Midterm Exam

CIS-687 OBJECT ORIENTED DESIGN

PROF. SCOTT ROUECHE

5/26/2022

Anthony Redamonti
SYRACUSE UNIVERSITY

Question 1)

a) What are the stages of C++ compilation? (You can just list them.)

1. Preprocessing
2. Compilation
3. Assembler
4. Linking

b) Explain what happens during the preprocessor stage of compilation.

The preprocessor takes C++ source code and deals with the #includes, #defines and other preprocessor directives. The output of this step is a "pure" C++ file without pre-processor directives. All the header files are combined in this step.

c) What relationship does (b) have for the reason we need to use #ifndef statements in our header files?

#IFNDEF header guards are important as we do not want the same header file included more than once in the "pure" C++ output file, which would create cyclical dependencies.

d) What implications does the way pre-compilation work have on Template classes?

Template classes must be defined with the source code for all their methods in the header file. If we put the method implementation into a CPP file, the compiler would not have access to the template class's source file, since it compiles one file at a time. The main.cpp file would have no way of knowing how to use the template class.

Question 2)

a) What is the difference between implicit and explicit dynamic library linking?

Implicit dynamic library loading is when the DLL is linked by the linker at the same time as the executable file that is going to use it. It is known as ahead of time, and a .lib file is used for the DLL loading.

Explicit dynamic library loading is when the DLL is not known at compilation time. The program must manually load the methods it intends to use into the process address space.

b) What are the tradeoffs between the two?

Implicit dynamic loading is similar to a static library in that the .lib file is verified to contain all methods defined in the header file. The program will not run if an incorrect .lib file is being used. The tradeoff is slower startup time for a program, since all DLL's are loaded at application startup.

An explicitly linked DLL does not have to load all DLL's at the start of the application. It can choose which to load in different scenarios (programmed by the developer). One issue is that the developer must be more careful to call FreeLibrary after using LoadLibrary. Failure to do so would result in a failure to load the library at a different point in the same process.

c) When would you choose explicit over implicit DLLs?

One reason to use explicit linking over implicit is that the DLL name might not be known at compilation time. We may need to ask the user to provide it. Another reason to use explicit linking is to try to recover from failing to load the library. An explicitly linking library can attempt to handle this error. An implicitly linked library would not even load if this error were present.

Maybe to handle certain situations, we would need to load methods from a DLL. Only explicit loading has this ability, as implicit loading loads all the DLL's when the application loads.

Question 4)

Write a templated class Matrix<T> with the following methods:

a) Constructor takes the initial x, y number of elements in the matrix.
b) Provides methods for getting/setting a single entry.
c) Provides copy constructor.
d) Provides the ability to take another matrix add them together if the dimension match.

```
#include<iostream>
#include<vector>
using std::getchar;
using std::vector;
using std::cout;
using std::endl;

template <typename T> class Matrix {

public:

    // initialization constructor
    Matrix(const size_t xNum, const size_t yNum) {

        // assign the private data member variables.
        numberOfRows = xNum;
        numberOfColumns = yNum;

        // set the number of rows in the matrix.
        matrixVector.size = numberOfRows;

        // set the size of all columns to yNum
        for (size_t i = 0; i < numberOfRows; i++) {
            matrixVector[i].size = numberOfColumns;
        }
    }

    // += operator overloading. Add the two matrices together if their dimensions
match.
    Matrix& operator += (const Matrix& secondMatrix) {
        try {
            // if the two dimensions match, add the matrices together.
            if ((numberOfColumns == secondMatrix.numberOfColumns) && (numberOfRows
== secondMatrix.numberOfRows)) {
                for (size_t i = 0; i < numberOfRows; i++) {
                    for (size_t j = 0; j < numberOfColumns; j++) {
                        matrixVector[i][j] = matrixVector[i][j] +
secondMatrix[i][j];
                    }
                }
            }
        }
        catch (...) {
            cout << "\nException occurred in Matrix += operator: " << endl;
            throw;
        }
    }
```

```
    // copy constructor
    Matrix(const Matrix& oldObj) {

        // overwrite the private data members
        numberOfRows = oldObj.numberOfRows;
        numberOfColumns = oldObj.numberOfColumns;

        // clear the contents of the current vector
        matrixVector.clear();

        // set the number of rows in the matrix.
        matrixVector.size = numberOfRows;

        // set the size of all columns to yNum
        for (size_t i = 0; i < numberOfRows; i++) {
            matrixVector[i].size = numberOfColumns;
        }

        // perform a deep copy of the elements
        for (size_t i = 0; i < numberOfRows; i++) {
            for (size_t j = 0; j < numberOfColumns; j++) {
                matrixVector[i][j] = oldObj[i][j];
            }
        }
    }

    // retrieve the entry from the matrix.
    T getEntry(size_t row, size_t column) {
        if ((row >= numberOfRows) || (column >= numberOfColumns) || (column < 0) ||
(row < 0)) {
            cout << "Please do not request an entry outside the bounds of the
matrix." << endl;
            return;
        }
        else {
            return matrixVector[row][column];
        }
    }

    // set an entry in the matrix.
    void setEntry(size_t row, size_t column, T value) {
        try {
            if ((row >= numberOfRows) || (column >= numberOfColumns) || (column < 0)
|| (row < 0)) {
                cout << "Please do not request an entry outside the bounds of the
matrix." << endl;
                return;
            }
            else {
                matrixVector[row][column] = value;
            }
        }
        catch (...) {
            cout << "\nException occurred in Matrix::setEntry method." << endl;
            throw;
        }
    }
```

```
    // destructor
    ~Matrix() {

        // clear the multidimensional vector
        matrixVector.clear();

    }

private:

    vector<vector<T>> matrixVector;
    size_t numberOfRows;
    size_t numberOfColumns;
};
```

Question 5)

a) What are the two main performance-related advantages of using threads?

Multitasking: Threads take advantage of the multiple cores in a processor. A processor can manage multiple threads in a process through context switching and pipelining. Multiple cores in the processor can execute separate threads at the same time. It leads to a more responsive program with higher throughput.

Shared data: Because threads belong to the same process, they can have shared data – divide and conquer. A large amount of input data can be shared among multiple threads and processed in less time than using one thread to process the same data.

b) Spurious Wake

i) What is a spurious wake?
A spurious wake is when a sleeping thread is woken up prematurely by the operating system. Usually this is due to scheduling queues and the amount of time the thread spends sleeping without seeing the processor.

ii) What can cause them?
A spurious wake can be caused by the operating system managing the threads and making sure that each one receives an appropriate amount of processing time.
They can also be caused in multithreaded applications when one thread changes the state of the conditional variable while another thread is running.

iii) How do we protect against them in our code?
To prevent spurious wakes, use conditional variables with predicate values which must be checked for the thread to by allowed to continue. If the predicate condition is not met, the thread will go back to sleep.

c) There is a program where N threads have been assigned ids. There exists a function *Foo* which the threads must call in the order that their ids have been assigned. Threads must wait until all other threads have called foo before they can continue onwards (they should wait in ExitFoo until all other threads have finished).

```
void ThreadFunction(int id,  … ) {
```

```
        …
        EnterFoo(...);
        Foo();
        ExitFoo(...);
        …
    }
```

Implement the EnterFoo() and ExitFoo() methods using mutex, condition variables and any other variables you see fit. Threads will be created during ThreadFunction, but you may specify additional arguments for ThreadFunction, EnterFoo and ExitFoo methods, if necessary.

```
#include <condition_variable>
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex mutex1;
std::mutex mutex2;
std::condition_variable conditionVariable1;
std::condition_variable conditionVariable2;
using std::endl;
using std::cout;
using std::vector;

bool beingUsed{ false };
int threadRun{ 0 };

void Foo(int x, int totalNumberOfThreads);
void ExitFoo(int x, int totalNumberOfThreads);


void EnterFoo(int threadNumber, int totalNumberOfThreads){

    std::unique_lock<std::mutex> lck(mutex1);
    conditionVariable1.wait(lck, []{ return !beingUsed; });   // (4)

    beingUsed = true; // this thread is using Foo.

    cout << threadNumber << " wants to enter foo " << endl;

    Foo(threadNumber, totalNumberOfThreads);

    beingUsed = false;

    lck.unlock();
    conditionVariable1.notify_one();
}

void Foo(int x, int totalNumberOfThreads) {

    // increment the global variable.
    threadRun = threadRun + 1;
```

```
    cout << x << " has entered Foo." << endl;
    cout << threadRun << endl;
}

void ExitFoo(int x, int totalNumberOfThreads){

    std::unique_lock<std::mutex> lck(mutex2);
    conditionVariable2.wait(lck, [&] { return (threadRun == totalNumberOfThreads);
});   // (4)

    // notify all the threads to exit.
    conditionVariable2.notify_all();

    cout << "All threads are exiting." << endl;
                        // (3)
}

void ThreadFunction(int x) {

    vector<std::thread> threadVector;

    // create x number of threads and call enterfoo
    for (int i = 0; i < x; i++) {
        std::thread thread1(EnterFoo, i, x);

        std::thread thread2(ExitFoo, i, x);

        threadVector.push_back(thread1);
        threadVector.push_back(thread2);
    }

    for (int i = 0; i < threadVector.size(); i++) {
        threadVector[i].join();
    }


}

int main(){
    ThreadFunction(5);
}
```

Question 6)

a) What are the main differences between a lambda expression and a C-style function pointer?

Function Pointers: Functions that we can pass around, even though we may not have the name at compile time. Functions exist in the text segment of the process. They have an address, which can be stored in a pointer. We can pass functions as parameters to other functions.

Lambdas: an anonymous function object. Function pointers require a function to be defined ahead of time. Then we create a pointer to that function and pass the pointer to another function. Instead of writing all these lines of code, we can create a lambda expression, which outlines the function behavior, replacing the function pointer. We can also pass them into a function pointer if desired.

b) What are the main differences between a lambda expression and a functor?

Functors: A class that has overridden the () operator. Allows instances of the class to be treated as a function. Functors have their own type. Can contain member variables that remember their state. We can pass in parameters. We can overload the () operator (return different types / accept different arguments).

A lambda expression is not an instance of a class, and therefore does not have any static memory between usages.

c) What is the problem in the following snippet of code?

```
class Adder {
public:
    Adder(int x) {
        x_ = x;
    }
    int Add(int y) {
        return x_ + y;
    }
private:
    int x_;
};

std::function<int(int)> Function (int x) {
    Adder adder(x);
    return  [&](int x) { return adder.Add(x);};
```

```
}
```

While the program below will compile and run, it does not make sense.

```
#include <iostream>
#include <functional>

using std::function;
using std::endl;
using std::cout;

class Adder {
public:
    Adder(int x) {
        x_ = x;
    }
    int Add(int y) {
        return x_ + y;
    }
private:
    int x_;
};

std::function<int(int)> Function(int x) {
    Adder adder(x);
    return  [&](int x) { return adder.Add(x); };
}

int main(void) {
    std::function<int(int)> funcPointer = Function(1);

    funcPointer(1);
}
```

The function pointer is returned from the Function method. However, the return value from the lambda function it is calling is lost forever. We would want to rewrite this code so that the function pointer is passed as an argument into the Function method, and Function should return an integer. Then we could pass the lambda expression as an argument into the Function method.