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

1. (a) 6 marks
   i.  
   1 if ( ! C ) goto L1;  
   … // then-clause  
   1 goto L2;  
   1 L1:  
   … // else-clause  
   - L2:  
   ii.  
   1 L3: if ( ! C ) goto L4;  
   … // loop-body  
   1 goto L3;  
   1 L4:  

(b) 2 marks
   i. multi-exit loop has multiple exit-points from within a single loop, while static multi-level exit has multiple exit-points from within any nested control-structures.
   ii. static multi-level exit-points are known at compile time, while dynamic multi-level exit-points are only known at runtime.

(c) 1 mark False

(d) 2 marks Example from notes, but other examples are possible.

   1 if ( printf( . . . ) < 0 ) {  // check return code for error  
   1 perror( "printf:"");  // errno describes specific error

(e) 2 marks Sequel call/return is static/static, while routine-pointer call/return is dynamic/dynamic.

(f) 2 marks A destructor may not raise an exception during propagation, because it is unclear which exception should be propagated.

(g) 2 marks C++ zero-cost exceptions means entering and leaving a try block has no cost. The term is misleading because there is a large cost in storage and time to raise an exception.

2. (a) 1 mark A communication variable transfers information between member routines via class-global variables.

(b) 1 mark The “Zen” of the coroutine means letting the coroutine do as much work as possible. (Not duplicating the work/state the coroutine does/knows about.)

(c) 2 marks The stacks for a µC++ coroutine is fixed size. Allocate storage in the heap or increase the stack size at creation.

(d) 2 marks Interface members and communication variables provide different parameter and return types, which is impossible using a single interface for the coroutine main.

(e) 1 mark False

(f) 2 marks 1 - call to mem, context switch to self, return from call  
   1 mark 2 - context switch to self  
   1 mark 3 - suspend back to last resumer  
   1 mark 4 - resume starter

(g) 1 mark To control when pending non-local exceptions may be raised.
3. (a) **2 marks** Preemption allows rescheduling between any two instruction using interrupts.
   Non-preemptive allows rescheduling at fixed places, like blocking and crossing the kernel boundary.

(b) **3 marks** \( i += 1 \) is implemented as a load, add, store with a register, which can be interrupted between any of these instructions, resulting in overwriting prior computations.

(c) **2 marks** Multiple user threads scheduled across multiple kernel threads.

(d) **1 mark** COBEGIN/COEND and COFOR may create less threads than requested by the programmer.

(e) **1 mark** termination synchronization

(f) **3 marks** The task entering the critical section copies its id into the shared variable. If mutual exclusion is correct, the id should not change for the duration of the critical section. If it changes, the task in the critical section detects the changes and knows the mutual exclusion is violated.

(g) **2 marks** Unbounded overtaking allows entry into the critical section until the other task that retracted its intent redeclares it.
   Bounded overtaking disallows entry into the critical section because the other task never retracted its intent.

(h) **2 marks**
   i. The race assigning to Last breaks ties on simultaneous arrival.
   ii. Racing alone, while the other task waits, loses the race, forcing progress of the waiting task.

4. (a) **2 marks** no yield is a direct (hard) busy wait checking for an event to occur, yield relinquish the task’s time-slice after every N checks.

(b) **2 marks** yieldNoSchedule does not put the executing task on the ready queue.
   The lock argument is atomically released after the executing task blocks.

(c) **2 marks** Avoidance allows barging tasks but prevents them from running ahead of waiting tasks, while prevention precludes barging tasks altogether.

(d) **3 marks**

```c
1 Barrier start(N+1), end(N+1);

  coordinator           workers
1 start.block();       start.block();
1 end.block();         end.block();
```

(e) **6 marks**

<table>
<thead>
<tr>
<th>synchronization</th>
<th>mutual exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>task1</strong></td>
<td><strong>task2</strong></td>
</tr>
<tr>
<td>BinSem lock( 0 );</td>
<td>lock.P();</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>lock.V();</td>
<td>lock.V();</td>
</tr>
<tr>
<td></td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td>1 lock.V();</td>
</tr>
</tbody>
</table>
5. 17 marks

```c
void main() {
    try {
        _Enable {
            for ( ;; ) {
                // for all characters in file
                for ( ;; ) { // for all characters in line
                    ch = get();
                    while ( isblank( ch ) || ch == ' \n' ) suspend(); // skip starting blanks or line
                    line:
                        for ( ;; ) { // for all characters in line
                            for ( ;; ) { // for all characters in non-blank sequence
                                cout << ch;
                                if ( ch == ' \n' ) break line; // end of line, start new line ?
                                suspend();
                                if ( isblank( ch ) ) break; // end of non-blank sequence ?
                            } // for
                        } // for
                        if ( ch != ' \n' ) cout << ' ';
                        // single blank between words
                    } // for
                    suspend(); // first character of next line
                } // for
            } // _Enable
        } catch ( Eof ) {}
    }
}
```

Maximum 8 if not using coroutine state.
```cpp
#include <iostream>

using namespace std;

_Event NotDS {};
_Task DiagSymmetric {}

const int (*M)[10], row, cols;

uBaseTask & prgMain;

void main() {
  try {
    _Enable {
      for ( int r = row + 1; r < cols; r += 1 ) {
        if ( M[row][r] != M[r][row] ) { // check row against column
          _Resume NotDS( row ) _At prgMain;
          return;
        } // exit
      } // for
      if ( M[row][row] != M[cols - 1][cols - 1] ) { // check diagonal
        _Resume NotDS( row ) _At prgMain;
      } // if
    } // _Enable
  }
}

public:

DiagSymmetric( const int M[][10], const int cols, uBaseTask & prgMain ) :
  M( M ), row( row ), cols( cols ), prgMain( prgMain )
};

int main() {
  int rows;

  cin >> rows;

  for ( r = 0; r < rows; r += 1 ) {
    for ( c = 0; c < rows; c += 1 ) {
      cin >> M[r][c];
      cout << M[r][c] << ", ";
    }
    cout << endl;
  }

  DiagSymmetric *workers[rows];

  for ( r = 0; r < rows - 1; r += 1 ) {
    workers[r] = new DiagSymmetric( M, r, rows, uThisTask() );
  }

  bool notDS = false;

  try {
    r = 0;
    _Enable {
      for ( ; r < rows - 1; r += 1 ) { // wait for completion and delete tasks
        delete workers[r];
      } // for
    } // _Enable
  }
}

_CatchResume( NotDS ) {
  if ( ! notDS ) { // if not diagonal symmetric
    for ( int i = r + 1; i < rows - 1; i += 1 ) { // try to stop other tasks
      _Resume DiagSymmetric::Stop() _At *workers[i];
    } // for
    notDS = true; // do not do this again
  } // if
}

} catch( Stop ) {} 

// main
```