

# **CSCI 200: Foundational Programming Concepts & Design**

## **Lecture 19**



Objects and Functions

# Previously in CSCI 200



- Create a vector of courses

Course	
- enrollment: int	
- title: string	
+ Course()	
+ Course(string)	
+ getTitle(): string	
+ getEnrollment(): int	
+ registerStudent(): void	
+ withdrawStudent(): void	

// initializes to zero

// sets title to CSM101

// sets title to param

// returns title of course

// returns enrollment of course

// increments enrollment by 1

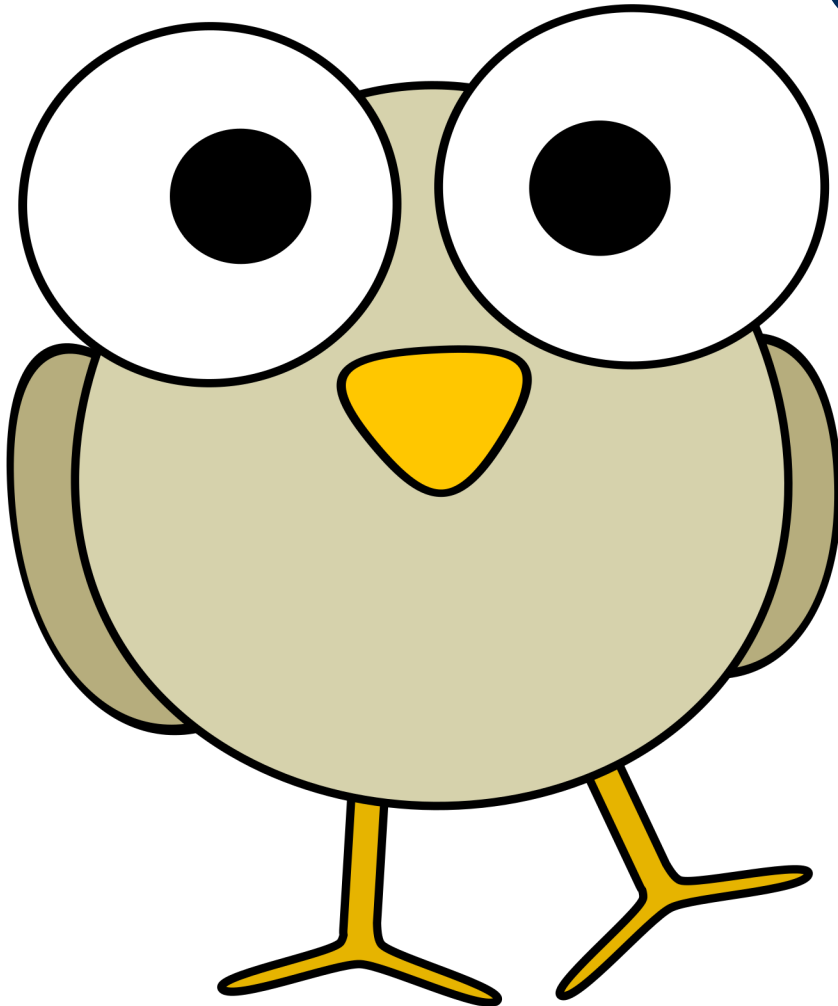
// decrements enrollment by 1

# Previously in CSCI 200



```
class Course {
public:
    Course() {
        _enrollment = 0;
        _title = "CSM 101";
    }
    Course(const string TITLE) {
        _enrollment = 0;
        _title = TITLE;
    }
    string getTitle() { return _title; }
    int getEnrollment() { return _enrollment; }
    void registerStudent() { _enrollment++; }
    void withdrawStudent() { if(_enrollment > 0) _enrollment--; }
private:
    int _enrollment;
    string _title;
};
```

# Questions?



??

# Learning Outcomes For Today



- Construct a program that accesses an element in a vector, returns the length of a vector, changes the length of the vector, and other vector operations.
- Construct a program that accesses an element in a string, returns the length of a string, changes the length of the string, and other string operations.
- Compare and contrast Procedural Programming with Object-Oriented Programming
- Explain the following terms and how they are used (1) dot operator / member access operator (2) data member (3) scope resolution operator
- Discuss the concept of scope within and outside a class & struct

# On Tap For Today



- Collections of Objects
- Passing Objects to Functions
- Practice

# On Tap For Today



- Collections of Objects
- Passing Objects to Functions
- Practice

# Sample Class



```
class Course {
public:
    Course() {
        _enrollment = 0;
        _title = "CSM 101";
    }
    Course(const string TITLE) {
        _enrollment = 0;
        _title = TITLE;
    }
    string getTitle() { return _title; }
    int getEnrollment() { return _enrollment; }
    void registerStudent() { _enrollment++; }
    void withdrawStudent() { if(_enrollment > 0) _enrollment--; }
private:
    int _enrollment;
    string _title;
};
```



# Consider V1



```
vector<Course> courseCatalog;
courseCatalog.push_back( Course() );
courseCatalog.push_back( Course("CSCI 200") );

// enroll students
for(size_t i = 0; i < courseCatalog.size(); i++) {
    courseCatalog.at(i).registerStudent();
    courseCatalog.at(i).registerStudent();
}

// print enrollments
for(size_t i = 0; i < courseCatalog.size(); i++) {
    cout << courseCatalog.at(i).getTitle() << " "
         << courseCatalog.at(i).getEnrollment() << endl;
}

// what does it print?
```

# Consider V2



```
vector<Course> courseCatalog;
courseCatalog.push_back( Course() );
courseCatalog.push_back( Course("CSCI 200") );

// enroll students
for(size_t i = 0; i < courseCatalog.size(); i++) {
    Course currentCourse = courseCatalog.at(i);
    currentCourse.registerStudent();
    currentCourse.registerStudent();
}

// print enrollments
for(size_t i = 0; i < courseCatalog.size(); i++) {
    cout << courseCatalog.at(i).getTitle() << " "
         << courseCatalog.at(i).getEnrollment() << endl;
}

// what does it print?
```

# Storing Objects on the Free Store



- Use a pointer!

```
int *pNumCars = new int;           // *pNumCars initialized to 0
int *pNumCars2 = new int(5);       // *pNumCars2 initialized to 5
```

```
Course *pCSM101 = new Course; // automatically calls constructor
Course *pCSCI200 = new Course("CSCI 200");
```

- **new** – “Computer, allocate enough memory in the free store for one object and tell me the starting address where the object will be stored. Initialize the object at that location.”

# Access Members of Object Pointers



- Must first dereference pointer before accessing

```
Course *pCSCI200 = new Course("CSCI 200");  
(*pCSCI200).getEnrollment();
```

- But this gets ugly when members return pointers

```
vector<Course*> *pCourses = new vector<Course*>;  
(*(*pCourses).at(0)).getEnrollment();
```

# Use the Arrow Operator



- Dereference and access in one operation

```
Course *pCSCI200 = new Course("CSCI 200");  
pCSCI200->getEnrollment();
```

- Much cleaner interface and denotes what type of thing we are working with at each level

```
vector<Course*> courses;  
courses.at(0)->getEnrollment();
```

# Precedence Table

Precedence	Operator	Associativity
1	Parenthesis: ( )	Innermost First
2	Postfix Unary Operators: a++ a-- a. p-> f()	Left to Right
3	Prefix Unary Operators: ++a --a +a -a !a (type)a &a *p new delete	Right to Left
4	Binary Operators: a*b a/b a%b	Left to Right
5	Binary Operators: a+b a-b	
6	Relational Operators: a<b a>b a<=b a>=b	
7	Relational Operators: a==b a!=b	
8	Logical Operators: a&&b	
9	Logical Operators: a  b	
10	Assignment Operators: a=b a+=b a-=b a*=b a/=b a%=b	Right to Left

# Consider V2 - solved



```
vector<Course*> courseCatalog;
courseCatalog.push_back( new Course() );
courseCatalog.push_back( new Course("CSCI 200") );

// enroll students
for(size_t i = 0; i < courseCatalog.size(); i++) {
    Course* pCurrentCourse = courseCatalog.at(i);
    pCurrentCourse->registerStudent();
    pCurrentCourse->registerStudent();
}

// print enrollments
for(size_t i = 0; i < courseCatalog.size(); i++) {
    cout << courseCatalog.at(i)->getTitle() << " "
         << courseCatalog.at(i)->getEnrollment() << endl;
}

// what does it print?
```

# On Tap For Today



- Collections of Objects
- Passing Objects to Functions
- Practice



# Passing Vectors/Strings to Function?



- Like any other single value: PBV or PBP

```
void print_vector_b_v( vector<int> vec ) {  
    for( int i = 0; i < vec.size(); i++ )  
        cout << vec.at(i) << endl;  
}  
  
void print_vector_b_p( vector<int> *pVec ) {  
    for( int i = 0; i < pVec->size(); i++ )  
        cout << pVec->at(i) << endl;  
}
```

- Be aware of PBV / PBP implications.  
Concerns?

# Passing Vectors to Function



- Like any other single value: PBV or PBP

```
void add_to_vector_b_v( vector<int> vec ) {  
    vec.push_back( 100 );  
}  
  
void add_to_vector_b_p( vector<int> *pVec ) {  
    pVec->push_back( 200 );  
}  
  
// ...  
  
vector<int> myVec;  
add_to_vector_b_v( myVec ); cout << myVec.size() << endl;  
add_to_vector_b_p( &myVec ); cout << myVec.size() << endl;
```

# String Parameter Beware



```
void string_func1( string str ) {...}
void string_func2( string *pStr ) {...}

...

string word = "does this work?";
string_func1( word );
string_func2( &word );
string_func1( "does this work?" );
string_func2( "does this work?" );
```

# String Parameter Beware



```
void string_func1( string str ) {...}
```

```
void string_func2( string *pStr ) {...}
```

```
...
```

```
string word = "does this work?";
```

```
string_func1( word ); // YES
```

```
string_func2( &word ); // YES
```

```
string_func1( "does this work?" ); // YES
```

```
string_func2( "does this work?" ); // NO!
```

# How Much Memory Used?



```
void string_func1( string str ) {...}
```

```
void string_func2( string *pStr ) {...}
```

```
...
```

```
string word = "does this work?";
```

```
string_func1( word );
```

```
string_func2( &word );
```

```
string_func1( "does this work?" );
```

# How Much Memory Used?



```
void string_func1( string str ) {...}
```

```
void string_func2( string *pStr ) {...}
```

```
...
```

```
string word = "does this work?";
```

```
string_func1( word );           // 2 copies
```

```
string_func2( &word );         // 1 copy
```

```
string_func1( "does this work?" ); // 1 copy
```

# When To Use PBV or PBP?



```
void string_func1( string str ) {...}
```

```
void string_func2( string *pStr ) {...}
```

```
...
```

```
string word = "does this work?";
```

```
string_func1( word );           // 2 copies
```

```
string_func2( &word );         // 1 copy
```

```
string_func1( "does this work?" ); // 1 copy
```

# How About Vectors?



- Memory Usage? Runtime?

```
void vector_func1( vector<int> vec ) {...}
```

```
void vector_func2( vector<int> *pVec ) {...}
```

```
...
```

```
vector<int> numbers = {...};
```

```
vector_func1( numbers );
```

```
vector_func2( &numbers );
```



# How About Vectors?



- Memory Usage? Runtime?

```
void vector_func1( vector<int> vec ) { ... }
```

```
void vector_func2( vector<int> *pVec ) { ... }
```

...

```
vector<int> numbers = { ... };
```

```
vector_func1( numbers );           // 2 copies - O(n)
```

```
vector_func2( &numbers );         // 1 copy - O(1)
```

# Vector Operations



- Element Access –  $O(1)$
- Vector Traversal –  $O(n)$
- (Will continue to add to)

# How About Vectors?



- Use PBP

```
// RW
```

```
void vector_func1( vector<int> *pVec ) {...}
```

```
// R only
```

```
void vector_func2( const vector<int> * const P_VEC )  
{...}
```

```
...
```

```
vector<int> numbers = {...};
```

```
vector_func1( &numbers );           // 1 copy - O(1)
```

```
vector_func2( &numbers );           // 1 copy - O(1)
```

# On Tap For Today



- Collections of Objects
- Passing Objects to Functions
- Practice

# To Do For Next Time



- Quiz 3 on Monday
  - File I/O + vector & string
- Set3 due Tuesday
- Final Project Proposal due Wednesday