CS 162
Intro to Programming II

Big 3
Notation

- The -> operator is used on pointers:
  - `Student *s1 = new Student("Bob");
    s1->getName();`
- If s were a regular non-pointer variable, you would use the . Operator
  - `Student s2("Bob");
    s2.getName();`
- The -> operator is equivalent to
  - `(*s1).getName();`
Notation

• The this operator refers to the calling object
eg. suppose I have a Student class

```cpp
class Student {
    public:
        std::string getName();
        void print();
    private:
        std::string name;
};
```
Example

• You can use *this* as follows:

```cpp
void Student::print() {
  /* Not good style but here to illustrate a point */
  std::cout << this->getName() << std::endl;
}

void Student::print() {
  /* Not good style but here to illustrate a point */
  std::cout << this->name << std::endl;
}
```
The Big Three

• In a class with complex objects (i.e. not using just standard data types) you should define all 3 of these:

1. Overloaded ‘==’ operator
2. Copy constructor
3. Destructor
Example

- intArray.hpp
  
  class IntArray {
  
  public:
    IntArray(int sizeValue, int defaultValue);
    IntArray(const IntArray& other);
    IntArray& operator=(const IntArray& rightSide);
    void print();
    ~IntArray();
  
  private:
    int size;
    int *array;
  };
Example

intArray.cpp

IntArray::IntArray(int sizeValue, int defaultValue) {
    size = sizeValue;
    array = new int[size];
    for( int i = 0; i < size; i++ ) {
        array[i] = defaultValue;
    }
}

void IntArray::print() {
    for( int i = 0; i < size; i++ ) {
        std::cout << array[i] << std::endl;
    }
}
Copy Constructor

A copy constructor is called automatically when:

1. A class object is declared and initialized by another object of the same type given in parentheses eg.
   
   ```
   IntArray ia1(5,10);
   IntArray ia2(ia1);
   ```

2. When a function returns a value of the class type

3. Whenever an argument of the class type is passed as a call-by-value parameter
Copy Constructor

• If you don’t define a copy constructor, C++ will automatically generate one for you
• This simply copies the contents of member variables (shallow copy)
  – Memberwise assignment
  – Won’t work correctly with dynamic member variables
• If you create your own copy constructor, you can do a deep copy
Copy Constructor

• Suppose we execute the following code:
  
  ```
  IntArray ia1(5,10);
  IntArray ia2(ia1);
  ```

• Shallow copy: copies the array pointer

```
 ia1:array
```

```
 ia2:array
```

• Deep copy: creates a copy of the object pointed to. In this case, we create a second copy of the array.

```
 ia1:array
```

```
 ia2:array
```

```
 10 10 10 10 10
```
Example

Copy constructor in IntArray.cpp:

Why?

IntArray::IntArray(const IntArray &other) : size(other.size) {
  array = new int[size];
  for (int i = 0; i < size; i++) {
    array[i] = other.array[i];
  }
}

OSU Oregon State University
Overloaded Assignment Operator

• If you don’t define one, C++ will automatically generate one for you
• This simply copies the contents of member variables (shallow copy)
• For the IntArray class, it will copy the pointers of the array member variable.
Example

IntArray& IntArray::operator=(const IntArray& rightSide) {
  if( this == &rightSide ) {
    return *this;
  } else {
    if( array != NULL )
      delete [] array;
    array = new int[rightSide.size];
    size = rightSide.size;
    for( int i = 0; i < size; i++ ) {
      array[i] = rightSide.array[i];
    }
    return *this;
  }
}
Example

IntArray& IntArray::operator=(const IntArray& rightSide) {

  Returns a reference of the object of the same type. Allows you to do things like: (ia1 = ia2).print()

  Overloaded = operator must be a member of the class
Example

if( this == &rightSide ) {
  return *this;
} else {

This check is necessary to allow self-assignment, e.g. ia1 = ia1
Destructor

IntArray::~IntArray() {
    delete [] array;
}

Destructor starts with ~ (tilda)
Automatically called when an object is destroyed
Have no return type
Have no parameter list, which means-
    - can only have one
    - cannot be overloaded
Destructors

• A destructor is required when:
  • A member variable is dynamically allocated
    ```
    IntArray::IntArray(int sizeValue, int defaultValue) {
        ...
        array = new int[size]; ...
    }
    ```

• You **may** want one for special processing
  – e.g. saving data when exiting