**CS 170, Spring Semester 2022, Assignment 1**

Due Friday March 11 by 5:00 pm. Course weighting 15%

File name: AS1\_98S1.DOC

Last revised: Friday, 25 February 2022 at 09:31 AM

This assignment requires you to solve three programming problems, and to implement your solution in C++. You will be assessed by your final delivery. This is an individual assignment. **No collaboration is permitted.**

Problems :

1. Write a program to calculate the number of days between two dates. Each date is represented by three integers: *day*, *month*, and *year*. Validate the user input and if it is invalid, ask the user to enter again. If user enters three times an invalid day or month, announce that the program will exit and exit the program. Use the following rules for determining leap years: A leap yearis one whose number is exactly divisible by four. Century years, however, are only leap years if they are exactly divisible by 400. Hence 1900 was not a leap year but 2000 was. Your program should not use arrays or vectors.

[max 30 points]

1. The game of Nim. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one, but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble looses.

Write a program in which the computer plays against a human opponent. The computer can play in two modes: smart or stupid. In stupid mode, the computer simply takes a random legal value (between 1 and half of the marbles) from the pile whenever it has a turn. In smart mode the computer takes off enough marbles to make the size of the pile a power of two minus 1 (that is 3, 7, 15, 63, etc.). That is always a legal move, except if the size of the pile is currently one less than a power of 2. In that case, the computer makes a random legal move. (Note that the computer cannot be beaten in smart mode when it has the first move unless the pile size happens to be a power of 2 minus 1. Of course, a human player who has the first turn and knows the winning strategy can win against the computer.)

Minimum requirements: ask the user to enter the initial size of the pile between 10 and 100, to decide who has the first turn, and to decide whether the computer plays in stupid or smart mode. Then play the game accordingly.

[max 35 points]

1. Credit card number check. The last digit of a credit card number is the *check digit*, which protects again transcription errors such as an error in a single digit or switching two digits. The following method is used to verify actual credit card numbers, but for simplicity we will describe it for 8 digits instead of 16:

* Starting from the rightmost digit, calculate the sum of every other digit. For example, if the credit card number is 43589795, then you form the sum 5 + 7 + 8 + 3 = 23.
* Double all the digits that were not included in the preceding step. For example, with the number give above you double the digits 9, 9, 5, and 4, which yield 18, 18, 10, and 8. Adding all the digits in these numbers gives 1+8+1+8+1+0+8 = 27.
* Add the sums of the two preceding steps. If the last digit of the result is 0, the number is valid. In our example, 23 + 27 = 50, so the number is valid.

Write a program that implements this algorithm. Do not use arrays or vectors. Do not use a string representation of numbers. The user should supply a number (not individual digits, but a single number with any number of digits that you can store in a variable of type long), and you should print out whether the number is valid or not. Suggested extension: also print the value of the check digit that would make the number valid.

[max 35 points]

**Development Requirements**

1. **Program organization**. Coding must use C++ streams for input and output and generate an executable file. Do not use goto statements.
2. **Dependencies**. You are encouraged to use global constants, but your program must not declare any global variables.
3. **Standards**. Your programs must meet the attached programming standards.

Delivery

All source files (only the source files – not the whole .NET projects!!) must be placed in the directory X:\Dropoff\CS\ganchevg\CS170\Assignment1, in a subdirectory with a name starting with your **last** name. You should name the source files containing your main programs PROBLEM1.CPP, PROBLEM2.CPP, and PROBLEM3.CPP. If for any of the problems these files are missing, incorrectly named, in the wrong directory, or the program will not compile and link correctly, I will not be able to grade your work for this problem. Double-check this. You should also submit in a subfolder Documentation of your submission folder:

1. Grading guide (supplied at the end of this document) with **sections 1, 2 and 3 completed** to show what you have done.
2. For each problem:

* A structure chart showing the structure of the program.
* Optionally: pseudocode or flowchart or structure diagram showing the logic of your solution.

1. A test plan as described in the attached Programming Standards, containing*:*
2. Chekpoints: Yes/No questions describing the program behaviour that does not depend on the input data
3. Test data and results in a table with three columns:
   * test input
   * expected results
   * the actual results of your testing written in by hand
4. A brief analysis of any errors, which the program still produces or a clear statement that the program does not have any known errors.
5. Optionally: sample copies of the output produced by your program.

Grading Schedule

The assignment will be graded on an A to F scale of grades.

Work which barely meets the minimum requirements and either has problems with usability or reliability or does not meet the programming standards will be graded D to C. Work which shows a useable solution with all the minimum requirements met and meets the programming standards will be graded C+ to B+. Work which in addition demonstrates initiative in design and implementation as evidenced by superior user interaction, additional functionality, robustness and reliability will be graded A- to A. Pay special attention to the good breakdown of your program to functions, and thorough validation of input data. All grading criteria are listed in the attached Grading Guide.

Grading Notes:

1. To get credit for a program feature, it must be coded, tested and documented correctly according to the given standards and be working in all respects.
2. A feature that is either not shown on the test plan as tested or does not work correctly will be given no credit.
3. Penalty: points will be deducted for a faulty feature that is shown as working and for not satisfied non-functional requirements (development requirements, delivery requirements, programming standards).

Programming Standards

In the commercial environment, programmers work to the standards of the organization that employs them. Standards usually cover proposals, contracts, analysis and design documents, testing, program documentation, user's guides and project management. In this course we require standards in these specific areas: function interface design, program organization, test plans, style of user interaction and internal program documentation.

Function Interface Design

Each function performs a single task and has a clearly defined interface to users or callers. There is no use of global variables.

Program Organization

A simple program comprises function prototypes, global declarations of constants, main program and function definitions. There is no use of goto statements.

Test Plans

A test plan is prepared at the design stage to ensure that a program meets every requirement of its specification. It identifies every requirement in the specification and specifies the necessary test or tests to ensure the program meets each requirement. In this course, your test plans should include a list of *check-points* which describe the expected program behaviour that does not depend on input data*,* and a table of *test data* with expected results.

At the testing stage, perform the tests and record the actual results on the test plan by circling or checking the check-point result, either Y or N, and indicating whether the expected output was given for the test data. Record your results *by hand*, not by typing.

User Interaction

Prompts and messages should be simple but informative. A user should be provided with a way out of any data entry sequence that is long or difficult to complete. There should be no bad surprises, and definitely no inescapable loops or program crashes!

Internal Documentation

Good internal documentation serves the author and anyone who needs to assess, maintain or work with a program. This is what is expected of your programs:

1. Program Heading and Declarations  
   At the start of each source file is a program heading that identifies the source file's name, the project (assignment), author and the date written. There is a brief description of the file’s purpose in clear and simple English. There is a list of the major items that appear in the source file. Variables, constants and functions are given clear and useful names. Single-letter identifiers are not acceptable.
2. Function Documentation  
   Each function is documented with a brief (one or two sentence) explanation of its purpose, and the purpose of its parameters and return value.
3. Indentation, Layout and Comments  
   Named constants are used in preference to unexplained number or string literals. Indentation is consistent and aids understanding. Each section of the program (function or group of related functions) is separated visually by several blank lines or a row of asterisks, etc. Comments are required for global declarations, at the start of every function and also where any special language feature or technique is used. Comments are *not* required on every line!

Backup and Version Control

You are responsible for disaster recovery of your programs. Keep numbered versions of your source code files, and take back-up copies every day! Lost programs are not an excuse for late delivery.

Assignment 1 Grading Guide

Name and email:

*(type your full name and email address)*

1. Minimum requirements satisfied

*(check the boxes of problems solved)*

* Problem 1
* Problem 2
* Problem 3

2. Additional functionality

*(list extensions completed)*

3. Documentation submitted

*(check boxes of items attached)*

* Structure charts
* Structure diagrams or pseudocode (optional)
* Test plans
* Source code
* Printed reports (optional)

4. Non-functional requirements grading criteria

- screen layout

- user interaction

- data validation

5. Development requirements grading criteria

- program organization

- dependencies

- data structures

- function/method design

- program layout

- internal documentation

Grade \_\_\_\_