ARRAYS & POINTERS

Project One

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

mehr Bericht!

- What are the three types of loops in C++?
- What does **break** do? **continue**?
- What does a C++ function look like?
- What does the **return** keyword do, and how is it used?
- What's in a header file?
- What is pass-by-reference?

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?

Default Arguments

- This is a nifty way to specify defaults for some (or all) arguments to a function
- When you're calling that function, you don't have to specify every argument if there is a default
- Very handy, very widely used

Default Arguments Example

```
void printLetterOnScreen( char letter,
                          int xPos = 10, int yPos = 10,
                          int repeatCnt = 1 )
{
    // do stuff
}
```

These are all valid ways to call this function:

```
printLetterOnScreen( ‘g’ );
printLetterOnScreen( ‘p’, 15 );
printLetterOnScreen( ‘w’, 15, 42 );
printLetterOnScreen( ‘x’, 15, 42, 5 );
```

Default Arguments Example

```
void printLetterOnScreen( char letter,
                          int xPos = 10, int yPos = 10,
                          int repeatCnt = 1 )
{
    // do stuff
}
```

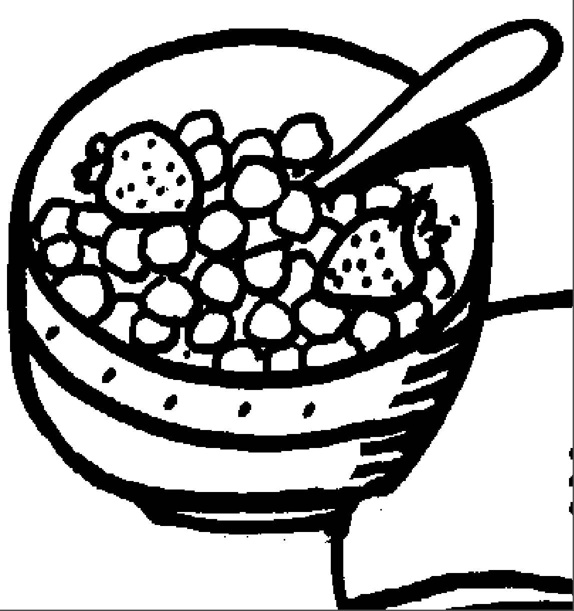
- Only *trailing* arguments can have default values
  - If a argument has a default, *all* of the following arguments also need them
- When calling a function, “skipping” arguments is illegal

```
printLetterOnScreen( ‘p’, 15 );
```

15 will be the value of xPos, not yPos or repeatCnt

Default Arguments and Function Prototypes

- By convention, default arguments usually go in the the function prototype
- They can also be put in the function definition itself - but *not* in both places
  - some compilers allow this, as long as the default arguments match - g++ doesn't



Function Overloading

- Don't be fooled by the scary-sounding name: function overloading is a *good* thing!
- The idea: multiple functions can be defined with the same name
- The compiler will automatically pick which function to call, based on the number and type of arguments

overloading examples

which function gets called?

```
void blegh( char letter )
{
}

void blegh( char letter, int reps )
{
}

void blegh( int number )
{
}

void blegh( float realNum )
{
}

void blegh( bool maybe )
{
}
```

```
blegh( 25 );

blegh( 'a' );

blegh( false );

blegh( 'q', 5 );

blegh( 5 > 2 );

blegh( 97, 5 );

blegh( 32.0 );
```

Ambiguity

- When the compiler can't figure out which version of an overloaded function to call, the function is said to be **ambiguous**
- This isn't always obvious, as you saw with the 32.0
- The previous example, now with a default parameter:

```
void blegh( char letter )
{
}

void blegh( char letter, int reps = 0 )
{
}
```

`blegh( 'a' );`
goes to which function?

These are ambiguous, so you get a compiler error

Overloading and return types

- Overloaded functions need to have differing *parameters* - different *return types* is not enough

```
int doStuff()
{
    // ...
}
```

```
double doStuff()
{
    // ...
}
```

- This will cause a compiler error
- Why do you think this is?

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;
```

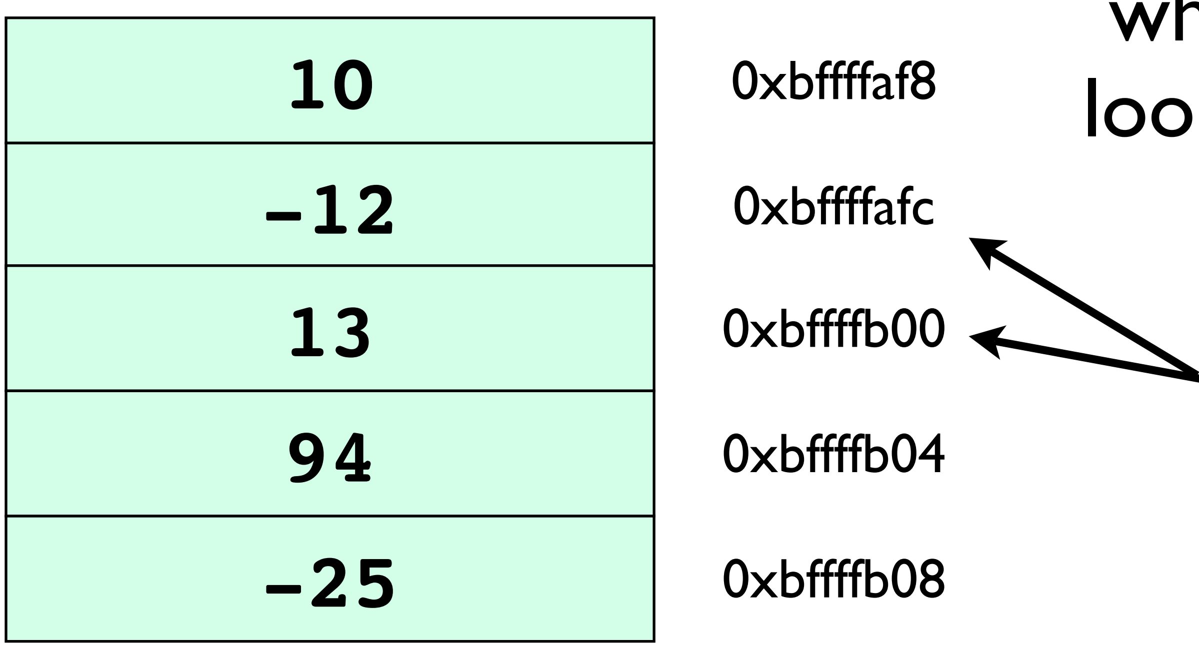
- Example with a for loop...

Array Storage

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

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

what this might end up looking like in memory...



what are these?

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;
```

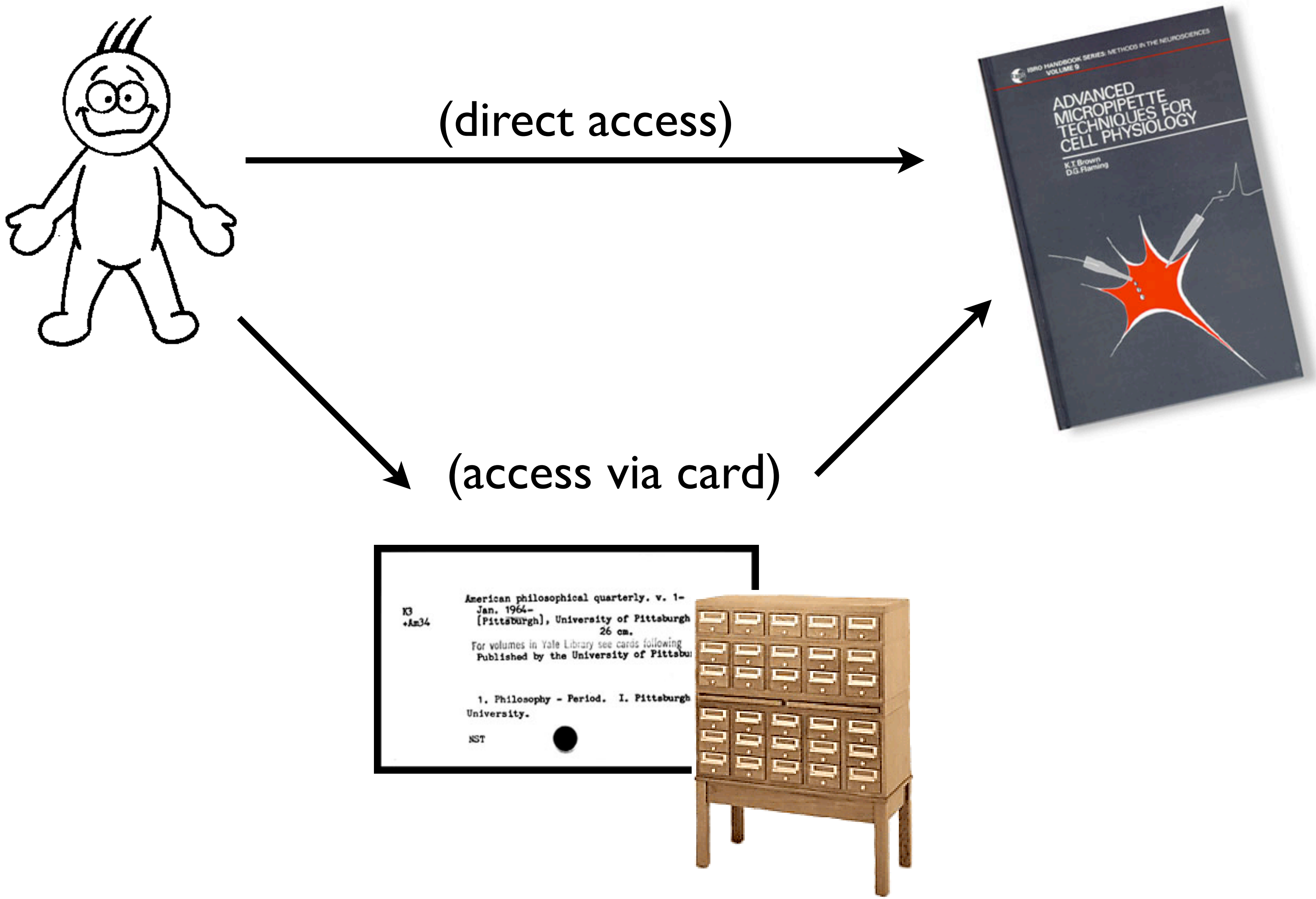
Multidimensional Arrays

- 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!!!

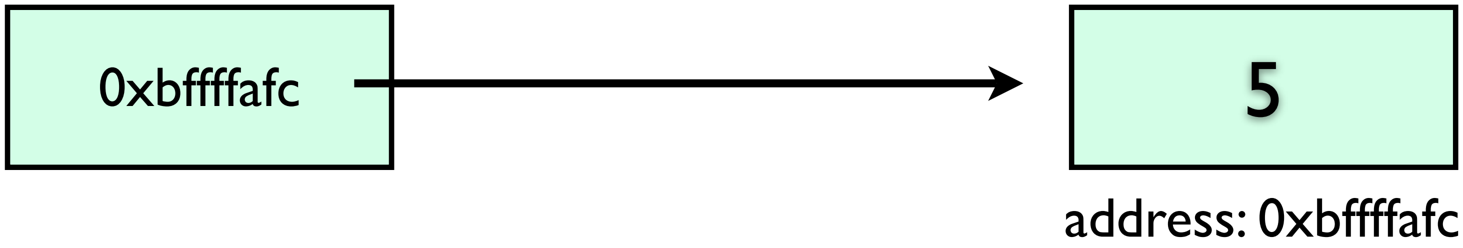


(direct access)
IBRO HANDBOOK SERIES: METHODS IN THE NEUROSCIENCES
VOLUME 9
ADVANCED MICROPIPETTE TECHNIQUES FOR CELL PHYSIOLOGY
K.T. Brown
D.G. Flaming
(access via card)
K3
+Am34
American philosophical quarterly. v. 1-
Jan. 1964-
[Pittsburgh], University of Pittsburgh
26 cm.
For volumes in Yale Library see cards following
Published by the University of Pittsbu
1. Philosophy - Period. I. Pittsburgh
University.
NST

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

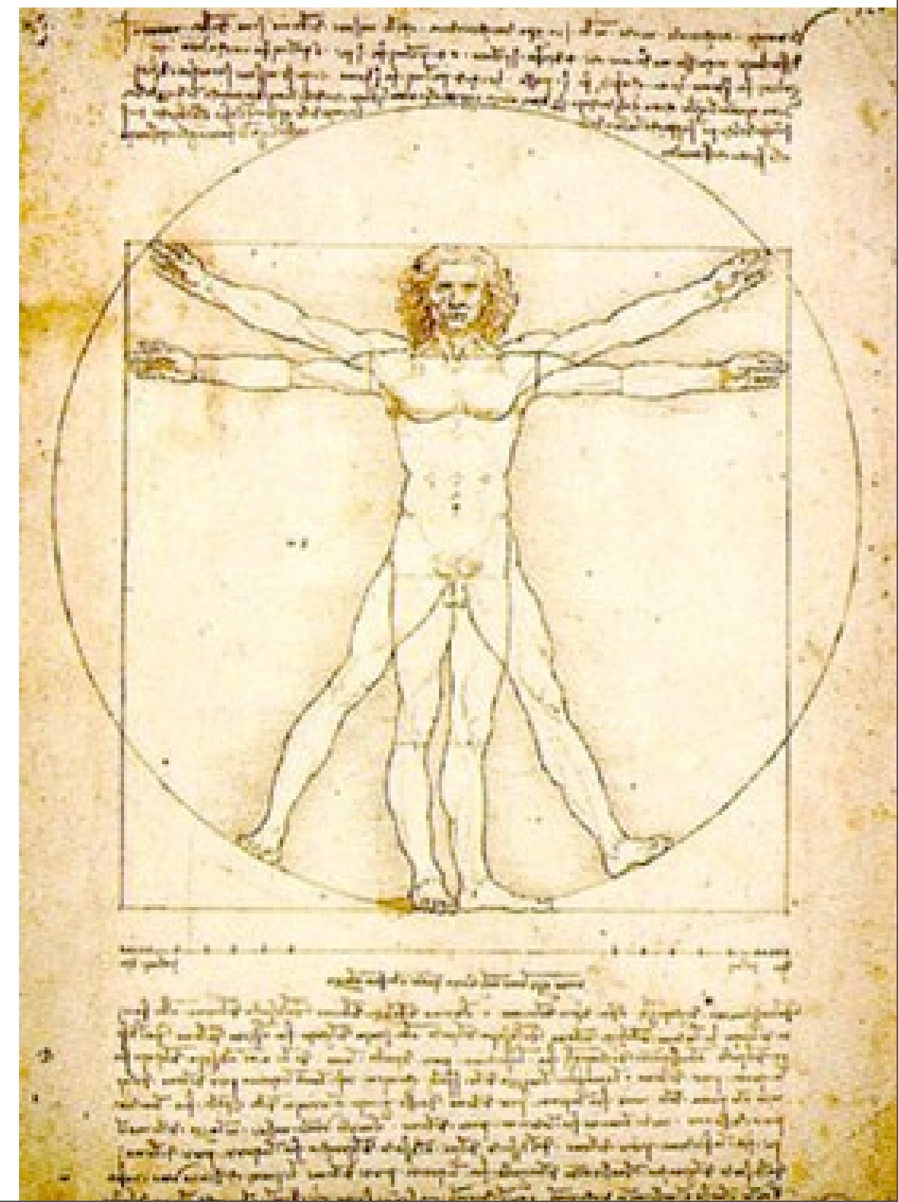
```
int *pointer = NULL;
```

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

```c
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:

```c
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?

Pointer Quizlet

```
int main()
{
    float ff = 5.5;
    float* ptr = &ff;

    cout << " 1: " << &ff << endl;
    cout << " 2: " << ptr << endl;
    cout << " 3: " << &ptr << endl;
    cout << " 4: " << *ptr << endl;
    cout << " 5: " << ff << endl;
    cout << " 6: " << *&ff << endl;

    return 0;
}
```

Scope and Lifetime

- **Scope** is the context in which a C++ variable name exists. You can use the same variable name in two (or more) functions, because the functions will have different scopes.
- Scope is defined by curly brackets: { }

```
void sunshine()
{
    ...
}
```

The scope of sunshine()

Local Scope

- Each function has its own scope - variables that are usable between the functions starting and ending curly brackets { }

```
int doSomething( int quux )
{
    int foo = 0;
    while( value < 10 )
    {
        int count =0;
        ...
    }
    int baz;
}
```

foo and quux are visible within this scope. What about baz?

Local Scope Part Deux

- A while loop (or *any* set of curly brackets) will create its own scope, and can have its own variables.

```
int doSomething( int quux )
{
    int foo = 0;
    while( value < 10 )
    {
        int count =0;
        ...
    }
    int baz;
}
```

`count` is only visible within the scope of the `while` loop.

Local Scope #3

```
for( int i = 0; i < 5; i++ )
{
    ...
}
```

what's the scope for
these variables?

```
int doSomething( int quux )
{
    ...
}
```

functions and for loops have variables declared in their headers - the scope of those is the scope of the function or loop

```
int foo()
{
    int low = 6;
    bool flag = true;
    cout << "low2: " << low << endl;
    while( flag )
    {
        int low = 7;
        int count = 8;
        cout << "low3: " << low << endl;
        flag = false;
    }
    cout << "count: " << count << endl;
    cout << "low4: " << low << endl;
}

int main()
{
    int low = 5;
    cout << "low1: " << low << endl;
    foo();
    return 0;
}
```

local definition of low in while() hides previous definition

flag visible here because no declaration overrides it

count not visible outside of while()

This is the low declared in the scope of foo()

This low is in the scope of main - it is not accessible from foo

Global Scope

- A function declaration in global scope: a global function
- A variable declaration in global scope: a global variable (or object)
- A global object is visible from everywhere: exists throughout the duration of the program

```
int GLOBAL = 42;

int main()
{
        return 0;
}
```

Global Variables == 

- Mostly.
- Why? Using global variables in a function can hide the behavior of the function.
- Any function can modify a global variable – changing the behavior of other functions that might use it.
- When are globals useful?

Lifetimes of Variables

- A lifetime is how long a variable "lives" - how long the program keeps memory allocated for it
- Local variables are "born" when the program enters their scope. They "die" when when the program leaves their scope.
- What is the lifetime of a global variable?

Static Memory

- So far we've been dealing with **static memory** - variables allocated statically, at compile time.
- Static memory is declared *on the stack*
- Static memory is very easy for the compiler to deal with:
  - amount of memory fixed at compile time
  - no chance of memory leaks
- Downside(s) of static memory?

Dynamic Memory

- **Dynamic memory** is more powerful - you don't need to know the size until runtime
- Can be used as necessary
- Dynamic memory comes from *the heap* - a pool of memory set aside for this
- Downside(s) of dynamic memory?

Dynamic Allocation

- Memory is dynamically allocated through...
- ***POINTERS!!!!!!!!*** (woo!)

introducing the new keyword:

```
int* foo = new int;
```

- This syntax allocates a *single int.* You can also do this for arrays:

```
int* baz = new int[50];
```

Yet Another Review:

```
int* foo = new int;
```

foo is a dynamically allocated integer.
How do we use it?

```
int* baz = new int[50];
```

baz is a dynamically allocated *array* of integers. How do we use it?

How are these two things different?

dynamic arrays

- Arrays allocated via dynamic memory are used *exactly* the same way that arrays allocated statically are.
- Only one minor difference regarding the array pointer variable - anybody remember what it is?

Some Questions

- When does the life of a *statically* allocated variable end?
- When does the life of a *dynamically* allocated variable end?

```
for( int i = 0; i < 10; i++ )
{
    int array = new int[15];
    ...
}
```

Cleaning Up

- See the problem with the above code?
- Static variables get de-allocated right when they go out of scope - dynamic variables *need to be deleted explicitly!*
- Otherwise you get memory leaks

Memory Leaks

- When you use a pointer to dynamically allocate memory...
- ... and the pointer goes out of scope before you have *deallocated* the memory...
- Then you have a memory leak.
- These are (usually) cleaned up by the operating system after the program exits, but the program can still run out of memory while it is running

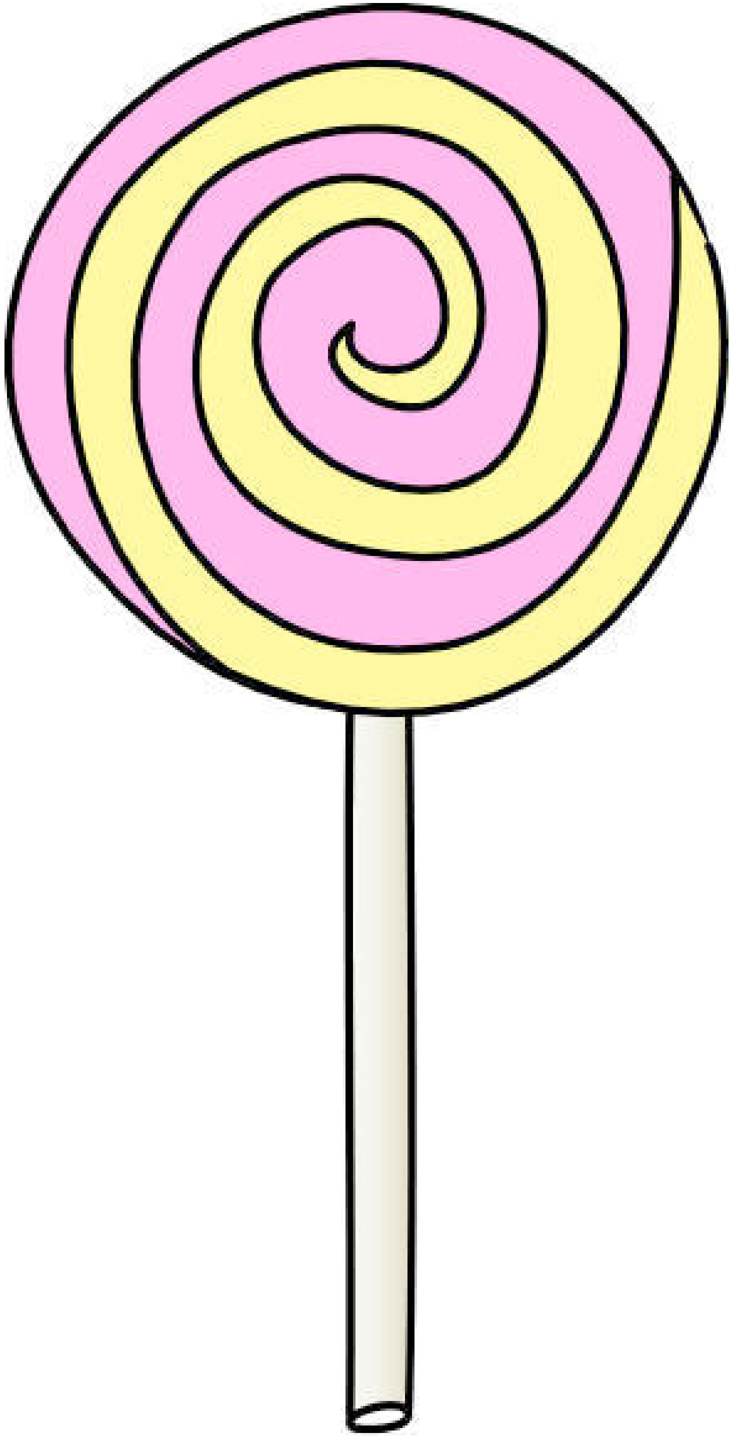
Cleaning Up

- *Single objects*, allocated with **`new`**, get cleaned up with the keyword **`delete`**:

```
int* foo = new int;
...
delete foo;
```

- *Arrays*, allocated with **`new`** and **`[]`**, get cleaned up with the keyword **`delete[]`**:

```
int* baz = new int[10];
...
delete[] baz;
```



Fun with `delete`!

- What happens if we try and **`delete`** an *array* of dynamically allocated stuff?
- What if we try and **`delete`** a pointer that has been assigned the address of a static variable?
- What if we try to **`delete[]`** a pointer that has been allocated with a single **`new`**?

Useless Program Time!

Let's write a program that gets a number from the user, dynamically an array of that size, fills it with n powers of two, and prints 'em all out.