```
/**
 * ******* Lasalle College Vancouver *******.
 *
 * Object Oriented Programming in C++ II
 * Week 10 - Multithreading
 * @author
 * Ivaldo Tributino de Sousa <ISousa@lasallecollegevancouver.com>
*/
#pragma once

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

// Containers library
#include<vector>
#include<list>
#include<map>
#include <queue>
#include <stack>

using std :: vector;
using std :: list;
using std :: map;
using std :: queue;
using std::pair;

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

```
// Numerics library
#include <cmath>
#include <numeric>
#include <random>

// Utilities library
# include <functional>
# include <chrono>
using std::chrono::system_clock;
# include <ctime>
// 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;
// Error handling
# include <stdexcept>

// Algorithms library
// #include<algorithm>
//Non-modifying sequence operations
using std::for_each;

// C compatibility headers
#include <cassert>

// Thread support library
# include <condition_variable>
# include <future>
# include <thread>
# include <mutex>
```

thread_guard.h

```
#include "library.h"

class thread_guard
{
    std::thread& _t;

public:
    //Explicit, that is, it cannot be used for implicit conversions and copy-
initialization.
    explicit thread_guard(std::thread& t): _t(t) {}

    ~thread_guard()
    {
        if(_t.joinable()) _t.join();
        cout << "thread object was destroyed." << endl;
    }
    // to ensure that they’re not automatically provided by the compiler.
    thread_guard(thread_guard const&)=delete;

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

};
```

Polygon.h

```
#pragma once
#include "library.h"

class Polygon{

  private: // Private members:

    // Data Members (underscore indicates a private member variable)
    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(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

   double operator()(double length) const; // Function operator

    bool operator<(const Polygon& obj) const; // operator < overloading

    bool operator>(const Polygon& obj) const; // operator > overloading
    /**
      * 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;

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

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

    /**
      * Gets and sets
      */
    void setNumberSides(unsigned int numberSides);

    unsigned int getNumberSides() const;

};
```

Polygon.cpp

```
#include "Polygon.h"
#define PI 3.14159265
Polygon :: Polygon() : numberSides_(3)
{
  cout << "Default Constructor Invoked"  << endl;
}

Polygon :: Polygon(unsigned int numberSides) : numberSides_(numberSides)
{
  assert(("Polygon is a geometrical figure with three or more sides.",
numberSides_ >= 3));

  cout << "Constructor Invoked"  << endl;

}

double Polygon :: operator()(double length) const{
  double perimeter = numberSides_*length;
  double apothem = (length)/(2*tan(PI/numberSides_));
  return perimeter*apothem/2 ;

}

bool Polygon :: operator<(const Polygon& obj) const {
  return this->numberSides_ < obj.numberSides_;
}

bool Polygon :: operator>(const Polygon& obj) const{
  return this->numberSides_ > obj.numberSides_;
}

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;
}

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

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 :: setNumberSides(unsigned int numberSides){

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

}

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

main.cpp

```
#include "thread_guard.h"
#include "Polygon.h"

std::mutex m;
std::condition_variable cv;

bool ready = false;
static const int NUM = 100000000;

void function1();

void function2();

void function3(char c);

void push(vector<int>& v);

void pop(vector<int>& v);

void print_name();

void input_name();

long long getDotProduct(std::vector<int>& v,std::vector<int>& w);

long long getDotProductFuture(std::vector<int>& v,std::vector<int>& w);

int main(){

    // ************************************************
    //         ---- Creating Threads -----
    // ************************************************

    function1();
    function2();
```

```
std::thread t1(function1);
std::thread t2(function2);

t1.join();
t2.join();

if(!t1.joinable()){
    cout << "thread object is no longer joinable" << '\n';
}

//*****************************************************
// - Initializing thread with an object -
//*****************************************************
cout << "--- Initializing thread with an object ---" << '\n';

{
    Polygon square(4);
    std::thread t3(square, 8);

    if(t3.joinable()) t3.join();
}
cout << "- Using std::ref ---" << '\n';

{
    Polygon square(4);
    std::thread t3(std::ref(square), 8);

    if(t3.joinable()) t3.join();
}

// cout << "--- USING REF AND RAII ---" << '\n';
{
    Polygon square(4);
    std::thread t(std::ref(square), 8);
    thread_guard tg(t);
```

```
}

//************************************************
//   - Transferring ownership of a thread  -
//************************************************
cout << "--- Transferring ownership of a thread ---" << '\n';
{
    std::thread t_1(function1);
    std::thread t_2 = std::move(t_1);
    if(!t_1.joinable()){
        cout << "thread t_1 is no longer joinable" << '\n';
    }
    if(!t_2.joinable()){
        cout << "thread t_2 is no longer joinable" << '\n';
    };
    thread_guard g(t_2);
}

// ************************************************
//   - std::thread::hardware_ concurrency()  -
// ************************************************
cout << "--- How Many Threads Should We Create? ---" << '\n';

{
unsigned int c = std::thread::hardware_concurrency();
 cout << "Number of cores: " << c << '\n';;
}

//************************************************
//              ---    Thread ID ---
//************************************************
cout << "--- Thread ID ---" << '\n';
using SC = std::chrono::steady_clock;
auto dealine = SC::now() + std::chrono::milliseconds(1);
```

```
    // Accesses to atomic objects can establish synchronization between
threads.
    std::atomic<int> counter = 0;

    //create 3 different threads
    std::thread t_1([&](){
        while(SC::now()<dealine)
            printf("t_1: %d\n", ++ counter);

    });
    std::thread t_2([&](){
        while(SC::now()<dealine)
            printf("t_2: %d\n", ++ counter);

    });
    std::thread t_3([&](){
        while(SC::now()<dealine)
            printf("t_3: %d\n", ++ counter);

    });
    //get id of threads t_1, t_2 and t_3
    std::thread::id id1 = t_1.get_id();
    std::thread::id id2 = t_2.get_id();
    std::thread::id id3 = t_3.get_id();

    if(t_1.joinable())
    {
        t_1.join();
        cout << "Thread with id " << id1 << " is terminated" << '\n';
    }
    if (t_2.joinable())
    {
        t_2.join();
        cout << "Thread with id " << id2 << " is terminated" << '\n';
    }
    if (t_3.joinable())
```

```
    {
        t_3.join();
        cout << "Thread with id " << id3 << " is terminated" << '\n';
    }

    std::thread::id main = std::this_thread::get_id();
    cout << "Main thread id is :" << main << '\n';

    //*************************************************
    //         -- Using mutexes in C++ --
    //*************************************************
    cout << "--- Using mutexes in C++ ---" << '\n';
    {
    // vector<int> v;

    // std::thread thr_push(push,std::ref(v));
    // std::thread thr_pop(pop,std::ref(v));

    // if (thr_push.joinable()) thr_push.join();
    // if (thr_pop.joinable()) thr_pop.join();

    // vector<std::thread> threads;

    // char characters[] = {'+', '-', '*', '/'};
    // for(unsigned i=0;i < 4 ;++i)
    // {
    //     threads.push_back(std::thread(function3,characters[i]));
    // }
    //
for_each(threads.begin(),threads.end(),std::mem_fn(&std::thread::join)); //
std::mem_fn invokes the member function pointed by pm.

    }

    //*************************************************
    //  - Waiting for an event or other condition  -
```

```
    //************************************************
    // cout << "--- Waiting for an event or other condition ---" << '\n';

    // // SOURCE : https://www.tutorialcup.com/cplusplus/multithreading.htm
    // cout << "The id of current thread is " << std::this_thread::get_id()
<< '\n';

    // //get current time
    // time_t timet = system_clock::to_time_t(system_clock::now());

    // //convert it to tm struct
    // struct tm * time = localtime(&timet);
    // cout << "Current time: " << std::put_time(time, "%X") << '\n';

    // std::cout << "Waiting for the next minute to begin...\n";
    // time->tm_min++, time->tm_sec = 0;
    // //sleep until next minute is not reached
    //
std::this_thread::sleep_until(system_clock::from_time_t(mktime(time)));
    // cout << std::put_time(time, "%X") << " reached!\n";

    // //sleep for 5 seconds
    // std::this_thread::sleep_for(std::chrono::seconds(5));

    // //get current time
    // timet = system_clock::to_time_t(system_clock::now());

    // // convert it to tm struct
    // time = std::localtime(&timet);
    // cout << "Current time: " << std::put_time(time, "%X") << '\n';

    // ----------------------------------------
    cout << "--- By condition " << '\n';

    std::thread thr_print(print_name);
    std::thread thr_input(input_name);
```

```
    if(thr_print.joinable()) thr_print.join();
    if(thr_input.joinable()) thr_input.join();

    //*************************************************
    //         --- Using std::future ---
    //*************************************************
    cout << "--- Using std::future ---" <<'\n';
    // Example from: https://www.modernescpp.com/index.php/asynchronous-
function-calls

    // get NUM random numbers from 0 .. 100
    std::random_device seed;

    // generator
    std::mt19937 engine(seed()); // pseudo-random generator

    // distribution
    std::uniform_int_distribution<int> dist(0,100);

    // fill the vectors
    std::vector<int> v, w;
    v.reserve(NUM);
    w.reserve(NUM);
    for (int i=0; i< NUM; ++i){
        v.push_back(dist(engine));
        w.push_back(dist(engine));
    }

    // measure the execution time
    system_clock::time_point start = system_clock::now();
    cout << "getDotProduc(v,w): " << getDotProduct(v,w) << '\n';
    std::chrono::duration<double> dur  = system_clock::now() - start;
    cout << "Total Time Taken = " << dur.count() << '\n';

    // measure the execution time using Future
```

```
    start = std::chrono::system_clock::now();
    cout << "getDotProducFuture(v,w): " << getDotProductFuture(v,w) << '\n';
    dur  = std::chrono::system_clock::now() - start;
    cout << "Total Time Taken using Future = " << dur.count() << '\n';

    return 0;

}

void function1()
{
    for(unsigned i =0; i<10; i++){
        for(unsigned j=0; j<10; j++){
            cout << '1';
        }
        cout << '\n' ;
    }
}

void function2()
{
    for(unsigned i =0; i<10; i++){
        for(unsigned j=0; j<10; j++){
            cout << '2';
        }
        cout << '\n' ;
    }
}

void function3(char c)
{
    std::lock_guard<std::mutex> guard(m);
    std::this_thread::sleep_for(std::chrono::seconds(2));
    for(unsigned i =0; i<200; i++){
        cout << c << " ";
    }
```

```
}

void push(vector<int>& v)
{
    m.lock();
    for (int i = 0; i < 10; ++i)
    {
      cout << "Push " << i << '\n';
      std::this_thread::sleep_for(std::chrono::seconds(1));
      v.push_back(i);
    }
    m.unlock();

}
void pop(vector<int>& v)
{
    m.lock();
    for (int i = 0; i < 10; ++i)
    {
        if(v.size()!=0)
        {
            int val = v.back();
            v.pop_back();
            cout << "Pop " << val << '\n';
        }
        std::this_thread::sleep_for(std::chrono::seconds(3));
    }
    m.unlock();
}

void print_name () {
  std::unique_lock<std::mutex> lck(m);
  while (!ready) cv.wait(lck);
  string name;
  std::cin >> name;
  std::cout << "My name is: " << name << '\n';
```

```
}

void input_name() {
  std::unique_lock<std::mutex> lck(m);
  cout << "Enter your name: ";
  ready = true;
  cv.notify_one();
}

long long getDotProduct(std::vector<int>& v,std::vector<int>& w){
  return std::inner_product(v.begin(),v.end(),w.begin(),0LL);
}

long long getDotProductFuture(std::vector<int>& v, std::vector<int>& w){

    auto future1 = std::async([&]{return
std::inner_product(&v[0],&v[v.size()/4],&w[0],0LL);});
    auto future2 = std::async([&]{return std::inner_product(&v[v.size()/
4],&v[v.size()/2],&w[v.size()/4],0LL);});
    auto future3 = std::async([&]{return std::inner_product(&v[v.size()/
2],&v[v.size()*3/4],&w[v.size()/2],0LL);});
    auto future4 = std::async([&]{return
std::inner_product(&v[v.size()*3/4],&v[v.size()],&w[v.size()*3/4],0LL);});

    return future1.get() + future2.get() + future3.get() + future4.get();
}
```