CS 343 Winter 2021 – Assignment 6
Instructor: Caroline Kierstead
Due Date: Monday, April 12, 2021 at 22:00
Late Date: Wednesday, April 14, 2021 at *23:55*

January 28, 2021

This assignment continues task communication in μC++ and high-level techniques for structuring complex interactions among tasks. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution.

As part of a tax evasion scheme, a billionaire UW alumnus has agreed to make a charitable donation to fund a local light-rail transit system between the various Waterloo campuses: University of Waterloo, Wilfrid Laurier, and Conestoga College. The route forms a loop, and there are two trains in service running continuously. One train runs in a clockwise direction, the other runs in a counter-clockwise direction.

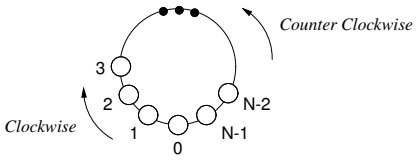

Figure 1: Train Stops on Transit Loop

This assignment simulates a simple train service using the objects and relationships in Figure 2. (Not all possible communication paths are shown in the diagram.)

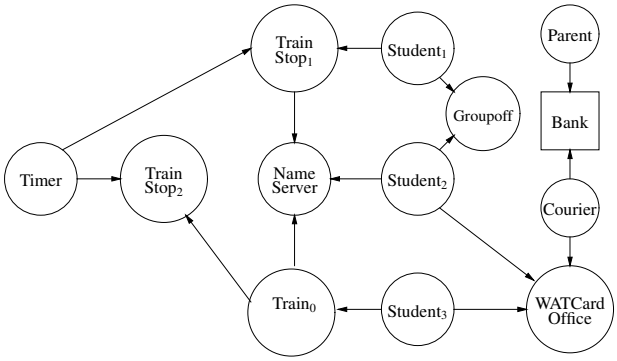

Figure 2: Task Relationships

The following constants are used to configure the simulation, and are read from a text file:

```
StopCost          1   # amount to charge per train stop
NumStudents       2   # number of students to create
NumStops          5   # number of train stops; minimum of 2
MaxNumStudents    5   # maximum students each train can carry
TimerDelay        2   # length of time between each tick of the timer
MaxStudentDelay   10  # maximum random student delay between trips
MaxStudentTrips   3   # maximum number of train trips each student takes
GroupoffDelay     10  # length of time between initializing gift cards
ConductorDelay    5   # length of time between checking on passenger POPs
ParentalDelay     10  # length of time between cash deposits
NumCouriers       1   # number of WATCard office couriers in the pool
```

Comments in the file (from # to the end-of-line), as well as blank lines, are ignored. The constants may appear in any order. Any number of spaces/tabs may appear around a constant name, value or comment. You may assume each constant appears in the configuration file, is syntactically correct, its value is within an appropriate range (i.e., no error checking is required), and only one constant is defined per line. You may have to modify the values in the provided sample file to obtain interesting results when testing.

The following types and routines are required in the assignment (**you may add only a public destructor and private/protected members**):

1. ```
   struct ConfigParms {
       unsigned int stopCost;          // amount to charge per train stop
       unsigned int numStudents;       // number of students to create
       unsigned int numStops;          // number of train stops; minimum of 2
       unsigned int maxNumStudents;    // maximum students each train can carry
       unsigned int timerDelay;        // length of time between each tick of the timer
       unsigned int maxStudentDelay;   // maximum random student delay between trips
       unsigned int maxStudentTrips;   // maximum number of train trips each student takes
       unsigned int groupoffDelay;     // length of time between initializing gift cards
       unsigned int conductorDelay;    // length of time between checking on passenger POPs
       unsigned int parentalDelay;     // length of time between cash deposits
       unsigned int numCouriers;       // number of WATCard office couriers in the pool
   };

   void processConfigFile( const char * configFile, ConfigParms & cparms );
   ```

   Routine processConfigFile is called by the main program to read and parse the configuration file, and places the parsed values into the fields of the argument for output parameter cparms. Note that there must be at least 2 stops, at least 1 student, and at least 1 courier in the system. The stopCost and maxNumStudents must be greater than 0.

2. ```
   _Task Student {
     public:
       Student( Printer & prt, NameServer & nameServer, WATCardOffice & cardOffice, Groupoff & groupoff,
           unsigned int id, unsigned int numStops, unsigned int stopCost, unsigned int maxStudentDelay,
           unsigned int maxStudentTrips );
       };
   ```

   Each student is passed an id in the range [0, NumStudents) for identification. A Student's function is to make a random number of train trips in the range [1, maxStudentTrips], paying for a trip with either their gift card or their WATcard. Before each trip is started, the student delays a random amount of time [0, maxStudentDelay] by calling yield. The first trip is between two randomly-selected distinct stops with id numbers in the range [0, numStops), i.e., the start and end destinations may not be the same stop. Each subsequent trip uses the previous ending stop as the new start stop, and picks a new random distinct ending stop. The student then obtains the location of the starting stop from the name server (see point 3).

   Students use either their gift card or their WATCard, initialized to the maximum cost of a trip, to pay the heavily-subsidized fare, which is based upon the number of stops travelled. (First choice of use is the gift card, then the WATCard. Note that in order to keep things simple, a gift card can only be used once, even if there are still funds left on it i.e. they cannot be transferred to a WATCard or used for another trip. Proof of purchase (POP) is encoded in the WATCard, see point 8, p. 4.) However, occasionally a student is tempted to avoid paying. If the trip is only 1 stop, there is a 50% chance a student attempts to ride without paying; for all other distances, there is a 30% chance.

   To save money, it is in the student's best interest to take the train that travels the fewest number of stops to reach their destination. If the number of stops is the same in either direction, the student picks the *clockwise* direction of travel. If they have insufficient funds, a request is made for the missing amount plus the maximum cost of a trip. If their WATCard is lost, they request a replacement initialized to the maximum cost of a trip. Once the gift card has been used once, it is reset since there's no mechanism to transfer any remaining balance to the student's WATCard.

   A student terminates after completing all of their trips or after being caught by a conductor and ejected from the train.

3.
```
_Task NameServer {
  public:
    NameServer( Printer & prt, unsigned int numStops, unsigned int numStudents );
    ~NameServer();
    void registerStop( unsigned int trainStopId );
    TrainStop * getStop( unsigned int studentId, unsigned int trainStopId );
    TrainStop ** getStopList();  // called by Timer
    TrainStop ** getStopList( unsigned int trainId );          // called by Train
    unsigned int getNumStops();
};
```

   The NameServer is a server task used to manage the train stop names. Each TrainStop must register itself upon creation with the name server by calling registerStop, which stores the address of the calling task. (The simulation cannot start until all of the trains stops have registered themselves.) The timer task and trains obtain the list of stops from the name server by calling getStopList. Students call getStop to know the appropriate train stop in order to buy a ticket and embark/disembark.

4.
```
_Task TrainStop {
  public:
    _Event Funds {
      public:
        unsigned int amount;
        Funds( unsigned int amt );
    };

    TrainStop( Printer & prt, NameServer & nameServer, unsigned int id, unsigned int stopCost );
    _Nomutex unsigned int getId() const;
    void buy( unsigned int numStops, WATCard & card );
    Train * wait( unsigned int studentId, Train::Direction direction );
    void disembark( unsigned int studentId );
    void tick();
    unsigned int arrive( unsigned int trainId, Train::Direction direction, unsigned int maxNumStudents );
};
```

   A TrainStop's function is to act as a synchronization point between the trains and the students. It is given its id and the amount to charge per stop travelled, stopCost, used when a student asks to buy a ticket.

   A student initially calls the buy method, passing in how many stops they are travelling and their Gift/WAT card in payment. The buy method returns the card, marked internally as paid, which is used as a proof-of-purchase to the conductor that the student has paid their fare. If the student has sufficient funds in the gift card or the WATCard, then the amount is debited; otherwise, the exception Funds is raised, which contains the amount the student needs in order to be able to complete the payment after the card balance is debited. (Note that, at any point, use of the card may end up with it being lost (see point 9, p. 5 for details).

   A student calls wait, which blocks the student at the stop until the train travelling in the appropriate direction arrives. Upon returning from the wait call, the student calls the train's embark method.

   The train calls arrive, specifying its direction of travel and the maximum number of students it can take from this stop after taking into account the number it is currently transporting. It then blocks until the next call to tick. Within arrive, the TrainStop unblocks the appropriate number of waiting students and these students call Train::embark (see point 5) to get on the train. Note that the students remain blocked on the train's entry queue until the train accepts their calls, which will only happen after a call to tick has been made.

   A student calls disembark to indicate that it is getting off the train at this stop.

   The TrainStop then blocks until tick is called.

   The Timer calls tick to advance the system clock, waking any trains blocked at this stop.

5. **_Task** Train {
   public:
   Train(Printer & prt, NameServer & nameServer, **unsigned int** id, **unsigned int** maxNumStudents,
   unsigned int numStops);
   ~Train();
   _Nomutex unsigned int getId() **const**;
   TrainStop * embark(**unsigned int** studentId, **unsigned int** destStop, WATCard& card);
   void scanPassengers();
   };

   A Train's function is to first determine its direction of travel, where train 0 travels in a clockwise direction starting at stop 0, while train 1 travels in a counter-clockwise direction starting at stop $\lceil numStops/2 \rceil$. Travelling in a *clockwise* direction means that it travels from stop 0 to stop 1, from stop 1 to stop 2, etc, wrapping back around from stop numStops−1 back to stop 0. Travelling in a *counter-clockwise* direction reverses the order of stops. Since there is a train travelling in each direction, the maximum cost of any trip is stopCost $\times \lfloor numStops/2 \rfloor$. The train then obtains a list of stops from the NameServer. It visits each stop in turn at every "tick" of the Timer task.

   At every stop, it takes on waiting passengers through embark until it reaches its limit of maxNumStudents, and signals any blocked students who have reached their destination stop so that they can leave and disembark at the station. A student that unblocks at a train stop calls embark and blocks again "within" the train until it reaches the appropriate train stop, at which point they must call the disembark method for the new TrainStop. Note that even if a student is cheating, it still needs to present a WATCard to embark so that the conductor can check for proof of purchase, which requires the future WATCard to be accessed, and thus having the potential of being lost.

   Each train has a Conductor, responsible for checking periodically that students on the train have paid their fare.

6. **_Task** Conductor {
   public:
   Conductor(Printer & prt, **unsigned int** id, Train * train, **unsigned int** delay);
   };

   There is one conductor per train, with the same id number as the train it patrols. Its main loop consists of yielding the CPU conductorDelay times and then calling the train's scanPassengers method, which lets it check all passengers on board to verify that they have a ticket. Any passengers without a POP are summarily ejected from the train at the next stop.

7. **_Task** Timer {
   public:
   Timer(Printer & prt, NameServer & nameServer, **int** timerDelay);
   };

   A Timer's function is to keep the simulation clock ticking by advancing the simulation in measured increments. It calls tick on each train stop after having delayed timerDelay times by calling yield.

8. **class** WATCard {
   friend class TrainStop;
   void markPaid();

   WATCard(**const** WATCard &) = **delete**; *// prevent copying*
   WATCard & **operator**=(**const** WATCard &) = **delete**;
   public:
   typedef Future_ISM<WATCard *> FWATCard; *// future watcard pointer*
   WATCard();
   void deposit(**unsigned int** amount);
   void withdraw(**unsigned int** amount);
   unsigned int getBalance();
   bool paidForTicket();
   void resetPOP();
   };

The WATCard manages the money associated with a card and contains the "proof of purchase" (POP) for a fare. Its balance is set to 0 at creation.

The WATCard office calls deposit after a funds transfer.

A train stop calls withdraw and then markPaid when a fare is purchased. The markPaid method is declared **private** to keep the student from calling it, which is why the TrainStop is declared a **friend**.

A student and a train stop call getBalance to determine the balance.

A conductor calls paidForTicket to see the POP.

9.
```
_Task WATCardOffice {
    struct Job {                                  // marshalled arguments and return future
        Args args;                                // call arguments (YOU DEFINE "Args")
        WATCard::FWATCard result;                 // return future
        Job( Args args ) : args( args ) {}
    };
    _Task Courier { ... };                        // communicates with bank

    void main();
  public:
    _Event Lost {};                               // lost WATCard
    WATCardOffice( Printer & prt, Bank & bank, int numCouriers );
    WATCard::FWATCard create( int sid, int amount ); // called by student
    WATCard::FWATCard transfer( int sid, int amount, WATCard * card ); // called by student
    Job * requestWork();                          // called by courier to request/return work
};
```

The WATCardOffice is an administrator task used by a student to transfer funds from their bank account to their WATCard to buy a train ticket. Initially, the WATCard office creates a fixed-sized courier pool of numCouriers courier tasks to communicate with the bank. (Additional couriers may not be created after the WATCardOffice begins.)

A student performs an asynchronous call to create to create a "real" WATCard with an initial balance. A future WATCard is returned and sufficient funds are subsequently obtained from the bank (see Parent task) via a courier to satisfy the create request.

A student performs an asynchronous call to transfer when its WATCard indicates there is insufficient funds to buy a ticket. A future WATCard is returned and sufficient funds are subsequently obtained from the bank (see Parent task) via a courier to satisfy the transfer request. The WATCard office is empowered to transfer funds from a student's bank-account to its WATCard by sending a request through a courier to the bank.

Each courier task calls requestWork, blocks until a Job request is ready, and then receives the next Job request as the result of the call. As soon as the request is satisfied (i.e., money is obtained from the bank), the courier updates the student's WATCard. There is a 1 in 6 chance a courier loses a student's WATCard after the update. When the card is lost, the exception WATCardOffice::Lost is inserted into the future, rather than making the future available, and the current WATCard is deleted.

10.
```
_Monitor Bank {
  public:
    Bank( int numStudents );
    void deposit( unsigned int id, int amount ); // deposit "amount" $ for student "id";

     // withdraw "amount" $ from student "id"; block until student has enough funds
    void withdraw( unsigned int id, int amount );
};
```

The Bank is a monitor, which behaves like a server, that manages student-account information for all students. Each student's account initially starts with a balance of $0.

The parent calls deposit to endow gifts to a specific student.

A courier calls withdraw to transfer money on behalf of the WATCard office for a specific student. The courier *waits until enough money has been deposited*, which may require multiple deposits.

11. **_Task** Parent {
    public:
    Parent(Printer & prt, Bank & bank, **unsigned int** numStudents, **unsigned int** parentalDelay,
    unsigned int maxTripCost);
    };

    The Parent task periodically gives a random amount of money (one-third, two-thirds, or the full maximum cost of a trip) i.e. std::max(1, stopCost $\times \lfloor \frac{numStops}{2} \rfloor \times \frac{[1,3]}{3}$) to a random student. Before each gift is transferred, the parent yields for parentalDelay times (not random). The parent must check for a call to its destructor to know when to terminate. Since it must not block on this call, it is necessary to use a terminating **_Else** on the accept statement. (Hence, the parent is busy waiting for the call to its destructor.)

12. **_Task** Groupoff {
    public:
    Groupoff(Printer & prt, **unsigned int** numStudents, **unsigned int** maxTripCost, **unsigned int** groupoffDelay);
    ~Groupoff();
    WATCard::FWATCard giftCard();
    };

    The Groupoff task begins by accepting a call from each student to obtain a future gift-card. Then groupoff periodically puts a real WATCard with value maxTripCost into a random future gift-card. A future gift-card is assigned only once per student.

    Before each future gift-card is assigned a real WATCard, groupoff yields for groupoffDelay times (not random). The groupoff loops until all the future gift-cards are assigned a real WATCard or a call to its destructor occurs Since it must not block on the destructor call, it is necessary to use a terminating **_Else** on the accept statement. (Note, this use of **_Else** is not busy waiting because there are a finite number of students.)

13. **_Monitor** / **_Cormonitor** Printer {
    public:
    Printer(**unsigned int** numStudents, **unsigned int** numTrains, **unsigned int** numStops, **unsigned int** numCouriers);
    ~Printer();
    void print(Kind kind, **char** state);
    void print(Kind kind, **char** state, **unsigned int** value1);
    void print(Kind kind, **char** state, **unsigned int** value1, **unsigned int** value2);
    void print(Kind kind, **unsigned int** lid, **char** state);
    void print(Kind kind, **unsigned int** lid, **char** state, **unsigned int** value1);
    void print(Kind kind, **unsigned int** lid, **char** state, **unsigned int** value1, **unsigned int** value2);
    void print(Kind kind, **unsigned int** lid, **char** state, **unsigned int** oid, **unsigned int** value1, **unsigned int** value2);
    void print(Kind kind, **unsigned int** lid, **char** state, **char** c);
    void print(Kind kind, **unsigned int** lid, **char** state, **unsigned int** value1, **char** c);
    void print(Kind kind, **unsigned int** lid, **char** state, **unsigned int** value1, **unsigned int** value2, **char** c);
    };

    All output from the program is generated by calls to a printer, excluding error messages. The printer generates output like that in Figure 3. Each column is assigned to a particular kind of object. There are 9 kinds of objects: parent, groupoff, WATCard office, WATCard office courier, name server, train, train stop, student, and the timer. Student, train, conductor, train stop, and WATCard office courier have multiple instances. For the objects with multiple instances, these objects pass in their local identifier [0,N) when printing. Each kind of object prints specific information in its column:

    - The parent prints the following information:

State	Meaning	Additional Information
S	starting	
D*s*,*g*	deposit gift	student *s* receiving gift, amount of gift *g*
F	finished	

    - The groupoff prints the following information:

State	Meaning	Additional Information
S	starting	
D*g*	deposit gift	amount of gift *g*
F	finished	

```
$ lrt lrt.config 31708
Parent  Gropoff WATOff  Names   Timer   Train0  Train1  Cond0   Cond1   Stop0   Stop1   Stud0   Stud1   WCour0
*******  *******  *******  *******  *******  *******  *******  *******  *******  *******  *******  *******  *******
S       S       S       S       S       S0,<    S1,>    S       S       S       S       S2              S
                C0,1    R0                                                                      t0,1
D0,2            W       R1                                                              S3
                C1,1    L                       c       c                               T0,1
                W       L               A0,5,0  A1,5,0          A0,5,0          T0,1,<          t1,1
D0,3                    L                                               A1,5,0
                        T0,0                                                    B1,0
                                                                                W0      T0,1,<
D0,2    D1              T1,0    t0                              W0,<                    f
D1,3                            t1      E0,1    A0,5,0                  t       t       E0      W0      T1,1
D0,3                                    A1,4,1                  c       A1,5,0
        D1                                                      W1,<    A0,4,0
        F                       t2                      c               t       t       D1
                                                                                T1,0,<
                                                                                f
D0,3                    T0,1            A0,5,0  A1,5,0                  t       D0      W1
                                                                c       A0,5,1  W0,<
D0,2                                                    c                       A1,5,0
D0,1                            t3              A0,5,0                  t               E0
D1,1                            t4      E1,1            e1      c       A1,5,0  t
                                t5      A1,4,1                          t       t
D0,1                                                    A1,5,0  c       t       A0,4,1
D0,2                            t6      E0,0                    c       t       t       E0
                                                                                A1,5,0          e
D1,2                            t7      A0,4,1  A0,5,0  e0              t       t               F
                                t8                      c               A0,4,0  t
D1,3                                                            c       A1,5,0
                                t9                                      t       t       e
D0,1                            t10     A1,5,0  A1,5,0  F               t       t       F
D0,2                                                            c       t       A1,5,0
                                                                                A0,5,0
D0,3                            t11     F       A0,5,0                  t       t
D1,1                            t12                                     A1,5,0  t
                                t13             A1,5,0          c       t       t
D1,1                                                                    t       A1,5,0
                                t14             A0,5,0                  t       t
D1,3                            t15                             F       A1,5,0  t
D1,1                            t16             F                       t       t
                                t17                                     t       t
D1,2                            F                                       t       t
D1,2            F       F                                               F       F               F
F
**********************
```

Figure 3: WATLRT : Example Output

- The WATCard office prints the following information:

State	Meaning	Additional Information
S	starting	
W	request work call complete	
C*s*,*a*	create call complete	student *s*, transfer amount *a*
T*s*,*a*	transfer call complete	student *s*, transfer amount *a*
F	finished	

- The name server prints the following information:

State	Meaning	Additional Information
S	starting	
R*s*	register train stop	train stop *s* registering
L	get list of stops	timer requesting list of all train stops
L*t*	get list of stops	train *t* requesting list of all train stops
T*s*,*t*	identity of train stop	student *s* requesting identity of train stop *t*
F	finished	

- A train prints the following information:

State	Meaning	Additional Information
S*s*,*d*	starting	train starting at stop *s* traveling in direction *d*, where '<' is clockwise and '>' is counter-clockwise
A*s*,*n*,*m*	arrives	arrives at station *s* and can take up to *n* students since already carrying *m* students
E*s*,*d*	embarks	student *s* embarks for train stop *d*
F	finished	

- A train stop prints the following information:

State	Meaning	Additional Information
S	starting	
A*id*,*n*,*m*	arrives	train *id* arrived that can take up to *n* students and has *m* waiting for a train traveling in that direction
B*c*	bought train ticket	student purchased ticket that cost *c* dollars
W*s*,*d*	waiting at train stop	student *s* waiting for train traveling in direction *d*, where '<' is clockwise and '>' is counter-clockwise
D*s*	disembarked	student *s* got off train at this train stop
t	tick	a tick has occurred
F	finished	

- A conductor prints the following information:

State	Meaning	Additional Information
S	starting	
c	checking passengers	
e*n*	eject student	student *n* was found without proof of having purchased a ticket
F	finished	

- A student prints the following information:

State	Meaning	Additional Information
S*n*	starting	starting and will make *n* trips
T*a*,*b*,*d*	trip	trip starting at stop *a* and ending at stop *b*; *d* is the direction, where '<' is clockwise and '>' is counter-clockwise
G*c*,*b*	gift-card train ticket	purchased ticket that cost *c* dollars through the gift card, gift card balance *b*
B*c*,*b*	WATCard train ticket	purchased ticket that cost *c* dollars, WATCard balance *b*
f	not paying for the ride	going to try and ride for free
e	ejected	ejected from the train and going home
W*s*	waiting at train stop	waiting at train stop *s*
E*t*	embarked	embarked onto train *t*
D*s*	disembarked	disembarked at train stop *s*
L	WATCard lost	
F	finished	

- A WATCard office courier prints the following information:

State	Meaning	Additional Information
S	starting	
t*s*,*a*	start funds transfer	student *s* requesting transfer of amount *a* o
L*s*	lost WATCard card	student *s* requesting transfer
T*s*,*a*	complete funds transfer	student *s* requesting transfer of amount *a*
F	finished	

- A timer prints the following information:

State	Meaning	Additional Information
S	starting	
t*t*	next tick of the clock	tick number *t* of the clock
F	finished	

Information is buffered until a column is overwritten for a particular entry, which causes the buffered data to be flushed. If there is no new stored information for a column since the last buffer flush, an empty column is printed.

After an object has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in internal representation; **do not build and store strings of text for output**.

The program main starts by calling processConfigFile to read and parse the simulation configurations. It then creates in order the printer, bank, WATCard office, groupoff, parent, name server, timer, train stops, trains, and students. The entire simulation must shut down in an orderly fashion once all of the students have completed their trips.

The executable program is named lrt and has the following shell interface:

```
lrt [ config-file [ Seed ] ]
```

config-file is the text (formatted) file containing the configuration constants. If unspecified, use the file name lrt.config. Seed is the seed for the random-number generator and must be greater than 0. If the seed is unspecified, use a random value like the process identifier (getpid) or current time (time), so each run of the program generates different output. Use the monitor MPRNG to safely generate random values. Note, because of the non-deterministic execution of concurrent programs, multiple runs with a common seed may not generate the same output. Nevertheless, shorts runs are often the same so the seed can be useful for testing.

Print an appropriate usage message and terminate the program if there are missing/invalid number arguments, invalid configuration parameters, or you are unable to open the given input file.

Do not try to code this program all at once. Write each task separately and test it before putting the pieces together. Play with the configuration file and sample executable to familiarize yourself with the system before starting to write code.

Submission Guidelines

This assignment should be done by a team of two people because of its size. Both people receive the same grade (no exceptions). **Only one member of a team submits the assignment.** The instructor and/or instructional-coordinator does not arbitrate team disputes; team members must handle any and all problems.

Follow these guidelines carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text or test-document file, e.g., *.{txt,doc} file, must be ASCII text and not exceed 750 lines in length, using the command fold -w120 lrt.doc | wc -l.** Programs should be divided into separate compilation units, i.e., *.{h,cc,C,cpp} files, where applicable. Use the submit command to electronically copy the following files to the course account.

1. *.{h,cc,C,cpp} – code for the assignment. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**

2. lrt.testdoc – test documentation for the assignment, which includes the input and output of your tests. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

3. group.txt – give the userid of each group member, with one userid per line (i.e., 2 lines in this file), e.g.:

   ```
   sjholmes
   jjwatson
   ```

   Do not submit this file if you are working by yourself.

4. Makefile – construct a makefile with target lrt, which creates the executable file lrt from source code in the makefile directory when the command make lrt is issued. This makefile must NOT contain hand-coded dependencies. This Makefile must be submitted with the assignment to build the program, so it must be correct. Use the web tool Request Test Compilation to ensure you have submitted the appropriate files, your makefile is correct, and your code compiles in the testing environment.

Follow these guidelines. Your grade depends on it!