Part A - Multiple Choice

For these multiple choice questions, fill in the bubble sheet on the last page of the answer booklet. Answers on these pages will not be graded, and time will not be given to transfer your answers to the last page. For all given code fragments, assume that the appropriate header files have been included.

Use the following pseudo-code fragment to answer multiple choice questions 1 and 2:

```cpp
bool validate( const std::string & s ); // returns true if s is valid

int main( int argc, char * argv[] ) {
    sequel ArgError( int i ) { std::cerr << "2 != " << i; }
    std::cout << "1"
    if ( argc != 2 ) ArgError( argc );
    std::cout << "2"
    std::ifstream infile{ argv[1] };
    std::string s;
    for (;;) {
        std::cout << "3"
        sequel InputError( std::string s ) { std::cerr << "Invalid input: " << s; }
        getline( infile, s );
        if ( infile.eof() ) break;
        if ( ! validate(s) ) InputError(s);
    }
    std::cout << "4"
}
```

1. 1 mark What is the output if argc has the value of 1?
   A. 2 != 112 != 1
   B. 2 != 12
   C. 123...3
   D. 12 != 1
   E. None of these

2. 1 mark What is the output if argc equals 2, and the second line of input from the specified file consists of the string "cat" and is not valid i.e. validate returns false?
   A. 2 != 213Invalid input: cat
   B. 123Invalid input: cat
   C. 1233Invalid input: cat4
   D. 1233...34
   E. None of these
Use the following code to answer multiple choice questions 3 to 5, assuming that the appropriate keywords for the specified exception handling model have been filled in for the values of **xxx** and **yyy**, respectively raising and handling the exception:

```cpp
_event E{};
int i = 0;
void bar() {
    std::cout << "1";
    if ( i == 0 ) { i++; xxx; }
    std::cout << "2";
}
void foo() {
    try {
        bar();
        std::cout << "3";
    } yyy( E ) {
        std::cout << "4";
    }
    std::cout << "5";
}

int main() {
    foo();
}
```

3. **1 mark** What is the output produced by the termination exception handling model?
   A. 14235
   B. 145
   C. 1425
   D. 1435
   E. None of these

4. **1 mark** What is the output produced by the resumption exception handling model?
   A. 14235
   B. 145
   C. 1425
   D. 1435
   E. None of these

5. **1 mark** What is the output produced by the retry exception handling model?
   A. 14235
   B. 145
   C. 1425
   D. 1435
   E. None of these
Use the following code to answer multiple choice questions 6 to 9:

```cpp
_Coroutine X {
    void main() {
        std::cout << "1";
        suspend();
        std::cout << "2";
    } // main
}
_public :
    X() { resume(); }
    void next() { resume(); }
};
_Coroutine Y {
    void main() {
        std::cout << "3";
        X x;
        std::cout << "4";
        suspend();
x.next();
        std::cout << "5";
    } // main
}
_public :
    Y() {}
    void next() { resume(); }
};
int main() {
    Y y;
    std::cout << "6";
y.next();
    std::cout << "7";
y.next();
    std::cout << "8";
}
```

6. **1 mark** Who is the *cocaller* of x?
   A. main    B. x  C. y

7. **1 mark** Who is the *cocaller* of y?
   A. main    B. x  C. y

8. **1 mark** When y terminates, who gets control next?
   A. main    B. x  C. y

9. **1 mark** What is the output of the program?
   A. 53471258
   B. 6341523478
   C. 63124578
   D. 63147258
   E. None of these
10. **1 mark** Which rules of the *Mutual Exclusion Game* are broken by the following software solution?

```cpp
enum Intent { WantIn, DontWantIn };
_TASK Person {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;
            if ( you == DontWantIn ) break;
            me = DontWantIn;
            while ( you == WantIn ) {} } } } } } } } } } } } } } } }

```public:
    Person( Intent & me, Intent & you ) : me(me), you(you) {} ;
    int main() {
        Intent me = DontWantIn, you = DontWantIn;
        Person p0( me, you ), p1( you, me );
    }

A. safety i.e. mutual exclusion
B. threads run in arbitrary speed and order
C. if not in entry/exit/critical section, cannot prevent entry
D. liveness (no indefinite postponement)
E. eventually entry (no starvation)

11. **1 mark** Which rules of the *Mutual Exclusion Game* are broken by the following hardware solution?

```cpp
int Lock = OPEN; // shared

void AtomicSwap( int & a, & b ) {
    int temp;
    // begin atomic
    temp = a;
    a = b;
    b = temp;
    // end atomic
}

void Task::main() {
    int dummy = CLOSED;
    do {
        AtomicSwap( Lock, dummy );
    } while( dummy == CLOSED );
    CriticalSection();
    Lock = OPEN;
}
```

A. safety i.e. mutual exclusion
B. threads run in arbitrary speed and order
C. if not in entry/exit/critical section, cannot prevent entry
D. liveness (no indefinite postponement)
E. eventually entry (no starvation)
Part B - Short Answer and Code

1. (a) 2 marks Name the two language mechanisms used to perform a *static multi-level exit*.
   
   (b) 2 marks There is one reason, the same for both, why these two mechanisms cannot be used to perform a non-local transfer. Explain the reason.
   
   (c) *C longjmp and C++ throw* are two different ways of accomplishing a *dynamic multi-level exit*.
      
      i. 1 mark Describe how these two mechanisms are *similar* in terms of their control flow and what happens.
      
      ii. 1 mark Describe how these two mechanisms are *different* in terms of their control flow and what happens.
      
   (d) i. 4 marks Fill in the table in your answer booklet with the names of the appropriate mechanisms. (The numbers in the cells are used in subsequent parts of this question.)

<table>
<thead>
<tr>
<th>return/handled</th>
<th>call/raise</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>dynamic</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

   ii. 2 marks Explain the difference in *purpose* between 3) and 4).
   
   iii. 1 mark What is the disadvantage of 1)?
   
   (e) 2 marks Complete the chart in your answer booklet with the *C++* keywords needed to replace *xxx* and *yyy* so that the code below uses the specified exception handling model *raise* and *handler*.

```cpp
_Event E{};
void bar() {
    xxx E();
}
void foo() {
    try {
        bar();
    } yyy( E ) {
        // do nothing
    }
}
int main() {
    foo();
}
```

2. (a) i. 1 mark Why is the ability to receive a nonlocal exception *initially* turned off in coroutines?
   
   ii. 1 mark How is it turned on?
   
   iii. 1 mark When does the coroutine poll for nonlocal exceptions?
   
   iv. 2 marks What is meant by a *proxy* raise? Why is this important i.e. what is it used for?
   
   (b) 2 marks During execution, a *C++* coroutine *locally* raises a resumption exception; however, there is no matching resumption handler. There is, however, a termination handler that would match. Would this handler be executed? Explain your answer.
   
   (c) 2 marks What actions occur in *C++* if a coroutine does not handle an exception?
   
   (d) As the name implies, a *stackless* coroutine does not have its own runtime stack.
      
      i. 1 mark How can it still behave like a coroutine?
      
      ii. 1 mark What are its limitations?
      
      iii. 1 mark Name one programming language that supports them.
3. (a) Given the concurrency terms process and task.
   i. 1 mark How are they similar?
   ii. 1 mark How are they different?
(b) 2 marks Concurrent systems are categorized as being of three possible types: discovered, implicit, or explicit. Name one implicit system/mechanism or programming language, and one explicit system/mechanism or programming language.
(c) The following code shows Dekker’s algorithm without Hesselink’s modification to deal with flicker.

```cpp
enum Intent { WantIn, DontWantIn }
Intent * Last;
_Task Dekker {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {
                me = WantIn;
                if ( you == DontWantIn ) break;
                if ( ::Last == &me ) {
                    me = DontWantIn;
                    while ( ::Last == &me ) {} 
                }
            }
        }
    CriticalSection();
    ::Last = &me;
    me = DontWantIn;

    public:
    Dekker( Intent & me, Intent & you ) : me(me), you(you) {} }
```

i. 1 mark Explain what is meant by flicker.
ii. 2 marks Describe a situation where flicker of setting intent in line 9 would lead to breaking one of the rules of the Mutual Exclusion Game.

4. (a) i. 5 marks Given the statements:

   \[
   S_1 : A \leftarrow 1 \\
   S_2 : B \leftarrow 2 \\
   S_3 : C \leftarrow A + B \\
   S_4 : D \leftarrow 2 \times A \\
   S_5 : E \leftarrow C + D
   \]

   construct a precedence graph that shows the maximum amount of possible concurrency that provides the same results as sequential execution.

   ii. 6 marks Re-code the statements using any number of COBEGIN/COEND concurrency primitives and BEGIN/END blocks to try to achieve the concurrency of the precedence graph. Either structure can be nested. If the maximum possible concurrency cannot be achieved, do the best you can and explain why. (SEMAPHORES ARE NOT ALLOWED IN THE SOLUTION!)
(b) **2 marks** All hardware solutions that rely upon an atomic instruction are described as only being *probabilistically correct* since they all break one of the rules of the Mutual Exclusion Game. Why does this not end up being a problem in practice?

(c) The initial implementation of a blocking lock used a special routine call, `yieldNoSchedule`. 

```cpp
class MutexLock {
    bool avail; // resource available?
    Task *owner; // lock owner
    queue<Task> blocked; // blocked tasks
    SpinLock lock; // mutex nonblocking lock
public:
    MutexLock() : avail(true), owner(nullptr) {} // resource available, no owner

    void acquire() {
        lock.acquire(); // barging
        while (!avail && owner != currThread()) { // busy waiting
            // add self to lock's blocked list
            yieldNoSchedule(); // do not reschedule to ready queue
            lock.acquire(); // reacquire spinlock
        }
        avail = false; owner = currThread(); // set new owner
        lock.release();
    }

    void release() {
        lock.acquire();
        if (owner != currThread()) ... // ERROR CHECK
        owner = nullptr; // no owner
        if (!blocked.empty()) {
            // remove task from blocked list and make ready
        }
        avail = true; // reset
        lock.release(); // RACE }
};
```

i. **4 marks** Explain the two cases where the initial version fails.

ii. **2 marks** What is the modification to `yieldNoSchedule` to make it work? How does this solve the previous two problems? Make sure your answer discusses the necessary cooperation.

iii. **2 marks** There is still a problem with the corrected version. Name the problem. What were the names of the two approaches to solving the problem?

5. **18 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```cpp
Coroutine IBAN {
    public:
    _Event Match {}; // character passed by cocaller
    _Event Error {};

    private:
    char ch; // character passed by cocaller
    void main(); // YOU WRITE THIS ROUTINE

    public:
    void next(char c) {
        ch = c;
        resume();
    }
};
```
which verifies a string of characters corresponds to a International Bank Account Number (IBAN). (The IBAN is being implemented across Europe to facilitate cross-border payments.) An IBAN consists of an ISO 3166 two-letter country code (upper-case only), followed by a one-digit check digit (check digit must be > 1), and a BBAN that consists of a unique bank identifier and an account number (length of BBAN must be 2 ≤ |BBAN| ≤ 30 digits). Since the length of the BBAN is variable, its end is marked with an exclamation point (‘!’). To check the correctness of the BBAN, the digits are summed and divided by the check digit, and the remainder must be equal to 1. For example, the following are valid IBAN numbers:

BE298!
BE62514789766545031!
FR2130041010050200013602606!

Assume the following C library routines are available i.e. the appropriate library headers have been included: isdigit, isalpha, isalnum and isupper; isdigit(c) returns true if c is a digit; isalpha(c) returns true if c is a letter; isalnum(c) returns true if c is an alphanumeric (letter or digit) character; isupper(c) returns true if c is an upper case letter. (If you cannot remember how to convert a character to an integer, assume a cttoi function that takes a character as a parameter and returns an integer i.e. int cttoi(c), though there is a 1 mark deduction.)

After creation, the coroutine is resumed with a series of characters (1 character at a time). The coroutine accepts characters until:

- the characters form a valid string in the language, and it then raises the exception Grammar::Match IBAN::Match at the last resumer;
- the last character results in a string not in the language, and it then raises the exception Grammar::Error IBAN::Error at the last resumer.

After the coroutine raises a Match or Error exception, it must terminate; sending more characters to the coroutine after this point is undefined. (You may use multiple return statements in IBAN::main.)

Write ONLY IBAN::main, do NOT write a main program that uses it! No documentation or error checking of any form is required.

Note: Few marks will be given for a solution that does not take advantage of the capabilities of the coroutine, i.e., you must use the coroutine’s ability to retain data and execution state. A C-style solution will receive little or no marks.

6. Divide and conquer is a technique that can be applied to certain kinds of problems. These problems are characterized by the ability to subdivide the work across the data, such that the work can be performed independently on the data. In general, the work performed on each group of data is identical to the work that is performed on the data as a whole. What is important is that only termination synchronization is required to know the work is done; the partial results can then be processed further.

Write a program to concurrently rotate the values in the rows of a matrix. For example, in:

<table>
<thead>
<tr>
<th>original</th>
<th>rotation</th>
<th>rotated original</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td>0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td>3</td>
<td>4 5 1 2 3</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td>2</td>
<td>3 4 5 1 2</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td>1</td>
<td>2 3 4 5 1</td>
</tr>
</tbody>
</table>

each row of the original matrix is rotated to the left the number of places given in the rotation vector. All rotations are 0 or positive values. Specifically, row 1 is rotated 0 places, row 2 is rotated 3 places, row 3 is rotated 2 places, and row 4 is rotated 1 place.
Assume the given routine to rotate a single row in a matrix to the left by the specified number of positions, amount, which is greater than or equal to 0. The matrix row is passed in as an (int *), and numCols is the number of columns it has.

```c
void rotate( const unsigned int amount, int * const row, const unsigned int numCols );
```

(a) **3 marks** Using routine rotate above and the following declarations, write a COFOR statement to concurrently rotate each row of matrix M by the amount specified in rotateVector:

```c
int M[rows][cols], rotateVector[rows];
COFOR( . . . // YOU WRITE THIS STATEMENT (ASSUME APPROPRIATE INCLUDES) 
       . . .
);
```

(b) **10 marks** Using routine rotate and the declarations M and rotateVector above, write a message and actor to concurrently rotate each row of matrix M by the amount specified in rotateVector:

```c
struct WorkMsg : public uActor::Message {
   // WRITE THIS TYPE
};
_Actor Rotate {
   // WRITE THIS TYPE
};

uActorStart();
   // USE ACTOR AND MESSAGE TYPE TO PERFORM ROTATION
uActorStop();
```

All information needed to rotate a row is passed in the dynamically allocated message to the actor, not in the actor’s constructor.

(c)  

i. **3 marks** Complete the definition of task Rotate.

```c
_Task Rotate {
   . . . // ADD HERE
   void main(); // WRITE THIS ROUTINE
   public:
      Rotate( const int amount, // amount to rotate the row by
               const int row[], // row of matrix to rotate
               const int cols, // number of columns in row
               ); // WRITE THIS ROUTINE
};
```

ii. **5 marks** Assume that the program main has properly declared and read in the matrix M and the rotation vector rotateVector, setting the dimension variables rows and cols as appropriate. Fill in the section of main that concurrently rotates each row of the matrix according to the corresponding amount in the rotation vector, and prints the rotated matrix. Each row of the matrix is printed on a separate line, and every value in the row is followed by a single space.