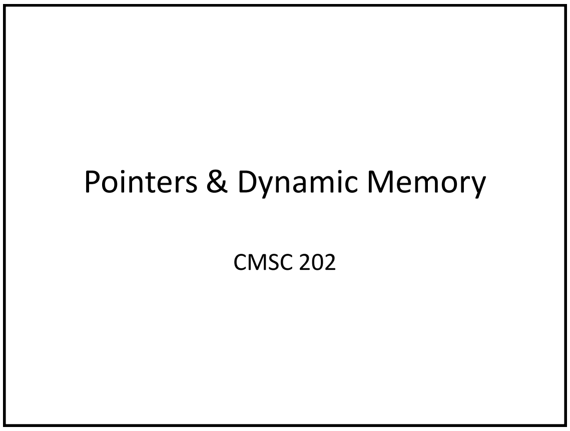
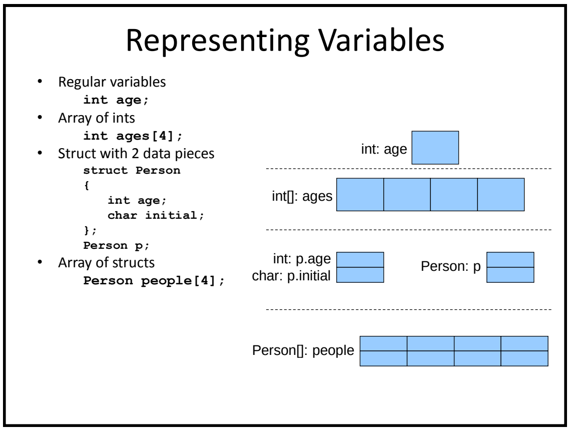


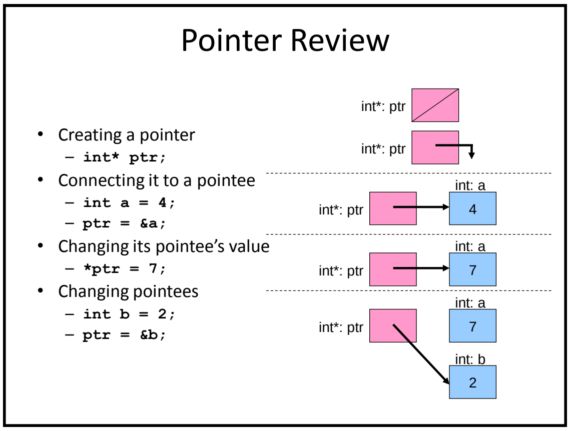
# Pointers & Dynamic Memory

CMSC 202



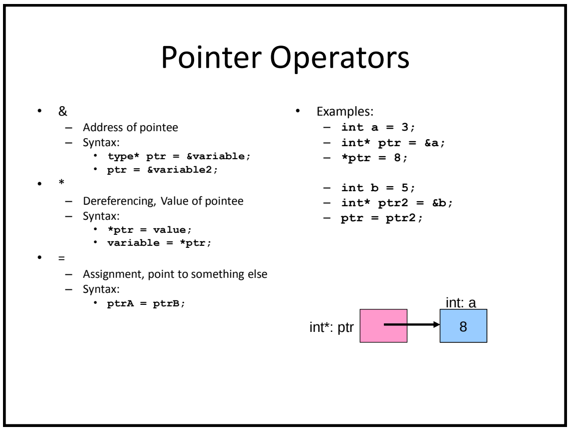
## Representing Variables

- Regular variables
  ```
  int age;
  ```
- Array of ints
  ```
  int ages[4];
  ```
- Struct with 2 data pieces
  ```
  struct Person
  {
      int age;
      char initial;
  };
  Person p;
  ```
- Array of structs
  ```
  Person people[4];
  ```



## Pointer Review

- Creating a pointer
  - `int* ptr;`
- Connecting it to a pointee
  - `int a = 4;`
  - `ptr = &a;`
- Changing its pointee's value
  - `*ptr = 7;`
- Changing pointees
  - `int b = 2;`
  - `ptr = &b;`



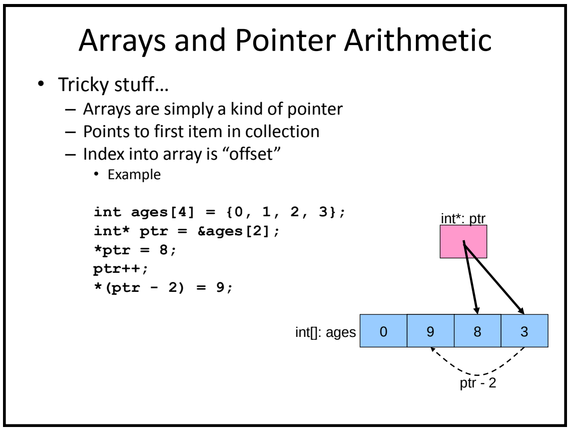
## Pointer Operators

- &
  - Address of pointee
  - Syntax:
    - `type* ptr = &variable;`
    - `ptr = &variable2;`
- *
  - Dereferencing, Value of pointee
  - Syntax:
    - `*ptr = value;`
    - `variable = *ptr;`
- =
  - Assignment, point to something else
  - Syntax:
    - `ptrA = ptrB;`

- Examples:
  - `int a = 3;`
  - `int* ptr = &a;`
  - `*ptr = 8;`

  - `int b = 5;`
  - `int* ptr2 = &b;`
  - `ptr = ptr2;`

int: a
int*: ptr
8

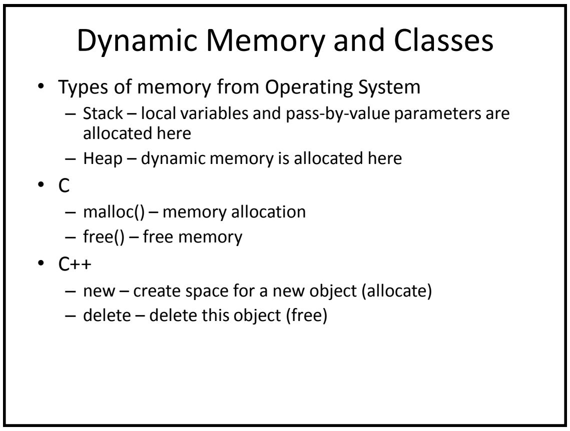


## Arrays and Pointer Arithmetic

- Tricky stuff…
  - Arrays are simply a kind of pointer
  - Points to first item in collection
  - Index into array is "offset"
    - Example

```
int ages[4] = {0, 1, 2, 3};
int* ptr = &ages[2];
*ptr = 8;
ptr++;
*(ptr - 2) = 9;
```

int*: ptr
int[]: ages
0
9
8
3
ptr - 2



## Dynamic Memory and Classes

- Types of memory from Operating System
  - Stack – local variables and pass-by-value parameters are allocated here
  - Heap – dynamic memory is allocated here
- C
  - malloc() – memory allocation
  - free() – free memory
- C++
  - new – create space for a new object (allocate)
  - delete – delete this object (free)

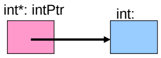
## New Objects

- new
  - Works with primitives
  - Works with class-types
- Syntax:
  - `type* ptrName = new type;`
  - `type* ptrName = new type( params );`

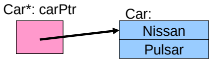
Constructor!

## New Examples

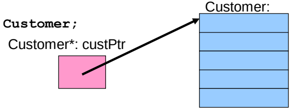
```
int* intPtr = new int;
```



```
Car* carPtr = new Car("Nissan", "Pulsar");
```



```
Customer* custPtr = new Customer;
```



**Notice:**

These are unnamed objects! The only way we can get to them is through the pointer!

Pointers are the same size no matter how big the data is!

## Deletion of Objects

- delete
  - Called on the pointer to an object
  - Works with primitives & class-types
- Syntax:
  - delete ptrName;
- Example:
  - `delete intPtr;`
  - `intPtr = NULL;`

  - `delete carPtr;`
  - `carPtr = NULL;`

  - `delete custPtr;`
  - `custPtr = NULL;`

**Set to NULL so that you can use it later – protect yourself from accidentally using that object!**

## Video!

Pointer Fun with Binky
http://cslibrary.stanford.edu/104/

## Practice

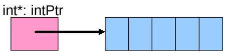
- Assume you have a Shoe class:
  - Create a pointer to a Shoe
  - Connect the pointer to a new Shoe object
  - Delete your Shoe object
  - Set pointer to null

```
Shoe* shoePtr;

shoePtr = new Shoe;

delete shoePtr;

shoePtr = NULL;
```

## Dynamic Arrays?!

- Syntax
  - `type* arrayName = new type[ size ];`
  - `type* arrayName = new type[ size ] ( params );`
  - `delete [ ] arrayName;`
- Example

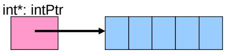
```
int* intPtr;
intPtr = new int[ 5 ];

Car* carPtr;
carPtr = new Car[ 10 ];

Customer* custPtr;
custPtr = new Customer[ 3 ];
```



Constructor!

## Dynamic Arrays?!

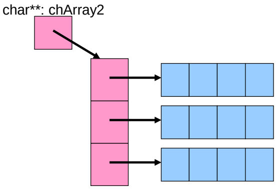
- Syntax
  - `type* arrayName = new type[ size ];`
  - `type* arrayName = new type[ size ] ( params );`
  - `delete [ ] arrayName;`
- Example



```
int* intPtr;
intPtr = new int[ 5 ];

Car* carPtr;
carPtr = new Car[ 10 ] ( "Nissan", "Pulsar" );

Customer* custPtr;
custPtr = new Customer[ 3 ];
```

Constructor!

## Dynamic 2D Arrays



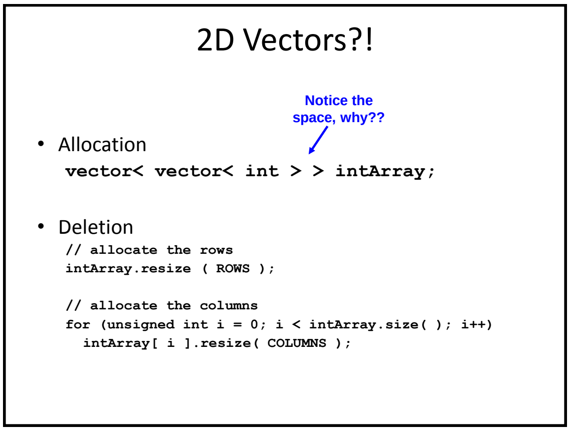
- Algorithm
  - Allocate the number of rows
  - For each row
    - Allocate the columns
- Example

```
const int ROWS = 3;
const int COLUMNS = 4;

char **chArray2;

// allocate the rows
chArray2 = new char* [ ROWS ];

// allocate the (pointer) elements for each row
for (int row = 0; row < ROWS; row++ )
   chArray2[ row ] = new char[ COLUMNS ];
```

## Dynamic 2D Arrays

- Delete?
  - Reverse the creation algorithm
    - For each row
      - Delete the columns
    - Delete the rows
- Example

```
// delete the columns
for (int row = 0; row < ROWS; row++)
{
   delete [ ] chArray2[ row ];
   chArray2[ row ] = NULL;
}

// delete the rows
delete [ ] chArray2;
chArray2 = NULL;
```



# 2D Vectors?!

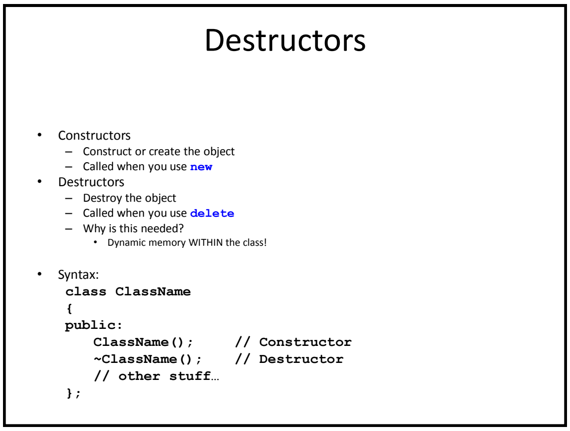
- Allocation

Notice the space, why??

```
vector< vector< int > > intArray;
```

- Deletion

```
// allocate the rows
intArray.resize ( ROWS );

// allocate the columns
for (unsigned int i = 0; i < intArray.size( ); i++)
  intArray[ i ].resize( COLUMNS );
```

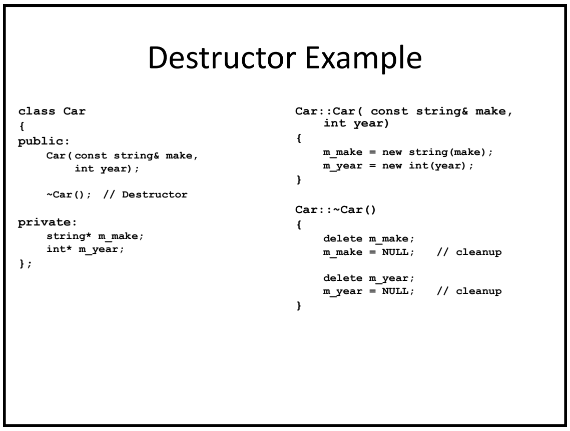


# Destructors

- Constructors
  - Construct or create the object
  - Called when you use `new`
- Destructors
  - Destroy the object
  - Called when you use `delete`
  - Why is this needed?
    - Dynamic memory WITHIN the class!

- Syntax:

```
class ClassName
{
public:
    ClassName();     // Constructor
    ~ClassName();    // Destructor
    // other stuff…
};
```



# Destructor Example

```
class Car
{
public:
    Car(const string& make,
        int year);

    ~Car();  // Destructor

private:
    string* m_make;
    int* m_year;
};
```

```
Car::Car( const string& make,
    int year)
{
    m_make = new string(make);
    m_year = new int(year);
}

Car::~Car()
{
    delete m_make;
    m_make = NULL;   // cleanup

    delete m_year;
    m_year = NULL;   // cleanup
}
```

## Dynamic Memory Rules

- Classes
  - If dynamic data
    - MUST have constructor
    - MUST have destructor
- Delete
  - After delete – always set pointer to NULL
    - Security
- “For every **`new`**, there must be a **`delete`**.”

## Practice

- Dynamically create an array of 50 Shoes
- Delete your array of shoes
- “Clear” the pointer

```
Shoe* shoeArray = new Shoe[ 50 ];

delete shoeArray;
shoeArray = NULL;
```

## Challenge

- Create a very simple Car class
  - Dynamically allocate an array of Passengers within the car
  - Create a constructor to allocate the array
  - Create a deconstructor to delete the array