More on C++

CS A250 – C++ Programming II

Topics

- A couple of topics we need to cover…
  - The big three
  - Static variables and functions
  - Virtual functions

The Big Three

The Big Three

- The so-called **Big Three** are:
  - **Copy constructor**
  - **Overloaded assignment operator**
  - **Destructor**

- If any of these is missing from your class, the compiler will create it
  - **But** they might **not** behave as you expected when you have **dynamic variables.**
  - Therefore, if you have **pointers** and the **new** operator, you need to **implement the big three**.

Copy Constructor

- A **copy constructor** is a constructor that has one **call-by-reference parameter** that is of the **same type** of the class
  - The parameter **must** be passed by **reference**
  - The parameter **must** have a **const** modifier

- The copy constructor is called ***automatically*** when
  - A function returns a **value** of the class type
  - An argument is plugged in for a call-by-value parameter of the class type.

Copy Constructor (cont.)

- How to use it:

```
DArray myArray(20);

for (int i = 0; i < 20; ++i)
      b.addElement(i);

DArray anotherArray(myArray);
      // anotherArray is initialized
      // by the copy constructor
```

CAUTION!

- The **overloaded assignment operator** works ***only*** if the object is declared as a **separate** statement:

```
Darray a1; // assume we inserted elements
Darray a2;
a2 = a1;
```

- If you have a single statement, the compiler will call the **copy constructor**:

```
Darray a1;   // assume we inserted elements
Darray a2 = a1;
```

- **Why?**

What Is a Copy Constructor?

- A **copy constructor** is an **overloaded** constructor
- This means that you need to initialize the object just the same as you would do with any other constructor.
- In the example

```
Darray a1;    // assume we inserted elements
Darray a2 = a1;
```

  - **a2** is never initialized by calling a constructor; therefore, the **copy constructor** will need to initialize the array.

Copy Constructor (cont.)

- A copy constructor should be defined so that the object being initialized becomes a **complete**, ***independent copy*** of its argument.

- How do you implement a copy constructor?

Copy Constructor (cont.)

- A copy constructor should be defined so that the object being initialized becomes a **complete**, ***independent copy*** of its argument.

- How do you implement a copy constructor?
  - Just as you would implement a default constructor, with the difference that the new object must have the same properties of the parameter object.

Copy Constructor (cont.)

- This is the **default constructor** of the class **DArray**:

```
DArray::DArray()
{
    capacity = CAPACITY;
    numOfElem = 0;
    a = new int[capacity];
}
```

Copy Constructor (cont.)

- This is the implementation of the **copy constructor**:

```
DArray::DArray(const DArray& otherArray)
{
    capacity = otherArray.capacity;
    numOfElem = otherArray.numOfElem;
    a = new int[capacity];
    for (int i = 0; i < numOfElem; ++i)
          a[i] = otherArray.a[i];
}
```

What If…

- What if you do **not** implement the copy constructor on a class that uses pointers?
  - Assume you have:

    ```
    DArray anotherArray(myArray);
    ```

  - A **default** copy constructor will be automatically called.
  - This default copy constructor will ***not*** make a copy of the array, but will make a copy of the object that contains a pointer to the array.
  - When the function call ends, the destructor will be called automatically and will destroy the array (`myArray`).

Overloaded Assignment Operator

- We have seen how to overload the assignment operator, so let's review all **key factors**:

  - A **default overloaded assignment operator** is available, **BUT**
    - If **dynamic variables** are used, you should overload the assignment operator.

Overloaded Assignment Operator

- We have seen how to overload the assignment operator, so let's review all **key factors**:

  - Must be a **member of the class**

    - **Cannot** be a non-member and **cannot** be a friend.

Overloaded Assignment Operator

- We have seen how to overload the assignment operator, so let's review all **key factors**:
  - It is important to ***prevent*** **self assignment**
    - This is to avoid that the **operator =** deletes the dynamic memory associated with the object before the assignment is completed.
    - This would lead to "*fatal runtime errors*."

DESTRUCTORS

- A **destructor** is called automatically when an object of the class passes out of scope.
- You need to always make sure that your destructor deletes any **dynamically-allocated variables** and any **static variables**.
  - If you create a **dynamic variable *v*** and a **dynamic array *a***, you need to include in the destructor:

```
delete v;
delete [ ] a;
```

Note: If you forget **[ ]** when deleting the array, you will **only** be deleting the *first* element in the array.

Example 2

- Project: The Big Three

Static Variables and Functions

Topic 1: Static Class Members

- A **static class member** can be shared by **any object** the class
  - Represents a class-wide information shared by all instances (*objects*) of the class, not just a specific instance (*object*)
- Three types of class members:
  - **static variable**
  - **static constant**
  - **static function**

Static Member Variable

- A **static member variable** is a variable that contains data shared by ***all*** objects of a class
  - This is different from ***member variables***, which have **<u>distinct</u>** data for **<u>each</u>** object
- All objects of class "share" one copy
  - If one object changes it → all other objects will see the change
- Useful for "tracking"
  - How many objects exist at a given time
  - How often a member function is called

Static Variables - Example

- Suppose you have a **video game** that contains a class that creates **spaceships** (objects of the class)
  - When your class creates a spaceship, the spaceship appears on the screen
  - Every time there are more than 20 spaceships, your weapon changes from a one-directional laser to a radial laser
  - How does your game keep track of how many spaceships (objects) have been created?

Static Variables – Example (cont.)

- If we have a **static variable** that stores the number of objects created
  - Every time we create a spaceship, we can increment the static variable
- This requires **less memory**
  - Instead of having each object keeping track of how many objects were created
    - *Think about how redundant this implementation would be!*

Declaring a Static Variable

- How and where do you implement them?
  - **Declaration** is in the **private** section of the class definition, preceded by the keyword **static**

```
private:
    static int count;
    //other member variables
```

  - **Initialization** is in the **implementation** file, ***before*** all functions
    - **No** need to write the keyword **static**

```
int ClassName::count = 0;
```

Note: You need to **re-declare** the variable.

Scope of Static Variables

- Although they may seem like *global variables*, a class's **static variable** has **class scope**
- Can be declared
  - **`private`**
  - **`public`**
  - **`protected`**
- Needs to be updated in the **destructor**
  - And in any other function where it is required.

Static Constant

- A **constant** can also be **static**

```
const static double INTEREST = 0.3;
```

- Can be **declared** *and* **initialized** in the **class interface** (.h file)

Accessing Static Class Members

- From the **class**
  - ***All*** **static members** can be accessed <u>directly</u> (just like any other member variable)

- From **outside the class**
  - You will need an **accessor function** that is also **static**

Static Member Functions

- A **member functions** that **returns** a static variable must be **static**
  - Declaration

```
static int getCount();
```

  - Definition

```
int ClassName::getCount()
{
        return count;
}
```

Static Member Functions

- A **member functions** that **returns** a static variable must be **static**
  - Declaration

```
static int getCount();
```

Need keyword "static" needed

  - Definition

```
int ClassName::getCount()
{
        return count;
}
```

No keyword "static" needed

Static Member Functions

- A **member functions** that **returns** a static variable must be **static**
  - Declaration

```
static int getCount();
```

Need keyword "static" needed

Cannot be a const function

  - Definition

```
int ClassName::getCount()
{
        return count;
}
```

No keyword "static" needed

Static Member Functions

- A **static member functions** can be called ***outside*** the class in two different ways:
  - If **no** objects were created

```
ClassName::staticFunctionName()
```

  - If objects were created, you can use ***any*** object

```
objName.staticFunctionName()
```

EXAMPLE 1

- Project: CourseType Class

Virtual Functions

Virtual Functions Basics

- **Polymorphism**
  - Associating many meanings to one function
    - Values of different data types handled by using a uniform interface
  - Fundamental principle of **object-oriented programming**
  - Virtual functions provide this capability
- **Virtual**
  - Existing in "essence" though not in fact
- **Virtual Function**
  - Can be "used" before it is "defined"

Virtual Functions

- Classes for several kinds of shapes
  - Rectangles, circles, ovals, etc.
  - Each shape is an object of a different class
    - **Rectangle data:** height, width, center point
    - **Circle data:** radius, center point
- All derive from one parent class → Shape
- Require function for all classes: `draw()`
  - Different instructions for each shape

Virtual Functions (cont.)

- Each class needs a different *draw* function
- Can be called "draw" in each class, so:

```
Rectangle r;
Circle c;
r.draw();  //calls Rectangle class's draw
c.draw();  //calls Circle class's draw
```

- Nothing new here yet…

Virtual Functions (cont.)

```
class Shape
{
public:
    Shape();
    void draw() const;
    void center() const;

    ~Shape();
}
```

Function **center** moves a shape to the center of the screen.
First erases what is on the screen, and then re-draws the shape using the function **draw**.
All children will **inherit** the function **center**.

Complications: Which **draw** function to use? From which class?

```
class Circle: public Shape
{
public:
    Circle();
    void draw() const;
    ~Circle();
private:
    double radius;
}
```

```
class Rectangle: public Shape
{
public:
    Rectangle();
    void draw() const;
    ~Rectangle();
private:
    double height;
    double width;
}
```

Virtual Functions (cont.)

- Consider a new kind of shape comes along:

  `Triangle class`

  derived from `Shape` class
- Function `center()` inherited from `Shape`
  - Will it work for triangles?
  - It uses `draw()`, which is different for each shape!
  - It will use `Shape::draw()` → will ***not*** work for triangles
- Want inherited function `center()` to use function `Shape::draw()` *not* function `Shape::draw()`
  - But class `Triangle` was not even written when `Shape::center()` was! Does not know "triangles"!

Virtual Functions (cont.)

- **Virtual functions** are the answer

- Tell compiler:
  - "Don’t know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance“

- Called **late binding** or **dynamic binding**
  - Virtual functions implement late binding

Overriding

- When a **virtual function** ***definition*** is changed in a **derived class**
  - We say it is been "**overridden**"
  - Similar to ***redefined***

- So:
  - Virtual functions are ***overridden***
  - **Non**-virtual functions are ***redefined***

Virtual Functions: Why Not All?

- One major ***disadvantage***: **overhead**
  - Uses ***more*** storage
  - Late binding is "on the fly", so programs run slower.
- So if virtual functions are not needed, they should not be used.

Pure Virtual Functions

- Base class might not have "meaningful" definition
  - Its purpose solely for others to derive from

- Recall class **`Shape`**
  - All shapes are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Shape has no idea how to draw!

- Make it a ***pure* virtual function**:

```
virtual void draw() = 0;
```

Abstract Base Classes

- **Pure virtual functions** require **no** definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is an **abstract base class**
  - Can ***only*** be used as base class
  - No objects can ever be created from it
    - Since it does not have complete "definitions" of all its members

Virtual Destructors

- Recall:
  - **Destructors** are automatically executed when the class object goes out of scope
- Now consider:
  - If we pass the **derived** object to the **non-member** function print as type **base** class, when the object is destroyed, the **destructor** of the **base** class executes regardless of whether the derived class object is passed by reference or by value.
  - Logically, you would think that the **destructor** of the **derived** class is also executed when the class object goes out of scope.
    - Correct?

Virtual Destructors (cont.)

- No, it is not correct. The **destructor** of the derived class will ***not*** be executed.
- To correct the problem:
  - The **destructor** of the base class must be **virtual**.
  - The virtual destructor of a base class automatically makes the destructor of a derived class be virtual so that it can also be executed when the object is out of scope.
    - The derived class destructor will be executed first, then the base class destructor will be executed.

Virtual Destructors

- Any class that includes ***at least one*** **virtual member function** should define a **virtual destructor**

- If you are using inheritance, it is a good idea to have the **destructor** of the base class declared as **virtual**

SUMMARY

- **Late binding** delays decision of which member function is called until **runtime**
  - In C++, virtual functions use **late binding**
- **Pure virtual functions** have no definition
  - Classes with at least one are **abstract**
  - **No** objects can be created from abstract class
  - Used ***strictly*** as base for others to derive
- Make **all destructors virtual**
  - Good programming practice
  - Ensures memory correctly de-allocated

Virtual Functions - Examples

- These examples have walk-through explanations that are easy to follow in the project instead of having them in the slides:
  - Virtual_1
  - Virtual_2
  - Virtual_3
  - Virtual_4
  - Virtual_5

More on C++ (end)