CPOLYNOMIAL CLASS: AN IMPLEMENTATION OF POLYNOMIALS IN C++

Kailash Chandra
James D. Harris
Sapana Suhani Chandra
Computer Science/Information Systems
Pittsburg State University
Pittsburg, KS 66762
http://www.pittstate.edu/

ABSTRACT

A complete C++ implementation of single variable polynomials is presented. This paper illustrates work completed while teaching a data structures course. During the course, a class that implements polynomials with one unknown variable was developed and used to introduce students to dynamic data structures, several algorithms, and several advanced features of C++. The class is implemented using a doubly linked list and features of C++, such as, constructors, destructors, function overloading, operator overloading, and container classes. Several incremental programming assignments are also presented that can be used during a course focused on data structures.

INTRODUCTION

This paper describes a C++ implementation of single variable polynomials along with functions and operators that can be used to manipulate these polynomials. This implementation of polynomials was used to introduce students in a data structures course to dynamic data structures and several advanced features of the C++ programming language. It was important for students to be exposed to such features of C++, as, operator overloading and function overloading. Efficiency of the implementations was also important. The polynomial class and its associated functions were developed in an incremental fashion with students being required to provide test functions for each function or operator defined for the class. Students were also required to implement various simple functions and build on them to define other functions. This process exposed the students to modular programming, software reuse, and interface problems.

There are various references that encourage introducing data structures in the classroom while keeping it interesting and challenging at the same time [1-8]. These authors have introduced and implemented different data structures using various unique features of different

programming languages. In the case of C++, most operators can easily be overloaded without introducing unnecessary function names.

Several textbooks [9-11] have dealt with implementing polynomials by briefly describing their methods. Some have introduced polynomial ADT implementations using a dense-list and three functions: initialize, addition, and multiplication. Other texts have enhanced the typical representation approach which uses an array with a singly linked list for representing polynomials. In our case we have used a doubly linked list representation. The cost of an extra link in dynamic representation simplifies the process of traversing lists backwards and makes the deletion of an element much easier.

The polynomials used for the case presented in this paper have only one unknown variable *x*. Numeric coefficients are associated with each term as illustrated in the following example:

$$p(x) = 4x^5 + 2x^4 - 5x^2 + 23$$

A univariate polynomial of degree *d* has the form

$$c_d x^d + c_{d-1} x^{d-1} + c_{d-2} x^{d-2} + c_{d-3} x^{d-3} \ldots + c_0$$

where the coefficients $c_d \neq 0$. By definition, the exponents are nonnegative integers. Each $c_i x^i$ is a term of the polynomial. We have developed two classes, CTerm to implement the terms in a polynomials and CPolynomial to implement the polynomials. Each polynomial is represented as a doubly linked list of terms. The data fields of each term include a coefficient and an exponent.

The class CTerm has four private members: prev to hold the address of the previous term, `coef` to hold the coefficient of the term, `expo` to hold the exponent of x, and `next` to hold the address of the next term. In addition to several constructors and functions, the operators =, ==, !=, <<, >>, binary +, binary -, unary +, unary -, *, and / are supported.

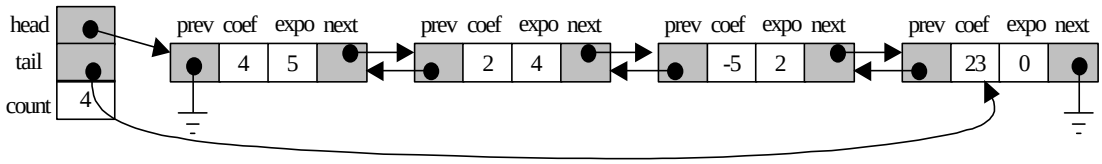
The class CPolynomial has three private members: count to hold the number of terms in the polynomial, head to hold the address of the first node, and tail to hold the address of the last node. In addition to several constructors and other useful functions, the operators =, ==, !=, <<, >>, binary +, binary -, unary +, unary -, *, /, +=, -=, *=, and /= are supported. We have also implemented mixed mode arithmetic operations between int, float, double, CTerm, and CPolynomial types. The two classes, CTerm and CPolynomial, are used to fully implement the univariate polynomials and perform many simple arithmetic operations with the help of member and friend functions of CTerm and CPolynomial classes.

Terms with a coefficient of zero are not represented. The terms are maintained in decreasing order of exponent.

An example of the output for the polynomial $4x^5 + 2x^4 - 5x^2 + 23$ would be:

+4*x^5 +2*x^4 –5*x^2 +23

The polynomial above can be represented by a doubly linked list as follows:



There are two types of structures in this dynamic representation: (1) the control block that holds the address of the first term, the address of the last term, and number of terms in the polynomial; and (2) term nodes that hold the coefficient, the exponent of *x*, the previous pointer, and the next pointer.

The following sections describe features of C++ used in the implementation, provide a briefly description of the CTerm and CPolynomial classes along with their members, and list several small student projects that build upon these classes.

SUMMARY OF C++ FEATURES INCLUDED IN THIS CASE

In this section, we briefly review the features of C++ that students are exposed to in the process of implementing polynomials as doubly linked lists.

Class types are defined using the class keyword and its variables and functions are called members.

A member function with the same name as its class is called a constructor function. Constructors cannot return values, even if they have return statements. If a class has a constructor, each object of that type is initialized with the constructor prior to use in a program. Constructors are called automatically when an object is created. Dynamic objects are created using the `new` operator.

Destructor functions are the inverse of constructor functions. They are called when objects are destroyed (deallocated). Destructor is a member function with the same name as its class name but preceded with a tilde (~). The destructor is commonly used to "clean up" when an object is no longer necessary. The objects are destroyed automatically when a statement outside its scope is executed. A `delete` operator can also destroy an object if it was created by a `new` operator. For example, if an object of type `CPolynomial` was created by the statement

```
CPolynomial *cp new CPolynomial;
```

then it can be destroyed by the statement `delete cp;` at which time the destructor function for `CPolynomial` will be called automatically. It is not necessary to define a destructor function for a class if the objects do not include dynamically allocated storage.

C++ allows specification of more than one function with the same name in the same scope. These are called "overloaded functions." Overloaded functions enable programmers to supply different semantics for a function, depending on the types and number of arguments. Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call.

One can redefine the function of most built-in operators. These operators can be redefined, or "overloaded," globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be class-members or friends. The syntax for the operator overloading is similar to overloading of a function, instead of the function name we put **operator** *x*, where *x* is the operator as it appears in Table 12.2 [12]. The operators shown in Table 12.3 [12] cannot be overloaded. For example,

```
CTerm operator + (const CTerm &cn);
```

To declare a unary operator function as a non-static member, you must declare it in the form: *ret-type* `operator` *op***()** where *ret-type* is the return type and *op* is one of the operators listed in Table 12.4[12]. For example,

```
CTerm operator -() const;
```

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions: it must be a non-static member function, no **operator=** can be declared as a nonmember function, and it is not inherited by derived classes. A default **operator=** function can be generated by the compiler for class types if none exists. The operator returns the object to preserve the behavior of the assignment operator, which returns the value of the left side after the assignment is complete. For example,

```
CTerm & operator = (const CTerm &cn);
```

A pointer holds the address of an object and is declared as: *type *variable.*

```
CTerm *ptr;
```

A reference holds the address of an object but behaves syntactically like an object, it is an alias for another variable. A reference declaration consists of an optional list of specifiers followed by a reference declarator.

DOUBLY-LINKED LIST DATA STRUCTURE

As mentioned earlier, two classes are defined to implement the polynomials. The first is CTerm to hold the terms in a polynomial and the second is CPolynomial to hold the doubly linked list of objects of `CTerm` type. The `CTerm` class was defined outside the `CPolynomial` class so it can be used outside the `CPolynomial` class and to demonstrate the ability of C++ to define new classes based on existing classes. The class `CTerm` has no destruction function member as it has no memory allocation at run-time. Their definitions and brief descriptions of the class members are as follows:

GLOBAL DECLARATIONS

```
const int MAX_TERMS = 15;
const int MAX_COEF = 10;
const int MAX_EXPO = 5;
```

CTERM CLASS DEFINITION

```
class CTerm
  {
  private:
    double coef;
    int expo;
    CTerm *prev;
    CTerm *next;
  public:
    CTerm(void);
    CTerm(const double &coef);
    CTerm(const double &coef, const int &expo);
    CTerm(const CTerm &cn);
    CTerm(const char *cp);
    CTerm(char ch);
    CTerm & operator = (const CTerm &cn);
    CTerm operator + (const CTerm &cn) const;
    CTerm operator + (const double &c) const;
    CTerm operator - (const CTerm &cn) const;
    CTerm operator - (const double &c) const;
    CTerm operator * (const CTerm &cn) const;
    CTerm operator * (const double &c) const;
    CTerm operator / (const CTerm &cn) const;
    CTerm operator / (const double &c) const;
    bool operator == (const CTerm &cn) const;
    bool operator == (const double &c) const;
    bool operator != (const CTerm &cn) const;
    bool operator != (const double &c) const;
    CTerm operator -() const;
    CTerm operator +() const;
  friend CTerm operator + (const double &c, const CTerm &
cn);
  friend CTerm operator - (const double &c, const CTerm &
cn);
  friend CTerm operator * (const double &c, const CTerm &
cn);
  friend CTerm operator / (const double &c, const CTerm &
cn);
  friend bool operator == (const double &c, const CTerm &
cn);
  friend bool operator != (const double &c, const CTerm &
cn);
  friend void swap(CTerm &n1, CTerm &n2);
  friend ostream & operator << (ostream &os, const CTerm &p);
  friend istream & operator >> (istream &os, CTerm &p);
  friend class CPolynomial;
  friend ostream & operator << (ostream &os, const
CPolynomial &poly);
  friend istream & operator >> (istream &os, CPolynomial
&poly);
```

```
  friend CPolynomial & operator * (const CTerm &cn, const
CPolynomial &poly);
  friend bool CrossLinked(const CPolynomial &p1, const
CPolynomial &p2);
  };
```

The following table briefly describes each member or friend of the class `CTerm`.

Member or friend	**Description (briefly describe the purpose of the function)**
`double coef;`	Holds coefficient of a term in a polynomial.
`int expo;`	Holds exponent of x for a term in a polynomial, it is assumed to be non-negative.
`CTerm *prev;`	Pointer to the previous term if any in the doubly linked list representation of a polynomial.
`CTerm *next;`	Pointer to the next term if any in the doubly linked list.
`CTerm(void);`	Default constructor for CTerm. It sets the coef and expo fields to zero and prev and next fields to NULL.
`CTerm(const double &coef);`	A constructor for Cterm for given value of coefficient. It sets the `coef` to the given value, `expo` to zero, and `prev` and `next` fields to `NULL`.
`CTerm(const double &coef, const int &expo);`	Constructor for CTerm when the values of coefficient and exponent of a term are given. It creates a node with given values for coef and expo. The pointer fields prev and next fields are set to NULL.
`CTerm(const CTerm &cn);`	Copy constructor.
`CTerm(const char *cp);`	Another constructor for CTerm when the value of a term is given through a string of characters, such as, “-2.5*x^3” resulting coef =-2.5, and expo=3. It creates a node with these values for coef and expo. The pointer fields prev and next fields are set to NULL.
`CTerm(char ch);`	A constructor to create some random terms depending on the supplied argument for ch, for example, ‘r’ to create a random term. This is useful for generating random terms and polynomials for testing purposes.

`CTerm & operator = (const CTerm &cn);`	Assignment operator overloaded for CTerm to assign only the coef and expo fields and leave the next and prev pointer fields intact.
`CTerm operator + (const CTerm &cn) const;`	Addition operator (+) overloaded to allow adding terms with equal exponents.
`CTerm operator + (const double &c) const;`	Addition operator (+) overloaded to allow adding coefficients with terms that have zero exponents.
`CTerm operator - (const CTerm &cn) const;`	Subtraction operator `(-)` overloaded to allow subtracting terms with equal exponents.
`CTerm operator - (const double &c) const;`	Subtraction operator (-) overloaded to allow subtracting coefficients with terms that have zero exponents.
`CTerm operator * (const CTerm &cn) const;`	Multiplication operator (*) overloaded to allow multiplication of terms.
`CTerm operator * (const double &c) const;`	Multiplication operator (*) overloaded to allow multiplication of coefficients with terms.
`CTerm operator / (const CTerm &cn) const;`	Division operator (/) overloaded to allow division of two terms provided that the resulting term does not have negative exponent.
`CTerm operator / (const double &c) const;`	Division operator (/) overloaded to allow division of two terms with coefficients.
`bool operator == (const CTerm &cn) const;`	Equality operator (==) overloaded to allow comparing for equality of two terms.
`bool operator == (const double &c) const;`	Equality operator `(==)` overloaded to allow comparing a term with a coefficient.
`bool operator != (const CTerm &cn) const;`	Inequality operator `(!=)` overloaded to allow comparing for inequality of two terms.
`bool operator != (const double &c) const;`	Inequality operator `(!=)` overloaded to allow comparing for inequality of a term with a coefficient.
`CTerm operator -() const;`	Unary minus operator `(-)` overloaded for terms.
`CTerm operator +() const;`	Unary plus operator `(+)` overloaded for terms.
`friend CTerm operator + (const double &c, const CTerm & cn);`	Addition operator overloaded to allow addition of a coefficient with a term.

`friend CTerm operator - (const double &c, const CTerm & cn);`	Subtraction operator overloaded to allow subtraction of a coefficient and a term.
`friend CTerm operator * (const double &c, const CTerm & cn);`	Multiplication operator overloaded to allow multiplication of a coefficient and a term.
`friend CTerm operator / (const double &c, const CTerm & cn);`	Division operator overloaded to allow division of a coefficient by a term provided that the resulting term does not have negative exponent.
`friend bool operator == (const double &c, const CTerm & cn);`	Equality operator `(==)` overloaded to allow comparing for equality of a coefficient with a term.
`friend bool operator != (const double &c, const CTerm & cn);`	Inequality operator `(!=)` overloaded to allow comparing for inequality of a coefficient and a term.
`Friend swap(CTerm &n1, CTerm &n2);`	A friend function to allow swapping of two terms of a polynomial. It swaps only the `coef` and `expo` fields of the two terms involved.
`friend ostream & operator << (ostream &os, const CTerm &p);`	Allows insertion operator (<<) overloaded for `CTerm` class to have access to the private data members of `CTerm` class.
`friend istream & operator >> (istream &os, CTerm &p);`	Allows extraction operator `(>>)` overloaded for `CTerm` class to have access to the private data members of `CTerm` class.
`Friend class CPolynomial;`	`CPolynomial`, the container class is declared as a friend to allow all of its member functions to have access to the private data members of `CTerm` class objects.
`friend ostream & operator << (ostream &os, const CPolynomial &poly);`	Allows insertion operator (`<<`) overloaded for `CPolynomial` class to have access to the private data members of `CTerm` class.
`friend istream & operator >> (istream &os, CPolynomial &poly);`	Allows extraction operator (`>>`) overloaded for `CPolynomial` class to have access to the private data members of `CTerm` class.

`friend CPolynomial & operator * (const CTerm &cn, const CPolynomial &poly);`	Multiplication operator (*) overloaded to allow multiplication of a term with a polynomial.
`friend bool areCrossLinked(const CPolynomial &poly1, const CPolynomial &poly2);`	Allows the global function `areCrossLinked` to have access to the private data members of `CTerm` class. The purpose of this function is to check if two polynomials have any term in common

CPOLYNOMIAL CLASS

```
class CPolynomial
  {
  private:
    int count;
    CTerm * head;
    CTerm * tail;
  public:
    CPolynomial(void);
    CPolynomial(const int &c);
    CPolynomial(const double &c);
    CPolynomial(const CTerm &cn);
    CPolynomial(const CPolynomial &poly);
    CPolynomial(const char *cp);
    CPolynomial(char ch);
    ~CPolynomial(void);

    void insertAtTail(int coef, int expo);
    void insertAtTail(const CTerm &cn);
    void insertAtHead(double coef, int expo);
    void insertAtHead(const CTerm &cn);

    void populate(int n);
    void deleteAll(void);
    void deleteAt(CTerm *p);
    void displayNodes(void) const;
    void displayReverse(void) const;
    int getCount(void) const;
    int getDegree(void) const;
    bool isOK(void) const;
    CTerm * addressOfNodeAtPos(int pos) const;

    CPolynomial & operator + (const CPolynomial &poly);
    CPolynomial & operator - (const CPolynomial &poly);
    CPolynomial & operator = (const CPolynomial &poly);
    CPolynomial & operator * (const CPolynomial &poly);
    CPolynomial & operator / (const CPolynomial &poly);
    CPolynomial & operator % (const CPolynomial &poly);
```

```
    CPolynomial & operator -() const;
    CPolynomial & operator +() const;

    void operator += (const CPolynomial &poly);
    void operator -= (const CPolynomial &poly);
    void operator *= (const CPolynomial &poly);
    void operator /= (const CPolynomial &poly);
    void operator %= (const CPolynomial &poly);

    bool operator == (const CPolynomial &poly) const;
    bool operator != (const CPolynomial &poly) const;

    void simplify(void);
    void sortDesc(void);

  friend CPolynomial & operator + (const CTerm &cn, const
CPolynomial &poly);
  friend CPolynomial & operator - (const CTerm &cn, const
CPolynomial &poly);
  friend CPolynomial & operator * (const CTerm &cn, const
CPolynomial &poly);
  friend bool operator == (const CTerm &cn, const CPolynomial
&poly);
  friend bool operator != (const CTerm &cn, const CPolynomial
&poly);

  friend ostream & operator << (ostream &os, const
CPolynomial &poly);
  friend istream & operator >> (istream &os, CPolynomial
&poly);
  friend bool areCrossLinked(const CPolynomial &p1, const
CPolynomial &p2);
  };
```

The following table briefly describes each member of friend of the class `CPolynomial`.

Member or friend	**Description**
`int count;`	Holds the number of terms in a polynomial.
`CTerm *head;`	Holds the address of the first node representing a term in a polynomial.
`CTerm *tail;`	Holds the address of the last node representing a term in a polynomial.
`CPolynomial(void);`	Default constructor that provides an empty or zero polynomial.

`CPolynomial(int &c);`	A constructor that provides a polynomial that has one term whose coefficient has the value of c and the exponent is zero.
`CPolynomial(double &c);`	A constructor that provides a polynomial that has one term whose coefficient has the value of c and the exponent is zero.
`CPolynomial(CTerm &cn);`	A constructor that provides a polynomial that has one term whose value is given by the value of cn.
`CPolynomial(const CPolynomial &poly);`	Copy constructor.
`CPolynomial(const char *cp);`	A constructor that provides a polynomial that has terms coming from an array of characters, for example, `CPolynomial poly("1.2*x^3+2.5*x^2")`. It extracts terms from the string argument and builds a polynomial.
`CPolynomial(char ch);`	A constructor to provide a different polynomial with a different character argument. For example, if the character argument is `'r'` then a random polynomial with random nodes and random number of terms is created.
`~CPolynomial(void);`	Destructor for `CPolynomial` class.
`void insertAtTail(int coef, int expo);`	Builds and inserts a node for given coefficient and the exponent values.
`void insertAtTail(const CTerm &cn);`	Builds and inserts a node at the tail given by cn.
`void insertAtHead(int coef, int expo);`	Builds and inserts a node at the head for given coefficient and the exponent values. This function is useful when new items are being added at the tail.
`void insertAtHead(const CTerm &cn);`	Builds and inserts a node at the head given by cn.
`void populate(int n);`	Creates a random polynomial with n terms, it is called by different constructors.
`void deleteAll(void);`	Deletes all the terms in a polynomial. It is called by other member functions including the destructor.
`void deleteAt(CTerm *p);`	Deletes a term from a polynomial whose memory address is given by `p`.
`void displayNodes(void) const;`	Displays nodes and their memory addresses.

`void displayReverse(void) const;`	Displays a polynomial in reverse order from last term to first term.
`int getCount(void) const;`	Returns the number of terms in a polynomial.
`int getDegree(void) const;`	Returns the degree of the polynomial, in other words the highest exponent value of the terms in a polynomial.
`bool isOK(void) const;`	Checks if the polynomial representation is corrupt or not.
`CTerm * addressOfNodeAtPos(int pos) const;`	Returns the address of a node for a given relative position in `pos`. The position of the left-most term pointed to by `head` is assumed to be 0.
`bool insertAtTail(const CTerm &cn);`	Builds and inserts a node for given coefficient and the exponent values.
`CPolynomial & operator + (const CPolynomial &poly);`	Addition operator `(+)` overloaded for `CPolynomial` class.
`CPolynomial & operator - (const CPolynomial &poly);`	Subtraction operator `(-)` overloaded for `CPolynomial` class.
`CPolynomial & operator = (const CPolynomial &poly);`	Assignment operator `(=)` overloaded for `CPolynomial` class.
`CPolynomial & operator * (const CPolynomial &poly);`	Multiplication operator `(*)` overloaded for `CPolynomial` class.
`CPolynomial & operator / (const CPolynomial &poly);`	Division operator `(/)` overloaded for `CPolynomial` class.
`CPolynomial & operator % (const CPolynomial &poly);`	Modulo operator `(%)` overloaded for `CPolynomial` class.
`CPolynomial & operator -() const;`	Unary operator minus `(-)` overloaded for `CPolynomial` class.
`CPolynomial & operator +() const;`	Unary operator plus `(+)` overloaded for `CPolynomial` class.
`void operator += (const CPolynomial &poly);`	Add and assign `(+=)` operator overloaded for `CPolynomial` class.
`void operator -= (const CPolynomial &poly);`	Subtract and assign `(-=)` operator overloaded for `CPolynomial` class.
`void operator *= (const CPolynomial &poly);`	Multiply and assign `(*=)` operator overloaded for `CPolynomial` class.
`void operator /= (const CPolynomial &poly);`	Divide and assign operator `(/=)` overloaded for `CPolynomial` class.
`void operator %= (const CPolynomial &poly);`	Modulo and assign operator `(%=)` overloaded for `CPolynomial` class.

`bool operator == (const CPolynomial &poly) const;`	Equality operator `==` overloaded for `CPolynomial` class.
`bool operator != (const CPolynomial &poly) const;`	Inequality operator `!=` overloaded for `CPolynomial` class.
`void simplify(void);`	Combines same exponent terms in a polynomial and deletes the terms with zero coefficient.
`void sortDesc(void);`	Sorts the terms of a polynomial in decreasing order of their exponent values.
`friend CPolynomial & operator + (const CTerm &cn, const CPolynomial &poly);`	Addition operator overloaded to allow adding a term with a polynomial.
`friend CPolynomial & operator - (const CTerm &cn, const CPolynomial &poly);`	Subtraction operator overloaded to allow subtracting a polynomial from a term.
`friend CPolynomial & operator * (const CTerm &cn, const CPolynomial &poly);`	Multiplication operator overloaded to allow multiplying a term with a polynomial.
`friend ostream & operator << (ostream &os, const CPolynomial &poly);`	Insertion operator `(<<)` overloaded for displaying the polynomials
`friend istream & operator >> (istream &os, CPolynomial &poly);`	Extraction operator `(>>)` overloaded to input a polynomial from the keyboard.
`friend bool areCrossLinked(const CPolynomial &poly1, const CPolynomial &poly2);`	Global function to check if two polynomials are areCrossLinked, i.e., have a term in two polynomials.

POSSIBLE PROGRAMMING PROJECTS

Two approaches have been used by the instructor for incorporating the polynomial classes in the data structures course. First, the class descriptions as given above were developed by the instructor and a preliminary version was reviewed during the fifty-minute class session. Students were assigned to complete the class implementations working on their own. This approach proved to be less than satisfactory. Out of the twenty students, only three were able to complete the assignment and it was found that those implementations were not very good or efficient. Completing one large project took an enormous amount of time and some students fell behind. The second approach for incorpating the polynomial classes in the data structures course involved the use of several mini projects. This approach makes the learning process gradual and the problems and difficulties are discovered earlier in the process. Also, there are students who procrastinate; those students need to be engaged in this process early on. The mini projects achieves this objective and reinforces key ideas through repetition. This second

approach has proven to be much more successful with most students achieving a reasonable level of success in completing the mini projects. There are many possible mini projects that can be assigned using these concepts. Below are brief descriptions of some of the possible programming projects. As one will notice, it is suggested that a test function should be required for each function developed.

- Implement `CTerm` class with default constructor, copy constructor, constructor with given coefficient and exponent of x, overload insertion operator `<<` for output, and test functions for each member function.
- Add the definition of `CPolynomial` to the previous project with a default constructor, a constructor to create a random polynomial, destructor to properly release the memory, overloaded insertion operator `<<` for output, `insertAtTail`, and test functions for each member functions.
- Add `isOK`, `getCount`, `simplify`, `sortDesc`, `swap`, and `areCrossLinked` member functions to the previous project. Make sure there is also a test function for each function added.
- Overload the operators addition `(+)`, subtraction `(-)`, assignment `(=)`, multiplication `(*)` to the previous project and also add a test function for each overloaded operator.
- Overload the operators unary negation `(-)`, unary positive `(+)`, add & assign `(+=)`, subtract & assign `(-=)`, multiply & assign `(*=)`, equal `(==)`, and not equal `(!=)` to the previous project. Also write test functions to thoroughly test the implementations of each new member function.
- Overload divide `(/)` operator and develop a test function to the previous project.
- Add `differentiate`, `integrate`, and `evaluate` functions to the previous project in addition to the test function for each function.

CONCLUSION AND SUMMARY

We have discussed various C++ features, implemented the polynomial class, and overloaded the various simple functions and operators so that polynomials can be handled just like any other primitive data type. Along with these, a set of mini projects has been presented that will step-by-step involve the students in learning more about the dynamic data structures, reinforce student knowledge concerning class definition and usage, and introduce students to more advanced features of C++ such as operator and function overloading. The complete source code that includes function definitions and all the test functions can be obtained by contacting the authors.

REFERENCES

[1] M. T. Goodrich and Robert Tamassia, Using Randomization in the Teaching of Data Structures and Algorithms, SIGCSE 1999, March 1999, New Orleans, pp 53-57.

[2] Joseph Turner and Joseph Zachary, Using Course-Long Programming Projects in CS2, SIGCSE 1999, March 1999, New Orleans, pp 43-47.

[3] Joel Adams, Chance-It: An Object-Oriented Capstone Project For CS-1, SIGCSE 1998, Atlanta, pp 10-14.

[4] Michael Mitzenmacher, Designing Stimulating Programming Assignments for an Algorithmic Course: A Collection of Exercises Based on Random Graphs, SIGCSE Bulletin, Vol. 28, No. 3, 1996, pp 36.

[5] Erkki Makinen, Programming Projects on Chess, SIGCSE Bulletin, Vol. 28, No. 4, December 1996, pp 41-44.

[6] V. Meisalo, E. Sutinen, and J. Tarhio, CLAP: teaching data structures in a creative way, SIGCSE Bulletin, Vol. 29, No. 3, September 1997, pp 117-119.

[7] Chaya Gurwitz, Teaching Linked Lists: Computer Science Education, Vol. 9, No. 1, 1999, pp 36-42.

[8] Richard Pifile, Using a Model Train to Illustrate a Doubly Linked List in a Microprocessor Laboratory, Computers in Education Journal, Vol. 7, No. 1, Jan-May 1997, pp 23-26.

[9] Mark Allen Weiss, Data Structures And Algorithm Analysis: The Benjamin/Cummings Publishing Company, Inc., 1995.

[10] John Beidler, An Introduction to Data Structures: Allyn And Bacon, Inc., 1982.

[11] Robert J. Baron and Linda G. Shapiro, Data Structures And Their Implementations: PWS Publishers, 1980.

[12] Microsoft MSDN Library April 2000.