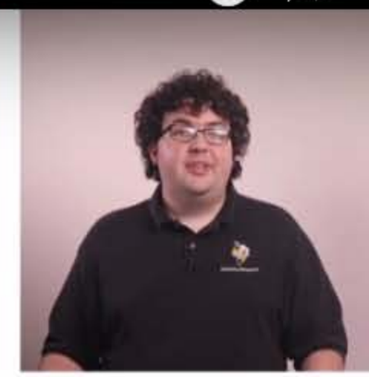
Pointers

- Indirect handle or alias for another object
- Can change which object they point to
- Are nullable (can point to no object)

Address-of Operator

- A pointer holds a memory address
- "Address-of operator" (&) gives us the address of an object
- Used to get a pointer to an existing object



Nullability

- A "null" pointer doesn't point at any object
- The `nullptr` constant makes a pointer null

```
int* n_ptr = nullptr;
```

- Dereferencing a null pointer often causes a "segmentation fault"

Computer Memory

- The "stack"
  - Used for function parameters and local variables
- The "heap"
  - General purpose memory
  - Stores objects whose lifetimes aren't bound to a specific scope

Managing the heap

- "new" creates an object on the heap

```
std::string* p = new std::string("The heap!");
```

- "delete" destroys an object on the heap

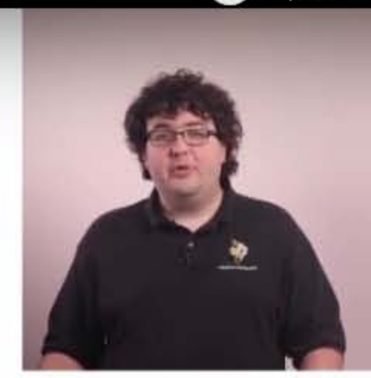
```
delete p;
```

Dynamic Arrays

- new / delete can create arrays of objects on the heap

```
int* arr = new int[10];
delete[] arr;
```

- arr points the first element of the array
- All elements are stored consecutively



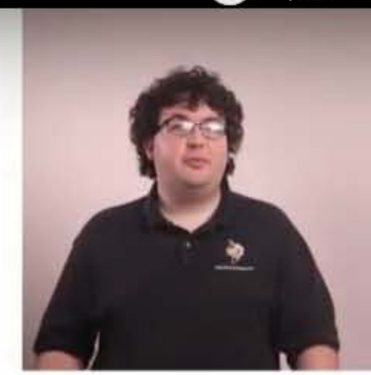
Dynamic Arrays

- Element access via square brackets

```
int* arr = new int[10];
arr[2] = 11;
```

- Iteration by incrementing pointers

```
for(int* c = arr; (c - arr) < 10; ++c) {
    *c *= 2;
}
```



Problems with new and delete

```
void F(int x) {
   Tool* tool = new Tool();
   ...
   if (x == 2) {
      return;  // no delete, memory leaked
   }
   ...
   delete tool;
}
```

Smart Pointers

- Wrap raw pointers
- Clean up unused heap objects automatically
- Take advantage of when destructors are called

std::unique_ptr

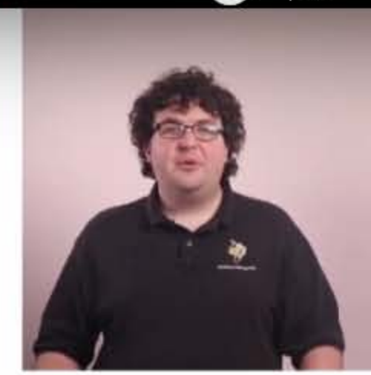
- For pointers with a single owner
- Cannot be copied
- Destructor calls delete

```
std::unique_ptr<T> u= std::make_unique<T>();
```

std::shared_ptr

- For objects with multiple owners
- Uses reference counting
- Destructor calls delete when reference count drops to zero

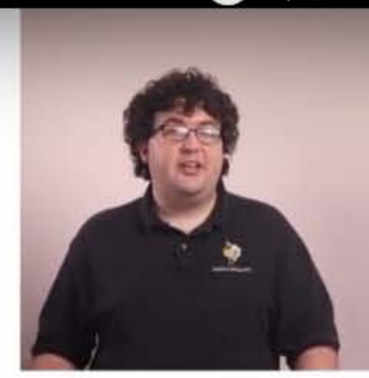
```
std::shared_ptr<T> s = std::make_shared<T>();
```



std::weak_ptr

- A non-owning pointer to an object managed by shared_ptr's
- Do not affect the reference count
  - they can't keep the object from being destroyed
- Initialize with a shared_ptr

```
std::shared_ptr<T> s = …;
std::weak_ptr<T> w = s;
```



std::weak_ptr

- Check if the object has been destroyed

```
If(w.expired()) {
  // the object has already been destroyed
}
```

- Get a new shared_ptr from a weak_ptr with lock()

```
std::weak_ptr<T> w = …;
std::shared_ptr<T> s = w.lock();
```