POINTERS & DYNAMIC MEMORY

Array Review ?'s

- How do you declare an array?
  - What's important while declaring one?
  - A multi-dimensional array?
- How do you access an array?
  - A multi-dimensional array?
- How do you pass an array to a function?
- How do you determine how many elements are in an array?
- How do you make a copy of an array?

Pointer Review ?'s

- How do you declare a pointer?
- How do you make a pointer point to something?
- Can you change which variable a pointer points to?
- Can you change the *value* of what a pointer points to?
- Can you make a pointer point to a different type of variable?
- What is *dereferencing*?
- What do you need to be careful of when dereferencing?

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;
}
```

Grokking Pointers

- How are arrays related to 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? *(We haven't covered this explicitly but hopefully we can figure it out)*
- Can you have two pointers that point to the same variable?

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;
```

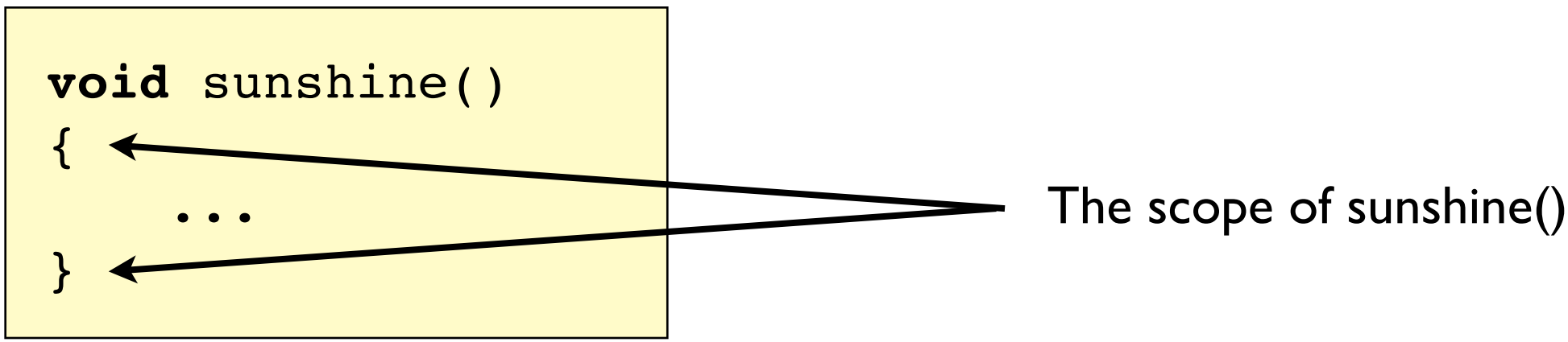
What are some different ways to refer to the third element of this array, `15`?

What would happen if we did this:

```
ptr += 3;
```

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: { }



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

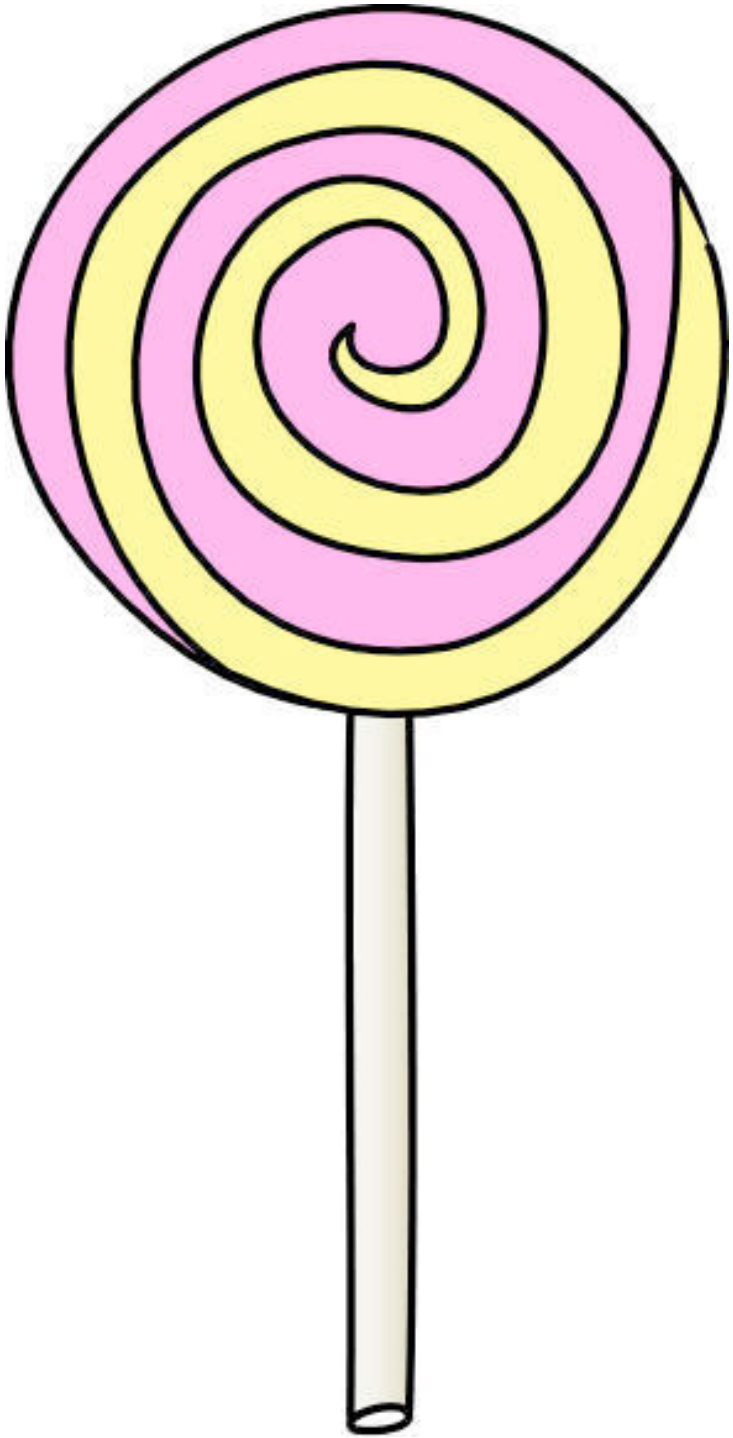
Cleaning Up

- *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 allocates an array, fills it with n powers of two, and prints 'em all out.