```
/**
 * ******* Lasalle College Vancouver *******.
 *
 * Object Oriented Programming in C++ II
 * Week 7 - Value categories Move Semanticss
 * @author
 * Ivaldo Tributino de Sousa <ISousa@lasallecollegevancouver.com>
 */
#pragma once

// Input/output library
#include <iostream>
using std :: cout;
using std :: endl;

// Containers library
#include<vector>
using std :: vector;

// Strings library
#include <string>
using std :: string;
using std :: to_string;

// Numerics library
#include <cmath>

// Utilities library
# include <utility>
// Dynamic memory management
#include <memory>
using std :: unique_ptr;
using std :: shared_ptr;
using std :: weak_ptr;
using std :: make_unique;
using std :: make_shared;
```

SmartPtr.h

```
#pragma once

template <class T>
class SmartPtr {

private:

    T* ptr;

public:

    // Constructor
    explicit SmartPtr(T* p= nullptr); // controls unwanted implicit type conversions.

    // Destructor
    ~SmartPtr();

    // Move Constructor
    SmartPtr(SmartPtr<T>&& obj) noexcept;

    // Move Assignment operator
    SmartPtr & operator=(SmartPtr<T> && obj) noexcept;

    // Overloading derefercing operator
    T& operator*();

    // Overloading arrow operator
    T* operator->();


};

template <class T>
SmartPtr<T> :: SmartPtr(T* p) : ptr(p)
```

```
{
    cout << "Pointer Constructor Invoked"  << endl;
}

template <class T>
SmartPtr<T> :: ~SmartPtr()
{
    delete ptr;
    cout << "Pointer destroyed" << endl;
}

template <class T>
SmartPtr<T> :: SmartPtr(SmartPtr<T>&& obj) noexcept
{
    ptr = obj.ptr;
    obj.ptr = nullptr;
    cout << "Move Constructor Invoked" << endl;
}

template <class T>
SmartPtr<T> & SmartPtr<T> :: operator=(SmartPtr<T>&& obj) noexcept
{
    if (this != &obj)      // beware of self-assignment
    {
        delete ptr;       // release the old resource

        ptr = obj.ptr;    // acquire the new resource
        obj.ptr = nullptr;
    }
    cout << "Move Assignment operator invoked"  << endl;
    return *this;
}


template <class T>
T& SmartPtr<T> :: operator*()
```

```
{
    return *ptr;
}

template <class T>
T* SmartPtr<T> :: operator->()
{
    return ptr;
}
```

Polygon.h

```
class Polygon {

  private: // Private members:

    // Data Members (underscore indicates a private member variable)
    unsigned int numberSides_;

  protected: // Protected mebers:
    string solidName;

  public:  // Public members:
      /**
      * Creates a triangle.
      */
    Polygon(); // Custom default constructor

    /**
      * Creates a numberSides sided Polygon.
      */
    Polygon(int numberSides);

    /**
    * Copy constructor: creates a new Polygon from another.
    * @param obj polygon to be copied.
    */
    Polygon(const Polygon & obj); // Custom Copy constructor

    ~Polygon(); // Destructor

    /**
      * Assignment operator for setting two Polygon equal to one another.
      * @param obj Polygon to copy into the current Polygon.
      * @return The current image for assignment chaining.
      */
    Polygon & operator=(const Polygon & obj);  // Custom assignment operator;
    /**
      * Function Call Operator () Overloading:
      */
    double operator()(float lenght);

    bool operator<(const Polygon & obj);

    bool operator>(const Polygon & obj);

    /**
    * Return the polygon name by its number of sides.
    */
    string shapeName() const;

    /**
      * Gets and Sets
      */

    unsigned int getNumberSides() const;

    void setNumberSides(unsigned int n);

};
```

Polygon.cpp

```
#include "Polygon.h"

// #define Allows the programmer to give a name to a constant value before
the program is compiled
#define PI 3.14159265

Polygon :: Polygon() : numberSides_(3){
  cout << "Default Constructor Invoked"  << endl;
}

Polygon :: Polygon(int numberSides){
  (numberSides > 2)? numberSides_ = numberSides : numberSides_ = 3;
  cout << "Constructor Invoked"  << endl;
}

Polygon :: ~Polygon(){
  cout << "Polygon was destructive"  << endl;
}

// function to overload the operator
double Polygon :: operator()(float length){
  double perimeter = numberSides_*length;
  double apothem = (length)/(2*tan(PI/numberSides_));
  return perimeter*apothem/2;

}

Polygon :: Polygon(const Polygon & obj){
  numberSides_ = obj.numberSides_;
  cout << "Copy Constructor Invoked"  << endl;
}

Polygon & Polygon :: operator=(const Polygon & obj){
  numberSides_ = obj.numberSides_;
  cout << "Assignment operator invoked"  << endl;
  return *this;
}

bool Polygon :: operator <(const Polygon & obj){

  return this->numberSides_ < obj.getNumberSides();
}

bool Polygon :: operator >(const Polygon & obj){

  return this->numberSides_ > obj.getNumberSides();
}

string Polygon::shapeName() const {

  string arrayName[6] = {"triangle" , "square", "pentagon",
  "hexagon", "heptagon", "octagon"};

  string name = (numberSides_<9)? arrayName[numberSides_-3]:
to_string(numberSides_)+"_polygon";

  return name;
}

unsigned int Polygon ::getNumberSides() const {
  return numberSides_;
}

void Polygon :: setNumberSides(unsigned int n){
  numberSides_ = (n > 2)? n : 3;
}
```

PolyArray.h

```
class PolyArray
{
private:
    // Polygon* _data;
    unique_ptr<Polygon[]> _data;
    int _size;

public:

    PolyArray ();

    PolyArray (int n);

    // copy constructor
    PolyArray (PolyArray& other);

    // move constructor
    PolyArray (PolyArray&& other);

    // move assignment operator
    PolyArray& operator=(PolyArray&& other);

    // Overloading operator[]
    Polygon& operator[](int index);

    int getSize();

    void setSize(unsigned size);

    ~PolyArray() = default;

};
```

PolyArray.cpp

```
# include "polyArray.h"

PolyArray :: PolyArray ()
    : _data(new Polygon[1])
    , _size(1)
    {}

PolyArray :: PolyArray (int n)
    : _data(new Polygon[n])
    , _size(n)
{
    for(int i=0; i < _size; ++i){
        _data[i].setNumberSides(i+3);
    }
}

// Copy constructor
PolyArray :: PolyArray (PolyArray& other)
    : _data( new Polygon[other._size] )
    , _size( other._size )
{
    cout << "Copy constructor in PolyArray Invoked" << endl;
    for ( int i = 0; i < _size; ++i )
    {
        _data[i].setNumberSides(i+3);
    }

}

// Move constructor
PolyArray :: PolyArray (PolyArray&& other)
    // : _data( other._data  )
    // , _size( other._size )
    : _data(std::move(other._data))
{
    // other._data = nullptr;
    // other._size = 0;
    cout << "Move constructor in PolyArray Invoked" << endl;
}

// move assignment operator
PolyArray & PolyArray :: operator=(PolyArray && other)
{
    if (this != &other)
    {
    //     // Free the existing resource.
    //     delete[] _data;

    //     // Copy the data pointer and its size_size from the
    //     // source object.
    //     _data = other._data;
        _size = other._size;

    //     // Release the data pointer from the source object so that
    //     // the destructor does not free the memory multiple times.
    //     other._data = nullptr;
    //     other._size = 0;
        _data = std::move(other._data);
        cout << "Move assignment in PolyArray Invoked" << endl;
    }

    return *this;

}

// Overloading operator[]
Polygon& PolyArray :: operator[](int index){
    return _data[index];
}

int PolyArray :: getSize(){
    return _size;
}

void PolyArray :: setSize(unsigned size){
    _size = size;
    _data.reset(new Polygon[_size]);

    for(int i=0; i < _size; ++i){
        _data[i].setNumberSides(i+3);
    }
}

// PolyArray :: ~PolyArray ()
// {
//     delete [] _data;
// }
```

main.cpp

```
static int x = 23;

//A function that returns an lvalue.
int& getLvalue(){
  return x;
}

void passRvalue(int&& x){
    cout << ++x << endl;
}

void passLvalue(int& x){
    cout << x << endl;
}

void passAllvalue(const int& x){
    cout << x << endl;
}

int main(){

    //************************************************
    //       -------- lvalues and rvalues --------
    //************************************************
    cout << "-------- lvalues and rvalues --------" << '\n';

    x = x + 1;
    cout << &x << '\n';

    // error: cannot take the address of an rvalue of type 'int'
    // cout << &13 << '\n';

    // Expression must be an lvalue or a function designator.
    // cout << &(x+1) << '\n';
```

```cpp
    // Taking the memory address of x and setting it to y(pointer), using the & (ampersand) operator.
    int* y = &x;
    cout << y << '\n';
    cout << *y << '\n';

    // Expression must be an lvalue or a function designator.
    // int* y = &14;

    cout << "- A function that returns an lvalue "<< '\n';

    cout << getLvalue() << '\n';
    cout << &getLvalue() << '\n';

    getLvalue() = 100;

    cout << getLvalue() << '\n';
    cout << &getLvalue() << '\n';

    getLvalue()++;  // Increment operator
    cout << getLvalue() << '\n';

    cout << "- Assigned an integer directly to my reference" << '\n';
    {
    int& ref = x;
    cout << &ref << endl;
    // error: initial value of reference to non-const must be an lvalue
    // int& ref = 10;
    }

    cout << "--- Lvalue ---" << endl;
    int a = 10;
    passLvalue(a);
    // passLvalue(10);
```

```
cout << "--- Lvalue & Rvalue ---" << endl;
{
const int& ref = 10;
cout << ref << endl;
int a = 10;
passAllvalue(a);
passAllvalue(10);
}

// {
// // error: variable 'ref' declared const here
// const int& ref = 10;
// cout << ref++ << endl;
// }

cout << "- T&& (double ampersand)" << '\n';

int &&refRvalue = 101;
cout << refRvalue << '\n';
cout << &refRvalue << '\n';
refRvalue++;
cout << refRvalue << '\n';

passRvalue(14);

{

Polygon poly(4);

auto f_Lvalue = [](Polygon& p){ return p.shapeName();};

auto f_Allvalue = [](const Polygon& p){ return p.shapeName();};

auto f_Rvalue = [](Polygon&& p){ return p.shapeName();};

cout << f_Lvalue(poly) << endl;
cout << f_Allvalue(poly) << endl;
cout << f_Allvalue(Polygon(4)) << endl;

cout << f_Rvalue(Polygon(4)) << endl;
cout << f_Rvalue(std::move(poly)) << endl;
}

cout << "- std::move" << '\n';

// Error: An rValue reference cannot be pointed to a lValue.
// int &&refRvalue = x;
refRvalue = x;

cout << &refRvalue << '\n';
cout << &x << '\n';

cout << refRvalue << '\n';
cout << x << '\n';

int && refRvalue1 = std::move(x);

cout << &refRvalue1 << '\n';
cout << &x << '\n';

cout << refRvalue1 << '\n';
cout << x << '\n';



cout << "-std::unique_ptrs" << '\n';

/*
int* array1 = new int[8]{1,2,3,4,5,6,7,8};
int* array2 = new int[8];
array2 = array1; // Memory leak
cout << array1 << '\n';
```

```
    cout << array2 << '\n';
    delete[] array1; // array2 dangling pointer
    */

    int* array1 = new int[8]{1,2,3,4,5,6,7,8};
    int* array2 = new int[8];
    for(int i = 0; i<8; ++i){
        array2[i] = array1[i];
    }
    delete[] array1;


    unique_ptr<int[]> ptr1(new int[8]{1,2,3,4,5,6,7,8});
    unique_ptr<int[]> ptr2 = std::move(ptr1); // memory resource is
transferred to another unique_ptr

    for(unsigned i=0; i<8;++i){
        cout << ptr2[i] << "\n";
    }



    //************************************************
    //          ------ Move Semantics ------
    //************************************************
    cout << "-------- Move Semantics --------" << '\n';

    cout << "- Our Smart Pointer" << '\n';
    {
        SmartPtr<Polygon> ptr1(new Polygon(5));
        SmartPtr<Polygon> ptr2 = std :: move(ptr1);
        cout << ptr2->shapeName() << '\n';
        cout << &(*ptr1) << '\n';

        SmartPtr<Polygon> ptr3(new Polygon(6));
        ptr3 = std::move(ptr2);
        cout << ptr3->shapeName() << '\n';
        cout << &(*ptr2) << '\n';

    }

    //************************************************
    //          ------ PolyArray ------
    //************************************************
    cout << "------- PolyArray ---------" << '\n';

    {
        int size = 5;
        PolyArray array1(size);
        // PolyArray array2 = array1;
        PolyArray array2 = std :: move(array1);

        for(int i=0; i<size; ++i){
            cout <<"Shape Name: " << array2[i].shapeName() << ';';
            cout <<" Number of sides: " << array2[i].getNumberSides()<< ';';
            cout <<" Area: " << array2[i](4)<< '\n';
        }

    }

    cout << "- Push_back in vector<Polygon>" << '\n';
    {
        vector<Polygon> v;
        v.reserve(3); //only allocation
        v.push_back(Polygon(3));
        v.push_back(Polygon(4));
        v.push_back(Polygon(5));
    }

    cout << "- Push_back in vector<PolyArray>" << '\n';
    {
        vector<PolyArray> matrix;
        matrix.reserve(3); //only allocation
```

```
        matrix.push_back(PolyArray(3));
        matrix.push_back(PolyArray(4));
        matrix.push_back(PolyArray(5));
    }

    cout << "- PolyMatrix " << '\n';
    {
        int column = 5;
        int row = 6;

        PolyArray arrayPoly(row);
        for(int i=0; i<row; ++i){
            for(int j=0; j<column; ++j){
                cout<< arrayPoly[i](j+1) << '|';
            }
            cout << '\n';
        }
    }

    return 0;
}
```