This exercise is about inheritance. The most important thing that you must understand about inheritance is that it is used too often. Meaningful examples of inheritance are hierarchies of graphical objects, and file hierarchies. Inheritance should only be used when the following conditions are met:

1. some group of types have something (important) in common,
2. you want to be able to mix these types at run time, e.g. by putting them in the same container,
3. you don’t know in advance how many types there will be. (You want to be able to add more types later.) Put differently: Adding another type would not mess up all your algorithms.

It is probably a good idea to have a look at http://www.ii.uni.wroc.pl/~nivelle/teaching/object2011/inheritance.pdf first.

1. Consider the following hierarchy:

```cpp
class surf
{
    virtual double area( ) const = 0;
    virtual double circumference( ) const = 0;
    virtual surf* clone( ) const = 0;
    virtual print( std::ostream& ) const = 0;
    ~surf( );
};

class rectangle : public surf
{
    double x1, y1;
    double x2, y2;

    double area( ) const;
    double circumference( ) const;
```
rectangle* clone( ) const;
void print( std::ostream& ) const;
};

class triangle : public surf
{
    double x1, y1; // Positions of corners.
double x2, y2;
double x3, y3;

double area( ) const;
double circumference( ) const;
triangle* clone( ) const;
void print( std::ostream& ) const;
};

class circle : public surf
{
    double x; // Position of center.
double y;
double radius;

double area( ) const;
double circumference( ) const;
circle* clone( ) const;
void print( std::ostream& ) const;
}

Write suitable constructors for each of the subclasses, and implement the area( ) const, circumference( ) const, clone( ) const, and print( std::ostream& ) const methods.

2. We want to be able to put a mixture of rectangles, triangles, and circles in an std::vector in a robust way, without memory leakage.
   In order to do this, we must define a class

struct surface
{
    surf* ref;

    surface( const surface& s )
        : ref( s. ref -> clone( ))
    { }

    surface( const surf& s )
    {
        // Constructor
    }

    // Destructor

    void print( std::ostream& ) const;

    surface( )
    {
        // Default constructor
    }

    // Copy constructor

    // Assignment operator

    // Destructor

    surf* clone( ) const;

    // Equality operator

    // Comparison operators

};
void operator = ( const surface& s )
{
    if( ref != s.ref )
    {
        delete ref;
        ref = s.ref -> clone( );
    }
    // I am not in favour of making assignment
    // return something.
}

~surface( )
{
    delete ref;
}

const surf& getsurf() const { return *ref; }
    // There is no non-const method, because
    // changing would be dangerous.
};

Define a print function

std::ostream& operator << ( std::ostream& stream, const surface& s );

according to the pattern on the slides.

3. Fill an std::vector< surface > with a couple of surfaces, and make sure that the following functions work correctly

std::ostream& operator << ( std::ostream& stream,
    const std::vector< surface > & table )
{
    for( unsigned int i = 0; i < table.size(); ++ i )
    {
        stream << i << ",-th element = " << table[i] << "\n";
    }
    return stream;
}
4. Convince yourself, by writing a `for` loop, that there are no memory leaks.