```cpp
**
 * ******* Lasalle College Vancouver *******.
 *
 * Object Oriented Programming in C++ II
 * Week 6 - Smart Pointers
 * @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 <chrono>
// 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) noexcept; // controls unwanted implicit
type conversions.

    // Destructor
    ~SmartPtr();

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

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

    // Preventing the compiler from generating copy
    SmartPtr(const SmartPtr&) = delete;

    SmartPtr& operator=(const SmartPtr&) = delete;

};

template <class T>
SmartPtr<T> :: SmartPtr(T* p) noexcept : ptr(p)
{
    cout << "Pointer Constructor Invoked"  << endl;
}

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

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)
    double length_;
    unsigned int numberSides_;

  public:  // Public members:
    /**
      * Creates a triangle with one side measuring 1.
      */
    Polygon(); // Custom default constructor

    /**
      * Create a polygon using the following parameters:
      * @param numberSides.
      * @param  length.
      */
    Polygon(double length, unsigned int numberSides); // Custom Constructor

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

    // Polygon(Polygon&& obj); // Move Constructor

    /**
      * 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;
    Polygon operator+(const Polygon & obj); // Operators + Overloading

    /**
      * Destructor: frees all memory associated with a given Polygon object.
      * Invoked by the system.
      */
    ~Polygon(); // Destructor

     /**
      * Functions get name and get area
      */
    string shapeName();

    double area();

    /**
      * Gets and sets
      */

    void setlength(double length);

    void setNumberSides(unsigned int numberSides) ;

    double getlength();

    unsigned int getNumberSides();

};
```

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(){
  length_ = 1;
  numberSides_ = 3;
  cout << "Default Constructor Invoked"  << endl;
}



Polygon :: Polygon(double length, unsigned int numberSides){
  length_ = (length>0)? length: 1;
  numberSides_ = (numberSides>2)? numberSides : 3;
  cout << "Constructor Invoked"  << endl;
}

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

// Polygon :: Polygon(Polygon&& obj){
//   length_ = obj.length_;
//   numberSides_ = obj.numberSides_;
//   cout << "Move Constructor Invoked"  << endl;

// }

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

Polygon Polygon :: operator+(const Polygon & obj){
  Polygon A;
  A.length_ = this->length_ + obj.length_;
  A.numberSides_ = this->numberSides_ + obj.numberSides_;
  return A;
}

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

double Polygon:: area(){
  double perimeter = numberSides_*length_;
  double apothem = (length_)/(2*tan(PI/numberSides_));
  return perimeter*apothem/2;
}

string Polygon::shapeName() {

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

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

  return name;
}

void Polygon::setlength(double length) {
  if (length>0){
    length_ = length;
  }
  else{
    cout << "Please, set a value greater than 0" << endl;
  };
}

void Polygon :: setNumberSides(unsigned int numberSides) {

  if (numberSides>2){
    numberSides_ = numberSides;
  }
  else{
    cout << "Please, only set values above 2." << endl;
  };

}

double Polygon ::getlength() {
  return length_ ;
}

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

main.cpp

```cpp
#include "Polygon.h"
#include "smartPtr.h"

struct Pair
{
    string name;
    // shared_ptr<Pair> companion;
    weak_ptr<Pair> companion;

    Pair(string na) : name(na){};

    ~Pair() {cout << name << " was deleted\n"; };
};
int main(){

    //***********************************************
    //       -------- Memory leak --------
    //***********************************************
    cout << "-------- Memory leak --------" << endl;

    /* Note: If your object is in the stack memory, your destructor is going
to be called as
    soon as the function returns. If it's on the heap, the destructor is only
called when
    that delete keyword is used.*/

    cout << "- Polygon on stack" << endl;
    {
        Polygon poly;
        cout << poly.shapeName() << endl;
    }
    cout << "- Polygon on heap" << endl;
    {
        Polygon* p1 = new Polygon();
        Polygon* p2 = new Polygon(4,4); // memory leak
        cout << p1->shapeName() << endl;
        cout << p2->shapeName() << endl;
        delete p1;
    }
    {
        Polygon* p = new Polygon;  // memory leak
        Polygon square(2,4);
        p = &square;
    }

    //************************************************
    //       -------- Dangling pointer --------
    //************************************************
    cout << "-------- Dangling pointer --------" << endl;

    //Declaring two pointer variables to Polygon
    Polygon * p1;
    Polygon * p2;

    // Allocating dynamic memory in the heap
    p1 = new Polygon(3,5);
    p2 = p1;    // Having both pointers to point same dynamic memory location

    //deleting the dynamic memory location
    delete p1; // release the memory
    p1 = nullptr;

    /*
    p2 is still pointing the already deleted memory location, We call p2
dangling pointer

    If we try to release that location once more like delete p2, it will
crash after
    throwing an exception.
    */
    //**************************************************
    //   -------- Our Smart pointer in action --------
    //**************************************************
    cout << "-------- Our Smart pointer in action --------" << endl;
    {
        SmartPtr<Polygon> p1(new Polygon());
        SmartPtr<Polygon> p2(new Polygon(4,3));
        cout << p1->area() << endl;
        cout << p2->area() << endl;
    }

    //**************************************************
    //   -------- std::unique_ptrs --------
    //**************************************************
    cout << "-------- std::unique_ptrs --------" << endl;

    // not recommended  unique_ptr<int> ptr1(new int(23));
    unique_ptr<int> ptr1 = make_unique<int>(23);
    cout << ptr1 << endl;
    cout << *ptr1 << endl;

    int *ptr2;
    // ptr2 = ptr1; incompatibility
    ptr2 = ptr1.get(); // Returns the stored pointer
    cout << ptr2 << endl;
    cout << *ptr2 << endl;

    // std::unique_ptr does not have a copy constructor:
    // unique_ptr<int> ptr3 = ptr1;
    unique_ptr<int> ptr3 = move(ptr1); // memory resource is transferred to
another unique_ptr
    cout << ptr1 << endl;
    cout << ptr2 << endl;
    cout << ptr3 << endl;

    {
```

```
unique_ptr<Polygon> unPolyPtr = make_unique<Polygon>(2,10);
cout << unPolyPtr->getNumberSides() << endl;

unPolyPtr.reset(new Polygon(2,4));
cout << "Using reset" << endl;

Polygon *polyPtr;
polyPtr = unPolyPtr.release();
cout << "Using release" << endl;
cout << polyPtr->shapeName() << endl;
if (unPolyPtr == nullptr){
    cout << "null pointer" << endl;
};
delete polyPtr;
}

//************************************************
//   -------- std::shared_ptrs --------
//************************************************
cout << "-------- std::shared_ptrs --------" << endl;

{
    shared_ptr<Polygon> shPtr1 = make_shared<Polygon>(2,5);
    cout << shPtr1->shapeName() << endl;
    cout << "use_count() = " << shPtr1.use_count() << endl;
    {
        shared_ptr<Polygon> shPtr2 = shPtr1;
        cout << "use_count() = " << shPtr1.use_count() << endl;
    }
    cout << "use_count() = " << shPtr1.use_count() << endl;
}

// ------------- Issues with arrays ------------- //
cout << "------- Issues with arrays --------" << endl;

// {
    //     Polygon* polys = new Polygon[3]{Polygon(1,3), Polygon(1,4),
Polygon(1,5)};

    //     cout << polys[0].shapeName() << endl;
    //     cout << polys[1].shapeName() << endl;
    //     cout << polys[2].shapeName() << endl;

    //     delete[] polys;
    // }

    // {
    //     // Error: pointer being freed was not allocated
    //     // shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3),
Polygon(1,4), Polygon(1,5)});
    //     // Soltuion:
    //     shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3),
Polygon(1,4), Polygon(1,5)},
    //     [](Polygon *p) { delete[] p; });

    //     cout << shPtr.get()[0].shapeName() << endl;
    //     cout << shPtr.get()[1].shapeName() << endl;
    //     cout << shPtr.get()[2].shapeName() << endl;
    // }
    {
        std::shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3),
Polygon(1,4), Polygon(1,5)},
        std::default_delete<Polygon[]>());
        cout << shPtr.get()[0].shapeName() << endl;
        cout << shPtr.get()[1].shapeName() << endl;
        cout << shPtr.get()[2].shapeName() << endl;
    }
```

```
 // ------------- Circular reference ------------- //
cout << "----- Circular reference ------" << endl;
{
    shared_ptr<Pair> triangle = make_shared<Pair>("triangle");
    shared_ptr<Pair> square = make_shared<Pair>("square");

    triangle->companion = square;
    square->companion = triangle;
    cout << square.owner_before(triangle) << endl;
    cout << triangle.owner_before(square) << endl;
}

// ************************************************
//   -------- std::weak_ptrs --------
// ************************************************
cout << "-------- std::weak_ptrs --------" << endl;

weak_ptr<Polygon> wPtr1;
weak_ptr<Polygon> wPtr2;
{
    shared_ptr<Polygon> shPtr =  make_shared<Polygon>(3,7);
    cout << "use_count() = " << shPtr.use_count() << endl;
    wPtr1 = shPtr;
    if(auto temp = wPtr1.lock()){
        cout << (*temp).shapeName() << endl;
    }
    else
        std::cout << "weak_ptr is expired\n";
    wPtr2 = wPtr1;
    if(auto temp = wPtr2.lock()){
        cout << (*temp).shapeName() << endl;
    }
    else
        std::cout << "weak_ptr is expired\n";

    cout << "use_count() = " << shPtr.use_count() << endl;
```

```cpp
    }
     if(auto temp = wPtr1.lock()){
         cout << (*temp).shapeName() << endl;
     }
     else
         std::cout << "weak_ptr is expired\n";

     cout << "The polygon has already been destroyed."<< endl;



     // ************************************************
     //   -------- Performance Comparison --------
     // ************************************************
     cout << "------ Performance Comparison ------" << endl;

     auto st = std::chrono::system_clock::now();

     for(long long i = 0; i < pow(10,7) ; ++i ){
         // int *test(new int(i));
         // delete test;
         // std::unique_ptr<int> tmp(new int(i));
         // std::unique_ptr<int> tmp(std::make_unique<int>(i));
         // std::shared_ptr<int> tmp(new int(i));
         // std::shared_ptr<int> tmp(std::make_shared<int>(i));

     }
     std::chrono::duration<double> dur = std::chrono::system_clock::now() -
st;
     std::cout << "time: " << dur.count() << " seconds" << std::endl;

     return 0;
}
```