# San Francisco Bay University

## CS360 - Programming in C and C++
## Homework Assignment #4

**Due day: 3/23/2024**

**Instruction:**

1. **Push the answer sheets/source code to Github**
2. **Please follow the code style rule like programs on handout.**
3. **Overdue homework assignment submission can't be accepted.**
4. **Take academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)**

1. One nice example of overloading the function call operator () is to allow another form of double-array subscripting popular in some programming languages. Instead of saying

   *chessBoard[ row ][ column ]*

   for an array of objects, overload the function call operator to allow the alternate form

   *chessBoard( row, column )*

   Create a class *DoubleSubscriptedArray* that has similar features to class *Array* as following example programs. At construction time, the class should be able to create a *DoubleSubscriptedArray* of any number of rows and columns. The class should supply *operator()* to perform double-subscripting operations. For example, in a *3-by-5 DoubleSubscriptedArray* called *chessBoard*, the user could write *chessBoard(1, 3)* to access the element at row *1* and column *3*. Remember that operator() can receive *any* number of arguments. The underlying representation of the *DoubleSubscriptedArray* could be a one-dimensional array of integers with *rows * columns* number of elements. Function *operator()* should perform the proper pointer arithmetic to access each element of the underlying array. There should be two versions of *operator()* - one that returns *int &* (so that an element of a *DoubleSubscriptedArray* can be used as an *lvalue*) and one that returns *int*. The class should also provide the following operators: ==, *!*=, =, << (for outputting the *DoubleSubscriptedArray* in row and column format) and >> (for inputting the entire *DoubleSubscriptedArray* contents).

```
//Array.h
// Array class definition with overloaded operators.
#ifndef ARRAY_H
#define ARRAY_H
```

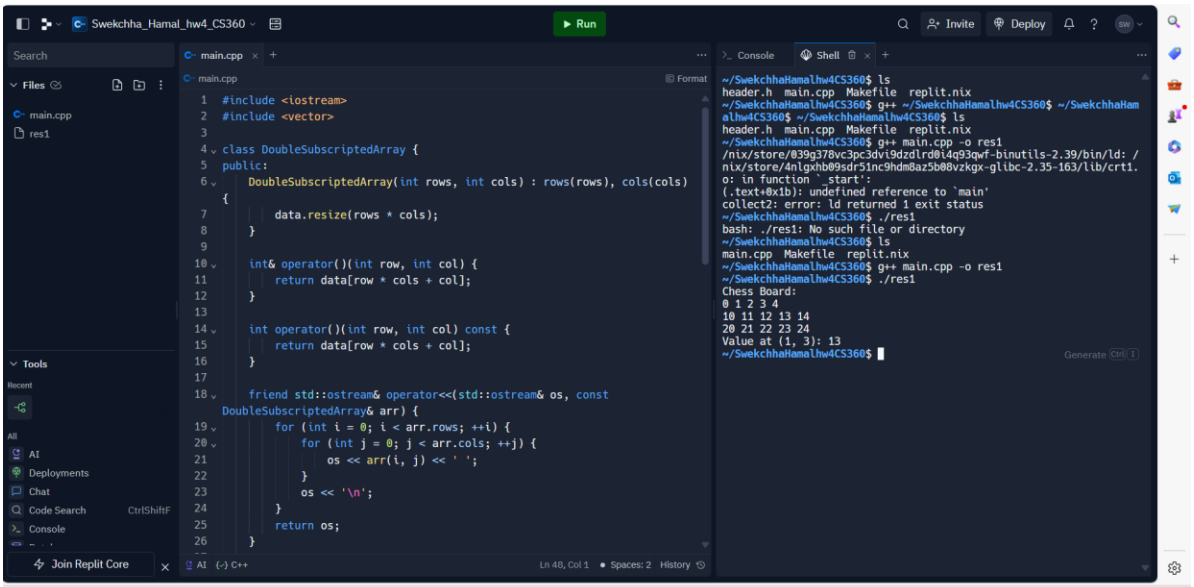
```
#include <iostream>

class Array{

  friend std::ostream &operator<<( std::ostream &, const Array & );
  friend std::istream &operator>>( std::istream &, Array & );

  public:
    explicit Array( int = 10 ); // default constructor
    Array( const Array & ); // copy constructor
    ~Array(); // destructor
    size_t getSize() const; // return size

    const Array &operator=( const Array & ); // assignment operator
    bool operator==( const Array & ) const; // equality operator

    // inequality operator; returns opposite of == operator
    bool operator!=( const Array &right ) const{
      return ! ( *this == right ); // invokes Array::operator==
    } // end function operator!=
    // subscript operator for non-const objects returns modifiable
lvalue

    int &operator[]( int );

    // subscript operator for const objects returns rvalue
    int operator[]( int ) const;

  private:
    size_t size; // pointer-based array size
    int *ptr; // pointer to first element of pointer-based array
}; // end class Array

#endif


//Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <stdexcept>

#include "Array.h" // Array class definition
using namespace std;

// default constructor for class Array (default size 10)
Array::Array( int arraySize ): size( arraySize > 0 ? arraySize :
```

```
  throw invalid_argument( "Array size must be greater than 0" ) ),
  ptr( new int[ size ] )
{
  for ( size_t i = 0; i < size; ++i )
    ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to an Array
Array::Array( const Array &arrayToCopy ): size( arrayToCopy.size ),
  ptr( new int[ size ] )
{
  for ( size_t i = 0; i < size; ++i )
    ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array(){
  delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
size_t Array::getSize() const{
  return size; // number of elements in Array
} // end function getSize

// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right ){
  if ( &right != this )// avoid self-assignment
  {
  // for Arrays of different sizes, deallocate original
  // left-side Array, then allocate new left-side Array
    if ( size != right.size ){
      delete [] ptr; // release space
      size = right.size; // resize this object
      ptr = new int[ size ]; // create space for Array copy
    } // end inner if

    for ( size_t i = 0; i < size; ++i )
      ptr[ i ] = right.ptr[ i ]; // copy array into object
  } // end outer if

  return *this; // enables x = y = z, for example
} // end function operator=

// determine if two Arrays are equal and
```

```
// return true, otherwise return false
bool Array::operator==( const Array &right ) const{
  if ( size != right.size )
    return false; // arrays of different number of elements

  for ( size_t i = 0; i < size; ++i )
    if ( ptr[ i ] != right.ptr[ i ] )
      return false; // Array contents are not equal

  return true; // Arrays are equal
} // end function operator==

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript ){
// check for subscript out-of-range error
  if ( subscript < 0 || subscript >= size )
    throw out_of_range( "Subscript out of range" );
  return ptr[ subscript ]; // reference return
} // end function operator[]

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const{
// check for subscript out-of-range error
  if ( subscript < 0 || subscript >= size )
    throw out_of_range( "Subscript out of range" );

  return ptr[ subscript ]; // returns copy of this element
} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a ){
  for ( size_t i = 0; i < a.size; ++i )
    input >> a.ptr[ i ];

  return input; // enables cin >> x >> y;
} // end function

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a ){
// output private ptr-based array
  for ( size_t i = 0; i < a.size; ++i ){
    output << setw( 12 ) << a.ptr[ i ];

    if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
```

```
        output << endl;
    } // end for

    if ( a.size % 4 != 0 ) // end last line of output
      output << endl;

    return output; // enables cout << x << y;
  } // end function operator<<
```

**Ans:**




```
private:
    int rows;
    int cols;
    std::vector<int> data;
};

int main() {
    DoubleSubscriptedArray chessBoard(3, 5);

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 5; ++j) {
            chessBoard(i, j) = i * 10 + j;
        }
    }

    std::cout << "Chess Board:\n" << chessBoard;
    std::cout << "Value at (1, 3): " << chessBoard(1, 3) << '\n';

    return 0;
}
```

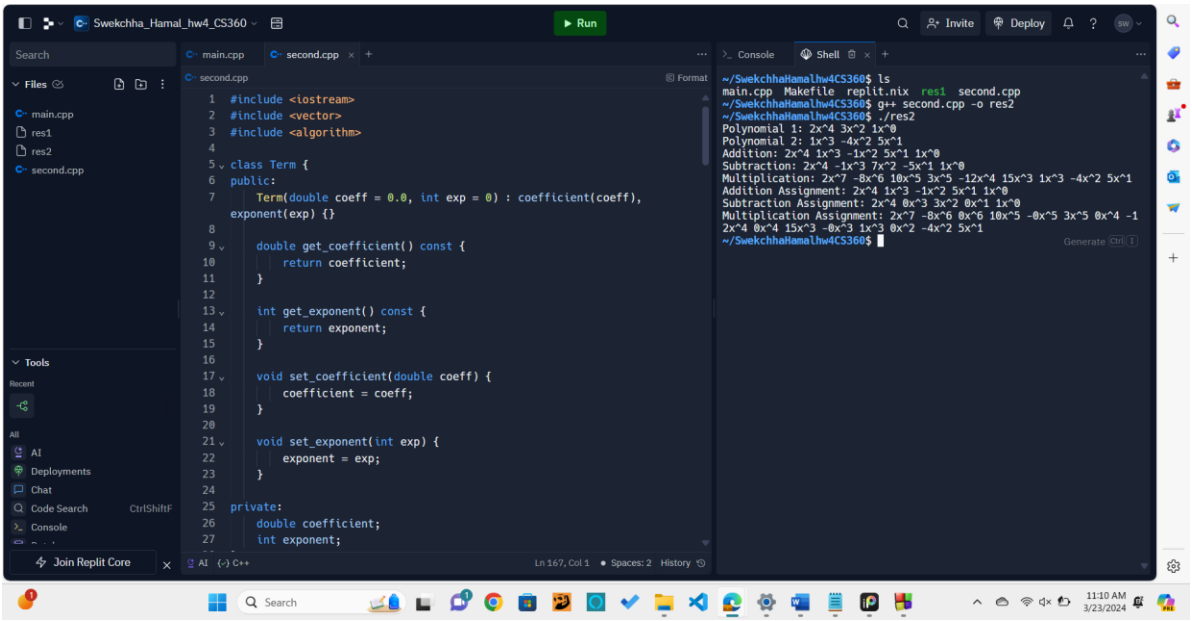
**Output:**

```
Chess Board:
0 1 2 3 4
10 11 12 13 14
20 21 22 23 24
Value at (1, 3): 13
```

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. The class should also provide the following overloaded operator capabilities:

    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 *(-=)*, and multiplication assignment operator *(*=)*.

**Ans:**



```
class Polynomial {
public:
    Polynomial() {}

    Polynomial(const std::vector<Term>& terms) : terms(terms) {}

    void add_term(const Term& term) {
        terms.push_back(term);
        std::sort(terms.begin(), terms.end(), [](const Term& a, const
Term& b) {
            return a.get_exponent() > b.get_exponent();
        });
    }

    void clear() {
        terms.clear();
    }

    Polynomial operator+(const Polynomial& other) const {
        Polynomial result;
        std::vector<Term> combined_terms = terms;
        combined_terms.insert(combined_terms.end(),
other.terms.begin(), other.terms.end());
        result.terms = combined_terms;
        result.combine_like_terms();
        return result;
    }
```

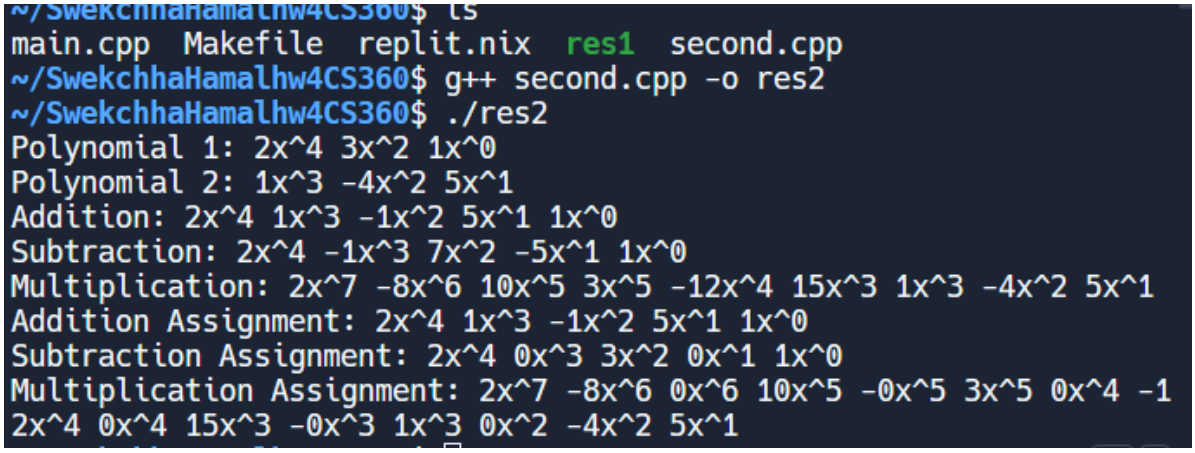
```
    Polynomial operator-(const Polynomial& other) const {
        Polynomial result = *this;
        for (const auto& term : other.terms) {
            result.add_term(Term(-term.get_coefficient(),
term.get_exponent()));
        }
        result.combine_like_terms();
        return result;
    }

    Polynomial operator*(const Polynomial& other) const {
        Polynomial result;
        for (const auto& term1 : terms) {
            for (const auto& term2 : other.terms) {
                double coeff = term1.get_coefficient() *
term2.get_coefficient();
                int exp = term1.get_exponent() + term2.get_exponent();
                Term new_term(coeff, exp);
                result.add_term(new_term);
            }
        }
        return result;
    }

    Polynomial& operator+=(const Polynomial& other) {
        *this = *this + other;
        return *this;
    }
```

```
    Polynomial& operator*=(const Polynomial& other) {
        *this = *this * other;
        return *this;
    }

    Polynomial& operator=(const Polynomial& other) {
        if (this != &other) {
            terms = other.terms;
        }
        return *this;
    }

    void combine_like_terms() {
        std::sort(terms.begin(), terms.end(), [](const Term& a, const
Term& b) {
            return a.get_exponent() > b.get_exponent();
        });

        std::vector<Term> combined_terms;
        for (const auto& term : terms) {
            bool found = false;
            for (auto& combined_term : combined_terms) {
                if (combined_term.get_exponent() ==
term.get_exponent()) {

combined_term.set_coefficient(combined_term.get_coefficient() +
term.get_coefficient());
                    found = true;
```

```
                    break;
                }
            }
            if (!found) {
                combined_terms.push_back(term);
            }
        }
        terms = combined_terms;
    }

    friend std::ostream& operator<<(std::ostream& os, const
Polynomial& poly) {
        for (const auto& term : poly.terms) {
            os << term.get_coefficient() << "x^" <<
term.get_exponent() << " ";
        }
        return os;
    }

private:
    std::vector<Term> terms;
};

int main() {
    Polynomial poly1, poly2, result;

    poly1.add_term(Term(2.0, 4));
    poly1.add_term(Term(3.0, 2));
```

```
  poly1.add_term(Term(2.0, 4));
  poly1.add_term(Term(3.0, 2));
  poly1.add_term(Term(1.0, 0));

  poly2.add_term(Term(1.0, 3));
  poly2.add_term(Term(-4.0, 2));
  poly2.add_term(Term(5.0, 1));

  std::cout << "Polynomial 1: " << poly1 << std::endl;
  std::cout << "Polynomial 2: " << poly2 << std::endl;

  result = poly1 + poly2;
  std::cout << "Addition: " << result << std::endl;

  result = poly1 - poly2;
  std::cout << "Subtraction: " << result << std::endl;

  result = poly1 * poly2;
  std::cout << "Multiplication: " << result << std::endl;

  poly1 += poly2;
  std::cout << "Addition Assignment: " << poly1 << std::endl;

  poly1 -= poly2;
  std::cout << "Subtraction Assignment: " << poly1 << std::endl;

  poly1 *= poly2;
  std::cout << "Multiplication Assignment: " << poly1 << std::endl;
```

**Output:**


~/SwekchhaHamalhw4CS360$ ls
main.cpp Makefile replit.nix res1 second.cpp
~/SwekchhaHamalhw4CS360$ g++ second.cpp -o res2
~/SwekchhaHamalhw4CS360$ ./res2
Polynomial 1: 2x^4 3x^2 1x^0
Polynomial 2: 1x^3 -4x^2 5x^1
Addition: 2x^4 1x^3 -1x^2 5x^1 1x^0
Subtraction: 2x^4 -1x^3 7x^2 -5x^1 1x^0
Multiplication: 2x^7 -8x^6 10x^5 3x^5 -12x^4 15x^3 1x^3 -4x^2 5x^1
Addition Assignment: 2x^4 1x^3 -1x^2 5x^1 1x^0
Subtraction Assignment: 2x^4 0x^3 3x^2 0x^1 1x^0
Multiplication Assignment: 2x^7 -8x^6 0x^6 10x^5 -0x^5 3x^5 0x^4 -1
2x^4 0x^4 15x^3 -0x^3 1x^3 0x^2 -4x^2 5x^1