# CSCI 200: Foundational Programming Concepts & Design Lecture 22

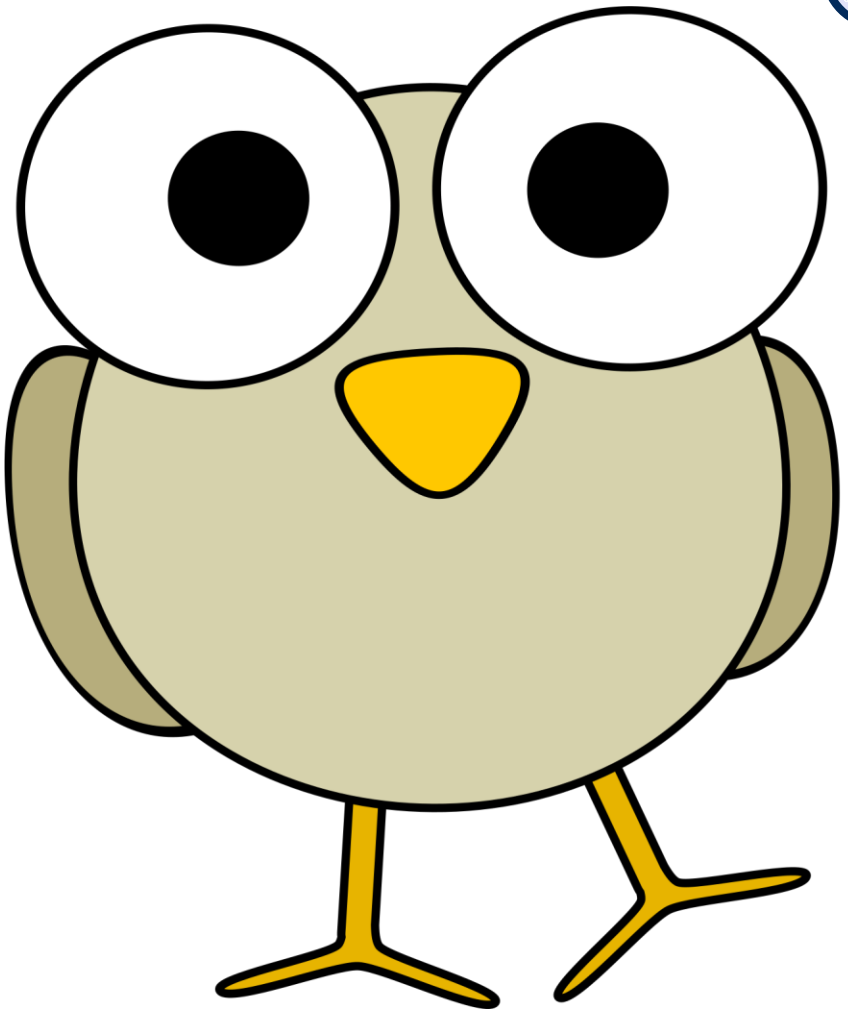
Memory Management via The Big Three

Deep v Shallow Copy

# Previously in CSCI 200

- Four uses of `const`
  - Variable modifier
  - Parameter modifier
  - Pointer modifier
  - Member function modifier

# Questions?



# Exam 2

- Monday, October 27 In Class
  - Autograded via Canvas Quiz & Hand written code portions
  - **Closed Book, Notes, Resources**
  - Review materials posted

# Exam 2 Question Makeup

1. TF, MC, FitB
2. What is the Output
3. Write code that does XYZ
   - Graded on:
     - Is the task accomplished?
     - Syntax
     - Style

- (Same as Exam 1)

# Exam 2 Topics

1. **C++**: *Variables, Data Types, Math, Conditionals, Loops, Functions*, File I/O, Formatting, Pointers, Classes
2. **CLI**: *Makefile & Debugging*
3. **DE**: *Structured & Procedural Programming, Multifile Projects, Makefiles, Debugging*, Big-O
4. **MM**: *Memory & Call Stack*, Stack & Free Store, Big 3
5. **OOP**: Classes, Access Modifiers, Big 3

# Exam 2 Review Materials

- Updated compiled Daily Learning Outcomes

- Review Questions for Extra Credit
  - Complete review questions
  - Show to instructor at start of class Oct 24
  - Receive up to 3 points XC for completion (not correctness, but attempt needs to be made)

# Example Box Class

```
// Box.h
class Box {
public:
    Box(const int SIZE);
    int getBoxSize() const;
private:
    int _size;
};
```

```
// Box.cpp
#include "Box.h"

Box::Box(const int SIZE) {
  _size = SIZE;
}
int Box::getBoxSize() const {
  return _size;
}
```

# Example Warehouse Class

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};
```

```
// Warehouse.cpp
#include "Warehouse.h"

Warehouse::Warehouse() {
    _pBoxen = new vector<Box*>;
}

void Warehouse::storeInBox(const int SIZE) {
    _pBoxen->push_back(new Box(SIZE));
}

Box* Warehouse::getBox(const size_t POS) {
  return _pBoxen->at(POS);
}

size_t Warehouse::getNumberBoxes() const {
  return _pBoxen->size();
}
```

```
// main.cpp

Warehouse *pWarehouseH = new Warehouse;    // new calls constructor

pWarehouseH->storeInBox(4);
```

# Learning Outcomes For Today

- Define, list, and implement the Big 3.
- Explain the difference between a shallow copy and a deep copy. Implement both.
- Overload common operators and discuss reasons why operator overloading is useful.

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# Example Warehouse Class

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};
```

```
// Warehouse.cpp
#include "Warehouse.h"
Warehouse::Warehouse() {
    _pBoxen = new vector<Box*>;
}
void Warehouse::storeInBox(const int SIZE) {
    _pBoxen->push_back(new Box(SIZE));
}
Box* Warehouse::getBox(const size_t POS) {
  return _pBoxen->at(POS);
}
size_t Warehouse::getNumberBoxes() const {
  return _pBoxen->size();
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);
cout << pWarehouseH << endl;      // what happens?
```

# Example Warehouse Class

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};
```

```
// Warehouse.cpp
#include "Warehouse.h"
Warehouse::Warehouse() {
    _pBoxen = new vector<Box*>;
}
void Warehouse::storeInBox(const int SIZE) {
    _pBoxen->push_back(new Box(SIZE));
}
Box* Warehouse::getBox(const size_t POS) {
  return _pBoxen->at(POS);
}
size_t Warehouse::getNumberBoxes() const {
  return _pBoxen->size();
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);
cout << pWarehouseH << endl;      // prints address 0x42dc28ad
```

# Example Warehouse Class

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};
```

```
// Warehouse.cpp
#include "Warehouse.h"
Warehouse::Warehouse() {
    _pBoxen = new vector<Box*>;
}
void Warehouse::storeInBox(const int SIZE) {
    _pBoxen->push_back(new Box(SIZE));
}
Box* Warehouse::getBox(const size_t POS) {
  return _pBoxen->at(POS);
}
size_t Warehouse::getNumberBoxes() const {
  return _pBoxen->size();
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);
cout << pWarehouseH << endl;      // prints address 0x42dc28ad
cout << *pWarehouseH << endl;     // what happens?
```

# Example Warehouse Class

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};
```

```
// Warehouse.cpp
#include "Warehouse.h"
Warehouse::Warehouse() {
    _pBoxen = new vector<Box*>;
}
void Warehouse::storeInBox(const int SIZE) {
    _pBoxen->push_back(new Box(SIZE));
}
Box* Warehouse::getBox(const size_t POS) {
  return _pBoxen->at(POS);
}
size_t Warehouse::getNumberBoxes() const {
  return _pBoxen->size();
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);
cout << pWarehouseH << endl;      // prints address 0x42dc28ad
cout << *pWarehouseH << endl;     // ERROR!
// invalid operands to binary expression ('std::__1::ostream' (aka
// 'basic_ostream<char>') and 'Warehouse')
```

# Operator Overloading

- What does overloading mean?
- What operators do we have?

# Precedence Table

| Category | Precedence | Operator | Associativity |
|---|---|---|---|
| Parenthesis | 1 | ( ) | Innermost First |
| Scope Resolution | 2 | S:: | Left to Right |
| Postfix Unary Operators | 3 | a++ a-- a. p-> f() | |
| Prefix Unary Operators | 4 | ++a --a +a -a !a ~a (type)a &a *p new delete | Right to Left |
| Binary Operators | 5 | a*b a/b a%b | Left to Right |
| | 6 | a+b a-b | |
| Shift Operators | 7 | a<<b a>>b | |
| Relational Operators | 8 | a<b a>b a<=b a>=b | |
| | 9 | a==b a!=b | |
| Bitwise Operators | 10 | a&b | |
| | 11 | a^b | |
| | 12 | a\|b | |
| Logical Operators | 13 | a&&b | |
| | 14 | a\|\|b | |
| Assignment | 15 | a=b a+=b a-=b a*=b a/=b a%=b a&=b a^=b a\|=b | Right to Left |

# Operator Overloading

- What does overloading mean?
- What operators do we have?
- Which operators can we overload?

# Precedence Table

| Category | Precedence | Operator | Associativity |
|---|---|---|---|
| Parenthesis | 1 | ( ) | Innermost First |
| Scope Resolution | 2 | S:: | Left to Right |
| Postfix Unary Operators | 3 | a++ a-- a. p-> f() | |
| Prefix Unary Operators | 4 | ++a --a +a -a !a ~a (type)a &a *p new delete | Right to Left |
| Binary Operators | 5 | a*b a/b a%b | Left to Right |
| | 6 | a+b a-b | |
| Shift Operators | 7 | a<<b a>>b | |
| Relational Operators | 8 | a<b a>b a<=b a>=b | |
| | 9 | a==b a!=b | |
| Bitwise Operators | 10 | a&b | |
| | 11 | a^b | |
| | 12 | a\|b | |
| Logical Operators | 13 | a&&b | |
| | 14 | a\|\|b | |
| Assignment | 15 | a=b a+=b a-=b a*=b a/=b a%=b a&=b a^=b a\|=b | Right to Left |

# Operator Overloading

- What does overloading mean?
- What operators do we have?
- Which operators can we overload?
- And more

# Printing the Warehouse

```h
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};

std::ostream& operator<<(
  std::ostream&, const Warehouse&
);
```

```cpp
// Warehouse.cpp
#include "Warehouse.h"

/* ... */

std::ostream& operator<<(
  std::ostream& os, const Warehouse& WH
) {
  os << "Warehouse has "
     << WH.getNumberBoxes() << " boxes";
  return os;
}
```

```cpp
// main.cpp
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);
cout << pWarehouseH << endl;      // prints address 0x42dc28ad
cout << *pWarehouseH << endl;     // prints "Warehouse has 1 boxes"
```

# Now What?

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};

std::ostream& operator<<(
  std::ostream&, const Warehouse&
);
```

```
// Warehouse.cpp
#include "Warehouse.h"

/* ... */

std::ostream& operator<<(
  std::ostream& os, const Warehouse& WH
) {
  os << "Warehouse has "
     << WH.getNumberBoxes() << " boxes";
  return os;
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse, *pWarehouseC = new Warehouse;
pWarehouseH->storeInBox(4);
pWarehouseC->storeInBox(2);
pWarehouseC = pWarehouseH;        // what does this do?
```

# Now What?

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};

std::ostream& operator<<(
  std::ostream&, const Warehouse&
);
```

```
// Warehouse.cpp
#include "Warehouse.h"

/* ... */

std::ostream& operator<<(
  std::ostream& os, const Warehouse& WH
) {
  os << "Warehouse has "
     << WH.getNumberBoxes() << " boxes";
  return os;
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse, *pWarehouseC = new Warehouse;
pWarehouseH->storeInBox(4);
pWarehouseC->storeInBox(2);
*pWarehouseC = *pWarehouseH;      // what does this do?
```

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# Assignment

- Generally

```
lhs = rhs
```

- Assign the right hand side to the left hand side

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# Copying

- Performed with two `lvalue`s that are both backed by memory

- Can be done in two ways
  1. Reuse existing memory
  2. Duplicate memory

- AKA Shallow Copy or Deep Copy

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# Shallow Copy vs. Deep Copy

- Shallow Copy: create new `lvalue` backed by same memory

- Deep Copy: create new `lvalue` with new memory

# Shallow Copy vs. Deep Copy

- Shallow Copy: create new `lvalue` backed by same memory
  - Makes a new alias
- Deep Copy: create new `lvalue` with new memory
  - Makes a new instance

# Shallow Copy? Deep Copy?

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
private:
    std::vector<Box*>* _pBoxen;
};

std::ostream& operator<<(
  std::ostream&, const Warehouse&
);
```

```
// Warehouse.cpp
#include "Warehouse.h"

/* ... */

std::ostream& operator<<(
  std::ostream& os, const Warehouse& WH
) {
  os << "Warehouse has "
     << WH.getNumberBoxes() << " boxes";
  return os;
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse, *pWarehouseC = new Warehouse;
pWarehouseH->storeInBox(4);
pWarehouseC->storeInBox(2);
pWarehouseC = pWarehouseH;        // shallow or deep?
*pWarehouseC = *pWarehouseH;      // shallow or deep?
```

# Specify Copy Assignment Operator

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

/* ... */

Warehouse& Warehouse::operator=(
  const Warehouse& OTHER
) {
  // guard against self assignment
  if(this == &OTHER) return *this;

  // delete existing contents

  // perform deep copy from OTHER to this

  return *this;
}
```

```
// main.cpp
Warehouse *pWarehouseH = new Warehouse, *pWarehouseC = new Warehouse;
pWarehouseH->storeInBox(4);
pWarehouseC->storeInBox(2);
pWarehouseC = pWarehouseH;        // shallow by definition
*pWarehouseC = *pWarehouseH;      // deep by overloaded definition
```

# Now what happens?

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

// constructor

// copy assignment operator

// methods to use class
```

```
void someFunction(Warehouse wh) { /* ... */ }

someFunction( Warehouse() );      // what gets called?
```

# The What?

```
void someFunction(Warehouse wh) { /* ... */ }

someFunction( Warehouse() );       // what gets called?


// main.cpp
someFunction( Warehouse() );       // the constructor to make Warehouse
                                   // the copy constructor to make wh
```

# Copy Constructor

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    Warehouse(const Warehouse&);
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

// constructor

// copy constructor
Warehouse::Warehouse(const Warehouse& OTHER) {
  // perform deep copy from OTHER to this
}

// copy assignment operator

// methods to use class
```

```
Warehouse warehouseH;
warehouseH.storeInBox(4);

Warehouse warehouseC( warehouseH ); // copy constructor

Warehouse warehouseD;               // initialize w/ default constructor
warehouseD = warehouseH;            // copy assignment operator
```

# Copy Constructor

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    Warehouse(const Warehouse&);
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

// constructor

// copy constructor
Warehouse::Warehouse(const Warehouse& OTHER) {
  // perform deep copy from OTHER to this
}

// copy assignment operator

// methods to use class
```

```
Warehouse *pWarehouseH = new Warehouse;
pWarehouseH->storeInBox(4);

Warehouse *pWarehouseC = new Warehouse( *pWarehouseH ); // copy constructor

Warehouse *pWarehouseD = new Warehouse(); // initialize w/ default constructor
*pWarehouseD = *pWarehouseH;              // copy assignment operator
```

# Cleanup Time

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    Warehouse(const Warehouse&);
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

// constructor

// copy constructor
Warehouse::Warehouse(const Warehouse& OTHER) {
  // perform deep copy from OTHER to this
}

// copy assignment operator

// methods to use class
```

```
Warehouse *pWarehouseH = new Warehouse; // new + constructor allocates memory
pWarehouseH->storeInBox(4);             // storing in a box allocates memory
delete pWarehouse;                      // dellocate all that memory
```

# Removing object calls Destructor

```
// Warehouse.h
class Warehouse {
public:
    Warehouse();
    Warehouse(const Warehouse&);
    ~Warehouse();
    void storeInBox(const int SIZE);
    Box* getBox(const size_t POS);
    size_t getNumberBoxes() const;
    Warehouse& operator=(
      const Warehouse&
    );
private:
    std::vector<Box*>* _pBoxen;
};
std::ostream& operator<<( ... );
```

```
// Warehouse.cpp
#include "Warehouse.h"

// constructor

// copy constructor

// destructor
Warehouse::~Warehouse() {
  // delete entire contents of object
}

// copy assignment operator

// methods to use class
```

```
Warehouse *pWarehouseH = new Warehouse; // new + constructor allocates memory
pWarehouseH->storeInBox(4);             // storing in a box allocates memory
delete pWarehouse;                      // dellocate all that memory
```

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# The Big 3

- The Big 3
  - Destructor (default: delete references)
  - Copy Assignment Operator (default: shallow)
  - Copy Constructor (default: shallow)

- Rule of 3
  - If you explicitly make one of them, you should explicitly make all three

# Object Lifecycle

- Where do the following fit into an object's life cycle? When are each applied?
  - Constructor
  - Copy Assignment
  - Destructor

# Getter Beware!

- Consider this scenario

# What gets printed?

```
class InnerClass {
public:
  InnerClass() { x = 1; }
  int x;
};
class OuterClass {
public:
  InnerClass getIC();
private:
  InnerClass mIc;
};
```

```
int main() {
  OuterClass oc;
  cout << oc.getIC().x << endl;
  oc.getIC().x = 5;
  cout << oc.getIC().x << endl;
  return 0;
}
```

```
1
1
```

# What gets printed?

```
class InnerClass {
public:
  InnerClass() { x = 1; }
  int x;
};
class OuterClass {
public:
  InnerClass getIC();
private:
  InnerClass mIc;
};
```

```
int main() {
  OuterClass oc;
  cout << oc.getIC().x << endl;
  InnerClass ic = oc.getIC();
  ic.x = 5;
  cout << oc.getIC().x << endl;
  return 0;
}
```

```
1
1
```

# V1 - What gets printed?

```
class InnerClass {
public:
  InnerClass() { x = 1; }
  int x;
};
class OuterClass {
public:
  InnerClass* getIC();
private:
  InnerClass* mpIc;
};
```

```
int main() {
  OuterClass oc;
  cout << oc.getIC()->x << endl;
  InnerClass* ic = oc.getIC();
  ic->x = 5;
  cout << oc.getIC()->x << endl;
  return 0;
}
```

```
1
5
```

# V1 - What gets printed?

```
class InnerClass {
public:
  InnerClass() { x = 1; }
  int x;
};
class OuterClass {
public:
  InnerClass* getIC();
private:
  InnerClass* mpIc;
};
```

```
int main() {
  OuterClass oc;
  cout << oc.getIC()->x << endl;
  oc.getIC()->x = 5;
  cout << oc.getIC()->x << endl;
  return 0;
}
```

```
1
5
```

# V2 - What gets printed?

```
class InnerClass {
public:
  InnerClass() { x = 1; }
  int x;
};
class OuterClass {
public:
  InnerClass& getIC();
private:
  InnerClass mIc;
};
```

```
int main() {
  OuterClass oc;
  cout << oc.getIC().x << endl;
  oc.getIC().x = 5;
  cout << oc.getIC().x << endl;
  return 0;
}
```

```
1
5
```

# On Tap For Today

- Operator Overloading
- Assignment
  - Copy
    - Shallow vs. Deep
- The Big 3
- Practice

# To Do For Next Time

- Proposal due tonight
- Have a great break!
- OOP Quiz Wednesday (thru today)
- A3 due Thursday
- XC due Friday