CS 343 Winter 2024 – Assignment 1
Instructor: Colby Parsons
Due Date: Monday, January 22, 2024 at 22:00
Late Date: Wednesday, January 24, 2024 at 22:00

January 18, 2024

This assignment is designed for you to practice your knowledge and understanding of advanced control flow. It also 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 receives little or no marks. (You may freely use the code from these example programs.)

1. (a) Transform the C++ program in Figure 1 replacing the **throw/catch** for exceptions Ex1, Ex2, and Ex3 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 exceptions by value or pointer (using inheritance) in the variant return-type.
   iii. C program using a tagged **union** return-type as return codes, plus replace the exception handling used to handle the command-line arguments with multi-level exits.

   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 exception 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 exception period decreases, and the reason for the difference.

2. (a) Transform the C++ program in Figure 2, p. 3 by replacing *only* the **throw/catch** for exceptions E with longjmp/setjmp. No changes are allowed to the return type or parameters of routine Ackermann. No dynamic allocation is allowed, but creation of a global variable is allowed. No more calls to setjmp are allowed 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**.

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

struct Ex1 { short int code; };
struct Ex2 { int code; };
struct Ex3 { long int code; };

intmax_t eperiod = 10000;                               // exception period
int randcnt = 0;
int Rand() { randcnt += 1; return rand(); }

double rtn1( double i ) {
  if ( Rand() % eperiod == 0 ) { throw Ex1{ (short int)Rand() }; } // replace
    return i + Rand();
}
double rtn2( double i ) {
  if ( Rand() % eperiod == 0 ) { throw Ex2{ Rand() }; }  // replace
    return rtn1( i ) + Rand();
}
double rtn3( double i ) {
  if ( Rand() % eperiod == 0 ) { throw Ex3{ Rand() }; }  // replace
    return rtn2( i ) + Rand();
}

static intmax_t convert( const char * str ); // copy from https://student.cs.uwaterloo.ca/~cs343/examples/convert.h

int main( int argc, char * argv[] ) {
    intmax_t times = 100000000, seed = getpid();        // default values
    struct cmd_error {};
    try {
        switch ( argc ) {
          case 4: if ( strcmp( argv[3], "d" ) != 0 ) {  // default ?
            seed = convert( argv[3] ); if ( seed <= 0 ) throw cmd_error(); }
          case 3: if ( strcmp( argv[2], "d" ) != 0 ) {  // default ?
            eperiod = convert( argv[2] ); if ( eperiod <= 0 ) throw cmd_error(); }
          case 2: if ( strcmp( argv[1], "d" ) != 0 ) {  // default ?
            times = convert( argv[1] ); if ( times <= 0 ) throw cmd_error(); }
          case 1: break;                                // use all defaults
          default: throw cmd_error();
        } // switch
    } catch( ... ) {
        cerr << "Usage: " << argv[0] << " [ times > 0 | d [ eperiod > 0 | d [ seed > 0 | d ] ] ]" << 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; }               // replace
        // analyse exception
        catch( Ex1 ev ) { ev1 += ev.code; ec1 += 1; }   // replace
        catch( Ex2 ev ) { ev2 += ev.code; ec2 += 1; }   // replace
        catch( Ex3 ev ) { ev3 += ev.code; ec3 += 1; }   // replace
    } // for
    cout << "randcnt " << randcnt << endl;
    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 PRINT( stmt )
#else
#define PRINT( stmt ) stmt
#endif // NOOUTPUT
struct E {};                                            // exception type
intmax_t eperiod = 100, excepts = 0, calls = 0, dtors = 0, depth = 0; // counters
PRINT( struct T { ~T() { dtors += 1; } }; )
long int Ackermann( long int m, long int n, long int depth ) {
    calls += 1;
    if ( m == 0 ) {
        if ( rand() % eperiod <= 2 ) { PRINT( T t; ) excepts += 1; throw E(); } // replace
        return n + 1;
    } else if ( n == 0 ) {
        try { return Ackermann( m - 1, 1, depth + 1 );  // replace
        } catch( E ) {                                  // replace
            PRINT( cout << " depth " << depth << " E1 " << m << " " << n << " |" );
            if ( rand() % eperiod <= 3 ) { PRINT( T t; ) excepts += 1; throw E(); } // replace
        } // try
        PRINT( cout << " E1X " << m << " " << n << endl );
    } else {
        try { return Ackermann( m - 1, Ackermann( m, n - 1, depth + 1 ), depth + 1 ); // replace
        } catch( E ) {                                  // replace
            PRINT( cout << " depth " << depth << " E2 " << m << " " << n << " |" );
            if ( rand() % eperiod == 0 ) { PRINT( T t; ) excepts += 1; throw E(); } // replace
        } // try
        PRINT( cout << " E2X " << m << " " << n << endl );
    } // if
    return 0;                                           // recover by returning 0
}
static intmax_t convert( const char * str ); // copy from https://student.cs.uwaterloo.ca/~cs343/examples/convert.h
int main( int argc, char * argv[] ) {
    volatile intmax_t m = 4, n = 6, seed = getpid();    // default values (volatile needed for longjmp)
    struct cmd_error {};                                // command-line errors
    try {                                               // process command-line arguments
        switch ( argc ) {
          case 5: if ( strcmp( argv[4], "d" ) != 0 ) {  // default ?
            eperiod = convert( argv[4] ); if ( eperiod <= 0 ) throw cmd_error(); }
          case 4: if ( strcmp( argv[3], "d" ) != 0 ) {  // default ?
            seed = convert( argv[3] ); if ( seed <= 0 ) throw cmd_error(); }
          case 3: if ( strcmp( argv[2], "d" ) != 0 ) {  // default ?
            n = convert( argv[2] ); if ( n < 0 ) throw cmd_error(); }
          case 2: if ( strcmp( argv[1], "d" ) != 0 ) {  // default ?
            m = convert( argv[1] ); if ( m < 0 ) throw cmd_error(); }
          case 1: break;                                // use all defaults
          default: throw cmd_error();
        } // 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 {                                               // replace
        PRINT( cout << "Arguments " << m << " " << n << " " << seed << " " << eperiod << endl );
        long int val = Ackermann( m, n, 0 );
        PRINT( cout << "Ackermann " << val << endl );
    } catch( E ) {                                      // replace
        PRINT( cout << "E3" << endl );
    } // try
    cout << "calls " << calls << " exceptions " << excepts << " destructors " << dtors << endl;
}
```

Figure 2: Throw/Catch

(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 12 12 103 28
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 command-line parameters 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 4 timing results to validate the experiments.

iii. State the performance difference (larger/smaller/by how much) between the original and transformed programs, and the reason for 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, which is available on the undergraduate computing environment.

Brainfun is a programming language that was made with the goal of requiring the smallest compiler possible. It is Turing-complete, yet nearly unusable since it consists of only the following 6 characters: '<' '>' '+' '-' '.' ',' '[' ']', which each denote an instruction. The language is based on a Turing Machine, a computing model that consists of a memory tape split into cells (similar to an array), and a pointer into the tape. The commands in Brainfun move the pointer up and down the tape, increment or decrement values in a tape cell, or branch based on the value of the current cell.

Character	Meaning
>	Increment the data pointer by one (to point to the next cell to the right).
<	Decrement the data pointer by one (to point to the next cell to the left).
+	Increment the byte at the data pointer by one.
−	Decrement the byte at the data pointer by one.
.	Output the byte at the data pointer.
,	Accept one byte of input, storing its value in the byte at the data pointer.
[	If the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.
]	If the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

Write a program brainfun that transpiles a given Brainfun program and checks its validity Transpilation is a process where source code is taken from one language and transformed and output as another language. The shell interface to the brainfun program is as follows:

```
brainfun [ infile [ outfile ] ]
```

(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. If no output file name is specified, 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.* There is an example μC++ programs demonstrating how to handling command-line parameters.

Some valid Brainfun programs	Some invalid Brainfun programs
[ ]	[
[[[ ]]]	]
[[ ][ ]][ ]	[ ][[ ]

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

```
_Coroutine Brainfun {
    std::ostream * out;          // output stream
    char ch;                     // character passed by cocaller

    // YOU ADD MEMBERS HERE
    void main();                 // coroutine main

  public:
    _Exception Error {};         // thrown if program is invalid
    _Exception EndOfFile {};     // thrown at coroutine to indicate EOF

    void next( char c ) {
        ch = c;                  // communication input
        resume();                // activate
    } // Brainfun::next
};
```

which outputs the corresponding C code for a Brainfun program as it is passed character by character.

A Brainfun program can consist of any characters, including those not listed in the language. All characters should be passed to the coroutine, which will ignore characters not in the language, and outputs the corrsponding C code for characters in the language. Your Brainfun coroutine should print the corresponding C code for each Brainfun instruction before suspending back to main. In other words, each input character that is in the Brainfun language should have its corresponding C code output printed before the next character is read and passed to the coroutine. When a character not in the Brainfun language is passed to the coroutine it does not print anything.

Your output C program will have the following format:

```
#include <stdio.h>
#include <string.h>

int main() {
    int pc = 0;                          // data pointer
    char cells[30000];                   // turing machine tape
    memset( cells, 0, sizeof(cells) );

    // Your output C code from the coroutine goes here!

    return 0;
}
```

Your Brainfun coroutine must perform one other task. It must ensure that the transpiled program is *valid*, which in this case means that it has *matching opening and closing square brackets*. Other than brackets, there is no other criterion for validity of programs that you should check. When your coroutine detects that a transpiled program is invalid it should resume the Error exception at the program main, which will print "Invalid Brainfun Program!".

You will likely find most of the Brainfun instructions trivial to convert, with the exception of the bracket instructions which require some nuance. We suggest simulating the bracket instructions using if statements and goto statements in your output C code. To get you started, for the Brainfun instruction '>', your coroutine should output the C code "pc++;". This transpiler coroutine can be implemented with very little state, and does not require any dynamic allocation or storing of strings. *Any coroutine implementations that store strings or use dynamic allocation will have marks deducted, as per the general mark deductions.* That being said, it may be a

good starting point to make a working solution using dynamic allocation or strings and then work from there to find the "zen" of the coroutine.

The program main should:

- print the provided boilerplate leading lines of C output
- create a Brainfun coroutine,
- read in input from a file or input stream character by character,
- pass characters from the input to the coroutine one at time.
- once all input has been read, resume the EndOfFile exception at the coroutine
- if an Error exception is thrown by the coroutine, handle it and print "Invalid Brainfun Program!"
- terminate the coroutine
- print the provided boilerplate closing lines of C output

Your program will be tested by passing it a Brainfun program. If the Brainfun program is invalid, your output will be searched for a line containing *solely* "Invalid Brainfun Program!" (not including the quotes). No whitespace or any other characters should share a line in the output with the line "Invalid Brainfun Program!" if it is printed. Other than "Invalid Brainfun Program!", the testing script does not care what is output when an invalid Brainfun program is transpiled. If the Brainfun program is valid, your output C code will be compiled with gcc with no other arguments into a binary. That binary will be run and checked for correctness by ensuring it's output matched the expected output from the original Brainfun program. Given a valid brainfun program

helloworld.bf, and its expected output helloworld.txt, your code can be tested as follows:

```
./brainfun program.bf out.c
gcc out.c
diff program.txt <(./a.out) # should output nothing if correct
```

Given an invalid brainfun program invalid.bf, and the expected output for invalid programs, invalid.txt, your code can be tested as follows:

```
./brainfun invalid.bf out.c
diff invalid.txt <(grep "Invalid Brainfun Program!" out.c) # should output nothing if correct
```

The programs helloworld.bf, brainfun.bf and the .txt files used in the example above can be found on the assignments webpage along with a sample executable of helloworld.

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,testdoc} file, must be ASCII text and not exceed 500 lines in length, using the command fold –w120 *.testdoc | 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 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.**

```
[ This program prints "Hello World!" and a newline to the screen, its
length is 106 active command characters. [It is not the shortest.]
This loop is an "initial comment loop", a simple way of adding a comment
to a BF program such that you don't have to worry about any command
characters. Any ".", ",", "+", "-", "<" and ">" characters are simply
ignored, the "[" and "]" characters just have to be balanced. This
loop and the commands it contains are ignored because the current cell
defaults to a value of 0; the 0 value causes this loop to be skipped.
]
++++++++               Set Cell #0 to 8   [
>++++                  Add 4 to Cell #1; this will always set Cell #1 to 4
[                      as the cell will be cleared by the loop
>++                    Add 2 to Cell #2
>+++                   Add 3 to Cell #3
>+++                   Add 3 to Cell #4
>+                     Add 1 to Cell #5
<<<<-                  Decrement the loop counter in Cell #1
]                      Loop until Cell #1 is zero; number of iterations is 4
>+                     Add 1 to Cell #2
>+                     Add 1 to Cell #3
>-                     Subtract 1 from Cell #4
>>+                    Add 1 to Cell #6
[<]                    Move back to the first zero cell you find; this will
be Cell #1 which was cleared by the previous loop
<-                     Decrement the loop Counter in Cell #0
]                      Loop until Cell #0 is zero; number of iterations is 8

The result of this is:
Cell no :   0   1   2   3   4   5   6
Contents:   0   0  72 104  88  32   8
Pointer :   ^

>>.                    Cell #2 has value 72 which is 'H'
>---.                  Subtract 3 from Cell #3 to get 101 which is 'e'
+++++++..+++.          Likewise for 'llo' from Cell #3
>>.                    Cell #5 is 32 for the space
<-.                    Subtract 1 from Cell #4 for 87 to give a 'W'
<.                     Cell #3 was set to 'o' from the end of 'Hello'
+++.------.--------.    Cell #3 for 'rl' and 'd'
>>+.                   Add 1 to Cell #5 gives us an exclamation point
>++.                   And finally a newline from Cell #6
```

Figure 3: Brainfun Hello World Example (Taken from the Brainfun Wikipedia page)

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 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. q3*.{h,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 from the transpiled programs this question is checked via a marking program, so it must match exactly with the original Brainfun program.**

6. 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.**

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.

```
OUTPUT = OUTPUT
GVERSION = -12
CXX = u++                                   # uC++ compiler
#CXX = /u/cs343/cfa-cc/bin/cfa              # CFA compiler
CXXFLAGS = -g -Wall -Wextra -MMD -Wno-implicit-fallthrough -D${OUTPUT} # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}}# makefile name

.SUFFIXES : .cfa                            # CFA default rules
.cfa.o :
    ${CXX} ${CXXFLAGS} -c $<

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 = # object files forming 5th executable with prefix "q3"
EXEC5 = brainfun                                # 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} : ${OBJECTS01}                        # optional build of given program
    g++${GVERSION} ${CXXFLAGS} $^ -o $@

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

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

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

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

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

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

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

${EXEC4} : ${OBJECTS4}                          # compile and link 4th executable
    g++${GVERSION} ${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 brainfun
$ ./brainfun ...
```

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 brainfun 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!