Assignment 5

Overview

One problem with the program created in Assignments 1 to 4 is that the items are hard-coded in. Another is that, whenever the player takes or leaves an item, the game has to search through the item list to find it.

In this assignment, you will load the items from a file. This will allow you to load several different games worlds using the same code. You will speed up finding items by sorting items when the program starts and then using binary search to find individual items.

You will also add a score table, which is like a high score table except that it includes all the scores. It will be stored in a file and loaded and updated each time the game runs. While it is in memory, the score table will be represented as a linked list.

The purpose of this assignment is to ensure that you understand how to use pointers, dynamic memory allocation, linked data structures, and how classes can be used to avoid memory leaks. For Part A, you will load `Item`s from a file and sort them. For Part B, you create an element for a linked list. For Part C, you will use that element to program a score table. For Part D, you will update your `main` function to use the score table and load a different game world.

The Items Data File

The following is a sample data file for items:

```
3

b     2     -10
There is a boat (b) here.
You are in a boat (b).

a     10    5
There is an apple (a) here.
You are carrying an apple (a).

c     13    0
There is a cat (c) hiding somewhere near here.
You are trying to keep hold of a struggling cat (c).
```

The first line is the number of items. After that, there are four lines in the file for each item. The first is always blank. The second contains the item id (a `char`), the starting location (an `unsigned int`), and the points value (an `int`). The third line contains the world description and the fourth line contains the inventory description. The file ends with one or more blank lines.

Requirements

Copy the code and data files of your Assignment 4. Do not just modify Assignment 4.

(If you are using Visual Studio, you must start by creating a new project for Assignment 5. Do NOT copy the whole folder including the `.sln` file or massive confusion will result!)

Part A: Load `Item`s from a File [50% = 45% test program + 5% code]

Change the `ItemManager` class to load the items from a file and store them in a dynamically allocated array (see Section 11 of the notes). Also put the `ItemManager` class in canonical form (as described in Section 10 of the online notes) and sort the items by id after they have been loaded.

By the end of Part A, your `ItemManager` class will have `public` member functions with the following prototypes:

- **`ItemManager ();`**
- `ItemManager (const string& game_name);`
- **`ItemManager (const ItemManager& to_copy);`**
- **`~ItemManager ();`**
- **`ItemManager& operator= (const ItemManager& to_copy);`**
- `unsigned int getCount () const;`
- `int getScore () const;`
- `void printAtLocation (const Location& location) const;`
- `void printInventory () const;`
- `bool isInInventory (char id) const;`
- `void reset ();`
- `bool take  (char id, const Location& player_location);`
- `bool leave (char id, const Location& player_location);`

The `ItemManager` class will also have `private` member functions with the following prototypes:

- **`void load (const string& filename);`**
- `unsigned int find (char id) const;`
- **`void sort ();`**
- `bool isInvariantTrue () const;`

Perform the following steps:

1. Download the new `TestHelper.h` and `TestHelper.cpp` files. They have been expanded to allow the test programs to give partial marks if your program crashes.
2. Add a member variable of the `int` type to the `ItemManager` class to store the number of items. This variable should not be accessible outside the class. Replace every use of the `ITEM_COUNT` constant in any member function in the `ItemManager` class with the member variable. Remove the `ITEM_COUNT` constant.
    - **Note:** Do not dynamically allocate memory for the item array in the header file. We will allocate it in the constructors. Allocating memory in the header file is legal (starting in C++11), but it won't do what you want. The biggest reason it won't work is that you have not read the data file yet, so you don't know how much memory to allocate.

3. In the class definition for the `ItemManager` class, replace the statically allocated array of `Item`s with a pointer to an `Item` (type `Item*`). We will use this pointer to hold the address of a dynamically allocated array of `Item`s.

4. Add an additional check to the class invariant to ensure that the item array pointer is never `nullptr` (or `NULL`).

    - **Reminder:** If the class invariant is not true, the `isInvariantTrue` function should return false. In this case, it is (also) not true if the item array pointer is `nullptr` (or `NULL`).

5. Add a default constructor to set the item count to `0` and dynamically allocate an array of zero `Item`s. Use an `assert` to check the class invariant at the end of the default constructor.

6. Add a copy constructor. It should make a deep copy of the dynamic array of `Item`s. Put an `assert` at the beginning of the copy constructor to check the class invariant for `to_copy` (which is the `ItemManager` you are copying from) and an `assert` at the end of the constructor to check the class invariant for this `ItemManager`.

    - **Note:** To make a deep copy of the `ItemManager` named `to_copy`, you should first copy the value of the item count field. Then allocate space for the new array according to this count. Use the member pointer declared in the `ItemManager` class to hold the address of the new dynamically allocated array. Finally, copy the contents from the `to_copy` array to the new array. See Section 11 of the online notes and look for "Pointer to Array".

    - **Reminder:** The copy constructor for the `ItemManager` class has the original `ItemManager` passed by constant reference (`const ItemManager& to_copy`). If you pass it by value (`ItemManager to_copy`), the copy constructor will invoke itself repeatedly until your program crashes from too many nested function calls.

    - **Note:** We do <u>not</u> have to check the class invariant for `to_copy` at the end of the constructor. It is declared as `const`, so if it was valid at the beginning of the constructor, it will still be at the end of the constructor.

7. Add a destructor that frees the dynamically allocated memory for the `Item` array. Use an `assert` to check the class invariant at the beginning of the destructor. Do <u>not</u> check the class invariant at the end.

    - **Reminder:** Use `delete[]` to deallocate the array rather than `delete`.

8. Add an assignment operator. It should check for self-assignment, free the existing memory, make a deep copy, and finally return `*this`. Include asserts to check the class invariants for both `ItemManager`s at the beginning of the function and for this instance of the `ItemManager` at the end.

    - **Reminder:** Use `*this` to refer to the whole of this `ItemManager` instance.

    - **Reminder:** A typical assignment operator looks as follows

```
// add asserts
```

```
if(this != &to_copy) // checks for self-assignment
{
      // everything from destructor here
      // everything from copy constructor here
}
// add asserts here
return *this;
```

9. Add a `private` member function named `load` that takes a `string` representing a file name as a parameter. It should open the file, read the item count and store it in the appropriate member variable, allocate an array large enough to hold that many items (and store a pointer to it in the appropriate member variable), and then read the items from the file. Use an `assert` to ensure that the item array pointer is not `nullptr` (or `NULL`) at the end of the function.

    - **Hint:** You will need to copy/paste the contents of the `ghostwood_items.txt` into a file on the computer.
    - **Hint:** Use the `loadNodes` function from the `World` class to remind yourself about how to read from files. Also, look at `ghostwood_items.txt` to see an example of what needs to be read.
    - **Hint:** Use `getline` to get read the newline character after the number of items.
    - **Hint:** When reading an item, start by using `getline` to read the blank line. Then read the id , starting location , and points value into local variable using formatted I/O (`>>` notation). Then use `getline` to remove anything else on the line (such as the newline character). Then read each of the descriptions with a separate `getline`. Finally, use the Item constructor to combine all the values into an `Item` object and store it in the array:
      `itemArray[index] = Item(id, location, …)`
    - **Hint:** We did not define an `operator>>` function for the `Location` type, so you cannot write `my_file >> my_location`. Instead, you should read the location into an `int` and then convert it to a `Location`.
    - **Hint:** As you read in the first few lines of the file, immediately display the value of each variable that you have read in, so that you can tell that they are being read properly. Print them before calling the `Item` constructor.
    - **Hint:** When you are developing this function, use the `Item::debugPrint` function to print out each item after you have created it using the `Item` constructor. Then, when you are sure the items are being loaded correctly, remove or comment out the call to `debugPrint`.
    - **Hint:** Do not put a check on the `isInvariantTrue` function at the end of the `load` function. The `load` function is a `private` helper function, so it does not have to ensure that the invariant is true. This is good because the array has not yet been sorted.

10. Write a `private` member function named `sort` to sort the item array based on their ids. You must use either selection sort or insertion sort, and you must include a comment at the top of the function saying which you used. The items should be sorted from smallest id to largest.

- **Reminder:** The `Item` class has a less than operator (from Assignment 3), so you can write `item1 < item2`. However, it does not have ==, !=, <=, >, or >= operators.
- **Hint:** After sorting the items, print out their ids. This will allow you to see if they have been sorted correctly. If you see more than one of the same id, there is something wrong with your `sort` function.

11. Update the constructor that takes the game name as a parameter. It should start by setting the item array to `nullptr` (or `NULL`). Then it should calculate the name of the item data file and load the items from that file. Then it should sort the items. Finally, it should use an `assert` to check the class invariant.
    - **Hint:** The `ItemManager` class has a member function to load the items from a file and another one to sort the items.
    - **Note:** If you are using the provided solution, remove the `initBlizzard` and `initGhostwood` functions. They are no longer needed.
12. Test your `ItemManager` module with the `TestItemManager5.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program. It should give you full marks.
    - **Reminder:** There are new versions of the `TestHelper` files. The test programs will not compile with the old ones.
    - **Hint:** `g++ Location.cpp Item.cpp ItemManager.cpp TestHelper.cpp TestItemManager5.cpp`
13. Run the game test cases from Assignment 4. They should still work.
14. Update the `isInvariantTrue` function class invariant to also require that the items in the array are in sorted order, i.e. every item must have an id strictly smaller than the item after it. For example, the apple `'a'` must occur before the boat `'b'`, which must occur before the cat `'c'`.
    - **Hint:** If there are `N` items, you have to make `N - 1` comparisons.
15. Update the `find` function to use a binary search. The function should still return the index of the item with the specified id if there is one and `NO_SUCH_ITEM` otherwise.
    - **Note:** You do not have to worry about the case where there is more than one item with the same id because the class invariant ensures that cannot happen.
    - **Note:** You must use the `find` function to find the item index in `isInInventory`. You may also use it in `take` and `leave`, but you are not required to.
16. Re-test your `ItemManager` module with the TestItemManager5 program. It should still work.
    - **Hint:** `g++ Location.cpp Item.cpp ItemManager.cpp TestHelper.cpp TestItemManager5.cpp`

Part B: The Score Table Element [15% Marked with Part C]

Add a class named `Element` to represent a node in a linked list of scores. The `Element` class will not be encapsulated and it will not be placed in canonical form. Its purpose is to keep track of the number of `Element` objects that exist at any given time. Because the `Element` class is a simple class and it is tightly coupled to the `ScoreTable` class, you should place the two classes together in the same files. Put the definition information for the `Element` class in a header file named `ScoreTable.h` and the implementation in a source file named `ScoreTable.cpp`. We will add the score table itself in Part C.

By the end of Part B, your `Element` class will have a default constructor and destructor with the following prototypes:

- **`Element();`**
- **`~Element();`**

It will also have associated non-member functions with the following prototypes:

- **`int getAllocatedElementCount ();`**
- **`Element* copyLinkedList (const Element* p_head1);`**
- **`void destroyLinkedList (Element* p_head);`**

Perform the following steps:

1. Create a class named `Element` to represent a single element in the linked list. It should contain a player name named `name` (which is a `string`), a score named `score` (which is an `int`), and a pointer to the next list element named `p_next` (which is an `Element*`). All of these member variables should be publicly accessible.
    - **Reminder:** `Element` should be defined in the `ScoreTable.h` header file.
    - **Note:** The test program requires exactly these names for the member fields.
2. Create a global `int` variable (not `unsigned int`) in the `ScoreTable.cpp` source file to represent how many `Element`s exist. Encapsulate this variable by using the `static` keyword. Initialize it to `0`.
3. Add a non-member function named `getAllocatedElementCount` that returns the count of how many elements exist. Even though it is not in a class, its prototype should be in `ScoreTable.h` and its implementation should be in `ScoreTable.cpp`.
4. Add a default constructor to the `Element` class that initializes the member variables of `Element` to appropriate values. It should also increment the count of how many elements exist.
    - **Note:** The correct initial value for the next pointer is `nullptr` (or `NULL`).
5. Add a destructor that decrements the count of how many elements exist.
6. Add a non-member function named `copyLinkedList` that creates a deep copy of a linked list. It should take a pointer to the head of the existing linked list to copy as a parameter and

return a pointer to the head of the new linked list. If the parameter is `nullptr` (or `NULL`), is should return `nullptr` (or `NULL`). Refer to Section 12 of the online notes for sample code.

7. Add a non-member function named `destroyLinkedList` that frees the memory associated with a linked list. It should take a pointer to the head of the linked list as a parameter. If the parameter is `nullptr` (or `NULL`), there should be no effect. Refer to Section 12 of the online notes for sample code.

    - **Reminder:** You cannot use the next pointer of an element after you delete it.

8. Test your `Element` functions with the `TestElement5.cpp` program provided. You will also need the `TestHelper` module. The test program should give you `12/40` marks. The remainder of the marks are in Part C.

    - **Hint:** `g++ ScoreTable.cpp TestHelper.cpp TestElement5.cpp`

Part C: The Score Table [40% = 30% test program + 10% documentation]

Loading and saving the score table became optional Part E on March 25, 2022.

Add a class named `ScoreTable`. The scores will be stored in a linked list that is kept sorted from highest score to lowest.

By the end of Part C, your `ScoreTable` class will have `public` member functions with the following prototypes:

- **`ScoreTable ();`**
- **`ScoreTable (const ScoreTable& to_copy);`**
- **`~ScoreTable ();`**
- **`ScoreTable& operator= (const ScoreTable& to_copy);`**
- **`void print () const;`**
- **`void insert (const string& player_name, int score);`**

The `ScoreTable` class will also have a `private` member function with the following prototype:

- **`bool isInvariantTrue () const;`**

Finally, you will still have the `Element` functions from Part B.

Perform the following steps:

1. Create a class named `ScoreTable`, also in the `ScoreTable.h` header file. It should contain a pointer to the head of a linked list (of type `Element*`) as its only member field.

    - **Note:** The `ScoreTable` class should be declared after the `Element` record. If you declare it before, `Element` will be undefined when the compiler reaches the `ScoreTable` function prototypes.

2. Add a default constructor to the `ScoreTable` class. It should set the head pointer to `nullptr` (or `NULL`).

3. Add a function named `print` that prints the scores to the screen. It should start by printing `"Scores:"` on a line by itself. Then it should print one line for each `Element` in the linked list. Each line should consist of the score, followed by a tab, followed by the player name.

    - **Note:** The test program requires you to match this format exactly, including using a tab (`'\t'`) instead of spaces for alignment.

4. Add a function named `insert` that inserts a new entry into the score table. The function should dynamically allocate a new `Element`, set it to contain the score and player name, and insert it into the linked list. If there are no elements, the new `Element` should become the head. Otherwise, insert the new `Element` immediately before the first existing `Element` with a lower score. If there are no `Element`s with a lower score, insert it at the end.

    - **Hint:** Refer to Section 12 of the online notes for sample code for `insert` that can be adapted to the `ScoreTable` class.

    - **Hint:** Conceptually, there are four cases you have to consider. The new element can be inserted (a) into an empty list, (b) at the head of the list, (c) between two elements, and (d) at the tail of the list. However, the implementation may be able to collapse them into fewer. The implementation in the online notes collapses them into only two cases.

5. Add a copy constructor, destructor, and assignment operator to the `ScoreTable` class.

    - **Reminder:** Every constructor must assign a value to every member variable.

    - **Hint:** Call the `copyLinkedList` function from your copy constructor, call the `destroyLinkedList` function from your destructor, and call both of them from your assignment operator. See the `Multiset` class in Section 12 of the online notes for ideas on how to use these kinds of functions.

    - **Hint:** The copy constructor should create a deep copy using the `copyLinkedList` function, rather than a shallow copy.

    - **Hint:** The destructor should deallocate the linked list using the `destroyLinkedList` function.

    - **Hint:** The assignment operator should check for self-assignment, deallocate the existing linked list in this `ScoreTable` instance, create a deep copy of the existing linked list (the one in the other `ScoreTable` instance), and return `*this`. The order matters.

6. Test your `ScoreTable` module with the `TestScoreTable5.cpp` program. You will also need the `TestHelper` module. The test program should give you full marks.

    - **Hint:** `g++ ScoreTable.cpp TestHelper.cpp TestScoreTable5.cpp`

    - **Note:** This test program includes all the tests from `TestElement5.cpp`.

7. Add a class invariant checked by a `private` helper function named `isInvariantTrue`. The class invariant requires that every linked list element has a score greater than or equal to the next element.

8. Use `asserts` to check that the class invariant at the end of each `public` member function not declared as `const` except the destructor.
    - **Note:** If you use `return` to leave a function part way through, you will need to `assert` the class invariant before every `return` statement too.
    - **Reminder:** Constructors are member functions, so you should check the class invariant at the end of.
9. Use `asserts` to check that the class invariant at the beginning of every `public` member function except the constructors.
10. Use `asserts` to check that the class invariant is true for the `to_copy` object at the start of the copy constructor and the assignment operator.
11. Add documentation for each `public` function in the `ScoreTable` class using the style shown in the class notes. There are six of them.
    - **Reminder:** You do not have to document `private` functions. They are not part of the interface.
    - **Reminder:** Do not document the class invariant as a precondition.
    - **Note:** You do not have to document the functions in the `Element` class. They are not members of the `ScoreTable` class.

Part D: Update the **`main`** Function [10% = 4% stability + 6% test output]

Update your `main` function and the `Game` class so that your program loads a different world and keeps track of scores between games.

Perform the following steps:

1. After printing the welcome message, ask the user his/her name. Read it in and store it.
    - **Hint:** Read in the user name with `getline`, not with formatted I/O (>> notation). Otherwise you will read user names containing spaces (e.g. `"Jar Jar"`) incorrectly.
2. After reading the name, print `"Hello, XXX!"`, where **`XXX`** is the user name.
3. Add a `ScoreTable` member variable to the `Game` class.
4. Add a function to the `Game` class named `updateScoreTable` that updates and displays the score table.
    - **Hint:** Your function in the `Game` class will need one parameter that gives the player's name.
5. In the `main` function, after printing the player score, update and display the score table. This should be done when the game is over and on the restart (`'r'`) command.
    - **Reminder:** If the player enters the quit (`'q'`) command, the game is over.

6. Change the `main` function to load a game named `jungle`. The only thing you should have to change is the argument to the `Game` constructor. You will need the `jungle_nodes.txt`, `jungle_text.txt`, and `jungle_items.txt` data files.

7. Test your program with the five test cases provided: `testcase5A.txt`, `testcase5B.txt`, `testcase5C.txt`, `testcase5D.txt`, and `testcase5E.txt`.

    - **Hint:** `g++ Location.cpp Node.cpp World.cpp Item.cpp ItemManager.cpp ScoreTable.cpp Game.cpp main.cpp`

    - **Hint:** In replit, you can test your game by redirecting input from the test case into your program:
      `./game < testcase5A.txt`
      This does not work in Visual Studio Code because the Microsoft programmers did not implement the < command.
      It is possible in (full) Visual Studio, but it is harder and you do not need to because Visual Studio handles pasting into the output window well.

Optional Part E: Store [0%]

Copy all your files to another folder before continuing. If you cannot make your program work after completing E, hand in the unmodified version. Remember that Part E is not for marks.

Modify the `ScoreTable` class to store its data in a file. This will allow scores to persist between different runs of the program.

By the end of Part E, your `ScoreTable` class will have `public` member functions with the following prototypes:

```
ScoreTable ();
ScoreTable (const string& game_name);  // load constructor
ScoreTable (const ScoreTable& to_copy);
~ScoreTable ();
ScoreTable& operator= (const ScoreTable& to_copy);
void print () const;
void save (const string& game_name) const;
void insert (const string& player_name, int score);
```

The `ScoreTable` class will also have `private` member functions with the following prototypes:

```
void printToStream (ostream& out) const;
string getFilename (const string& game_name) const;
bool isInvariantTrue () const;
```

Finally, you will still have the `Element` functions from Part B.

Perform the following steps:

1. Refactor your `print` function to create a private helper function named `printToStream` that takes a non-constant reference to an output stream (`ostream&`) as a parameter. The

print function should print the "Scores:" line and then call printToStream with cout as the argument. Everything else should be in printToStream. Change the output in the printToStream function to print to the ostream& parameter instead of cout.

- **Hint:** ostream is the base class of ofstream (for file output) and cout. You can print to an ostream exactly the same way as you print to cout.

2. Add a private helper function named getFilename that takes the game name as a parameter and calculates the name of the scores file. This is the name of the game with "_scores.txt" appended to it.

3. Add a function named save that takes the game name as a parameter. It should calculate the filename for the game and then save the current scores to that file, overwriting the previous file contents.

   - **Hint:** Call the printToStream function with the file output stream as its argument. Your file output stream should have type ofstream.

   - **Hint:** You can make a file output stream (ofstream) truncate the file (i.e. erase its previous contents) by passing ios::trunc as a second parameter to the open function or to the initializing constructor.

   - **Note:** Don't print "Scores:" to the save file.

4. Add a constructor that takes the game name as a parameter and loads the score table for that game. Assume that the scores are in the same format as the save function saves them. If the file does not exist, the constructor should initialize the ScoreTable with an empty list.

   - **Reminder:** Each line consists of the score, followed by a tab, followed by the player name. There is no "Scores:" line in the file.

   - **Hint:** You can add the elements to the linked list by repeatedly calling the insert function.

   - **Note:** If the scores file does not exist, you program should not print an error message or terminate. A non-existent file is not a problem; it is just a case your program has to handle. You do <u>not</u> have to handle the case where the file exists but has bad data in it.

   - **Note:** The player names may start with digits and may have spaces in them. Your program should be able to load them anyway. One way is to load the score with formatted IO (<< notation) and then the rest of the line with getline. Then use str.substr(1) to remove the tab from the beginning of the line you read.

5. Test your ScoreTable module with the TestScoreTable5E.cpp program. You will also need the TestHelper module. The test program should give you full marks.

   - **Hint:** g++ ScoresTable.cpp TestHelper.cpp TestScoresTable5E.cpp

   - **Note:** This test program includes all the tests from TestElement5.cpp and TestScoreTable5.cpp.

6. Replace the `ScoreTable` member variable in the Game class with one that stores the game name. Choose an appropriate data type.

7. Change the `updateScoreTable` to update and display the score table as follows: First, load the existing score table from a file by declaring a local variable of the `ScoreTable` type and initializing it using a suitable constructor. Next, add the player name and current score to the score table. Then immediately save the updated scores to the same file. Finally, print the score table.

    - **Note:** We want there to be as little time as possible between when we read and write the score table. This is to reduce the chance that two programs both read the file, Program 1 updates it, Program 2 updates it based on the original file, and the changes made by Program 1 are lost. Although the approach here reduces the chance of a problem, it does not eliminate it. Completely reliable methods are covered in CS 375.

8. Test your program with the five test cases used in Part D: `testcase5A.txt`, `testcase5B.txt`, `testcase5C.txt`, `testcase5D.txt`, and `testcase5E.txt`. It should still work.

    - **Hint:** `g++ Location.cpp Node.cpp World.cpp Item.cpp ItemManager.cpp ScoreTable.cpp Game.cpp main.cpp`

Formatting [−10% if not done]

1. Neatly indent your program using a consistent indentation scheme.

2. Put spaces around your arithmetic operators:
   `x = x + 3;`

3. Use symbolic constants, such as `INACCESSIBLE`, when appropriate.

4. Include a comment at the top of `Main.cpp` that states your name and student number.

5. Format your program so that it is easily readable. Things that make a program hard to read include:

    - **Very many blank lines**. If more than half your lines are blank, you probably have too many. The correct use of blank lies is to separate logically distinct sections of your program.
    - **Multiple commands on the same line.** In general, don't do this. You can do it if it makes the program clearer than if the same commands were on separate lines.
    - **Uninformative variable names.** For a local variable that is only used for a few lines, it doesn't really matter. But a variable that is used over a larger area (including all global and member variables) should have a name that documents its purpose. Similarly, parameters should have self-documenting names because the function will be called from elsewhere in the program.
    - **No variable names in function prototypes**. Function parameters should have the same name in the prototype as in the implementation. This makes calling the function much less confusing.

Submission

- Submit a complete copy of your source code. You should have the following files with exactly these names:
    1. `Game.h`
    2. `Game.cpp`
    3. `Item.h`
    4. `Item.cpp`
    5. **`ItemManager.h`**
    6. **`ItemManager.cpp`**
    7. `Location.h`
    8. `Location.cpp`
    9. **`main.cpp`**
    10. `Node.h`
    11. `Node.cpp`
    12. **`ScoreTable.h`**
    13. **`ScoreTable.cpp`**
    14. `World.h`
    15. `World.cpp`
    - **Note:** A Visual Studio `.sln` file does NOT contain the source code; it is just a text file. You do not need to submit it. Make sure you submit the `.cpp` files and `.h` files.
    - **Note:** You do not need to submit the test programs or data files. The marker has those already.
- If possible, convert all your files to a single archive (`.zip` file) before handing them in
- Do NOT submit a compiled version
- Do NOT submit intermediate files, such as:
    - `*.o` files
    - `Debug` folder
    - `Release` folder
    - `ipch` folder
    - `*.ncb`, `*.sdf`, or `*.db` files
- Do NOT submit a screenshot