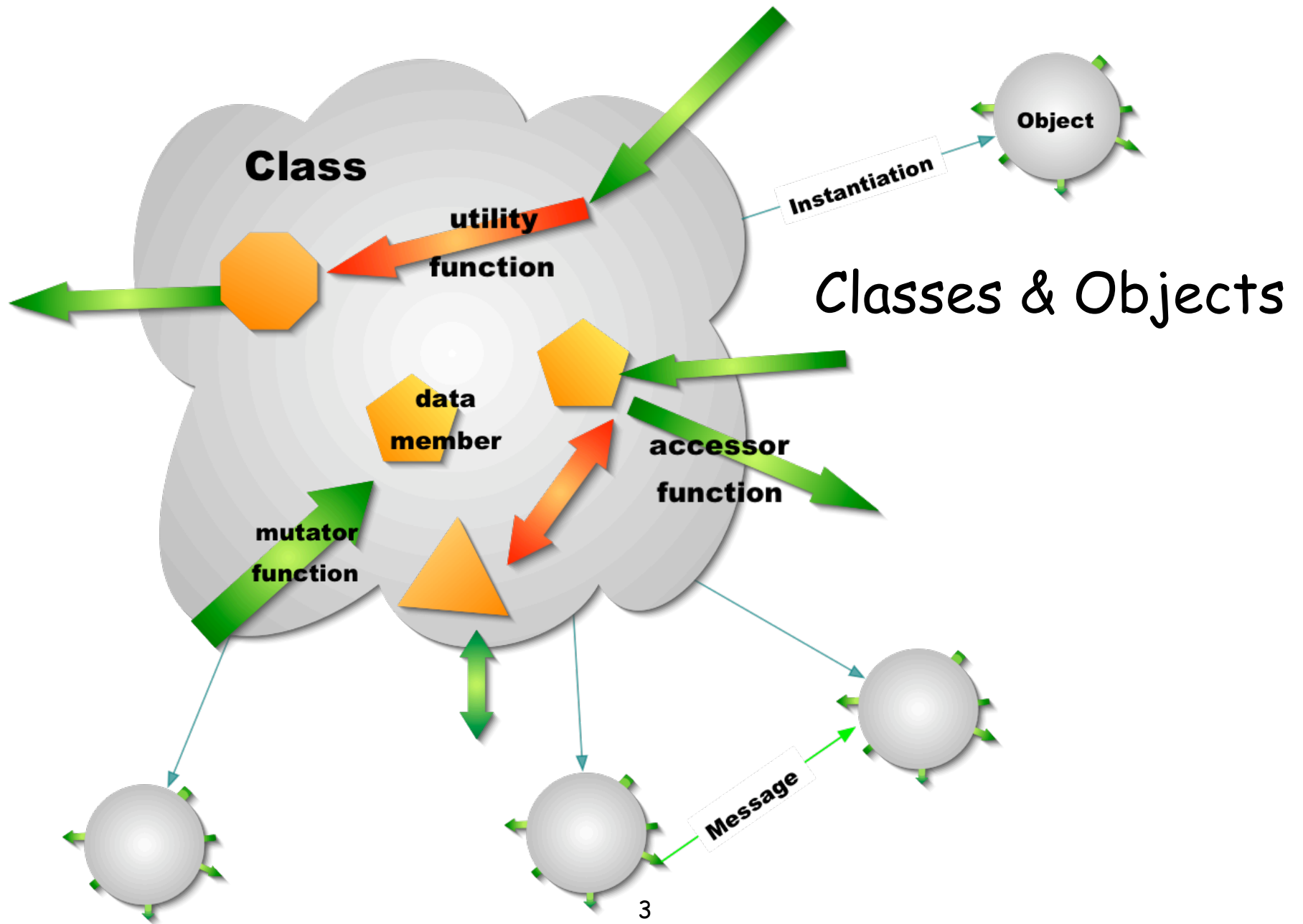


Ch 2

ADTs and C++ Classes

- Object Oriented Programming & Design
- Constructing Objects
- Hiding the Implementation
- Objects as Arguments and Return Values
- Operator Overloading

Object-Oriented Programming & Design



Some OOP Features of C++

- Accessor functions can be declared to be **const**.
- Methods (member functions) may be represented by prototypes in the class definition. Their definition may include the **inline** request.
- Methods defined within a class definition request **inline** implicitly.
- External method definition requires the scope resolution operator.

Constructor Properties and Rules

- Constructors are methods whose names are the same as the class'.
- They may not have a return type. Not even `void`.
- The default constructor (one with no parameters) is invoked with no argument list (no " ()").
- If no constructors are written, the default constructor is provided automatically. Otherwise not.
- The default constructor should be written if other constructors are defined.

Class Definition

```
class throttle {  
public:  
    throttle( );  
    throttle(int size);  
    void shut_off( ) {  
        position = 0; }  
    void shift(int amount);  
    double flow( ) const {  
        return position / double(top_position); }  
    bool is_on( ) const {  
        return (position > 0); }  
private:  
    int top_position;  
    int position;  
};
```

Class Implementation

```
throttle::throttle( ) {  
    top_position = 1;  
    position = 0;  
}  
throttle::throttle(int size) {  
    assert(size > 0);  
    top_position = size;  
    position = 0;  
}  
void throttle::shift(int amount) {  
    position += amount;  
    if (position < 0)  
        position = 0;  
    else if (position > top_position)  
        position = top_position;  
}
```

Client Code

```
#include <iostream>
#include <cstdlib>
#include "throttle.h"
using namespace std;

const int DEMO_SIZE = 5;

int main( ) {
    throttle sample(DEMO_SIZE);
    int user_input;

    cout << "I have a throttle with " << DEMO_SIZE << " positions." << endl;
    cout << "Where would you like to set the throttle?" << endl;
    cout << "Please type a number from 0 to " << DEMO_SIZE << ": ";

    cin >> user_input;
    sample.shift(user_input);

    while (sample.is_on( )) {
        cout << "The flow is now " << sample.flow( ) << endl;
        sample.shift( -1); }
    cout << "The flow is now off" << endl;

    return EXIT_SUCCESS;
}
```

Using `<ctime>`

Timing Processes

```
#include <ctime>
#include <iostream>
#include <cassert>
using namespace std;

long fib(long n);
void showTime(clock_t t);

int main() {
    clock_t start, stop;
    long f, n;

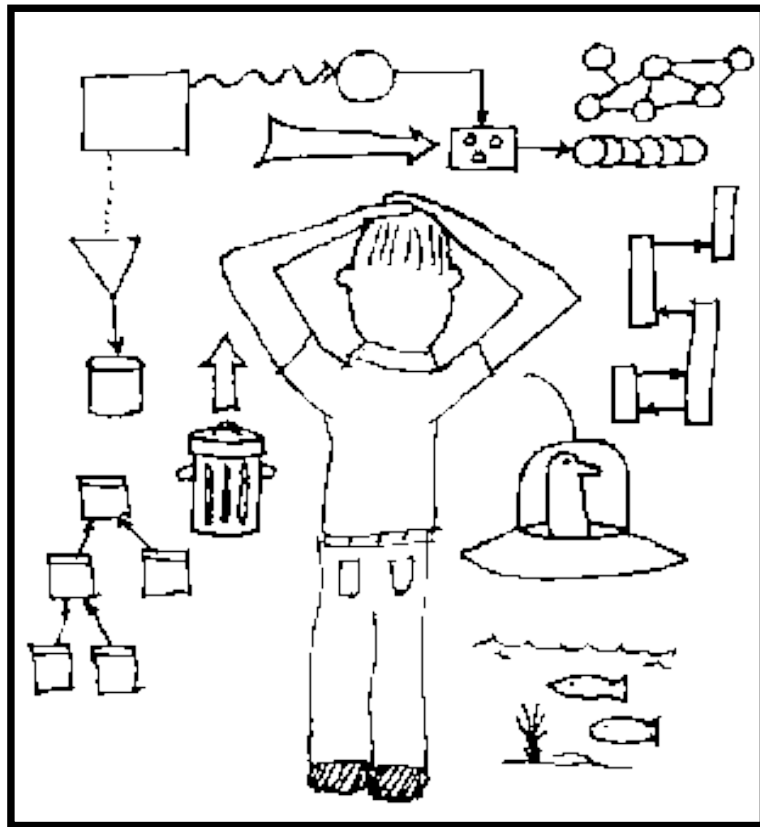
    do {
        cout << "Number: "; cin >> n;
        if (n > 0) {
            start = clock();
            f = fib(n);
            stop = clock();
            cout << "The " << n << "th Fibonacci Number is "
                << f << endl << endl;
            showTime(stop - start); } }
    while (n > 0);
}
```

Timing Processes, cont'd

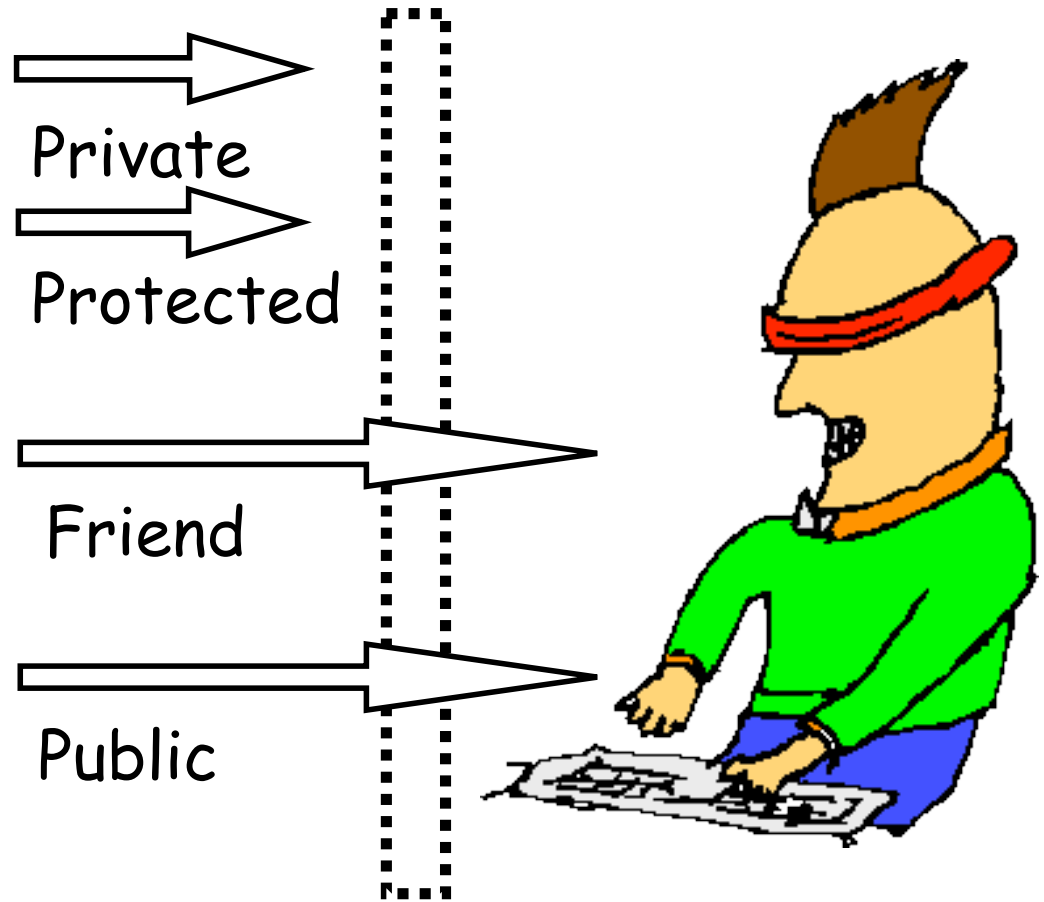
```
long fib(long n) {
    assert(n > 0);
    if (n > 2)
        return fib(n - 1) + fib(n - 2);
    else
        return 1;
}

void showTime(clock_t t) {
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(5);
    cout << "    time: "
         << (double)t/CLOCKS_PER_SEC << " sec" << endl
         << "    ticks: "
         << (long)t << endl
         << "ticks/sec: "
         << (long)CLOCKS_PER_SEC << endl << endl;
}
```

Hiding the Implementation



Designer



Client

The Interface File

- The class definition should be provided in a separated header file.
- Begin with the class documentation. Include a statement documenting the class' value semantics.
- Wrap the definition in
 - a preprocessor conditional to prevent duplicate inclusion
 - a namespace formed as
edu_santarosa_staff_gbrown_myClass
- Define only those methods intended to be inlined.
- Avoid **using** statements in the interface file.

Value Semantics

- The value semantics of a class determines how values are copied from one object to another.
- The assignment operator may be the automatic (default) one or overloaded.
- Likewise for the copy constructor. It is one which has a single parameter whose type is the same as the class. It is invoked automatically when
 - passing and returning objects,
 - cloning during instantiation as in
 - `throttle y(x);` or
 - `throttle y = x; >>`

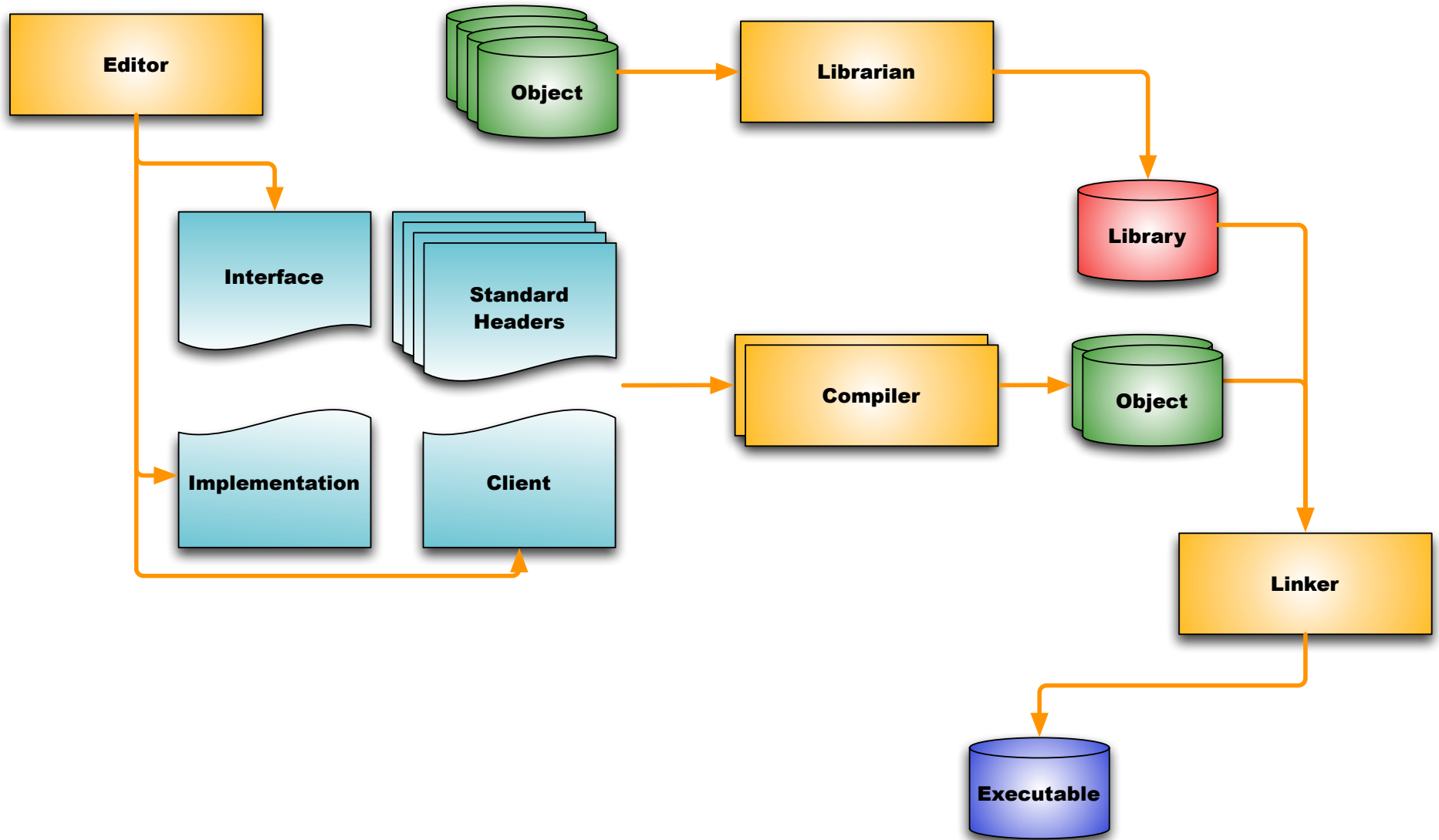
The Implementation File

- A class' method definitions should be provided in a separate implementation file.
- The file is compiled and its corresponding object file is distributed along with the interface file.
- Begin with an appropriate comment, any necessary preprocessor includes and appropriate **using** statements.
- Wrap the method definitions in the same namespace that was used in the interface file.

Using a Namespace

- To use all identifiers from a namespace, put a **using** statement after all **include** directives:
using <namespaceName>;
- To use a specific identifier use scope resolution:
using <namespaceName>::<identifier>;
- With no using statement, fully qualify the identifier:
<namespaceName>::<identifier> myVar; >>

Development Flow



Objects as Arguments and Return Values

Arguments & Parameters

- An argument is a value or reference passed to a function when it is invoked.
- A value parameter is a special local variable that is automatically initialized to the value of the argument passed to the function.
- A reference parameter is a special local identifier that serves as an alias to the variable passed to the function. It may be modified with `const`. Except for upcasting, the types must match exactly.

Default Arguments

- The argument's default(s) is/are specified only once, in the prototype and not in the definition.
- If all the arguments don't have defaults, those with defaults must be the rightmost ones.
- In a function call, arguments with default values may be omitted from the right end of the argument list.

point.h

```
#ifndef MAIN_SAVITCH_POINT_H
#define MAIN_SAVITCH_POINT_H

namespace main_savitch_2A {

class point {
public:
    point(double initial_x = 0.0, double initial_y = 0.0);

    void shift(double x_amount, double y_amount);
    void rotate90( );

    double get_x( ) const {
        return x; }
    double get_y( ) const {
        return y; }
private:
    double x;
    double y;
};

} // end namespace main_savitch_2A

#endif
```

point.cpp

```
#include "point.h"

namespace main_savitch_2A {

point::point(double initial_x, double initial_y) {
    x = initial_x;
    y = initial_y;
}

void point::shift(double x_amount, double y_amount) {
    x += x_amount;
    y += y_amount;
}

void point::rotate90( ) {
    double new_x;
    double new_y;
    new_x = y;
    new_y = -x;
    x = new_x;
    y = new_y;
}

} // end namespace main_savitch_2A
```

Operator Overloading

Rules

- At least one parameter must be a class object
- A friend function must be used if the left parameter is not an object of the same class.
- If the function is a class member its number of parameters is one less than the operator's arity. The missing parameter is the current object.
- New operators cannot be created.
- The arity, precedence and associativity cannot be changed.
- `= [] ++` and `--` must be overloaded in special ways.
- Some operators cannot be overloaded: membership, scope resolution and decision.

newpoint.h

```
#ifndef MAIN_SAVITCH_NEWPOINT_H
#define MAIN_SAVITCH_NEWPOINT_H
#include <iostream>

namespace main_savitch_2B {

class point {
public:
    point(double initial_x = 0.0, double initial_y = 0.0);

    void shift(double x_amount, double y_amount);
    void rotate90( );

    double get_x( ) const {
        return x; }
    double get_y( ) const {
        return y; }

    friend std::istream& operator >>(std::istream& ins, point& target);
private:
    double x, y;
};

double distance(const point& p1, const point& p2);
point middle(const point& p1, const point& p2);
point operator +(const point& p1, const point& p2);
bool operator ==(const point& p1, const point& p2);
bool operator !=(const point& p1, const point& p2);
std::ostream& operator <<(std::ostream & outs, const point& source);

}
#endif
```

newpoint.cpp...

```
#include <iostream>
#include <math.h>
#include "newpoint.h"
using namespace std;
namespace main_savitch_2B {

point::point(double initial_x, double initial_y) {
    x = initial_x;
    y = initial_y;
}

void point::shift(double x_amount, double y_amount) {
    x += x_amount;
    y += y_amount;
}

void point::rotate90( ) {
    double new_x;
    double new_y;
    new_x = y;
    new_y = -x;
    x = new_x;
    y = new_y;
}
```

...newpoint.cpp...

```
bool operator ==(const point& p1, const point& p2) {
    return
        (p1.get_x( ) == p2.get_x( ))
        &&
        (p1.get_y( ) == p2.get_y( ));
}

bool operator !=(const point& p1, const point& p2) {
    return !(p1 == p2);
}

point operator +(const point& p1, const point& p2) {
    double x_sum, y_sum;
    x_sum = (p1.get_x( ) + p2.get_x( ));
    y_sum = (p1.get_y( ) + p2.get_y( ));
    point sum(x_sum, y_sum);
    return sum;
}

ostream& operator <<(ostream& outs, const point& source) {
    outs << source.get_x( ) << " " << source.get_y( );
    return outs;
}

istream& operator >>(istream& ins, point& target) {
    ins >> target.x >> target.y;
    return ins;
}
```

...newpoint.cpp

```
int rotations_needed(point p) {
    int answer = 0;
    while ((p.get_x( ) < 0) || (p.get_y( ) < 0)) {
        p.rotate90( );
        ++answer; }
    return answer;
}

void rotate_to_upper_right(point& p) {
    while ((p.get_x( ) < 0) || (p.get_y( ) < 0))
        p.rotate90( );
}

double distance(const point& p1, const point& p2) {
    double a, b, c_squared;
    a = p1.get_x( ) - p2.get_x( );
    b = p1.get_y( ) - p2.get_y( );
    c_squared = a * a + b * b;
    return sqrt(c_squared);
}

point middle(const point& p1, const point& p2) {
    double x_midpoint, y_midpoint;
    x_midpoint = (p1.get_x( ) + p2.get_x( )) / 2;
    y_midpoint = (p1.get_y( ) + p2.get_y( )) / 2;
    point midpoint(x_midpoint, y_midpoint);
    return midpoint;
}
} // end namespace main_savitch_2B
```

Project A

Complete Project 5 from Chapter 2 of the text. Directions are on page 90.

- Name your **class** `point3D`. You will submit its definition in `point3D.h` and the implementation in `point3D.cpp`.
- Include member functions whose prototypes are:

```
point3D(double x = 0, double y = 0, double z = 0);  
void get(double& x, double& y, double& z) const;  
void set(double x, double y, double z);  
void shift(double xDistance, double yDistance, double zDistance);  
void rotate(double xDegrees, double yDegrees, double zDegrees);
```

- Use `assert()` to insure that at least two arguments to `rotate()` are zero. Notice that the parameter names suggest that the units for the arguments are degrees. Be sure to test accordingly.
- Create a **namespace** for your class: `cis11`.
- Be sure to include appropriate documentation including
 - value semantics,
 - description of functionality,
 - precondition(s) and
 - postcondition(s).
- Test your **class** thoroughly with your own client code.

When you're satisfied with the definition and implementation, submit your work via the [CATE form for Project A](#).

Legend: *method/function* **keyword** *literal*