Operator Overloading

…Continuation

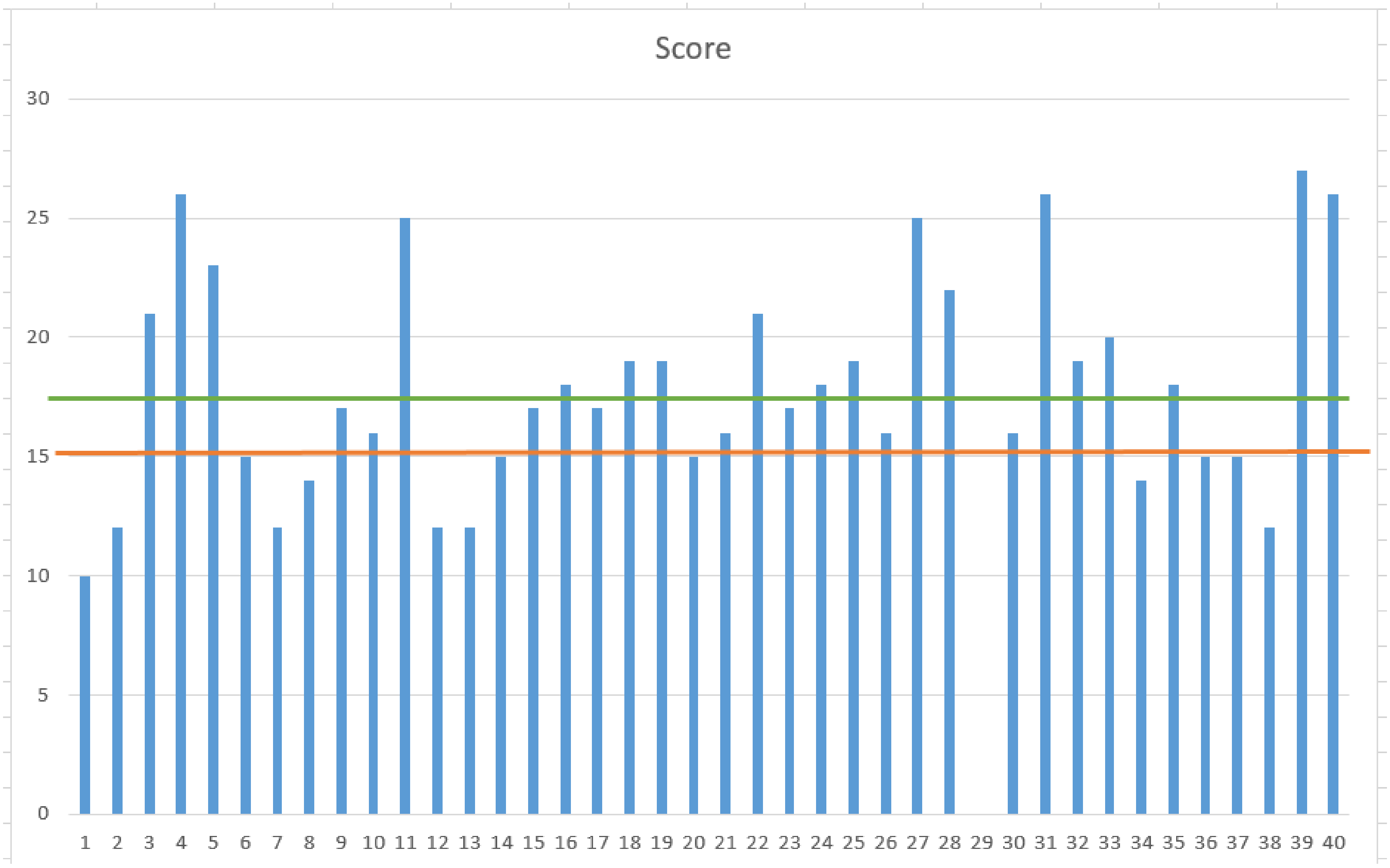
Contents

- Proxy Classes
- Overloading Unary/Binary Operators
- Talk on HW 2
- Type Conversion

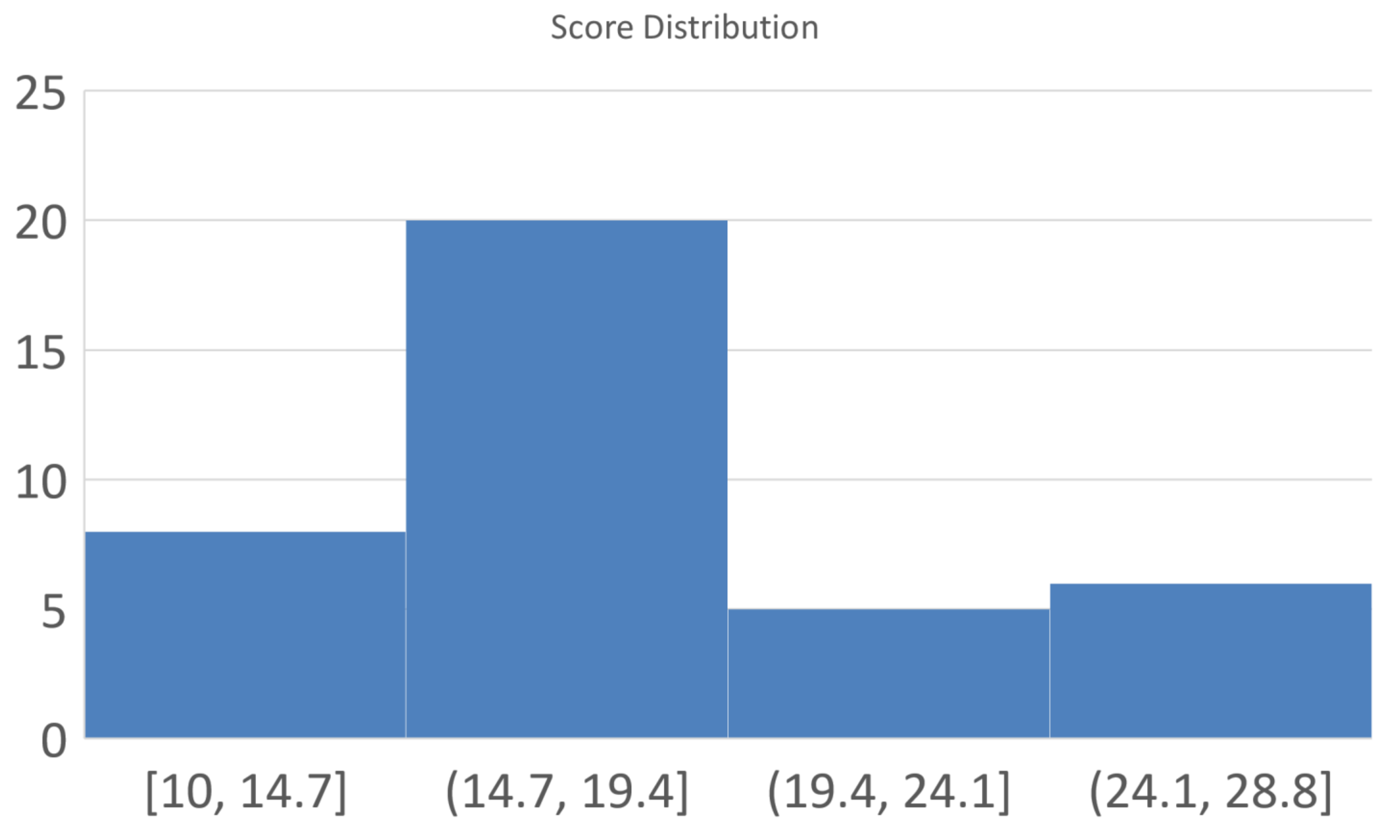
MidTerm Statistics

Score		%
10	Min	33.33
27	Max	90
17.87179	Average	59.57

MidTerm Statistics



MidTerm Statistics



Proxy Classes

- A proxy class allows us to hide even the private data of a class from clients of the class.
- Providing clients of a class with <u>**a proxy class that knows only the public interface**</u> to the class enables the clients to use the class’s services without giving the clients access to the class’s implementation details.

```cpp
// Fig. 11.16: Implementation.h
// Implementation class definition.

class Implementation
{
public:
   // constructor
   Implementation( int v )
      : value( v ) // initialize value with v
   {
      // empty body
   } // end constructor Implementation

   // set value to v
   void setValue( int v )
   {
      value = v; // should validate v
   } // end function setValue

   // return value
   int getValue() const
   {
      return value;
   } // end function getValue
private:
   int value; // data that we would like to hide from the client
}; // end class Implementation
```

Proxy Class

```cpp
// Fig. 11.17: Interface.h
// Proxy class Interface definition.
// Client sees this source code, but the source code does not reveal
// the data layout of class Implementation.

class Implementation; // forward class declaration required by line 17

class Interface
{
public:
   Interface( int ); // constructor
   void setValue( int ); // same public interface as
   int getValue() const; // class Implementation has
   ~Interface(); // destructor
private:
   // requires previous forward declaration (line 6)
   Implementation *ptr;
}; // end class Interface
```

Proxy Class

```
// Fig. 11.18: Interface.cpp
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.
#include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition

// constructor
Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) // initialize ptr to point to
{                                    // a new Implementation object
   // empty body
} // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v )
{

   ptr->setValue( v );
} // end function setValue

// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();
} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;
} // end ~Interface destructor
```

```cpp
// Fig. 11.19: fig11_19.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>
#include "Interface.h" // Interface class definition
using namespace std;

int main()
{
   Interface i( 5 ); // create Interface object

   cout << "Interface contains: " << i.getValue()
      << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
      << " after setValue" << endl;
} // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Overloading Unary Operator

- A unary operator for a class can be overloaded as a non-static member function with no arguments or as a global function with one argument that must be an object (or a reference to an object) of the class.

- Member functions that implement overloaded operators must be non-static so that they can access the non-static data in each object of the class.
- Static member functions can access only static members of the class.)

- Example: To test whether an object of the **String** class we create is empty and return a bool result

  !s to s.operator!()

```
class String
{
public:
   bool operator!() const;
   ...
}; // end class String
```

```
bool operator!( const String & );
```

Overloading Binary Operator

- A binary operator can be overloaded as a non-static member function with one parameter

OR, As a global function with two parameters (one of those parameters must be either a class object or a reference to a class object).

Example: Overloading < as a non-static member function of a String class with one argument

if y and z are String-class objects, then

y < z ➔ y.operator<(z)

```
class String

public:
   bool operator<( const String & ) const;
   ...
}; // end class String
```

Example: Global: If y and z are String-class objects or references to String-class objects, then

y < z ➔ operator<(y, z)

```
bool operator<( const String &, const String & );
```

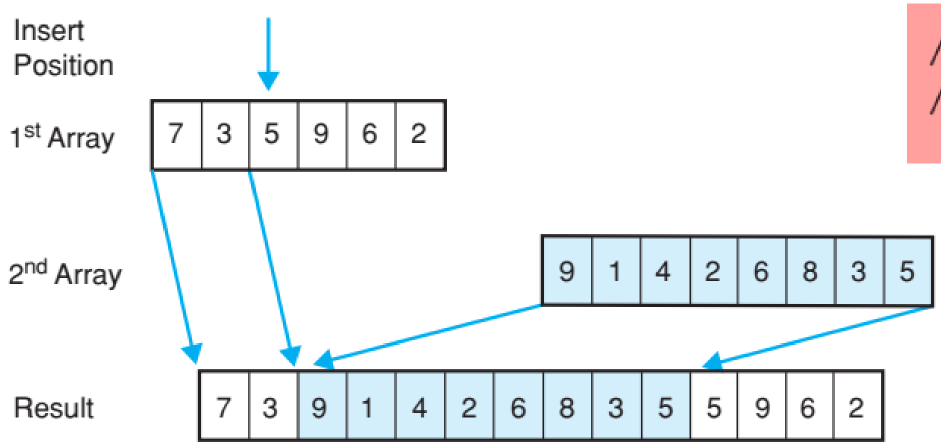
Talk on HW 2

Write a global function called arraySplit() that splits two **int** arrays together by **first allocating memory for a dynamic array** with enough space for both arrays, and then copying the elements from both arrays to the new array, as follows:

1) first, the elements of the first array are inserted up to a given position,
2) then the second array is inserted,
3) then the remainder of the first array is appended.

Arguments: The two int arrays, their length, and the position at which they are to be spliced.

Return value: A pointer to the new array



```
// For srand() and rand()
// and for time().
```

Talk on HW 2

Develop Array class

- Performs range checking to ensure that subscripts remain within the bounds of the Array.
- Allows one array object to be assigned to another with the assignment operator.
- Array objects know their size, so the size does not need to be passed separately to functions that receive Array parameters.
- Entire Arrays can be input or output with the stream extraction and stream insertion operators, respectively.
- Compare Arrays with the equality operators == and !=.

Case Study: Array Class

Talk on HW 2

Develop class Polynomial

- The internal representation of a Polynomial is an array of terms. Each term contains a coefficient and an exponent, e.g., the term

$$2x^4$$

has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as set and get functions.

a) Overload the addition operator (+) to add two Polynomials.

b) Overload the subtraction operator (-) to subtract two Polynomials.

c) Overload the assignment operator to assign one Polynomial to another.

d) Overload the multiplication operator (*) to multiply two Polynomials.

e) Overload the addition assignment operator (+=), subtraction assignment operator (-=),

f) and multiplication assignment operator (*=).

Converting Between Types

- Example, adding an int to an int produces an int. It’s often necessary, However, to convert data of one type to data of another type.

- This can happen in assignments, in calculations, in passing values to functions and in returning values from functions. The compiler knows how to perform certain conversions among fundamental types.

- What about user-defined types? The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so we must specify how to do this.

Converting Between Types

- A **conversion operator** (also called a cast operator)

Convert an object of one class into an object of another class or into an object of a fundamental type.

Such a conversion operator must be a non-static member function.

```
A::operator char *() const;
```

declares an overloaded cast operator function for converting an object of user-defined type A into a temporary char * object.

```
static_cast< char * >( s )
```

```
s.operator char *()
```

```
A::operator int() const;
A::operator OtherClass() const;
```