CS 343 Fall 2013 – Assignment 1
Instructors: Bernard Wong and Peter Buhr
Due Date: Monday, September 23, 2013 at 22:00
Late Date: Wednesday, September 25, 2013 at 22:00

September 18, 2013

This assignment introduces exception handling and coroutines in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution, i.e., writing a C-style solution for questions is unacceptable, and will receive little or no marks. (You may freely use the code from these example programs.)

1. Convert the C++ program in Figure 1 from dynamic multi-level exits to:

   (a) NO **break**/**throw**, one **return** per routine, return codes, and flag variables. The return type of routines may be changed.

   (b) **break**, multiple **return**, return codes, and NO flag variables. The return type of routines may be changed.

   (c) global status-flag, **break**, multiple **return**, and ~~return codes~~ NO flag variables. The return type of routines may NOT be changed.

2. This question requires the use of μC++, which means compiling the program with the u++ command and replacing routine main with member uMain::main.

   In languages without resumption exceptions, resumption can be simulated by explicitly passing handler routines as arguments (often called fix-up routines), which are called where the resumption exception is raised. Given the μC++ program in Figure 2, p. 3:

   (a) ~~Construct a simulation of the program in C++ by passing handler functors rather than handler routines among the routines.~~ Construct a simulation of the program in C++ by passing fixup routines or functors. Do not use **void** * to represent a routine pointer because C++ does not guarantee a **void** * can hold a routine pointer. For example, on some systems, routine pointers are actually a structure with 2 or more fields.

   (b) ~~Explain why this program *cannot* be simulated by passing handler (fix-up) routines among the routines.~~ Describe your algorithm (method) for transforming from resumption to fixup routines.

3. This question requires the use of μC++, which means compiling the program with the u++ command and replacing routine main with member uMain::main.

   Write a *semi-coroutine* to verify a string of bytes is a valid Unicode Transformation Format 8-bit character (UTF-8). UTF-8 allows any universal character to be represented while maintaining full backwards-compatibility with ASCII encoding, which is achieved by using a variable-length encoding. The following table provides a summary of the Unicode value ranges in hexadecimal, and how they are represented in binary for UTF-8.

Unicode ranges	UTF-8 binary encoding
000000-00007F	0xxxxxxx
000080-0007FF	110xxxxx 10xxxxxx
000800-00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

   For example, the symbol £ is represented by Unicode value 0xA3 (binary 1010 0011). Since £ falls within the range of 0x80 to 0x7FF, it is encoded by the UTF-8 bit string 110xxxxx 10xxxxxx. To fit the character into the eleven bits of the UTF-8 encoding, it is padded on the left with zeroes to 00010100011. The UTF-8 encoding becomes 11000010 10100011, where the x's are replaced with the 11-bit binary encoding giving the UTF-8

```
#include <iostream>
#include <cstdlib>          // access: rand, srand
#include <unistd.h>         // access: getpid
using namespace std;

int times = 1000;

void rtn1( int i ) {
    for ( int j = 0; j < times; j += 1 ) {
        if ( rand() % 100000000 == 42 ) throw j;
    }
}
void rtn2( int i ) {
    for ( int j = 0; -j < times; j -= 1 ) {
        if ( rand() % 100000000 == 42 ) throw j;
    }
}
void g( int i ) {
    for ( int j = 0; j < times; j += 1 ) {
        if ( rand() % 2 == 0 ) rtn1( i );
        else rtn2( i );
    }
    if ( i % 2 ) rtn2( i );
    rtn1( i );
}
void f( int i ) {
    for ( int j = 0; j < times; j += 1 ) {
        g( i );
    }
    if ( i % 2 ) g( i );
    g( i );
}
int main( int argc, char *argv[] ) {
    int seed = getpid();
    if ( argc >= 2 ) seed = atoi( argv[1] );
    srand( seed );
    if ( argc == 3 ) times = atoi( argv[2] );
    try {
        f( 3 );
        cout << "seed:" << seed << " times:" << times << " complete" << endl;
    } catch( int rc ) {
        cout << "seed:" << seed << " times:" << times << " rc:" << rc << endl;
    }
}
```

Figure 1: Dynamic Multi-Level Exit

character encoding 0xC2A3 for symbol £. Note, UTF-8 is a minimal encoding; e.g., it is incorrect to represent the value 0 by any encoding other than the first one. Use unformatted I/O to read the Unicode bytes and the public interface given in Figure 3, p. 4 (you may only add a public constructor/destructor and private members)

After creation, the coroutine is resumed with a series of bytes from a string (one byte at a time). The coroutine returns after for each character or raises one of these two exception:

- Match means the bytes form an encoding and no more bytes can be sent,
- Error means the last byte resulted in an invalid encoding.

After the coroutine raises an exception, it must terminate; sending more bytes to the coroutine after this point is undefined.

Write a program utf8 that checks if a string follows the UTF-8 encoding. The shell interface to the utf8 program is as follows:

```
#include <iostream>
using namespace std;

_Event H {                                            // uC++ exception type
  public:
    int &i;                                           // pointer to fixup variable at raise
    H( int &i ) : i( i ) {}
};

void f( int &i );                                     // mutually recursive routines
void g( int &i );

void f( int &i ) {
    cout << "f " << i << endl;
    try {
        if ( rand() % 5 == 0 ) _Resume H( i );        // require correction ?
        if ( rand() % 7 == 0 ) g( i );                // mutual recursion
    } _CatchResume( H &h ) {                          // fixup action
        cout << "f handler, i:" << h.i << endl;
        h.i -= 7;                                     // fix variable at raise
        if ( rand() % 7 == 0 ) g( h.i );              // mutual recursion
    } // try
    if ( 0 < i ) f( i );                              // recursion
}
void g( int &i ) {
    cout << "g " << i << endl;
    try {
        if ( rand() % 7 == 0 ) _Resume H( i );        // require correction ?
        if ( rand() % 5 == 0 ) f( i );                // mutual recursion
    } _CatchResume( H &h ) {                          // fixup action
        cout << "g handler, i:" << h.i << endl;
        h.i -= 5;                                     // fix variable at raise
        if ( rand() % 5 == 0 ) f( h.i );              // mutual recursion
    } // try
    if ( 0 < i ) g( i );                              // recursion
}
void uMain::main() {
    int times = 25, seed = getpid();
    if ( argc >= 2 ) times = atoi( argv[1] );         // control recursion depth
    if ( argc == 3 ) seed  = atoi( argv[2] );         // allow repeatable experiment
    srand( seed );                                    // fixed or random seed
    f( times );
}
```

Figure 2: Resumption

```
utf8 [ filename ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no input file name is specified, input comes from standard input. Output is sent to standard output. Issue appropriate runtime error messages for incorrect usage or if a file cannot be opened.

The program should:

- read a line from the file,
- create a Utf8 coroutine,
- pass bytes from the input line to the coroutine one at,
- print an appropriate message after the coroutine raises an exception,
- terminate the coroutine,
- repeat these steps for each line in the file.

```
_Coroutine Utf8 {
  public:
    struct Match {
        unsigned int unicode;
        Match( unsigned int unicode ) : unicode( unicode ) {}
    };
    struct Error {};
  private:
    union UTF8 {
        unsigned char ch;                   // character passed by cocaller
#if defined( _BIG_ENDIAN ) || BYTE_ORDER == BIG_ENDIAN // BIG ENDIAN architecture
        struct {                            // types for 1st utf-8 byte
            unsigned char ck : 1;           // check
            unsigned char dt : 7;           // data
        } t1;
        struct {
            unsigned char ck : 3;           // check
            unsigned char dt : 5;           // data
        } t2;
        struct {
            // you figure it out
        } t3;
        struct {
            // you figure it out
        } t4;
        struct {                            // type for extra utf-8 bytes
            // you figure it out
        } dt;
#else                                       // LITTLE ENDIAN architecture
        struct {                            // types for 1st utf-8 byte
            unsigned char dt : 7;           // data
            unsigned char ck : 1;           // check
        } t1;
        struct {
            unsigned char dt : 5;           // data
            unsigned char ck : 3;           // check
        } t2;
        struct {
            // you figure it out
        } t3;
        struct {
            // you figure it out
        } t4;
        struct {                            // type for extra utf-8 bytes
            // you figure it out
        } dt;
#endif
    } utf8;
    // YOU MAY ADD PRIVATE MEMBERS
  public:
    // YOU MAY ADD CONSTRUCTOR/DESTRUCTOR IF NEEDED
    void next( unsigned char c ) {
        utf8.ch = c;                        // insert character into union for analysis
        resume();
        // if necessary throw Match or Error exception
    }
};
```

Figure 3: UTF-8 Interface

To be able to read a line from a file, the value `0xa` cannot be a UTF-8 character as it denotes newline (`'\n'`).

The input file contains an unknown number of UTF-8 characters separated by newline characters. Hence, the value `0xa` cannot be a UTF-8 character as it denotes newline (`'\n'`). For every non-empty input line, print the bytes of the Unicode character in hexadecimal as each is checked. If a valid UTF-8 character is found, print `"valid"` followed by the value of the UTF-8 character in hexadecimal; if an invalid UTF-8 character is found, print `"invalid"`. If there are any additional bytes on a line after determining if a byte sequence is valid/invalid, the coroutine raises Match or Error, print an appropriate warning message and the additional bytes in hexadecimal. Print a warning for an empty input line (i.e., a line containing only `'\n'`). Hint: to print a character in hexadecimal use the following cast:

```
char ch = 0xff;
cout << hex << (unsigned int)(unsigned char)ch << endl;
```

The following is example output:

```
0x23 : valid 0x23
0x23 : valid 0x23. Extra characters 0x23
0xd790 : valid 0x5d0
0xd7 : invalid
0xc2a3 : valid 0xa3
 : Warning! Blank line.
0xb0 : invalid
0xe0e3 : invalid
0xe98080 : valid 0x9000
0xe98080 : valid 0x9000. Extra characters 0xfff8
0xe09390 : invalid
0xff : invalid. Extra characters 0x9a84
0xf09089 : invalid
0xf0908980 : valid 0x10240
0x1 : valid 0x1
```

WARNING: On little-endian architectures (e.g., like AMD/Intel x86), the compiler reverses the bit order; hence, the bit-fields in variable utf8 above must be reversed. While it is unfortunate C/C++ bit-fields lack portability across hardware architectures, they are the highest-level mechanism to manipulate bit-specific information. Your assignment may be tested on either a big or little endian computer.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. Use of execution state variables in a coroutine usually indicates that you are not using the ability of the coroutine to remember execution location. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 4.3.1 in *Understanding Control Flow: with Concurrent Programming using μC++* for details on this issue.

Submission Guidelines

Please follow these guidelines very carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text file, i.e., *.*txt file, must be ASCII text and not exceed 500 lines in length, where a line is a maximum of 120 characters.** 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. q1flags.{cc,C,cpp}, q1returncodes.{cc,C,cpp}, q1globalstatusflag.{cc,C,cpp} – code for question 1, p. 1. **No program documentation needs to be present in your submitted code. No test, user or system documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

2. q2noresumption.txt – contains the information required by question 2b, p. 1.

3. q2noresumption.{cc,C,cpp} – code for question 2a, p. 1. **No program documentation needs to be present in your submitted code. No test, user or system documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

4. q3*.{h,cc,C,cpp} – code for question 3, p. 1. Split your code across *.h and *.{cc,C,cpp} files as needed. **Program documentation must be present in your submitted code.**

5. q3*.testdoc – test documentation for question 3, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

6. Use the following Makefile to compile the programs for questions 1, 2 and 3:

```
CXX = u++                                          # compiler
CXXFLAGS = -g -Wall -Wno-unused-label -MMD         # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}}      # makefile name

OBJECTS0 = q1exception.o                           # optional build of given program
EXEC0 = exception

OBJECTS1 = q1flags.o
EXEC1 = flags

OBJECTS2 = q1globalstatusflag.o
EXEC2 = globalstatusflag

OBJECTS3 = q1returncodes.o
EXEC3 = returncodes

OBJECTS00 = q2resumption.o                         # optional build of given program
EXEC00 = resumption

OBJECTS4 = q2noresumption.o
EXEC4 = noresumption

OBJECTS5 = # object files forming 5th executable with prefix "q3"
EXEC5 = utf8

OBJECTS = ${OBJECTS1} ${OBJECTS2} ${OBJECTS3} ${OBJECTS4} ${OBJECTS5}
DEPENDS = ${OBJECTS:.o=.d}
EXECS = ${EXEC1} ${EXEC2} ${EXEC3} ${EXEC4} ${EXEC5}

#############################################################

.PHONY : all clean

all : ${EXECS}                                     # build all executables

q1%.o : q1%.cc                                     # change compiler 1st executable
	g++ ${CXXFLAGS} -c $< -o $@

${EXEC0} : ${OBJECTS0}
	g++ ${CXXFLAGS} $^ -o $@

${EXEC1} : ${OBJECTS1}
	g++ ${CXXFLAGS} $^ -o $@

${EXEC2} : ${OBJECTS2}
	g++ ${CXXFLAGS} $^ -o $@

${EXEC3} : ${OBJECTS3}
	g++ ${CXXFLAGS} $^ -o $@

q2%.o : q2%.cc                                     # change compiler 2nd executable
	g++ ${CXXFLAGS} -c $< -o $@
```

```
${EXEC00} : ${OBJECTS00}
    g++ ${CXXFLAGS} $^ -o $@

${EXEC4} : ${OBJECTS4}
    g++ ${CXXFLAGS} $^ -o $@

${EXEC5} : ${OBJECTS5}
    ${CXX} $^ -o $@
#############################################################

${OBJECTS} : ${MAKEFILE_NAME}                   # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}                             # include *.d files containing program dependences

clean :                                         # remove files that can be regenerated
    rm -f *.d *.o ${EXEC0} ${EXEC00} ${EXECS}
```

This makefile is used as follows:

```
$ make flags
$ ./flags
$ make globalstatusflag
$ ./globalstatusflag
$ make returncodes
$ ./returncodes
$ make resumption
$ ./resumption
$ make utf8
$ utf8 ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make flags or make multiexit or make globalfixup or make grammar in the directory to compile the programs. 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. **If the makefile fails or does not produce correctly named executables, or if a program does not compile, you receive zero for all "Testing" marks.**

Follow these guidelines. Your grade depends on it!