Midterm Answers – CS 343 Winter 2025

Instructor: Peter Buhr

March 5, 2025

These are not the only answers that are acceptable, but these answers come from the notes, assignments, or lectures.

1. (a) **1 mark** cin is overloaded to return a **bool** and an istream, so the boolean is called in the conditional context.

   (b) **5 marks**

```
1   if ( ! C ) goto ELSE;
1       S1;
1       goto ENDIF;
1   ELSE: ;
-       S2;
1   ENDIF: ;
```

   (c) **1 mark** Outdenting exits makes all loop exit-points visible by glancing down the loop body.

   (d) **2 marks**

```
        for ( ;; ) {
            S1
1         if ( C1 ) { E1; break; }
            S2
1         if ( C2 ) { E2; break; }
            S3
        }
```

   (e) **1 mark** A flag variable is used solely to affect control flow, i.e., it does not contain data associated with the computation.

   (f) **1 mark** When storage must outlive the block in which it is allocated (ownership change).

   (g) **2 marks** normal outcome: return normal result and transfer after call
   exceptional outcome: return alternative result and not transfer after call

   (h) **2 marks** A routine passed to another routine that is called for an exceptional event to fix-up and return a corrective result so a computation can continue.

   (i) **2 marks** Recursion allows multiple stack frames for the same routine, so a transfer point within the routine is not unique.

   (j) **2 marks** Stack unwinding walks down (up) the stack, removing stack frames, until a specific target frame is found, and control continues within that frame.

   (k) **2 marks** After the **catch** clause (static return) or after the raise point (dynamic return).

2. (a) **2 marks** **this** changes when execution starts in the interface member.
uThisCoroutine changes when the resume statement is executed.

   (b) **1 mark** The coroutine that started (did the first resume) of the coroutine.

   (c) **1 mark** The suspends cause the full coroutines to *reverse* their cycle.

   (d) **2 marks** There is no stack growth because the coroutines context switch rather than call.

   (e) **2 marks** A generator/iterator cannot call helper routines or do full coroutining, while a coroutine can.

   (f) **2 marks** The **_Enable** statement controls when nonlocal exception can be delivered, so a coroutine can prepare (add **try** block) for handling their delivery.

3. (a) **1 mark** A C++ destructor cannot raise an exception, if an exception is being propagated (stack is being unwound).

   (b) **3 marks** An increment statement is composed of 3 instructions, with interrupts (context switching) among them.

```
ld r1,i             // load into register 1 the value of i
// PREEMPTION
add r1,#1           // add 1 to register 1
// PREEMPTION
st r1,i             // store register 1 into i
```

   (c) **2 marks** In implicit concurrency, the programmer defines the concurrent units of work but not the threads.
   In explicit concurrency, the programmer defines both concurrent units of work and the threads.

   (d) **2 marks** Program speedup is $S_C = T_1/T_C$, where C is number of CPUs and T_1 is sequential execution.

   (e) **1 mark** The longest execution path among a group of threads bounds speedup.

   (f) **1 mark** No, actors do not have a thread.

   (g) **1 mark** In selecting a thread for entry to a critical section, the selection cannot be postponed indefinitely.

   (h) **5 marks**

```
1   int Lock = OPEN; // shared
    void Task::main() { // each task does
1       int dummy = CLOSED;
        do {
1           Swap( Lock, dummy );
1       while( dummy == CLOSED );
        /* critical section */
1       Lock = OPEN;
    }
```

   (i) **1 mark** Blocking locks reduce busy waiting by having the releasing task do additional work (cooperation).

   (j) **2 marks** Barging avoidance allows barging threads to enter a lock but immediately blocks them, ensuring an unblocked (signalled) thread enters the critical section next.

4. **14 marks**

```
1   try {
1       _Enable {
          Start:
1           for ( ;; ) {
1               while ( ch != word[0] ) { next->put( ch ); suspend(); }
1               for ( unsigned int i = 1; i < len; i += 1 ) {
2                   next->put( ch ); suspend();
1                   if ( ch != word[i] ) continue Start;
                } // for
1               cnt += 1;
1               next->put( ch ); suspend();
            } // for
        } // _Enable
1   } catch( Eof ) {
1       cout << cnt << ' ' << word << endl;
1       _Resume Eof{} _At *next;
1       next->put( ch );                        // activate coroutine to propagate execption
    } // try
```

-4 if not using coroutine state.

5. (a) **5 marks**

```
1   min = max = row[0];
1   for ( unsigned int r = 1; r < cols; r += 1 ) {          // find min, max of row
1       if( row[r] < min ) min = row[r];
1       if( row[r] > max ) max = row[r];
    } // for
1   if ( min == max ) _Resume NonUnique() _At prgMain;
```

(b) **9 marks**

```
1   try {
1       _Enable {
1           COFOR( r, 0, rows,                          // thread per row
1                   minmax( M[r], cols, rmin[r], rmax[r], prgMain );
            ); // COFOR
        } // _Enable
1       for ( r = 0; r < rows; r += 1 ) {               // find overall min and max
1           if( rmin[r] < min ) min = rmin[r];
1           if( rmax[r] > max ) max = rmax[r];
        } // for
1   } _CatchResume( NonUnique ) {
1       nonunique += 1;
    } // try
```

(c) **8 marks**

```
struct WorkMsg : public uActor::Message {
1   const int * row, cols;
1   int & min, & max;
1   uBaseTask & prgMain;
1   WorkMsg( const int row[], const unsigned int cols, int & min, int & max, uBaseTask & prgMain ) :
        Message( uActor::Delete ), row( row ), cols( cols ), min( min ), max( max ), prgMain( prgMain ) {}
};
_Actor MinMax {
1   Allocation receive( Message & msg ) {
1       Case( WorkMsg, msg ) {                          // discriminate derived message
            WorkMsg & w = *msg_d;                       // eye candy
1           minmax( w.row, w.cols, w.min, w.max, w.prgMain );
        } else assert( false );                         // bad message
1       return Finished;                                // one-shot
    } // MinMax::receive
}; // MinMax
```

(d) **8 marks**

```
1   try {
-       _Enable {
1           uActor::start();                            // start actor system
1           MinMax minmax[rows];
1           for ( unsigned int r = 0; r < rows; r += 1 ) {
1               minmax[r] | *new WorkMsg( M[r], rows, rmin[r], rmax[r], prgMain );
            } // for
1           uActor::stop();                             // wait for all actors to terminate
        } // _Enable
1       for ( r = 0; r < rows; r += 1 ) {               // find overall min and max
-           if( rmin[r] < min ) min = rmin[r];
-           if( rmax[r] > max ) max = rmax[r];
        } // for
1   } _CatchResume( NonUnique ) {
-       nonunique += 1;
    } // try
```

(e) **8 marks**

```
_Task MinMax {                                          // check row of matrix
  public:
    _Exception Stop {};                                 // concurrent exception
  private:
1   const int * row, cols;
1   int & min, & max;
1   uBaseTask & prgMain;
    void main() {
1       try {
1           _Enable {
1               minmax( row, cols, min, max, prgMain );
            } // _Enable
1       } catch( Stop & ) {}
    } // MinMax::main
  public:
1   MinMax( const int row[], const int cols, int & min, int & max, uBaseTask & prgMain ) :
            row( row ), cols( cols ), min( min ), max( max ), prgMain( prgMain ) {}
};
```

(f) **14 marks**

```
1   MinMax * workers[rows];
1   for ( r = 0; r < rows; r += 1 ) {                    // create tasks to process rows
1       workers[r] = new MinMax( M[r], r, cols, rmin[r], rmax[r], uThisTask() );
    } // for

1   bool nonunique = false;
1   try {
1       r = 0;                                           // initialize before Enable
1       _Enable {
1           for ( ; r < rows; r += 1 ) {                 // find overall min and max
1               delete workers[r];                       // wait for completion and delete tasks
            } // for
        } // _Enable
1   } _CatchResume( NonUnique ) {
1       if ( ! nonunique ) {
1           for ( int i = r + 1; i < rows; i += 1 ) {
1               _Resume MinMax::Stop() _At *workers[i];
            } // for
1           nonunique = true;
        } // if
    } // try
```