# 6.S096 Lecture 5 – Object-Oriented C++

## Abstraction, Inheritance, STL

Andre Kessler

## Outline

## Assignment 2

### Linked list library (list)

- Writing linked list code that other programs can use
- No memory leaks!

### Minimum spanning tree (mst)

- Using the STL
- Implementing an algorithm
- Writing fast C++

### Rational numbers library (rational)

- Overloading arithmetic functions
- Edge cases
- Exceptions

## Crash course in the STL

### <**vector**>

Remember the Array class we were writing last time? A better version already exists in C++, so we don't need to write it ourselves!

```
#include <vector>
// In our code:
std::vector<int> intArray;
while( /* getting data */ ) {
  intArray.push_back( data );
}
int tenthItem = intArray[9]; // like an array
// Automatically destroys data when done
```

The angle brackets < and > let us specialize this type: we can replace `T` with any other type `T`, like `std::vector<T>`.

## Crash course in the STL

### STL? What's that?

STL stands for Standard Template Library. Containers and algorithms to use on those containers, all with a common interface. Another vector:

```
#include <vector>
#include <string>
// In our code:
std::vector<std::string> stringList;
stringList.push_back( "C99" );
stringList.push_back( "C++03" );
stringList.push_back( "C++11" );
for( auto str : stringList ) { // "range-for"
  std::cout << str << "\n";
}
```

## Some (BAD!) code that needs to be refactored

We've got a struct to hold some shape data for different ShapeTypes.

```
// BAD CODE!
enum ShapeType { CIRCLE = 0, SQUARE = 1,
                 RECTANGLE = 2, TRIANGLE = 3 };

// needs to be big enough to hold the shape
struct Shape {
  ShapeType type;
  double a, b, c, d;
};
```

And we want a function to compute the area, given a shape.

## Some (BAD!) code that needs to be refactored

Without some good object-oriented practices, code like this can become a tangled mess of switch or if/else statements.

```
// BAD CODE!
double area( const Shape &shape ) {
  switch( shape.type ) {
    case CIRCLE: return M_PI * shape.a * shape.a;
    case SQUARE: return shape.a * shape.a;
    case RECTANGLE: return shape.a * shape.b;
    case TRIANGLE: return 0.5 * shape.a * shape.b;
    default: std::cerr << "Error, invalid shape!\n";
  }
  return 0.0;
}
```

## What's so bad?

**Any time we want to make a function that works differently on different types of shapes, we need this same switch statement.**

**Lots of code repetition.**

**The member variable names do not describe their purpose.**

**Not extensible: when we add a new shape, we have to modify every one of these functions.**

## Our refactoring: an abstract class

- Let's create an abstract class Shape. We do this by giving the class some virtual functions; these are functions which child classes can override.
- Could have member variables or not (in this case, we won't)
- Pure virtual functions (the = 0).
- Notice the destructor is virtual.

```
class Shape {
public:
  virtual double area() const = 0; //pure virtual
  virtual ~Shape() {}
};
```

## Inheritance

```
class Shape {
public:
  virtual double area() const = 0; //pure virtual
  virtual ~Shape() {}
};
class Circle : public Shape {
  double _radius;
public:
  Circle( double theRadius ) : _radius{theRadius} {}
  ~Circle() {}
  inline double radius() const { return _radius; }
  double area() const { return
                               _radius * _radius * M_PI; }
};
```

## Closer look at the child class...

- Syntax is `class Derived : public Base`

```
class Circle : public Shape {
  double _radius;
public:
  Circle( double theRadius ) : _radius{theRadius} {}
  ~Circle() {}
  inline double radius() const { return _radius; }
  double area() const { return
                              _radius * _radius * M_PI; }
};
```

## Know the functions C++ automatically creates!

**Looks like a pretty emtpy class, right?**

```
class Empty{};
```

## Know the functions C++ automatically creates!

**Looks like a pretty emtpy class, right? Wrong!**

```
class Empty{
public:
  Empty() { /*...*/ } // constructor
  // copy constructor
  Empty( const Empty &rhs ) { /*...*/ }
  // copy assignment
  Empty& operator=( const Empty& rhs ) { /*...*/ }
  ~Empty() { /*...*/ } // destructor
};
```

If we don't want these functions, have to disallow by making them private and indicating `= delete`!

## Examples

**Some other tips from Scott Meyers:**

- Item 7: Declare destructors virtual in polymorphic classes.
- Item 10: Have assignment operators return a reference to `*this`.
- Item 12: Copy all parts of an object.
- Item 22: Declare data members private
- Item 32: Make sure public inheritance models “is-a”.

**Read his book, Effective C++!**

## Examples

**Let's see some examples...**

## Wrap-up & Wednesday

**Monday is a holiday!**

**Second assignment due Weds. at midnight**

**Class on Weds.**

- Design patterns and anti-patterns

**Questions?**

- Office hours Mon, Tues in 26-142