Programming Assignment #3
CS 302 Programming Methodologies and Software Implementation

***** Make sure to read the Background Information first! *****

*****This program is about Dynamic Binding *****

Program #3 - Goals

We have three goals for our third programming assignment.

> GOAL 1: The primary goal is to experience **dynamic binding**. As such, this program is smaller than the first two programs and has a specific design. There will be a hierarchy where there are three derived classes which will have the same functions, with the same arguments and return type as much as possible. As such, it will be important to pay close attention to this assignment's writeup as it outlines the specific hierarchy expected.
>
> GOAL 2: The second goal of this assignment is to experience **exception handling** and modern C++ this time with **smart pointers**. With modern programming languages in general, it will be very important to understand how to handle erroneous situations with exception handling. You are allowed to use structs to create new "data type" names for exceptions being thrown.
>
> GOAL 3: The third is to become fluent using **Make** and turn in a **Makefile** with the assignment.

Program #3 - Specifics

We had guests over the very long winter break and it was great because each evening we played games. And, they weren't all the same type of game. Some evenings we were on the Xbox playing Halo or Destiny, other nights we played cards, and on occasion we played a series of new board games (which were not successful). Next time we have visitors, I want to be able to quickly go through my collection of different kinds of games and offer the best selection based on what we are most interested in.

We are going to write a program to help find the best game that is right for the user given the genre, skill level, type (computer, board, card, etc.) and other criteria.

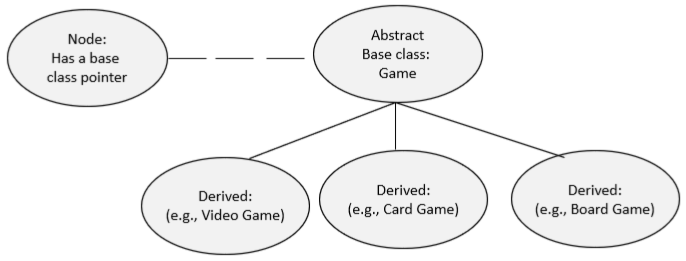
First Progress Submission: The core hierarchy

In this programming assignment, your job will be to pick three different types of games with similar but yet unique characteristics. For each type of game your program could support the following data (you may alter this list):

1. Name (e.g., Minecraft)
2. Genre (e.g., Sandbox, sub genres are Survival and Adventure)
3. Skill Level (e.g., rated for 8 years)
4. Accessibility (e.g., what are the requirements for playing)
5. Number of Players (e.g., single player, multi-player (or like Sorry it will have a specific number of people who can play at one time)
6. At least two unique characteristics about a game that is not common (such as if there are cards, a board, dice, avatars, game level, etc.).
7. Pick at least one other of your own choice that would be important in helping determine if this game was a good match for someone looking to play. This could be the reviews received by other players or the level of violence, for example.

You are going to support three different types of games that people might consider playing; they could be video games, board games, and card games. Or, you may decide to include instead team games (e.g., volleyball), field games (e.g., bad mitten) or table games (e.g., ping pong). Whatever you pick needs to have a self-similar public interface (same functionality) but some unique qualities in the implementation of those methods. For example, a card game will have a deck of cards that other types of games might not have. A board game has a player's piece (with a shape and color) and either dice, something to spin or a card to draw. The similar characteristics will be pushed up and the unique information will be derived. The three types of games must be derived from one common base class which is the hub of dynamic binding. Make this class an **abstract base class**.

Base Class: Make a general purpose "Game" class as the Abstract Base Class (ABC); it will be the glue used to hold your hierarchy together. You can still push up the common and derive the differences which means the ABC will have all of the data and functions that would be common among the derived classes. Then have three different types of games as the three derived classes.


Node: Has a base class pointer
Abstract Base class: Game
Derived: (e.g., Video Game)
Derived: (e.g., Card Game)
Derived: (e.g., Board Game)

Now let's talk about the methods that you should be implementing. Besides constructors, setters, display, and destructors, there should be three functions that are the same from the client perspective for each of the three derived game classes. Remember this program isn't about playing the games but rather helping users decide what games would be the best fit. It will be important to pick three functions and implement them in the derived classes – the idea is that they have exactly the same return type, same name, and same arguments – but the implementation of the body of the functions will be different.

Add one function for each of the derived classes that is related to the different feature that you added in the derived game classes.

Here are a few possible public member functions to select from. You may select from this list or add others of your own design:

1. Check to see if the level of accessibility for the game fits the clients expectations
2. Check to see if the game is age appropriate for the clients needs
3. Check to see if the genre includes the desired features that the client is looking for.
4. Determine if the reviews has a history of red flags to be aware of
5. Determine if the game is highly recommended

With dynamic binding, we need a self-similar interface for all public methods. The constructors cannot be virtual, so they can have arguments that are unique. But, all other functions must be the same arguments (and return type) across the hierarchy and placed as virtual in the ABC. For each of the derived classes, pick three methods that make the most sense to you!

In addition, you will need to experience RTTI. To work with the different characteristics where the functionality is not "self similar" **you must have** one method that is different, which is not virtual in the base class, so you can experience calling it with the dynamic_cast operation – performing downcasting. This only needs to be done for ONE of your classes.

Second Progress Submission: Creating the Application

For this assignment, you will be implementing one data structure that can hold a collection of the three different types of games, all in one data structure! Each node must have a **base class pointer** (to the **abstract base class)** allowing **upcasting** to take place. This allows the three different types of Games to be pointed at in the data structure and the right functions will be called!

We will be implementing a tree as our data structure, organized by two search keys to assist in the traversal algorithms when looking for the best possible fit. You get to decide which two search keys would make the most sense give the methods you are electing to support. All repetitive data structure algorithms should be implemented using recursion. You have a choice of the following:

For the second progress submission, implement at least two of the BST methods or one of the Balanced tree methods.

Choice 1: Binary Search Tree organized by two search keys of your choice. This requires full support of insert, removal, display, and remove_all.

Choice 2: A Balanced Tree (pick from this list: 2-3, or Red-Black). Pick a tree that you are most comfortable after performing some research; if you want to implement a tree not in this list, please first seek authorization. Implement insert, display, and remove_all. (no need to remove an individual item). Use the STL when there are multiple matches of the same search key.

- If you select a **Binary Search Tree** for this assignment, please be aware that a balanced tree will be required for Program #4-5 in Python.
- If you select a **Balanced Tree** for this assignment, then a standard Binary Search Tree will be required for Program #4-5 in Python.

READ THIS: In Program #3, make sure to follow these guidelines:

- Use entirely the string class (**no char *'s in this assignment allowe**d),
- Use exception handling for detecting and handling erroneous situations
- Write a Makefile to help the grader compile your work
- Submit a README file with instructions!
- Use smart pointers (limit your raw pointers)
- Use the STL whenever a list of data is required (except for the one data structure you are required to implement). Make sure to be comfortable using Vectors, Arrays, and Lists.