CS 343 Winter 2020 – Assignment 1
Instructor: Caroline Kierstead
Due Date: Monday, January 20, 2020 at 22:00
Late Date: Wednesday, January 22, 2020 at 22:00

December 17, 2019

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. Unless otherwise specified, 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) Except for the code handling the command-line arguments in the C++ versions or the C version without exceptions, transform the C++ program in Figure 1 replacing **throw/catch** with:

   i. C++ program using global status-flag variables. Return codes may NOT be returned from the routines.

   ii. C++ program using a C++17 variant return-type as return codes. There are two approaches: passing the errors by value or pointer (using inheritance) in the variant return-type.

   iii. C program using a tagged **union** return-type as return codes.

   Output from the transformed programs must be identical to the original program. Use printf format "%g" to print floating-point numbers in C.

   (b) i. Compare the original and transformed programs with respect to performance by doing the following:

   - Time the executions using the time command:

     ```
     $ /usr/bin/time -f "%Uu %Ss %E" ./a.out 100000000 10000 1003
     3.21u 0.02s 0:03.32
     ```

     Output from time differs depending on the shell, so use the system time command. 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).
   - If necessary, change the first command-line parameter times to adjust program execution into the range 1 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 the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2).
   - Include the 8 timing results to validate the experiments.

   ii. State the performance difference (larger/smaller/by how much) between the original and transformed programs, and the reason for the difference.

   iii. State the performance difference (larger/smaller/by how much) between the original and transformed programs when compiler optimization is used.

   (c) i. Run a similar experiment with compiler optimization turned on but vary the error period (second command-line parameter eperiod) with values 1000, 100, and 50.

   - Include the 12 timing results to validate the experiments.

   ii. State the performance difference (larger/smaller/by how much) between the original and transformed programs as the error period decreases, and the reason for the difference.

2. (a) Except for the code handling the command-line arguments, transform the C++ program in Figure 2, p. 3 replacing **throw/catch** with longjmp/setjmp. No additional parameters may be added to routine Ackermann. No dynamic allocation is allowed, but creation of a global variable is allowed. No more calls to setjmp

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

struct Er1 { short int code; };
struct Er2 { int code; };
struct Er3 { long int code; };

int eperiod = 10000;                                   // error period

double rtn1( double i ) {
    if ( rand() % eperiod == 0 ) throw Er1{ (short int)rand() };
    return i;
}
double rtn2( double i  ) {
    if ( rand() % eperiod == 0 ) throw Er2{ rand() };
    return rtn1( i ) + i;
}
double rtn3( double i  ) {
    if ( rand() % eperiod == 0 ) throw Er3{ rand() };
    return rtn2( i ) + i;
}
int main( int argc, char * argv[] ) {
    int times = 100000000, seed = getpid();            // default values
    try {
        switch ( argc ) {
          case 4: if ( strcmp( argv[3], "d" ) != 0 ) {     // default ?
                seed = stoi( argv[3] ); if ( seed <= 0 ) throw 1;
            } // if
          case 3: if ( strcmp( argv[2], "d" ) != 0 ) {     // default ?
                eperiod = stoi( argv[2] ); if ( eperiod <= 0 ) throw 1;
            } // if
          case 2: if ( strcmp( argv[1], "d" ) != 0 ) {     // default ?
                times = stoi( argv[1] ); if ( times <= 0 ) throw 1;
            } // if
          case 1: break;                               // use all defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cerr << "Usage: " << argv[0] << " [ times > 0 | d [ eperiod > 0 | d [ seed > 0 ] ] ]" << endl;
        exit( EXIT_FAILURE );
    } // try
    srand( seed );

    double rv = 0.0;
    int ev1 = 0, ev2 = 0, ev3 = 0;
    int rc = 0, ec1 = 0, ec2 = 0, ec3 = 0;

    for ( int i = 0; i < times; i += 1 ) {
        try { rv += rtn3( i ); rc += 1; }
        // analyse error
        catch( Er1 ev ) { ev1 += ev.code; ec1 += 1; }
        catch( Er2 ev ) { ev2 += ev.code; ec2 += 1; }
        catch( Er3 ev ) { ev3 += ev.code; ec3 += 1; }
    } // for
    cout << "normal result " << rv << " exception results " << ev1 << ' ' << ev2 << ' ' << ev3 << endl;
    cout << "calls "  << rc << " exceptions " << ec1 << ' ' << ec2 << ' ' << ec3 << endl;
}
```

Figure 1: Dynamic Multi-Level Exit

```
#include <iostream>
#include <cstdlib>                                      // access: rand, srand
#include <cstring>                                      // access: strcmp
using namespace std;
#include <unistd.h>                                     // access: getpid
#ifdef NOOUTPUT
#define PRT( stmt )
#else
#define PRT( stmt ) stmt
#endif // NOOUTPUT
struct E {};                                            // exception type
PRT( struct T { ~T() { cout << "~"; } }; )
long int eperiod = 100, excepts = 0, calls = 0;         // exception period

long int Ackermann( long int m, long int n ) {
    calls += 1;
    if ( m == 0 ) {
        if ( rand() % eperiod == 0 ) { PRT( T t; ) excepts += 1; throw E(); }
        return n + 1;
    } else if ( n == 0 ) {
        try { return Ackermann( m - 1, 1 );
        } catch( E ) {
            PRT( cout << "E1 " << m << " " << n << endl );
            if ( rand() % eperiod == 0 ) { PRT( T t; ) excepts += 1; throw E(); }
        } // try
    } else {
        try { return Ackermann( m - 1, Ackermann( m, n - 1 ) );
        } catch( E ) {
            PRT( cout << "E2 " << m << " " << n << endl );
            if ( rand() % eperiod == 0 ) { PRT( T t; ) excepts += 1; throw E(); }
        } // try
    } // if
    return 0;                                           // recover by returning 0
}
int main( int argc, char * argv[] ) {
    long int m = 4, n = 6, seed = getpid();             // default values
    try {                                               // process command-line arguments
        switch ( argc ) {
          case 5: if ( strcmp( argv[4], "d" ) != 0 ) {  // default ?
                eperiod = stoi( argv[4] ); if ( eperiod <= 0 ) throw 1; } // if
          case 4: if ( strcmp( argv[3], "d" ) != 0 ) {  // default ?
                seed = stoi( argv[3] ); if ( seed <= 0 ) throw 1; } // if
          case 3: if ( strcmp( argv[2], "d" ) != 0 ) {  // default ?
                n = stoi( argv[2] ); if ( n < 0 ) throw 1; } // if
          case 2: if ( strcmp( argv[1], "d" ) != 0 ) {  // default ?
                m = stoi( argv[1] ); if ( m < 0 ) throw 1; } // if
          case 1: break;                                // use all defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cerr << "Usage: " << argv[0] << " [ m (>= 0) | d [ n (>= 0) | d"
             " [ seed (> 0) | d [ eperiod (> 0) | d ] ] ] ]" << endl;
        exit( EXIT_FAILURE );
    } // try
    srand( seed );                                      // seed random number
    try {                                               // begin program
        PRT( cout << m << " " << n << " " << seed << " " << eperiod << endl );
        long int val = Ackermann( m, n );
        PRT( cout << val << endl );
    } catch( E ) {
        PRT( cout << "E3" << endl );
    } // try
    cout << "calls " << calls << ' ' << " exceptions " << excepts << endl;
}
```

Figure 2: Throw/Catch

than the number of **try** ... **catch**(E) statements. Note, type jmp_buf is an array allowing instances to be passed to setjmp/longjmp without having to take the address of the argument. Output from the transformed program must be identical to the original program, **except for one aspect, which you will discover in the transformed program**.

(b) i. Explain why the output is not the same between the original and transformed program.

ii. Compare the original and transformed programs with respect to performance by doing the following:

- Recompile both the programs with preprocessor option -DNOOUTPUT to suppress output.
- Time the executions using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %E" ./a.out 11 11 103 13
3.21u 0.02s 0:03.32
```

  Output from time differs depending on the shell, so use the system time command. 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 (as necessary) to adjust program execution into the range 1 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 the 4 timing results to validate the experiments.

iii. State the performance difference (larger/smaller/by how much) between the original and transformed programs, and what caused the difference.

iv. State the performance difference (larger/smaller/by how much) between the original and transformed programs when compiler optimization is used.

3. This question requires the use of μC++, which means compiling the program with the u++ command.

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 3.1.3 in the Course Notes for details on this issue. Also, make sure a coroutine's public methods are used for passing information to the coroutine, but not for doing the coroutine's work, which must be done in the coroutine's main.

Submission Guidelines

Follow these guidelines carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text or test-document file, e.g., *.{txt,doc} file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.doc | wc -l.** 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. q1returnglobal.{cc,C,cpp}, q1returntype.{cc,C,cpp}, q1returntypec.c – code for question 1a, p. 1. **No program documentation needs to be present in your submitted code. No test 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. q1returntype.txt – contains the information required by questions 1b, p. 1 and 1c, p. 1.

3. q2longjmp.{cc,C,cpp} – code for question 2a, p. 1. **No program documentation needs to be present in your submitted code. No test 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. q2longjmp.txt – contains the information required by question 2b, p. 4.

5. q3floatconstant.{h,cc,C,cpp}, q3main.{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. Output for this question is checked via a marking program, so it must match exactly with the given program.**

6. q3.doc – test documentation for question 3, which includes the input and output of your tests. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

7. Modify the following Makefile to compile the programs for question 1, p. 1, question 2a, p. 1, and question 3, p. 4 by inserting the object-file names matching your source-file names.

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

OBJECTS01 = q1exception.o                   # optional build of given program
EXEC01 = exception                          # given executable name

OBJECTS1 = q1returnglobal.o                 # 1st executable object files
EXEC1 = returnglobal                        # 1st executable name

OBJECTS2 = q1returntype.o                   # 2nd executable object files
EXEC2 = returntype                          # 2nd executable name

OBJECTS3 = q1returntypec.o                  # 3rd executable object files
EXEC3 = returntypec                         # 3rd executable name

OBJECTS02 = q2throwcatch.o                  # optional build of given program
EXEC02 = throwcatch                         # given executable name

OBJECTS4 = q2longjmp.o                      # 4th executable object files
EXEC4 = longjmp                             # 4th executable name

OBJECTS5 = q3floatconstant.o q3main.o       # 5th executable object files
EXEC5 = floatconstant                       # 5th executable name

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

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

.PHONY : all clean

all : ${EXECS}                              # build all executables

${EXEC01} : ${OBJECTS01}                    # optional build of given program
	g++ ${CXXFLAGS} $^ -o $@

q1%.o : q1%.cc                              # change compiler 1st executable, ADJUST SUFFIX (for .C/.cpp)
	g++ ${CXXFLAGS} -std=c++17 -c $< -o $@

${EXEC1} : ${OBJECTS1}                      # compile and link 1st executable
	g++ ${CXXFLAGS} $^ -o $@

${EXEC2} : ${OBJECTS2}                      # compile and link 2nd executable
	g++ ${CXXFLAGS} $^ -o $@

q1%.o : q1%.c                               # change compiler 2nd executable
	gcc ${CXXFLAGS} -c $< -o $@

${EXEC02} : ${OBJECTS02}                    # optional build of given program
	g++ ${CXXFLAGS} $^ -o $@

${EXEC3} : ${OBJECTS3}                      # compile and link 3rd executable
	g++ ${CXXFLAGS} $^ -o $@

q2%.o : q2%.cc                              # change compiler 4th executable, ADJUST SUFFIX (for .C/.cpp)
	g++ ${CXXFLAGS} -c $< -o $@
```

```
${EXEC4} : ${OBJECTS4}                    # compile and link 4th executable
    g++ ${CXXFLAGS} $^ -o $@

${EXEC5} : ${OBJECTS5}                    # compile and link 5th executable
    ${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 returnglobal
$ returnglobal ...
$ make returntype
$ returntype ...
$ make returntypec
$ returntypec ...
$ make longjmp
$ longjmp ...
$ make floatconstant
$ floatconstant ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make returnglobal, make returntype, make returntypec, make longjmp, 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!