CS 343 Winter 2018 – Assignment 1
Instructor: Peter Buhr
Due Date: Wednesday, January 17, 2018 at 22:00
Late Date: Friday, January 19, 2018 at 22:00

January 1, 2018

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. Except for the code handling the command-line arguments, transform 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 and may be a structure.

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

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

   Output from the transformed programs must be identical to the original program.

2. (a) Except for the code handling the command-line arguments, transform the 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. 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
     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 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.

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

int times = 1000;                           // default value

void rtn1( int i ) {
    for ( int j = 0; j < times; j += 1 ) {
        if ( rand() % 10000 == 42 ) throw j;
    }
}
void rtn2( int i ) {
    for ( int j = times; j >= 0; j -= 1 ) {
        if ( rand() % 10000 == 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();                    // default value
    try {                                   // process command-line arguments
        switch ( argc ) {
          case 3: times = stoi( argv[2] ); if ( times <= 0 ) throw 1;
          case 2: seed = stoi( argv[1] ); if ( seed <= 0 ) throw 1;
          case 1: break;                    // use defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ seed (> 0) [ times (> 0) ] ]" << endl;
        exit( 1 );
    } // try
    srand( seed );                          // set random-number seed
    try {                                   // begin program
        f( 3 );
        cout << "seed:" << seed << " times:" << times << " complete" << endl;
    } catch( int rc ) {
        cout << "seed:" << seed << " times:" << times << " rc:" << rc << endl;
    } // try
}
```

Figure 1: Dynamic Multi-Level Exit

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

#ifdef NOOUTPUT
#define PRT( stmt )
#else
#define PRT( stmt ) stmt
#endif // NOOUTPUT

PRT( struct T { ~T() { cout << "~"; } }; )

struct E {};                           // exception type
long int freq = 5;                     // exception frequency

long int Ackermann( long int m, long int n ) {
    PRT( T t; )
    if ( m == 0 ) {
        if ( rand() % freq == 0 ) 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() % freq == 0 ) throw E();
        } // try
    } else {
        try {
            return Ackermann( m - 1, Ackermann( m, n - 1 ) );
        } catch( E ) {
            PRT( cout << "E2 " << m << " " << n << endl );
        } // 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: freq = stoi( argv[4] ); if ( freq <= 0 ) throw 1;
          case 4: seed = stoi( argv[3] ); if ( seed <= 0 ) throw 1;
          case 3: n = stoi( argv[2] ); if ( n < 0 ) throw 1;
          case 2: m = stoi( argv[1] ); if ( m < 0 ) throw 1;
          case 1: break;               // use defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ m (> 0) [ n (> 0) [ seed (> 0)"
            " [ freq (> 0) ] ] ] ]" << endl;
        exit( 1 );
    } // try
    srand( seed );                     // seed random number
    try {                              // begin program
        PRT( cout << m << " " << n << " " << seed << " " << freq << endl );
        long int val = Ackermann( m, n );
        PRT( cout << val << endl );
    } catch( E ) {
        PRT( cout << "E3" << endl );
    } // try
}
```

Figure 2: Throw/Catch

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 Money {
    // YOU ADD MEMBERS HERE
    void main();                        // coroutine main
  public:
    _Event Match {};                    // last character match
    _Event Error {};                    // last character invalid
    void next( char c ) {
        ch = c;                         // communication input
        resume();                       // activate
    }
};
```

   which verifies a string of characters corresponds to a valid representation of money. The format for money starts with a currency $ or E (Euro), followed by an optional minus -, followed by 1-3 digits, followed by groups of 3 digits separated by a separator character, followed by a decimal point, followed by 2 digits. For a starting character of $, the separator is ',' and the decimal point is '.', and for a starting character of E, the separator and decimal are reversed. There is a maximum size of 5 groups of 3 digits. For example:

valid strings	invalid strings
$1.00	$1.0
E1,00	E,00
$-1.00	$+1.00
E1.234,00	E1,234,00
$-12,345.67	$1234.56
E123.456,78	E123.456,7

   After creation, the coroutine is resumed with a series of characters from a string (one character at a time). The coroutine raises one of the following exceptions at its resumer:

   - Match means the characters form a valid string.
   - Error means the last character forms an invalid string.

   After the coroutine raises an exception, it must NOT be resumed again; sending more characters to the coroutine after this point is undefined and should generate an error.

   Write a program money that checks if strings are valid representations of money. The shell interface to the money program is as follows:

```
money [ 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. *For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.*

   The program should:

   - read a line from the file,
   - create a Money coroutine,
   - pass characters from the input line to the coroutine one at time.
   - print an appropriate message when the coroutine returns exception Match or Error, or if there are no more characters to send.
   - check for extra characters,
   - terminate the coroutine, and
   - repeat these steps for each line in the file.

For every non-empty input line, print the line, how much of the line is parsed, and the string yes if the string is valid and no otherwise. If there are extra characters (including whitespace) on a line after parsing, print these characters with an appropriate warning. Print an appropriate warning for an empty input line, i.e., a line containing only '\n'. The following is some example output:

```
'$1.00' : '$1.00' yes
'E1,00' : 'E1,00' yes
'$-1.00' : '$-1.00' yes
'E1.234,00' : 'E1.234,00' yes
'$-12,345.67' : '$-12,345.67' yes
'E123.456,78' : 'E123.456,78' yes
'$1,222,333,444,555.45' : '$1,222,333,444,555.45' yes
'' : Warning! Blank line.
'$1.0' : '$1.0' no
'E,00' : 'E,' no -- extraneous characters '00'
'$+1.00' : '$+' no -- extraneous characters '1.00'
'E1,234,00' : 'E1,23' yes -- extraneous characters '4,00'
'$1234.56' : '$1234' no -- extraneous characters '.56'
'E123.456,7' : 'E123.456,7' no
'E-1.222.333.444.555.666,45' : 'E-1.222.333.444.555.' no -- extraneous characters '666,45'
```

Assume a *valid* string starts at the beginning of the input line, i.e., there is no leading whitespace. See the C library routine isdigit(d), which returns true if character d is a decimal 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.

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}, q1globalstatusflag.{cc,C,cpp}, q1returncodes.{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. q2longjmp.{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.**

3. q2longjmp.txt – contains the information required by question 2b, p. 1.

4. q3*.{h,cc,C,cpp} – code for question 3. 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.**

5. q3*.testdoc – 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.**

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

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

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

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

OBJECTS2 = # object files forming 2nd executable with prefix "q1"
EXEC2 = returncodes                      # 2nd executable name

OBJECTS3 = # object files forming 3rd executable with prefix "q1"
EXEC3 = globalstatusflag                 # 3rd executable name

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

OBJECTS4 = # object files forming 4th executable with prefix "q2"
EXEC4 = longjmp                          # 4th executable name

OBJECTS5 = # object files forming 5th executable with prefix "q3"
EXEC5 = money                      # 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++-6 ${CXXFLAGS} $^ -o $@

q1.o : q1%.cc                            # change compiler 1st executable, ADJUST SUFFIX (for .C/.cpp)
	g++-6 ${CXXFLAGS} -c $< -o $@

q1%.o : q1%.cc                           # change compiler 1st executable, ADJUST SUFFIX (for .C/.cpp)
	g++-6 ${CXXFLAGS} -c $< -o $@

${EXEC1} : ${OBJECTS1}                   # compile and link 1st executable
	g++-6 ${CXXFLAGS} $^ -o $@

${EXEC02} : ${OBJECTS02}                 # optional build of given program
	g++-6 ${CXXFLAGS} $^ -o $@

${EXEC2} : ${OBJECTS2}                   # compile and link 2nd executable
	g++-6 ${CXXFLAGS} $^ -o $@

${EXEC3} : ${OBJECTS3}                   # compile and link 3rd executable
	g++-6 ${CXXFLAGS} $^ -o $@

q2.o : q2.cc                             # change compiler 2nd executable, ADJUST SUFFIX (for .C/.cpp)
	g++-6 ${CXXFLAGS} -c $< -o $@

q2%.o : q2%.cc                           # change compiler 2nd executable, ADJUST SUFFIX (for .C/.cpp)
	g++-6 ${CXXFLAGS} -c $< -o $@
```

```
${EXEC4} : ${OBJECTS4}                # compile and link 4th executable
    g++-6 ${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 exception
$ exception ...
$ make flags
$ flags ...
$ make returncodes
$ returncodes ...
$ make globalstatusflag
$ globalstatusflag ...
$ make longjmp
$ longjmp ...
$ make money
$ money ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make exception, make flags, make globalstatusflag, make returncodes, make longjmp, or make money 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!