```
//buttonEventList.h
#pragma once

#include <vector>
#include <string>
using namespace std;

class CButtonEventList
{
public:
	vector<string> event_tokens;
	CButtonEventList(void);
	~CButtonEventList(void);
};
CButtonEventList::CButtonEventList(void)
{
}

CButtonEventList::~CButtonEventList(void)
{
}


#include "ButtonEventList.h"

CButtonEventList::CButtonEventList(void)
{
}

CButtonEventList::~CButtonEventList(void)
{
	event_tokens.clear();
}
////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//controller
#pragma once

#include "ElevatorEvent.h"
#include "Motor.h"
#include "ButtonEventList.h"
#include "RequestQueue.h"
#include "Logger.h"


class CController
{
public:
	CController(void);
	~CController(void);

	unsigned long Run(void);
	bool HasRequests(void);
	void MarkingEvent(CButtonEventList btn_event_lst);
	void LeapOneStep(void);
	unsigned int SingleMarker(string id, unsigned int flag);

	char GetCurrentDirection();
	char GetCurrentFloor();

	CMotor motor;
	CRequestQueue logic_queue;
	CElevatorEventManager elevator_event;
	CLogger log;

private:
	unsigned int Hash(string event_name);


public:
	void Idle(void);
};

#include "Controller.h"

using namespace std;

CController::CController(void)
{
}

CController::~CController(void)
{
}
```

```
unsigned long CController::Run(void)
{
	while( !elevator_event.Empty()   || NoRequests() )
	{
		MarkingEvent(elevator_event.GetEventTimeSlice());
		LeapOneStep();

	}
	return 0;
}

bool CController::NoRequests(void)
{
	return logicQueue.NoRequestInQueue();
}

void CController::MarkingEvent(CButtonEventList btn_event_lst)
{
	//

}

void CController::LeapOneStep(void)
{
}

unsigned int CController::SingleMarker(unsigned int id, unsigned int flag)
{
	// find element by hashed id
	return logicQueue.Mark(id, flag);
}

unsigned int CController::Hash(string event_name)
{

	unsigned int r = 0;
	for (unsigned int i = 0; i<event_name.length(); i++)
		r = r*100+event_name.at(i);

	return r;
}

#include "Controller.h"
#include "Global.h"


using namespace std;

CController::CController(void)
```

```
{
}

CController::~CController(void)
{
}

unsigned long CController::Run(void)
{
	while( !elevator_event.Empty()   || HasRequests() )
	{
		MarkingEvent(elevator_event.GetEventTimeSlice());
		LeapOneStep();

	}
	return motor.dist.ShowDistance();
}

bool CController::HasRequests(void)
{
	return logic_queue.HasRequestInQueue();
}

void CController::MarkingEvent(CButtonEventList btn_event_lst)
{
	// process events and build a set of requests. requests are represented by a circularly linked list
	for (unsigned int i=0; i<btn_event_lst.event_tokens.size(); i++)
	{
		// marking requests in terms of strings like "3U", "1D" or "5"
		SingleMarker(btn_event_lst.event_tokens[i], HAS_REQUEST);

	}

}

void CController::LeapOneStep(void)
{
	// where am i
	char current_floor = GetCurrentFloor();
	char current_direction = GetCurrentDirection();

	// where is the next marked request
	if ( logic_queue.HasRequestInQueue() )
	{
		CRequest r = logic_queue.GetNextRequest();
		char direction_next_request = r.direction;
		char floor_next_request = r.floor;
```

```
if (current_direction == 'U')
{
        // climb to the summit
        if ( floor_next_request > current_floor )
        {       // one step up
                logic_queue.RollToNext();
                motor.OneStepUp();
                log.LogMessage(
                        ::format(string("U %3d |->\n"), current_floor)
                );


        }

        if ( floor_next_request < current_floor )
        {       // right-about-face (change direction)
                logic_queue.ChangeDirectionTo(direction_next_request, current_floor);
        }

}


if (current_direction == 'D')
{
        // down to the hell
        if ( floor_next_request < current_floor )
        {       // one step down
                logic_queue.RollToNext();
                motor.OneStepDown();
                log.LogMessage(
                        ::format(string("D %3d <-|\n"), current_floor)
                );

        }

        if ( floor_next_request > current_floor )
        {       // right-about-face (change direction)
                logic_queue.ChangeDirectionTo(direction_next_request, current_floor);
        }

}




if (current_floor == floor_next_request)
{
        // Beep and Stop
        logic_queue.ChangeDirectionTo(direction_next_request, current_floor);
        logic_queue.ClearCurrentRequest();
```

```
                    log.LogMessage(
                        ::format(string("[Stop]----------%d\n"), current_floor )
                    );
                }



        }

        Idle();


}

unsigned int CController::SingleMarker(string id, unsigned int flag)
{
        return logic_queue.Mark(id, flag);
}

unsigned int CController::Hash(string event_name)
{

        return ::FastHash(event_name);
}

char CController::GetCurrentDirection()
{
        return logic_queue.GetCurrentDirection();
}

char CController::GetCurrentFloor()
{
        return logic_queue.GetCurrentFloor();
}

void CController::Idle(void)
{
}


////////////////////////////////////////////////
```

```
//Distance
#pragma once

class CDistanceMeter
{
public:
	CDistanceMeter(void);
	  ~CDistanceMeter(void);
	  unsigned long Increase(unsigned long step_length = 1);
	  unsigned int ShowDistance(void);
private:
	  unsigned int distance;
};
CDistanceMeter::CDistanceMeter(void)
{
	  distance=0;
}
CDistanceMeter::~CDistanceMeter(void)
{
}
unsigned long CDistanceMeter::Increase(unsigned long step_length)
{
	  distance += step_length;
	  return distance;
}

unsigned int CDistanceMeter::ShowDistance(void)
{
	  return distance;
}


#include "Distance.h"

CDistanceMeter::CDistanceMeter(void)
: distance(0)
{
}
CDistanceMeter::~CDistanceMeter(void)
{
}
unsigned long CDistanceMeter::Increase(unsigned long step_length)
{
	  distance += step_length;
	  return distance;
}
unsigned int CDistanceMeter::ShowDistance(void)
{
	  return distance;
```

```
}
////////////////////////////////////
```

ElevatorEvent.h

```
#pragma once

#include "ButtonEventList.h"
#include <fstream>
using namespace std;
const int LINE_LENGTH = 4096;

class CElevatorEventManager
{
public:
	CElevatorEventManager(void);
	~CElevatorEventManager(void);

	void Tokenize( const string& str,
				vector<string>& tokens,
				const string& delimiters = " ");

	bool Empty(void);
	CButtonEventList GetEventTimeSlice(void);
	void SetEventSource(ifstream	*evtsrc);

private:
	ifstream	*event_source;

};

CElevatorEventManager::CElevatorEventManager(void)
{
}

CElevatorEventManager::~CElevatorEventManager(void)
{
}

bool CElevatorEventManager::Empty(void)
{
	return false;
}

CButtonEventList CElevatorEventManager::GetEventTimeSlice(void)
{
	// build a ButtonEventlist object from ifstream
	return CButtonEventList();
}

void CElevatorEventManager::SetEventSource(ifstream * evtsrc)
{
```

```
	event_source = evtsrc;
}


// elevator.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Controller.h"
#include <iostream>
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	CController elevator;

	ifstream event_source ("EVENTS.txt", ios::in);
	elevator.elevator_event.SetEventSource( &event_source);
	elevator.Run();

	elevator.log.LogMessage(::format("Odometer: %4d\n",
		elevator.motor.dist.ShowDistance())
	);


	return 0;
}
```

```
#include "ElevatorEvent.h"

CElevatorEventManager::CElevatorEventManager(void)
{
}

CElevatorEventManager::~CElevatorEventManager(void)
{
}
bool CElevatorEventManager::Empty(void)
{
	return event_source->eof();
}
void CElevatorEventManager::Tokenize(const string& str,
                                     vector<string>& tokens,
                                     const string& delimiters )
{
    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(delimiters, 0);
    // Find first "non-delimiter".
    string::size_type pos     = str.find_first_of(delimiters, lastPos);

    while (string::npos != pos || string::npos != lastPos)
    {
        // Found a token, add it to the vector.
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        // Skip delimiters.  Note the "not_of"
        lastPos = str.find_first_not_of(delimiters, pos);
        // Find next "non-delimiter"
        pos = str.find_first_of(delimiters, lastPos);
    }
}
CButtonEventList CElevatorEventManager::GetEventTimeSlice(void)
{
	// build a ButtonEventlist object from ifstream
	CButtonEventList btn_event_list;
	char s[LINE_LENGTH];
	if ( event_source->getline(s,LINE_LENGTH))
	{
		Tokenize(string(s), btn_event_list.event_tokens);
	}

	return btn_event_list;
}

void CElevatorEventManager::SetEventSource(ifstream   *evtsrc)
{
```

```
        event_source = evtsrc;
}

//////////////////////
//Global
#include "ElevatorEvent.h"

CElevatorEventManager::CElevatorEventManager(void)
{
}

CElevatorEventManager::~CElevatorEventManager(void)
{
}



bool CElevatorEventManager::Empty(void)
{

        return event_source->eof();
}

void CElevatorEventManager::Tokenize(const string& str,
                                     vector<string>& tokens,
                                     const string& delimiters )
{
    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(delimiters, 0);
    // Find first "non-delimiter".
    string::size_type pos     = str.find_first_of(delimiters, lastPos);

    while (string::npos != pos || string::npos != lastPos)
    {
        // Found a token, add it to the vector.
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        // Skip delimiters.  Note the "not_of"
        lastPos = str.find_first_not_of(delimiters, pos);
        // Find next "non-delimiter"
        pos = str.find_first_of(delimiters, lastPos);
    }
}



CButtonEventList CElevatorEventManager::GetEventTimeSlice(void)
{
        // build a ButtonEventlist object from ifstream
        CButtonEventList btn_event_list;
        char s[LINE_LENGTH];
        if ( event_source->getline(s,LINE_LENGTH))
```

```
        {
                Tokenize(string(s), btn_event_list.event_tokens);
        }

        return btn_event_list;
}

void CElevatorEventManager::SetEventSource(ifstream    *evtsrc)
{
        event_source = evtsrc;
}


#include "Global.h"

unsigned int FastHash(string event_name)
{

        unsigned int r = 0;
        for (unsigned int i = 0; i<event_name.length(); i++)
                r = (r << 8) + event_name.at(i);

        return r;
}



////////////////////////////////////
```

```
//Logger
#pragma once

#include <list>
#include <iostream>
#include <string>

using namespace std;

// build a formated string with printf-like input style
string format( const string msg, ... );

class CLogger
{
public:
	CLogger(void);
	~CLogger(void);

public:
	// push a string into a stream. we use the cout as default one.
	void LogMessage(string msg, ostream& stream_out = cout);

	// During the how logging process, all string messages are packed into a list
	// we could export all logged messages somewhere, ie. a file stream.
	void Export(ostream& stream_out);

	// log large data blocks.
	void LogBlock(unsigned char* block, unsigned int blocksize);

	// empty and clear all messages
	void ClearLog(void);

private:
	// all log objects share the same logsheet
	 static list<string> logsheet;



};


#include "StdAfx.h"
#include "Logger.h"
#include <cstdarg>

#define BUFF_SIZE 4096
#define MAX_COUNTER_VAL (4096-10)
```

```
string format( const string msg, ... )
{
	va_list ap;
	char text[BUFF_SIZE] = {0,};

	va_start(ap, msg);
	vsnprintf_s(text, BUFF_SIZE, MAX_COUNTER_VAL, msg.c_str(), ap);
	va_end(ap);

	return string( text );
}


list<string> CLogger::logsheet;

CLogger::CLogger(void)
{
}
CLogger::~CLogger(void)
{
}
void CLogger::ClearLog(void)
{
	logsheet.clear();
}
void CLogger::LogMessage(string msg, ostream& stream_out)
{
	logsheet.push_back(msg);
	stream_out << msg;
}
void CLogger::Export(ostream& stream_out)
{
	list<string>::iterator iter;
	for (iter = logsheet.begin(); iter != logsheet.end(); iter++)
		stream_out << *iter;
}
void CLogger::LogBlock(unsigned char* block, unsigned int blocksize)
{
	string s = "";
	LogMessage("\n\nBlock dump:");
	for (unsigned int i = 0; i<blocksize; i++)
	{
		if ((i%16) == 0)
		{

			LogMessage(s + string("\n"));
			s = "";
		}
		s += format("%02X ",block[i]);
```

```
        }
        LogMessage(s + string("\n"));
}

////////////////////////////////////////////////////////
//Motor
#pragma once


#include "Distance.h"



class CMotor
{
public:
        CDistanceMeter dist;

        CMotor(void);
        ~CMotor(void);
        int OneStepUp(void);
        int OneStepDown(void);
};
CMotor::CMotor(void)
{
}

CMotor::~CMotor(void)
{
}
int CMotor::OneStepUp(void)
{
        return dist.Increase(1);
}
int CMotor::OneStepDown(void)
{
        return dist.Increase(1);
}

#include "Motor.h"
CMotor::CMotor(void)
{
}
CMotor::~CMotor(void)
{
}
int CMotor::OneStepUp(void)
{
        return dist.Increase();
}
int CMotor::OneStepDown(void)
```

```
{
	return dist.Increase();
}


//////////////////////////////////////////////////////////
//Request
#pragma once
#include <string>

using namespace std;

class CRequest
{
public:
	CRequest(string Id, unsigned int flag);
	~CRequest(void);

	char floor;
	char direction;

	unsigned int Hash(string event_name);

	string _Id;
	unsigned int _flag;
	unsigned int _hash_signature;
};
CRequest::CRequest(unsigned int Id, unsigned int flag) : _Id(Id), _flag(flag)
{
}
CRequest::~CRequest(void)
{
}
#include "Request.h"
#include "Global.h"
CRequest::CRequest(string Id, unsigned int flag) : _Id(Id), _flag(flag)
{
	_hash_signature = Hash(Id);

	floor = atoi( Id.substr(0, Id.length() -1 ).c_str() );

	// the last character of the input string
	direction = Id.at( Id.length() - 1 );
}
CRequest::~CRequest(void)
{
}


unsigned int CRequest::Hash(string event_name)
```

```
{
	return ::FastHash(event_name);
}
//////////////////////////////////////////////////////////

//requestqueue
#pragma once
#include <deque>
#include "Request.h"

using namespace std;


class CRequestQueue
{
public:
	CRequestQueue(void);
	~CRequestQueue(void);

	void RollToNext(void);
	bool HasRequestInQueue(void);
	unsigned int Mark(string id, unsigned int flag);
	string GetCurrentState();
	char GetCurrentFloor();
	char GetCurrentDirection();
	CRequest GetNextRequest();

	void ChangeDirectionTo(char direction, char floor);
	void ClearCurrentRequest();

private:
	string ConvertInt(int number);
	deque<CRequest> merrygoround;


};
CRequestQueue::CRequestQueue(void)
{
	for (int i = 0; i<FLOORS; i++)
		merrygoround.push_back(
			CRequest(0x10 + 0x10 * i + 0xA, 0)
			); // 0x1A 0x2A 0x3A 0x4A 0x5A 0x6A

	for (int i = FLOORS; i >= 0; i--)
		merrygoround.push_back(
			CRequest( 0x10 + 0x10 * i + 0xD, 0)
			); //     0x6D 0x5D 0x4D 0x3D 0x2D 0x1D

}
CRequestQueue::~CRequestQueue(void)
```

```
{

}
void CRequestQueue::RoundMoveToNext(void)
{

	merrygoround.push_back(	merrygoround.front() );
	merrygoround.pop_front();
}


bool CRequestQueue::NoRequestInQueue(void)
{
	deque<CRequest>::iterator iter;
	for (iter=merrygoround.begin(); iter != merrygoround.end(); iter++)
	{
		if ( iter->_flag != 0)
			return false;

	}

	return true;
}


unsigned int CRequestQueue::Mark(unsigned int id, unsigned int flag)
{

	return 0;
}


#include "RequestQueue.h"
#include "Global.h"
#include <sstream>

using namespace std;

CRequestQueue::CRequestQueue(void)
{
	char temp[64];
	for (int i = 0; i<FLOORS-1; i++)	/*FLOORS-1 no "UP" request on top floor*/
	{
		sprintf_s(temp, "%dU", i+1);
		string s(temp);

		merrygoround.push_back(
			CRequest( s, NO_REQUEST)
			); // "1U" "2U" "3U" "4U" "5U" "6U (X)"
	}
```

```
        for (int i = FLOORS; i > 1; i--)
        {
                sprintf_s(temp, "%dD", i);
                string s(temp);
                merrygoround.push_back(
                        CRequest( s, NO_REQUEST)
                        ); //     "6D" "5D" "4D" "3D" "2D" "1D (X)"    no "DOWN" request on bottom floor
        }
}
CRequestQueue::~CRequestQueue(void)
{

        merrygoround.clear();
}
void CRequestQueue::RollToNext(void)
{
        merrygoround.push_back(    merrygoround.front() );
        merrygoround.pop_front();
}
bool CRequestQueue::HasRequestInQueue(void)
{
        deque<CRequest>::iterator iter;
        for (iter=merrygoround.begin(); iter != merrygoround.end(); iter++)
        {
                if ( iter->_flag != NO_REQUEST)
                        return true;

        }

        return false;
}

string CRequestQueue::ConvertInt(int number)
{
    stringstream ss;//create a stringstream
    ss << number;//add number to the stream
    return ss.str();//return a string with the contents of the stream
}
unsigned int CRequestQueue::Mark(string id, unsigned int flag)
{
        deque<CRequest>::iterator iter;

        // find element by hashed id
        unsigned int _target_id = ::FastHash(id);

        // append U/D to id
        unsigned int arm_sig, leg_sig;
        arm_sig = ::FastHash(id + "U");   /* 1 -> 1U,   4U -> 4UU, 4D -> 4DU ... */
        leg_sig = ::FastHash(id + "D");
```

```
	int r = EVENT_UNRECOGNIZED;
	for (iter=merrygoround.begin(); iter != merrygoround.end(); iter++)
	{
		if ( iter->_hash_signature == arm_sig   || iter->_hash_signature == leg_sig   // find nearest destination
			|| iter->_hash_signature == _target_id ) // search id
		{
			iter->_flag = flag; //
			r = OK;
			break;
		}
	}

	// unrecognized events ignored
	return r;
}

string CRequestQueue::GetCurrentState()
{
	return merrygoround.front()._Id;
}

CRequest CRequestQueue::GetNextRequest()
{
	deque<CRequest>::iterator iter;
	for (iter=merrygoround.begin(); iter != merrygoround.end(); iter++)
	{
		if ( iter->_flag != NO_REQUEST)
			return *iter;

	}

	return CRequest("nUlL", NO_REQUEST);

}

void CRequestQueue::ClearCurrentRequest()
{
	merrygoround.front()._flag = NO_REQUEST;
}

void CRequestQueue::ChangeDirectionTo(char direction, char floor)
{
	while (merrygoround.front().direction != direction || merrygoround.front().floor != floor)
		RollToNext();
}
char CRequestQueue::GetCurrentFloor()
{
	return merrygoround.front().floor;
}
```

```
char CRequestQueue::GetCurrentDirection()
{
	return merrygoround.front().direction;
}

////////////////////////////////////////////////////////////////////////
//stdafx
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>



// TODO: reference additional headers your program requires here



// stdafx.cpp : source file that includes just the standard includes
// elevator.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file



////////////////////////////////////////////////////////////
//targetver.h
#pragma once

// The following macros define the minimum required platform.    The minimum required platform
// is the earliest version of Windows, Internet Explorer etc. that has the necessary features to run
// your application.    The macros work by enabling all features available on platform versions up to and
// including the version specified.

// Modify the following defines if you have to target a platform prior to the ones specified below.
// Refer to MSDN for the latest info on corresponding values for different platforms.
#ifndef _WIN32_WINNT                    // Specifies that the minimum required platform is Windows Vista.
#define _WIN32_WINNT 0x0600        // Change this to the appropriate value to target other versions of Windows.
#endif
```