers point at

Pointer Comparison

- Comparing pointers can be tricky
  - Comparing two pointers is not the same as comparing the values the pointers point at

- Example

Pointers in Functions and Methods

- Variable pointers can be used as function and method arguments

Pointers in Functions and Methods

- Variable pointers can be used as function and method arguments

```
void double(int* value);
void foo(double* bar);
```

Pointers in Functions and Methods

- Once inside of the function or method, the pointer acts like any other pointer

Pointers in Functions and Methods

- Once inside of the function or method, the pointer acts like any other pointer

```
void foo(int* value)
{
      *value += 3;
}
```

Pointers in Functions and Methods

- Pointers can also be returned from a function

Pointers in Functions and Methods

- Pointers can also be returned from a function

```
int* findValue(int position);
double* foo(int bar);
```

- More on this when we cover Dynamic Memory

Pointers and Constants

- Pointers can be used with constants, the syntax changes a little

Pointers and Constants

- Pointers can be used to point at constant items

Pointers and Constants

- Pointers can be used to point at constant items

```
const int myNumbers[] = {2, 3, 5, 7, 11, 13, 17};
const int* pMyPointer = myNumbers;
```

Pointers and Constants

- Pointers can be used to point at constant variable

```
const int myNumbers[] = {2, 3, 5, 7, 11, 13, 17};
const int* pMyPointer = myNumbers;
```

- With a pointer to a constant variable, the value that the pointer points at may not be changed

Pointers and Constants

- Pointers to constants are often used in methods or functions

Pointers and Constants

- Pointers to constants are often used in methods or functions

```
void searchArray(const int* pArray, int size);
```

Pointers and Constants

- But what if we don’t want the pointer to change?

Pointers and Constants

- But what if we don’t want the pointer to change?

```
int* const pPointer = &value;
```

Pointers and Constants

- But what if we don’t want the pointer to change?

```
int* const pPointer = &value;
```

- This prevents pPointer from being pointed at anything else, but doesn not prevent the changing of what is at pPointer

Pointers and Constants

- What about a constant pointer to a constant variable?

Pointers and Constants

- What about a constant pointer to a constant variable?

```
const int* const pPointer = &value;
```

Dynamic Memory Allocation

- So far, most of you allocated static arrays
    - int aArray[100];
- What if you don’t know the size of the array before compilation?
    - Set during execution

Dynamic Memory Allocation

- So far, most of you allocated static arrays
    - int aArray[100];

- What if you don’t know the size of the array before compilation?
    - Set during execution
- What if memory needs to be created and deleted during exectution?

Dynamic Memory Allocation

- Memory that is allocated during execution is called Dynamic Memory
  - It is only possible through pointers

Dynamic Memory Allocation

- Memory that is allocated during execution is called Dynamic Memory
    - It is only possible through pointers

- Welcome to the ‘new’ way of doing things

Dynamic Memory Allocation

- When dealing with Dynamic Memory, there are two new operators to deal with
    - new
    - delete

Dynamic Memory Allocation

- The ‘new’ operator is used to request new memory

```
int* pMyInt;
pMyInt = new int;
```

Dynamic Memory Allocation

- The ‘new’ operator is used to request new memory

```
int* pMyInt;
pMyInt = new int;
```

- This operation allocates memory for holding an integer
  - This memory is only accessible through the pointer

Dynamic Memory Allocation

- The ‘new’ operator can also be used to allocate arrays of things

```
int* pArrayPointer = new int[4];
```

- This gives the ability to dynamically (re)size arrays

Dynamic Memory Allocation

- The ‘delete’ operator is used to deallocate memory that is no longer needed

```
int* pMyInt;
pMyInt = new int;
…
delete pMyInt;
pMyInt = NULL;
```

Dynamic Memory Allocation

- The ‘delete’ operator is used to deallocate memory that is no longer needed

```
int* pMyInt;
pMyInt = new int;
…
delete pMyInt;
pMyInt = NULL;
```

It is good practice to set deleted pointers to NULL

Dynamic Memory Allocation

- There is a special case when deleting dynamic arrays
  - You should use the [ ] after delete

```
int* pMyInt = new int[4];
…
delete [] pMyInt;
pMyInt = NULL;
```

Dangling Pointers and Memory Leaks

- The ‘new’ and ‘delete’ operations are very useful, but they are potentially dangerous

Dangling Pointers and Memory Leaks

- Dangling Pointer
  - The dynamic memory is deleted, but the pointer still points at the old location

Dangling Pointers and Memory Leaks

- Dangling Pointer
  - The dynamic memory is deleted, but the pointer still points at the old location
- Memory Leak
  - The memory is dynamically allocated but never deleted

Dangling Pointers and Memory Leaks

- Dangling Pointer
  - The dynamic memory is deleted, but the pointer still points at the old location
    - Potential problem?
- Memory Leak
  - The memory is dynamically allocated but never deleted
    - Potential problem?

Returning Pointers

- Now that we have dynamic memory, we can return pointer from functions

Returning Pointers

- Now that we have dynamic memory, we can return pointer from functions
    - Why would this be useful?

Pointers to Class Objects and Structures

- Pointers don’t just point at the primitive types like int, float, or char

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

```
MyClass myClass;
MyClass* pPointer = &myClass;
```

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

```
MyClass myClass;
MyClass* pPointer = &myClass;
```

- How can we access the class methods using abilities we already know?

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

```
MyClass myClass;
MyClass* pPointer = &myClass;
```

- How can we access the class methods using abilities we already know?

```
(*pPointer).foo();
```

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

```
MyClass myClass;
MyClass* pPointer = &myClass;
```

- How can we access the class methods using abilities we already know?

```
(*pPointer).foo();
```

- Why not *pPointer.foo()?

Pointers to Class Objects and Structures

- Pointers to class objects and structures work much like pointers to any other type

```
MyClass myClass;
MyClass* pPointer = &myClass;
```

- How can we access the class methods using abilities we already know?

```
(*pPointer).foo();
```

“Theres gotta be a better way”

- Why not *pPointer.foo()?

Pointers to Class Objects and Structures

“And there is Kevin!”

- The structure pointer operator offers us a better solution

Pointers to Class Objects and Structures

“And there is Kevin!”

- The structure pointer operator offers us a better solution

```
MyClass myClass;
MyClass* pPointer = &myClass;
pPointer->foo();
```

Pointers to Class Objects and Structures

“And there is Kevin!”

- The structure pointer operator offers us a better solution

```
MyClass myClass;
MyClass* pPointer = &myClass;
pPointer->foo();
```

- Caution must be taken when using this operator because it forces a dereference of the pointer

Dynamic Class Allocation

- Using dynamic allocation, classes and structs can also be created at runtime

Dynamic Class Allocation

- Using dynamic allocation, classes and structs can also be created at runtime

```
MyClass* pPointer = new MyClass();
```

Dynamic Class Allocation

- Using dynamic allocation, classes and structs can also be created at runtime

```
MyClass* pPointer = new MyClass();
```

- This method can be used to pass in arguments during construction as well

Good Practice

- When using dynamic memory in classes, it is good practice to delete the memory in the destructor

Good Practice

- When using dynamic memory in classes, it is good practice to delete the memory in the destructor
    - How will we know if its been deleted already?

Class Pointers as Function Parameters

- Classes can also be passed to methods or functions as pointers

Class Pointers as Function Parameters

- Classes can also be passed to methods or functions as pointers

```
void foo(MyClass* pClass)
{ /* do something*/}

MyClass nonPointer;
MyClass* pPointer = new MyClass();

foo(&nonPointer);
foo(pPointer);
```

Double Pointers

- Since pointers can point at objects, can pointers point at pointers?

Double Pointers

- Since pointers can point at objects, can pointers point at pointers?
    - Yes, with double pointers

Double Pointers

- Since pointers can point at objects, can pointers point at pointers?
    - Yes, with double pointers

```
int value;
int* pPointer1 = &value;
int** pPointer2 = &pPointer1;
```

Double Pointers

- Since pointers can point at objects, can pointers point at pointers?
  - Yes, with double pointers

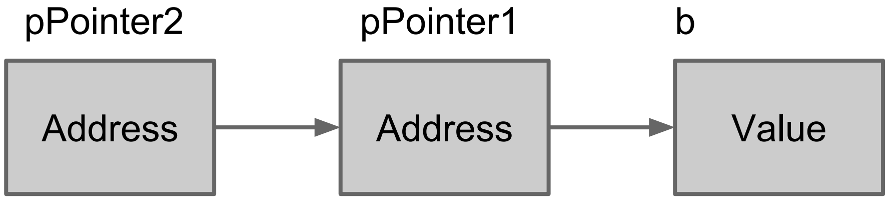
```
int value;
int* pPointer1 = &value;
int** pPointer2 = &pPointer1;
```

- What is pPointer2 actually pointing at?

Double Pointers

- Since pointers can point at objects, can pointers point at pointers?
    - Yes, with double pointers

```
int b;
int* pPointer1 = &b;
int** pPointer2 = &pPointer1;
```



- What is pPointer2 actually pointing at?

Double Pointers

- We can even dereference pPointer2 to get the value

Double Pointers

- We can even dereference pPointer2 to get the value

```
int b;
int* pPointer1 = &b;
int** pPointer2 = &pPointer1;

cout << “The value is “ << **pPointer << endl;
```

Double Pointers

- We can even dereference pPointer2 to get the value

```
int b;
int* pPointer1 = &b;
int** pPointer2 = &pPointer1;

cout << “The value is “ << **pPointer << endl;
```

Could this go on a while?

Double Pointers

- This just seems confusing, what good is it?
    - What about dynamically allocating 2 dimensional arrays

Double Pointers

- How would you declare a NON dynamically allocated 2d array?

Double Pointers

- How would you declare a NON dynamically allocated 2d array?

```
int my2dArray[10][10];
```

Double Pointers

- The dynamic allocation of a 2d array is a bit more difficult
  - We start by declaring the pointer

```
int** pTwoDimArray = NULL;
```

Double Pointers

- The dynamic allocation of a 2d array is a bit more difficult
  - We start by declaring the pointer
  - Next we allocate enough room for all of the pointers

```
int** pTwoDimArray = NULL;
pToDimArray = new int*[n];
```

Double Pointers

- The dynamic allocation of a 2d array is a bit more difficult
  - We start by declaring the pointer
  - Next we allocate enough room for all of the pointers
  - Finally, we allocate the object for each pointer

```
int** pTwoDimArray = NULL;
pToDimArray = new int*[n];
for (int i = 0; i < n; i++)
        pToDimArray[i] = new int[m];
```

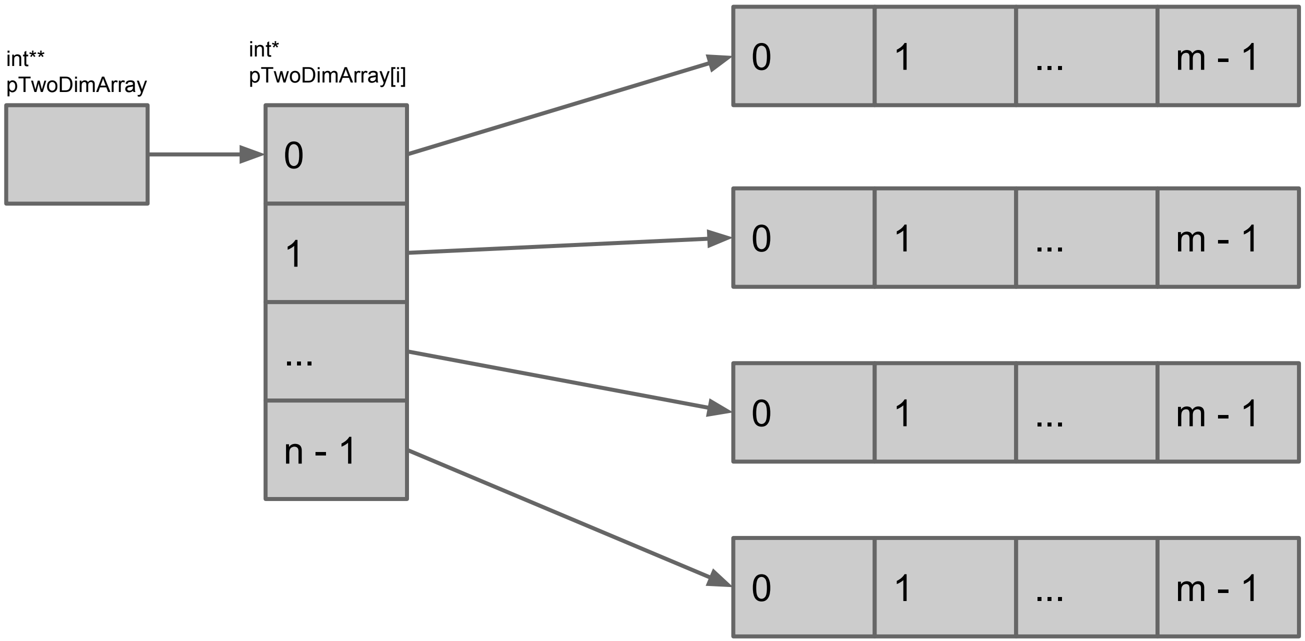
Double Pointers

- The dynamic allocation of a 2d array is a bit more difficult
  - We start by declaring the pointer
  - Next we allocate enough room for all of the pointers
  - Finally, we allocate the object for each pointer

```
int** pTwoDimArray = NULL;
pToDimArray = new int*[n];
for (int i = 0; i < n; i++)
        pToDimArray[i] = new int[m];
```

What is this even doing????

Double Pointers


int**
pTwoDimArray
int*
pTwoDimArray[i]
0
1
...
n - 1
0
1
...
m - 1
0
1
...
m - 1
0
1
...
m - 1
0
1
...
m - 1