Announcements

- Program 1 is up - due next Monday (pair project)

- Lab tomorrow
**3. Pointer variables**

Syntax: `int * d;`

- `d` is created as a variable that stores a memory address.

**Example:**

```
int b[8];
int* d;
d = &b[0];
```

- The address of `b` is stored in `d`.
- `d` is not an `int`.
- Can't write `d = b;`!
The new command

```cpp
int* c;
c = new int(12);
```

creates a separate piece of memory

Main use: the data persists even after the pointer is gone!

So can create or modify inside multiple functions.

```cpp
cout << *c << endl;  // print 12
```
Pointers: getting to the data

Called dereferencing.

Ex: Point * d;  // OK
Point b(3, 5);  // OK
d = &b;  // OK

2 options:

(*d). getX();

or

d -> getY();
Passing pointers - address stored in input parameter

```c
bool isOrigin(Point *pt) {
    return pt->getX() == 0 && pt->getY() == 0;
}
```

*pt, sety(s),

Similar to passing by reference, but allows passing a NULL pointer also.

```
Point* pt = input parameter; pt
```

Linked lists: □→□→□

pt □-y 10b1
Pointers in a class

Pointers are especially useful in classes.

Often, we don’t know all the details of private variables to put in the private declaration.

Example: arrays!

What do we need when creating an array?

Size & type
class MyIntArray {

  private:
  int _size; // size of this array
  int* _A;  // pointer to our array

  public:
  MyIntArray (int s = 10) : _size(s) {
    // need to create array
    _A = new int[_size];
  }

  int array[10];
  add, sort, average, max
}
Accessing the array:

With an array, can just pretend the variable isn't a pointer. (so no * or \rightarrow)

Ex:

\[ A[0] = 12 \]
\[ A[size - 1] = 1 \]
This lets you delay creating the array!

Also, if you need to change size:

```c
void resize(int newsize) {
    int *newarray = new int[newsize];
    // assume newsize > _size
    for (int i = 0; i < _size; i++)
        newarray[i] = A[i];
    delete _A;
    _A = newarray;
    _size = newsize;
    // need to delete old array
}
```
Variables (recap)

1. Value - standard

2. Reference - alias (usually used in function passing)

3. Pointer - just a memory address
Garbage Collection

In Python, variables that are no longer in use are automatically destroyed.

Pros: Easy (not our problem)

Cons: Less control
Slow
C++

In C++, things are sometimes handled for you.

Basically, any standard variable is automatically destroyed at the end of its scope.

This holds for any type of variable!

```cpp
int main ()
{
    int a;

    // a is destroyed
```
Problem: Pointers

While the pointer variable is deleted, the spot you created with a "new" is not.

```c
main()
    int * a = new int(5);
    delete a; // Memory leak
```

Rule: If you have a delete, you must have
Destructors

If your class opens files or allocates memory, then you must have a destructor.

~ClassName() ← no inputs  
  no return type

Ex: ~MyIntArray()  
    delete _A;
    3  // A is destroyed
Copy Constructor

Consider that MyIntArray class.

What if we have 2 MyIntArrays, set a=b?

By default, compiler sets each private variable equal to other.

a.size = b.size
a.A = b.A

Shallow copy
To avoid shallow copies we need to make a copy constructor function.

`MyIntArray(const MyIntArray & other)`