Deep vs shallow copy
Rule of Three
class vs struct
Topics

- Pointers and arrays
- Command line args
- classes vs. struct
- Another case for overloading an operator
  - Shallow copy
  - Deep copy
Pointers (again) and Arrays
C++ Command Line Arguments

main (int argc, char ** argv)

Let's look at:

    int argc, char** argv
char** argv

argv is a pointer to a pointer to char

The C-style “string” is a sequence of chars that are terminated by 0. Also called an ASCII-Z string.
If you REALLY want to avoid char*, you can...

    string myArg = string(argv[i]);

*keep in mind that argv[i] is a char POINTER
These mean the same thing

*(argv+i)

argv[i]
These mean the same thing*

int int_array[5];

int* int_ptr = new int(5);

*not really...what is the actual difference here?

Hint: Where are these two located in memory?
The following slides marked “Note”…

1. Are intended to help you understand exactly what is going on with pointers.
2. DO NOT use pointers and arrays interchangeably.
Note...

You can do this *(but don’t)*:

```c
int int_array[5];
int* int_ptr2 = int_array;
```
Note…

You can ALSO do this **(but don’t)**:

```c
int int_array[5];

int* int_ptr2 = &int_array[0];
```
You can ALSO do this *(but don’t)*:

```c
int int_array[5];
int* int_ptr2 = &int_array[2];
```
Note...

You can ALSO do this \textit{(but don’t)}:

```c
int int_array[5];
int* int_ptr2 = &int_array[2];
```

Which makes \texttt{int_ptr2[-2]} the same as \texttt{int_array[0]}
Array2D: dynamic memory use

Recall that the Array2D class allocated memory in the constructor based on the number of rows and columns specified.
Array2D: dynamic memory use

Array2D::Array2D(int iRows, int iColumns) {
    row = iRows;
    columns = iColumns;
    count = rows*columns;
    myInts = new int[ccount];
    for (int i=0; i<count; i++)
        myInts[i] = count-i;
}
Array2D: What does this mean?

Array2D obj1(2, 2);
Array2D obj2;
obj2 = obj1;

What would we LIKE it to mean?
Overloading “=”

What does this mean?

```
Array2D obj1(2, 2);
Array2D obj2;
obj2 = obj1;
```

If what we want is ANOTHER object that contains all of the information of the original object then we can just copy all of the values from the first to the second*.

*what if an object has a pointer to some data?
Rule of Three

The **rule of three** is a rule of thumb in C++ (prior to C++11) that claims that if a class defines any of the following then it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator
Array2D required a destructor*

Array2D::~Array2D(){
    cout<<"Array2D destructor: "<<name<<endl;
    delete[] myInts;
}

*why?
New things: Array2D

private:
    std::string name;

public:
    setValue(row, column, value)
    setName(name);

Overloaded “<<” has changed
New things: Array2D

Copy constructor:

Array2D(const Array2D& arr);
Overloading “=”

What does this mean?

Obj2 = Obj1;

What would we LIKE it to mean?
Overloading “=”

What does this mean?

Obj2 = Obj1;

If what we want is ANOTHER object that contains all of the information of the original object the we can just copy all of the values from the first to the second*. 

Overloading “=”

What does this mean?

Obj2 = Obj1;

If what we want is ANOTHER object that contains all of the information of the original object then we can just copy all of the values from the first to the second*.

This is what is known as a “shallow copy”.
Array2D overloaded “=” (shallow copy)

Array2D& Array2D::operator = (const Array2D &arr){
    rows = arr.rows;
    columns = arr.columns;
    count = arr.count;
    myInts = arr.myInts;
    return(*this);
}
Rule of Five

In addition to the 3 rules in The Rule of 3, the *rule of five* identifies that it is usually appropriate to provide the following functions to allow for optimized copies from temporary objects:

- move constructor
- move assignment operator
What is a “move constructor”?

The move constructor

1. copies the simple values from the source
2. copies pointers to resources from the source (shallow copy)
3. destroys the source’s access to the resources it has “hijacked”
struct vs class

- A struct is a user-defined data type that groups data types together.
- You can think of it as a class with no functions.
- You can use access modifiers on the fields of a struct.
struct vs class

struct Automobile {
    char manufacturer[48];
    char model[25];
    float weight;
    int year;
    string* story;
};
struct

struct Automobile car;
Automobile* car_ptr = &car;
Automobile car_array[5];
Automobile* new_car = new Automobile;
struct

struct Automobile car; // allocates memory right here for this data
Automobile* car_ptr; // declares a pointer
Automobile* car_ptr = &car; // declares a pointer and initializes it
Automobile car_array[5]; // declares an array of 5 Automobiles
Automobile* new_car = new Automobile;

// allocates memory and initializes a pointer
How much memory?

struct Automobile {
    char manufacturer[48];
    char model[25];
    float weight;
    int year;
    string* story;
};
How much memory?

48:   char manufacturer[48];
25:   char model[25];
4:     float weight;
4:     int year;
8:     string* story;

Calculated Total: 89  (not the ACTUAL total)