INHERITANCE - REVIEW
Derived Classes

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;
}
Generic functions

- Functions can declare parameters of the base type, but you can actually pass any instance derived from the base!

```cpp
void doSomething(Shape s);
Shape s;
Circle c;
doSomething(s);  // fine..
doSomething(c);  // also ok!
```
Refinement

- Derived classes may implement methods already defined in the base class.
- The derived class’s version is called whenever the compiler knows the instance is of the derived type.
- Shape s;
- Circle c;
- c.toString() will map to Circle’s version
- s.toString() will still map to Shape’s
void doSomething(Shape s) {
    cout << s.toString() << endl;
}

Circle c;
cout << c.toString() << endl;
doSomething(c);

Will use Circle’s version
Will use Shape’s version, even though we passed in a Circle
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.

```cpp
class Shape {
public:

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

class Circle {
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 References

- 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.
Pointers and Objects

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);
}

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);
}
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

```cpp
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;
  }
};
```
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 = \( \frac{1}{2} \) base * height.
Inheritance, Polymorphism, Abstract?

- Inheritance means a derived class **borrows** implementation and member variables from **base**
  - A derived class will often add members
  - Sometimes, a derived class **refines** some of the base’s methods

- Sometimes, a base class can mark one (or more) of its functions as **virtual** – meaning the method can be **overridden** and participate in **polymorphism**.
  - If the base doesn’t override the function, no polymorphism!
  - Polymorphism only works with references/pointers

- Sometimes the base class shouldn’t even be instantiated.
  - **You know all Shapes** should have **getArea()**
  - **Abstract class!**
The string class also works with many standard C++ operators:

- +
- [...]  
- << and >>

Can our own classes do this?

```cpp
#include <iostream>
#include <sstream>
using namespace std;

int main() {
    string s1("hello");
    string s2("world");
    string s3 = s1 + " " + s2;
    cout << s3 << endl;
    cout << s3[1] << endl;
}

> hello world
> e
Operator Overloading

- Overloading refers to the ability to add extra functionality to standard C++ operators.
- All of the following operators can be “overloaded” to work with your own types:

  ```
  +   -   *   /   %
  +=  -=  *=  /=  %=
  ++  --
  ^   &   |   ~   !
  =   ^=  &=  |=
  <   >   <=  >=  ==  !=
  <<  >>  &&  ||
  ->*  ,   ->  []  ()  new  delete
  ```
Example: Rational

- A rational number is anything with a numerator and denominator (both integers)

```cpp
class Rational {
    public:
    Rational();
    ...

    private:
    int num;
    int den;
};
```

Rational numbers can be added/subtracted...

They can be compared...

They can be printed… etc.

The full class description is in the text and on appiversity.
int main() {
    Rational r1 (5, 6);
    Rational r2 (10, 12);
    if ( r1.equals(r2) )
        cout << "Equal!";
    Rational r3;
    r3 = r2.add(r1);
    r3.print();
}

int main() {
    Rational r1 (5, 6);
    Rational r2 (10, 12);
    if ( r1 == r2 )
        cout << "Equal!";
    Rational r3;
    r3 = r1 + r2;
    cout << r3 << endl;
}
Overloading Operators

- Overloading operators is great for the *user* of your class...
  - However... the syntax to define the overloading is tricky...
- You need to think of an operator *as a function* involving your class
There are several classifications of operators:

- **Relational** (<, >, ==, etc.)
  - Normal functions with 2 Rational parameters (left and right hand side).
  - Returns true or false
- **Mathematical** (+, -, *, /)
  - Normal functions with 2 rational parameters (left and right hand side).
  - Returns new instance of Rational
- **Combined Assignment** (+=, -=, *=, /=)
  - Member functions with single parameter. Returns same instance of Rational
- **I/O Operators** << and >>
  - friend functions with stream and rational parameters. Returns stream

<table>
<thead>
<tr>
<th>return value</th>
<th>function name</th>
<th>parameters</th>
</tr>
</thead>
</table>
class Rational {
public:
    Rational();
    int compareTo(Rational & other) {
        ...
    }
    ...
};

bool operator < (const Rational & r1, const Rational & r2) {
    return r1.compareTo(r2) < 0 ;
}
const 

- We’ve all seen const used in variable declarations – what about parameters?
  - It tells the compiler that you will NOT change the parameter’s value within the function
  - Why do we do this?
    - Notice that they are passed by reference...
Mathematical Operators

```cpp
class Rational {
public:
    Rational();
    Rational add(Rational & other) {
        // returns new Rational instance which is this + other
    }
    ...
};

Rational operator + (const Rational & r1, const Rational & r2) {
    return r1.add(r2);
}
```
More const “problems”

- When defining the + operator, compiler errors are generated when calling the add function.
  - This is because we haven’t “promised” the compiler add won’t change:
    - 1) its parameter (r2)
    - 2) the instance itself (r1)
  - `return type functionName(params) const`
class Rational {
public:
    Rational();
    Rational &operator += (const Rational & r2);
    ...
};

Rational & Rational:: operator += (const Rational & r2) {
    *this = this->add(r2);
    return *this;
}
class Rational {
public:
    Rational();
friend ostream &operator << (ostream &, const Rational &);
};

ostream &operator << (ostream & out, const Rational & r) {
    out << r.numerator << " / " << r.denominator << endl;
    return out;
}
Why friend?

- The «<< and »>> operators are defined in ostream and istream, so the Rational class can’t overload them from within the Rational class.
- The only reason we make it a friend, and not just a normal external functions (like mathematical operators) is so we can access private data.
- If you don’t need to access private data, then you don’t need to use friend.
Exam 2

- Next week – 11/24
- Open Book/Notes/Computer etc.
- Focused entirely on Object Oriented Programming
You must know the various vocabulary associated with object oriented programming:

- Class, Instance, Object
- Member variables, properties, functions, methods.
- Composition, “Has a” relationships
- Encapsulation with public/private/protected
- Inheritance -> parent/child, base/derived, specialization, polymorphism
Syntax

- Know how to construct classes and to split them between header files and implementation files.
- Know how to create constructors and **destructors**.
- Know how to derive from a base class.
- Know how to work with **pointers to objects**.
- Know what the **const** keyword means in **all of the locations we’ve seen!**
Using classes

- You must be comfortable working with the following built-in types:
  - string
  - stringstream
Polymorphism

You must understand all the degrees of inheritance:

- Simple overrides
- Virtual functions and polymorphism
- Abstract classes and pure virtual functions
Overloading Operators

- Know the 4 categories of overloading syntax
- Be able to create overloaded functions
- Be able to tell which operators need to be overloaded given code that uses a class.
Practice Problem: Time Class

- Holds the time of day
  - Single member variable – seconds since midnight..

- Prints as hours, minutes, seconds

- Can be set to print in 12 am/pm or 24 hour mode
  - Based on a variable

- Can be printed with `<<`

- Supports `+` and `-`, `+=` and `-=` based on seconds
  - We’ll skip `++` and `--`, they aren’t pretty…

- We’ll write some unit tests before creating the full class to help us work towards a correct solution.
Study Problem

- Write a function that accepts a string
  - The string will be a binary number (“10011”).
  - The function should return the integer the binary string represents.

- Write a main program that allows the user to enter a string (no more than 32 characters) and prints out the corresponding integer.
Study Problem

- Read the input as a normal string.
- Start at the right-most character.
- Set value = 0
- Set addend = 1
- Working right to left, for each character:
  - Check that the character is ‘1’ or ‘0’.
  - If its ‘1’, then add addend to value
  - Multiply addend by 2
  - If its not ‘1’ or ‘0’, throw an exception