CS 343 Fall 2012 – Assignment 1
Instructor: Martin Karsten
Due Date: Monday, September 24, 2012 at 22:00
Late Date: Wednesday, September 26, 2012 at 22:00

September 17, 2012

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. Given the C++ program in Figure 1, run the program with and without preprocessor variable EXCEPT defined.

   (a) Compare the two versions of the program with respect to performance by doing the following:

```
#include <cstdlib>                          // atoi
void rtn( volatile int i ) {                // ignore volatile, prevents eliminating increment of i
    if ( i == 0 ) throw 0;                  // recursion base case
    rtn( i - 1 );
    i += 1;                                 // ignore, needed to prevent tail-recursion optimization
}
int main( int argc, char *argv[] ) {
    int times = 100000, recurse = -1;
    switch ( argc ) {
      case 3:
        recurse = atoi( argv[2] );
      case 2:
        times = atoi( argv[1] );
    } // switch
    for ( int t = 0; t < times; t += 1 ) {
#ifdef EXCEPT
        try {
#endif
            for ( int i = 0; i < 10; i += 1 ) {
                for ( int j = 0; j < 10; j += 1 ) {
                    for ( int k = 0; k < 10; k += 1 ) {
                        if ( argc >= 1 || ( i == 5 && j == 3 && k == 5 ) ) { // ignore, prevent loop elimination
#ifdef EXCEPT
                            if ( recurse < 0 ) throw 0;     // administrative cost, no stack unwinding
                            else rtn( recurse );            // try 2,4,8 for cost of stack unwinding
#else
                            goto fini;
#endif
                        } // if
                    } // for
                } // for
            } // for
          fini: ;
#ifdef EXCEPT
        } catch( int ) {
        } // try
#endif
    } // for
} // main
```

Figure 1: Static versus Dynamic Multi-Level Exit

- Time the 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 argument (if necessary) to adjust the number of times the experiment is performed to get execution times approximately in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line value for all experiments.
- 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.
- Explain the relative differences in the timing results with respect to static versus dynamic multi-level exit within a routine.
- State the performance difference when compiler optimization is used?
- **BONUS:** If there is a difference in performance using `-O2`, explain what compiler optimization is being performed to achieve the performance gain.

(b) Determine the cost of propagation and stack unwinding.

- Compile with `EXCEPT` defined and the compiler optimization turned on, and then run the program varying the number of recursive calls to `rtn` to increase the stack depth before the exception is raised.
- Explain the relative differences in the timing results with respect to propagation among routines. Include a number of timing results to validate your explanation.

(c) Explain when dynamic multi-level exit is necessary and unnecessary.

2. (a) Rewrite the program in Figure 2 replacing **throw**/**catch** with `longjmp`/`setjmp`. Except for a `jmp_buf` variable to replace the exception variable created by the **throw**, no new variables may be created to accomplish the transformation. ***Zero marks will be given to a transformation violating these restrictions.*** 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. Compare the original and your transformed program with respect to performance by doing the following:
   - Comment out *all* the print (`cout`) statements in the original and your rewritten version.
   - 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 your 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 (if necessary), e.g., "1000 1000 100000", to adjust the amount of program execution to get execution times in 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 your 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 your 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 major performance difference.

3. This question requires the use of μC++, which means compiling the program with the `u++` command, including `uC++.h` as the first include file in each translation unit, and replacing routine `main` with member `uMain::main`.

```
#include <iostream>
using namespace std;
#include <cstdlib>                              // exit, atoi

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

struct E {};
unsigned int hc, gc, fc, kc;

void f( volatile int i ) {                      // volatile, prevent dead-code optimizations
    T t;
    cout << "f enter" << endl;
  if ( i == 3 ) throw E();
    if ( i != 0 ) f( i - 1 );
    cout << "f exit" << endl;
    kc += 1;                                    // prevent tail recursion optimization
}

void g( volatile int i ) {
    cout << "g enter" << endl;
    if ( i % 2 == 0 ) f( fc );
    if ( i != 0 ) g( i - 1 );
    cout << "g exit" << endl;
    kc += 1;
}

void h( volatile int i ) {
    cout << "h enter" << endl;
    if ( i % 3 == 0 ) {
        try {
            f( fc );
        } catch( E ) {
            cout << "handler 1" << endl;
            try {
                g( gc );
            } catch( E ) {
                cout << "handler 2" << endl;
            }
        }
    }
    if ( i != 0 ) h( i - 1 );
    cout << "h exit" << endl;
    kc += 1;
}

int main( int argc, char *argv[] ) {
    if ( argc != 4 ) {
        cerr << "Usage: " << argv[0] << " hc gc fc" << endl;
        exit( EXIT_FAILURE );
    } // if
    // assume positive values
    hc = atoi( argv[1] );                       // h recursion depth
    gc = atoi( argv[2] );                       // g recursion depth
    fc = atoi( argv[3] );                       // f recursion depth

    h( hc );
}
```

Figure 2: Throw/Catch

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

```
_Coroutine Grammar {
  public:
    enum Status { CONT, MATCH, ERROR }; // possible status
  private:
    Status status;                      // current status of match
    char ch;                            // character passed by cocaller
    void main();                        // coroutine main
  public:
    Status next( char c ) {
        ch = c;                         // communication in
        resume();                       // activate
        return status;                  // communication out
    }
};
```

which verifies a string of characters matches the language ab^+c^*d, that is, the letter *a*, followed by one or more letter *b*s, followed by zero or more letter *c*s, followed by the letter *d*. In addition, the number of *b*s must be exactly one greater than the number of *c*s, e.g.:

valid strings	invalid strings
abd	acd
abbcd	abcd
abbbccd	ad

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:

- CONT means continue sending characters, may yet be a valid string of the language,
- MATCH means the characters form a valid string of the language and no more characters can be sent,
- ERROR means the last character resulted in a string not in the language.

After the coroutine returns a value of MATCH or ERROR, it must terminate; sending more characters to the coroutine after this point is undefined.

Write a program grammar that checks if a string is in the above language. The shell interface to the grammar program is as follows:

```
grammar [ 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 Grammar coroutine,
- pass characters from the input line to the coroutine one at time while the coroutine returns CONT,
- print an appropriate message when the coroutine returns MATCH or ERROR, or if there are no more characters to send on CONT,
- terminate the coroutine,
- print out result information, 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 line is in the language and the string no otherwise. If there are extra characters (including whitespace) on a line after the coroutine returns MATCH or ERROR, 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:

```
"abd" : "abd" yes
"abbcd" : "abbcd" yes
"abbbccd" : "abbbccd" yes
"abbbccdab" : "abbbccd" yes -- extraneous characters "ab"
"" : Warning! Blank line.
"acd" : "ac" no -- extraneous characters "d"
"abcd" : "abcd" no
"ad" : "ad" no
```

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 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. q1perf.txt – the information required by question 1, p. 1.
2. q2*.{h,cc,C,cpp} – code for question 2a, p. 2. **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, minus one aspect of its output.**
3. q2longjmp.txt – contains the information required by question 2b, p. 2.
4. q3*.{h,cc,C,cpp} – code for question 3, p. 2. **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. q3test.txt – test documentation for question question 3, p. 2, 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 g++ to compile the program for question 1, p. 1.
7. Use the following Makefile to compile the programs for question 2a, p. 2 and question 3, p. 2:

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

OBJECTS0 = q2.o                                 # optional build of given throwcatch program
EXEC0 = throwcatch                              # 0th executable name

OBJECTS1 = # object files forming 1st executable with prefix "q2"
EXEC1 = longjmp                                 # 1st executable name

OBJECTS2 = # object files forming 2nd executable with prefix "q3"
EXEC2 = grammar                                 # 2nd executable name

OBJECTS = ${OBJECTS0} ${OBJECTS1} ${OBJECTS2} # all object files
DEPENDS = ${OBJECTS:.o=.d}                      # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2}                       # all executables

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

.PHONY : all clean

all : ${EXECS}                                  # build all executables

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

${EXEC0} : ${OBJECTS0}                          # link step 0th executable
    g++ $^ -o $@

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

${EXEC2} : ${OBJECTS2}                          # link step 2nd executable
    ${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} ${EXECS}
```

This makefile is used as follows:

```
$ make longjmp
$ longjmp ...
$
$ make grammar
$ grammar ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, expand OBJECTS1 and OBJECTS2, and then type make longjmp 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!