CS 343 Fall 2011 – Assignment 1
Instructor: Peter Buhr
Due Date: Monday, September 26, 2011 at 22:00
Late Date: Wednesday, September 28, 2011 at 22:00

September 10, 2011

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. In exception handling, a *resumption handler* is an inline (nested) routine responsible for handling a raised exception. Therefore, understanding issues associated with nested routines is important. In particular, nested routines allow references to global variables in outer scopes, e.g.:

```
void h( int p ) {
    int v1 = 10;                // v1 is a global variable for g
    void g( int p ) {           // nest routine
        v1 = 5                  // global reference from g to v1
```

   However, recursion results in multiple instances of the global variable on the stack. When eventually a nested routine is called and makes its global reference, which of the multiple global instances is the reference accessing?

   The C program in Figure 1 prints a stack trace of its recursive calls. The number on the left denotes the level of the stack call-frame (there are 16 calls), then the value and address of the variables v1 and v2 visible in that frame, and finally the routine name (f, g or h) and its parameter value for the call.

   (a) i. Compile and run the C program in Figure 1.
   ii. Copy the output into a file.
   iii. In the last two columns, replace the underscores with the frame number where the referenced v1 and v2 variable is declared (allocated). For example, the top stack-frame for f(0) (frame number 1) has a reference to v2, which frame on the stack (denoted by its frame number) allocated this particular v2 variable? Hint, it must be a frame for routine g because v2 is declared at the start of g.

   (b) Define the term *lexical link* (*access/static link*) and explain how it is used to access global variables from a nested-routine's stack-frame.

2. (a) Rewrite the program in Figure 2, p. 3 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).

```
// MUST USE FILE-NAME SUFFIX ".c" AND COMPILE WITH gcc
#include <stdio.h>

int main() {
    int frameno = 1;
    void h( int p ) {                   // nest routine
        int v1 = 10;

        void g( int p ) {                   // nest routine
            int v2 = 100;

            void f( int p ) {                   // nest routine
                if ( p > 0 ) f( p - 1 );
                v2 += 1;
                printf( "%d\t%d\t%p\t%d\t%p\tf(%d)\t___\t___\n", frameno, v1, &v1, v2, &v2, p );
                frameno += 1;
            }

            if ( p % 2 == 0 ) f( p - 1 );
            v1 += 1;
            if ( p != 0 ) g( p - 1 );
            printf( "%d\t%d\t%p\t%d\t%p\tg(%d)\t___\n", frameno, v1, &v1, v2, &v2, p );
            frameno += 1;
        }

        if ( p % 2 == 0 ) g( 2 );
        if ( p != 0 ) h( p - 1 );
        printf( "%d\t%d\t%p\t\t\t\th(%d)\t\t\n", frameno, v1, &v1, p );
        frameno += 1;
    }

    printf( "frame#\tv1\t&v1\t\tv2\t&v2\t\trtn(p)\tfrm:v1\tfrm:v2\n"
            "=============================================================\n" );
    h( 3 );
}
```

Figure 1: Nested Routines

- 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.

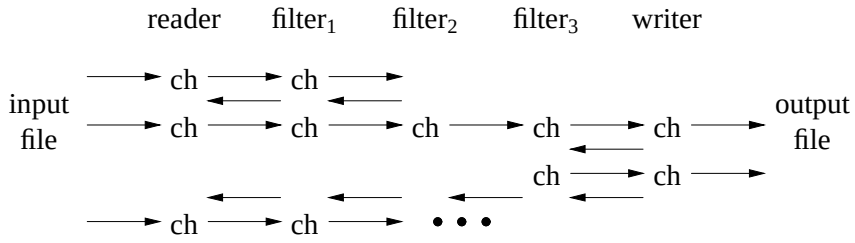
iii. Speculate as to the reason for the major performance difference.

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.

   Write a program that filters a stream of text. The filter semantics are specified by command-line options. The program creates a *semi-coroutine* filter for each command-line option joined together in a pipeline with a reader filter at the input end of the pipeline, followed by the command-line filters in the middle of the pipeline (maybe zero), and a writer filter at the output end of the pipeline. Control passes from the reader, through the filters, and to the writer, ultimately returning back to the reader. One character moves along the pipeline at any time. For example, a character starts from the reader filter, may be deleted or transformed by the command-line filters, and any character reaching the writer filter is printed.

```
#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



(In the following, you may not add, change or remove prototypes or given members; you may add a destructor and/or private and protected members.)

Each filter must inherit from the abstract class Filter:

```
_Coroutine Filter {
  protected:
    static const unsigned char End_Filter = '\377';
    unsigned char ch;
  public:
    void put( unsigned char c ) {
        ch = c;
        resume();
    }
};
```

which ensures each filter has a put routine that can be called to transfer a character along the pipeline.

The reader reads characters from the specified input file and passes these characters to the first coroutine in the filter:

```
_Coroutine Reader : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
  public:
    Reader( Filter *f, istream *i );
};
```

The reader constructor is passed the next filter object, which the reader passes one character at a time from the input stream, and an input stream object from which the reader reads characters. No coroutine calls the put routine of the reader; all other coroutines have their put routine called.

The writer is passed characters from the last coroutine in the filter pipeline and writes these characters to the specified output file:

```
_Coroutine Writer : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
  public:
    Writer( ostream *o );
};
```

The writer constructor is passed an output stream object to which this filter writes characters that have been filtered along the pipeline. No filter is passed to the writer because it is at the end of the pipeline.

All other filters have the following interface:

```
_Coroutine filter-name : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
  public:
    filter-name( Filter *f, ... );
};
```

Each filter constructor is passed the next filter object, which this filter passes one character at a time after performing its filtering action, and "..." is any additional information needed to perform the filtering action.

The pipeline is built by uMain from writer to reader, in reverse order to the data flow. Each newly created coroutine is passed to the constructor of its predecessor coroutine in the pipeline. The reader's constructor resumes

itself to begin the flow of data, and it calls the put routine of the next filter to begin moving characters through the pipeline to the writer. Normal characters, as well as control characters (e.g., '\n', '\t'), are passed through the pipeline. When the reader reaches end-of-file, it passes the sentinel character '\377' through the pipeline and then terminates. Similarly, each coroutine along the filter must pass the sentinel character through the pipeline and then terminate. The writer does not print the sentinel character. uMain ends when the reader declaration completes, implying all the input characters have been read and all the filter coroutines are terminated. The reader coroutine can read characters one at a time or in groups; the writer coroutine can write characters one at a time or in groups.

Filter options are passed to the program via command line arguments. For each filter option, create the appropriate coroutine and connect it into the pipeline. If no filter options are specified, then the output should simply be an echo of the input from the reader to the writer. *Assume all filter options are correctly specified, i.e., no error checking is required on the filter options.*

The filter options that must be supported are:

-h The *hex dump* option replaces each character in the stream with its corresponding ASCII 2-hexadecimal digit value. For example, the character 'a' is transformed into the two characters '6' and '1', as 61 is the ASCII hexadecimal value for character 'a'. The hexadecimal filter also formats its output by transforming two characters into 4 hexadecimal digits, adding one space, transforming two characters into 4 hexadecimal digits, adding three spaces, repeating this sequence four times, and adding a newline. For example, this input sequence:

```
The quick brown fox jumps over the lazy dog.
```

generates the following:

```
5468 6520   7175 6963   6b20 6272   6f77  6e20
666f 7820   6a75 6d70   7320 6f76   6572 2074
6865 206c   617a 7920   646f 672e   0a
```

There are two spaces at the end of each line. Note, it is possible to convert a character to its hexadecimal value using a simple, short expression.

-s The *capitalize* option capitalizes the first letter of every sentence. A sentence starts with the letter following whitespace (space, tab, newline) characters after a period, question mark, or exclamation point. If the starting letter is lower case, the filter transforms the letter to upper case. There is a special case for capitalizing the first letter in the pipeline as there is no preceding punctuation character.

-w The *whitespace* option removes all spaces and tabs from the start and end of lines, and collapses multiple spaces and tabs within a line into a single space.

-T *base-width* The *triangle* option arranges the characters in the input stream into a *triangle* shape. (Assume a space separates -T and *base-width*.) The triangle is isosceles with a base width of *base-width* characters. The base width must be odd; increment an even value so it is odd. For example, this input sequence:

```
The quick brown fox jumps over the lazy dog.
```

and a base width of 9, generates the following:

```
    T
   he
  quick
  brown
fox jumps

   ove
  r the
  lazy d
og.
```

Progressively fewer spaces are added before characters to form a triangle until a row of 9 characters (base-width) is reached and then a newline is added. There are no spaces added at the end of each line. Tab

and newline input characters are changed to a single space. The triangles are repeated until the input ends, forming a connected chain.

In addition, design and implement one other filter operation that might be useful. Fully document the new filter in the code; i.e., what is does, how to use it, and an example of its usage.

The order in which filter options appear on the command line is significant, i.e., the left-to-right order of the filter options on the command line is the first-to-last order of the filter coroutines. As well, a filter option may appear more than once in a command. Each filter should be constructed without regard to what any other filters do, i.e., there is no communication among filters using global variables; all information is passed using member put. **Hint:** scan the command line left-to-right to locate and remember the position of each option, and then scan the option-position information right-to-left (reverse order) to create the filters with their specified arguments.

The executable program is to be named filter and has the following shell interface:

filter [*-filter-options ...*] [infile [outfile]]

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If filter options appear, assume they appear *before* the file names. If no input file name is specified, input from standard input and output to standard output. If no output file name is specified, output to standard output. If an input or output file is specified, check that it can be opened.

WARNING: In general, there should be no execution "state" variables and **switch** statements that use them in your program. 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. q1nestedroutine.txt – contains the information required by question 1, p. 1.
2. q2*.{h,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, minus one aspect of its output.**
3. q2longjmp.txt – contains the information required by question 2b, p. 1.
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. q3filter.testtxt – 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 the following Makefile to compile the programs for question 2a, p. 1 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 = q2throwcatch.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 = filter                                  # 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 filter
$ filter ...
```

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