**

**ECE 4960: Computational and Software Engineering**

**Spring 2018**

**Programming Assignment 1: Arithmetic Exception Handling**

**1. Goal**

The program assignment will be the implementation of a utility program that can observe and validate the exception handling of the integer and floating-point arithmetics on your given platform. This will include representations of integers and floating points. You will implement computational schemes to observe the exception handling rules, but are not limited to them. The code should capture exception occurrence, along with the lack of any exception detection and report them as well. For the software engineering part, we will practice the first step of modular programming, i.e., breaking a large program into functional pieces for better sharing and debugging, and eventually leading to robust code reuse. Each independent exception check should be implemented as a function. You can choose any general-purpose programming language and platform for this practice. Notice that not all platforms abide by the IEEE standards, even though they claimed so.

**2. Observation of exception rules**

* Integer overflows: Use faster approach methods such as multiplication, factorials or Fibonossi numbers to cause integer overflow.
* Integer divided by 0: Direct division by 0 will often be a compiler or editor error. Use a variable to create such exception.
* Floating-point overflows: Estimate the number of iteration needed for this to happen beforehand. Try out several check criteria for overflows.
* Floating-point operations of INF and NINF: Generate INF and NINF and observe their behavior in functions such as *1/x*, sin(*x*), and exp(*x*). Also test their propagation and interaction.
* Floating-point operations of NaN: Generation, detection, propagation and interaction with other exception cases.
* Signed zero: Observe signed zero handling by performing , ,, and operations.



* Floating point gradual underflow: Perform (*x – y*), (*x/y*) and with careful selection of *x* and *y*, to observe floating point gradual underflow.



**3. Additional requirements**

* Program input: none required
* Program output: at least two levels of reporting, one for an IEEE compliance report with only the violation to the default program output (< 10 lines), and the other for detailed log of the behavior for every rule in a file (< 4 page). Both outputs should be concise and easily readable to other programmers and should be generated by the code.
* Find at least two platforms (language or OS) to run the same tests. Compare the results in a short report.
* Preferably you will have two students in a group so you have easy access to two platforms.

**4. Applications of floating-point precision**

After understanding how floating-point precision is handled, you will choose one of the following two tasks to demonstrate your understanding (the second one is a bit more difficult). It should be implemented as an additional function and reported with the previous results.

1. Calculate and report π with 30 digits of precision. For approximation, see the web page at: https://en.wikipedia.org/wiki/Approximations\_of\_π.
2. Implement a quad-precision addition/subtraction with two input double-precision floating numbers. The output will be two double-precision floating-point numbers whose sum can represent the quad precision. For example, to represent the quad precision 1.2345678901234567890123456789012e10, the two double-precision output should be: 1.23456789012345e10 and 6.7890123456789012e-5. You should implement all exception rules except for soft landing. Propose tests for your function such as if the small number can be retrieved after adding and then subtracting a number that is about 1015 times larger.

**5.** **Software engineering practice**

* Useful reading: Chap. 21, Jumping into C++, pp. 239 – 251.
* Understanding the “build” steps in your IDE for C++ (just as an example):
  + **Preprocessing**: substituting and expanding texts for compilers: resolving macros and IFDEF; definition of constant and identifiers
  + **Compiling**: turning a source code (.cpp) to object files (.o). Each source code with its #include files is compiled separately to machine language instructions. Each object file cannot be executed yet because the functions may be defined in other object (.o) and library files (.dll).
  + **Linking**: creating a single executable file (.exe) out of all related object (.o) and library (.dll) files. All names of variables and functions and all definitions of functions and classes have to be resolved at this stage.
  + In the generic Unix environment, synchronization of .h, .cpp, .dll, .o and .exe files is handled by “makefiles”, which defines the dependence of these files. The synchronization function is mostly absorbed in the “build” function of your IDE.
  + Separation of compiling and linking is meant for large program development, as well as to separate various functions to different files. Interpretive languages such as python has one step of compilation and linking, and are often meant for programmer efficiency for smaller programs or for scripting user interface.
* Split program across multiple files:
  + Three main purposes for splitting: (1) Smaller files can be “checked out” by different programmers for development (this will be clear when we introduce Git in the next assignement); (2) Parts of the program that are not touched do not need to be recompiled or debugged; (3) Well-encapsulated and well-design modules are fundamental to code reuse, which does not only improve coding efficiency in the long term but also improve robustness as the module has been tested.
  + We will first practice separation of header files (for functions and global variables) here. Notice that C++ variables always have an implied “scope”. This feature is so critical in the debugging stage as it limits the possible range of codes for a perceived error. Global variables should be reserved only for access of all functions, global constants, or minimal memory usage purposes.
  + In this assignment, if your usual way is to have one large program as indicated below, separate at least the variable declaration in the .h header file (for example, global constants and function names) and the shared functions (for example, handling of exception outputs). This will seem to be an overkill for the small tasks at hand, but it is meaningful to go through the exercise.
* Naming convention and comments: we will cover this in later sections. Use meaningful names but not too long. Conventional C programs use names like: globalName or glbNme.

Shared variable declaration

Non-.dll function declaration

Function specific declaration

Shared implementation

main functions

Shared variable declaration

Non-.dll function declaration

Shared implementation

Function implementation

#include “head.h”

main();

Shared implementation

Function implementation

**main.cpp**

**main.cpp**

**head.h**

**shared1.cpp**

**shared2.cpp**

**function1.cpp**

**function2.cpp**

**Fig. 1.** The first step of modular programming.

Two useful shared functions will be given here for examples only. This may be platform dependent.

bool IsNumber(double x)

{

// This looks like always be true,

// but false is returned if x is NaN

return (x == x);

}

bool IsPositiveFiniteNumber(double x)

{

// Separate handling of INF and NINF

// DBL\_MAX is defined in float.h or math.h

return (x >0 && x <= DBL\_MAX);

}

**6. Due dates and grading**

* You should plan to finish your programming (or at least a major part) before 2/14. Your group will give a short demonstration during the lab time on 2/14. No slide is needed. You will connect your computer to the projector and explain how your program is designed and tested.
* Commit your .cpp, .h and output files to your Git hub before 2/16 5pm. This will be used for your assignment grading. Notice that you should use the Git address that you have sent to the TA in the beginning of the semester.

**4. Further information for exploration**

Exception in C++ can further include other unexpected circumstance that the executable cannot handle such as memory violation (ex. pointer giving a negative address or stack overflow). The exception handling in C++ propagates up the functional stack automatically, so the intermediate functions do not need to accommodate exception handling explicitly. This is done by the “throw” and “catch” statements. For more details, see: <http://www.cprogramming.com/tutorial/exceptions.html>, or the tutorial at: <http://www.dev-hq.net/c++/22--try-and-catch>.

When a floating-point operation raises a floating-point exception, the status of the floating-point environment changes (ex. the environmental variable SIGFPE changes from 0), which can be tested with [std::fetestexcept](http://en.cppreference.com/w/cpp/numeric/fenv/fetestexcept" \o "cpp/numeric/fenv/fetestexcept), but the execution of a C++ program on most implementations continues uninterrupted.

There are compiler extensions that may be used to generate C++ exceptions automatically whenever a floating-point exception is raised (not needed in your assignment this time):

* GNU libc function [feenableexcept()](http://www.gnu.org/s/hello/manual/libc/Control-Functions.html) enables trapping of the floating-point exceptions, which generates the signal SIGFPE. If the compiler option -fnon-call-exceptions was used, the handler for that signal may throw a user-defined C++ exception.
* MSVC function [\_control87()](http://msdn.microsoft.com/en-us/library/e9b52ceh.aspx) enables trapping of the floating-point exceptions, which generates a hardware exception, which can be converted to C++ exceptions with [\_set\_se\_translator](http://msdn.microsoft.com/en-us/library/5z4bw5h5.aspx)