

Laboratory Exercise 9 (Template classes)

Topics: Template classes

Goals: Upon successful completion of this lab you should be able to:

1. Define a template class
2. Test a template class using standard C++ types
3. Test a template class using a user defined type

Related text sections:

Chapter 16, 17

Laboratory Exercise 9 Instructions

1. Create a new directory (folder) called Lab9.
2. Concatenate your LList.h and LList.cpp files from Lab 8 Part 2 into a new file called LList.tmp in your new Lab9 directory. Modify the class description as indicated below. Modify the function implementations as described in lecture (the modified default constructors are illustrated below).

```
#ifndef LLIST_TMP
#define LLIST_TMP

#include <iostream>
using namespace std;

template <class LT> class LList;

template <class LT> ostream & operator << (ostream & outs, const LList<LT> & L);

template <class LT>
class LList
{
private:
    class LNode
    {
public:
        LNode ();
        LT data;
        LNode * next;
    };

public:
    LList ();
    LList (const LList & other);
    ~LList ();
    LList & operator = (const LList & other);
    bool operator == (const LList & other);
    int Size () const;
    friend ostream & operator << <> (ostream & outs, const LList<LT> & L);
    bool InsertFirst (const LT & value);
    bool InsertLast (const LT & value);
    bool DeleteFirst ();
    bool DeleteLast ();
private:
    LNode * first;
    int size;
};

template <class LT>
LList<LT>::LNode::LNode ()
{
    next = NULL;
}

template <class LT>
LList<LT>::LList ()
{
    first = NULL;
    size = 0;
}

    . . . More modified functions here . . .

#endif
```

3. Create an application program to test your linked list template. This program should instantiate LList objects using two of the standard C++ types: int, float, double, char, or bool. For example:

```
LList <int> L1;
LList <char> L2;
```

4. Your application program should specifically test each of the functions in the template class. For example:

```
L1.InsertFirst (10);
L2.InsertLast ('c');
cout << L1 << endl;
cout << L2.Size() << endl;
```

5. Compile and execute your program. Since there is only one .cpp file, you can simply use a g++ command to compile the program.
6. Once you are convinced that your linked list template is working correctly for standard C++ types, make a copy of your application program to test user/compiler defined types.
7. Copy the files CoordPt.h and CoordPt.cpp from CS215pickup/Lab9. (Listings for these files start on the next page.)

8. Add include statements to your new application program to include the string library and CoordPt.h.

9. Modify the LList instantiations to use string and CoordinatePoint:

```
LList <string> L1;
LList <CoordinatePoint> L2;
```

10. Modify your application program so that the functions test the new LList instantiations:

```
L1.InsertFirst ("Hello");
L2.InsertLast (CoordinatePoint (1,2));
cout << L1 << endl;
cout << L2.Size() << endl;
```

11. Create a makefile to compile and link your application program with the CoordPt implementation.

```
lab9 : application.o CoordPt.o
      g++ -o lab9 application.o CoordPt.o -g

application.o : application.cpp LList.tmp CoordPt.h
      g++ -c application.cpp -g

CoordPt.o : CoordPt.cpp CoordPt.h
      g++ -c CoordPt.cpp -g

clean :
      rm -f core.* *.o lab
```

12. Compile and test your new application program.

13. When you are convinced that your LList template class is working correctly, copy your LList.tmp file to the dropbox as yourlastnameL9.tmp.

```

#ifndef COORDPT_H
#define COORDPT_H

#include <iostream>
using namespace std;

class CoordinatePoint
{
public:
    CoordinatePoint();
        // Precondition: default constructor - no parameters required.
        // Postcondition: a new object with x set to 0 and y set to 0 will be
        // created.

    CoordinatePoint(int x_coord, int y_coord);
        // Precondition: x_coord and y_coord must be integer values.
        // Postcondition: a new object with x set to x_coord and y set to
        // y_coord will be created.

    CoordinatePoint(const CoordinatePoint & Other);
        // Precondition: copy constructor - Other must be a valid object.
        // Postcondition: a new object with x set to Other.x and y set to Other.y
        // will be created.

    ~CoordinatePoint();
        // Precondition: destructor - no parameters required.
        // Postcondition: no action since there is no dynamic memory.

    CoordinatePoint & operator = (const CoordinatePoint & Other);
        // Precondition: assignment operator - Other must be a valid object.
        // Postcondition: for the existing object, x will be set to Other.x
        // and y will be set to Other.y.

    float Magnitude () const;
        // Precondition: x and y are integer values.
        // Postcondition: the distance from the origin (0,0) will be
        // calculated (using the Pythagorean Theorem) and returned.

    float Distance (const CoordinatePoint & Other) const;
        // Precondition: *this and Other must be a valid objects.
        // Postcondition: the distance from the *this to Other will be
        // calculated (using the Pythagorean Theorem) and returned.

    CoordinatePoint operator + (const CoordinatePoint & Other);
        // Precondition: *this and Other must be a valid objects.
        // Postcondition: the sum of *this and Other will be
        // calculated and returned.

    friend istream & operator >> (istream & input, CoordinatePoint & p);
        // Precondition: p is an object of type CoordinatePoint.
        // Postcondition: the contents of p will be read from the input stream -
        // input will be in the format (x,y) - where x and y are integer values.
        // The parenthesis and comma will be required but will not be stored -
        // the fail flag will be set if the input format is incorrect.

    friend ostream & operator << (ostream & output, const CoordinatePoint & p);
        // Precondition: p is an object of type CoordinatePoint.
        // Postcondition: the contents of p will be sent to the output stream
        // using the format (x,y) where x and y are the x and y coordinates of p.
private:
    int x, y;
};

```

```

#endif
#include "CoordPt.h"
#include <cmath>
using namespace std;

CoordinatePoint::CoordinatePoint()
// Precondition: default constructor - no parameters required.
// Postcondition: a new object with x set to 0 and y set to
// 0 will be created.
{
    x = 0;
    y = 0;
}

CoordinatePoint::CoordinatePoint(int x_coord, int y_coord)
// Precondition: x_coord and y_coord must be integer values.
// Postcondition: a new object with x set to x_coord and y set to
// y_coord will be created.
{
    x = x_coord;
    y = y_coord;
}

CoordinatePoint::CoordinatePoint(const CoordinatePoint & Other)
// Precondition: copy constructor - Other must be a valid object.
// Postcondition: a new object with x set to Other.x and
// y set to Other.y will be created.
{
    x = Other.x;
    y = Other.y;
}

CoordinatePoint::~CoordinatePoint()
// Precondition: destructor - no parameters required.
// Postcondition: no action since there is no dynamic memory.
{
}

CoordinatePoint & CoordinatePoint::operator = (const CoordinatePoint & Other)
// Precondition: assignment operator - Other must be a valid object.
// Postcondition: for the existing object, x will be set to Other.x
// and y will be set to Other.y.
{
    x = Other.x;
    y = Other.y;
    return *this;
}

float CoordinatePoint::Magnitude () const
// Precondition: x and y are integer values.
// Postcondition: the distance from the origin (0,0) will be
// calculated (using the Pythagorean Theorem) and returned.
{
    return sqrt (float (x * x + y * y));
}

float CoordinatePoint::Distance (const CoordinatePoint & Other) const
// Precondition: *this and Other must be a valid objects.
// Postcondition: the distance from the *this to Other will be
// calculated (using the Pythagorean Theorem) and returned.
{
    int deltax = x - Other.x;
    int deltay = y - Other.y;
}

```

```

        return sqrt (float (deltax * deltax + deltax * deltax));
    }

CoordinatePoint CoordinatePoint::operator + (const CoordinatePoint & Other)
// Precondition: *this and Other must be a valid objects.
// Postcondition: the sum of *this and Other will be
// calculated and returned.
{
    int xvalue = x + Other.x;
    int yvalue = y + Other.y;
    return CoordinatePoint (xvalue, yvalue);
}

istream & operator >> (istream & input, CoordinatePoint & p)
// Precondition: p is an object of type CoordinatePoint.
// Postcondition: the contents of p will be read from the input stream -
// input will be in the format (x,y) - where x and y are integer values.
// The parenthesis and comma will are required but will not be stored -
// the fail flag will be set if the input format is incorrect.
{
    char format;
    input >> format >> p.x >> format >> p.y >> format;
    return input;
}

ostream & operator << (ostream & output, const CoordinatePoint & p)
// Precondition: p is an object of type CoordinatePoint.
// Postcondition: the contents of p will be sent to the output stream
// using the format (x,y) where x and y are the x and y coordinates
// of p.
{
    output << '(' << p.x << ',' << p.y << ')';
    return output;
}

```