Program:	**CPA2/3**
Course:	**INFO1156 "Object Oriented Programming in C++", W21**
Professors:	**Michael Feeney**
Project # 2:	**MugBook, V 2.0**
Weight:	**10% of your final mark**
Due Date:	**Friday, March 12th, @ 11:59 PM**

Description and Purpose

Building on project #1, you will make the following changes to the cMugBook class:

- cMugBook can now "persist" (save/load) data
- The main storage are now ***pointers*** to Users
  (There is now a private `std::vector< cPerson* > m_vecUsers` variable;
  all users need to be placed into this vector)
- There's some slight changes and additions to cPerson
- There's some additions to cMugBook (including that addUser() and getUser() now use pointers)

All the requirements from project #1 still apply. In other words, the methods are still expected to work in the same way they did before, even if the signature is slightly changed. For example, if you add a user, I should expect to find that user when I call getUser(), or if I change a user's last name, I should see that updated name when I get a friend list using getFriendList().

You will make a C/C++ console based 32/64 bit application that allows you to add users and their friends.

Like project #1 (and any project), you need to write code to comply with the API, specifically:

- You can usually *add* what you want to the classes
- You can't remove or change the existing method/function signatures, the member variable types or names, etc.
- You should assume that your code is working within a larger, existing system – i.e. if you change and/or remove things, it will "break" the larger system, right?
- I'll be using some testing code (that act like this larger system).

You only have to update the cMugBook and cPerson classes. There is no longer a cPersonGenerator (since I gave you one to use, right?).

You will submit:

Your (compressed) amazing **Visual Studio solution + accompanying files** to the FOL drop box.

Be sure to read the "**Submission Rules and Restrictions**" to *avoid getting a mark of zero*.

Details

Mike Acton lists “3 big lies” of software development (https://cellperformance.beyond3d.com/articles/2008/03/three-big-lies.html):

- Software is the platform.
- Code should be designed around a model of the world.
- Code is more important than data.

Read them again, remembering that they are “Big **Lies**”.

The key one here is #3: the only reason you write a program is to manipulate data.

This is *always* the case. You should always be suspicious of people who suggest otherwise.

You must implement the following functions, using the data structures listed below.
(these are also in the INFO1156_P2_API.7z file)

cPerson **MUST** be in a file called cPerson.h (class definition) and cPerson.cpp (implementation).

cMugBook **MUST** be in file called cMugBook.h (class definition). You can use any .cpp (implementation) files you want *except* the file that contains your main() function.

This structure is for dates, like birthdates. It’s a struct, which is a C thing, but in C++, it’s treated like a class with default public values. So it can have a constructor, like this one does.

For cMugBook, this is used to store the person’s birthday as “one thing” (sDataYMD) instead of three (3) separate unsigned integers.

```
#ifndef _sDateYMD_HG_
#define _sDateYMD_HG_

struct sDateYMD
{
	sDateYMD: year(0), month(0), day(0) {};
	unsigned int year;
	unsigned int month;
	unsigned int day;
};

#endif // _sDateYMD_HG_
```

```
class cPerson
{
public:
	// Constructor should set these defaults ('invalid' values):
	cPerson();		// constructor (c'tor)
	~cPerson();		// destructor (d'tor)

	std::string first;	std::string middle;	std::string last;

	enum eGenderType
	{
		MALE = 0,		FEMALE,		NON_BINARY,
		RATHER_NOT_SAY_UNKNOWN
	};
	eGenderType gender;
	std::string getGenderAsString(void);
	sDateYMD birthDay;	// within valid ranges
	std::string getAge(sDateYMD todaysDate);
	int streetNumber;			// default = -1		(which is invalid)
	std::string streetName;		// default for these is blank (i.e. "")
	std::string streetType;
	std::string streetDirection;
	std::string city;

	enum eProvinces
	{
		NUNAVUT = 20,				QUEBEC = 37,
		NORTHWEST_TERRITORIES  = 52,		ONTARIO = 190,
		BRITISH_COLUMBIA = 263,			ALBERTA = 329,
		SASKATCHEWAN = 418,			MANITOBA = 587,
		YUKON = 612,
		NEWFOUNDLAND_AND_LABRADOR = 719,
		NEW_BRUNSWICK = 765,			NOVA_SCOTIA = 852,
		PRINCE_EDWARD_ISLAND = 984
	};
	eProvinces province;
	// Converts the postal code as a string (that can be printed with cout, etc.)
	std::string getProvinceAsString(void);
	// Canadian postal codes in this format X0X 0X0
	// X = capitol letter, 0 = number (stored as alpha numeric, so '0', not 0
	char postalCode[7];
	// Converts the postal code as a string (that can be printed with cout, etc.)
	std::string getPostalCodeAsString(void);

	// Social insurance numbers are 9 digits
	unsigned int SIN;	// = 0

	// Phone numbers [AAA] [BBB]-[CCC]-[DDDD]
	//  [AAA] is the country code (001 to 999)
	//  [BBB] is the area code (000 to 9999)
	//  [CCC][DDDD] is the actual number.
	//			...instead of: "0 186 5xx xxxx"
	unsigned char phoneNumber[13];
	// Should be in this format (with brackets, space and hyphens): "(AAA) BBB-CCC-DDDD"
	// (Replace the ABCD values with the numbers indicated above)
	std::string getPhoneNumberAsString(void);
};
```

```
class cMugBook
{
public:
	// Some functions how have a std::string &error being passed by reference.
	// If the call is successful, this value is ignored by the caller.
	// If the call fails, this should return some "printable" message indicating why.

	// You MUST implement a constructor (c'tor) and destructor (d'tor)
	// EVEN IF IT DOESN'T ACTUALLY DO ANYTHING
	cMugBook();
	~cMugBook();

	// Returns true of the user is already there AND the values are identical
	// Returns false if data is invalid or missing
	bool addUser( cPerson *pThePerson );
	// This would call delete the pointer to the cPerson
	bool deleteUser( unsigned int SIN );

	// This will only update the data that is DIFFERENT **AND** VALID
	// In other words, the user has to exist already
	//
	//     bool updateUser(cPerson thePersonWithNewInfo);   **REMOVED**
	bool updateUserLastName( unsigned int SIN, std::string newLastName );
	bool updateUserFirstName( unsigned int SIN, std::string newFirstName );
	// The province and postal code needs to be valid
	// std::string &error returns printable reason for failure (bad province, empty field, etc.):
	// NOTE: you only have to return the 1st error, not all of them.
	// You can decide what is the "1st" thing you check.
	bool updateUserAddress( unsigned int SIN, int streetNumber, std::string streetName,
	                        std::string streetType, std::string streetDirection,
	                        std::string city, std::string province, /*Province now as enum)*/
	                        std::string postalCode, std::string &error );
	bool updateUserGender( unsigned int SIN, cPerson::eGenderType newGenderExpression );
	bool updateUserPhoneNumber( unsigned int SIN, unsigned char phoneNumber[13] );

	// Returns NULL if not found.
	cPerson* getUser( unsigned int SIN );

	// Returns true if it's OK.
	// Returns false if:
	//     * user doesn't exist
	//     * friend doesn't exist (or is invalid)
	//     * user and friend are the same person
	// std::string &error returns printable reason for failure:
	//     - User doesn't exist
	//     - Friend doesn't exit
	bool addFriend( unsigned int UserSIN, unsigned int FriendSIN, std::string &error );
	bool unFriend( unsigned int UserSIN, unsigned int FormerFriendSIN, std::string &error );

	// Returns false if user doesn't exit
	// vecFreindList can be zero (which is sad, but valid)
	bool getFriendList( unsigned int UsersSIN, std::vector<cPerson*> &vecFriendList );

	bool saveState( std::string fileName );
	// This will CLEAR all the present state and replace it with what's in the file
	bool loadState( std::string fileName );


private:
	// You *must* now store the users in this container:
	std::vector< cPerson* > m_vecUsers;
};
```

What it's supposed to actually do:

cPerson:

- The provinces are now stored as an enum.
- There is a phone number field, storing the value as an array of 13 *unsigned char* (char = byte) digits. You should store these digits as ***actual numbers***, rather than characters (i.e. 0 instead of character '0')
- There are functions to return these non-printable values as printable strings:
    - getProvinceAsString() returns an ALL CAPITOL string for the province. Provinces with spaces in them should be returns like that: "NEWFOUNDLAND AND LABRADOR"
    - getPostalCodeAsString() returns the postal code as string with capitol letters and numbers, with a space in the middle, so "A1B 2C3"
    - getPhoneNumberAsString() returns a printable and formatted string, in this format: "(AAA) BBB-CCC-DDDD"
        - Remove any leading zeros (0) in the country code (AAA), for instance:
            - 001 519 555 1212 should be returned as (1) 519-555-1212
            - 012 327 228 1234 should be returned as (12) 327-228-1234
- Some update methods still take the values as string, and need to be parsed/translated into the internal representations:
    - Provinces need to be valid (see enum eProvince as a guide) and can be any case, but the spacing and use of "and" has to be correct, for instance:
        - "Ontario", "ONTARIO", or even "OnTaRiO" would be the enum ONTARIO
        - "NEWFOUNDLAND AND LABRADOR", "newfoundland and labrador ", and even "NeWfOuNdLaNd aNd lAbRaDoR " all would be the enum NEWFOUNDLAND_AND_LABRADOR
        - For provinces with spaces, they have to have only 1 space, so:
            - "BRITISH COLUMBIA" or "bRiTiSh cOlUmBiA" are OK
            - "BRITISH COLUMBIA" isn't.
        - The easiest way to do this is to convert the strings into all upper or all lower case, then compare with "valid" province strings.
        - Note: "&" ***can't*** be used in place of the word "and"
- Age should be a "adult human" age, so maybe 16 to 18 to some reasonable "old person" age.
  No, "40" isn't "old". Just saying... I'm over it.
  Some students mentioned people can live longer than 100 (my Dad's mom lived to 103), so if you want to make that a little higher, feel free. Apparently, Jeanne Calment lived to the age of 122 https://en.wikipedia.org/wiki/List_of_the_verified_oldest_people. Now Commander Data (from Star Trek) lived to 475, and Bender from Futurama had his head live to 1055 while his body was only 25. But those two are a) robots and b) fictional (I know, I know, I'm sad Bender isn't real, either!).

  Anyway, it should stop ridiculous ages like 1,000,000 from being entered.

- getAge() takes today's date (in the sDateYMD struct format) and calculates the person's age. There's a ton of ways to do this, and here's just one: https://www.geeksforgeeks.org/program-calculate-age/.
  I'm only concerned with the YEAR age, not the months and days, etc.

cMugBook:

- The functions are pretty self explanatory, and the same as project #1.
- Default gender is “prefer not to say”.
- The addUser(), getUser(), and getFriendList() uses pointers.
    - addUser() takes the pointer of a person, created by the caller, and stores it.
        - It does NOT store a copy of the object, just the pointer
        - The caller *will not* delete the user; once addUser() is called, it’s assumed that cMugBook will be “responsible” for the user from then on (i.e. it deletes it)
    - getUser() returns the pointer to the user.
        - It is NOT a copy of the object, but a pointer to the actual object. In other words, you simply copy the pointer, but you *don’t* call “new” again.
        - Assume the caller *will not* delete or redirect the pointer (but you actually have no control over this).
        - It returns NULL (or 0) if it can’t find the user. (if you *really* want to return nullptr, that’s fine, too)
    - getFriendList() returns a vector of pointers to the users.
        - Like getUser() these are NOT copies of the objects, but point to the object.
        - Assume the caller will *not* delete or redirect the pointer.
        - This will *never* return NULL (0) pointers. In other words, all the values in the *returned* vector point to a valid user (if there are no friends, the vector would be empty, right?).
- Some methods have an additional “std::string &error” variable that is used to get more details on why a method might fail. This is passed by reference by the caller, then the caller can use this (print it out or whatever) as an error message for the rest of the system.
    - This should return “human readable” text of your choosing, but it has to be clear and “make sense” if presented (as is) to the user.
      Like “Missing last name” might be OK, but “Yo, yo, something is wrong, dawg!” or just “last name” isn’t great. i.e. the dialog on the right is a little confusing...

    - I’m going to be marking that:
        - There’s *something* in the error string
        - It’s more or less “sensible” and “clear” (like if I showed it to some random person, they could guess what went wrong).
    - If the function works correctly and/or returns OK (true), then the “error” string is ignored completely (by the caller). It’s only when it *doesn’t* work.
    - If there’s more than one possible error, you can just return the first one you find. You *don’t* have to return *all* the errors (unless you want to). For instance, if updateUserAddress() had all blank fields, and you happened to check “city” first, then it’s 100% OK to *just* return an error message about that alone. This is pretty typical on web forms, where the user keeps correcting and resubmitting until they don’t get any error messages.

      In general, unless you *need* to, it’s always easier, clearer, and faster to just exit at the 1st sign of trouble. What’s the point of checking the rest of the user’s update information if the city is not right? (Your friendly C/C++ compiler does *not* do this, and keeps barrelling along, giving several screens of error messages as you’ve likely noticed... helpful? Not really, since you just go to the 1st error message and correct that, then try again, am I right?)

- saveState() and loadState():
    - saveState() saves all the current information into a text file. This can be “human readable” or not (I’d *strongly* suggest that you make it human readable, though).

      Basically, you would go through all your vectors and save the text, separated by a space (or comma or whatever), in some organized way:
        - Save all users
        - Save all user’s friends lists (maybe just the SINs?)

    - loadState() opens that file and does the reverse. You just read the information in the same order that you wrote it out.

    - I’d also suggest:
        - You make saveState() first, *then* loadState() because it’s hard to make a “loader” when you don’t have anything to load, right?
        - Make it AS SIMPLE AS POSSIBLE.
        - Make the initial saveState() so that it only does, say, jus the user’s names, then put that loading functionality into loadState() and keep iterating back and forth, as you add stuff. This way, if you “screw up”, you’ll know it immediately.
        - I would STRONGLY discourage you from “writing it all at once”.
        - I would ALSO VERY strongly DISCOURAGE you from making your own meta language thing (like “I know, I’ll save it as JSON or XML!!!” No, don’t do that. Seriously.).

How you get your marks:

- It's basically a list of the methods, where things that are more difficult are worth more.
- A few things were added to cPerson, with some coding that has to happen to get them to work (like getAge(), the returning the province and phone number, etc.

  So you'd get some marks for those.

- cMugBook has the most changes, and like project #1, it's divided into functionality. There's around 13 methods, but many of them have already been "handled" in project #1. I'll be focusing on the ones that would require more work:
  - If the cMugBook doesn't even have the functionality from project #1, then it's at my discretion to even mark project #1 – hint: I likely *won't* continue marking .
  - 10%: However, since the people are stored as pointers now, there's *some* changes that have to be made, so the functionality and validation that exists gets 10%. If it's all still there, you get 10%, and proportionally less if there's stuff missing (I'll be using pretty much the identical testing code)
  - 40%: saveState() and loadState(): this is the bulk of the mark, as it involves the most work and the most possible issues.
  - 40%: The additions to cPerson and cMugBook (outside of the state). This is the validation, the getAsString() methods, etc. These are broken down into functionality. Like getPhoneNumberAsString() would be given the same marks as how you validate the updateUserAddress(), which involves parsing the province strings that are passed.

Bonuses:

If you have any ideas for what could be a bonus, please let me know and I'll add them here.

They'd be worth 5%-10% depending on how difficult/involved they would be.

These are actual bonuses, so can give you *above* 100%. (if you can't get *above* 100% or you can't even get to 100%, then it's not a bloody bonus, is it?)

While you "have to get 100% on the rest of the project, that's no "hard and fast", like if you got 95% (or something, like you missed the odd little thing) *and* did the bonuses, I'd give them to you.

Now, if you barely had any of the functionality, and did the bonuses... honestly, why are you even doing the bonuses, then? Seriously?

Bonuses are added to your mark. In other words, this project is worth 10% of your final mark, but if you got some bonus that gave you, say 11.5% (out of 10%), then that 1.5% would be directly added to your overall mark. Some instructors (and FOL) calling this "can exceed category").

A few initial bonuses:

- **"State" related bonuses.** Note: These save files would have to be compatible to the saveState() and loadState() methods. In other words, if I created a new and empty cMugBook, I could call these "append" or "load" methods and users would be added, and I if I only had 1 user, and called saveState(), I should be able to use that generated file as a valid input to loadUser().
    - 5%: **`appendState(string fileName):`**
        - Takes *another* configuration file, and adds those users to the existing set of users and friends *IF* they weren't already there.
        - Any "duplicate" people are ignored.
    - 5%: **`saveUser(unsigned int SIN, string fileName):`**
        - This would save *only* this particular person's state to a file.
        - This would be like the EU's law with facebook and Google, where you can ask to get ALL the information on yourself, and you can also ask that they completely purge you from their records (apparently it's a LOT!: https://youtu.be/O90PShJVu58?t=429)
    - 5%: **`loadUser(string fileName):`**
        - This would load a single user (and their friends, if they were valid) into the current information, taking the file that saveUser() generated.
        - If the user already existed, this would be ignored.
    - 5%: **`loadAndReplaceUser(string fileName):`**
        - This would do the same as loadUser() but would *replace* an exiting user's data.
        - If the user *didn't* exist, it *wouldn't* load them (i.e. it's *replacing*).

Submission Rules and Restrictions:

Things that will **get you zero** and are **non-negotiable**:

- Changing the API so it won't build. This includes:
    - The classes, methods, etc.
    - The "signature" of the methods/functions/whatever.
    - What the format of the files you have to input/output.
      (In this case, the output of the INFO1156NameGenerator program is what you have to "deal with", so do *not* change it "because reasons").
- Using the "`auto`" keyword (nor can you use a #define/typedef to circumvent this).
- Using or including the boost library.
- Using "lamba" functions. You *may* use "predicate functions" or "functors".
- Using another external library that you didn't write. You may only use the STL (standard template library) libraries – i.e. the ones "built in" to the language.

  This includes meta parsing/handling things like JSON/XML/Lua/whatever libraries that you are using for saving and loading state.

Project Corrections

If any corrections or changes are necessary they will be posted to the course web site and you will be notified of any changes in class. It is your responsibility to check the site periodically for changes to the project. Additional resources relating to the project may also be posted.

75/10-year old “squinty eye” plagiarism test:

I have very little tolerance for plagiarism, but many students are unclear about what it is.

Basically, it’s submitting **somebody else’s work as your own**.

There is sometimes some confusion over this because you could argue nothing is actually “unique” (see: http://everythingisaremix.info/ for a fascinating overview of this).

The whole point of assignments/tests/projects in this course (or any course, really) is to try to see if you are actually able to ***do*** the coding that’s asked of you. In other words: How competent are you? Handing me someone else’s code and/or making a trivial change isn’t good enough.

Also, it’s illegal:

- http://www.plagiarism.org/ask-the-experts/faq/
- http://definitions.uslegal.com/p/plagiarism/
- http://en.wikipedia.org/wiki/Plagiarism
- https://www.legalzoom.com/articles/plagiarism-what-is-it-exactly

In other words, I’m not going to be drawn into a giant debate over how “different” your code is from mine or anyone else’s, if any sensible person (including me) would conclude that the code/application is pretty much the same thing, then it is. It is up to my discretion to decide this.

- While you may freely “borrow” mine (or anyone other) code ***but*** your code should be “sufficiently” different from mine (you might want to replace the word “sufficiently” with “significantly”).
- In other words, you *cannot* simply use an existing game engine (or part of a game engine) to complete this assignment; it should be either completely new of **significantly** modified.
- How will I determine this?

    - If I showed your application and/or your source code to either my pragmatic 83-year-old mother, or a typical 10-year-old, or even some random person walking down the hallway (i.e. a non-expert), and they looked at it, tilted their heads, squinted their eyes, and said “you know, they look the same,” then they ***are*** the same.

    - Another test would: How much time it would take for a "competent programmer" (for example, *me*) to make the changes you are submitting? The point here is that I don’t “care” if you tell me “But it took me *weeks* to make the changes!” Fine, but if I can make those same changes in 10 minutes, then not a lot of work has been done (certainly **not** sufficient work – these projects should show take **days** of work having been done).