CS 343 Fall 2016 – Assignment 1
Instructors: Peter Buhr and Aaron Moss
Due Date: Monday, September 26, 2016 at 22:00
Late Date: Wednesday, September 28, 2016 at 22:00

September 5, 2016

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. (a) Transform the program in Figure 1 replacing **throw/catch** with longjmp/setjmp. Output from the transformed program must be identical to the original program. No additional parameters may be added to routine Ackermann. Note, type jmp_buf is an array allowing instances to be passed to setjmp/longjmp without having to take the address of the argument.

   (b) i. Compare the original and transformed program with respect to performance by doing the following:
      - Compile the original **throw/catch** and setjmp/longjmp programs without print statements.
      - Time each execution using the time command:
        ```
        % time ./a.out
        3.21u 0.02s 0:03.32 100.0%
        ```
        (Output from time differs depending on the shell, but all provide user, system and real time.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).
      - Use the program command-line arguments to adjust the amount of program execution to get execution times in the range 10 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
      - Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate the experiments.

      ii. State the observed performance difference between the original and transformed program, without and with optimization.

      iii. Speculate as to the reason for the performance difference.

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.

   (a) Transform the program in Figure 2, p. 3 into a C++11 program replacing **_Resume/_CatchResume** with fixup routines. Use of C++11 lambda routines to mimic the **_CatchResume** handlers or **template** routines are not allowed because they cannot be separately complied. Output and control flow from the transformed program must be identical to the original program. No global variables may be created; additional parameters may be created for routines B and C.

   (b) i. Compare the original and transformed program with respect to performance by doing the following:
      - Compile the original **_Resume/_CatchResume** and fixup-routine programs without print statements.
      - Time each execution using the time command:
        ```
        % time ./a.out
        3.21u 0.02s 0:03.32 100.0%
        ```

```
#include <iostream>
#include <cstdlib>
using namespace std;
#include <unistd.h>                // getpid

#ifdef NOOUTPUT
#define print( x )
#else
#define print( x ) x
#endif

struct E {};

long int freq = 5;

long int Ackermann( long int m, long int n ) {
    if ( m == 0 ) {
        if ( random() % freq == 0 ) throw E();
        return n + 1;
    } else if ( n == 0 ) {
        if ( random() % freq == 0 ) throw E();
        try {
            return Ackermann( m - 1, 1 );
        } catch( E ) {
            print( cout << "E1 " << m << " " << n << endl );
        } // try
    } else {
        try {
            return Ackermann( m - 1, Ackermann( m, n - 1 ) );
        } catch( E ) {
            print( cout << "E2 " << m << " " << n << endl );
        } // try
    } // if
    return 0;    // recover by returning 0
}

int main( int argc, const char *argv[] ) {
    long int Ackermann( long int m, long int n );
    long int m = 4, n = 6, seed = getpid();  // default values

    switch ( argc ) {
      case 5: freq = atoi( argv[4] );
      case 4: seed = atoi( argv[3] );
      case 3: n = atoi( argv[2] );
      case 2: m = atoi( argv[1] );
    } // switch
    srandom( seed );
    cout << m << " " << n << " " << seed << " " << freq << endl;
    try {
        cout << Ackermann( m, n ) << endl;
    } catch( E ) {
        print( cout << "E3" << endl );
    } // try
}
```

Figure 1: Throw/Catch

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

#ifdef NOOUTPUT
#define print( x )
#else
#define print( x ) x
#endif

_Event E1 {
  public:
    int &i, &j;
    E1( int &i, int &j ) : i( i ), j( j ) {}
};
_Event E2 {
  public:
    int &i;
    E2( int &i ) : i( i ) {}
};
_Event E3 {
  public:
    int i;
    E3( int i ) : i( i ) {}
};

int C( int i, int j ) {
    print( cout << i << " " << j << endl );
    _Resume E1( i, j );
    print( cout << i << " " << j << endl );
    _Resume E2( j );
    print( cout << i << " " << j << endl );
    _Resume E3( 27 );
    return i;
}
int B( int i, int j ) {
    if ( i > 0 ) B( i - 1, j );
    return C( i, j );
}
int A( int i, int j, int times ) {
    int k = 27, ret;
    try {
        for ( int i = 0; i < times; i += 1 ) {
            ret = B( i, j );
        }
        return ret;
    } _CatchResume( E1 e ) {
        e.i = i;
        e.j = j;
    } _CatchResume( E2 e ) {
        e.i = k;
    } _CatchResume( E3 e ) {
        print( cout << e.i << " " << j << endl );
    }
}
void uMain::main() {
    long int m = 4, n = 6, times = 1;     // default values

    switch ( argc ) {
      case 4: times = atoi( argv[3] );
      case 3: n = atoi( argv[2] );
      case 2: m = atoi( argv[1] );
    } // switch
    cout << m << " " << n << " " << A( m, n, times ) << endl;
}
```

Figure 2: Resume/CatchResume

(Output from time differs depending on the shell, but all provide user, system and real time.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).

- Use the program command-line arguments to adjust the amount of program execution to get execution times in the range 10 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate the experiments.

ii. State the observed performance difference between the original and transformed program, without and with optimization.

iii. Speculate as to the reason for the performance difference.

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* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine FloatConstant {
  public:
    enum Status { CONT, MATCH };      // possible status
  private:
    // YOU ADD MEMBERS HERE
    void main();                      // coroutine main
  public:
    _Event Error {};                  // last character is invalid
    Status next( char c ) {
        ch = c;                       // communication in
        resume();                     // activate
        return status;                // communication out
    }
};
```

   which verifies a string of characters corresponds to a C++ floating-point constant described by the following grammar:

   floating-constant : $sign_{opt}$ *fractional-constant* $exponent\text{-}part_{opt}$ $floating\text{-}suffix_{opt}$ |
   $sign_{opt}$ *digit-sequence exponent-part* $floating\text{-}suffix_{opt}$

   fractional-constant : $digit\text{-}sequence_{opt}$ "." *digit-sequence* | *digit-sequence* "."

   exponent-part : { "e" | "E" } $sign_{opt}$ *digit-sequence*

   sign : "+" | "-"

   digit-sequence : *digit* | *digit-sequence digit*

   floating-suffix : "f" | "l" | "F" | "L"

   digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

   Where X_{opt} means X | ε and ε means empty. In addition, there is a maximum of 16 digits for the mantissa (non-exponent digits) and 3 digits for the characteristic (exponent digits). For example, the following are valid C/C++ floating-point constants:

```
123.456
-.099
+555.
2.7E+1
-3.555E-12
```

   After creation, the coroutine is resumed with a series of characters from a string (one character at a time). The coroutine returns a status for each character or throws an exception:

- status CONT means another character must be sent or the string is not a floating-point constant.
- status MATCH means the characters currently form a valid substring of a floating-point constant (e.g., 1., .1) but more characters can be sent (e.g., e12),
- exception Error means the last character resulted in a string that is not a floating-point constant.

If the coroutine returns CONT and there are no more characters, the string is not a floating-point constant. Otherwise, continue passing characters until all the characters are checked. If the coroutine raises Error, the last character passed is not part of a floating-point constant. After the coroutine identifies a complete floating point number (i.e., no more characters can be added to the string) or raises Error, the coroutine terminates; sending more characters to the coroutine after this point is undefined.

Write a program floatconstant that checks if a string is a floating-point constant. The shell interface to the floatconstant program is as follows:

```
floatconstant [ infile ]
```

(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 input file contains an unknown number of floating-point constants characters separated by newline characters. For every non-empty input line, print the line, how much of the line is parsed, and the string yes if the string is a valid floating-point constant and the string no otherwise. If there are extra characters on a line after parsing, print these characters with an appropriate warning. Print a warning for an empty input line, i.e., a line containing only '\n'. The following is example output:

```
"+1234567890123456." : "+1234567890123456." yes
"+12.E-2" : "+12.E-2" yes
"-12.5" : "-12.5" yes
"12." : "12." yes
"-.5" : "-.5" yes
".1E+123" : ".1E+123" yes
"-12.5F" : "-12.5F" yes
"" : Warning! Blank line.
"a" : "a" no
"+." : "+." no
" 12.0" : " " no -- extraneous characters "12.0"
"12.0  " : "12.0 " no -- extraneous characters "  "
"1.2.0a" : "1.2." no -- extraneous characters "0a"
"-12.5F  " : "-12.5F " no -- extraneous characters " "
"123.ff" : "123.ff" no
"0123456789.0123456E-0124" : "0123456789.0123456" no -- extraneous characters "E-0124"
```

Assume a *valid* floating-point constant starts at the beginning of the input line, i.e., there is no leading white-space. See the C library routine isdigit(c), which returns true if character c is a digit.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. A state variable contains information that is not part of the computation and exclusively used for control-flow purposes (like flag variables). Use of execution state variables in a coroutine usually indicates you are not using the ability of the coroutine to remember prior execution information. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 4.4 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. q1longjmp.{cc,C,cpp} – code for question 1a, 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. q1longjmp.txt – contains the information required by question 1b, p. 1.

3. q2fixup.{cc,C,cpp} – code for question 2b, 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. q2fixup.txt – contains the information required by question 2a, p. 1.

5. q3*.{h,cc,C,cpp} – code for question 3, p. 4. Split your code across *.h and *.{cc,C,cpp} files as needed. **Program documentation must be present in your submitted code. No 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.**

6. 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.**

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

```
CXX = u++                                     # compiler
CXXFLAGS = -g -Wall -Wno-unused-label -MMD -std=c++11 # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}}# makefile name

OBJECTS01 = q1throwcatch.o                    # optional build of given program
EXEC01 = throwcatch                           # 0th executable name

OBJECTS1 = # object files forming 1st executable with prefix "q1"
EXEC1 = longjmp                               # 1st executable name

OBJECTS02 = q2resumption.o                    # optional build of given program
EXEC02 = resumption

OBJECTS2 = # object files forming 2nd executable with prefix "q2"
EXEC2 = fixup                                 # 2nd executable name

OBJECTS3 = # object files forming 3rd executable with prefix "q3"
EXEC3 = floatconstant                         # 3rd executable name

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

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

.PHONY : all clean

all : ${EXECS}                                # build all executables

q1.o : q1.cc                                  # change compiler 1st executable, ADJUST SUFFIX (.cc)
	g++-4.9 ${CXXFLAGS} -c $< -o $@

q1%.o : q1%.cc                                # change compiler 1st executable, ADJUST SUFFIX (.cc)
	g++-4.9 ${CXXFLAGS} -c $< -o $@

${EXEC01} : ${OBJECTS01}
	g++-4.9 ${CXXFLAGS} $^ -o $@

${EXEC1} : ${OBJECTS1}
	g++-4.9 ${CXXFLAGS} $^ -o $@
```

```
${OBJECTS02} : q2resumption.cc
	${CXX} ${CXXFLAGS} -c $< -o $@

${EXEC02} : ${OBJECTS02}
	${CXX} ${CXXFLAGS} $^ -o $@

q2.o : q2.cc                          # change compiler 2nd executable, ADJUST SUFFIX (.cc)
	g++-4.9 ${CXXFLAGS} -c $< -o $@

q2%.o : q2%.cc                        # change compiler 2nd executable, ADJUST SUFFIX (.cc)
	g++-4.9 ${CXXFLAGS} -c $< -o $@

${EXEC2} : ${OBJECTS2}
	g++-4.9 ${CXXFLAGS} $^ -o $@

${EXEC3} : ${OBJECTS3}
	${CXX} ${CXXFLAGS} $^ -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 ${EXEC01} ${EXEC02} ${EXECS}
```

This makefile is used as follows:

```
$ make longjmp
$ longjmp . . .
$ make fixup
$ fixup
$ make floatconstant
$ floatconstant . . .
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make longjmp or make fixup or make floatconstant 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.

Follow these guidelines. Your grade depends on it!