ARRAYS & POINTERS

Project One

- You should be working on this, if you're not already
- Due Friday, midnight-ish
- Any questions on this?

review:

```
#include <iostream>

int main()
{
   int a = 10, b = 15;
   swap( a, b );
   return EXIT_SUCCESS;
}

void swap( int a, int b )
{
   int temp = a;
   a = b;
   b = temp;
}
```

What does this need to work?

More review

- How do we set up default parameters?
- How do we set up function overloading?
- What are the pros and cons of either/both of these things?

The Problem:

- What if we wanted to store the first 8 elements of the Fibonacci sequence? (1,1,2,3,5,8,13,21)
- You could use variables, but that would be clumsy...

```
int fib1 = 1;     // not good
int fib2 = 1;
...
int fib8 = 21;
```

- Also, you'd have to declare all the required variables at compile time - what if we needed 100? 1000?

Arrays: a solution

- Data structure built into C++
- Arrays are a consecutive group of memory locations that have the same type, and are all referred to by the same name
  - i.e., 10 integers in a row, all referred to by the same name - `listOfGrades`
- Think of a list in everyday life - except each element in the list has the same type

Declaring Arrays

in general:

```
type arrayName[arraySize];
```

expression that can be evaluated at compile time

example:

```
int listOfNums[5];
```

example with initialization:

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

- What are the initial values of these?
- Size of the array has to be determined at compile time and can't be changed later (sort of)

Array Indices

- What is an array index? (starts at **0**, not **1**!)
- Using the array name, along with the array index, an array location can be treated just like a variable:

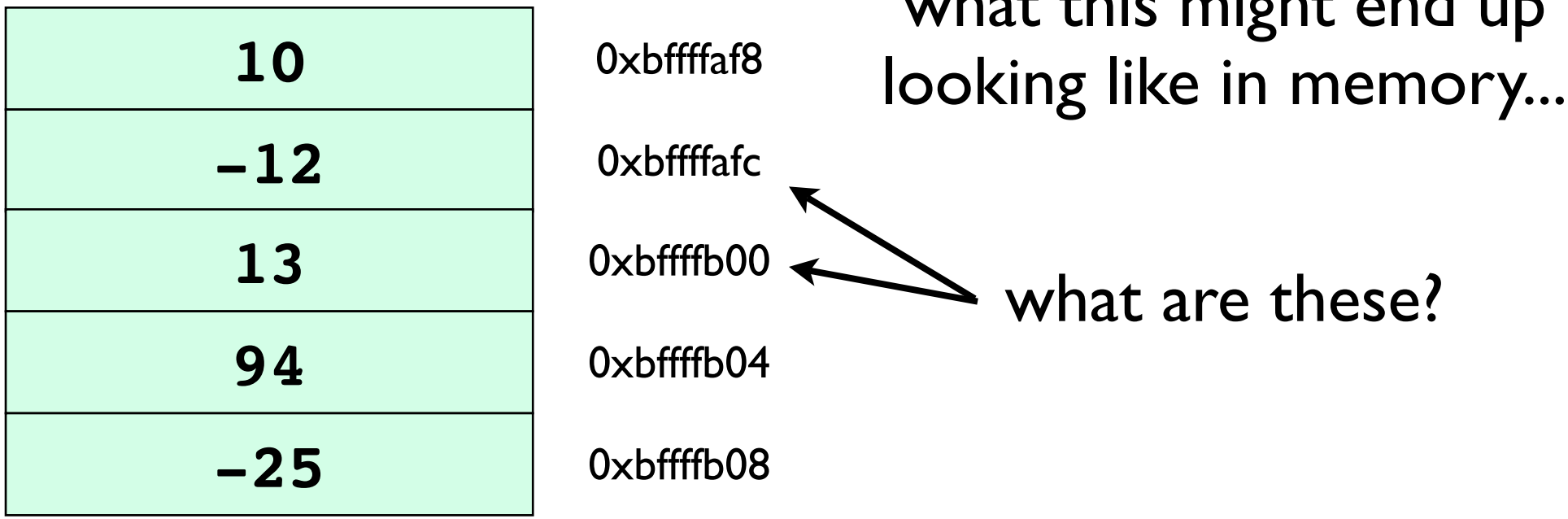
```
int testArray[10];

// writing into an array
testArray[5] = 234;

// reading from an array
cout << testArray[3*2] << endl;
```

- Example with a for loop...

Array Storage

- The elements of an array are stored *consecutively* in memory

```
int listOfNums[5] = {10,-2,13,94,-25};
```



How Arrays Work

- To figure out how to access an array element, the compiler/program needs:
  - the base address of the array in memory
  - the index of the element
  - the size of the data type in bytes

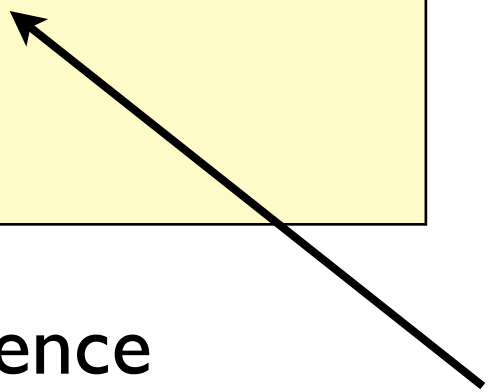
element address = base address + (data size * index)

- This works because arrays are stored contiguously
- First element of an array is at **0**, not **1** !

Passing Arrays to Functions

- To pass an array to a function, you use this notation:

```
int sum( int list[] )
{
}
```



square brackets indicate that this is an array

- Are arrays passed by reference or by value?
- Let's write this function...

Another example

- Let's write a function to determine and return the biggest and smallest value in an array of floats.

(float) 

More about Arrays

- Arrays are passed by reference, *and here's why*:
  - What is actually getting passed is the *address* of the beginning of the chunk of memory - the array's first value
- Can we make copies of an array like this? Why or why not?

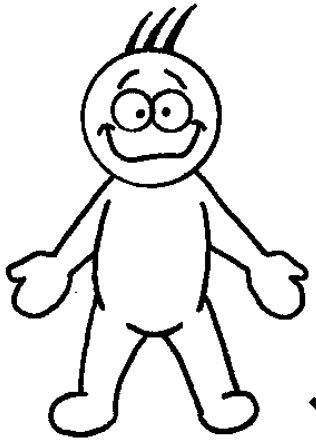
```
int arrayOne[5] = {1,2,3,4,5};
int arrayTwo[5];

arrayTwo = arrayOne;
```

Multidimensional Arrays

- You can declare arrays with as many dimensions as you want
- All elements still are the same type, though

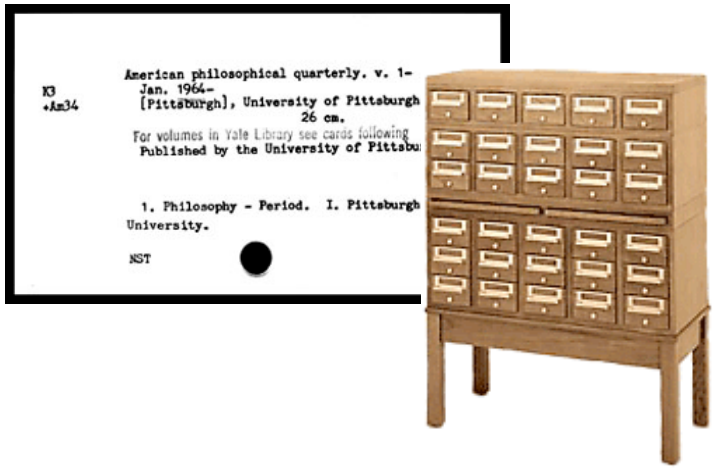
```
// declaring
int array[2][2] = { {1,2}, {3,4} };

// using
cout << array[0][0] << endl;
cout << array[1][1] << endl;
```

POINTERS!!!



(direct access)

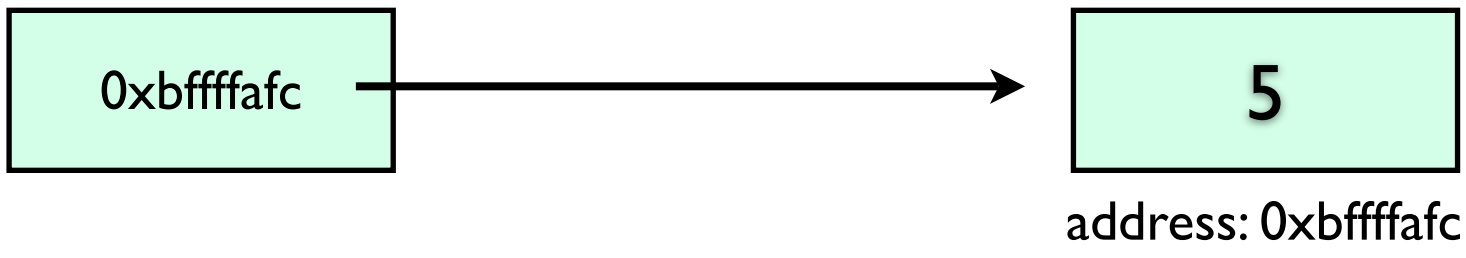
(access via card)



POINTERS!!!

- Pointers are one of the most powerful (and tricky) features of C/C++
- A **pointer** is a kind of variable that contains a memory location as its value
  - The pointer is "pointing" to whatever is in that memory location

```
int count = 5;
int* countPtr = &count;
```



Pointer Anatomy

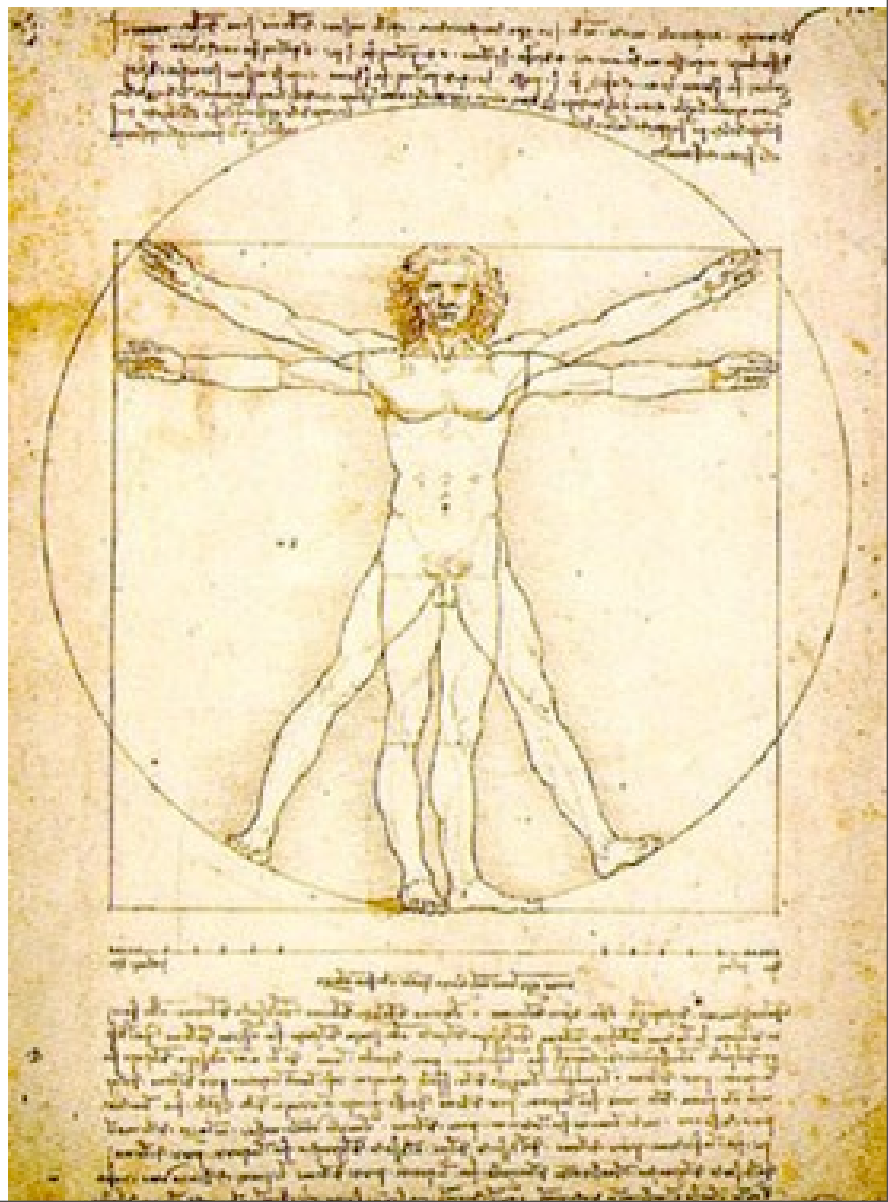
```
int *pointer = NULL;
```

either make the pointer point somewhere, or assign NULL so it doesn't point somewhere unintended

name follows the standard C++ variable naming rules

* lets the compiler know that this is a pointer variable

pointers must have a type - lets the compiler know that this pointer is pointing to an **int**, for example



declaring pointers

- The * modifies the variable name, not the type!

```
int* a, b;
int jennysNumber = 8675309;
```

- In this example, **a** is a pointer to an integer...
  b is just a plain old integer, not a pointer
- This will not compile.

Making the Pointer "Point" Somewhere

- Pointers store the ***address*** of a variable.
- You *get* the address of something with the reference (or address-of) operator: **&**

```
int count = 5;
int *countPtr = &count;
```

- **&** is a unary operator that returns the memory address of its operand

null pointers

- A pointer that doesn't point to anything is known as a **null pointer**

```
// these are equivalent
int *ptr1 = NULL;
int *ptr2 = 0;
```

NULL is a constant that means 0

- Pointers should *always* be initialized! Make them point somewhere, or make them a null pointer. (What happens if you don't?)

“Using Pointers”

- What does the following code output?

```
int count = 5;
int *countPtr = &count;

cout << countPtr << endl;
```

- The numeric value of a pointer is almost never useful - we mainly care about what the pointer points to
- When *is* the numeric value useful?

“Using Pointers” 2

(electric boogaloo)

- Introducing: another use for the * symbol, this time known as a **dereference operator**

```
int count = 5;
int *countPtr = &count;

cout << *countPtr << endl;
```

this code will print out **5**

- * in front of a pointer means: “return the value of what this is pointing to”. This is known as ***dereferencing*** the pointer

One *, two meanings

- When you see a * in a variable declaration, after a type, then you are *declaring a pointer*.

```
int* thisIsAPointer;
char* lassie;
```

- When you see a * before variable (or expression) that's *not being declared*, it's a dereference.

```
cout << *pointer << endl;
number += *count;
```

Son of "Using Pointers"

So:

& gets returns the address of a variable

and:

* takes an address and returns the value of what is at that address

& and * are sort of each others' inverses:

```
int gazonk = 5;
cout << *(&gazonk) << endl;
```

"Using Pointers" Strikes Back

- Dereferencing is what gets you into trouble if your pointers are somehow incorrect!
- This is the root cause of many, many, many bugs in software

what do these do?

```
int *ptr = NULL;
cout << *ptr << endl;

int *ptr2;
cout << *ptr2 << endl;
```

One more time...

```
int* var = 1234;

// what does this do?
var = 89;

// how about this one?
*var = 89;
```

Why do we care about any of this pointer stuff?

- Pointers allow:
  - dynamic memory allocation of stuff
  - complicated data structures
  - iterating through strings
  - ... and much much more

Pointers and Arrays

- Simply put:
  - an array **is** a pointer - it points to the first element of the array.
  - A pointer can be used exactly like an array

```
int numbers[] = {4,8,15,16,23,42};
int *array = numbers;
cout << numbers[2] << endl;
```

  - At this point, `numbers` and `array` are basically equivalent!

Pointer Arithmetic

- Pointers are variables, and you can do math on them...
- ... but it's not the kind of math you're probably expecting.
- What would this do?

```
int quux = 42;
int *ptr = &quux;

ptr *= 2;
```

Pointer Arithmetic 2

- Only addition and subtraction are allowed
  - The other arithmetic ops make no sense!
- The math doesn't work the way you'd expect:

```
int numbers[] = {4,8,15,16,23,42};
int *ptr = numbers;
ptr++;
```

- If `ptr` was pointing to memory location 8064 before, where is it pointing now?

```
int numbers[] = {4,8,15,16,23,42};
int *ptr = numbers;
ptr++;
```

- If `ptr` was pointing to memory location 8064 before, where is it pointing now?
- Pointer arithmetic units are the ***same as the type size!***
- Aka, **int** pointers work in units of 4, because the size of an **int** is 4 bytes
- This is handy: in this example, what value is ptr pointing to now?

```
int numbers[] = {4,8,15,16,23,42};
int *ptr = numbers;
```

What are some different ways to refer to the third element of this array, `15`?

What would happen if we did this:

```
ptr += 3;
```

Grokking Pointers

- How could we make a swap function with pointers instead of pass-by-reference?
- How would you declare (and use) a pointer to a pointer?
- Can you have two pointers that point to the same variable?