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

1. (a) i. **3 marks**

<table>
<thead>
<tr>
<th>GOOD</th>
<th>GOOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (; ;;) {</td>
<td>for (; ;;) {</td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>1 if (! C1) break;</td>
<td>1 if ( C1) break;</td>
</tr>
<tr>
<td>1 S2</td>
<td>- S2</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>} }</td>
<td></td>
</tr>
</tbody>
</table>

ii. **1 mark** The BAD form associates S2 with the if statement rather than the loop body.

(b) **2 marks** The labelled break statement is easier to read (better eye-candy) than a goto statement because the labels are at the start rather than the end of the control structures.

(c) **1 mark** A flag variable is used solely to affect control flow OR does not contain data associated with a computation.

(d) i. **2 marks** Any two of: may not be tested, expands return values, poor performance

ii. **1 mark** slower

(e) **2 marks** The finally clause is executed for normal or exceptional return of the try block.

(f) **2 marks** The source execution delivers an exception to a faulting execution, and the faulting execution propagates it.

(g) i. **1 mark** next statement

ii. **1 mark** after the raise (Resume)

(h) **1 mark** Non-local __Throw is unsupported because the faulting execution is forced to unwind its stack, resulting in poor software-engineering control.

(i) **1 mark** A variable’s storage must outlive the block in which it is allocated.

(j) **1 mark** Concurrent use of the heap causes high contention on the serial heap-resource causing performance slowdown.

2. (a) **2 marks** Coroutines share a thread deterministically versus tasks with their own threads running non-deterministically.

(b) **1 mark** stack

(c) **1 mark** active (calling) coroutine’s stack

(d) **2 marks** When a non-terminated coroutine is deallocated, its stack is unwound and any destructors executed, otherwise cleanup actions are not executed leaving the environment unsound.

(e) **1 mark** Linearize means to convert multiple loops into a single loop with flag variable and if statements.

(f) **1 mark** µC++ verify check for stack overflow.

(g) **1 mark** The suspend goes back to the last resume, which reverses the cycle.

(h) **2 marks** Python coroutines are stackless and µC++ coroutines are stackful. Python coroutines cannot be modularized OR no full coroutines.
3. (a) 1 mark A concurrent bottleneck is an execution location that restricts or serializes concurrency.
(b) 1 mark Keep sequential code as small as possible.
(c) 2 marks A critical path is the longest execution path among a set of concurrent tasks, which bounds speedup.
(d) 2 marks No, COBEGIN/COEND can only create a tree (lattice) process-graph, while START/WAIT can create a network (arbitrary) graph.
(e) 1 mark Task static variables are shared, and hence require mutual exclusion for safe read/write access.
(f) 2 marks Liveness (rule 4) means tasks do not execute forever outside the critical section to determine entry, while eventual progress means all tasks waiting entry to the critical section enter it.
(g) 2 marks Liveness (rule 4) is violated because both tasks may see the other task wants-in simultaneously and both wait forever for the other task to retract their intent.
(h) 2 marks

```
1 while( TestSet( Lock ) == CLOSED );
  // critical section
1  Lock = OPEN;
```

4. (a) 2 marks An independent critical section does not share variables (objects) with other critical sections, whereas a dependent critical section does share.
(b) 1 mark One lock per independent critical-section.
(c) 1 mark 1 check
(d) 2 marks Avoidance allows barging tasks but prevents them from running ahead of waiting tasks, while prevention precludes barging tasks altogether.
(e) 2 marks A synchronization wait provides a service to block and unlock a mutex-lock atomically to prevent a race condition.
(f) 2 marks A mutex lock starts open so synchronization fails to block if the event has not occurred. A synchronization lock starts closed (always block) so no task can enter the critical section.
void main() {
    char X, Y;
    int open, pair;

    for ( open = 0;; open += 1 ) {
        if ( ch != '(' ) break;
        suspend();
    } // for
    if ( open == 0 ) { _Resume Error() _At resumer(); return; }

    X = ch;
    if ( X == ')' || X == '(' ) { _Resume Error() _At resumer(); return; }
    suspend();
    if ( Y == ')' || Y == '(' ) { _Resume Error() _At resumer(); return; }
    Y = ch;
    for ( pair = 1; pair < open; pair += 1 ) {
        suspend();
        if ( ch != X ) { _Resume Error() _At resumer(); return; }
        suspend();
        if ( ch != Y ) { _Resume Error() _At resumer(); return; }
    } // for
    for ( ; open > 0; open -= 1 ) {
        suspend();
        if ( ch != ')' ) { _Resume Error() _At resumer(); return; }
    } // for
    _Resume Match() _At resumer(); return;
} // Grammar::main

Maximum 10 if not using coroutine state.
```cpp
#include <iostream>

using namespace std;

_Event Schmilblick {};
_Task Schmilblicks {
    const int *row, cols, schmilblick;
    uBaseTask &prgMain;

    void main() {
        int cnt = 0;
        try {
            _Enable {
                for (int c = 0; c < cols; c += 1) {
                    if (row[c] == schmilblick) {
                        cnt += 1;
                        if (cnt == 2) {
                            _Resume Schmilblick() _At prgMain;
                            break;
                        }
                    }
                }
            } // if
            } // for
        } // _Enable
        catch( Stop ) {}
    } // Schmilblicks::main

public:
    Schmilblicks( const int row[], const int cols, uBaseTask &prgMain, int schmilblick ) :
        row( row ), cols( cols ), prgMain( prgMain ), schmilblick( schmilblick ) {}
}; // Schmilblicks

int main() {
    int schmilblick, rows, cols;
    cin >> schmilblick >> rows >> cols;
    int M[rows][cols], r, c;
    for (r = 0; r < rows; r += 1) {
        // read/print matrix
        for (c = 0; c < cols; c += 1) {
            cin >> M[r][c];
            cout << M[r][c] <<", ";
        }
        cout << endl;
    }
    // for
    Schmilblicks *workers[rows];
    for (r = 0; r < rows; r += 1) {
        // create task to calculate rows
        workers[r] = new Schmilblicks( M[r], cols, uThisTask(), schmilblick );
    } // for
    bool found = false;
    try {
        r = 0; // initialize before Enable
        _Enable {
            for ( ; r < rows; r += 1 ) { // wait for completion and delete tasks
                delete workers[r];
            } // for
        } // _Enable
        _CatchResume( Schmilblick ) {
            if (!found) {
                for (int i = r + 1; i < rows; i += 1) {
                    _Resume Schmilblicks::Stop() _At -workers[i];
                } // for
                found = true;
            } // if
        } // try
        cout << "Schmilblicks" << (found ? " " : " not ") << "found" << endl;
    } // main
```