

# Course C++, Exercise Number 11

Deadline: 09.06.2017

This exercise is about inheritance.

1. Consider the following hierarchy:

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

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

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

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

    double area( ) const override;
    double circumference( ) const override;
```

```

        triangle* clone( ) const & override;
        triangle* clone( ) && override;
        void print( std::ostream& ) const override;
};

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

    double area( ) const override;
    double circumference( ) const override;
    circle* clone( ) const & override;
    circle* clone( ) && override;

    void print( std::ostream& ) const override;
}

```

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

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 will construct a wrapper class.

```

struct surface
{
    surf* ref;

    surface( const surface& s );
    surface( surface&& s );

    surface( const surf& s );
    surface( surf&& s );

    void operator = ( const surface& s );
    void operator = ( surface&& s );
    void operator = ( const surf& s );
    void operator = ( surf&& s );

    ~surface( )
    {
        delete ref;
    }
}

```

```

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

```

Implement the methods.

3. Define a print function

```

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

```

following the pattern on the slides.

4. 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( size_t i = 0; i < table.size( ); ++ i )
    {
        stream << i << "-th element = " << table[i] << "\n";
    }
    return stream;
}

```

```

void print_statistics( const std::vector< surface > & table )
{
    double total_area = 0.0;
    double total_circumference = 0.0;
    for( const auto& s : table )
    {
        std::cout << "adding info about " << s << "\n";
        total_area += s.getsurf().area( );
        total_circumference += s.getsurf().circumference( );
    }

    std::cout << "total area is " << total_area << "\n";
    std::cout << "total circumference is " << total_circumference << "\n";
}

```

5. Convince yourself, by using **valgrind**, that there are no memory leaks. Be sure to include all methods of **surface** in the test.