This week

- Lab Solution – Anagrams
  - To be posted
- Memory Management - Destructors
- Inheritance
  - Base Class, Sub-Class
- Polymorphism
  - Flexibility through pointers
- Abstract Classes
Example Problem

- Lets create a new “container” class called Collection
  - Stores integers
  - Constructor specifies the maximum size
  - Add
  - Resize (new size)

- Recall – we’ve done something similar to this before…
Problem: Memory Leak?

- We have a problem - when we construct our Collection we allocated using new
- This memory will **never** be reclaimed
  - If we are just using in main, no big deal...
  - If we had collections in other functions, this is a major problem
Solution: Destructor

- A constructor is automatically called when an instance of your class is being created.
- A destructor is automatically called when an instance is being reclaimed:
  - Stack instances going out of scope
  - Delete called on heap instances

```c
~List() {
  if ( contents != NULL ) delete contents;
}
```
- Object-Oriented programming allows for classes to **extend** other classes
  
  - Other terminology:
    - Derived classes extend base classes
    - Derived classes extend **parent** classes
    - Inheritance allows **is a** relationships

Base Class: Shape

Derived Class: Circle

Derived Class: Rectangle
Derived Classes

- Derived classes extend base classes by adding properties and/or functions.
  - Example: Shape
    - Property: Color [string]
    - Property: Filled [bool] (for drawing)
    - String getColor(), void setColor(string)
    - Bool isFilled(), void setFilled(bool)
    - String toString() -> “A solid red shape”
  - Circle extends Shape (Circle is a Shape)
    - Property: Radius
    - Double get/set Radius, getArea(), getPerimeter()
class Shape {
  public:
    Shape();
    Shape(string color, bool filled);
    string getColor();
    void setColor(string c);
    bool isFilled();
    void setFilled(bool f);
    string toString();
  private:
    string color;
    bool filled;
};

class Circle : public Shape {
  public:
    Circle();
    Circle(double radius);
    Circle(double radius, string color, bool filled);
    double getRadius();
    void setRadius(double r);
    double getArea();
    double getPerimeter();
    double getDiameter();
  private:
    double radius;
};

int main() {
    Circle c(5, "white", true);
    cout << c.getColor() << endl;
}
Many classes can extend a common base

```cpp
class Rectangle : public Shape {
public:
    Rectangle();
    Rectangle(double width, height);
    double getWidth();
    void setWidth();
    double getHeight();
    void setHeight();
    double getArea();
    double getPerimeter();
private:
    double width;
    double height;
};
```
**Constructors**

- A Circle is a Shape, so it makes sense that when you create a Circle, you also create a Shape...
- By default, the default Constructor of the base class is called right before the code in the derived class

```cpp
class Shape {
public:
    Shape() {
        color = "white";
        filled = false;
    }
    ...
}
class Circle : public Shape {
public:
    Circle(double r) {
        radius = r;
    }
    ...
}
```

If shape doesn’t have a default constructor… compiler error!!!
Constructors

- You can call *specific* base constructors using very special syntax...

```cpp
Circle(double radius, string color, bool filled) : Shape(color, filled) {
    radius = 1;
}
```
Destructors

- Base Class constructors are **always** called before their sub-classes

- Base Class **destructors** are **always** called after their sub-classes
Object inheritance allows us to write functions that accept “generic” base classes.

```cpp
void printShape(Shape s)
    { cout << s.toString() << endl; }

int main()
    { Shape s;
      Circle c(1, "black", false);
      Rectangle(r(3, 4, "red", true);
      printShape(s); printShape(c); printShape(r); }
```

It works because Circle is a Shape and Rectangle is a Shape.
The Shape class’s `toString` doesn’t have any dimensions (radius, width, etc.)

- You can declare `toString` methods in the derive class to “override” the default behavior

```cpp
class Shape {
public:
  ...
string toString() {
  stringstream ss;
  ss << "A " << getColor();
  if ( isFilled() ) ss << " solid ";
  else ss << " outlined ";
  ss << "shape."
  return ss.str();
}

class Circle : public Shape {
public:
  string toString() {
    stringstream ss;
    ss << "A " << getColor();
    if ( isFilled() ) ss << " solid ";
    else ss << " outlined ";
    ss << "circle with radius = " << getRadius();
    return ss.str();
  }
}

Shape s;
Circle c(2);
cout << s.toString() << " " << c.toString() << endl;
```
Keyword: protected

- When a derived class extends a base type, it has access only to the public functions and methods of its base.

```cpp
class Circle : public Shape {
public:
  string toString() {
    stringstream ss;
    ss << "A " << color;
    if (isFilled()) ss << " solid ";
    else ss << " outlined ";
    ss << " circle with radius = " << getRadius();
    return ss.str();
  }
}
```

Compiler error: Circle cannot access private data within Shape.
Keyword: protected

- There are some situations where the base class has good reason to limit access to its data.
- However often children (derived classes) should be allowed...
- To resolve this, we use “protected” rather than “private”.
- Protected data is still hidden from code outside of the class, but it is accessible within derived classes.
Limitations to Refinement

When using base classes in functions, C++ can only do so much:

```cpp
void printShape(Shape s) {
  cout << s.toString() << endl;
}

int main() {
  Shape s;
  Circle c(1, "black", false);
  Rectangle(r(3, 4, "red", true);
  printShape(s);
  printShape(c);
  printShape(r);
}
```

- At runtime, `printShape` will think `s`, `c`, and `r` are just ordinary “shapes”, and use `Shape`’s `toString()`
- C++ lacks “dynamic” type checking in this situation
Polymorphism

- The concept of polymorphism takes "refinement" to a more powerful level.

- Polymorphism will allow a reference/pointer to a base class to work intelligently when pointing to derived types.

- We will need some additional syntax however…
For a method to participate in polymorphism, it must be marked as `virtual` in the base class’s definition.

class Shape {
public:
  virtual string toString() {
    stringstream ss;
    ss << "A " << getColor();
    if (isFilled()) ss << " solid ";
    else ss << " outlined ";
    ss << "shape.";
    return ss.ToString();
  }
}

class Cicle {
public:
  ... string toString() {
    stringstream ss;
    ss << "A " << getColor();
    if (isFilled()) ss << " solid ";
    else ss << " outlined ";
    ss << "circle with radius = " << getRadius();
    return ss.ToString();
  }
}
Polymorphism with Pointers

- Polymorphism works when using pass-by-reference or pointers.
- When a function takes a reference to a base type as a parameter, calls on the passed object will map to the derived type.

```cpp
void printShape(Shape & s) {
    cout << s.toString() << endl;
}

int main() {
    Shape s;
    Circle c(1, "black", false);
    Rectangle(r(3, 4, "red", true);
    printShape(s);
    printShape(c);
    printShape(r);
    
    At runtime, `printShape` call the toString function on Shape for s, Circle for c, and Rectangle for r.
```
More abstraction

- Notice that Rectangle and Circle have some common methods (behaviors)
  - getArea()
  - getPerimeter()

- While all shapes have areas and perimeters, we cannot move those functions into the Shape class… why?
More abstraction

- Thinking carefully - it might not even make much sense to ever instantiate a “Shape”… there is no such thing!
  - Shape is a “generic” term for a set of real things.
  - Shape is considered “abstract” - it’s not “real”

- Although one cannot calculate the area or perimeter of a “shape”, we know that it should be possible to do so...

```cpp
void printAreaToPerimeterRatio(Shape * s) {
  cout << “The ratio of area to perimeter is”
  << s->getArea() / s->getPerimeter() << endl;
}
```
Abstract Classes

- An abstract class represents a “generic” thing, that cannot be used directly:
  - It defines “pure” virtual functions, with no implementation
  - All classes that derive from the abstract class must provide a full implementation of all pure virtual functions
  - Your abstract class defines an interface for using a bunch of different types of objects...

- Example: A shape must have an area and perimeter, but its up to Circle and Rectangle to figure it out...
Abstract Classes

class Shape {
public:
...
    virtual double getPerimeter() = 0;
    virtual double getArea() = 0;
...

class Circle : public Shape {
public:
    double getPerimeter() {
        return 2 * PI * radius;
    }
    double getArea() {
        return PI * radius * radius;
    }
};

class Rectangle : public Shape {
public:
    double getPerimeter() {
        return 2 * height * width;
    }
    double getArea() {
        return height * width;
    }
};
Abstract Classes

```cpp
void printAreaToPerimeterRatio(Shape & s) {
    cout << "The ratio of area to perimeter is" << s.getArea() / s.getPerimeter() << endl;
}

int main() {
    Shape s;
    Circle c(2);
    Rectangle r(4, 5);
    printAreaToPerimeterRatio(c);
    printAreaToPerimeterRatio(r);
}
```

X - compiler error, cannot instantiate abstract class
Lab 8 – Complete at home

- Create a Triangle class which extends Shape
- Use the same functions as in main
  - make sure you can create instances
  - call the print shape method

- Triangle can be assumed to be a right triangle, which means the area = \( \frac{1}{2} \text{ base} \times \text{ height} \).