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, transform the C++ program in Figure 1 replacing throw/catch with:
   i. C++ program using global status-flag variables. The return type of routines may NOT be changed to have return codes.
   ii. C++ program using a C++17 variant return-type as return codes. It is possible to use inheritance to simplify the solution.
   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 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 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.
```cpp
#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 ) {
                seed = stoi( argv[3] );
                if ( seed <= 0 ) throw 1;
            } // if
            case 3: if ( strcmp( argv[2], "d" ) != 0 ) {
                eperiod = stoi( argv[2] );
                if ( eperiod <= 0 ) throw 1;
            } // if
            case 2: if ( strcmp( argv[1], "d" ) != 0 ) {
                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 );
    } 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
```cpp
#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 {
        switch( argc ) {
        case 5: if ( strcmp( argv[4], "d" ) != 0 ) { eperiod = stoi( argv[4] ); if ( eperiod <= 0 ) throw 1; } // if
        case 4: if ( strcmp( argv[3], "d" ) != 0 ) { seed = stoi( argv[3] ); if ( seed <= 0 ) throw 1; } // if
        case 3: if ( strcmp( argv[2], "d" ) != 0 ) { n = stoi( argv[2] ); if ( n < 0 ) throw 1; } // if
        case 2: if ( strcmp( argv[1], "d" ) != 0 ) { m = stoi( argv[1] ); if ( m < 0 ) throw 1; } // if
        case 1: break; // use all defaults
        default: throw 1;
        } // switch
    } catch( ... ) {
        cerr << "Usage: " << arg[0] << " [ m (= 0) | d [ n (= 0) | d
        " [ seed (> 0) | d [ eperiod (> 0) | d ] ] ] ]" << endl;
        exit( EXIT_FAILURE );
    } // try
    srand( seed ); // seed random number
    try {
        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;
} // main

Figure 2: Throw/Catch
```
No dynamic allocation is allowed, but creation of a global variable is allowed. No more calls to `setjmp` than the number of `try ... catch` 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 `-DNOWERROR` to suppress output.
- Time the executions using the `time` command:

  ```
  $ /usr/bin/time -f "%Uu %Su %Es %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 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 Printf {
    char ch; // character passed by cocaller
    // 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; // communicate input
        resume(); // activate
    }
};
```

which verifies a string of characters corresponds to a valid printf format described by the following grammar:

```
format : '%%' flags opt width opt precision opt length opt specifier
flags : '-' | '+' | 'L' | '#' | '0'
width : digits+ | '***'
precision : '.'(digits+ | '***')
length : '[h]' | '[hh]' | '[l]' | '[11]' | '[j]' | '[z]' | '[c]' | '[L]
specifier : 'd' | '[i]' | '[u]' | '[o]' | '[x]' | '[X]' | '[f]' | '[F]' | '[e]' | '[E]' | '[g]' | '[G]' | '[a]' | '[A]' | '[c]' | '[s]' | '[p]' | '[n]' | '[s]'
digits : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```
where the parentheses (sequence), plus (1 or more), bar (alternative), _opt_ (0 or 1), quotation marks (literals), and '⊔' (space) are metasymbols and not part of the described language. Also, there are _restrictions_ on combinations of format fields, e.g., flags, width, precision and length, if present, must be compatible with the format specifier. (Hint: printf prints warnings for an invalid format.) Because the specifier is _after_ the fields in the format, some state must be retained (flag variables) to check for invalid combinations.

The following are examples of valid and invalid printf formats:

<table>
<thead>
<tr>
<th>valid</th>
<th>invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>%3.</td>
</tr>
<tr>
<td>%-d</td>
<td>%3.h</td>
</tr>
<tr>
<td>%-3d</td>
<td>%1d</td>
</tr>
<tr>
<td>%-03d</td>
<td>%hhf</td>
</tr>
<tr>
<td>%.3d</td>
<td>%zA</td>
</tr>
<tr>
<td>%3.3d</td>
<td>%wd</td>
</tr>
</tbody>
</table>

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 _printf_ that checks if strings are valid formats. The shell interface to the _printf_ program is as follows:

```
printf [ 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 main program should:

- read a line from the file,
- create a _Printf_ coroutine,
- pass characters from the input line to the coroutine one at a 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 input 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 ' 
'. **There is no printing in the coroutine.**

The following is some example output:
'\%d' : '%d' yes  
'%-d' : '%-d' yes  
'%-3d' : '%-3d' yes  
'%-03d' : '%-03d' yes  
'%-3.3d' : '%-3.3d' yes  
'' : Warning! Blank line.  
'%.3.' : '%.3.' no  
'%.3.h' : '%.3.h' no  
'%.3d' : '%.3d' no  
'%.3hf' : '%.3hf' no  
'%.3A' : '%.3A' no  
'%.wd' : '%.wd' no, extraneous characters 'd'

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) to valid digit characters.

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

Please follow these guidelines very carefully. Review the Assignment Guidelines and C++ Coding Guidelines before starting each assignment. Each text or document file, e.g., *.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. 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. q3printf.{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. q3printf.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 = q3printf.o q3main.o  # 5th executable object files
EXEC5 = printf  # 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)
g++ -8 $(CXXFLAGS) ^ -o @$1

q1%.o : q1%.cc
  g++ -8 $(CXXFLAGS) -std=c++17 -c $< -o @$1

$(EXEC1) : $(OBJECTS1)
g++ -8 $(CXXFLAGS) ^ -o @$1

$(EXEC2) : $(OBJECTS2)
g++ -8 $(CXXFLAGS) ^ -o @$1

q1%.o : q1%.c
gcc -8 $(CXXFLAGS) -c $< -o @$1

$(EXEC02) : $(OBJECTS02)
g++ -8 $(CXXFLAGS) ^ -o @$1

$(EXEC3) : $(OBJECTS3)
g++ -8 $(CXXFLAGS) ^ -o @$1

q2%.o : q2%.cc
  g++ -8 $(CXXFLAGS) -c $< -o @$1

$(EXEC4) : $(OBJECTS4)
g++ -8 $(CXXFLAGS) ^ -o @$1

$(EXEC5) : $(OBJECTS5)
$(CXX) $(CXXFLAGS) ^ -o @$1
#(OBJECTS) : #(MAKEFILE_NAME)  # OPTIONAL : changes to this file => recompile

-include $(DEPENDS)  # include *.d files containing program dependences

clean :
    rm -f *.d *o $(EXEC01) $(EXEC02) $(EXEC03)

This makefile is used as follows:
$ make returnglobal
$ returnglobal ...
$ make returntype
$ returntype ...
$ make returntypec
$ returntypec ...
$ make longjmp
$ longjmp ...
$ make printf
$ printf ...

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 printf 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!